Eberhard Karls Universität Tübingen

Wilhelm-Schickard-Institut für Informatik
Technische Informatik

Implementation of PAFF-interlacing into an existing H.264 decoder for the Cell microprocessor

Vorgelegt von: Arno Fleck
Betreuer: Prof. Dr. Wilhelm Spruth

Diplomarbeit

Tübingen, November 2007

Executive Summary

This thesis presents an H.264 video decoder with Picture-adaptive Frame/Field (PAFF) interlacing support for the Cell Broadband Engine microprocessor. The existing H.264 decoder Aeolus, which was developed by a team at IBM China, was the basis for this work. It supports the real-time decoding of two progressive video streams in Full-HD resolution (1920x1080) on the Sony Playstation3 using only the Cell processor and no graphics acceleration features.

The Cell Broadband Engine (CBE) features a total of nine processors: one PowerPC Processing Element (PPE), which is fully compatible with other PowerPC processors, and eight Synergistic Processing Elements (SPE), which are optimized for floating-point operations. All these processors support SIMD instructions (Single Instruction Multiple Data) on 128 bit registers. Each SPE operates on its own 256 kilobytes local store. They access the main memory using DMA transfers between the main memory and the local stores so that it is possible to perform calculations while transferring data. The maximum performance of the CBE exceeds 200 gigaflops.

PAFF interlacing allows the encoder to choose between encoding a complementary field pair as one picture with both fields or as two separate pictures for each field. This flexibility increases the complexity of the decoder, because it has to handle references to frames or fields in the Inter-prediction process appropriately. In order to implement PAFF support into the Aeolus decoder, the following steps were required: Reading the corresponding syntax elements in the header data of the H.264 video stream, a new zig-zag scan pattern for the transform coefficients, the detection of the first slice of a new picture, a new and extended implementation of the Decoded Picture Buffer, and changes to the Inter-prediction and Deblocking processes.

As a result of this work, the Aeolus decoder can now decode video streams with PAFF interlacing in Full-HD resolution in real-time on a Cell blade.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Tübingen, den 9. November 2007

Acknowledgments

I would like to thank Prof. Wilhelm Spruth, of the University of Tübingen and Leipzig, as well as Friedemann Baitinger, Dr. Hanno Ulrich, and Stefan Wald at IBM Böblingen for giving me the opportunity to realize this thesis and for their support throughout the development process. Thank you to the Aeolus development team at IBM Beijing for their valuable assistance and cooperation, especially Huoding Li, Xing Liu, Xu Sheng, Rong Yan, and Yu Yuan. I also wish to thank my student colleagues in the Cell HDTV team at IBM Böblingen, especially René Becker, Thimo Emmerich, and Carsten Schmitt for creating such a pleasant team environment and their help with the Cell processor.

Special thanks go to Lida Sophia Townsley for proof-reading this thesis. Finally, I am especially grateful to my parents who have provided constant support and the finances to pursue my studies in Computer Science at the University of Tübingen and the California State University, Chico.

Contents

Chapter 1

Overview

1.1 Task formulation

The Cell HDTV project at the IBM research lab in Böblingen aims to create a software which allows any user to watch HDTV programs from a digital satellite receiver on the Sony Playstation3. This console features an IBM Cell Broadband Engine processor and runs a Linux operating system. The team in Böblingen designed a new hardware device which receives HDTV programs from a built-in satellite receiver and sends the video stream data over a gigabit Ethernet connection to a computer, handheld device, or Sony Playstation3 for decoding and display. The project uses the Network-Integrated Multimedia Middleware (NMM) to handle the network traffic and to ensure a correct playback of the video streams. NMM was developed at the Saarland University and is fully described in Marco Lohse's PhD thesis [9]. It is also planned to extend the Cell HDTV project so that it will be possible in the future to watch HDTV programs on devices like the Sony Playstation Portable (PSP) that are not powerful enough to decode and display such video data. In this case, the Playstation3 would transcode the video material into a format that is suitable for the little screen and limited processing power of the handheld device.

In Germany, most HDTV stations, including Sat.1, Pro7, and the Astra demo channel, broadcast their program encoded with the H.264 video codec, in 1080i resolution (1920x1080 interlaced). The interlaced video is encoded using the Picture-Adaptive Frame/Field (PAFF) interlacing method, which I will introduce in section 3.8.1.

In order to decode the video streams from these HDTV stations, an appropriate H.264 video decoder was needed. This decoder would need to support decoding the Main profile of H.264, including PAFF interlacing, in real-time on the Sony Playstation3. A team at IBM China has developed the Aeolus decoder which allows this, but it only supports progressive (non-interlaced) video material. The goal of this thesis was to identify the necessary changes for PAFF-support and to implement these changes into the Aeolus decoder.

1.2 Overview of this thesis

Chapter 2 gives an introduction to the techniques used in today's video compression methods, including the YCbCr color space, Inter- and Intra-coding methods, and the differences between progressive and interlaced video material. Chapter 3 introduces the H.264 video coding standard and outlines the features that distinguish it from previous video

codecs, including the two different methods, called PAFF and MBAFF, that are used to encode interlaced video material. Chapter 4 gives an overview of the particularities of the Cell Broadband Engine microprocessor, the Software Development Toolkit for it, and a description of the Linux environment on the Playstation3. Chapter 5 introduces the Aeolus H.264 decoder, which was developed by a team at IBM China and was the base for the implementation of this thesis. Chapter 6 describes the changes that are necessary in order to add PAFF interlacing support to an existing H.264 decoder and describes the actual implementation of these changes into the Aeolus decoder. Finally, chapter 7 discusses the results of this thesis and the future work.

Chapter 2

Introduction to video compression and HDTV

2.1 Introduction to video compression

Nowadays, digital video is becoming more and more common, particularly in areas such as digital video discs (DVD, Blu-ray), digital video broadcasting over satellite (DVB-S), antenna (DVB-T) or cable (DVB-C), or videos on websites. All these digital videos are compressed with a video codec, such as MPEG-2 or H.264.

Were the video to be transmitted without compression, one minute of video in PAL resolution (720x576) would need about 1.7 gigabytes of data and one minute of video in Full-HD resolution (1920x1080) with 25 frames per second (fps) would require 8.7 gigabytes. With these massive amounts of data, a common Dual Layer DVD (DVD-9) with 8 gigabytes of capacity could not fit even five minutes of video material in standard PAL resolution. Thus, it is apparent that storing or transmitting video uncompressed is not feasible.

Modern video codecs, such as MPEG-2, MPEG-4, DivX, VC-1, or H.264, use a number of techniques to reduce the amount of data that is required for storing or transmitting the video with very few errors (*compression artifacts*) visible to the human eye. The following sections will give an overview of these techniques. A more detailed description of these techniques can be found in [2], [13] (chapters 2 and 3), and [18].

2.2 RGB and YCbCr color space

Usually, digital images (and videos) are stored and processed in the RGB color space. This means that every pixel of the image consists of three different values: one for the red component, one for the green component, and one for the blue component. It is common to use eight bits for every component, so one pixel requires 24 bits of data. The RGB color space is also used for displaying an image. In computer monitors, TVs or projectors, the three color components are displayed separately. From regular viewing distance, the three color components merge for the human eye and the viewer sees the original color.

It is possible to display any visible color with the RGB color model. However, it does not take the particularities of the human visual system into account. The human eye is more sensitive to brightness than to color. In the RGB color model, the brightness affects all three components. In order to store image data with less bits per pixel, but

without visible flaws, a color space in which brightness (luminance or *luma*) and color information (chrominance or *chroma*) are handled separately is needed.

The most commonly used color space with this property is YCbCr. Here, the Y component contains the luma and Cb and Cr together contain the chroma. The value of the Cb channel describes the difference between Y and the blue component of the color, while Yr describes the difference between Y and the red component of the color. A conversion between the RGB and YCbCr color space is lossless, which means that this transformation does not lead to a loss of quality. Historically, YCbCr is derived from the YUV color model used in the analog PAL video standard, and is similar, although not identical, to it. Often, the YCbCr color space is also referred to as YUV, which is inaccurate, because YUV describes an analog model and uses a different definition for the color channels.

In order to reduce the amount of data required to store one pixel, reducing the number of bits for the chroma part while storing the full luma part is practicable. The sample format that does not reduce the data volume and stores as many chroma as luma components, is called 4:4:4 sampling. Reducing the amount of chroma samples in the horizontal direction by a factor of two is called 4:2:2 sampling (or YUY2). Finally, reducing the amount of chroma samples in both the horizontal and vertical direction by a factor of two is called 4:2:0 sampling (or YV12) and is illustrated in figure 2.1. In this case, the amount of chroma samples in a picture is only one quarter of the amount of luma samples, so only half of the data of 4:4:4 sampling is required. So when eight bits per channel are used, one pixel only requires twelve bits of data on average. This is the most commonly used method in video compression today and the loss of information resulting from this method is virtually invisible to the human eye.

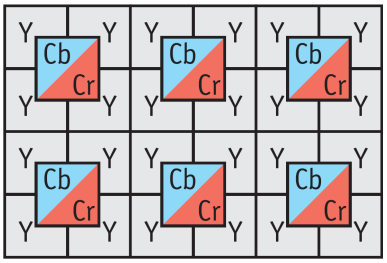


Figure 2.1: 4:2:0 sampling [2]

2.3 Intra coding methods

In addition to the reduction of bits per pixel by using 4:2:0 sampling, further methods are applied to only store information that is visible to the human eye. For the perceived quality of an image, it is more important to keep the low frequencies of color or brightness changes intact (i.e. gradual changes of brightness or color) than keeping high frequencies of color or brightness changes (i.e. quick changes in a small area). If the compression methods neglects the high frequencies of an image, the image will only look slightly smoother to a viewer.

The method that is commonly used to separate the different frequencies of an image is the Discrete Cosine Transformation (DCT), which has similar effects as the Fourier transformation and is lossless. DCT is part of the JPEG and MPEG standards including the MPEG-4 derivatives like DivX or VC-1.

In order to perform the DCT, one first needs to divide the color components of an image into blocks with 8x8 sample values, which are called *macroblocks*. Once this step is done, the DCT can be applied to these macroblocks, resulting in an 8x8 matrix that contains the values of the different frequencies of the image. In the upper left corner is the average brightness of the block (referred to as the *DC-coefficient*) and toward the lower right corner, the values represent the higher frequencies of the macroblock (called *AC-coefficients*). Figure 2.2 shows an example of a DCT. One can see the original brightness values on the left and the corresponding DCT coefficients on the right.

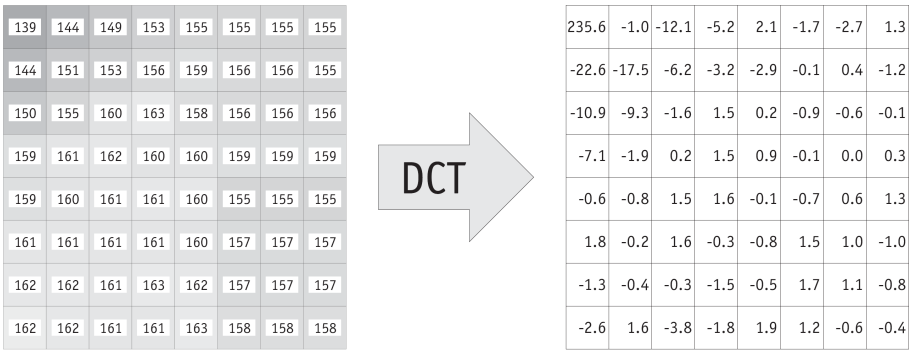


Figure 2.2: Discrete cosine transformation of a macroblock [2]

In the next step, the values in the 8x8 matrix are quantized. For this, the coefficients are divided by a certain predetermined factor, which becomes greater with higher frequencies. Higher values in the quantization matrix lead to a higher quality loss, but to smaller values and more zeros among the values representing higher frequencies. Thus, the result can be compressed more efficiently using entropy encoding. Figure 2.3 shows the effect of a sample quantization process. One can see that most of the coefficients are now zero.

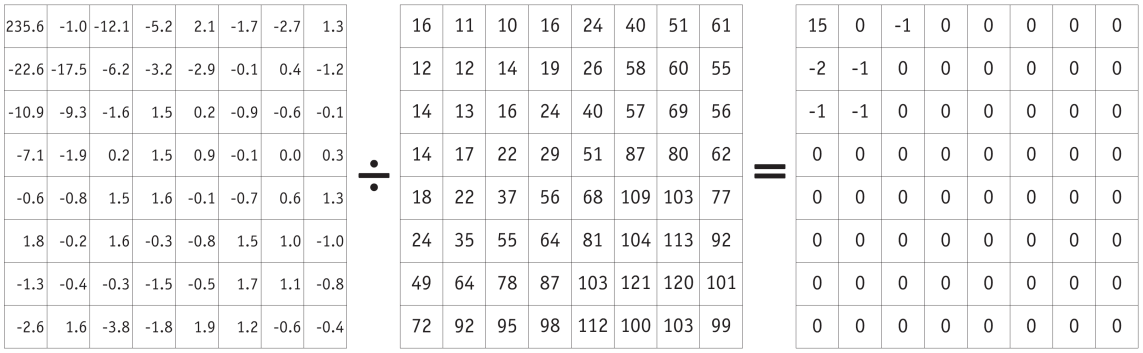


Figure 2.3: Quantization of the DCT coefficients [2]

Finally, the encoder stores the DC-coefficient and scans the AC-coefficients, starting in the top left corner and then following the zig-zag pattern that is shown in figure 2.4. For this example, the resulting number sequence is 0, -2, -1, -1, -1, 0, 0, -1, 0, 0, 0, ... Since all the values but for the first eight are zero, a decoder would be able to reconstruct the full sequence from only these eight values and a marker that represents the information that the rest of the block is zero. In addition to this, the encoder will apply entropy encoding to the AC coefficients with a method like Huffman, which once again significantly reduces the number of bits that are needed to store the macroblock.

Since all these coding methods only work on a single picture of a video, without references to other pictures, they are referred to as “Intra-coding” methods. The technique

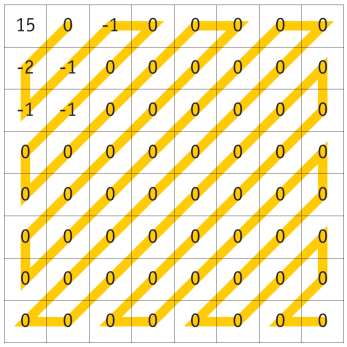


Figure 2.4: Pattern in which the DCT coefficients are stored after quantization [2]

described in this section is also part of the JPEG standard, which only defines the compression of a single picture.

2.4 Inter coding methods

Every video has high temporal redundancy, meaning that large areas in any given frame often do not change compared to the previous frame. To take advantage of this, the first international video coding standard H.120 from 1984 used a method that is called *conditional replenishment* and offers two modes for any macroblock: *Skip* means to copy the macroblock from the previous frame and *Intra* means to store the macroblock in Intra-mode. Later standards introduced a third mode (*Prediction*) in which the macroblock is also copied from the previous picture, but then enhanced with a refinement difference approximation. In this case, the encoder also stores the difference between the approximation and the input macroblock. The reasoning behind this method is that often the corresponding block in the previous frame gives a decent approximation for the block in the current picture. Because of this, storing the difference between the two blocks needs less data than storing the whole block in Intra-mode.

