Diploma Thesis

# Slice-Level Trading of Quality and Performance in Decoding H.264 Video

Michael Roitzsch <mroi@os.inf.tu-dresden.de>
Operating Systems Group
Computer Science Department
Technische Universität Dresden

June 7, 2006

tutored by Dipl. Inf. Martin Pohlack,
department headed by Prof. Dr. rer. nat. Hermann Härtig

## Acknowledgments

I want to thank Prof. Dr. Hermann Härtig for assembling an outstanding research group and creating a collaborative working environment that is both productive and pleasant to work in. This environment and the dedication of the staff members have been a strong and supporting foundation for my studies and have helped to make this diploma thesis possible. Among the staff members, I am especially thankful to Dipl. Inf. Martin Pohlack, who has tutored my diploma thesis and who has been a reliable source of useful suggestions and helpful comments. He encouraged me to find and follow my own working style while still providing the guidance I needed to get this work on the right tracks. I also thank Adam Lackorzyński for satisfying the hardware needs and Dr. rer. nat. Claude-Joachim Hamann for his help on a mathematical question. Furthermore I want to express my gratitude and respect to the entire team of the xine media player as I own a lot of my experience to my collaboration with them. Finally, I thank my friends who participated in a verification test and my parents for all their enduring support.

When a demanding video decoding task requires more CPU resources then available, playback degrades ungracefully today: The decoder skips frames selected arbitrarily or by simple heuristics, which is noticed by the viewer as jerky motion in the good case or as images completely breaking up in the bad case. The latter can happen due to missing reference frames. This thesis provides a way to schedule individual decoding tasks based on a cost for performance trade. Therefore, I will present a way to preprocess a video, generating estimates for the cost in terms of execution time and the performance in terms of perceived visual quality. The granularity of the scheduling decision is a single slice, which leads to a much more fine-grained approach than dealing with entire frames. Together with an actual scheduler implementation that uses the generated estimates, this work allows for higher perceived quality video playback in case of CPU overload.

# Contents

# List of Figures

# 1 Introduction

## 1.1 Motivation

Modern video coding standards achieve incredible compression efficiency: datarates around 0.1 bits per pixel are enough to satisfy even high quality needs. Unfortunately, such state-of-the-art compression comes at a price: The system recommendations for Apple's high definition gallery [1] list only the latest dual-core processors for full 1080p playback at 24 frames per second. But the computing demands fluctuate heavily over time during video playback. Therefore those immense peak requirements are caused by only a minuscule fraction of the total decoding process. However, it is difficult to relax these requirements as current video players cannot cope with resource shortage in a graceful way. What usually happens is that the decoding of a frame does not finish in time and it is discarded. But current video standards do not easily allow to skip a frame's decoding, so subsequent frames will also be late and the playback stalls. The user perceives this as jerky motion. If the decoder skips the decoding of a frame, ignoring the frame interdependencies, future frames cannot be decoded correctly and the image will totally break up.

What is needed is a way to deal with resource shortages more gracefully. This would allow to reduce the current resource overprovisioning, which is especially interesting in embedded and power-aware systems, but also in video software like players or compositing and editing systems, where the handling of multiple high definition streams is common.

## 1.2 Vision

The model of a video is a three-dimensional function in two spatial and the temporal direction. Initially, this function is continuous along all three axes, but is typically discretized along the time axis into frames and along the spatial axes into pixels. A modern video coding algorithm will exploit both spatial and temporal redundancy in this representation to reduce the amount of data. Further compression is achieved by approximating the video function instead of matching it exactly. This yields a good compression ratio, but recreating the approximated video function from the resulting bitstream is expensive. If this is to be done with lower resource usage, some deficiencies in the final video function will have to be tolerated. The obvious candidate is the quality of the approximation. Hence, the goal of this thesis is to provide a way to trade resource usage against the perceived difference of the recreated video to its original.

I therefore provide a way to skip the decoding of certain parts of the bitstream. These outages would lead to parts of the video function being undefined, so these parts are synthesized differently. The key problem is to divide the video into partitions and then judge, which of these parts are visually important. This importance should convey, how much the user would notice or object, if the given partition is not fully decoded but synthesized otherwise. This synthesis should be visually close to the original content, but should consume less resources.

The synthesis of a video partition is done by filling it with video content from previously decoded partitions. This brings up the challenge of selecting visually similar replacement content. The remaining perceptional difference between the original video and the replacement has to be quantified. This difference also affects the decoding of future video partitions, as current decoding standards exploit inter-partition redundancy to increase coding efficiency. This affection has to be quantified as well. All the data necessary for these steps is gathered by preprocessing the video and is stored alongside the video bitstream as an additional sideband channel.

Finally, the perceptional relevance value can be used by a decoding scheduler to admit partitions for decoding or to select them for skipping, if short on resources. The entire approach

should be modular, with video partitioning, partition replacement, quality loss quantification, and scheduling being largely independent to allow for easy incorporation of future research results on these individual topics.

## 1.3 Outline

I begin with a coverage of related work on resource usage prediction (Section 2.2), perceptional classifications for video (Section 2.3) and on quality loss algorithms (Section 2.4). This also includes an introduction into the chosen decoding algorithm (Section 2.1). I continue by explaining, how the video preprocessor partitions the video and selects content to replace the partitions (Section 3.2) and how it handles error propagation into future partitions (Section 3.3). The implementation of the preprocessor is described in Section 4.1, followed by the sideband data format (Section 4.2), and the design of the scheduler (Section 4.3). A thorough evaluation (Sections 5.1 and 5.2), followed by an outlook into future work (Section 5.4) concludes the thesis.

# 2 Related Work

My thesis is tied to current technologies in video coding and draws from a variety of previous research results. In this section, I portray the video decoding algorithm used, summarize previous work on decoding time prediction, provide an overview of existing approaches to exploit perceptional importance, and I compare methods to calculate the visual difference of two images.

## 2.1 H.264

The video decoding algorithm of choice for this work is H.264 [2]. The most important reason is that amidst other emerging technologies like VC-1 [3] or VP7 [4], H.264 is established as one of the most important modern coding standards, as proven by the fact that both upcoming high definition DVD successors, the Blu-ray [5] and HD-DVD [6] associations have elected H.264 as mandatory technology for their players. H.264's popularity grew even further, when Apple's latest incarnation of the iPod portable music device featured H.264 based video playback on the go [7]. Production quality H.264 decoder and encoder software is also available from Apple as part of their QuickTime media architecture [8].

Apart from industry interests, H.264 is also interesting for the technology itself. The standard targets the entire scale of video playback from low bitrate, low resolution handheld usage as in DVB-H [9] over video conferencing and video on demand applications [10] up to high definition broadcast quality content distribution [11]. It offers the same quality as MPEG-2 with about half the bitrate [12] and it also includes a lossless mode for archiving needs [13].

### 2.1.1 H.264 Improvements

This increased efficiency and wide scalability stems from a number of improvements compared to previous algorithms, the major ones of which I will name here. A more thorough walkthrough can be found in [14]. The reader is expected to be familiar with basic video coding terminology. Those not comfortable with the subject are invited to refer to [15] for some introductory reading.

**In-Loop Deblocking:** One of the most notable changes, which also raised discussion among experts regarding its usefulness [16, 17], is the integrated deblocking filter, which is used to smooth fake edges introduced by coding artifacts. This filter is mandatory because it is in-loop, meaning that the filter is applied before the image is kept for reference. If this step was skipped, the reference images intended by the encoder and the ones used by the decoder would drift apart. This behavior is different from previous post processing filters, which were applied after the frame was ready for referencing.

**Enhanced Entropy Coding:** The combination of Huffman and run length encoding used previously has been replaced with two alternative entropy coding methods: The variable length coding is similar to the previous approach. A new binary arithmetic coding accounts for a large share of the bitrate reduction achieved with H.264 [18]. Both coding methods are context adaptive.

**Network Abstraction Layer:** The H.264 standard includes a packaging layer that allows easy integration into various carrier media. It is also possible to partition the data into more important and less important syntax elements, which can be transmitted independently.

**Flexible Slice and Macroblock Structure:** Coding options previously assigned on a frame level by using frame types (I-frame, P-frame, B-frame) are now assigned on a slice level. The slice structure is much less rigid than in previous standards. With flexible macroblock or-

dering (FMO), slice groups can now cover arbitrary regions of a frame. Arbitrary slice ordering (ASO) allows to transmit the slices of one frame in any order.

**Adaptive Interlacing:** With adaptive frame/field coding (AFF), the encoder can select interlaced or progressive coding for each macroblock individually. The bitstream stores the macroblocks in a different order, so that pairs of vertically adjacent macroblocks are combined in the resulting image either by placing them in their natural order for progressively coded areas or by interweaving their lines for interlaced areas.

**Switching Pictures:** The new SI (switching intra) and SP (switching predictive) slice types can be used to encode switching frames that allow the decoder to switch seamlessly between different representations of the same video, like parallel live streams of different quality and complexity.

**Motion compensation:** H.264 is more flexible when assigning motion vectors to image areas. Blocks can be as large as 16×16 and as small as 4×4 pixels. Motion vectors offer quarter pixel accuracy, which was first available in the MPEG-4 Part 2 coding standard [19], but the interpolation scheme has been improved to better preserve the sharpness of the image. Global motion compensation was used in MPEG-4 Part 2 as a completely separate prediction mode. This has been integrated into the normal motion compensation step as an efficient way to store otherwise unchanged consecutive macroblocks with identical motion.

**Weighted Prediction:** Fades and flashes can be coded more efficiently, because the motion predicted image component can be scaled and offset before it is applied to the final frame.

**Deep Referencing:** Unlike the limitation to two frames in previous standards, H.264 allows motion compensation to choose freely among up to 32 reference frames. The display order of the frames is largely decoupled from the referencing. In addition, any frame can be used as a reference frame now, whilst in previous standards, images using certain coding methods (like the bi-predictive encoding used in B-frames) were not allowed as references for other frames.

**Spatial Prediction:** Intracoded regions can be predicted by extrapolating image features of already decoded neighboring parts of the frame. Unlike the intra prediction found in MPEG-4 Part 2, this prediction is not done in the frequency domain, but in the spatial domain.

**Hierarchical Block Transform:** H.264 does not use the up to then common 8×8 IDCT any more. Instead, it uses a 4×4 integer transform, which can be implemented without rounding errors using only 16 bit registers. The transform's low frequency components can be enlarged to 16×16 pixels areas, which will receive the high frequency components from the smaller subblocks superimposed.

### 2.1.2 H.264 Profiles

Video coding algorithms commonly specify the abilities required at the decoder by defining profiles and levels. Profiles restrict the set of coding options and tools a video stream is allowed to use, whereas levels restrict parameters of the bitstream.

Previous video coding standards usually had totally ordered profiles (so that with any pair of profiles, one would be a true subset of the other). This does not apply to H.264, whose three profiles have been designed more use-case oriented (see Part V in [14]):

- The **Baseline Profile** is intended for conversational services like video chats.

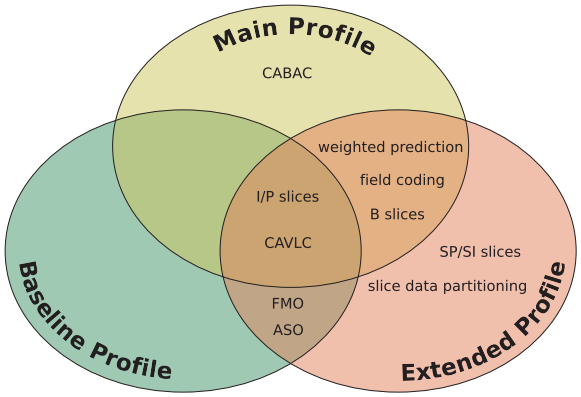



Figure 1: H.264 profiles

- Pre-encoded video for distribution on pre-recorded media or for broadcast purposes is targeted by the **Main Profile**.
- The **Extended Profile** is designed for high quality video streaming.

The features included in the various profiles can be seen in Figure 1. I will primarily focus on the Main Profile, but will make comments and suggestions regarding the features of the other profiles later in this thesis, mainly in Sections 3.1.1 and 3.2.1.

Levels in H.264 are completely orthogonal to profiles. The standard defines fifteen levels, which specify upper limits for parameters such as picture size, macroblock processing rate, overall video bitrate, and buffer sizes.

Decoder implementations can choose to support only some profiles, so they do not have to implement the entire feature set of the standard; or they can choose to support only some levels, so they can make assumptions on maximum CPU or memory usage derived from the constraints of the level.

### 2.1.3 H.264 Challenges for Selective Decoding

The ability to degrade playback gracefully in resource constrained situations is based on the ability to skip decoding of parts of the bitstream, selected in a way to minimize the visual impact. But H.264 poses new challenges to this task, which prevents the majority of earlier research results from being reused as is.

- The coding type is now assigned per slice instead of per frame.
  This alone makes it difficult to apply previous results that make use of the coding type, because the assumption of the type being constant throughout the frame no longer holds.
- Every frame can be used as a reference frame.
  In previous standards, the knowledge that a B-frame would never be used as a reference frame could be exploited easily to skip their decoding. This use of B-frames is no longer possible.
- Frames that inhibit all error propagation are far apart.
  Previous standards use I-frames to completely reset the decoder. But an I-slice in H.264 does not reset the decoder, because it may appear together with a B- or P-slice in the same frame. An I-slice not even completely resets the area of the frame it covers, because spatial prediction might use neighboring pixels from previously decoded non-I slices of the same frame. To fully reset the decoder's state, a frame must consist entirely of I-slices. Such a frame is tagged in the bitstream and is called an instantaneous decoder refresh frame (IDR-frame).

These circumstances support the necessity to conduct new research on the subject or at least to reevaluate existing work, which I will do more thoroughly in Section 2.3.

## 2.2 Decoding Time Prediction

As I want to trade decoding quality for decoding performance, we need to know the decoding

cost with fine granularity. The key resources used during video decoding are CPU and memory. But while the memory requirement is fixed and can be obtained easily from the frame size of the video, the CPU resource is more challenging. It varies greatly with playback time and is not trivially calculated. In addition, memory resources of today's desktop machines are usually sufficient to handle the decoding requirements, whereas the CPU is often not capable of coping with the load of a high definition video stream. For this reason, good estimates of CPU resource usage, quantified as decoding times, are vital for this work. They provide one half of the information required for the scheduling decision discussed later.

Previous work exists in the research area of decoding time prediction, but I will base my thesis on results obtained by myself [20]: I developed a method to obtain good estimates of execution times for the decoding of individual video frames of MPEG-1/2 and MPEG-4 Part 2 video. The remaining part of this section will provide a summary of my work. Those readers already familiar with this are invited to skip to Section 2.3.

My overall idea is to find a vector of metrics extractable from the bitstream for each frame. This vector is scalar multiplied with a vector of fixed coefficients to estimate the decoding time. I started with an execution model fit for recent decoder algorithms. This model can be seen in Figure 2. Mapping the decoding steps of the actual algorithms to the steps of this model, I had a simple way to break down a decoding algorithm into small subtasks. I then took an implementation for the decoding algorithm and rigged it with time sampling code to get execution times for the various subtasks.

For each of these subtasks, I searched for a metric that would provide a good linear fit with the execution time of this subtask. The choice of metrics was limited by the constraint that the values had to be extractable from the bitstream without fully decoding it. But it became clear that the subtasks were small and simple enough so that straightforward metrics like macroblock count provided quite accurate matches with execution time.

I used numerical algorithms, namely a linear least square problem solver, to calculate the coefficient vector that would, given the metrics vector for a frame, estimate the decoding time for that frame with the smallest error. I enhanced the linear least square solver to avoid negative coefficients and to provide numerically stable results.

Evaluation has shown that a coefficient vector obtained once with a set of training videos would apply to a large range of other videos with good estimation results. An example prediction plot can be found in Figure 3.

Compared to other approaches, my method has several advantages: Unlike Altenbernd, Burchard, and Stappert in [21], I do not need to modify the decoder itself, a preprocessor independent of the actual decoder to extract the metrics on the fly is sufficient. It is also not required to do detailed source code analysis, which would become outdated as decoder implementations improve. Also, the predictor has to be trained only once for a given machine to learn the necessary coefficient vector. Once this has been done, predictions can be made for other videos without any further calibration. These properties make this method well suited for this thesis.

However, the preceding findings are limited to MPEG-1/2 and MPEG-4 Part 2 video. It remains to be seen, how the technique can be adapted to predict the decoding times for H.264 video with all its coding features. In addition, the prediction needs to be broken down to work on a level smaller than frames, because I want to handle more fine grained video partitions. I discussed in [20] already, how H.264 can fit into the presented decoder model and I will present the final results for H.264 decoding time prediction in Section 3.1.

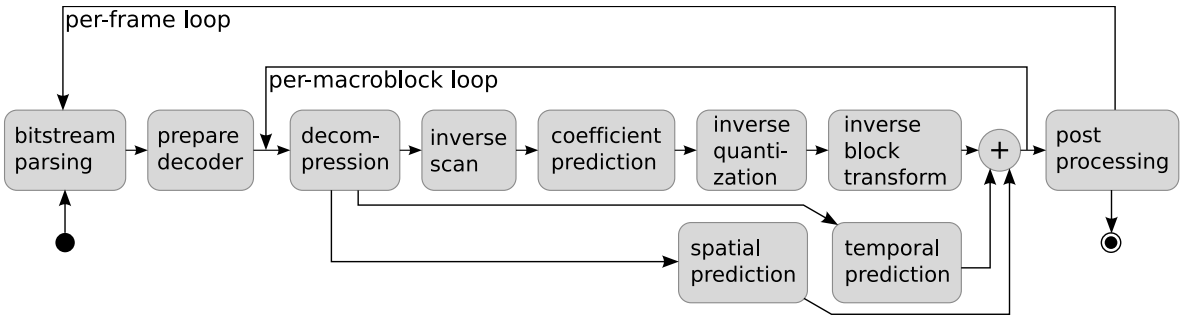



Figure 2: Decoder execution model as developed in [20]

## 2.3 Perceptional Importance

Earlier video coding standards, but also H.264 have already been subject to research on exploiting perceptional importance of video partitions. The objective is often to improve network streaming behavior in packet loss situations. I introduce model-based approaches, analysis-by-synthesis approaches and hybrid approaches, which combine ideas of the former two.