Although these methods already reduce the resulting file size significantly compared to a video that is only encoded in Intra-mode, they are not effective for objects that are moving through the picture or general camera movements. These movements lead to large differences in the values of samples in a picture, especially along the border of the moving object. A technique designed to efficiently encode these situations is *Motion-Compensated Prediction (MCP)* and was first introduced in the early 1970s. [18] gives a comprehensive overview of MCP, which will be summarized in the following paragraphs.

When using MCP, the encoder first analyzes the picture in a process called *Motion Estimation (ME)* and searches for *Motion Vectors (MVs)* that point to the places in the previous picture where the decoder can find a block containing a good approximation for the macroblock in the current picture. Finally, the encoder stores the difference between the actual samples of the macroblock in the current picture and the estimation taken from the previous picture. The data containing this difference is known as *MCP residual coding*.

There are several possible MCP methods. The following ones are used in the H.264/AVC standard, which will be described in chapter 3:

Fractional-sample-accurate MCP The elements of the motion vectors have more than integer precision, which requires the use of interpolation between the actual sample

values. The idea behind this method is to allow a more accurate motion representation and greater flexibility, which leads to a prediction that is more accurate than one that only uses integer values. Even for H.261 (from 1990), half-sample accuracy was considered, but not included due to the complexity limits of the time. MPEG-1 (from 1991) was the first international standard that included half-sample accurate MCP.

Bi-predictive MCP With this method, not only the previous frame can be used as a reference frame, but also the following frame. This means that the decoding order of frames is different than the input/output order, because the reference frames have to be decoded before the frame that has references to them. This method is especially helpful in cases when the scene contains smooth and consistent motion. The first standard that included this method was also MPEG-1 (from 1991).

MVs over picture boundaries This method allows the motion vectors to point outside the boundaries of the reference frame. When an objects moves into the picture, the samples that were outside the reference frame are extrapolated, usually by simply replicating the boundary samples. This method was first introduced in the H.263 standard (from 1995).

Variable block size MCP This describes the ability of the encoder to select the size of the region that is associated with the different MVs. This way, it is possible to choose between the accuracy of small blocks and the small amounts of data that are necessary for describing bigger blocks.

Multi-picture MCP With this method, the encoder can use more than one previous or succeeding frame for reference. This is useful to better make use of parts of the image that do not change over a longer period of time, like backgrounds.

Multi-hypothesis and weighted MCP This allows the encoder to use a combination of several MCP signals, possibly with different methods. It is also possible to blend the results together, for example to allow efficient coding of crossfades between two scenes. Fractional-sample-accurate MCP can be seen as a special case of multi-hypothesis and weighted MCP: For this, several integer-accurate MCP predictions are blended together.

2.5 Typical operation of a modern hybrid video encoder

All modern video coding standards since H.261 define *hybrid* codecs. This term refers to the design of a combination of the two different techniques outlined in the two previous sections, namely motion prediction and spatial frequency transform. In order to ensure accurate results of the video decoding, the encoder simulates the results of the decoder and uses this data as the base for the motion prediction calculations. If this was not done, the decoder would use different material for the motion prediction and thus the MCP residual coding would not fit the actual data that the decoder uses. Especially after a group of frames that were encoded in Inter-mode, the difference would become very significant because of error propagation.

In modern video encoding standards, only the decoder is encompassed by the standard. This means that only the interoperability is ensured, but the standard contains no

assurances of end-to-end reproduction quality. This leaves a great deal of flexibility for the developers of encoder implementations. They can balance compression quality and bitrate of the encoded video as well as implementation costs or time-to-market. One of the most important parts of the encoder design is the optimization of the many different encoding decisions that the Inter- and Intra-prediction methods offer. In any case, the encoder needs to balance between bitrate and distortion of the video signal, given the constraints on delay (difference to real-time in live broadcasts) and complexity (encoding time).

The typical operation of a modern hybrid video encoder is as follows:

1. The encoder splits each frame into macroblocks.

2. It encodes the first picture, every frame at the beginning of a scene change, and other frames in periodic intervals in Intra-mode. These periodic Intra-frames are called *keyframes*. All the other frames in the video are encoded in Inter-mode, using MCP.

3. The encoder transforms the residual of the Inter- or Intra-prediction with a frequency transformation, such as the DCT.

4. It scales, quantizes and zig-zag scans the transform coefficients. Then it applies entropy encoding to the result with a method like Huffman.

5. Finally, the result will be stored or transmitted together with prediction side information, containing which prediction steps were used so that the decoder can reproduce these.

2.6 Progressive and interlaced video signals

A video signal can either be transmitted as a series of full frames, which is called *progressive* encoding, or as a series of fields, which is called *interlaced* encoding. A field either contains all the odd lines of a picture (“top field”) or all the even lines (“bottom field”). Figure 2.5 illustrates the alternating order of fields of an interlaced video sequence. The advantage of interlaced coding in comparison to progressive coding is that it is possible to transmit twice the number of fields than frames per second with the same data rate, which gives the viewer the appearance of a smoother motion. This is especially true for analog video transmission where the frequency range to transmit 50 fields per second is exactly the same as for transmitting 25 frames per second. All the common analog TV standards (PAL, SECAM, NTSC) use interlaced video transmission, so that the digital TV broadcast for these TV standards also transmits the video as interlaced fields.

The biggest disadvantage of interlaced video today is that the modern LCD displays, as well as computer monitors, only work in progressive mode. This makes it necessary for the decoder software to *deinterlace* the video before it sends the raw video data to the output device. Simple forms of deinterlacing result in cutting the number of frames per second in half, which eliminates the advantages of interlaced video transmission. More advanced forms of deinterlacing, which keep the full frame rate, require significantly more processing time and thus more powerful hardware.

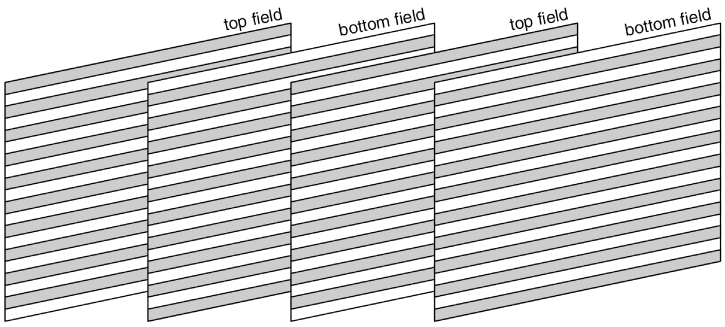


Figure 2.5: Fields in an interlaced video sequence [13]

2.7 High-definition Television (HDTV)

The common TV standard in Germany is PAL, which is an analog standard. When broadcasted digitally, PAL is transmitted with a resolution of 720x576 pixels. This resolution is also used for PAL-DVDs. In some occasions, a resolution of 768x576 pixels is used to represent PAL, because in this resolution square pixels represent the correct aspect ratio.

The resolution of the HDTV standards that are commonly used today are 1280x720 and 1920x1080 pixels. The latter is also referred to as “Full-HD”. Both HDTV-resolutions can either be transmitted in progressive mode with 25 or 30 fps or, in interlaced mode, with 50 or 60 fields per second. Progressive video in 1920x1080 resolution is commonly referred to as “1080p”, while interlaced video is referred to as “1080i”. Figure 2.6 illustrates the differences in resolutions in the different TV standards from NTSC to Full-HD and for reference also the XGA resolution of computer equipment. Figure 2.7 shows a comparison between PAL and 1080p resolution. I have taken the image from a movie trailer in 1080p and created the PAL image by downsizing the frame accordingly.

In the last years, commercial HDTV broadcasting has started in Europe. The three most famous TV stations that are currently broadcasting some of their material in HD resolution over the Astra satellites [1] are the free-TV stations Pro7 and Sat.1 [12], and the pay-TV station Premiere. All these TV stations broadcast in Full-HD resolution and interlaced mode (1080i). The material is compressed with the H.264 codec in PAFF mode, which I will describe in the following chapter.

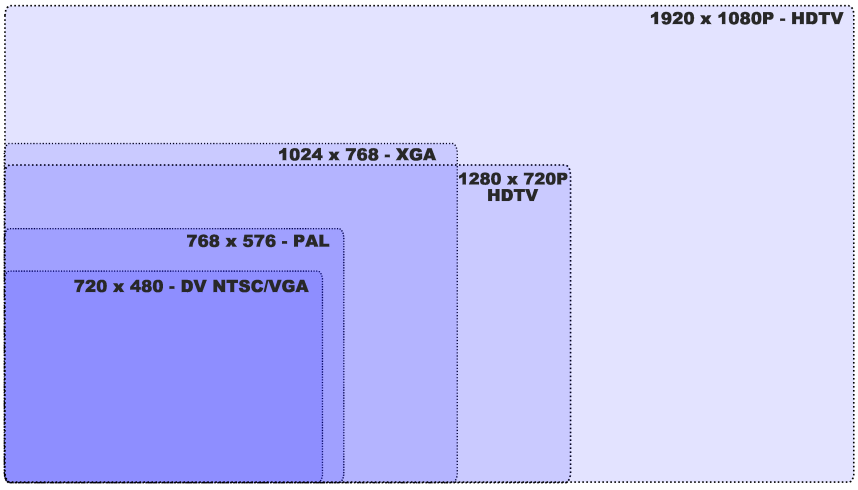


Figure 2.6: Resolutions of different video standards [20]

(a) PAL resolution

(b) 1080p resolution

Figure 2.7: Comparison between PAL and Full-HD (1080p) quality

Chapter 3

Overview of the H.264 / AVC video coding standard

3.1 Introduction

The H.264 / Advanced Video Coding (AVC) standard, also referred to as MPEG-4 Part 10, is the newest international video coding standard and can be used for a wide range of applications. The high coding efficiency and thus the small file sizes of the encoded video make it an attractive option for transmissions over channels with small bandwidths, like video broadcasting to handheld devices or video conferences over an ADSL Internet connection. The video on the Universal Media Disc (UMD) is also encoded in H.264. This disc, which holds 1.8 gigabytes of data, is used to store digital video for portable devices, especially the Sony Playstation Portable (PSP). H.264 is also commonly used for storing and broadcasting HDTV video material, like satellite broadcasts over the Astra satellites and the new HD video disc formats Blu-ray and HD DVD.

The introductory papers [19] and [18], the article [15], as well as the book [17] give an overview of the H.264 standard, while the book [13] provides a more detailed view. Finally, the ITU-T H.264 standard recommendation [7] contains all the details of the video coding standard.

Because of the complexity of H.264, the features are grouped into the profiles Baseline, Main, Extended, as well as four different High profiles. For the goal of the Cell HDTV project, of which this thesis is a part, it is sufficient to support the features of the Main-profile. Apart from the definition of profiles, the standard also contains the definition of levels. These define the maximum bitrate for the video stream.

The following sections provide an overview of the features of the Main profile, including all the features that are necessary for full interlacing support.

3.2 Video Coding Layer (VCL) and Network Abstraction Layer (NAL)

H.264 defines a Video Coding Layer (VCL) and a Network Abstraction Layer (NAL). The VCL efficiently represents the encoded video data, while the NAL defines a syntax to format the VCL representation and provides header information for transferring the data over a network. Choosing an appropriate format for network transmission is important because video streams are often sent over error prone channels, such as satellite connections.

Intra- and especially Inter-coding can lead to error propagation over several frames. Usually, the decoder cannot fully recover from transmission errors before the next Intra-mode frame. Also, depending on where the transmission error happens, the affected area in the decoded picture can vary. If the transmission error happens in the residual data of a macroblock, only one macroblock and possibly a low number of macroblocks in later pictures are affected. However, if MCP data or even header data that defines the image format or resolution of the video gets distorted, a large number of frames can be fully affected or the decoder might not be able to decode the picture at all. So it is useful to separate header information from image data, so that special measures can be taken in order to make sure that these parameters will be correctly transmitted to the decoder. A possible option to ensure this is *Forward Error Correction*, which adds redundant information to the data that is to be transmitted, so that the decoder can reconstruct the information if parts are lost.

The H.264 standard facilitates this separation of different data types by organizing the video stream into NAL units, which are packets of data that contain a certain number of bytes. In addition to VCL NAL units, there are also non-VCL NAL units. These may contain parameter sets (see below) or supplemental information, that may enhance the picture quality but which are not necessary to correctly decode the video stream. The set of NAL units that are associated with a single picture is called *access unit*.

Parameter sets contain important header information that can apply to a large number of VCL NAL units. There are two types of parameter sets: *Sequence Parameter Sets (SPS)* apply to a series of consecutive pictures and *Picture Parameter Sets (PPS)* apply to one or more individual pictures. The VCL NAL units point to their respective parameter set. It is possible to transmit parameter sets independently from the VCL NAL units, either clearly ahead of the VCL NAL units, to allow for repeated transmission if necessary, or even in a different transmission channel.

3.3 Video coding in H.264

A coded video sequence in H.264 consists of *coded pictures*. These pictures can either represent a field or frame. The color samples are stored in YCbCr color space with 4:2:0 sampling, with 8 bits per luma or chroma sample, resulting in an average 12 bits per pixel.

Macroblocks consist of 16x16 samples for luma and 8x8 samples each for the Cb and Cr channels.

Every picture in a video sequence consists of one or more *slices*. A slice is a self-contained group of macroblocks. It does not have references to other slices within the same picture. This makes it possible to decode all the slices of a picture simultaneously. The only exception to this rule is the in-loop deblocking filter (see section 3.7), which is the last stage of the video decoding for every frame. For applying this filter, it may be necessary to access samples from the bordering slices. The macroblocks of a slice are taken from the picture in the order of a raster scan, from top left to bottom right. Figure 3.1 shows an example for a possible partitioning of a picture into three slices. It is possible to choose the size of the slices freely, but the shape is predefined by the order of a raster scan. Only the Baseline and Extended profiles allow one to choose slices in arbitrary shapes, which is called “Arbitrary Slice Order”.

Each slice is of one of the following types:

I-slice The slice is encoded in Intra-mode, meaning that it does not contain references to

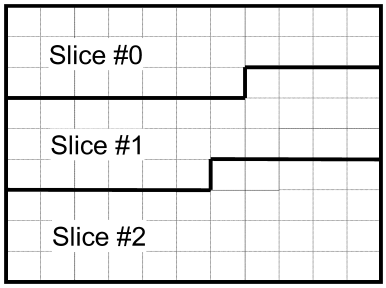


Figure 3.1: A possible partitioning of a picture into three slices [19]

other slices.

P-slice In addition to Intra-coding, the encoder can use Inter-prediction for the macroblocks with *one* motion-compensated prediction signal per prediction block. The reference for the prediction can either be one of the previous or one of the successive frames.

B-slice This slice type is similar to P slices, with the difference that the encoder can use *two* prediction signals per prediction block. These signals are then blended together, which can be especially useful when encoding crossfades between two pictures.

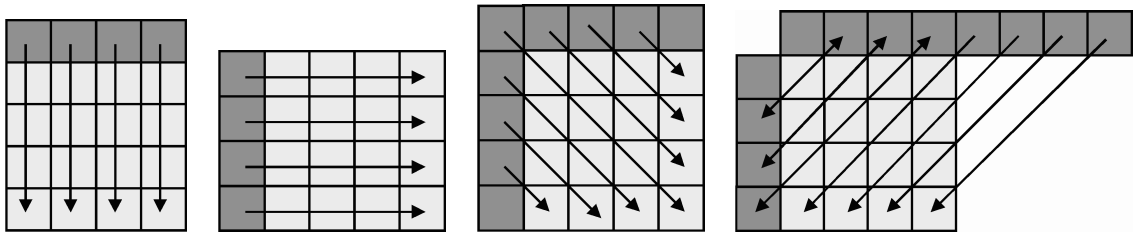
(SI- / SP-slice) These slice types are only defined in the Extended profile and are called switching slices because they are valid for two different streams. This means that a decoder can switch seamlessly from one stream to another when both of these streams use switching slices and were encoded together for this purpose.

3.4 Intra-picture prediction

In addition to the methods of Intra-picture coding that I have introduced in section 2.3, H.264 uses several Intra-picture prediction modes. The motivation behind this technique is that a macroblock is likely to be similar to its neighboring macroblocks, for example in a blue sky, within a patch of grass, or as a part of any other pattern. The residual coding of this prediction is likely to be smaller than the color samples themselves.

The size of the predicted area for the luma component can either be 16x16 or 4x4 samples. The encoder can choose between these two sizes depending on the smoothness of the predicted area. For chroma, the full 8x8 macroblock always serves as the predicted area. Intra-prediction always refers to previously decoded samples within the same slice, to the top and left of the current block. The Intra-prediction offers several possibilities to combine the samples from these blocks and create a prediction that closely matches the actual block. Figure 3.2 shows four examples of these Intra-prediction modes: In the vertical mode, the top samples are copied into the whole 4x4 block, while in the horizontal mode, the samples are taken from the left. Figure 3.2(d) shows an example for a prediction mode where several samples are combined using a weighted average function. There is a total of nine Intra-prediction modes that differ in the direction in which the samples are copied. The book [13] gives a full overview over the available modes.

In addition to these prediction modes, there is also a third mode called “PCM”. In this mode, the encoder uses neither Intra-prediction nor transform and quantization for the macroblock, which means that all the sample values are written to the output stream



(a) Vertical (b) Horizontal (c) Down / Right (d) Down / Left

Figure 3.2: Four different Intra coding modes [19]

without any changes. This is especially useful when it would be necessary to store more bits for the encoded video data than storing the samples without any compression. Such a situation, however, is quite unusual.

3.5 Inter-picture prediction

The motion compensation method that the H.264 standard defines gives the encoder greater flexibility than any other standard that was previously defined. In this section, I will give an overview of the Inter-picture prediction methods that the H.264 video coding standard includes.

3.5.1 Macroblock partitioning

Instead of using full 16x16 macroblocks as source regions for motion estimation, the H.264 standard allows the encoder to reference *macroblock partitions* that can be as small as 4x4 pixels. Figure 3.3 shows the valid partitions that the standard defines. This partitioning, which is a new feature of H.264 in comparison to older standards, enables the encoder to choose between the accuracy of motion compensation with small blocks and the small overall data size for encoding motion vectors of bigger blocks. For example, for a moving object, the borders of the object would be encoded with small blocks to reduce the amount of bits necessary to encode the residual. For the body of the object, however, large block sizes are more appropriate because every part of the object will move in the same direction, so motion compensation with a large block size will already provide a good prediction without the necessity to encode the additional information of many small blocks. The decision of block sizes thus greatly influences the encoding performance.

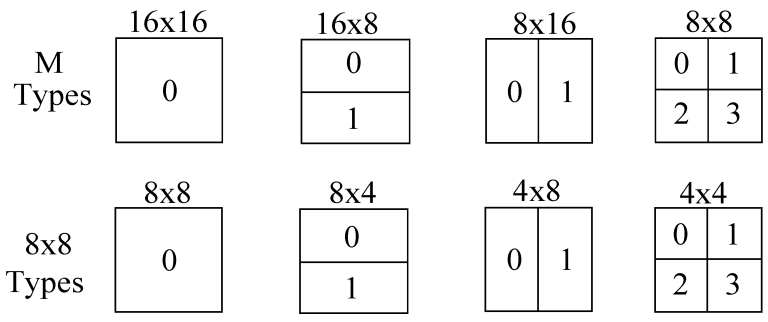


Figure 3.3: Possible macroblock partitions for Inter-picture prediction [19]

3.5.2 Motion vectors

Especially when small partitions are used, the encoding of the motion vectors can require a significant number of bits. However, a group of motion vectors in the same area of the picture usually shows a high correlation. For this reason, the H.264 standard includes a method called *Motion Vector Prediction (MVP)*. Instead of storing the full values of a motion vector, the encoder and decoder calculate the median of the motion vectors of the prediction blocks to the left and the top of the current block. The encoder will only store the difference between this predicted vector and the actual motion vector.

The resolution of the motion vectors is quarter-sample for the luma component and, because there are twice as many luma as chroma samples in each direction, eighth-sample for the chroma component. The syntax also allows motion vectors that point outside the picture boundary. In this case, samples outside the picture area are extrapolated by repeating the border samples.

The standard also allows more than one reference picture for the motion-compensated prediction. This makes it necessary to transmit a value Δ together with the respective motion vector which describes a certain reference picture in the decoded picture buffer. Figure 3.4 shows an example for this method, which is called *Multi-picture motion compensation*.

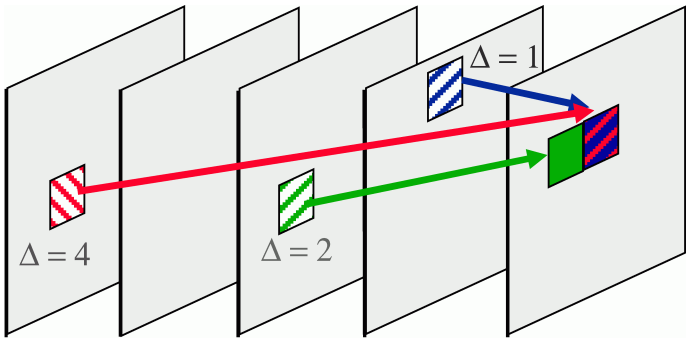


Figure 3.4: Multi-picture motion compensation in H.264 [19]

P-slices allow the usage of *one* prediction signal per prediction block. The signal points to any of the previously decoded pictures in the decoded picture buffer, no matter whether the position of these pictures in the output order is before or after the current picture. The green motion vector in figure 3.4 is an example for a possible prediction signal in a P slice.

B-slices also allow the combination of *two* prediction signals per prediction block. The two signals are not required to point to the same reference frame. The blue and red motion vectors in figure 3.4 show an example for this form of motion-compensated prediction. The two prediction signals can be combined with a weighted average. This new feature of H.264 is called *Weighted Prediction* and especially provides a significant improvement over previous video encoding standards for situations where once scene fades to another.

Finally, a macroblock can also be stored in the *P-Skip* or *B-Skip* mode. In this case, the encoder does not store any information relating to Inter-prediction. The decoder will generate the motion vector from those that were previously decoded. This method is similar to the method used for Motion Vector Prediction. The result of the interpolation will then be used to decode the macroblock. This allows the encoder to represent large areas with uniform or no motion in the picture with very few bits.

3.5.3 Decoded Picture Buffer (DPB)

The multi-picture motion-compensated prediction in H.264 requires the definition of a Decoded Picture Buffer (DPB) which contains a certain number of previously decoded pictures. It consists of two reference picture lists with the numbers 0 and 1. List 0 is both used for P-slices and the first prediction signal in B-slices, while list 1 is only used for the second prediction signal in B-slices. Both lists can contain pictures before and after the current picture in the display order. After decoding a picture, the default behavior of the decoder is to add the picture as a *short-term reference picture* to the DPB. It will handle pictures of this type according to the sliding-window memory control method: When the DPB already contains the maximum amount of reference pictures, the decoder will remove the oldest short-term reference picture from it and then add the new picture. These pictures in the DPB are ordered according to their *PicNum*, which is a value that is directly derived from the frame number. New pictures are added to the position 0 of the reference picture list and the indices of the other pictures are then incremented by 1. As a result of this method, the short-term reference pictures in the DPB are ordered by their *PicNum*, from the highest to the lowest value.

If a certain picture is not used for reference in the other pictures of the video sequence, the encoder can mark it as *unused for reference*. In this case, the picture will only be kept in the DPB as long as necessary to output it at the correct point in the output sequence. Pictures of this type do not count toward the maximum amount of reference pictures in the reference picture list.

H.264 also defines another type of reference pictures: The encoder can mark certain pictures in the video sequence as *long-term reference pictures*. The decoder addresses these pictures with the *LongTermPicNum* value and orders them from the lowest to the highest number. It will not remove the long-term reference pictures until it encounters a command in the input video stream to either remove the picture from the DPB or replace it with another one. These commands in the video stream are called *adaptive memory control commands*. With these, the encoder can signal the decoder to assign a long-term index to a short-term reference picture, to change the picture ordering, or to mark either long or short-term reference pictures as “unused for reference” and thus removing them from the reference picture list.

Finally, pictures that only consist of I- or SI-slices can be marked with the Instantaneous Decoder Refresh (IDR) flag. When a decoder receives an IDR picture, it will mark *all* the pictures in the DPB as “unused for reference”. Since these pictures cannot be used for Intra-prediction in the following pictures anymore, the images in the video sequence after the IDR picture will not contain references to pictures prior to the IDR picture.

3.6 Transform, scaling, quantization, entropy encoding

Previous video coding standards, such as MPEG-2, used the discrete cosine transform (DCT) method on 8x8 blocks to transform the prediction residual. H.264 uses an integer transformation on 4x4 blocks, which [10] describes in detail. This transformation method has similar properties as a 4x4 DCT, but with lower complexity. For implementing an inverse DCT transform, the developers can choose between an accurate but very slow inverse transformation using floating-point arithmetic or a fast but inaccurate inverse transformation using integer arithmetic. Especially in standards such as H.264 that heavily rely on predictions, the mismatch between the data of the encoder and decoder (referred to as

drift) that is a result of an inaccurate inverse transformation can lead to visible artifacts. The new integer transformation used in H.264, however, allows the implementation of an accurate inverse transformation using only additions and bit-shifting operations on 16-bit integer values, allowing a low-complexity decoding process that does not cause drift.

For quantization, H.264 uses a quantization parameter that can take 52 values, designed so that an increase of 1 of the parameter leads to an increase of quantization step size of 12%.

For the entropy coding, the H.264 standard allows the choice of either Context-Adaptive Variable Length Coding (CAVLC) or Context-Based Adaptive Binary Arithmetic Coding (CABAC), which are both described in [13]. Detailed information about CABAC was also published in [11]. CAVLC is an entropy coding method similar to Huffman. Several VLC tables for various syntax elements are used to adapt to the respective characteristics of the syntax element. For example, for coding the residual data, the number of non-zero quantized coefficients and the actual size and position of the coefficients are coded separately. CABAC, which is not part of the Baseline profile, is another entropy coding method which, according to [19], typically reduces the bitrate between 5 and 15% compared to data encoded with CAVLC, at the cost of a higher computation complexity.

3.7 In-Loop Deblocking Filter

Because every picture of a video is divided into macroblocks in today's video coding standards, lines at the border of the macroblocks can become visible in certain situations. They are often conspicuous in smooth areas of the picture when the video was compressed with a low target bitrate. These lines are referred to as *block artifacts*. In former video coding standards, such as MPEG-4, a deblocking filter was often applied in post-processing, meaning *after* the decoding process and immediately before displaying the picture in the video player. This filter smoothes the borders of the macroblocks, which clearly improves the subjective quality of the video.

Figure 3.5: An example for the performance of the In-Loop Deblocking Filter [14]

However, since deblocking is applied after the decoding process in these video coding standards, the encoder and decoder use reference pictures that were not processed by the deblocking filter and thus can not benefit from the improvement in video quality that is

caused by this post-processing step. Also, the filter can not use information from the decoder that could be valuable in determining which parts of the pictures should be left untouched by the deblocking filter. This could be the case when the original picture contains sharp borders which should not be smoothed. There is also the case where the filter strength should be specifically high in areas that are smooth in the original picture. For this reason, the deblocking filter was included into the decoding process of H.264 and is called *In-Loop Deblocking Filter*. Figure 3.5 shows an example for the performance of this filter. The left image shows the picture immediately after the decoding. The right image shows the picture after the in-loop deblocking filter was applied. According to [19], the required bitrate to achieve the same Signal-to-Noise ratio (SNR) when using this filter is 5-10% lower, compared to the video coding with the in-loop deblocking filter deactivated. The improvement in subjective quality is even higher.

3.8 Interlacing and adaptive frame/field coding

As described in section 2.6, digital TV broadcasts are transmitted in interlaced mode. This makes it necessary to implement support for interlaced video into the video coding standards. Without this support, an efficient video coding would not be possible because interlaced video introduces new challenges for the video encoder. I will present the approaches that were included into the H.264 standard to face these challenges.

3.8.1 Picture-Adaptive Frame/Field coding (PAFF)

For processing interlaced video signals, the video encoder can choose between encoding the two complementary fields as a single frame or encoding the two individual fields separately. Both approaches have their drawbacks in certain cases: When there is significant movement in the respective picture, encoding both fields together as a frame would be highly inefficient because many artificial borders are introduced. These borders are called *comb artifacts*. Sharp borders always require a significant increase in bitrate to conserve the details. Encoding both fields separately as two individual pictures avoids this problem. However, when there is no or very little movement in a certain picture, it is more efficient to encode both fields as a frame. This is because more statistical dependencies within a picture can be exploited. Also, because a field can only contain half of the original picture, horizontal borders tend to be artificially sharp, which would lead to an increase in bitrate.

H.264 addresses this issue with the introduction of *Picture-Adaptive Frame/Field coding (PAFF)*. In this mode, the encoder can choose for every single picture whether it will be encoded in frame mode (both fields together in a frame), or in field mode (both fields separately). According to [19], using the PAFF method leads to a 16-20% increase of encoder performance in comparison to encoding every picture in field mode. Figure 3.7 shows two sample pictures. For the picture on the left, the frame mode would lead to a more efficient coding, because comb artifacts are practically invisible. The picture on the right is an example where the field mode would be more efficient: It is an example of horizontal camera movement, which causes comb artifacts to be visible throughout the picture.

3.8.2 Macroblock-Adaptive Frame/Field coding (MBAFF)

There are still cases where neither frame nor field coding will allow for an efficient encoding of the respective picture. Figure 3.8 shows an example for such a case: The background is static, so it would be best encoded in frame mode. However, the person in the foreground is moving, so for this part it would be more efficient to use the field mode.