### 2.3.1 Model-based Approaches

Previous work exists that uses bitstream syntax to varying degrees to estimate the visual importance of video frames. Isović and Fohler proposed the quality aware frame selection algorithm (QAFS) in [22], which prioritizes frames of a group of pictures (GOP) according to an estimated visual importance index. The key criteria for this index are the frame type and the position of the frame within the GOP. As QAFS is targeted for MPEG-2, it assigns the lowest importance to B-frames, because these frames are never used as references. But as this assertion is no longer true for H.264 (see Section 2.1.3), the approach seems not beneficial. In addition, the importance index is only a relative ordering amongst the frames of one GOP, I require an absolute measure for all frames. However, the paper also shows a decision algorithm that uses the importance index to select the frames to skip if deadlines are missed. The slice scheduler I will describe in Section 4.3 is similar to this algorithm.

In [23], Zhang, Regunathan, and Rose describe a method to estimate the overall degradation caused by packet losses at the decoder, including distortion introduced by spatial and temporal prediction. The recursive optimal per-pixel estimate (ROPE) is calculated through precise accounting for the error of each pixel. It traverses all the different paths that might contribute an error caused by a degraded decoder state. The paper presents the calculations for the H.263 video compression algorithm, but simplifies by not handling subpixel motion compensation. The accuracy by which the algorithm estimates errors caused by losses in the video stream is impressive, but the required computation is difficult and it would need to be adapted to H.264 which is much more complicated than H.263. The in-loop deblocking would be hard to integrate into ROPE. And even if that could be done, the entire calculation is intrinsically tied to mean squared error as the distortion metric, which I will argue in the next section as being a bad choice for measuring visual importance. Changing the underlying metrics would require recreating ROPE from scratch, which seems too complicated.

### 2.3.2 Analysis-by-Synthesis Approaches

In contrast to the previous methods, the importance of parts of the bitstream can be derived with varying granularity by decoding the

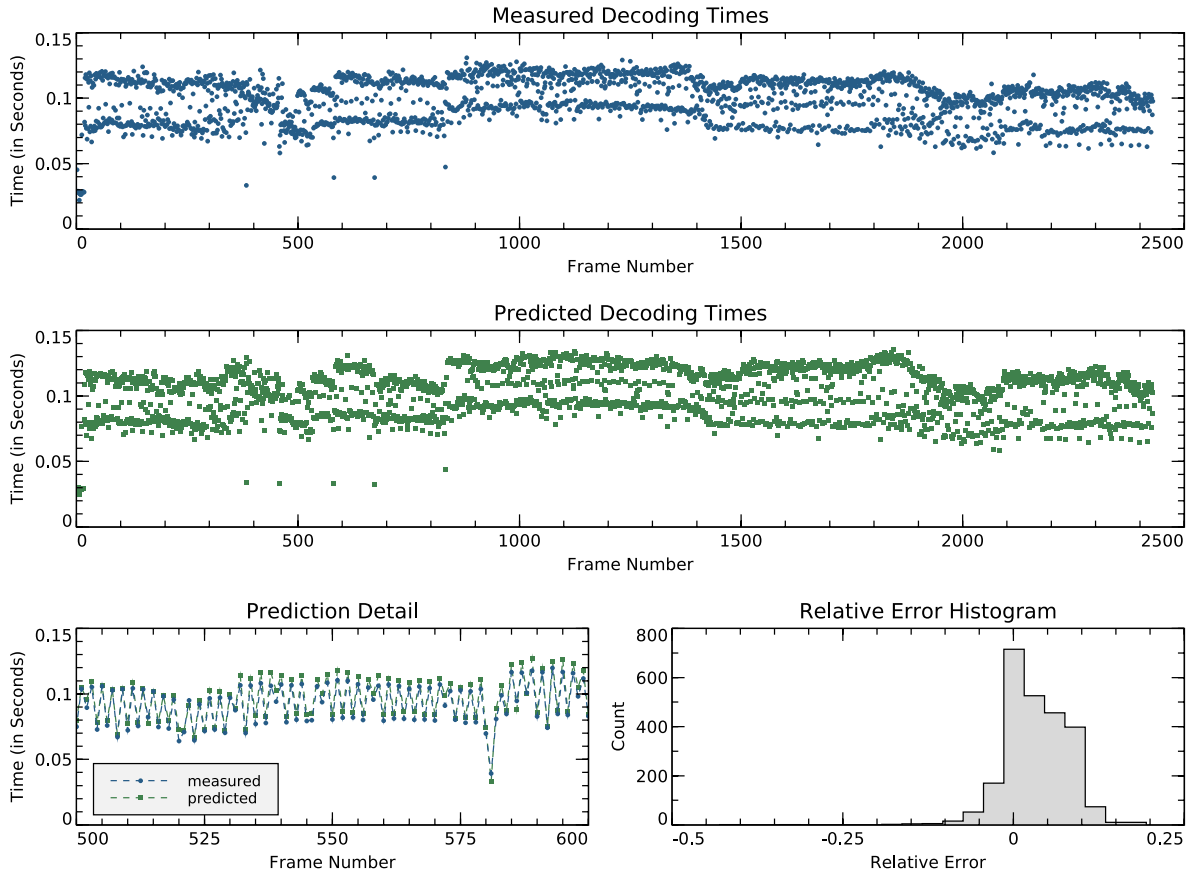



Figure 3: Decoding time prediction for a MPEG-4 Part 2 high definition movie. See [20] for details.

bitstream with and without the part in question and quantifying the resulting differences. These methods are usually called analysis-by-synthesis approaches.

Masala, Quaglia, and De Martin present such a method in [24]. For MPEG-2 video, they classify the visual importance of each macroblock individually by replacing it with the area at the same position in the previous image and measuring the differences using mean squared error. Although this method is fine grained, its problems are the simplistic error concealment and the use of mean squared error, which is not a good model of human perception. Even if those problems were fixed, this method is entirely unable to account for distortion caused by error propagation. As discussed in Section 2.1.3, this is one of the major challenges with H.264, as it can preserve errors much longer than previous coding standards.

De Vito, Farinetti, and De Martin then enhance the previous approach in [25] by measuring the difference caused by a lost macroblock not only for the current frame, but for all potentially degraded subsequent frames of that GOP. This GOP-level distortion is considerably more expensive to compute than frame-level distortion, because for each dropped and concealed macroblock, all frames until the end of the GOP have to be decoded.

Their goal is to find those macroblocks most important for the visual quality to optimize network streaming behavior in the presence of packet losses. They compare the behavior using both their GOP-level distortion calculation and frame-level distortion calculation similar to [24] and come to the conclusion that GOP-level cal-

culation does not provide enough gain to justify the heavy computational burden. This result is based on MPEG-2, so I doubt this conclusion is still valid for H.264. However, in Section 5.2, I directly compare my final scheduling method with an H.264-adaptation of frame-level distortion calculation to prove my doubts.

Another interesting result of this paper is the way network bandwidth is allocated for those macroblocks classified to be more important: The authors compare a frame-level allocation scheme with a GOP-level allocation scheme and find the latter to be superior. In Section 4.3.2, where I present my CPU allocation scheme, I also argue that allocating processing power works better, when data on more frames than just the current one is available.

Bucciol, Masala, and De Martin present a method tailored for H.264 video streaming in [26]. Their approach classifies the importance not per macroblock, but on a packet-level. Each packet is dropped and concealed, the resulting bitstream decoded and compared to the undegraded video in a pure analysis-by-synthesis approach. The paper admits that the method gets more expensive the more prediction is used in H.264, as for each packet, the entire sequence has to be decoded until the propagated error has diminished sufficiently. The authors suggest to remedy this by precomputing and storing the results, which resembles my preprocessing idea. They combine the derived importance value with expected deadlines to a single characterizing value per packet. I pursue a similar idea in 4.3.1.

However, an open problem is error accumulation: If a packet is dropped and concealment applied by the decoder, it is unclear how errors propagate if the material used for concealment is already degraded. The authors briefly mention the problem, but claim that this situation is unlikely. However, as H.264 can preserve decoding errors for a long time, I do not want to rely on this claim in my context. But accounting for error accumulation with an analysis-by-synthesis approach is entirely impractical, because the computational burden explodes: As each packet dropping pattern of $N$ packets results in a different error propagation behavior, $2^N$ such patterns would have to be analyzed.

### 2.3.3 Hybrid Approaches

The most promising idea is to combine the analysis-by-synthesis and the model-based approaches to get the elegance and good results of analysis-by-synthesis with the lower computational cost of model-based methods.

Extending on an idea already started in [25], De Vito, Quaglia, and De Martin present such a hybrid approach in [27] for H.264: The distortion introduced by a loss event is divided into distortion in the current frame and distortion in future frames through error propagation. The former is again quantified using analysis-by-synthesis, whereas the latter is estimated using an error propagation model. I will make exactly the same distinction and will discuss both errors in Section 3.2.5 and 3.3.2, respectively. However, the authors use a rather simplistic propagation model, which essentially estimates the error of all future frames to be equal to the error of the current frame until a decoder refresh takes place. Additionally, the paper works with H.264 video, but uses the terms GOP and I-frame, which have no clear meaning in H.264, because it does not operate with GOPs and assigns coding types on the slice level and not on frame level. The paper further states that macroblocks are never referenced when they belong to a B-frame, which is wrong. Therefore, although reusing ideas of their work, I redesign the propagation model more thoroughly in Section 3.3, taking the exact H.264 semantic into account. But nevertheless, I compare the performance of my final scheduler with a derivative of their propagation model in Section 5.2.

## 2.4 Video Quality Loss Metrics

As seen in the previous section, a building block for quantifying the quality degradation when

skipping parts of the decoding is the quantification of differences between the original video and a degraded version. The basic problem is to reduce two different, but similar sections of video to a number that correlates with the decoding error the user sees. The decoding scheduler I introduce later makes its skipping decisions on parts that lie within one video frame only, so the time dimension can be ignored, reducing the problem to the difference between two pictures. Of course, due to the usage of reference frames for temporal prediction, an error in one frame can affect other frames, but I deal with this separately in Section 3.3.

My preprocessor-based approach uses such a picture quality loss metric as a black-box function. Different methods to assess perceptional picture similarity can be dropped in there, given that they satisfy two basic requirements:

- The function works in a full-reference manner, meaning that it takes two aligned rectangular pictures as input. One of those pictures is considered the original image, the other the distorted image.
- The function returns a single value that correlates positively with the quality loss between two pictures as perceived by a human viewer. The correlation should be of reasonable linearity, because this is needed in Sections 3.2.7, 3.3.2, and 3.3.3.

For later speed optimization, the following additional properties are helpful, but not strictly required:

- The function can benefit from knowing, that differences of interest between the images are limited to a rectangular area smaller than the full picture.
- The function can be computed with scalable precision, where the result gets more accurate as more computation time is invested.

Of course, the "most correct" image quality loss function that can be used here is subjective evaluation by actual humans. But that is not feasible in the context of video decoding, where such an analysis would have to be done for every frame. Hence I was looking for existing mathematical approximations of image quality loss.

### 2.4.1 Error Sensitivity Approaches

The existing quality metrics range from simple mathematical operations to complex psychophysical models. The most widely used metric is the mean squared error (MSE), which is convenient, because it is easy to compute. It simply averages the squared difference of original and distorted image. Unfortunately, this does not always match perceived quality loss [28, 29], because errors with an equal impact on the MSE can vary greatly in their visibility. A related metric is peak signal to noise ratio (PSNR) [30], which is derived from MSE with

$$PSNR = -10 \log_{10} \frac{MSE}{L^2}$$

where $L$ is the maximum signal amplitude, which is 255 for an image component coded with 8 bit resolution. Being just a logarithmically scaled version of MSE, it performs equally bad in respect to perceived quality loss.

Within the group of psychophysical quality metrics, errors are not treated equally, but according to their estimated visibility. As with MSE and PSNR, the basic assumption is that perceived loss of quality relates to the visibility of an error signal superimposed on the image. The algorithms then quantify the visibility of the error signal by leveraging psychophysical measurements on humans. An example for such work can be found in [31]. Three inherent problems of these approaches (as listed in greater detail in [32]) shall be summarized here:

**Quality Definition Problem:** Loss of quality is not necessarily equivalent to error visi-

bility, as some errors may be visible, but not disturbing.

**Natural Image Complexity Problem:** The psychophysical measurements often use simple patterns like spots or bars as stimuli and treat natural images as combinations of those patterns. The results from the measurements are then similarly combined to results for natural images. Given the nonlinearity of the human visual system, it is questionable if such a combination is adequate.

**Cognitive Interaction Problem:** The cognitive processing and understanding of an image influences the perception of errors. When the viewer recognizes a known real-world object, the human brain can exploit a priori knowledge to cover up errors.

### 2.4.2 Structural Similarity Index

Motivated by those deficiencies, Wang, Bovik, Sheikh, and Simoncelli developed the Structural Similarity (SSIM) Index [32], which I chose to use. The reasons for this decision are:

- The elegance of the approach, which combines interesting properties from both categories discussed previously:
  - It is not a simplistic mathematical formula like MSE, but is still calculated easily enough to use it for video, where the function will be used often.
  - It does not use a complicated model of the human visual system, but still performs good compared to actual visual evaluation.
- Implementations of the algorithm are publicly available, although it turned out later, that these could not be used due to licensing issues (see Section 4.1).

The basic assumption of SSIM is that the human visual system is highly adapted to extract structural information from images. Therefore, this approach does not try to estimate degradation through the visibility of errors, but considers quality loss as perceived changes in structural information. This marks the change from the previous bottom-up algorithms, which tried to mimic the behavior of low-level components of the human visual system, to a top-down algorithm, which emulates the overall function of the human visual system. These differences make this method less vulnerable to the three problems listed in the previous section.

SSIM works by iteratively comparing aligned, limited local areas of two images. I first summarize the operation on a single local area, then I explain how this operation is expanded to calculate one result for an entire picture. More details can be found in [32]. Lastly, I comment on applying SSIM to video and how it satisfies my requirements for a video quality loss metric.

The local SSIM algorithm expects two aligned two-dimensional image areas as input, each spatially discretized into $n$ pixels. The pixel values of the two image areas are scanned into two column vectors $\underline{x}$ and $\underline{y}$, which need to have their components $(x_i, y_i)$ in matching order. The signal intensity[1] is quantified by the mean value $\mu$ for each vector:

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i$$

[1] The original work in [32] talks about "luminance" here, but this is misleading in the video context, because luminance denotes a linear light transfer characteristic. As sample values in video are gamma corrected, the term "luma" would be more appropriate. But as SSIM is later applied to chroma samples as well, I chose the more generic term "intensity" here.

An intensity comparison value $i$ is defined as:

$$i(\underline{x}, \underline{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

where $C_1$ is a constant to avoid instability when $\mu_x^2 + \mu_y^2$ is close to zero. The empirical variance $\sigma^2$ for each vector estimates the signal contrast:

$$\sigma_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)^2$$

$$\sigma_y^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \mu_y)^2$$

The contrast comparison value $c$ is defined as:

$$c(\underline{x}, \underline{y}) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

The structural information is represented by the empirical covariance $\sigma_{xy}$ of both vectors:

$$\sigma_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y)$$

The structural comparison value $s$ is defined as:

$$s(\underline{x}, \underline{y}) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

This resembles the correlation coefficient $\frac{\sigma_{xy}}{\sigma_x\sigma_y}$, so SSIM interprets correlating image areas as structurally similar. The constants $C_2$ and $C_3$ were introduced for similar reasons as $C_1$ and the SSIM authors suggest

$$C_1 = (K_1 L)^2,\ C_2 = (K_2 L)^2,\ C_3 = \frac{C_2}{2}$$

with $L$ being the peak signal amplitude (255 for 8-bit images) and $K_1 = 0.01$, $K_2 = 0.03$.

Now, the three comparison values $i(\underline{x}, \underline{y})$, $c(\underline{x}, \underline{y})$ and $s(\underline{x}, \underline{y})$ are multiplicatively combined to form the local SSIM index:

$$\mathrm{SSIM}(\underline{x}, \underline{y}) = i(\underline{x}, \underline{y}) \cdot c(\underline{x}, \underline{y}) \cdot s(\underline{x}, \underline{y})$$

This can be simplified to the final formula of the local SSIM index:

$$\mathrm{SSIM}(\underline{x}, \underline{y}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

Three interesting properties can be derived from this equation:

- Symmetry: $\mathrm{SSIM}(\underline{x}, \underline{y}) = \mathrm{SSIM}(\underline{y}, \underline{x})$
- Boundedness: $\mathrm{SSIM}(\underline{x}, \underline{y}) \leq 1$
- Unique Maximum: $\mathrm{SSIM}(\underline{x}, \underline{y}) = 1 \quad \text{iff} \quad \underline{x} = \underline{y}$

This local SSIM index is then applied within a sliding window that moves pixel by pixel over the entire picture. The average over all local SSIM results gives the mean SSIM (MSSIM) index, which quantifies the similarity of two images.

### 2.4.3 Extending SSIM for Video

The application of SSIM to video has been discussed in [33]. Like MSSIM, the local SSIM is applied inside a sliding window across individual frames. This window is now specified to be an 8×8 pixel square. But other than for MSSIM, the individual local SSIM results are not averaged uniformly, but in a weighted fashion along three criteria:

1. For video frames in the Y'$C_B C_R$ colorspace [34] used in H.264, the local SSIM values are calculated for each component separately. The local value from the Y' (luma) component is then weighted with 0.8, the values from the $C_B$ and $C_R$ color difference components are weighted with 0.1 each. These weighted values are added to form the local SSIM index for the given position of the sliding window.
2. Because bright areas are known to attract the observer more than dark areas, the local values are weighted according to their

corresponding window's average intensity $\mu_x$ before being aggregated over the whole frame.