H.264 offers a mode that specifically addresses this issue. *Macroblock-Adaptive Frame/Field coding (MBAFF)* allows the encoder to make the choice between frame and field coding per vertical adjacent pair of macroblocks. In frame mode, the two macroblocks are encoding normally. In field mode, the top macroblock of the macroblock pair contains the odd lines of the macroblock pair and the bottom macroblock contains the even lines of the macroblock pair. Figure 3.6 illustrates this method, which would be used to encode the head of the person in the sample picture (figure 3.8), while the background would be encoded in frame mode.

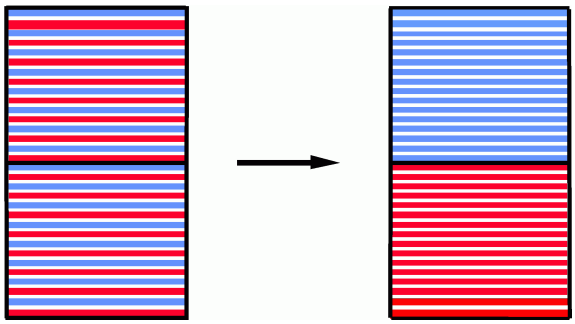


Figure 3.6: Illustration of the field mode of MBAFF encoding

According to [19], for scenes with a mixture of static and dynamic parts, using MBAFF can improve encoding performance by 14-16% in comparison to encoding the scenes with PAFF coding.

Figure 3.7: Sample pictures for efficient frame and field coding

Figure 3.8: Sample picture that cannot be encoded efficiently in either frame or field mode

Chapter 4

Cell Broadband Engine

The Cell Broadband Engine is a newly designed multiprocessor that was developed by Sony, Toshiba, and IBM and presented in 2005. It is most widely used today in Sony's Playstation3 gaming console, but there are also server blades available that feature two Cell processors as the their CPUs. In this chapter, I will introduce the Cell processor and describe the main design choices that were made in its development.

For further information, I recommend the following references: The course slides [6] give an overview of both the Cell processor itself and the development of applications for it. The article [8] focuses on the hardware, while the books [4] and [3] focus more on the software development. In addition, the article [16] in the c't magazine provides a general introduction to the new processor.

4.1 Overview

The Cell Broadband Engine (CBE) multiprocessor consists of a total of nine processors on a single chip that operate on a shared memory. All of these processors support Single-Instruction / Multiple Data (SIMD) commands with 16 byte registers. One of these processors is the the *PowerPC Processor Element (PPE)*, which is optimized for running the operating system and control tasks. Although it does not include all the features of a regular PowerPC processor, it is fully capable of running existing PowerPC applications, albeit with lower performance. However, the other eight processors of the CBE, the *Synergistic Processor Elements (SPEs)*, are optimized for compute-intensive applications. In a Cell-optimized application, the PPE will usually only run the operating system and the top-level control thread, while the SPEs process the computations of the application. In this case, the application will be executed at a much higher speed than what is achievable on conventional multiprocessors.

4.2 PowerPC Processor Element and Synergistic Processor Element

The PPE accesses the main memory like a conventional microprocessor with load and store commands that move data between the main memory and a local register file. The SPEs, however, each feature a Local Storage (LS) which is 256 Kilobytes in size. Data transfers between this local storage and the main memory occur with Direct Memory Access (DMA) commands. The Memory Flow Controller (MFC) in each SPE executes

them while the rest of the SPE can work on another task. In conventional microprocessors, memory accesses can take up to 1,000 cycles. The developer can use this time to compute other tasks and thus hide the memory latency.

The PPE is a general-purpose 64-bit RISC, dual-threaded PowerPC processor. Although it is able to execute regular PowerPC programs, this is not advisable for performance-critical applications. It is only designed for the overall control of the Cell Broadband Engine. The PPE supports the PowerPC Architecture Vector / SIMD Multimedia Extension, which is usually referred to as *AltiVec*. It allows the programmer to compute several results at the same time, with 16 byte vectors as the operands. The PPE encompasses the PowerPC Processor Unit (PPU) with the L1 cache as well as the PowerPC Processor Storage Subsystem with the L2 cache.

The SPEs are newly designed processors with a RISC core and a large register file that encompasses 128 entries with 128 bits each. Every SPE contains 256 Kilobytes of local storage for instructions, the stack, and data. Scalar datatypes such as `int` or `float` are not directly supported by these processors. Instead, they execute every operation on the full 128 bits of the SIMD registers. However, the compiler offers a software layer that allows the programmer to use scalar datatypes as if the processor would natively support these. Since the shifting and masking operations are executed in the second pipeline of the SPE in parallel to other operations, this additional layer only leads a negligible decrease in performance. Another highlight of the SPUs is a command that allows the multiplication and addition of three single precision floating point vectors in one cycle, in the form $d = a \cdot b + c$. Using this command and an appropriate program, the CBE can reach its maximum performance: The command executes two floating point operations and every SPE can perform four of these commands per cycle using a SIMD vector with single precision floating point elements. Finally, the CBE features eight SPEs that run with 3.2 GHz. This adds up to a performance of $2 \cdot 4 \cdot 8 \cdot 3.2 \times 10^9 = 204.8$ gigaflops for the SPEs. The PPE itself achieves a peak performance of about 25 gigaflops, which means that the peak performance of the CBE is 230 gigaflops.

However, double precision floating point operations are executed at a much lower speed. Each SPE features only one floating-point unit (FPU) that supports double precision floating point variables, so that SIMD vectors with these values are computed sequentially, needing seven cycles to complete. Consequently, the double precision floating point performance is 14 times lower than the single precision performance, offering less than 15 gigaflops on the SPEs. However, the "Enhanced Cell" design that is announced to be released in 2008 will offer an improved FPU design.

Another problem with developing applications for the SPEs is that these processors do not support a native 32-bit integer multiplication. Instead, they execute these multiplications by cascading five 16-bit integer operations with a seven cycle latency each. Finally, the SPEs do not feature a branch prediction feature and the penalty for a branch miss is eighteen cycles. Thus, it is important for the programmer to add sufficient branch hints to conditional commands.

4.3 Design challenges

The development of new, faster processors is nowadays mainly limited by three factors [4]:

Power Wall Increasingly, the maximum achievable performance of a new processor is

not limited by the number of transistors and wires that can be included, but rather by its power consumption. As a result, it is necessary to increase the power efficiency of the processor by the same factor as the performance increase. The Cell design faces this challenge by introducing specialized processors – the PPU for control-code such as the operating system and the SPUs for the computational code.

Memory Wall The latency for accessing the main memory is currently approaching 1,000 processor cycles, which means that instead of working on the computational problems, the processor often has to wait for a long time for loads and stores to and from the main memory. Although caches are incorporated into all modern processors and do help in certain situations, these do not solve the problem. The Cell processor includes a three-level memory structure: main memory as well as local stores and large register files for the SPUs. Data transfers between the main memory and the local stores are not controlled by the SPU, but instead by the MFC. A total of 128 simultaneous transfers between the eight local stores and the main memory can be executed. This allows the programmer to hide the memory latency by performing the computations and memory transfers in parallel.

Frequency Wall The increase of frequency requires deeper pipelines. We have reached a point where this even leads to diminishing returns in overall performance and even negative returns when power consumption is also taken into account. For this reason, the frequency of the Cell processor does not exceed 3.2 GHz. Instead, the measures described above (specialized processors, several levels of parallelism) were incorporated to face this challenge.

4.4 Linux environment on Sony's Playstation3

The Sony Playstation3 gaming console which is now readily available in stores for less than 600€ features a regular Cell processor, a 20 or 60 gigabyte hard drive and 256 megabytes of RAM. Sony implemented a feature called "Open Architecture" into this console which allows any user to install a Linux operating system (OS) that supports the Cell processor, such as Yellow Dog Linux 5. On this Linux OS, the users can install IBM's Cell SDK (see section 4.5) which allows them to write and compile their own native Cell BE applications. However, the hardware access is restricted by a hypervisor which only allows to use six of the Cell processors' eight SPEs. One of these two SPEs is unusable to improve the factory output for Cell processors. This way, even processors where one of the SPEs is defective can still be used in the Playstation3. The other SPE is used by the operating system and the hypervisor and is unavailable to the user because of that. The biggest limitation, however, is that only simple framebuffer access for graphics output is available. All of the graphics hardware's 2D and 3D acceleration functions are thus inaccessible under Linux, which makes it impossible to implement video games with sophisticated graphics. However, the framebuffer access is fast enough to output video in Full-HD resolution, which means that the performance of graphics output is sufficient for this project.

4.5 Cell BE Software Development Kit

IBM's Cell SDK, which is currently freely available in version 2.1, encompasses all the tools that are required to develop applications for the Cell Broadband Engine. It features 32- and 64-bit versions of the C-compilers GCC and XLC that were modified to generate binaries for the PPU or the SPU respectively. To facilitate the compilation process which usually includes both compiling C-code for the PPU and SPUs with different compilers, the SDK also includes a special GNU Make environment. It includes the necessary steps to compile the code for the PPU and SPU processors and link them to a single executable file. Finally, the SDK also includes modified versions for the PPU and SPUs of the GNU Debugger (GDB) which can be used to debug both the PPU and SPU parts of Cell BE applications on the Linux command line.

Unfortunately, there is currently no integrated development environment (IDE) available for developing Cell applications. The Cell SDK 2.1 includes a plug-in for the popular Eclipse IDE, but my own tests showed that this plug-in is not sufficiently stable at this point and, especially, debugging Cell applications is very slow in the Eclipse environment. For the development of this thesis, I have used Microsoft's Visual Studio 2005 with the setting "external build system" to write the code. In order to automatically transfer the code to the Cell blade or Playstation3 and compile it there with the regular Make environment by simply invoking Visual Studio's build command (F7), I have used a combination of WinSCP's "keep up-to-date" feature and the PuTTY Plink program. For debugging, I have mainly used the command-line version of the GDB debugger (`ppu-gdb` and `ppu-gdbtui`).

4.6 Conclusion

Much of the performance increase of the CBE in comparison to conventional processors is achieved by allowing the programmer to do many things at once: The PPE supports the execution of two threads at the same time, and there are eight SPEs that can work on different tasks independently. In addition to that, DMA transfers from the SPE's local storages to the main memory and vice versa allow the programmer to hide the memory latency on the SPEs by working on other data while the MFC is copying the data to or from the main memory. Also, every processor in the CBE supports SIMD commands with 16 byte registers so that it can work with multiple values in one instruction. Finally, the Cell server blade contains two CBEs, which appear to the programmer as one CBE with a PPE supporting four parallel threads and sixteen SPUs.

Programs that process their data in a linear fashion, like most of today's programs, do not run well on the CBE. Legacy PowerPC applications can run on the PPE, but according to [16] the execution performance will only measure up with a 1.3 GHz Athlon processor. However, when the code is optimized for the Cell processor and exploits the many ways of parallelism that the CBE design provides, according to [3], the resulting performance can be up to ten times better than on a conventional processor running at 3 GHz.

Chapter 5

Aeolus H.264 decoder

In 2007, a team at the IBM Research Lab in Beijing developed a highly optimized H.264 decoder for the Cell Broadband Engine: the Aeolus decoder. This decoder provides real-time decoding performance ($\geq$ 30 fps) for video streams up to Full-HD resolution (1920x1080). It uses three SPEs as well as about 20% of the maximum performance of the PPE.

It supports most of the features of the H.264 Main profile (Level 4), but supports neither PAFF nor MBAFF interlacing. In addition to this, neither Adaptive Memory Management for the decoded picture buffer nor Weighted Prediction are supported. However, Weighted Prediction is used only very rarely in commercial decoders and is not present in any of the video streams of the current HDTV stations in Europe. Finally, both the JM reference decoder and the open source decoder x264, as well as most commercial decoders, include several error concealment methods, but none of these are currently implemented in the Aeolus decoder. Flaws in the video stream, for example those due to bad satellite reception, lead to unpredictable behavior of the decoder.

The implementation of PAFF interlacing into the Aeolus decoder is the topic of this thesis. My development started while the team in China was still working on the decoder, so close cooperation was necessary.

In the following sections, I will give an overview of the parts of the Aeolus decoder, focusing primarily on the areas that I had to modify in order to implement PAFF support.

5.1 Data structures

The data structures of the Aeolus decoder are defined in the file `param.h` in the `include` directory and are used throughout the decoder. The size of these structures is always a multiple of 128 bytes, so that they can be efficiently copied between the local storages of the different SPUs and the main memory using DMA transfers. For this reason, each structure contains a `reserve` variable to increase the size of the structure to the next multiple of 128 bytes.

5.1.1 Video stream parameters

The following structures represent side information that is necessary to decode the video stream:

`sps_info_t` This structure represents the information that the H.264 video stream delivers in a Sequence Parameter Set and contains all the values that do not change

throughout a whole video sequence. Examples of this are the picture size, frame cropping parameters, and whether PAFF or MBAFF interlacing is activated for the video sequence.

pps_info_t This structure represents a Picture Parameter Set which can apply to one or more pictures. It contains values such as the entropy coding method (CAVLC or CABAC) that was used by the encoder to encode the coefficients of each picture to which this PPS applies.

slice_info_t and slice_info_ex_t The structure `slice_info_t` contains values that can be different for each slice of a picture, such as the number of macroblocks for a slice. The structure `slice_info_ex_t` contains the values that will remain the same for each slice of a picture, such as the number of active `L0` and `L1` reference pictures or information for the Adaptive Memory Management. For this reason, the `picture_info_t` structure contains an array of `slice_info_t` structures, but only a single `slice_info_ex_t` structure. However, for technical reasons, the distribution of values does not always follow this rule. For example, although the slice type (I, P, or B) and the slice structure (frame, top field, bottom field) cannot change between different slices of the same picture, they are still stored in the `slice_info_t` data structure.

macroblock_info_t This structure contains the prediction mode for each submacroblock and, in case of Inter-prediction, the motion vectors and the index for the reference pictures in the `L0` or `L1` list.

picture_info_t All the side information that the decoder needs to decode a picture in the video sequence is contained in this structure. It also contains arrays for the `slice_info_t` and `macroblock_info_t` structures. Almost all modules of the decoder access this structure.

Although these data structures are defined in `param.h`, which every module includes and uses, caution must be taken in applying even the slightest change to any of these structures. For performance reasons and possibly time constraints during the development of the decoder, several modules of the decoder access only parts of these structures using DMA transfers. For example, the reconstructor module defines its own data structure (`recon_spu_pic_info_t`) which is similar to `picture_info_t` for the first 144 bytes. A DMA transfer then copies these 144 bytes from one structure to another. Any change of the structure in `param.h` (even a change of order) would thus make the data in the reconstructor module invalid. Another example is that several modules access the slice or macroblock information within the `picture_info_t` structure directly using a fixed offset that is added to the base address of the `picture_info_t` structure. In certain cases, even the `reserve` bytes are used by another module in order to store some information. An example of this is the `reserve_b` field of the `picture_info_t` structure which is used to exchange a pointer between the `recon_ppu` and `reconstructor` module.