3. Due to the motion blur effect, structural loss is more tolerable in high motion scenes. Therefore the aggregated values for each frame are weighted by the average motion vector length when evaluating motion from each frame to its direct successor.

Wang, Lu, and Bovik compared the performance of the given algorithm with the candidate algorithms of the VQEG Phase I FR-TV test [35]. The goal of this test was to compare various video quality metrics against a given test set of video streams. The test set provides the video streams in different quality levels, for which the VQEG derived subjective quality values by conducting test viewings with real people. The candidate algorithms calculated objective quality indices for the videos, which were fitted to the subjective values. Some statistical indicators allowed to quantify the precision of the fit. SSIM was not part of the original test, but the authors ran it against the same test set and calculated the same statistical indicators. Table I on page 7 of [33] shows, that the algorithm described above outperforms all VQEG Phase I contenders. Even a simplified version, where the weightings 2 and 3 were replaced with uniform averaging, is better than most VQEG Phase I algorithms and only slightly outperformed by one of the candidates (the KPN/Swisscom CT algorithm, see [35] for details). Because of this good performance and the reduction in computational load due to the skipped motion analysis, I chose to use this simplified version.

The paper also proposes an even simpler version that additionally replaces weighting step 1: The $C_B$ and $C_R$ components are ignored and only the local SSIM value of the Y' component is used with a weight of 1.0. The original paper gives no evaluation of this simplification, so I conducted a small analysis, which I present in Section 3.2.6. This way of applying SSIM to video is equivalent to calculating MSSIM for the Y' component of each frame. A visualization compared to MSE can be seen in Figure 4.

In addition to the speed up by the simplification, the authors also provide a way to calculate SSIM faster with less precision. The idea is to not consider all possible positions of the sliding window, but a randomly selected portion thereof. Varying the ratio of calculated against ignored window positions is a straightforward way to scale the precision and performance of the computation. Also in Section 3.2.6, I determine, what values are beneficial.

Another approach toward expanding SSIM to videos is to extend the sliding window from a two dimensional to a three dimensional one, with time complementing the spatial dimensions as the third coordinate. The local SSIM operates on vectors of linearly ordered image samples, so it does not care about the dimensionality of the window. This SSIM version would be closer to the video model I presented in Section 1.2 and would be capable of quantifying degradations along the time axis like reduced framerate. However, a quality loss metric that works on single images is sufficient for my work and [33] briefly mentions that a three dimensional window does not yield significant improvements. Therefore I did not pursue this idea, but it remains an interesting point for future work to develop and analyze a three dimensional quality loss metric.

### 2.4.4 Electing SSIM

As I intend to use SSIM as my quality loss metric, I have to discuss, how it matches the requirements established in Section 2.4 on page 10.

SSIM clearly fulfills the first requirement of taking two aligned rectangular images as input. One does not even have to take care, which image is the original and which is the degraded version, because we saw that SSIM treats its inputs symmetrically.

The second requirement needs discussion: It states that the return value of the algorithm


Figure 4: Comparison of MSE and SSIM; upper left: original video frame (from BBC video, see Table 1 on page 18); upper right: video frame degraded by compression artifacts (reencoded original at 384 kbit MPEG-4 Part 2); lower left: map of Y' component MSE values (black represents 0, white represents $50^{-1}$ or larger); lower right: map of Y' component local SSIM values (black represents 1, white represents 0). It is clearly visible that SSIM detects the structural loss in the grass regions much better than MSE, which in turn penalizes errors in the zebras that are less disturbing to the viewer.

needs to correlate in a positive linear way with the perceived quality loss. $\mathrm{SSIM}(\underline{x}, \underline{y})$ returns the maximum of 1 for identical images and decreases as structural loss increases. By design of this metric, structural loss models the perceived quality loss, so a lower SSIM value indicates higher quality loss. This negative correlation is unwanted, so I will use $1 - \mathrm{SSIM}(\underline{x}, \underline{y})$, which results in 0 for identical images and correlates positively, because it increases with quality loss. However, to ensure a linear correlation, test viewings have to be conducted where human observers rate videos on a linear scale. Fortunately, two independent analyses have published plots of SSIM values against mean opinion scores (MOS) for videos. Both plots (Figure 5 c in [33] and Figure 17 in [29]) show acceptable linearity for pure SSIM indices without any fitting.

SSIM additionally fulfills the two optional optimization criteria: It can easily benefit from knowing that interesting differences between images are limited to a given area of the image by simply not calculating any positions of the sliding window that do not touch the area of changes. And by not calculating all possible window positions but only a portion of them, SSIM can be calculated with scalable precision.

### 2.4.5 Compressed Domain Metrics

Although I selected SSIM for use in my work, I want to make some additional comments on compressed domain metrics:

It is possible to develop a quality evaluation metric that does not operate on images represented as pixels, but on a representation closer to the

one used in a compressed video stream. The video quality metric (VQM) presented in [36] for example uses the existing DCT coefficients. This might reduce computational load, because images do not need to be fully decoded to calculate a quality metric on them. With such a technique, it may even become feasible to turn the preprocessing approach my entire thesis is based on into an on-line method.

The compressed domain metrics are also appealing, because instead of reinventing the wheel by defining a new perception model, they make use of the perception model built into today's video coding algorithms. Since it is the goal of the encoding process to reduce the video size without reducing the quality, the encoder already needs a built-in notion of quality, which a compressed domain metric can exploit.

But despite their advantages, such a compressed domain metric is naturally highly dependent on the video coding algorithm it is based on and reuse for different coding standards may be difficult or even impossible. Applying the VQM mentioned above to H.264 would not be trivial, because H.264 does not use DCT. Developing a metric more fit to H.264 is outside the scope of this thesis, but is left open for future work. In addition, [29] shows that SSIM matches perceived quality loss better than VQM.

As I have now discussed the video coding algorithm and the quality loss metric in depth, I continue by developing the video preprocessor from these building blocks.

# 3 Video Preprocessor Design

The video preprocessor partitions the video and provides a faster alternative to synthesize the content of each partition. Together with a quantification of the resulting costs and quality differences to the original video, the extracted data is stored in a sideband bitstream.

## 3.1 Metrics for Decoding Time

A good estimate of decoding time is vital for a sensible decision on which parts of the video to decode and which to skip. However, I already described the method for obtaining such estimates in [20], so not all details will be reiterated here. I first comment on the set of videos used throughout this thesis. Then I go through the various decoding steps and present the metrics used to estimate their execution time.

### 3.1.1 Test Videos

Table 1 lists the videos used throughout this thesis. Note that material from two different encoders is being tested: Three videos were encoded using the commercial FastVDO encoder, three others were reencoded with the open-source x264 encoder [40]. The reencoding was performed to control the number of slices per frame and to manufacture Baseline Profile content. I deliberately left out interlaced content, because the adaptive frame/field coding (AFF) of H.264 would render a large portion of my code more complicated without any benefit in proving the feasibility of my method.

Although x264 does not support several interesting features of the Extended Profile, like flexible macroblock ordering (FMO), arbitrary slice ordering (ASO) or slice data partitioning (SDP), it supports most coding options of the Main and Baseline Profiles like all slice types and all macroblock types with all prediction modes. It also supports both CAVLC and CABAC entropy compression methods. Consequently, [29] has shown that x264 visually outperforms other encoders, so the quality should be sufficient for my needs. However, the missing support for FMO forces me to use slices that are simple horizontal stripes.

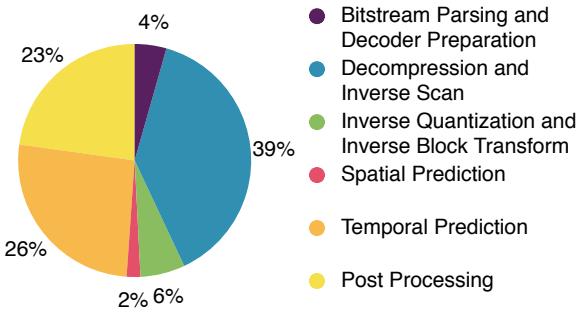



Figure 5: Execution time profile for the BBC test video

### 3.1.2 Decoding Steps

The decoder model from my earlier work [20] can be applied here quite straightforwardly (see Figure 2). Figure 5 shows an execution time profile for the BBC test video to give a feeling for the importance of the various decoding steps. All measurements have been made with the FFmpeg [41] H.264 decoder on an AMD Sempron 2200+ (1.5 GHz) under Linux.

Unlike [20], where I predicted video decoding times on frame level, I need something more fine grained here, because I want the scheduler to be able to handle parts of the video smaller than frames. Slices are an obvious candidate, because they are decoded mostly independently of one another and are potentially smaller than frames. I further elaborate why slices are a good choice for my purposes in Section 3.2. However, there are parts of the decoding process taking place outside a slice context, the most prominent being header parsing. These parts can be regarded as pseudo-slices between the actual ones.

The H.264 standard describes I-, P- and B-slice types, similar to the frame types in previous coding standards. But other than in previous standards, the actual decoding does not differ much

| Name | Content | Duration | Resolution | Size | Profile | Slices/ Frame | Properties | Source |
|---|---|---|---|---|---|---|---|---|
| Freeway | cars on a freeway | 0:09 min | 704×576 | 2.1 MB | Main | 1 | fixed camera scene | [37]* |
| Golf | golfer making a swing | 0:12 min | 176×144 | 55 KB | Main | 1 | fixed camera scene, very little motion | [37]* |
| Shore | flight over a shoreline at dawn | 0:27 min | 352×288 | 758 KB | Main | 1 | camera moving all the time | [37]* |
| BBC | various combined broadcast quality clips from BBC motion gallery | 1:29 min | 1280×720 | 57 MB | Main | 5 | clips with very different properties (low/high motion, local/global motion) | [38]** |
| Lady1 | movie trailer for "Lady In The Water" | 1:44 min | 1920×1080 | 92 MB | Main | 20 | high detail images with calm motion | [39]** |
| Lady2 | movie trailer for "Lady In The Water" | 1:44 min | 1920×1080 | 92 MB | Baseline | 4 | high detail images with calm motion | [39]** |

* videos have not been reencoded, the original FastVDO encoded material was used
** videos have been reencoded using x264

Table 1: Test videos used throughout this thesis

between those types, as they merely state, which coding options are allowed to be used in the respective slice. Consequently, I will not distinguish between those types in the following discussion, but rather come up with one set of metrics useful for all slice types.

**Bitstream Parsing and Decoder Preparation:** This decoding step is the one not directly associated with an actual slice. The decoder reads in and prepares the bitstream of the upcoming frame and processes any header information available. The number of slices in this frame is not even known yet. This step should therefore be treated as a pseudo-slice that precedes the first real slice of each frame. The decoder preparation part mainly consists of precomputing symbol tables to speed up the upcoming decompression. Its execution time is negligible, so I chose to treat these two steps as one. Because each pixel is represented somehow in the bitstream and the parsing depends on the bitstream length, the candidate metrics here are the pixel and byte counts. Figure 6 shows that a linear fit of both actually matches the execution time.

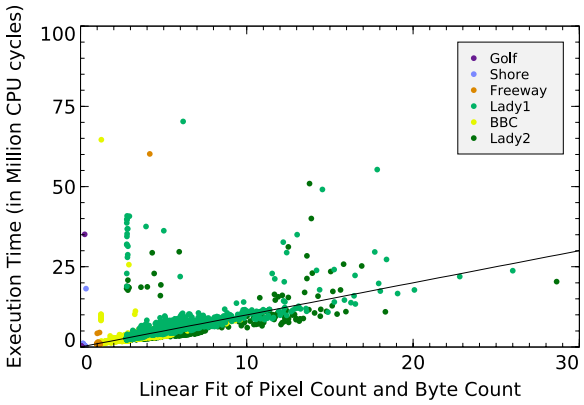



Figure 6: Frame-based execution time estimation for bitstream parsing

**Decompression and Inverse Scan:** The execution profile (see Figure 5) showed the decompression step to be the most expensive. This sets H.264 apart from other coding technologies like MPEG-4 Part 2, where the temporal prediction step was by far the most expen-

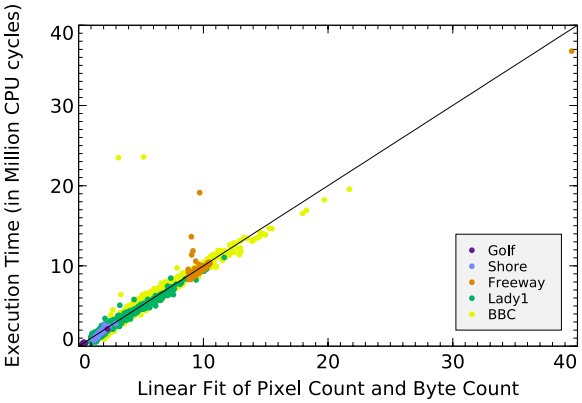



Figure 7: Slice-based execution time estimation for CABAC decompression

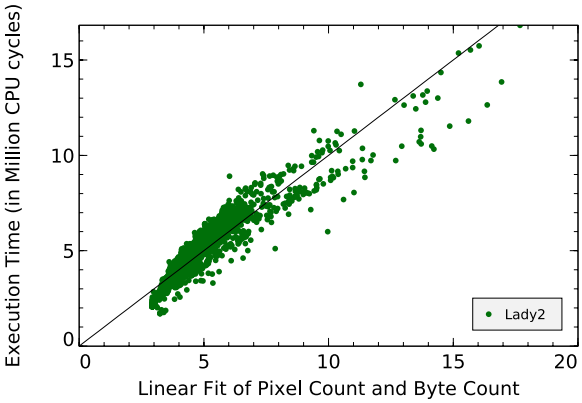



Figure 8: Slice-based execution time estimation for CAVLC decompression

sive. The reason for this shift is that the H.264 Main Profile uses a new binary arithmetic coding (CABAC) for compression, that is much harder to compute than the previous Huffman-like schemes. A less expensive variable length compression (CAVLC) is also available in H.264 and is used in the Baseline and Extended Profiles, where CABAC is not allowed. Both methods decompress the data for the individual macroblocks and already sort the data according to a scan pattern, so the inverse scan is a part of this step. Using the same rationale as for the preceding bitstream parsing, a linear fit of pixel and byte counts predicts the execution time well (Figures 7 and 8). As this step accounts for a large share of total execution time, it is fortunate that the match is accurate

**Coefficient Prediction:** Because H.264 contains a spatial prediction step, the coefficient prediction found in earlier standards is not used any more.

**Inverse Quantization and Inverse Block Transform:** These two steps convert the macroblock coefficients from the frequency domain to spatial domain, similarly to the IDCT in previous standards. However, H.264 knows two different transform block sizes of 4×4 or 8×8 pixels, which can even be applied hierarchically.

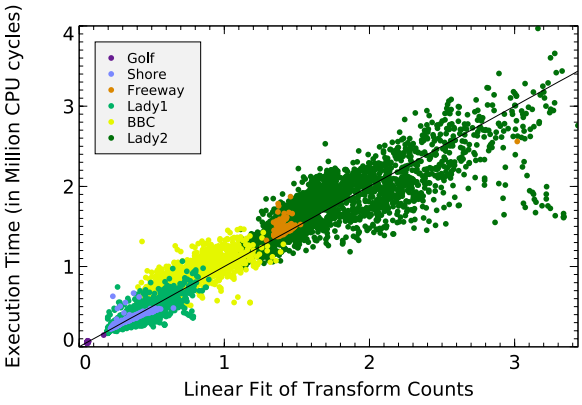



Figure 9: Slice-based execution time estimation for inverse block transform

Therefore, I count, how often each block size is transformed and use a linear fit of these two counts to predict the execution time. Figure 9 shows that this works. The remaining deviations are most likely caused by optimized versions of the block transform function for blocks, where only the DC coefficient is nonzero. But given the small percentage of total execution time this step contributes, I did not try to improve this prediction any further.

It is interesting to note that H.264 allows for another, entirely different type of macroblock: PCM macroblocks. Those are not transformed at all, the coefficients derived from the bitstream are already in the spatial domain, so they are merely copied directly into the output image. I

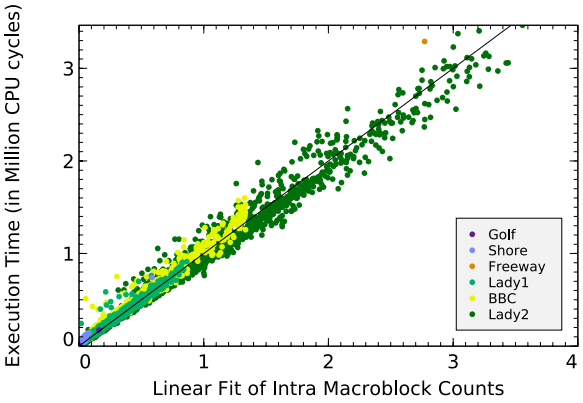



Figure 10: Slice-based execution time estimation for spatial prediction

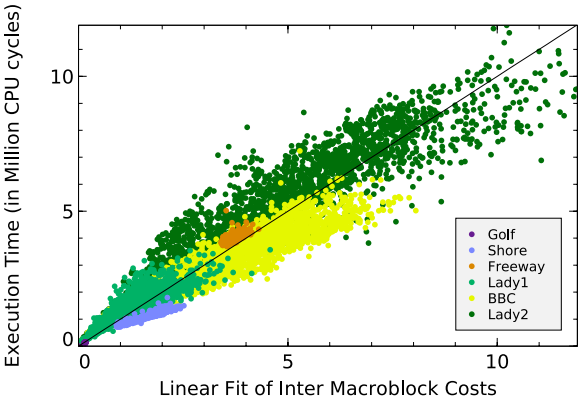



Figure 11: Slice-based execution time estimation for temporal prediction

did not experience such macroblocks in any test material and I assume they will only be inserted when H.264 is used in lossless mode. However, I am going to assign a metric for those macroblocks, should they ever occur in future videos.

**Spatial Prediction:** In this step, already decoded image data from the same frame is extrapolated with various patterns into the target area of the current macroblock. This prediction can use blocksizes of 4×4, 8×8, or 16×16 pixels, so I count those prediction sizes separately. A linear fit of those counts adequately predicts the execution time (see Figure 10).

**Temporal Prediction:** This step was the hardest to find a successful set of metrics for, because it is exceptionally diverse. Not only can motion compensation be used with square and rectangular blocks of different sizes, each block can also be predicted by a motion vector of full, half or quarter pixel accuracy. In addition to that, bi-predicted macroblocks use two motion vectors for each block and can apply arbitrary weighting factors to each contribution. In [20], I broke this problem down for MPEG-4 Part 2 to counting the number of memory accesses required. I used a similar approach here, but counting memory accesses in the code was impractical, as FFmpeg uses carefully interwoven preprocessor macros beyond my comprehension to generate the motion compensation code. Instead, I consulted the H.264 standard [2] and also did some empirical improvements to come up with motion cost values, depending on the pixel interpolation level (full, half or quarter pixel, independently for both x- and y-direction). These cost values are then accounted separately for the different blocksizes of 4×4, 8×8, or 16×16 pixels. The possible rectangular blocksizes of 4×8, 8×4, 8×16, or 16×8 are treated as two adjacent square blocks. Bidirectional prediction is treated as two separate motion operations. The resulting fit can be seen in Figure 11. This result is clearly not perfect, but evaluation will show it to be good enough. The reason for the rather large deviations from a linear match are probably caused by the remaining variation not tightly accounted for by the chosen metrics. Another obstacle toward a better fit are different memory access times due to largely unpredictable cache misses with the high definition images and H.264's exceptionally large reference buffers.

**Post Processing:** The mandatory post processing step tries to reduce block artifacts by selective blurring of macroblock edges. A sufficiently precise execution time prediction is pos-

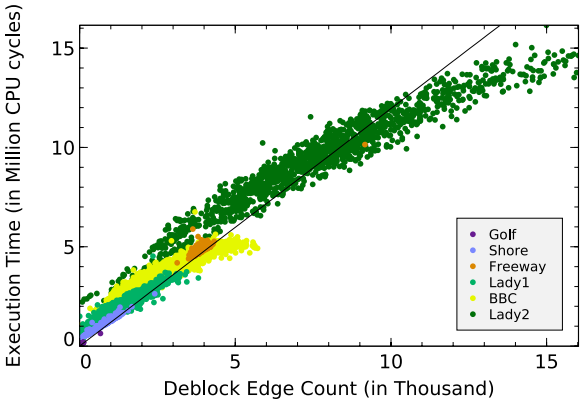



Figure 12: Slice-based execution time estimation for post processing

sible by just counting the number of edges being treated (see Figure 12).

### 3.1.3 Metrics Summary

The metrics selected for execution time prediction therefore are:

- pixel count,
- byte count,
- PCM-type macroblock count,
- count of intracoded blocks of size 4×4,
- count of intracoded blocks of size 8×8,
- count of intracoded blocks of size 16×16,
- motion cost for intercoded blocks of size 4×4,
- motion cost for intercoded blocks of size 8×8,
- motion cost for intercoded blocks of size 16×16,
- count of block transforms of size 4×4,
- count of block transforms of size 8×8,
- count of deblocked edges.

These metrics are accounted for each slice individually and stored in sideband data. With execution time discussed, the next section focuses on the partitioning of the video.

## 3.2 Partition Replacement

I want the final decoder to be able to decide for each video partition individually, whether to decode it or not. This immediately brings up two requirements:

- When a partition is not decoded, it must be easily skippable in the video bitstream. After the skipping, the decoder must be in a consistent state to be able to decode the next partition.
- When a partition is not decoded, it must be replaced with visually similar image material, so the resulting video does not have "holes", where its content is undefined. The resulting overall quality loss introduced by the replacement must be known in advance to help the decoder's scheduling decision.

The first requirement is fulfilled by H.264's built-in network abstraction layer (see Section 2.1). This layer divides the bitstream into individual packets (network abstraction layer units or NALUs), which always start byte aligned. Unless the transport medium provides a natural packetization of the bitstream, as common container formats like QuickTime do, each NALU is prefixed with a unique three byte start code of `0x00 0x00 0x01`, which does not appear anywhere else inside the NALU. This allows a decoder to easily find NALU boundaries in the bitstream and because one NALU can contain at most one slice of a coded frame, finding NALU boundaries automatically means finding slice boundaries. Therefore, I use slices as the video partitions the decoder will later schedule.

Of course I could combine multiple slices into one partition, but I expect the decreased granularity to be counterproductive. Slices are already large in videos found today. One to four

slices per frame are common, but I will use self-encoded material with more slices to demonstrate the effect of a finer scheduling granularity.

Using entities smaller than slices as skipping partitions is not immediately possible, because the slice bitstream is so densely compressed that it is hard to find boundaries of syntax elements without fully decompressing it. And as decompression is a large chunk of the entire decoding process (see Figure 5), such an approach does not appear beneficial. However, even if the skipping cannot use partitions beneath slice level, the replacement strategy can, as I show in the following section.

### 3.2.1 Replacement Partitions

To fulfill the second requirement, I need to find replacement content to fill in the parts the decoding scheduler might decide to skip. At first I wanted to find such replacements again on the slice level, but I quickly realized that to be infeasible, because replacement decisions have to be based on structural properties of the frame. Although hard to implement, an example that makes this rationale clear immediately is foreground/background separation: Low-motion background areas of the image might need a different replacement than more dynamic foreground objects. But in the H.264 videos commonly found, the slice structure does not follow any structural or semantical properties of the frame. Today, dividing a frame into slices is mostly done to exploit multiprocessor machines that can use slices to parallelize the decoding. With more sophisticated encoders, this can be changed, because H.264 allows arbitrarily shaped slices with its flexible macroblock ordering (FMO) feature (see Section 2.1). This can be leveraged in the future to encode slice structures that follow the structure of the frame's content. But in the open-source world, neither an encoder nor a decoder supporting FMO are available today. In addition, FMO might also be currently neglected, because only the Baseline and Extended Profiles of H.264 include it, but the Main Profile, which exclusively supports the highly efficient CABAC compression, does not allow FMO (see Figure 1).

To somewhat emulate the advantages of FMO, I decided to develop a replacement strategy using replacement partitions that are completely decoupled from the skipping partitions discussed previously. Of course, once the decoding scheduler decides to skip decoding of a frame's slice, exactly this slice's pixels will be reconstructed by replacing them. But different areas of the slice in question might be replaced differently. I will call these areas replacement partitions and skipping partitions, respectively. I already determined above that the skipping partitions are going to be identical to slices. The replacement partitions are left to be discussed now.

### 3.2.2 Replacement Strategy

Replacement content cannot be synthesized knowing just the frame, whose "hole" is to be filled. Although there are algorithms known in still image and video retouching [42], that can fill a specified region by extrapolating from surrounding image content, these methods are computationally far too demanding to be used during video decoding. Therefore, the best way to replace parts of a frame is to copy portions of previously decoded images into the hole. Because reference frames are previously decoded images the decoder keeps in memory, it is clear I should use them for my purposes. This idea is especially adequate for H.264, which, with its large buffer of up to 32 reference frames, offers a wide choice of candidate replacement regions to choose from.

Image content should also not be replaced by always copying from the same location of a different frame. I tried that, but the unsatisfying results made clear I should consider copying from a different region of a different frame to compensate for motion between the two images. This compensation is of additional importance, as I will apply the previously discussed quality

loss metric to the replacements to quantify the visual error they introduce. In Section 2.4 on Page 10, I defined the requirements for a quality loss metric and the first requirement includes the prerequisite of aligned images. This can be fulfilled or at least approximated by compensating for motion.

While motion analysis of a series of images is generally expensive, the H.264 coded video stream already provides motion vectors of good quality, because the reduction of temporal redundancy is vital to reduce the size of the compressed video. An example visualization of H.264 motion vectors can be seen in Figure 13.

However, I cannot simply extract and store all motion vectors, because then, the data rate of the preprocessor output could easily rival the data rate of the original H.264 feed as I would essentially duplicate a large portion of the stream. Therefore, a representation needs to be found that is considerably more lightweight, but still capable of approximating the motion of the frame to provide good partition replacement. In the next section, I lay out my idea of using a quadtree to accomplish that.

### 3.2.3 Motion Vector Quadtree

Quadtrees [43] are used in computer graphics contexts to partition two dimensional data. Starting with the root node representing the complete frame, I recursively and adaptively subdivide each node's region into four subregions. This leads to a nonuniform subdivision of the frame, represented by a nonuniform quadtree, meaning that nodes of maximal depth do not necessarily have the same depth. But the tree should cover the whole frame, so each node has either zero or four subnodes. An example of a possible quadtree subdivision is given in Figure 14. The subdivisions always cut a region in half vertically and horizontally, regardless of the frame's aspect ratio. But all cuts are rounded to integer multiples of the macroblocks size.

**Algorithm 1** Fully subdividing the quadtree

```
// Step 1
initialNode = entireFrame;
populateQuadtreeNode(initialNode);

function populateQuadtreeNode(quadtreeNode) {
  // Step 2
  referenceAccess[] = 0;
  foreach (macroblock in quadtreeNode)
    referenceAccess[macroblock.reference]++
  mostOftenUsedReference =
   indexOfMaximum(referenceAccess);
  quadtreeNode.reference = mostOftenUsedReference;

  // Step 3
  averageVector = 0;
  foreach (macroblock in quadtreeNode)
    if (macroblock.reference ==
     mostOftenUsedReference)
      averageVector += macroblock.vector;
  quadtreeNode.vector = averageVector;

  // Step 4
  quadtreeNode.subnodes[] =
   subdivideNode(quadtreeNode);
  foreach (subnode in quadtreeNode.subnodes)
    if (subnode.area >= singleMacroblock.area &&
     subnode.motionVectorCount >= 1)
      populateQuadtreeNode(subnode);
    else
      quadtreeNode.subnodes = null;
}
```

To associate motion vectors to quadtree nodes, I developed the following algorithm, which works in two parts. The first part recursively creates a fully subdivided quadtree. A pseudo-code description can be found in Algorithm 1, a textual description follows:

1. Start the iteration with the root node of the quadtree covering the entire frame.
2. For the region covered by the current node, iterate over all macroblocks and count, how often each reference frame is accessed. Determine the reference frame used most often. This reference is stored in the current node.
3. For the region covered by the current node, iterate over all macroblocks and select those motion vectors that use the reference


Figure 13: Visualization of H.264 motion vectors. Images are neighboring (but not directly consecutive) frames of BBC video, with motion vectors represented as arrows. Between frame one and two, the monkey looks up, between frame two and three, the camera zooms back. Both of these motions (the first being more local, the second global) are followed by the motion vectors.

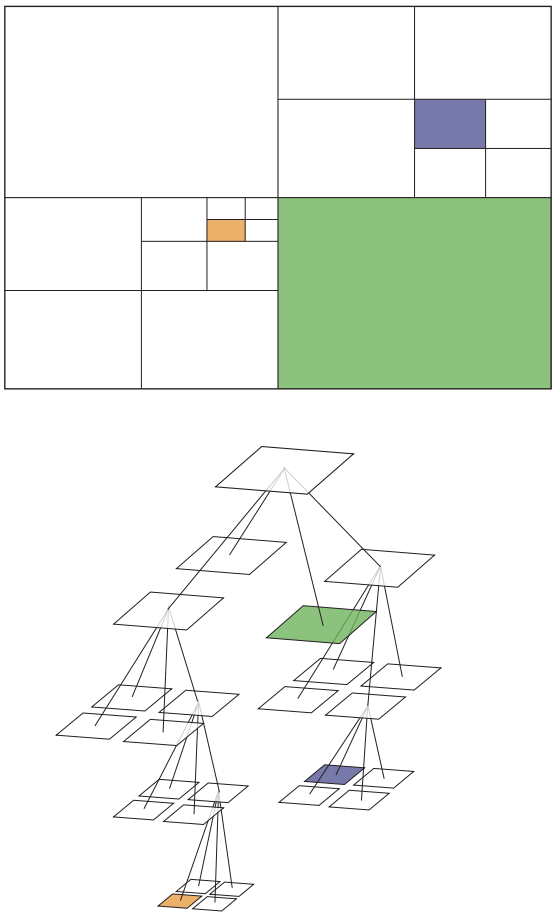


Figure 14: Quadtree subdivision example. For illustration, three pairs of corresponding nodes have been colored in both the 2D-plane and the tree representation.

frame determined in Step 2. Average all selected motion vectors. The resulting vector is stored in the current node, rounded to full pixel accuracy.

4. Subdivide the current node's region into four subregions, thus creating four subnodes of the current node. If the areas covered by the subnodes are each at least the size of one macroblock and contain each at least one motion vector, repeat Steps 2 to 4 for each subnode, otherwise delete the subnodes and return.

This yields a fully subdivided quadtree with a hierarchy of reference frames and motion vectors. All leaf nodes of this tree together fully cover the frame and provide the motion approximation needed to replace the image content. However, storing all these nodes results in too much data, so the algorithm continues by pruning the quadtree without sacrificing too much quality. The second part adaptively cuts off nodes from the leaves toward the root node to reduce the amount of data. The finer subdivisions provide better approximation of the frame's motion, so every time nodes are removed from the tree, the resulting quality loss is checked. A pseudo-code version is available in Algorithm 2.

1. Start the iteration with the root node of the quadtree covering the entire frame. Because the iteration in the following steps is a head-recursion, the algorithm will first descend into the tree and then process the nodes on the way up.

2. Return to the parent node, if the current node has no subnodes.

3. If the current node has subnodes, recurse to prune them first. This ensures bottom-up pruning.

4. Return to the parent node, if the current node's subnodes are not leaves (that is: they in turn have subnodes). This ensures that cutting is not performed here if it failed on one of the subnodes.

5. Perform region replacement for the entire frame by iterating over all leaves of the current version of the tree. The region covered by each leaf node is replaced with a region of equal size, taken from the reference frame stored in the corresponding leaf node. The replacement region is offset by the average motion vector stored in the leaf node.
   The replacement is performed for the entire frame so that the quality loss metric in the next step can compare two complete frames.

**Algorithm 2** Adaptive pruning of the quadtree

```
// Step 1
initialNode = entireFrame;
pruneQuadtreeNode(initialNode);

function pruneQuadtreeNode(quadtreeNode) {
  if (!quadtreeNode.subnodes)
    return;  // Step 2
  else
    foreach (subnode in quadtreeNode.subnodes)
      pruneQuadtreeNode(subnode);  // Step 3

  // Step 4
  foreach (subnode in quadtreeNode.subnodes)
    if (subnode.subnodes) return;

  // Step 5
  replaceFrame = clone(originalFrame);
  doReplacement(initialNode, replaceFrame);
  // Step 6
  qualityLoss1 = compare(replaceFrame, originalFrame);

  // Step 7
  temp = quadtreeNode.subnodes;
  quadtreeNode.subnodes = null;
  replaceFrame = clone(originalFrame);
  doReplacement(initialNode, replaceFrame);
  // Step 8
  qualityLoss2 = compare(replaceFrame, originalFrame);

  // Step 9
  if (acceptable(qualityLoss2 – qualityLoss1))
    temp = null;
  else
    quadtreeNode.subnodes = temp;
}

function doReplacement(quadtreeNode,
 replaceFrame) {
  if (quadtreeNode.subnodes)
    foreach (subnode in quadtreeNode.subnodes)
      doReplacement(subnode, replaceFrame);
  else
    replaceFrame[quadtreeNode.area] =
     quadtreeNode.reference[quadtreeNode.vector];
}
```

6. Calculate the quality loss of the entire frame with the replacements, compared to the original frame. This step can be accelerated by confining the quality metric to the area of the current node as will be discussed in Section 3.2.6.

7. Remove all subnodes of the current node, so that the current node becomes a leaf and execute the replacement again as described in Step 5.
   As the current node is now a leaf, the replacement will be determined by the reference frame and motion vector taken from the current node instead of the multiple reference/vector pairs from its subnodes.

8. Calculate the quality loss of the frame with the replacements again. This can again be accelerated by applying the quality metric to the current node only.

9. I expect the coarser subdivision with the subnodes removed to lead to a higher quality loss than the finer subdivision. If the coarser subdivision has increased the quality loss only moderately compared to the finer subdivision, the subnodes removed in Step 7 are discarded. Otherwise they are reattached. In both cases, control flow returns. What constitutes an acceptable increase is yet to be defined.

The result of this algorithm is a nonuniformly subdivided quadtree that approximates the motion in the frame with less data than the fully subdivided quadtree.

I now discuss two possible alterations to this algorithm, which I examined until I settled for the described method:

- The algorithm creates the final tree in bottom-up order, as formulated in steps 2 and 3 of the second part. The bottom-up traversal needs a fully subdivided tree as a starting point, which is why the first part of the algorithm is needed. Top-down creation which builds the nodes as it progresses down would be an alternative.
- Once a cutting step has not been performed because of a too high increase in quality loss, all subsequent nodes along the path to the root node will not be cut off either.

This inhibitive behavior is caused by Step 4 of the second part. Considering all possible cuts would be an alternative.

Figure 15 illustrates these options.

**Alternative 1: Top-Down Creation.** In a first iteration, I tried a different approach to build the quadtree, effectively traversing the nodes in reverse order, from the root toward the leaves and stopping at the first subdivision that appears to be beneficial. This top-down approach would have to look at considerably fewer nodes, so it is faster. However, this solution turned out to be inferior to the one presented above.