For this reason, any change to one of these data structures has to be carefully evaluated, because unexpected behavior in various fields of the decoder can occur with any change. Especially, it is not possible to change the value of the global constants `MAX_REF_PIC_NUM`, `MAX_MACROBLOCK_NUM_IN_PIC`, or `MAX_SLICE_PIC_NUM` without careful

analysis of the implications that such a change will have throughout the whole implementation of the decoder. Especially, any direct access with a fixed offset value will have to be adjusted, otherwise unexpected behavior will occur. Also, several fields of data structures are implicitly aligned to a 16-byte or 128-byte boundary to allow DMA transfers of these fields. Any change of the data structure might move these fields to an address that is not divisible by 16 or 128, causing the corresponding DMA transfers to fail.

It should also be noted that although the value for `MAX_REF_PIC_NUM` is currently set to *fifteen* and thus the appropriate amount of memory is allocated for this number of reference pictures in the data structures, the decoder currently only supports a maximum of *eight* reference pictures. This is because the highly optimized code of the reconstructor module uses only three bits to store the number of the reference picture in the prefetch function. According to the project leader Yu Yuan, changing this behavior would be very time consuming and even eight reference frames are actually more than today's commercial encoders use.

During the implementation of this thesis, these constraints had to be kept in mind, as it was necessary to change several data structures to incorporate the values necessary for PAFF interlacing.

5.1.2 Picture data: Coefficients and samples

The actual picture data is stored in the following three data structures:

`coef_macroblocks_picture_t` This structure contains a pointer to the coefficients that represent the residual data of the image. The reconstructor module uses these coefficients to transform them into the actual pixel values of the output picture.

`pixel_macroblocks_picture_t` This structure contains a pointer to the actual pixel values of a picture. For each macroblock, the 16x16 pixels of the luma pixel values are stored first, followed by the two 8x8 pixel planes of the chroma values. The macroblocks are then stored from top left to bottom right. So, in order to access the macroblock at position (x, y), the formula $((y \cdot w) + x) \cdot 384$ can be used, with w being the width of the picture in macroblocks.

`output_picture_t` This data structure contains three pointers to the luma and chroma planes of a picture. Unlike `pixel_macroblocks_picture_t`, the pixel values in this data structure are stored separately for luma and chroma and are not ordered by macroblocks but by actual sample positions, meaning pixel by pixel from top left to bottom right. Since the decoder works on 4:2:0 video streams, the width and height of the chroma planes are half the size of those of the luma plane. This is the same format that the decoder outputs after decoding a video stream.

5.1.3 Module parameters

These structures contain pointers to the input and output parameters of the three SPE modules:

`parser_param_t` This structure contains pointers to the input and output structures that the parser uses. The input is the bitstream buffer, which simply contains the

raw H.264 video stream. The output is the SPS, the picture information, and the coefficients of the picture.

`reconstructor_param_t` The input to the reconstructor is the picture information, the picture coefficients, and the picture information and sample values of the reference pictures. The output is the unfiltered reconstructed picture in macroblock format.

`filter_param_t` Finally, the filter module's input is the picture information and the picture in macroblock format. As the output, it will overwrite the picture in macroblock format with the filtered version and create an output picture in planar YCbCr.

5.2 Decoding a picture

Figure 5.1 shows the control flow during the decoding process of a picture in the video stream. First, the scheduler starts the parser module, which parses the incoming video stream and performs the entropy decoding process. When the input buffer does not contain a full picture, it sends a message to the scheduler, which then refills the input buffer with the following data. When the entropy decoding of a picture is complete, the parser module exits and the scheduler passes the resulting data to the reconstructor module, which performs the main part of the decoding process. The reconstructor module sends its output directly to the filter module, which applies the deblocking filter to the resulting picture data. Finally, when the picture is fully decoded and filtered, the filter SPU exits and the scheduler eventually outputs the picture, following the output order.

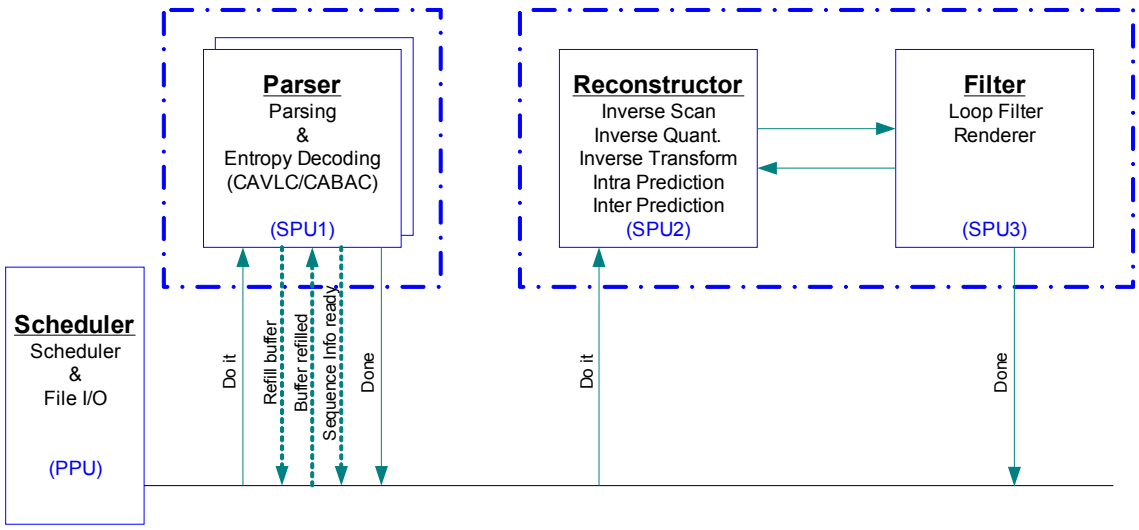


Figure 5.1: Control flow in the Aeolus H.264 decoder [5]

5.2.1 PPE: Scheduler

The PPE runs the main program which reads the input stream consisting of raw H.264 data and writes the output stream consisting of raw YCbCr data. It also sends the input and output data for the stream decoding to the different modules using buffers.

For each SPE, the scheduler has a state variable which can be set to `IDLE`, `DOING`, or `DONE`. At the start of the program, all state variables are set to `IDLE`.

The scheduler communicates with the SPEs using mailbox messages. In order to start a module whose state is currently `IDLE`, the scheduler puts a message `MSG_DO_IT` into the inbound mailbox of the SPE, followed by messages with the address of the parameter structure for the module. It then sets the state variable for this module to `DOING`. The SPE then loads the parameter structure into its Local Storage using a DMA transfer and starts working on the task. As soon as it is done with its work, it puts a `MSG_DONE` message into its outbound mailbox. The scheduler reads this message in its `state_check` function and changes the module's state variable to `DONE`. Finally, in the `done_to_idle` function, it changes the state back to `IDLE`.

5.2.2 SPE: NAL parsing and entropy decoding

In the current implementation of the Aeolus decoder, there are two distinct parser modules: the CAVLC- and the CABAC-parser. The modules are split into two parts: the "pre-parser" decodes the NAL structure of the video stream and reads the side information such as the SPS, PPS, and slice headers. The H.264 standard document [7] describes the syntax of these header informations in section 7.3. For the CABAC- and CAVLC-parser, the code of the pre-parser is virtually identical. It would be possible to use a single pre-parser module for both parsers, but since the CABAC and CAVLC modules were developed independently by different people in the development team in China, the pre-parser code of the CABAC-module was simply copied from the CAVLC-parser, which makes it necessary to also duplicate any change to the header parsing code to both modules.

The pre-parser fills the structures that define the stream side information which I introduced in section 5.1.1 and stores them in main memory. When the decoding of the SPS is complete, the parser puts a mailbox message into its outbound mailbox, so that the scheduler can initialize its data structures according to the picture size and stream parameters that are defined in the SPS.

After the pre-parser, the main parser performs the entropy decoding process for the residual data and stores these in the macroblock coefficients structure. It also performs the inverse zig-zag-scan of the transform coefficients (see section 2.3). Afterward, it stores the resulting data in the `coef_macroblocks_picture_t` structure.

5.2.3 PPE: Preparations for the Reconstruction

After the parser module, the scheduler passes the data to the `recon_ppu` module. It decodes the motion vectors along with the other side information that is necessary for the Intra- and Inter-prediction. This part is done entirely on the PPE because it is necessary to access many different values from the reference pictures and the current picture in order to decode the motion vectors. It would not be possible to keep all the data structures in the local storage of an SPE at the same time, so it is much easier to perform this work on the PPE. It produces a workload of about 20% on the PPU.

In the next step, which is also fully executed on the PPE, the current version of the Decoded Picture Buffer is generated. Since up to three pictures of the video stream are decoded at the same time with the Aeolus decoder, it is not sufficient to simply create a version of the DPB and dismiss it as soon as the parsing of the next picture is complete.

Instead, all the decoded pictures that can be used for reference in later pictures will be stored in a decoded picture pool. Then, a DPB snapshot will be stored with the picture information for each picture. A picture will not be removed from the decoded picture pool before the picture is not used anymore in any of the active DPB snapshots. Figure 5.2 illustrates this concept. The parser has just finished with picture $k + 2$, so the DPB state for this picture will be generated. Picture $k+1$ might just be in the `recon_ppu` step while the reconstructor module is processing picture k. Despite this parallelism, picture a is in the correct position within each picture's DPB.

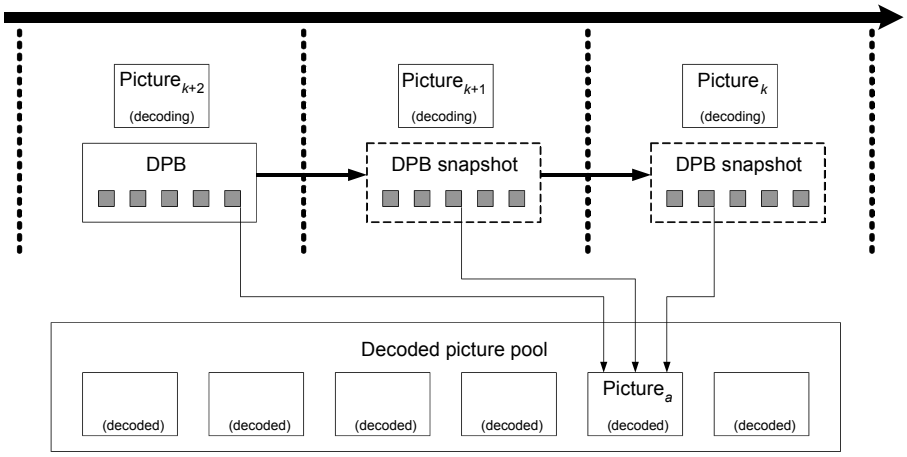


Figure 5.2: Implementation of the DPB in the Aeolus H.264 decoder [5]

5.2.4 SPE: Reconstruction process

After the preprocessing step, the scheduler passes the picture to the reconstructor module. This uses the coefficients from the parser and performs the inverse quantization and transformation. Afterward, it uses the motion vectors that the `recon_ppu` module decoded along with the reference pictures from the picture's DPB to perform the Intra- and Inter-prediction steps. The result of this step is a fully decoded (although unfiltered) picture whose sample values will be stored, in macroblock order, in a `pixel_macroblocks_picture_t` structure.

The `recon_ppu` module already decoded all the motion vectors and stored them in the picture info structure along with the information of which reference picture the vectors are pointing to. Because of this, the reconstructor module can immediately load the picture data from the reference picture that it needs to perform the motion compensation for the current picture. For this purpose, a prefetch routine starts a DMA transfer for the following macroblocks while a certain macroblock is being decoded. Because of this, the loading of the picture data for the motion compensation is performed while the reconstructor works on the calculation of the output sample values.

5.2.5 SPE: In-loop deblocking filter

The reconstructor module communicates directly with the filter module which starts with the deblocking process as soon as the reconstructor finishes with the first line of macroblocks of the picture. It filters these macroblocks and writes them back into the `pixel_`

`macroblocks_picture_t` structure in the main memory as well appropriately reordered to the output format into an `output_picture_t` structure.

5.2.6 Overview of the data flow

Figure 5.3 illustrates the data flow between the different SPEs and the main memory during the decoding process of a picture in the video stream, which I have described in this section.

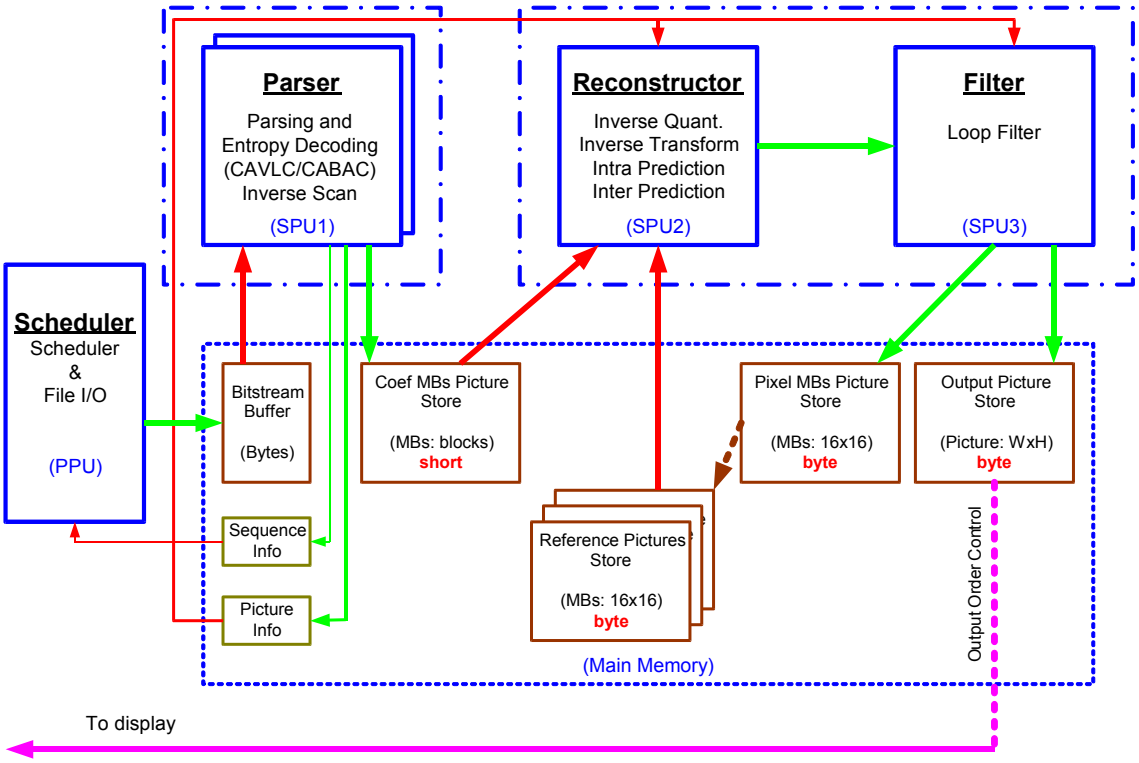


Figure 5.3: Data flow in the Aeolus H.264 decoder [5]

5.3 Performance evaluation

In order to measure the processor utilization, the Aeolus code uses the SPU performance counters, which provide a very exact timing method. This makes it possible to precisely measure the time that each SPE is working on its task by determining the time from the assignment of a new task (state transfer from `IDLE` to `DOING`) until its `DONE` message is sent. In addition, the time the the SPE is in its `IDLE` state can be measured. By comparing these two values, it is possible to calculate a percentage for the SPE utilization. A similar method was applied to measure the PPU utilization for the `recon_ppu` module.