The reason why the strategy of adaptive subdivision by threshold is better suited for bottom-up order compared to top-down order seems to be that sometimes the subdivisions of low depths, where nodes cover large areas, have a reversed effect on the quality: The quality loss with the subdivision may be higher than without, although one would expect the finer subdivision to have lower quality loss. I explain this behavior with edges in the frame's interior introduced by the subdivision: On subdivision depth zero, the frame is treated as a whole, whereas the first division introduces the two region borders crossing at the frame's center. Those borders might be disruptive to the structure of the image, resulting in a quality loss. While this situation basically occurs recursively in every additional subdivision, the quality increase by the more fine grained motion vectors seems to overcompensate for the potential negative effects of those edges with increasing depth.

Therefore, I chose to traverse the tree in bottom-up direction, starting with the most fine grained subdivision. To countereffect the speed loss caused by the larger number of visited nodes, the given algorithm is optimized as noted above by limiting replacement and quality loss calculation to the areas that actually change.

**Alternative 2: Non-Inhibitive Cutting.** I also tried to apply the original cutting algorithm without Step 4, so even if a cutting step fails because of a too high increase in quality loss, the algorithm does not bail out for that part of the tree, but still tries cutting all nodes of lower depths. With this change, I hoped to find the optimal cut regarding quality versus node count.

However, it turned out that this change causes the algorithm to destroy an already found beneficial subdivision too often by collapsing the quadtree into a few large nodes. Of course this can be compensated by more strongly penalizing the quality loss of those cuts in Step 8. But this countereffects the advantage, because it enlarges the sideband data size again. Another strong disadvantage of this modification is that the preprocessing takes a lot longer, due to all nodes being always examined.

Therefore I ultimately decided to use the algorithm as originally presented: Traversing the tree in bottom-up order and with a cutting failure inhibiting further cuts of ancestors of that node.

### 3.2.4 Encoder Assumptions

This approach makes heavy use of the motion vectors already present in the coded video, but aggregates them spatially into a considerably smaller number of vectors. For this to work, I make several implicit assumptions about the encoder:

- Areas of related motion are spatially contiguous.
- For an area of related motion, the encoder picks the perceptionally most similar reference frame.
- For an area of related motion, the motion vectors do not jump erratically, but are smooth in the sense that neighboring vectors are similar in direction and length.

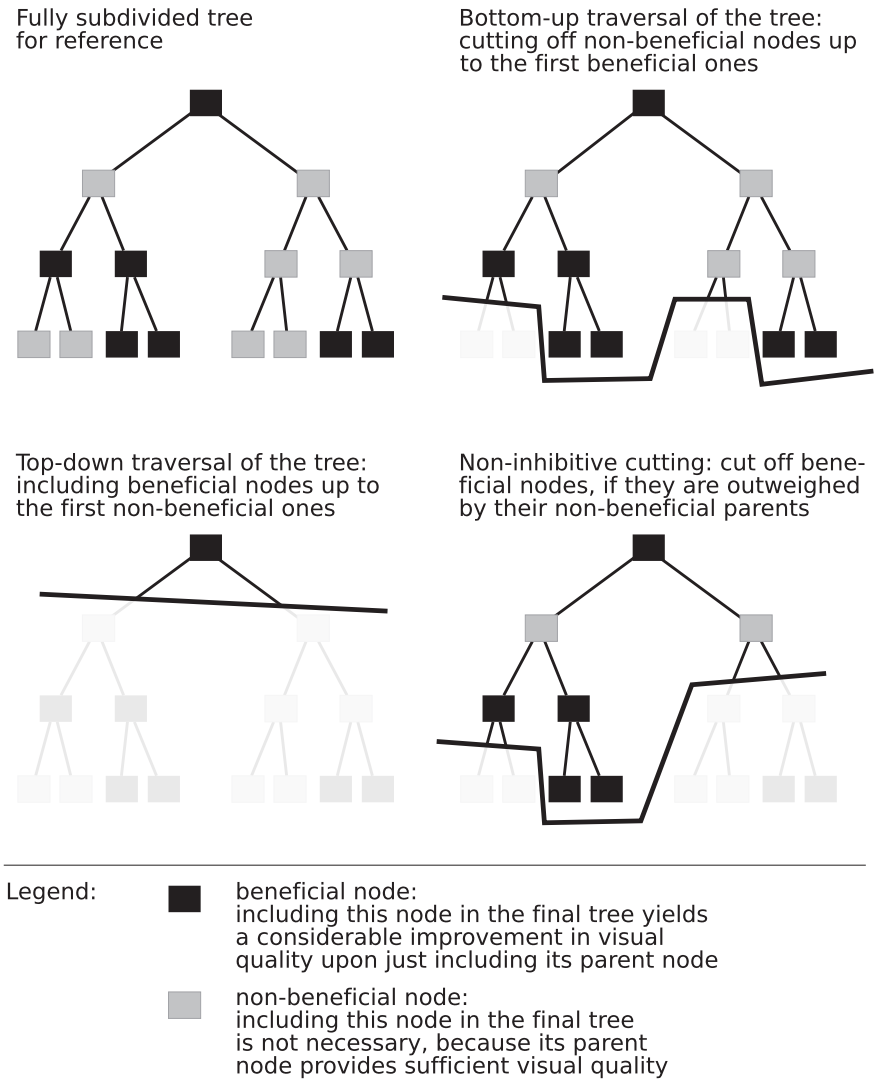



Figure 15: Alternative ways of cutting nodes off the quadtree. For simplification, the diagram uses a binary tree instead of a quadtree. Please note that the cuts must respect the invariant of each node having either zero or two subnodes (four in the original quadtree case).

These assumptions can be justified with the help of two other assumptions, which I consider to be assured:

- The encoded video content has natural motion.
- The encoder tries to minimize the size of the coded video bitstream.

Motion in video can only be perceived by an observer, if certain features in the moving part make it recognizable in every frame and thus trackable for the human eye. Although this claim can be weakened by occlusion or transparent objects, such recognizable features are most often spatial patterns. So in images with natural motion, the moving parts can be assumed spatially contiguous.

If the encoder tries to encode the video with as few bits as possible, it will have to pick a similar reference frame, because that results in a small residual error after temporal prediction and a smaller error needs fewer bits to be corrected. So the reference frames selected by the encoder can be assumed perceptionally similar.

Furthermore, it is a property of modern video coding standards, that neighboring motion vectors can be coded with fewer bits, if they follow a fluent pattern, because the motion vectors are not stored in the bitstream directly, but are also predicted. In addition, when searching for the best matching motion vector, the encoder does not employ brute-force search over the complete reference frame, but usually starts at a position, where the vector is expected to point to and then searches a small area in the vicinity of that point. Therefore, motion vectors can be assumed smooth. Figure 13 supports that.

A special case worth mentioning is the situation where a frame is only sparsely using motion vectors with the extreme case of no motion vectors at all. The latter happens regularly at frames consisting only of intracoded slices, which are usually IDR frames. Because the first part of the quadtree building algorithm stops the recursion when it would create a node with no motion vectors in them, the resulting tree for such frames will be shallow or even empty. I considered to do my own motion analysis to create additional motion vectors, but quickly dropped the idea as the approach would lose its elegance of only reusing data already provided by the encoder. In the light of the discussed assumptions, it is clear that the encoder does not make use of reference frames when using them would provide no benefit.

However, there are two situations, where this justification for not performing an additional motion analysis does not hold:

- When objects with high motion are rendered blurry in the frame, the encoder chooses intracoded macroblocks without any motion vectors, because the blurriness leads to a lack of high frequency contributions, making intra compression more efficient than motion compensated compression. Figure 16 gives an example for such a frame. But as can be seen in Figure 17, frames are typically densely covered with motion vectors, so such situations are not frequent enough to justify special treatment. I will show in Section 3.3 that this encoder behavior can even be beneficial for error propagation.

- The encoder might insert an IDR frame even though coding with reference frames is beneficial. Because IDR frames use no references, they are the possible starting positions when decoding is to begin in the middle of a video stream. Therefore, a certain rate of IDR frames has to be maintained for those starting positions not be too far apart. However, the encoder typically tries to conserve bandwidth by placing IDR frames at scene changes, where coding with reference frames would indeed be less effective.

I decided that special casing these situations does not yield enough benefit compared to the expected increase in computation time by an additional motion analysis.

I have now proposed and thoroughly discussed an algorithm to build a motion vector quadtree. I now continue by showing, how this quadtree is used and examine the influence on the decoded video's quality.

### 3.2.5 Performing a Replacement

The quadtree generated by the preprocessor describes what I call replacement partitions: areas of the frame that can be replaced with areas from previously decoded frames. What the decoder will later schedule are skipping partitions: portions of the bitstream whose decoding


Figure 16: Frame from BBC video with visualized motion vectors as an example for motion blur leading to intra compression. The ball on the right is moving fast toward the camera and will exit the frame on the right edge. Notice that although it is a moving object, no motion vectors encode that motion. Instead, the ball is fully intracoded.

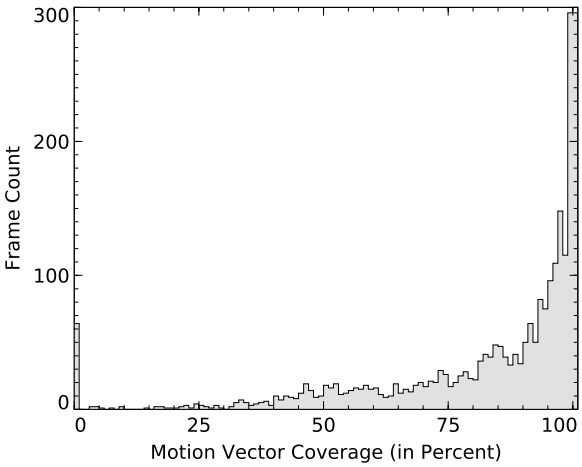



Figure 17: Motion vector coverage histogram over the entire BBC video. For each frame, the ratio of macroblocks bearing a motion vector is plotted. Note the spike on the very left, which is caused by IDR frames.

can be skipped because the decoder can be realigned to continue decoding after the skipped partition. Those two partition types are orthogonal in my approach, but they could be unified in the future, when encoder and decoder support for H.264's built-in partitioning features, namely FMO and ASO, receive more attention.

When the decoder decides to skip a skipping partition, exactly that area of the frame covered by the skipping partition is replaced. The replacement image data is patched together from the replacement partitions in that area as illustrated in Figure 18. By evaluating the motion vectors from the leaves of the quadtree, content is copied from the reference frames. When the motion vector exceeds the boundaries of the reference frame, it is clipped to the closest location within the frame.

This replacement is supposed to be considerably faster than the actual decoding. Figure 19 gives a feeling for the magnitudes of execution

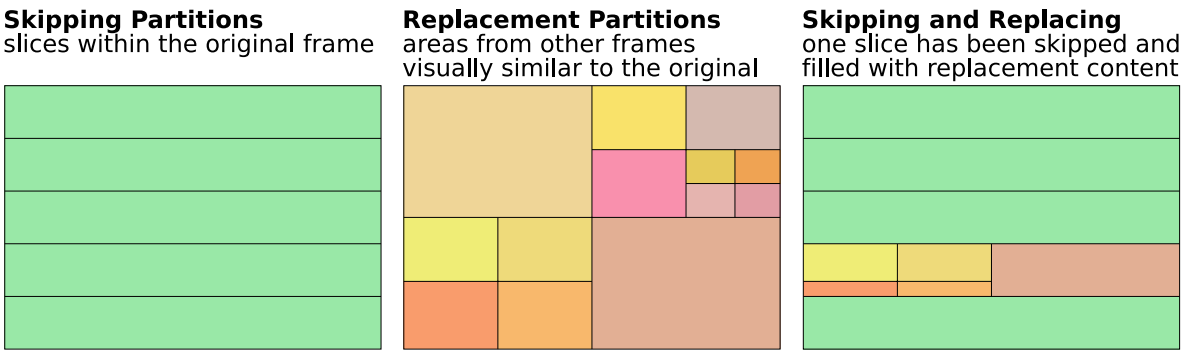



Figure 18: Skipping partitions and replacement partitions

times for full decoding compared to replacing. In average, replacing is 7.54 times faster than decoding and it has proven hard to optimize this further, as the replacement is mostly memory bandwidth limited. Because the scheduler will need both execution times, a replacement time prediction complements the decoding time prediction. This prediction is done similarly to the decoding time metrics discussed in Section 3.1, with the macroblock count of the skipping partition as the only metric, because the memory copying is by far the dominant task here. Figure 20 shows the quality of this metric. There are runaway values and mispredictions, but as this prediction basically tries to estimate memory access speed, I assume they are caused by unfortunate memory accesses, which are hard to predict more accurately.

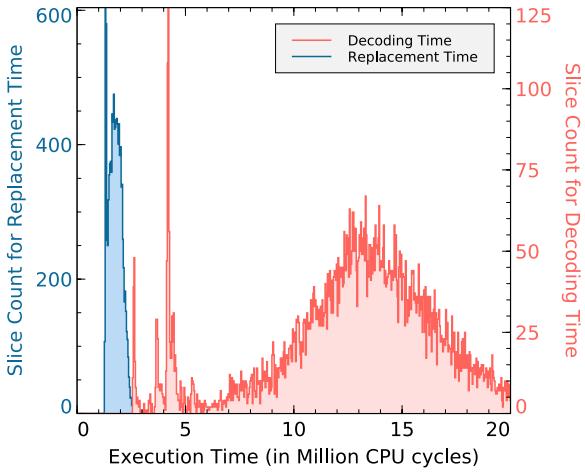



Figure 19: Decoding time and replacement time histograms, measured over BBC video on an AMD Sempron 2200+ (1.5 GHz)

Of course, when such a skipping and replacement happens, the resulting image will be different from the fully decoded original. To make a sensible decision on which parts to skip, the scheduler needs information about this quality loss. Therefore, once the preprocessor has built the quadtree, it will perform a replacement for each skipping partition individually and measure the error between replacement and original. These quality loss values are stored in sideband data for each skipping partition.

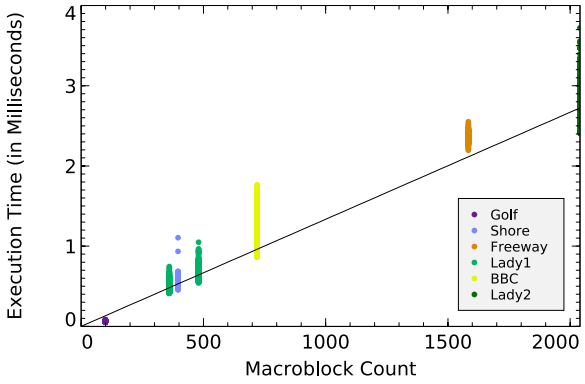



Figure 20: Execution time estimation for slice replacement

### 3.2.6 Quantifying Quality Loss

We have seen that throughout the partition replacement analysis, the preprocessor needs to calculate the quality loss between images multiple times. This comparison is a task for the SSIM algorithm, which I introduced in Section 2.4. There I mentioned two simplifications for SSIM that will reduce the computational load to speed up preprocessing:

- Instead of considering all three components of the Y'$C_BC_R$ colorspace, calculate SSIM for the Y' (luma) component only.
- Instead of considering all possible positions of the local SSIM sliding window, select only a portion of them.

The first simplification of omitting the chroma information has a moderate effect on the accuracy of SSIM as proven by Figure 21. The peak error being greater zero shows that Y'-only results are generally higher, presumably because the chroma components have less structure than the luma component. For a Y'$C_BC_R$ SSIM, Y' is only weighted with 0.8 compared to 1.0 for Y'-only. Due to chroma subsampling, the chroma planes are each only a quarter the size of the luma plane. Therefore, the achievable reduction in computation time is only $2/3$, so I decided not to take the path of ignoring the chroma planes. I think the loss in precision is not adequately translated into speedup and I do not want to disregard color information completely. While this may not be a problem for natural images, where color structure usually follows brightness structures, it may be problematic for synthetic images.

I tested the effect of the second simplification by introducing a precision value between 0 and 1 in the SSIM calculation. Sliding window positions are then selected randomly, with the precision value representing the probability for a specific position to be selected. Using various different precisions to calculate SSIM yields the results

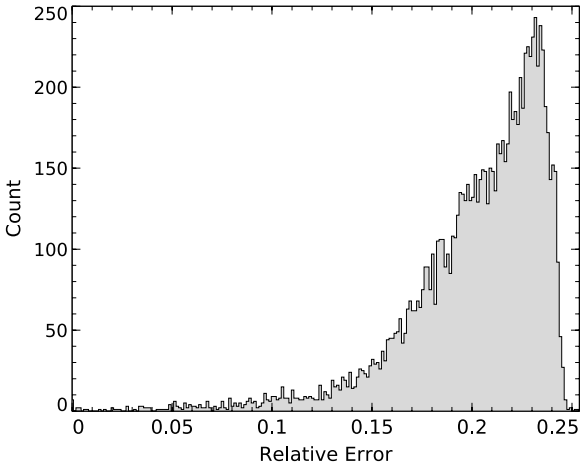



Figure 21: Histogram of the relative difference of Y'-only SSIM results compared to Y'$C_BC_R$ SSIM results

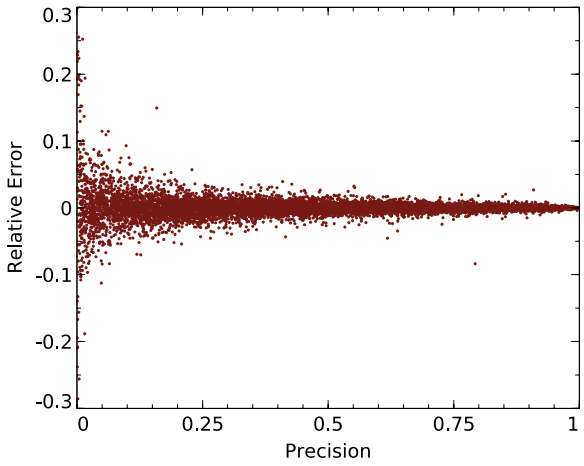



Figure 22: Relative error of imprecise SSIM calculation

presented in Figure 22. Because the execution time of SSIM decreases linearly with the precision, I chose a relatively low precision of 0.05, resulting in computation time being reduced down to 5 %. The overall error of a SSIM with this precision against a full-precision SSIM result can be seen in Figure 23. The standard deviation of the relative error is 0.031, which I consider acceptable.

Combining the findings from Section 2.4 with the ones above, the final quality loss metric which will be used throughout the remainder of

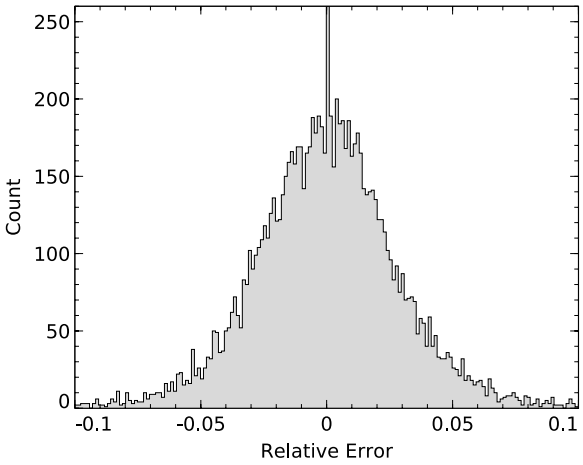



Figure 23: Relative error histogram for a SSIM precision of 0.05

this thesis is therefore

$$\begin{aligned} 1- \big(&0.8 \cdot \mathrm{MSSIM}_{\mathrm{Y'}}(\underline{x}, \underline{y}) \\ &+\ 0.1 \cdot \mathrm{MSSIM}_{\mathrm{C_B}}(\underline{x}, \underline{y}) \\ &+\ 0.1 \cdot \mathrm{MSSIM}_{\mathrm{C_R}}(\underline{x}, \underline{y})\big) \end{aligned}$$

with $\mathrm{MSSIM}(\underline{x}, \underline{y})$ being the mean SSIM index for a single image component, which is in turn calculated as a scaled sum of local SSIM values over a set $W$ of selected sliding window positions. $N$ is the amount of pixels in that image component. Each window is selected with a probability of $p = 0.05$:

$$\mathrm{MSSIM}(\underline{x}, \underline{y}) = \frac{1}{Np} \sum_{W} \mathrm{SSIM}(\underline{x}_{local}, \underline{y}_{local})$$

This calculation is similar to [33].

Further speed enhancement is possible by exploiting the locality of SSIM: When interesting changes between the two images are known to occur only within a confined region, a rectangle of interest allows to only select sliding window positions within that region. This locality is a specific property of SSIM and might not apply to other quality loss metrics. Therefore, my thesis does not rely on this behavior.

Because of the sliding window selection being random, I also considered, whether it is necessary to special case small rectangles of interest down to the size of one macroblock to avoid the situation of no sliding window being selected at all. However, with a sliding window size of 8×8 pixels, the chances of even a single pixel not being covered by any sliding window are small: $(1 - 0.05)^{8 \cdot 8} = 0.0375$, chances of an entire 16×16 pixel macroblock being ignored are minuscule: $(1 - 0.05)^{(8+16-1)(8+16-1)} < 10^{-11}$. As this probability roughly results in an average of one completely ignored macroblock every 35 days of continuously running 25 frames per second high definition video, I decided that special casing is not necessary.

### 3.2.7 Adaptive Subdivision Threshold

The quadtree subdivision algorithm given in Section 3.2.3 left open, what increases in quality loss are considered acceptable. The algorithm adaptively cuts off subnodes by comparing the quality loss of finer and coarser subdivisions and choosing the coarser one, if the quality difference is acceptable. This threshold balances the replacement quality against the size of the sideband data: A too low threshold will result in many quadtree subdivisions for the preprocessor to store without proportionate benefit, whereas a higher threshold results in a coarse subdivision with potentially too few motion vectors to properly approximate the motion in the frame.

As I assumed the quality loss metric to behave linearly (see Section 2.4), the quality decrease for a node should be weighted by the size of the area it covers. This area shrinks exponentially with increasing depth, so the threshold should shrink accordingly. In the quadtree, the number of nodes per depth level is $\left(2^{depth}\right)^2$, so the size of one node is proportional to $1/\left(2^{depth}\right)^2 = 1/2^{2 \cdot depth}$. Therefore, I use a threshold of $0.01/2^{2 \cdot depth}$ to constitute an acceptable quality loss for the SSIM metric. The threshold factor of $0.01$ has been determined empirically, but the magnitude of my choice is supported by Figure 24. The effect of other threshold factors are

shown in Figure 26, with the different resulting sideband data sizes in Figure 25.

It is visible that the chosen threshold factor of $0.01$ uses sufficiently fine subdivision along the areas of motion at the tips of the crystals while still maintaining an overall low node count resulting in moderately sized sideband data. The higher threshold of $0.1$ subdivides too coarsely to approximate the frame's motion adequately. The lower threshold of $0.001$ starts to subdivide areas of the background with no motion that can be represented well enough with larger nodes, resulting in an unnecessary increase of sideband data size. Threshold $0.0001$ applies an almost full subdivision, further increasing sideband data size. This visual inspection should suffice to justify my choice of $0.01$ as the threshold factor.

In addition to the additive threshold, I also tried a multiplicative threshold, where not the difference between the higher and the lower quality, but their ratio would be compared to the threshold. This value would not have to be weighted by the size of the node's area, because the ratio is automatically relative. I used a threshold of 1.7, meaning that an increase in quality loss of 70 % would be tolerable for nodes to be cut off. This might appear quite high, but I determined the value so that the sideband data size would be roughly the same as for the chosen additive threshold of $0.01/2^{2 \cdot depth}$. It is 12.0 %, which is about the same as the 12.3 % for $0.01$ in Figure 25, but Figure 27 shows that the resulting visual quality is worse. It appears that the relative nature of the threshold – high initial error leading to high tolerable error, low initial error leading to low tolerable error – distributes the subdivision pattern evenly across the frame, not particularly following the areas of motion. This leads to unnecessary subdivisions in low-error areas as well as missing subdivisions in high-error areas. Therefore I dropped the idea of a multiplicative threshold in favor of the additive threshold $0.01/2^{2 \cdot depth}$.

### 3.2.8 Summary of Partition Replacement

I have developed a strategy of dividing the frame into skipping partitions and replacement partitions. I formulated an algorithm in Section 3.2.3 to exploit existing motion vectors when creating a quadtree to describe the replacement partitions. The soundness of the algorithm was supported by a discussion of encoder behavior. The scheduler can use the replacement partitions to skip decoding of slices to save execution time and a quality loss metric enables the estimation of the error introduced in doing so. But so far, the effect of such a replacement has only been examined for the frame in which it takes place. The upcoming section deals with that limitation.

## 3.3 Error Propagation Estimation

Until now, I examined the error caused by skipping and replacing a slice strictly within the frame directly affected. But today's decoder algorithms in general and H.264 in particular draw a large part of their compression efficiency from the exploitation of inter-frame redundancy by using temporal prediction to encode frames. This causes errors in one frame to be propagated into other frames, which then in turn cause further frames to have errors. This way, an error introduced in one frame can affect any number of frames decoded later.

In addition, H.264 uses spatial prediction to exploit intra-frame redundancy, which could lead to errors in one slice being propagated into other slices of the same frame, spreading the error over a larger portion of the current frame, which also increases the pollution of future frames. However, I have not observed this effect during my analysis, so I will ignore it. I assume that encoders deliberately avoid cross-slice spatial prediction, because they want to allow decoders to decode multiple slices simultaneously when multiple CPU cores are available.

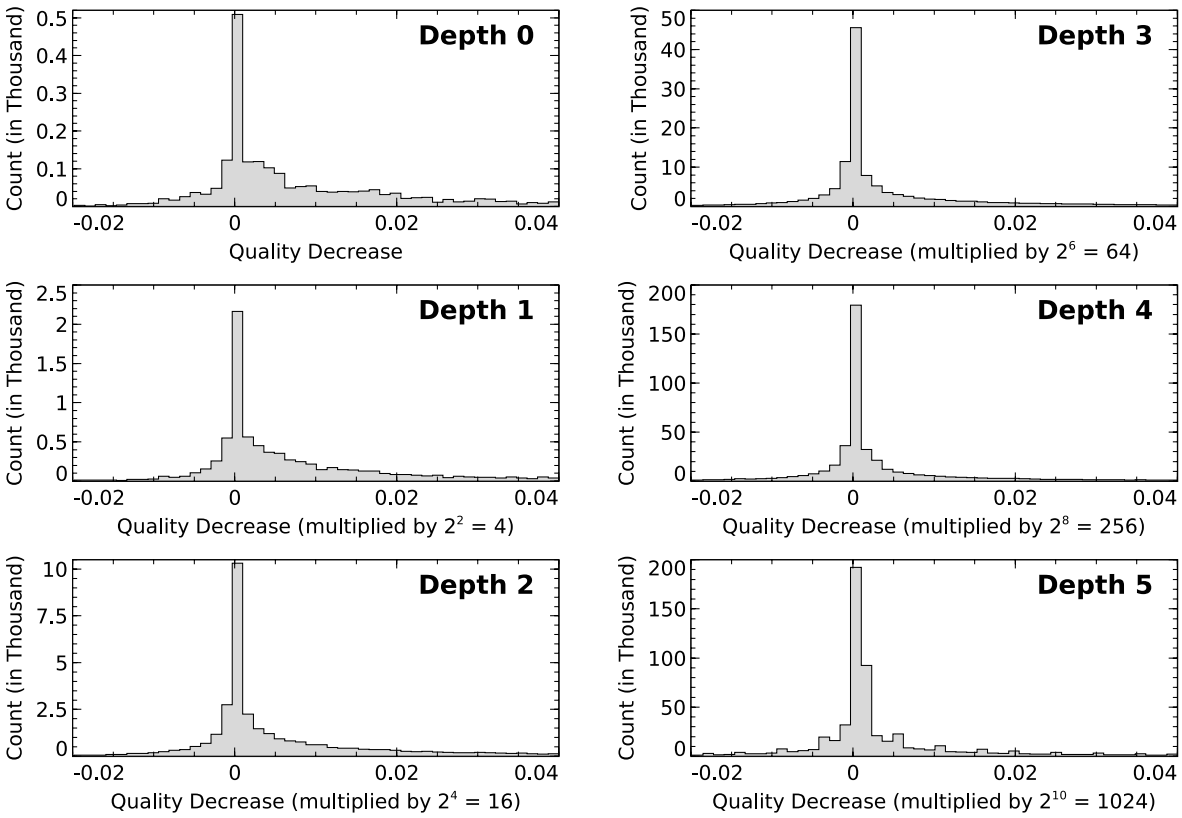



Figure 24: Quality loss difference histograms for cutting off nodes of higher depths, leaving one node of the given depth as a leave. The abscissae have each been multiplied by $2^{2 \cdot depth}$ for comparability. Note that the individual histograms have similar properties, which indicates that the weighing factor $2^{2 \cdot depth}$ is correct.

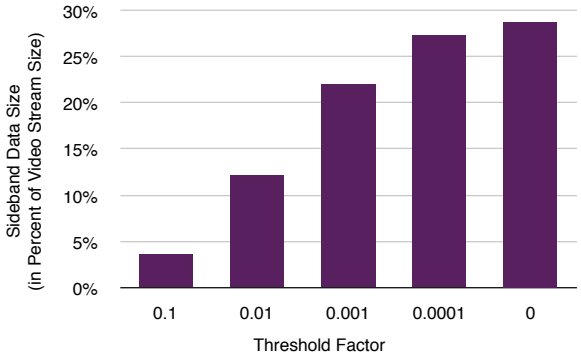



Figure 25: Sideband data sizes in relation to the video size (BBC video) for different threshold factors

### 3.3.1 Estimation or Measurement

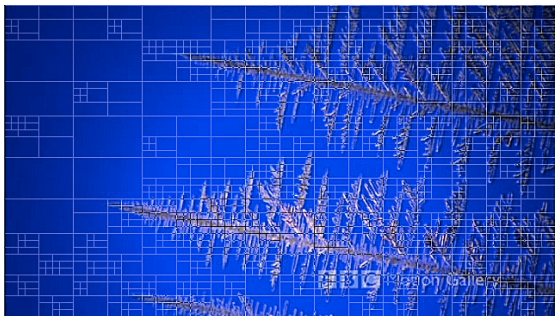
Of course the most accurate way to quantify the propagated error is to measure it similarly to the error directly induced by replacement (see Section 3.2.5). But what was straightforward for this direct error is a lot more complex for the propagated error: Errors are potentially propagated over great distances, only an IDR frame definitely inhibits all propagations. Therefore, the current slice's error can depend on the error of every slice back to the previous IDR. The number of those slices can reach 100 and more and is generally unbounded. Every single one of those slices could be skipped or not, which would change the error inflicted on the current slice. So for a thorough error measurement, given a slice that is $n$ slices away from the previous IDR,

The original sequence of growing ice crystals:

Threshold Factor 0.1:

Threshold Factor 0.01:

Threshold Factor 0.001:



Threshold Factor 0.0001:

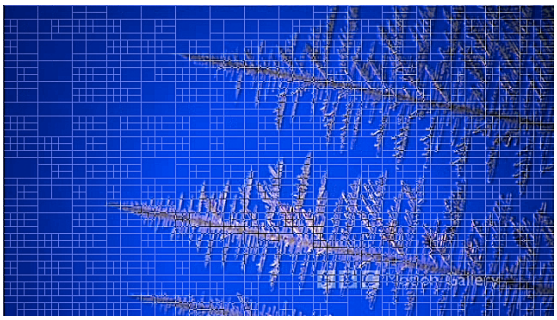


Figure 26: Visual effect of different subdivision thresholds for BBC video. The top row gives an idea about the movement: The ice crystals are growing from the lower right corner toward the upper left corner. Motion vectors encode the growth at the tips of the crystals. The four images below that visualize the quadtree subdivisions of the same video frame for various threshold factors.

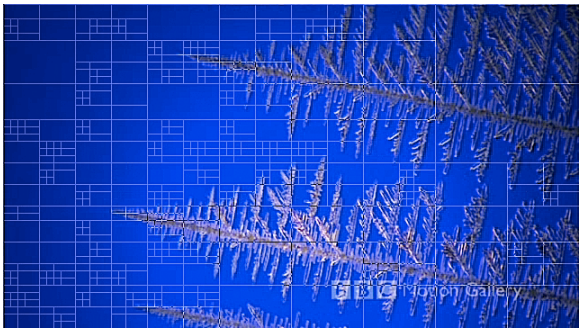


Figure 27: Visual effect of a multiplicative threshold of 1.7. Subdivision is visualized for the same frame as in Figure 26

$2^n$ different slice skipping patterns would have to be simulated and measured. This procedure would be repeated for every slice. It is quite clear that this way of measuring the error is completely infeasible.

Therefore I am using a different approach, trying to estimate the error instead of measuring it. For this, I will analyze and quantify a single propagation step. That is: Quantify how errors propagate into the current slice from those slices used by it directly. With this step, any propagation path can be predicted as proven by the following well-founded induction:

Let $\mathcal{S}$ be a set of contiguous slices in which error propagation occurs. $\mathcal{S}$ is totally ordered by the binary relation $D$ representing the decoding order, hence $(s_1, s_2) \in D$ if and only if $s_1$ is fully decoded before $s_2$ is fully decoded. $\mathcal{S}$ shall be chosen in a way that all propagation is confined to slices in $\mathcal{S}$. Such a selection is possible because IDR frames naturally inhibit any propagation, so if for example $\mathcal{S}$ starts with the first degraded slice and ends with the last slice of the frame before the next IDR, the requirement is met. I will now prove that, given a quantification for a single propagation step, the error of every slice in $\mathcal{S}$ can be estimated.

**Induction Basis:** The error estimate of every $D$-minimal element in $\mathcal{S}$ must be known. Because $D$ is a total order on $\mathcal{S}$, there is only one $D$-minimal element, which is the slice in $\mathcal{S}$ first to be decoded. Because there is by construction no error propagation from a frame outside of $\mathcal{S}$ into this slice, it is either error-free, in which case the error estimate is trivially 0, or it is a slice which will be skipped and replaced. In this case, the error is directly inflicted by the replacement and this error has been measured and is provided with sideband data as stated in Section 3.2.5. This value is used as the estimate.

**Induction Premise:** For an arbitrary but fixed slice $s$, the error estimates of all slices $r$ are known, if $(r, s) \in D$ (slice $r$ is decoded before slice $s$).

**Induction Hypothesis:** The error of $s$ can be estimated.

**Induction Step:** The decoding of slice $s$ uses other slices as references. These can already be degraded, which is the basis for error propagation. However, as assumed previously, the amount of error propagated in a single step from the reference slices into the current slice is known. Thus, if we had an estimate of the error of those reference slices, the error of the current slice could be estimated as well. Because any reference slice $r$ must be fully decoded before $s$, $(r, s) \in D$ holds. Therefore, the induction premise yields the required error estimates of the reference slices. Using the single propagation step quantification now gives an error estimate for $s$, which proves the induction hypothesis. Note that the slice $s$ itself could be skipped and replaced, in which case the error inflicted directly by the replacement is accumulated to the propagated error. This will be discussed in Section 3.3.3.

Applying the principle of well-founded induction leads to the conclusion, that the error of every slice in $\mathcal{S}$ can be estimated and since $\mathcal{S}$ can encompass an entire video without violating any requirement, the errors of all slices in any video

can be estimated, given a quantification for the single step propagation.