With these methods, the team of the IBM China Research Lab has evaluated the performance of the decoder. For a wide variety of streams in Full HD resolution that were decoded in real-time, a PPU load of about 20% for the `recon_ppu` module was determined, while the SPU loads can vary widely depending on the actual stream, but will clearly stay below 100%. [5]

However, when measured with the Linux `top` utility, the current code generates a 100% PPU load. This is because the main loop checks the SPE states as often as possible.

A possible solution to allow the PPU to work on other tasks while the three SPEs are decoding the video stream would be the implementation of `nanosleep`-commands into the main loop.

Chapter 6

Implementation of PAFF interlacing

In this chapter, I will describe the implementation of Picture-adaptive frame/field (PAFF) interlacing into the Aeolus H.264 decoder. In each section, I will outline the respective feature followed by a description of its implementation into the Aeolus decoder. The title of each section also contains the number of the sub-chapter of the H.264 standard document [7] in which each feature is described.

Since the development of the Aeolus decoder was not finished when I was working on my thesis, it was necessary to implement the code in a way that would not interfere with the original code and allow the team in China to perform the testing without influences from my code changes. For this reason, all the code changes are in `#ifdef _DISABLE_PAFF_` environments, so that a simple `#define _DISABLE_PAFF_` in the file `param.h` is sufficient to deactivate all my code changes and compile the original, frame-only decoder.

6.1 Syntax elements (7.3.2)

6.1.1 Description of relevant syntax elements

If the input video stream consists of interlaced material, such as SDTV streams or 1080i HDTV streams, the flag `frame_mbs_only_flag` in the Sequence Parameter Set (SPS) is set to 1. In this case, the flag `mb_adaptive_frame_field_flag` is also stored in the SPS and is set to 1 when the encoder has used the MBAFF method to encode the video stream and 0 if it only used the PAFF method.

The parameter `pic_height_in_mbs` of the SPS contains the height of a field in macroblocks if `frame_mbs_only_flag` is set to 1. Otherwise it contains the height of a frame.

Similarly, if `frame_mbs_only_flag` is 1, the slice header contains the parameter `field_pic_flag`, which is 0 if the slice is part of a frame. If, however, it is part of a field, the parameter will be 1 and the parameter `bottom_field_flag` immediately follows. The second parameter is 1 if the slice belongs to a bottom field and 0 if it belongs to a top field.

Finally, the parameters `delta_pic_order_cnt_bottom` and `delta_pic_order_cnt[1]` are only stored in the slice header if `field_pic_flag` is set to 1 for the same slice.

6.1.2 Implementation into the pre-parser

It was necessary to implement the code to parse these syntax elements into the pre-parser of both the CAVLC- and CABAC-parser modules. They are implemented in the file `parser_proc.c` in the directory of each parser module.

The field `pic_height_in_mbs` of the `picture_info_t` structure will be set to the actual height in macroblocks of the picture. This means that its value for a field will be half the value for a frame in the same video stream.

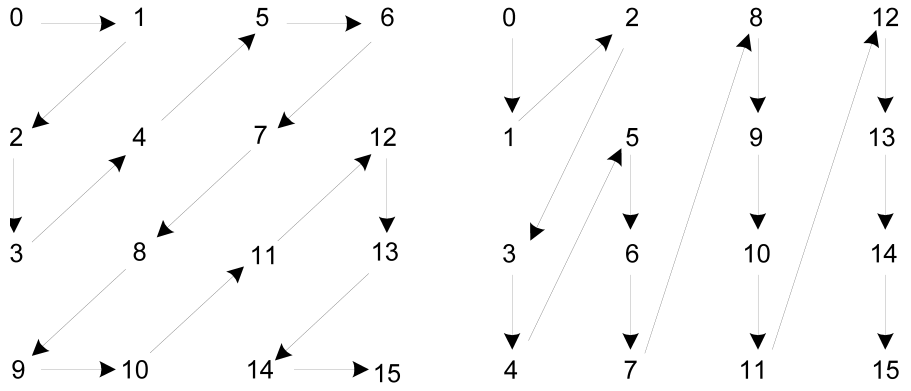
In order to store whether a slice is part of a frame or field, along with the parity of the field (i.e. top or bottom), I have introduced the new field `slice_structure` into the structure `slice_info_t`. The pre-parser will set this field to `FRAME`, `TOP_FIELD`, or `BOTTOM_FIELD`.

Since the parser for the residual data needs information about both the actual picture size and the slice structure, I have modified the data in the `swap_head_set` accordingly.

6.2 Inverse zig-zag scan for transform coefficients (8.5.5, 9.3)

6.2.1 Description of zig-zag pattern for frames and fields

In the transform and quantization step of the video encoding, the encoder stores the AC-coefficients of each macroblock in a zig-zag pattern to improve the entropy coding performance. In H.264, the encoder stores each 4x4 submacroblock separately. The pattern for the zig-zag scan for macroblocks in frames is different from the zig-zag scan pattern in fields. Figure 6.1 shows both patterns.



(a) Frames (b) Fields

Figure 6.1: 4x4 block zig-zag scans [7]

When a video stream was encoded with the CABAC entropy coding method, the initialization values for the CABAC encoding and decoding process are different depending on whether the current picture is a frame or a field. These values define the probability model that is used for the entropy coding. Because of the different zig-zag patterns, two distinct models were defined in the standard. The article [11] gives reasons for this choice in section B (“Context Modeling”).

6.2.2 Implementation

The inverse block scan is implemented in several parts of the Aeolus decoder: The CAVLC- or CABAC-parser processes the AC-coefficients and the reconstructor processes the DC-coefficients for Intra 16x16 encoding.

Both parser modules contain the function `shuffle_coef` which performs the inverse block scan followed by reordering the samples into another format which some parts of the decoder require. In this function, two calls to `spu_shuffle` perform this zig-zag reordering. The SIMD command `spu_shuffle` of the SPU processor allows the programmer to merge two existing vectors into a new one. The command takes a pattern which consists of the same number of bytes as the new vector and specifies how the new vector will be generated. Each byte of the pattern defines from which position in either of the two existing vectors the byte shall be copied. Each coefficient of a 4x4 submacroblock consists of two bytes, so this function allows the reordering with two SIMD-commands, which is a highly efficient way of performing the reordering.

In the CAVLC-parser, I have changed `shuffle_coef` so that it takes the shuffle patterns as a new parameter. Then, I changed the calls of this function to include the appropriate shuffle pattern, depending on the the structure of the current picture.

The CABAC-parser is implemented differently, so this method would not be feasible for this code. The CABAC-parser relies heavily on the use of global variables to define the state that the parser is in. For this reason, I have introduced two new global variables to store the shuffle pattern, which I set to the correct pattern in the function `cabac_slice_init`, where all the other global variables for the current slice are initialized. The function `shuffle_coef` uses these shuffle patterns to perform the inverse block scan.

Finally, in the reconstructor module, I was able to implement the selection of the appropriate shuffle patterns directly into the function `recon_transform_luma_dc`.

In order to implement the different probability models for the CABAC parser, some additional changes were necessary. The file `cabac_spu_tables.c` contains the different initialization values of the CABAC module. In this file, I have added the new arrays `cabac_init_sig_map_fld` and `cabac_init_last_coeff_fld`, both for I- and P-fields. Depending on whether the currently decoded picture is a frame or a field, the new global variable `g_field_flag` will be set to 0 or 1, respectively, in the function `cabac_slice_init`. Then, when the contexts are initialized, the appropriate values for frames or fields are used.

6.3 Detection of the first slice of a picture (7.4.1.2.4)

6.3.1 Description

Since each picture can consist of any number of slices, it is not a trivial matter to determine whether a new slice belongs to a new picture or is still part of the previous picture. The standard defines the following method to determine the first slice of a picture. If any of the following conditions is fulfilled, the current slice header belongs to a new picture:

- `frame_num` of the slice headers differs in value
- `pic_parameter_set_id` of the slice headers differs in value
- `field_pic_flag` of the slice headers differs in value

- `field_pic_flag` is equal to 1 in both slice headers and `bottom_field_flag` differs in value
- `nal_ref_idc` of one of the two slice headers is 0, while the other is not
- `nal_unit_type` of one of the two slice headers is 5 (i.e. the slice is part of an IDR-picture), while the other is not
- `nal_unit_type` of both of the slice headers is 5 and `idr_pic_id` differs in value
- `pic_order_cnt_type` from the associated SPS is equal to 0 for both and the slice header parameter `pic_order_cnt_lsb` or `pic_order_cnt_bottom` differs in value
- `pic_order_cnt_type` from the associated SPS is equal to 1 for both and the slice header parameter `delta_pic_order_cnt[0]` or `delta_pic_order_cnt[1]` differs in value

6.3.2 Implementation

This detection is also part of the pre-parser, so it was necessary to implement the code into the pre-parser of the CAVLC- and the CABAC-module. In the original code, this part of the standard is not implemented at all. Instead, the code only checks whether the value `first_mb_in_slice` of the slice header is 0. If it is, a new picture is assumed. This implementation is sufficient for the wide set of video streams that the development team used to test the decoder. However, when decoding video streams that were encoded with PAFF interlacing, this is not sufficient.

My implementation of the standard-compliant method is straightforward: A set of variables stores the last slice header. Then, for each new slice, the code compares the parameters of the current slice header with the previous one according to the rules described above. However, because the current version of the Aeolus decoder does not support the change of the SPS while decoding a stream, I did not implement the two rules that depend on `pic_order_cnt_type`.

6.4 Calculation of the Picture Order Count (8.2.1)

6.4.1 Description

In order to allow the decoder to output the pictures of a video stream in the correct output order, the H.264 standard defines a 16-bit signed variable called Picture Order Count (POC) which both the encoder and decoder derive from other variables in the video stream. For all pictures in the video sequence between two IDR pictures, the POC values are unique. Sorting the pictures in ascending order by their POCs defines the output order.

For each frame or complementary field pair, two POC values are defined: *TopFieldOrderCnt* and *BottomFieldOrderCnt* for the top and bottom fields, respectively. The frame's POC is then defined by the operation Min(*TopFieldOrderCnt*, *BottomFieldOrderCnt*). For fields, only the respective value is used. The POC of an IDR picture is always 0.

The actual calculation of the POC values depends on the SPS parameter `pic_order_cnt_type`, which can be 0, 1, or 2. The H.264 standard contains a

detailed description of how to calculate the values *TopFieldOrderCnt* and *BottomFieldOrderCnt* depending on the POC type in section 8.2.1.1, 8.2.1.2, and 8.2.1.3. The input values to this process are the slice header parameters `pic_order_cnt_lsb`, `delta_pic_order_cnt_bottom`, and `delta_pic_order_cnt` as well as the parameters of the previously decoded picture.

6.4.2 Implementation

In the Aeolus decoder, the function `calculate_poc` in `dpb.c` performs the calculation of the POC for each frame. Since this calculation process already includes the calculation of *TopFieldOrderCnt* and *BottomFieldOrderCnt*, only small changes to the code were necessary. I have added the array `pocs[]` to the `picture_info_t` structure, which contains the top, bottom, and frame poc for each picture. In the original code, only the frame poc was stored in the `poc` variable of the `picture_info_t` structure. The new function `calculate_pocs` is part of the new Decoded Picture Buffer.

6.5 Decoded Picture Buffer (8.2.4, 8.2.5)

6.5.1 Description

The Decoded Picture Buffer (DPB) was already introduced in section 3.5.3 of this thesis. For each frame, the DPB stores the top and bottom field as well as the full frame. Consequently, for a complementary field pair, the DPB will contain the two fields as well as the frame that is made up from the combination of the top and bottom field. When decoding a frame, the reference lists contain previously decoded frames as well as previously decoded complementary field pairs. When decoding a field, they will contain previously decoded fields as well as the fields of previously decoded frames.

The PPS and the slice header contain parameters that define the number of active reference pictures, i.e. `num_ref_idx_l0_active_minus1` and the corresponding parameter for the `L1`-list. If these are present in the slice header, they will override the ones in the PPS.

When adaptive memory management is activated in the video stream, which is currently not supported by Aeolus, pictures can be marked as long-term reference pictures with a *LongTermPicNum*. The decoder will not automatically remove long-term reference pictures from the DPB. Instead, they will remain in the DPB as long as the encoder defined.

6.5.1.1 Calculation of the picture numbers

At the beginning of the decoding process for a new picture, the decoder calculates picture numbers for all the pictures. The results will be used later to generate the reference picture lists `L0` and `L1`.

The value *FrameNumWrap(i)* of the picture *i* in the DPB is derived from the slice header parameter `frame_num` and calculated as follows: If the frame number *FNum(i)* of a picture in the DPB is smaller than the new picture's *FNum(c)*, *FrameNumWrap(i)* is set to *FNum(i)*. Otherwise, *FNum(i)* is set to *FNum(i)* minus *MaxFrameNum*, with *MaxFrameNum* calculated from the SPS parameter `log2_max_frame_num_minus4`.

The value *PicNum(i)* of a frame in the DPB is the same as the frame's *FrameNumWrap(i)*. For fields, the following rule applies: If the parity of the field in the DPB is the same as the new field's parity, *PicNum(i)* is 2 * *FrameNumWrap(i)* + 1, otherwise it is 2 * *FrameNumWrap(i)*.

6.5.1.2 Initialization of the reference lists for frames

For **P-frames**, the list `L0` starts with all the short-term reference pictures, ordered by their *PicNum* in descending order. The long-term reference pictures follow, ordered by their *LongTermPicNum* in ascending order.

In **B-frames**, the list `L0` starts with the short-term reference pictures that have a smaller POC than the current frame, sorted by their POCs in descending order. The list continues with the rest of the short-term pictures, sorted by their POCs in ascending order. Finally, all the long-term reference pictures will be added, sorted by their *LongTermPicNum* values in ascending order.

The list `L1` in B-frames starts with the short-term reference pictures that have a bigger POC than the current frame, sorted by their POCs in ascending order. Then, the rest of the short-term reference pictures are added, sorted by their POCs in descending order. Finally, the long-term reference pictures follow, sorted by their *LongTermPicNums* in ascending order.

If the reference lists `L0` and `L1` are identical after the initialization, the first two entries of the reference list `L1` will be switched.

6.5.1.3 Initialization of the reference lists for fields

In fields, the generation of the reference lists takes two steps: First, the decoder generates the lists *refFrameList0ShortTerm*, *refFrameListLongTerm* and in case of a B-field also *refFrameList1ShortTerm*. Then, it creates the actual reference lists `L0` and `L1` from these lists.

In **P-fields**, the list *refFrameList0ShortTerm* contains all the short-term reference pictures, sorted by their *FrameNumWrap* values in descending order. The list *refFrameList0LongTerm* contains all the long-term reference pictures, sorted by their *LongTermFrameIdx* values in ascending order.