### 3.3.2 Single Propagation Step

I can now reduce the problem of arbitrary propagation paths to just analyzing and estimating propagation to a slice from those slices it directly references. The key property of referencing potentially degraded slices is that the error from that reference slice is copied to a certain degree into the current slice. The approach I am going to pursue is to estimate that error in a straightforward way by checking motion vectors to see, what fraction of a particular reference slice's area is used by the current slice. The referenced area of one motion vector can differ in size due to motion vector subblocking, so some vectors will contribute a smaller area, others contribute larger ones. As these areas may cross slice boundaries, I always check for each pixel of the referenced area to what slice it belongs to account the areas coming from different reference slices separately. Another peculiarity to consider is that for B-slices, every macroblock may use two different references, which are overlaid in a weighted fashion. Disregarding the actual weighing factors, I multiply each contributed area with 0.5.

Of course, the actual temporal prediction is a lot more complex than just simple copying of reference frame data. Besides the subpixel interpolation and B-slice weighing, the result is overlaid with a block of correction samples and is subject to H.264's in-loop deblocking filter. But I hope that the simple approach of using motion vectors to determine area fractions being copied will be sufficient.

The result of this motion vector analysis is a table per slice that stores for each slice of each reference frame the area fraction that is copied into the current slice. The actual error estimation for a slice is then calculated by multiplying the error estimates of the reference slices with their respective area fraction and summing up these individual contributions. Remember that the error of the reference slices is known by induction premise. Figure 28 illustrates this with example tables.

**Error Behavior Assumptions:** It is clear that, using such a simple calculation for a process as complex as error propagation, I must make assumptions on error behavior. First, I derive the error contribution from a single reference slice by multiplying the reference slice's error estimate with the fraction of the area being copied. This assumes that the error is distributed equally over the reference slice, which is a pretty strong assumption. By skipping and replacing 100 randomly selected slices, obtaining the resulting degraded frames with the propagated error, I calculated an accumulated SSIM error map, which can be seen in Figure 29. This map shows, that the error does not follow any obvious patterns, like becoming smaller toward the frames' borders. Although the slice structure is visible in the map, the error does not clearly follow it either. Instead, the error more dominantly follows structures in the image content, which is proven by the clearly visible "BBC Motion Gallery" logo in the bottom right, with its very distinctive i-dot. So, although knowing that the error is not necessarily equally distributed, but might concentrate somewhere, I found no easy way to determine where such concentrations are. Without such a-priori knowledge, uniform distribution is the best assumption available. Of course, one way to increase the accuracy of the error localization is to use smaller slices. It remains open for future research to derive more efficient ways of localizing errors.

The second assumption made is that the individual error contributions can simply be summed up to a total error estimate. This works because the contributions are made up of area fractions of the current slice, which are spatially independent of one another and because the SSIM index has a linear behavior as discussed in Section 2.4.4.

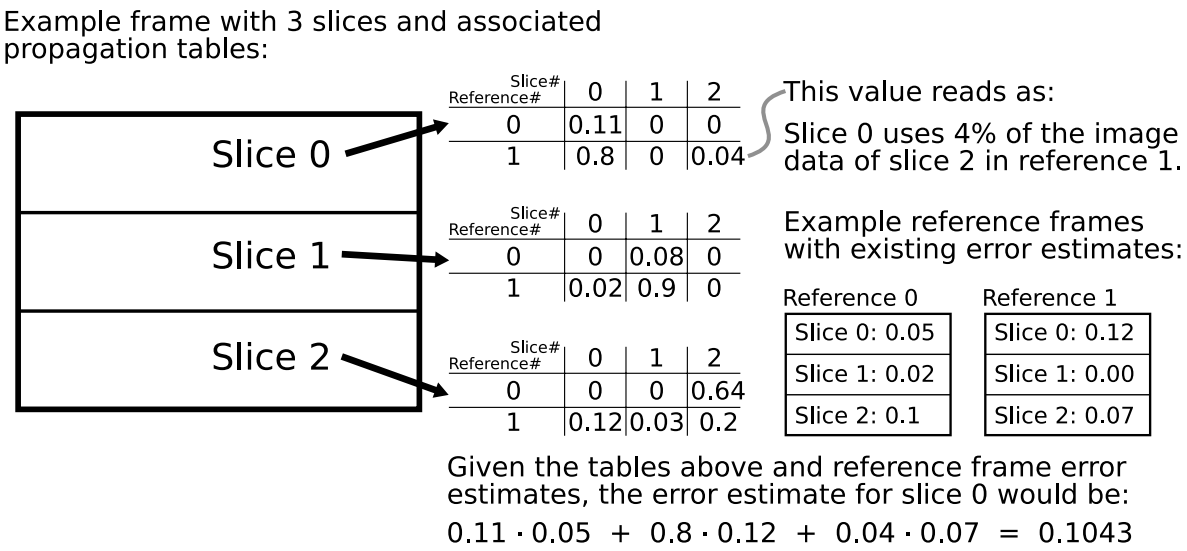



Figure 28: Example propagation step calculated from propagation tables and reference frame error estimates

**Estimation Quality:** I randomly selected 100 slices of the BBC video to be skipped and replaced and calculated the actual SSIM quality loss and the estimated quality loss for every frame. The absolute error of this estimation can be seen in Figure 30. The average absolute estimation error is -0.0002 at a standard deviation of 0.0011. As the measured actual SSIM quality loss ranges from 0 to 0.027, the estimation is reasonably good, despite the approximations mentioned.

**Correction Factor:** The average absolute error is negative, so the estimate is typically smaller than the actually measured quality loss. I tried compensating for this with a correction factor to be applied to the area fractions in the propagation tables: Each such table entry would simply be multiplied with a fixed factor. If this factor is slightly larger than 1, the table entries would all grow a bit and thus, the final quality loss estimate would grow with them. The problem is therefore finding a single, fixed factor, which is optimal in regard of reducing the absolute value of the average absolute error. Because there is a monotone bijection between the absolute value and the squared value, this problem is equivalent to finding a factor, which minimizes the mean squared error. The problem is therefore one-dimensional, but non-linear. Figure 31 visualizes the function to minimize. As expected, the minimum of this function is slightly larger than 1: I calculated a minimum of 1.01417913 by searching for the root of the first derivation using the Pegasus algorithm [44]. Unfortunately, this factor reduces the mean squared error by only 3.66 %. Given this rather small improvement and that this factor may depend on the video, I gave up on this idea. Future research could analyze more elaborate ways to apply correction to improve the estimation.

**Error Diminishment:** Figure 32 shows an exemplary error propagation for the skipping of one slice. Although the initial error is quite high, it can be observed that the error diminishes over time, which is caused by three effects:

- The decoding of a frame can use any frame currently available in the reference frame buffer. If a frame preferably uses references with smaller errors or no error at all, it will

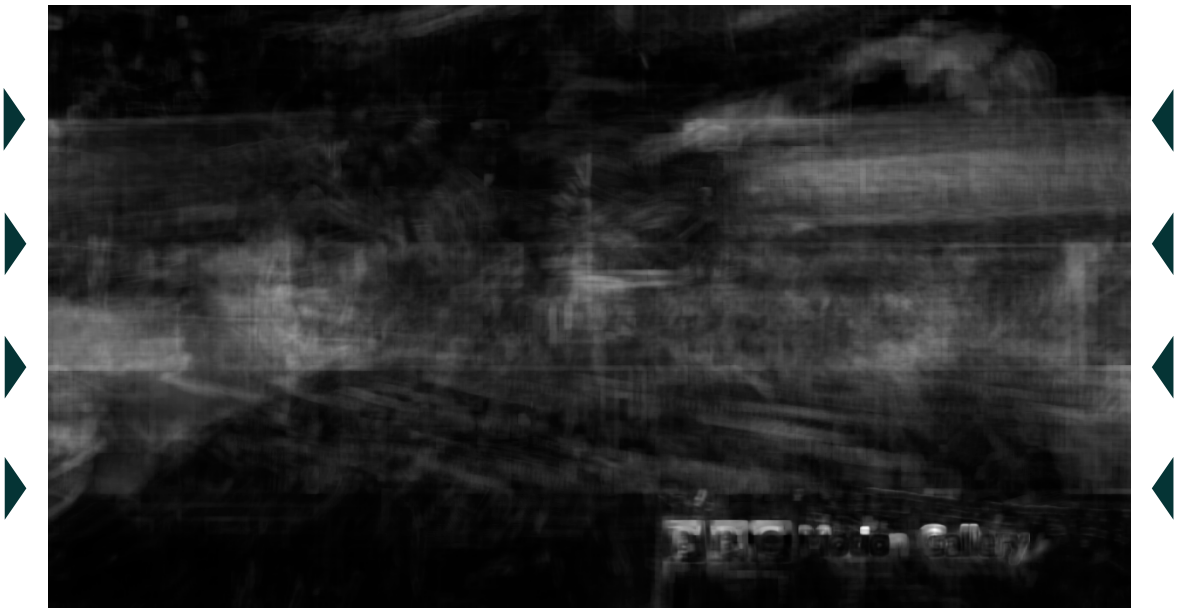


Figure 29: Accumulated SSIM error map of the error propagation processes from 100 randomly selected skipped and replaced slices. For each of those 100 slices individually, the entire BBC video would be compared to the undegraded original. This results in an SSIM error map for each frame, which were added up to the picture shown here. White represents the maximum error, black an error of zero. Therefore, dark areas were generally less affected by errors than bright areas. The arrows to the left and right mark the slice boundaries, which are faintly visible in the image.

in turn show a smaller error or even be error free.

- The reference frame buffer is of limited size. Therefore, frames are regularly pushed out of this buffer as new frames are added to it. If a frame with a high error is removed and a frame with a lower error is added, the potential for future frames to show errors is decreased. On the other hand, Figure 33 proves that some frames are alive as references for a long time, so that errors can also be preserved in the reference buffer.

- Even the intercoded slice types (P- and B-slices) can contain intracoded macroblocks. Because such macroblocks do not need any reference frames, they usually reduce the error. The amount of intracoded macroblocks is the reason, why the area fractions represented in the propagation tables for each slice do not necessarily sum up to 1 (see Figure 28).

The final point is particularly advantageous when remembering encoder behavior as described in Section 3.2.4: Encoders will often represent objects of high motion with more intracoded macroblocks, as the blurriness makes intracoding more effective than intercoding (see Figure 16). The result is a less accurate representation of the frame's motion in the motion vector quadtree used during replacement. So whereas low motion leads to a good replacement, high motion will generate rougher replacements due to more intracoded macroblocks. But on the other hand, high motion with more intracoded macroblocks has the advantage that the error diminishes more quickly.

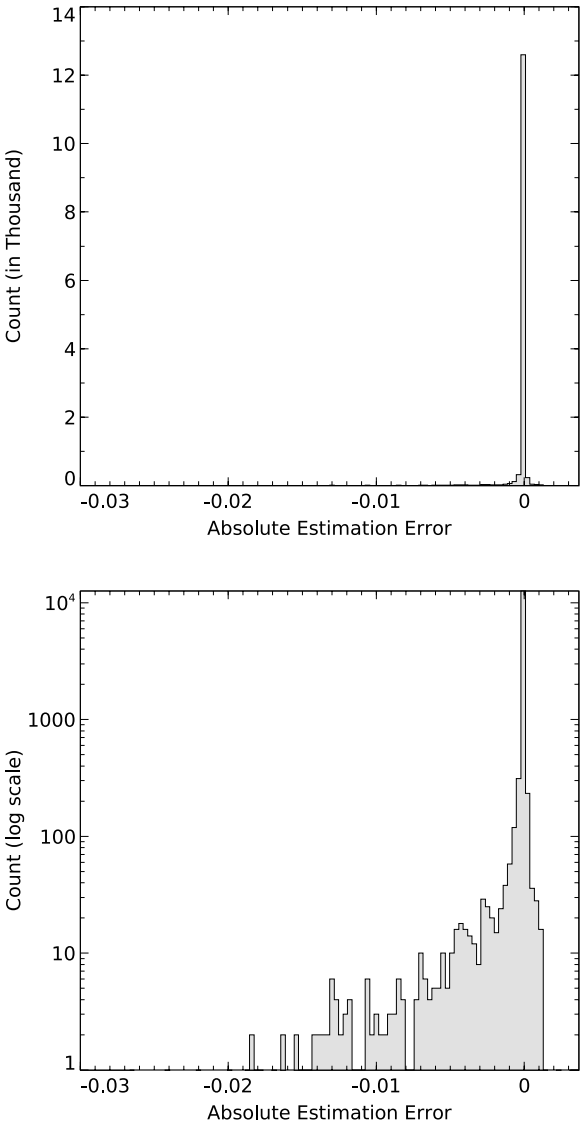



Figure 30: Absolute error histogram for the estimation of quality loss propagation of 100 randomly selected skipped and replaced slices. The plot is presented with both a linear and a logarithmic ordinate. The linear plot gives a better feeling for the accuracy of the method, whereas the logarithmic plot shows the distribution of the lower contributions.

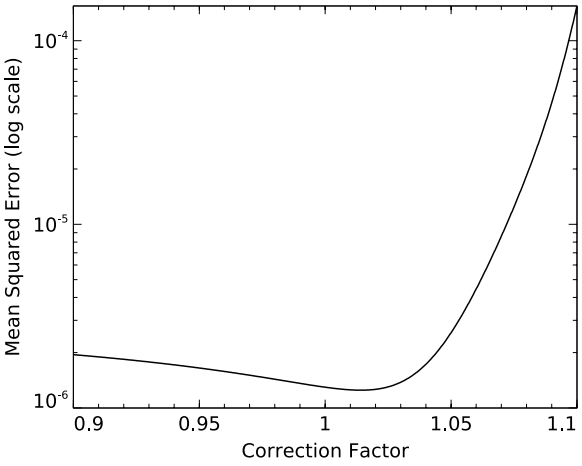



Figure 31: Mean squared error depending on the correction factor. Note that the ordinate is in logarithmic scale.

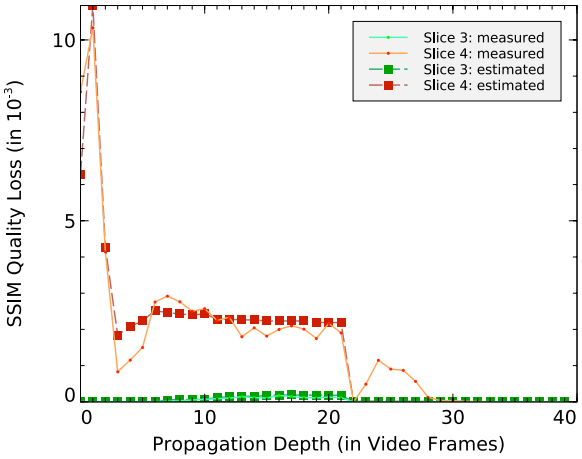



Figure 32: Measured and estimated error propagation for the skipping of one slice. The unmentioned slices 0, 1, and 2 of this 5-slices-per-frame video (BBC) showed no error in either estimation or propagation.

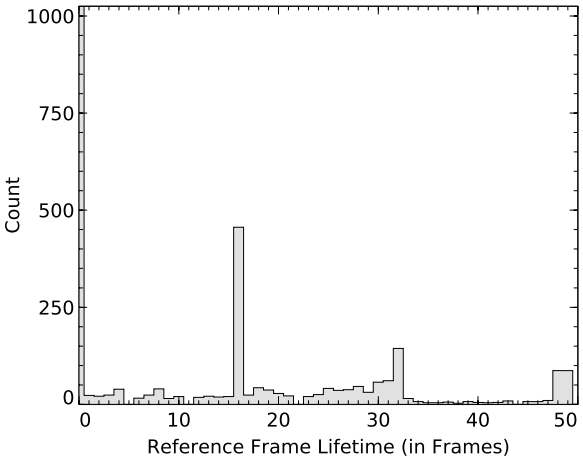



Figure 33: Histogram of the duration a frame is kept in the reference frame buffer. For the chosen Lady1 video, a high number of frames is never used as reference, but several frames are long-living. This behavior highly depends on encoder choices.

### 3.3.3 Error Accumulation

Now that I have thoroughly discussed a single propagation step, I will verify error propagation over an entire video, including error accumulation from multiple skipped slices in this section. In the induction-based explanation of error propagation, I already mentioned the need for differentiation between a single propagation step, where the slice in question is fully decoded, and a single propagation step, where the slice is skipped and replaced. So far, I have only analyzed the situation of a slice being fully decoded, but receiving an error from existing errors in reference frames. Those errors are initially caused by a slice being skipped and replaced. But what happens, if the replacement of a slice already uses degraded reference frames, has not been investigated yet. This basis for error accumulation is discussed now:

As the replacement copies data from various reference frames to fill in the skipped slice, it is obvious that I will use the same approach of using area fractions to quantify, how much the error of the reference in use will propagate into the current slice. The most accurate way to do this would be to check for each pixel of the replacement from which reference slice its value is copied and then account this reference slice's error accordingly. However, to speed up the replacement, this is approximated by checking only one pixel per macroblock.

This error propagation estimate using area fractions will only tell, how much the degraded replacement differs from the real replacement. But what is needed for error accumulation to be estimated properly is the difference between the degraded replacement and the original. During the creation of the motion vector quadtree, we already measured the difference between the undegraded replacement and the original (see Section 3.2.5). To obtain the full difference, I add both difference contributions, which should work thanks to the linearity of SSIM (see Section 2.4.4).

With error accumulation, we have all the tools for the complete error propagation estimation algorithm at hand:

1. If a slice is not skipped, its error estimate is calculated by multiplying all area fractions with the error estimate of the respective reference slice.
2. If a slice is skipped and replaced, its error estimate is calculated by multiplying approximated area fractions (only one pixel per macroblock is examined to increase speed) with the error estimate of the respective reference slice and adding the error directly induced by the replacement, which is available via sideband data from earlier measurement.

To evaluate this procedure, I ran the entire BBC video through the error propagation estimation, skipping randomly selected slices at an average rate of one slice every other frame. The actual quality loss was measured by comparing each slice to its original. It ranges from 0 to 0.067.

Figure 34 shows a histogram of the absolute error of the estimate. With an average absolute error of -0.0020 at a standard deviation of 0.0042, the estimation works quite well, despite all approximations made. Figure 35 shows the estimation error plotted over time to show that the estimation does not get worse as the video progresses, but fluctuates and recovers throughout the video's runtime.