For **B-fields**, the list *refFrameList0ShortTerm* starts with all the short-term reference pictures that have a POC that is smaller than the current picture's POC, sorted by their POCs in descending order. The rest of the list consists of the remainder of the short-term reference pictures, sorted by their POCs in ascending order. The list *refFrameList1ShortTerm* starts with all the short-term reference pictures that have a POC that is bigger than the current picture's POC, sorted by their POCs in ascending order. The rest of the list consists of the remainder of the short-term reference pictures, sorted by their POCs in descending order. The list *refFrameList0LongTerm* contains all the long-term reference pictures, sorted by their *LongTermFrameIdx* values in ascending order.

After these initializations, the decoder will create the actual reference lists `L0` and `L1`. The algorithm is identical for both lists, so the letter *X* in the following description means either 0 or 1. The reference list `LX` starts out with the first picture in the list *refFrameListXShortTerm* that has the same parity (i.e. top or bottom field) as the current field. If no such picture is present in the list, this step will be skipped. Then, the rest of the list `LX` will be filled with fields of alternating parity, keeping the order that the pictures had

in the list *refFrameListXShortTerm*. If at any point there are no pictures of the required parity left, the rest of the list `LX` will be filled with the pictures of the available parity.

After completing the process for the the short-term reference pictures, the decoder will add the long-term reference pictures from *refFrameListLongTerm* to `LX` with the same algorithm.

6.5.2 Implementation

The existing implementation of the DPB in `dpb.c` did not fully implement the rules of the H.264 standard. Instead, only a simplified version is implemented which offers enough functionality for video streams that contain only frames. The information for the DPB is stored in several different data structures, while the actual memory management is performed in the scheduler module (`decoder.c`). Because of the number of new rules that I had to implement for the support of fields, I implemented a completely new DPB which the decoder uses when the PAFF support is activated.

The new DPB is distributed over three files: `dpb2.c` contains the main part of the implementation. It manages the pictures in the DPB and offers functions to put a new picture in the DPB, generate the reference lists, and update the snapshot of the reference lists in the `picture_info_t` structure (see section 5.2.3). The module `dpb2_io.c` performs the memory management and sets the DPB information in the reconstructor parameters. In this way, it provides an interface to the rest of the decoder. Finally, `dpb_field_ref.c` creates the reference lists for fields. The functions in this file will be called by `dpb2.c`, not by the main program.

The picture information structure, the coefficients, macroblocks, and output data are stored in a picture pool. This pool contains all the pictures that are currently active in the decoder. Being active means that either the parser, reconstructor, or filter module is processing them, that they are part of the output queue, or that they are part of a DPB of another active picture.

Each picture in the DPB will be stored in a variable of the new data type `decoded_picture_t`. This structure contains the picture information, the picture data and the picture numbers. Another data structure of the new type `dpb_t` contains an array of these pictures along with the reference lists and the picture that is currently being decoded.

6.5.2.1 Memory management

In the original implementation of the Aeolus decoder, the decoder allocated the amount of memory that could be maximally needed at the start of the program and did not free it before the decoder finished the decoding process. The available memory positions were kept in an array of pointers. Another array of boolean values (defined in `pool.c`) was used to keep track which element of the memory pool is currently in use and which ones are free.

This strategy has one major drawback. No matter how many reference pictures the current stream uses, the memory requirement of the decoder will always be fixed to the maximum amount of reference pictures supported by the decoder. This problem intensifies in the case of PAFF streams, where not only the reference pictures themselves will be stored, but, in the case of frames, also the two fields, and, for fields, also the complementary field pairs. This necessity further increases the memory requirements. The Playstation3 only has 256 megabytes of main memory, so an economical usage of the

memory is absolutely necessary for the media center project, where a large number of other modules will be loaded at the same time and also require significant amounts of memory.

However, since there are several versions of the DPB active at any point in time in the Aeolus decoder, it is not sufficient to free the memory for the reference pictures when the decoding process of a picture is done. For this reason, the module `dpb2_io` contains a picture pool which contains pointers to all the data structures that belong to a picture along with the additional picture data for PAFF streams, for instance the fields of a frame or the complementary field pair for fields. The memory for these structures will be allocated and freed dynamically as needed.

Each element of the picture pool also contains a status variable which can be `INVALID`, `DECODING`, `WAITING_FOR_OUTPUT`, or `DONE`. The process for a picture is as follows:

Start of the decoding process (before parser) The `picture_info_t` structure as well as the `coef_macroblocks_picture_t` structure will be allocated and the status of the picture in the pool is set to `DECODING`. The parser module will then fill these structures with the appropriate information.

Before the reconstruction process The reconstructor module will decode the coefficients and generate the sample data in macroblock format. Hence, the `dpb2_io`-module allocates the `pixel_macroblocks_picture_t` structure. It uses the appropriate size for frames and fields, so that it allocates twice as much memory for a frame than it does for a field.

Start of the filter module The filter module applies the deblocking filter to the picture data and reorders the picture samples to planar YCbCr data. It writes its output to an `output_picture_t` structure. When the decoder processes a frame, the `dpb2_io` module will allocate the memory for the output picture. Since fields will be output in complementary pairs, the memory allocation is only required when the first field of a pair is being decoded. For the second field, the filter module will simply write the output data into the picture information of the first field.

After the reconstruction process Once the reconstruction process of a picture is completed, the macroblock coefficients are not needed anymore, so `dpb2_io` frees this structure at this point.

After the filter module If the picture is marked as “not used for reference”, neither the macroblock values nor the picture information are needed anymore, so these structures will be freed. Depending on whether the picture was already written to the output buffer or not, the picture status will then be either set to `WAITING_FOR_OUTPUT` or `INVALID`. If the picture is used for reference, these structures will be needed for the Inter-prediction process, so they cannot be freed yet. Instead, the picture status will either be set to `WAITING_FOR_OUTPUT` or `DONE`.

Output complete Once the picture was written into the output buffer, the picture information in the `output_picture_t` buffer are not needed anymore, so `dpb2_io` frees this structure. The picture status will then either be `DONE` if the picture is still used for reference or `INVALID` if it is not.

Free unused reference pictures When a new picture is added to the DPB, one or more pictures that were previously part of the DPB can be removed. However, since more than one picture is processed at the same time in the Aeolus decoder, it is not possible to free the memory right away. Instead, the DPB sets the pointer `free_with_pic` to the current picture information structure. When this picture is decoded, `dpb2_io` frees the picture information structure and the picture data (including the additional pictures for PAFF streams) of the picture that is now not used for reference anymore. If the picture has already been written to the output buffer and its status set to `DONE`, then all the picture information is freed and the status is set to `INVALID`.

6.5.2.2 Data structures

All the DPB information is stored in the `dpb_t` data structure, which contains the current reference picture lists `L0` and `L1`, as well as the picture data for all the pictures in the DPB. When PAFF interlacing is activated for the current stream, the `pics` array contains the two fields and the frame (or complementary field pair) for each picture. The implementation actually merges complementary fields into new reference frames in order to store them in the DPB and use them later for reference. Consequently, it also splits frames in the video stream into their two fields. Although it would be more memory efficient to always store full frames and access the required field directly in this frame, this solution would have required vast changes in the reconstructor module, which would have surpassed the scope of this thesis. For this reason, I have chosen to take the same approach as the JM reference decoder, which also creates the fields of a frame and the complementary field pair for two fields when they are added to the DPB. This made it possible to use the existing, highly efficient code in the reconstructor module with only few modifications.

6.5.2.3 Reference picture list generation

The new DPB has the same interface for `decoder.c` as the old one. An initialization function is called at the start of the decoder. The other functions have the following functionality:

`put_reconstructing_pic` The scheduler calls this function before it starts the reconstructor module for a new picture. The function will calculate the POCs of the new picture and put all the picture's information into the `dpb.new_pic` variable. If the second field of a field pair is passed to this function, the variable `new_comp_ref` will be set to the position of the complementary field for the current field, so that the second field can be stored together with the first field in the DPB and a complementary field pair can be created.

`generate_ref_lists` This function assigns the picture numbers to all the pictures in the DPB and then generates the reference picture lists. These lists are arrays that contain the indices of the pictures in the `dpb.pics` structure. The reference list generation is performed in two parts: first, a function copies all the required pictures (for example all the short-term reference pictures) into an array. This is then sorted in the required order using the C-function `qsort`. When the lists are completed, `generate_ref_lists` will copy the reference lists and an array of

the `frame_num` and `structure` values into the `picture_info` structure of the new picture. The reconstructor module will later use these values to perform the Inter-prediction process.

`update_dpb` After the reconstruction process of a picture is complete, this picture will be permanently added to the DPB, so it will not be stored in the `new_pic` variable anymore, but instead be a part of the `pics` array. If the DPB already contains the maximum number of reference frames for the current stream, as defined in the SPS, this function will perform sliding window memory management and delete the oldest picture with the smallest `frame_num_wrap` value from the DPB.

6.5.2.4 Generation of field pairs for reference pictures

As I stated in section 6.5.2.2, the DPB does not only contain the frames and fields in the video stream, but also the two fields that make up every frame, and the frame that is made up of each complementary field pair. The generation of these additional reference pictures takes some time to complete, especially when the video stream consists of pictures in Full-HD resolution. For this reason, I have implemented the process of merging these pictures in the filter module on the SPU. Since the filter SPU contains the last steps of the decoding process and thus the finished image data will stream through this SPU, this was a natural choice. In addition to that, the reconstructor and parser SPEs already use nearly all of their resources (both processing time and memory in the LS), while the filter SPU can still accommodate these additional functions.

The filter module outputs the filtered picture to the main memory one row of macroblocks at a time. For each of these rows of the frame, one half of the respective macroblock rows in the top and bottom field can be generated. Figure 6.2 illustrates this splitting process for one macroblock. The function processes each macroblock of the row that is already in the local store of the SPU. This macroblock row contains regular frame macroblocks, as illustrated in the figure on the top left. Then, the function copies the top and bottom lines to a new buffer in the LS. In this buffer, the halves of the top and bottom macroblocks are stored below each other, as shown in the figure on the top right. Finally, the function uses a DMA list transfer to copy these macroblock halves to the correct position in the field buffers in the main memory. For frame macroblock rows with an odd number, the top half of the field macroblocks is filled (as shown in the figure on the bottom), while for even rows the lower half will be filled. Each transfer is aligned to a memory address that is divisible by 16, because the transfer sizes are 128 and 32 bytes. Hence, the DMA list transfer works without further problems. The DMA transfers are double buffered, so that the DMA controller transfers one macroblock to the main memory while the splitting function is working on the next one.

For the combination of two fields to a reference frame, the process is more complicated. The lines of a single field macroblock need to be copied into two vertically adjacent macroblocks in the frame. Also, since only one field is in the local store of the SPU at one time, a direct implementation would require DMA transfers of 16- and 8-byte data blocks (for each Y, Cb, Cr line in the macroblock), which would be highly inefficient. Because of this, I implemented a different approach which merges the macroblocks in the local store. The concept behind this algorithm is to load two existing frame macroblocks into the local store and then fill the current field into these macroblocks. Figure 6.3 shows an example that illustrates this concept. In this example, the macroblocks in the main memory already contain the top field of the reference picture, while the filter module is

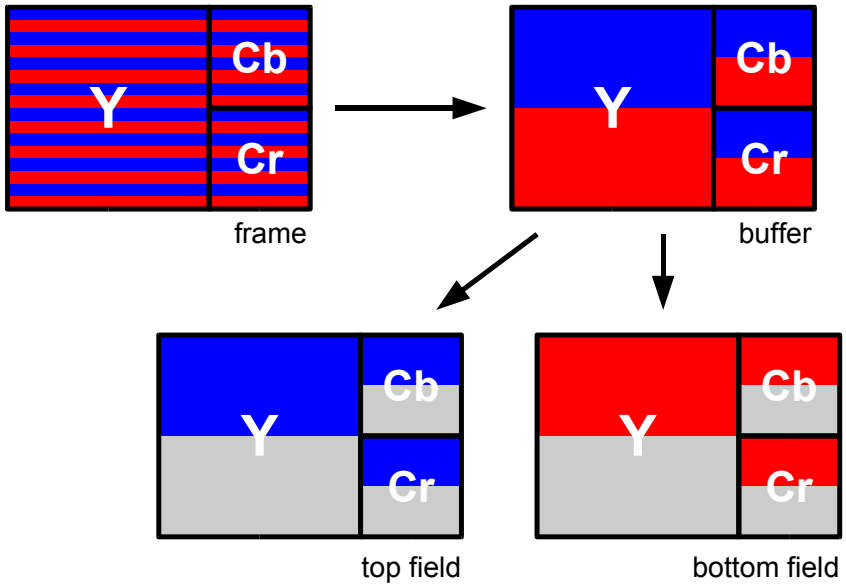


Figure 6.2: Generation of the two reference fields for a frame

just filtering the bottom field of the picture. The algorithm will load the two macroblocks of the frame with the top field into the LS, copy the bottom field into these macroblocks, and store the resulting frame macroblocks back in the main memory.

A ring buffer with four elements contains the macroblocks that the function is currently loading from main memory, storing to main memory and on which it is currently working. This is necessary to effectively hide the memory latency and perform the combination during the memory transfers. A total of three buffers would have been sufficient, but using four buffers allows a more efficient implementation, because the command `buf_num=(buf_num+1)&3;` is enough to both increment the buffer number and wrap around to buffer 0 after buffer 3. With only three elements in the ring buffer, a conditional statement would have been necessary, which is inefficient on the SPUs.

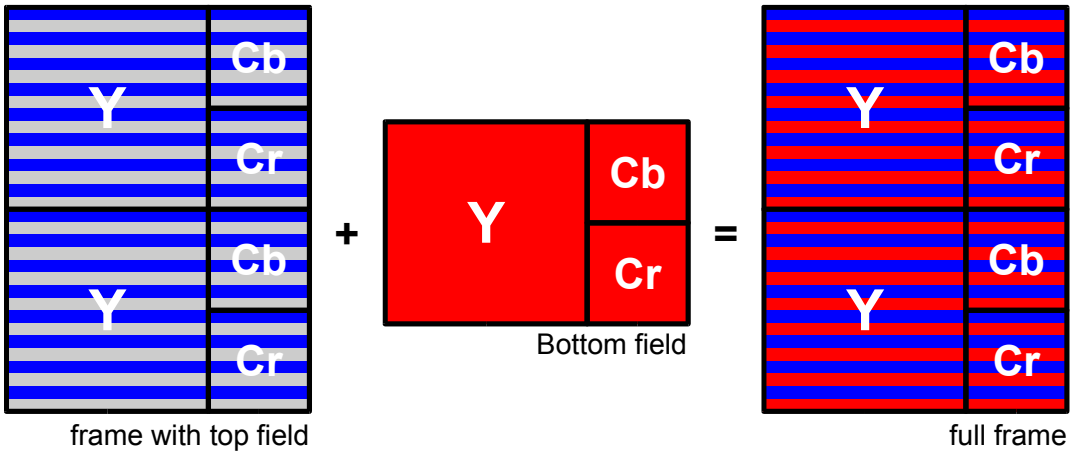


Figure 6.3: Generation of the reference frame for a complementary field pair

6.6 Inter-prediction process (8.4.1)

6.6.1 Description

For an Inter-predicted macroblock, the decoder generates the prediction from a co-located macroblock in a reference picture and a motion vector. In B-slices, the variable `colPic` contains the reference picture. In frame mode, it is simply set to the first entry in the reference picture list `L1` (i.e. `L1[0]`). The same is true if this reference picture and the current picture are both fields. If the current picture is a field and `L1[0]` is a field of a frame, then `colPic` is set to this frame. Finally, if the current picture is a frame and `L1[0]` is a complementary field pair, then `colPic` is set to either the top field or the bottom field of the complementary field pair. This decision is based on the POC values of these three pictures: First, the absolute difference of the frame's POC value to the POC values of the two fields of `L1[0]` is calculated. Then, `colPic` is set to the field which has the lower absolute difference to the field's POC. Table 8-6 in the H.264 standard contains the definition of this rule.

The next change for PAFF interlacing concerns the position of the co-located macroblock in the reference picture `mbAddrCol` and the vertical sample position in this co-located macroblock, which is called `yCol` in the standard document. If frames and fields are mixed in the Inter-prediction process, for example when a field references a field of a frame in the DPB, the macroblock number and the sample positions have to be adjusted to point to the corresponding position in the reference picture. Table 8-8 in the H.264 standard contains the formulas which need to be used to calculate the correct position `yM` in these cases. If both the current picture and `colPic` are frames or both are fields, `yM` is the same as `yCol`.

The third change in the Inter-prediction process involves the motion vectors for the chroma components `MVC`. These are usually identical to the motion vectors for the luma component `MV`. However, when both the current picture and the reference picture are fields and their parities differ, the vertical component of the chroma motion vectors `MVC(y)` are different from those of the luma motion vectors `MV(y)`. If the current field is a top field and the reference picture field is a bottom field, then the vertical component of the chroma motion vector `MVC(y)` is `MV(y)-2`. In the other case, in which is current picture is a bottom field and the reference picture is a top field, `MVC(y)` is `MV(y)+2`.

6.6.2 Implementation

The `recon_ppu` module contains the reference picture selection and thus the assignment of `colPic` as well as the calculation of the motion vectors in the picture. The new function `determine_colPic_and_scale` compares the structure of the current picture and `L1[0]` and returns both the `colPic` value and the scaling factor, which can be `ONE_TO_ONE`, `FRM_TO_FLD`, or `FLD_TO_FRM`.

Because of the implementation of the Decoded Picture Buffer which stores both the frame and the fields for each picture in the video stream, an implementation of the rules to determine `mbAddrCol` and `yM` is not necessary when accessing the picture data, which is done in the reconstructor module. The co-located macroblock is always in the same position as the current macroblock in the picture that is being decoded at the time. However, the picture information structures for frames are not merged and the structures for fields are not split. So it is necessary to implement these rules in order to address the correct co-located submacroblock information when determining the

motion vectors in the recon_ppu module. For this reason, I have implemented the determine_mbAddrCol_and_yM function, which, depending on the scaling factor mentioned above, determines the values for mbAddrCol and yM. They are then used in the functions that calculate the motion vectors for the different motion compensation methods.

The implementation of the chroma motion vectors is done in the recon_decode_motion_vectors function. Since the motion vectors are decoded entirely in the recon_ppu module, while the actual motion compensation process is realized on the reconstructor SPU, the chroma motion vectors need to be stored in a data structure that is accessible in both modules. The data type ppu_mb_info_t is suitable for this, because it is stored for each macroblock and contains information that is only relevant to these two modules, such as the motion compensation data. This structure still offers enough space for the vertical elements of the chroma motion vectors of the macroblock, so I added these to this structure. In the reconstructor module, the code that loads the image data from the reference picture needed to be changed. The reconstructor uses a preloading mechanism that loads this data into the local storage while it is processing other parts of the picture. This feature is realized in the prefetch_ref function and the prefetch_mb macro. These functions store the preloaded data in a set of global data structures. In order to support chroma motion vectors, I have combined these variables into the mb_cache_t structure and created an instance for both the luma and chroma cache. Then, prefetch_ref is called twice, once for the luma motion vectors and once for the chroma motion vectors. This results in two different buffers that contain the image data to which the two different motion vectors point. Finally, the functions that perform the motion compensation process for the luma or chroma elements are called with the appropriate cache as an argument.

6.7 Deblocking filter process (8.7.2.1)

6.7.1 Description

The strongest filtering strength (boundary filtering strength bS = 4) is not used for horizontal macroblock edges in fields. This is because field rows are spatially twice as far apart as frame rows and thus cover a larger area.

6.7.2 Implementation

These features were already implemented in the existing code. The boundary strength is calculated in the get_boundary_strength functions. These require a parameter pic_is_frame, which can take the same values as the slice_structure variable (i.e. FRAME, TOP_FIELD, BOTTOM_FIELD). The function filter_proc initializes this variable at the beginning of the slice filtering process.

The filter module also already contained the code for MBAFF support. However, the support is not fully complete, since the functions deblock_macroblock_chroma_core do not fully implement the standard. A developer of the Chinese team that developed Aeolus marked these problems with comments.

Chapter 7

Conclusions

7.1 Results

In this thesis, I have presented an implementation of the Picture-Adaptive Frame/Field (PAFF) interlacing method into an existing H.264 decoder for the Cell processor, called Aeolus. Experiments with a variety of test streams, which I created with the JM reference encoder and downloaded from different sources on the Internet, showed that the decoder is now fully able to decode video streams that were encoded with the PAFF interlacing method. The output of the new Aeolus and the JM decoder is identical for the same video streams, which indicates that the implementation of PAFF is complete and correct.

Benchmarks on a Cell blade show that the code with the PAFF-support has the same performance as the original code. It is still able to decode a video in Full-HD resolution in real-time on three SPUs with a similar PPU load compared to the original decoder. When it is run on the Playstation3, however, the execution of the new decoder is at least ten times slower than the original code. The reasons for this behavior are unclear, but tests gave strong evidence that the performance decrease is caused by the new implementation of the Decoded Picture Buffer with the dynamic memory allocation. This is the case even though the code uses less memory than the original implementation and virtual memory was deactivated during these tests. However, it should be noted that the Linux kernel for the Playstation3 is still under constant development and tests showed that using different versions of the kernel greatly influences the performance of the Decoded Picture Buffer. Changing the memory management from the dynamic allocation back to an allocation of the buffers at the start of the program may speed up the decoding process, at the cost of higher memory requirements.

The current version of Aeolus is, unfortunately, still not able to decode the video streams from the HDTV-stations Sat.1, Pro7, or the Astra demo channel. The reason for this, however, is not in the implementation of PAFF-interlacing, but because these channels use the H.264 features Adaptive Memory Management and Reference Picture List Reordering to encode the video. These features are not supported by the Aeolus decoder at this point.

In conclusion, the decoder can now flawlessly decode video streams with PAFF interlacing, as long as they do not utilize features of the H.264 standard that are not supported by the current implementation of Aeolus.

7.2 Future work

In order to decode the German HDTV channels, the features *Adaptive Memory Management* and *Reference Picture List Reordering* need to be implemented. The new implementation of the Decoded Picture Buffer was designed with this requirement in mind and I have already added the code for reading the syntax elements to the pre-parser modules.

The British channel SkyHD uses the *Macroblock-Adaptive Frame/Field* (MBAFF) interlacing method, which is also not supported by Aeolus. It is also part of the Main profile of H.264, so it is possible that the German TV channels could start using this method in the future as well. Also, *Weighted Prediction*, though apparently not utilized by the European HDTV channels, is part of the Main profile and not supported by the Aeolus decoder. Finally, the current implementation only supports a maximum of eight reference pictures, although throughout the decoder, memory for up to fifteen reference pictures is reserved. This is due to an optimization in the reconstructor module. Although commercial decoders usually only use very short reference picture lists, changing this behavior of Aeolus would make it more robust to future developments in the commercial encoders.

When the satellite receiver is affected by bad weather or poor reception quality, its output data is not always flawless. Aeolus currently does not support any error concealment and inconsistent input data may even crash the decoder. For usage beyond the prototype stage, an implementation of error concealment methods would thus be necessary.

Since the team at IBM Beijing was under time pressure when developing the original Aeolus decoder, not all parts of the code could be realized in the best way possible. For future work on the decoder, it would be beneficial to merge the pre-parser code of the CAVLC and CABAC parser module, since this code is virtually identical and any change has to applied to both modules. In order to make the code less vulnerable when changes are applied by future developers, the DMA accesses with fixed offsets to parts of memory structures, which are used in various areas of the decoder, should be changed. A possible option for this would be to use global constants with the offset values in `param.h` and to split the data structure definitions, so that it will not be necessary anymore to apply changes to the structures several times in different areas of the decoder. I have given an example for a data structure that is defined several times in the decoder in section 5.1.

Although the current implementation of PAFF interlacing is efficient in terms of processing time, it uses about twice as much memory as an optimal implementation would. This is because the Decoded Picture Buffer contains every reference picture as a frame and its two fields. It would be possible to access the respective field directly in the frame by simply accessing only every even or every odd line. This way, the memory requirement by the Decoded Picture Buffer could be cut in half. However, this requires a vast amount of changes in the highly optimized reconstructor module, which uses sophisticated pre-loading algorithms using DMA transfers. Further research is necessary to determine whether this implementation would still be sufficiently efficient.

Another way to optimize the current implementation of PAFF interlacing would be to improve the pre-loading mechanisms in the reconstructor module for the luma and chroma elements of the reference pictures. The current implementation loads the complete macroblocks from the main memory to which the luma and chroma motion vectors point. It would be sufficient to only load and store the luma or chroma elements. This would reduce the amount of data that needs to be transferred from the main memory and the amount of data needed in the local storage of the reconstructor SPU.

List of Figures

Nomenclature

AVC	Advanced Video Coding (H.264)
CABAC	Context-Based Adaptive Binary Arithmetic Coding
CAVLC	Context-Adaptive Variable Length Coding
CBE	Cell Broadband Engine
DCT	Discrete Cosine Transformation
DMA	Direct Memory Access
DPB	Decoded Picture Buffer
DVB	Digital Video Broadcasting
fps	frames per second
FPU	Floating-Point Unit
GCC	GNU C Compiler
GDB	GNU Debugger
GNU	GNU's not Unix
HDTV	High-definition Television
IDE	Integrated Development Environment
IDR	Instantaneous Decoder Refresh
JM	H.264/MPEG-4 AVC Reference Software
JPEG	Joint Photographic Experts Group
LS	Local Storage
MBAFF	Macroblock-Adaptive Frame/Field coding
MCP	Motion-Compensated Prediction
ME	Motion Estimation
MFC	Memory Flow Controller
MPEG	Moving Pictures Experts Group
MV	Motion Vector
MVC	Motion Vector Chroma Components
MVP	Motion Vector Prediction
NAL	Network Abstraction Layer
NMM	Network-Integrated Multimedia Middleware
NTSC	National Television System Committee
OS	Operating System
PAFF	Picture-Adaptive Frame/Field coding
PAL	Phase Alternating Line
PCM	Pulse-Code Modulation
POC	Picture Order Count
PPE	PowerPC Processor Element
PPS	Picture Parameter Set
PPU	PowerPC Processor Unit
PSP	Sony Playstation Portable

RISC	Reduced Instruction Set Computer
SDK	Software Development Kit
SDTV	Standard definition Television (PAL or NTSC)
SIMD	Single-Instruction / Multiple Data
SNR	Signal-to-Noise ratio
SPE	Synergistic Processor Element
SPS	Sequence Parameter Set
VCL	Video Coding Layer
XLC	IBM XL C Compiler

Bibliography

[1] ASTRA DEUTSCHLAND GMBH: *Programmliste ASTRA 19,2 Grad Ost - Digitales Fernsehen*. http://www.ses-astra.com/. Version: March 2007

[2] FIEDLER, Martin: *Videokompressionsverfahren: von MPEG-1 bis DivX, VC-1 und H.264*. http://keyj.s2000.ws/files/projects/videocomp.pdf. Version: October 2006

[3] IBM CORPORATION: *Cell Broadband Engine Programming Handbook*. Version 1.0, April 2006

[4] IBM CORPORATION: *Cell Broadband Engine Programming Tutorial*. Version 2.1, March 2007

[5] IBM QUASAR DEVELOPMENT: *H.264 High Definition Decoder*. Cell SW Tech Series, June 2007

[6] IBM SYSTEMS AND TECHNOLOGY GROUP: *Cell Ecosystem Solutions Enablement*. Course Code: L3T2H1-39, June 2006

[7] ITU-T: *Recommendation H.264: Advanced video coding for generic audiovisual services*. March 2005

[8] KAHLE, James ; DAY, Michael ; HOFSTEE, Peter ; JOHNS, Charles ; MAEURER, Theodore ; SHIPPY, David: Introduction to the Cell multiprocessor. In: *IBM J. Research and Development* 49 (2005), July / September, Nr. 4/5

[9] LOHSE, Marco: *Network-Integrated Multimedia Middleware, Services, and Applications*, Department of Computer Science, Saarland University, Germany, Diss., June 2005

[10] MALVAR, Henrique S. ; HALLAPURO, Antti ; KARCZEWICZ, Marta ; KEROFSKY, Louis: Low-Complexity Transform and Quantization in H.264/AVC. In: *IEEE Transactions on Circuits and Systems for Video Technology* 13 (2003), Nr. 7, S. 598–603

[11] MARPE, Detlev ; SCHWARZ, Heiko ; WIEGAND, Thomas: Context-Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. In: *IEEE Transactions on Circuits and Systems for Video Technology* 13 (2003), Nr. 7

[12] PROSIEBENSAT.1 PRODUKTION GMBH: *HDTV: FAQ*. http://www.prosieben.de/spielfilm_serie/hdtv/faq/. Version: February 2007

[13] RICHARDSON, Iain E. G.: *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley and Sons Ltd, 2003. – ISBN 0–470–84837–5

[14] SCHÄFER, Ralf ; WIEGAND, Thomas ; SCHWARZ, Heiko: The emerging H.264 / AVC standard. In: *EBU Technical Review* (2003), January

[15] SÜHRING, Karsten ; SCHWARZ, Heiko ; WIEGAND, Thomas: Effizienter kodieren: Details zum kommenden Videostandard H.264/AVC. In: *c't: Magazin für Computertechnik* (2003), Nr. 6, S. 266–273

[16] STILLER, Andreas: Zelluläre Strukturen: Die Architektur der Cell Broadband Engine. In: *c't: Magazin für Computertechnik* (2007), Nr. 12, S. 196–201

[17] STRUTZ, Tilo: *Bilddatenkompression*. 3. Auflage. Vieweg Praxiswissen, 2005. – ISBN 3–528–23922–0

[18] SULLIVAN, Gary J. ; WIEGAND, Thomas: Video Compression: From Concepts to the H.264/AVC Standard. In: *Proceedings of the IEEE* Bd. 93, 2005, S. 18–31

[19] WIEGAND, Thomas ; SULLIVAN, Gary J. ; BJNTEGAARD, Gisle ; LUTHRA, Ajay: Overview of the H.264/AVC video coding standard. In: *IEEE Transactions on Circuits and Systems for Video Technology* 13 (2003), Nr. 7, S. 560–576

[20] WIKIPEDIA: *High-definition television*. `http://en.wikipedia.org/wiki/Hdtv`. Version: March 2007