The achieved quality supports the conclusion that error estimates can be made by

- multiplying area fractions and
- summing up individual errors to form a total error estimate.

This conclusion may be valid only for the specific type of errors considered here, which are small compared to errors caused by severe outages in the video stream like the loss of entire slices or frames. Such small errors are more likely spatially disjoint and thus do not interact too strongly with one another, reducing effects like error amplification or annihilation.

But using this propagation estimation to account for each slice's error and propagate it to a potentially large number of future slices is still far too expensive to do online during scheduling, so a much more lightweight representation of error propagation is needed. In the next section, I build on the above findings to explain, how such a simplification is done so the scheduler does not need to iterate over large amounts of sideband data.

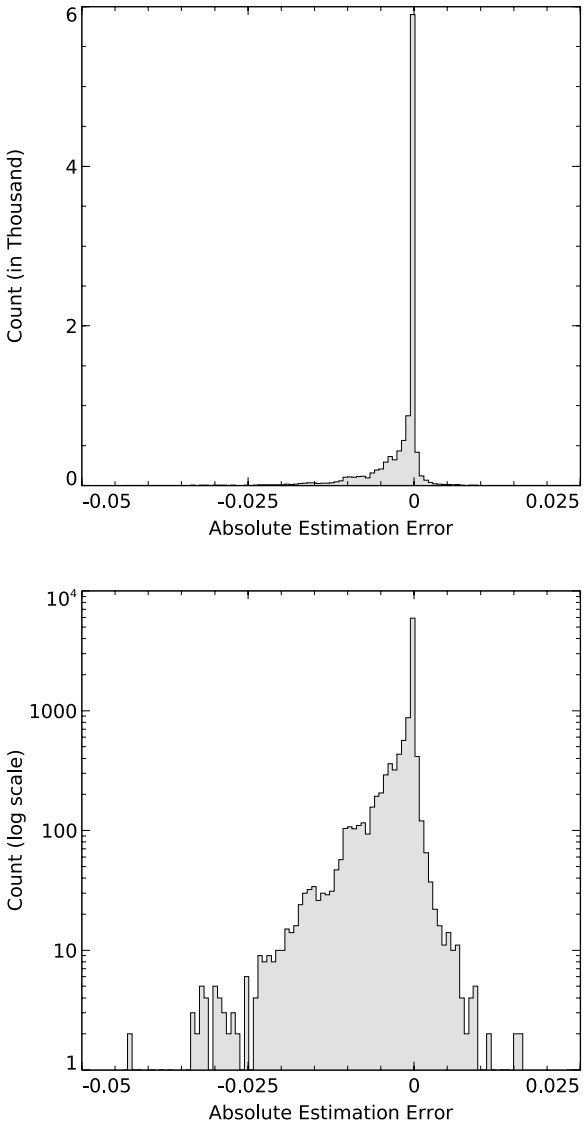



Figure 34: Absolute error histogram for the estimation of quality loss propagation including quality loss accumulation. About one randomly selected slice every other frame was skipped. The plot is presented with both a linear and a logarithmic ordinate. The linear plot gives a better feeling for the accuracy of the method, whereas the logarithmic plot shows the distribution of the lower contributions.

### 3.3.4 Error Emission

I have discussed, how decoding errors propagate from a reference slice decoded earlier into the current slice, so the point of view was from the current slice into the past. Now I am going to look from the current slice into the future and explain, how to provide an easy quantification of how errors propagate from the current slice

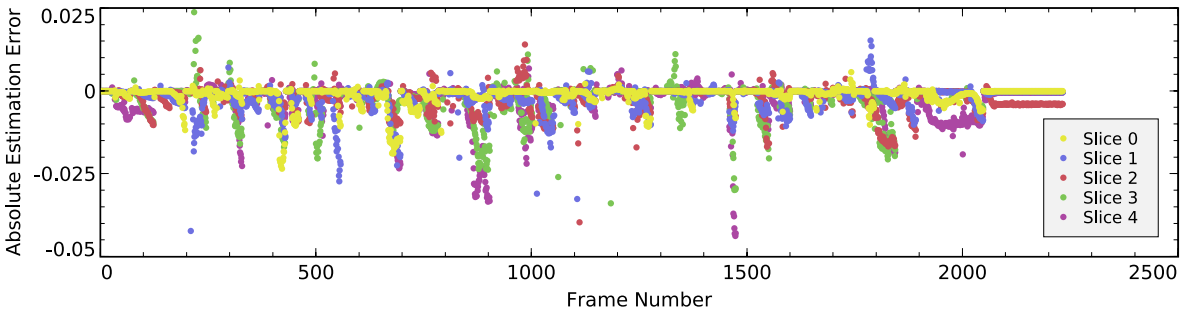



Figure 35: Absolute error of the quality loss estimation plotted over the video playback time. Each frame of the used BBC video contains five slices.

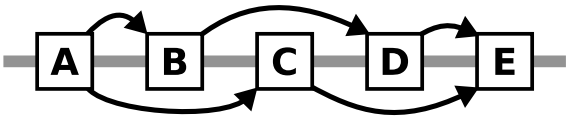



Figure 36: Example for error propagation to future slices. Slices A to E are given in decoding order with arrows indicating error propagation.

into any future slices. To differentiate between those two concepts, I will call them error immission and error emission, with error propagation being the superordinate term. Error immission has been the subject of the previous sections. Error emission will follow in this section.

The way an error is propagated is the same for immission and emission, only the standpoint differs, so the single propagation step quantification using area fractions still applies. However, for immission, those fractions were used against reference slices, now I am going to use those fractions against future slices: The idea is to condense all errors in future slices that will be caused by degradation of the current slice into one single number. According to the single propagation step, every error in a future slices is caused by directly or indirectly copying image content from the current slice into said future slice.

As the example in Figure 36 illustrates, the error from slice A is propagated along multiple paths to the slices B to E. For this example, let us assume that propagation ends at slice E. Integrating over time, the total error caused by degradation of A is the sum of all errors induced into B to E. The individual propagation steps are already described by area fractions, so that for example the error $e_B$ in slice B is the error $e_A$ in slice A multiplied by the respective immission factor $i_{AB}$ given by the area fraction.

$$e_B = e_A \cdot i_{AB}$$

The total error caused by A ($E_A$) is therefore:

$$\begin{aligned}
E_A &= e_A + e_B + e_C + e_D + e_E \\
e_B &= e_A \cdot i_{AB} \\
e_C &= e_A \cdot i_{AC} \\
e_D &= e_B \cdot i_{BD} \\
&= e_A \cdot i_{AB} \cdot i_{BD} \\
e_E &= e_D \cdot i_{DE} + e_C \cdot i_{CE} \\
&= e_A \cdot i_{AB} \cdot i_{BD} \cdot i_{DE} + e_A \cdot i_{AC} \cdot i_{CE}
\end{aligned}$$

Notice that my notion of future error includes the initial error $e_A$. This will be beneficial for the scheduler. But the actual error $e_A$ of A is only known at decoding time, so I use $f_A = E_A/e_A$ as the factor to quantify error emission:

$$\begin{aligned}
f_A &= 1 + i_{AB} + i_{AC} + i_{AB} \cdot i_{BD} + \\
&\quad i_{AB} \cdot i_{BD} \cdot i_{DE} + i_{AC} \cdot i_{CE}
\end{aligned}$$

Calculating $f_A$ this way can become very expensive, as the propagation chains are potentially very long and diverse. But reformulating the

problem in an equivalent way leads to:

$$\begin{aligned} f_A &= 1 + i_{AB} \cdot (1 + i_{BD} \cdot (1 + i_{DE})) + \\ &\quad i_{AC} \cdot (1 + i_{CE}) \\ &= 1 + i_{AB} \cdot f_B + i_{AC} \cdot f_C \end{aligned}$$

with

$$\begin{aligned} f_B &= 1 + i_{BD} \cdot f_D \\ f_C &= 1 + i_{CE} \\ f_D &= 1 + i_{DE} \end{aligned}$$

This shows that the emission factors $f$ can be calculated recursively by accounting only for propagations directly starting at the given slice ($i_{AB}$ and $i_{AC}$ for slice A) and multiplying them with the emission factor of the slice directly propagated to ($f_B$ and $f_C$ for slice A). To resolve the recursion, those emission factors have to be known, which is possible because

- all error propagation is inhibited at IDR frames and
- the propagation paths are cycle-free.

Starting at the next future IDR, the error emission factors can be calculated backwards, in the example from E down to A:

$$\begin{aligned} f_E &= 1 \\ f_D &= 1 + i_{DE} \cdot f_E \\ f_C &= 1 + i_{CE} \cdot f_E \\ f_B &= 1 + i_{BD} \cdot f_D \\ f_A &= 1 + i_{AB} \cdot f_A + i_{AC} \cdot f_C \end{aligned}$$

The resulting emission factors are then stored in sideband data. With these values available, the scheduler can multiply the current error of a slice with the given emission factor to quickly get a quantification of how much degradation will be visible to the user. This condenses all the previous results on error propagation in one single and easy to handle number.

In the next sections, I explain how the sideband data is extracted practically and how the scheduling is done with this data.

# 4 Implementation

## 4.1 Video Preprocessor

I now describe the steps I took in implementing a preprocessor for H.264 that extracts sideband data, which will be used later in Section 4.3 to improve the scheduling of the decoding task.

### 4.1.1 Prerequisites

To implement the video preprocessor, I first needed an H.264 decoder I could instrument to export the required data. As I already decided on x264 as the encoder of choice (see Section 3.1), I explored the possibility of using x264's decoder as well. Unfortunately, their decoder appears to be unmaintained and is not even compiled in a default x264 installation. Therefore, I dropped this idea and chose FFmpeg [41], whose avcodec decoder library is a quasi standard in the open source world. I constantly used a recent version from their CVS to always get the latest improvements.

Although the FFmpeg decoder is already quite mature and offers optimized SIMD assembler code for time consuming parts of H.264 decoding, it does not yet support all features of the standard. It especially lacks flexible macroblock ordering (FMO) and arbitrary slice ordering (ASO), but the H.264 Main Profile is fully implemented and I could neither find nor create content using these ordering features. Therefore, the shortcomings of FFmpeg's decoder are clearly outweighed by its high speed and stability, by it is wide use, and by the quality of its maintenance.

Next, I needed an implementation of the SSIM index. The authors of the original SSIM paper provide a Matlab implementation [45], but its use is limited to educational and research purposes and it requires a copyright notice. These restrictions are incompatible with the GNU General Public License [46] and, although FFmpeg is LGPL-licensed, I wanted to retain the possibility of linking with GPL code so no troubles occur when integrating my work in existing media players. Existing C++ implementations of SSIM claim to be derived from the Matlab code, so their licensing is unclear. Therefore I decided to cleanly reimplement SSIM from the original paper [32], which turned out to be less difficult than expected, thanks to the clarity of the description.

### 4.1.2 Instrumentation

To extract all the metrics and other metadata, like reference frame lists, macroblock types, and motion vectors, I instrumented the FFmpeg decoder to export the required information after the decoding of each slice via a `process_slice()` callback function. I decided not to support adaptive frame/field coding (AFF), because it stores macroblocks in a different order and thus introduces error-prone special cases in the handling of slice information without any benefit in the context regarded here.

To determine the metrics for the decoding time prediction (see Section 3.1), I also inserted time measurements at key positions in the decoder code. For that, the `read_time()` function of FFmpeg was helpful, as it provides easy access to low-overhead time measurement by using the timestamp counters on both the x86 and PowerPC architectures. The overall changes to FFmpeg are under 200 lines of added code, most of them being trivial value copying to make data visible outside the scope of the decoder.

The extraction of metadata for the sideband datastream requires the macroblock types and motion vectors for each slice of a frame. But the processing has to be done after the frame has been completely decoded, because I need to calculate the quality loss metric to evaluate possible replacements for parts of the frame. At first, I considered passing arbitrary-shaped image areas to the quality loss metric, so I could process each part individually. But I quickly dismissed this idea, because it would imply that the quality loss metric operates on locally confined

areas. But a quality loss metric that uses the overall structure of the entire frame to quantify local errors is conceivable. Therefore I decided that the quality loss metric is supposed to "see" the same image as the viewer, which is the entire frame. A metric that works locally can be provided for speedup with a rectangular bound that confines the area where the change of interest occurs.

This decision entails storing all per-slice data until all slices of the frame have been decoded. Fortunately, FFmpeg already keeps track of all required data internally and makes it easily usable through the `AVFrame` structure. In addition the `AVFrame` member opaque can be used to attach application specific data to each frame, which is helpful to store information that travels with the frame through H.264's reference buffers.

### 4.1.3 Directly Extracted Data

The following sideband data can be extracted directly from the stream:

- the size of the frame,
- the index of the first and last macroblock for each slice, and
- the decoding time metrics as listed in Section 3.1.3.

This data is gathered within FFmpeg by means of simple counters in the decoder code or it is taken directly from the decoder's internal state storage.

### 4.1.4 Replacement Partitions

To describe the replacement partitions I introduced in Section 3.2 the preprocessor has to build the motion vector quadtree, which is then linearized into the sideband data channel. Although conceptionally clear, this poses a number of practical problems, which I will address here:

**When to build?** First is the question, when to build the motion vector quadtree. As it needs the fully decoded frame to calculate the quality loss properly, the tree has to be built after the last slice of the frame has been decoded. But on the other hand, this step needs the reference frame buffer in the same state it was during the frame's decoding, because the analysis of the motion vectors will require exactly those reference frames. Unfortunately, the decoder reorders the reference frames right after the frame's decoding finishes. This might push now obsolete reference frames out of the buffer and these may be deallocated right away. Therefore, the motion vector analysis for creating the quadtree must be done right between the decoding of the last slice and the frame finalization.

**Coordinate Calculation.** The next interesting implementation detail is the calculation of the boundaries of the region covered by a specific quadtree node. I describe a quadtree node with two numbers: its depth in the tree, starting with 0 for the root node, and the index of the node within this depth. This index is hard to explain. It is the number of nodes with the same depth as the current node that are visited before the current node during tree traversal of a hypothetical fully subdivided quadtree. Figure 37 should help to understand the concept. The order in which the subnodes of a given node are visited during traversal is of course arbitrary, but I chose to visit them in the order of a raster scan.

Given a node's depth and index, we can now imagine the node's region as one field in a regular checkerboard pattern. The row and column numbers of the region can be derived by looking at the binary representation of the index. As illustrated in Figure 38, the even numbered bits form the column number, the odd numbered bits the row number. Once row and column numbers are known, the coordinates of the node's region in multiples of the macroblock size can be calculated by multiplying the row and column numbers with the width of one row or column,

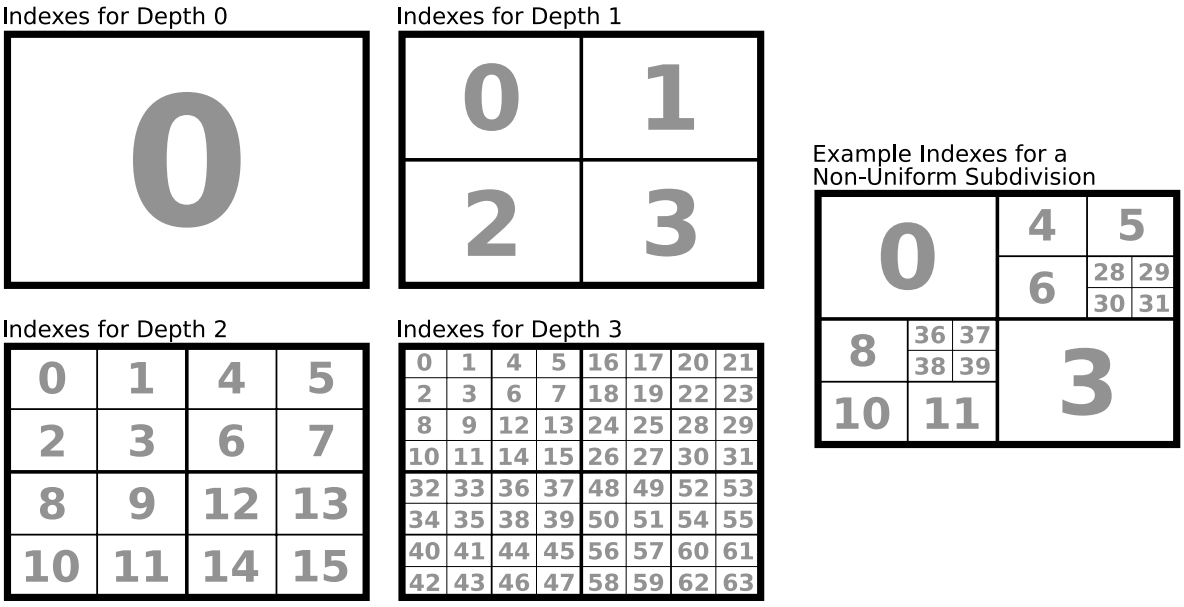



Figure 37: Quadtree node indices. The indices for each individual depth are determined using a fully subdivided quadtree. For any actual nonuniform subdivision, the indices are merely taken from the corresponding locations of the depth planes.

respectively. For a frame of width $w$ and a current tree depth $d$, one column is $w/2^d$ wide. The result is rounded to an integer multiple of the macroblock size. The row number is treated analogously.

**Reference Number Translation.** The next problem arises, when trying to count the reference frames to determine the reference used most often as suggested in my algorithm in Section 3.2.3. FFmpeg exports the reference frame numbers for each macroblock. But those numbers are indices into a slice-local reference buffer, which is refilled with different reference frames and in varying order at the start of each slice to always list the references in the way most suitable for coding efficiency. But because the motion vector quadtree spans an entire frame and therefore multiple slices, those slice-local indices have to be translated into a global numbering scheme. The decoder always keeps reference frames in two global buffers, one for short-term and one for long-term reference frames. I use the indices of frames in these buffers as global

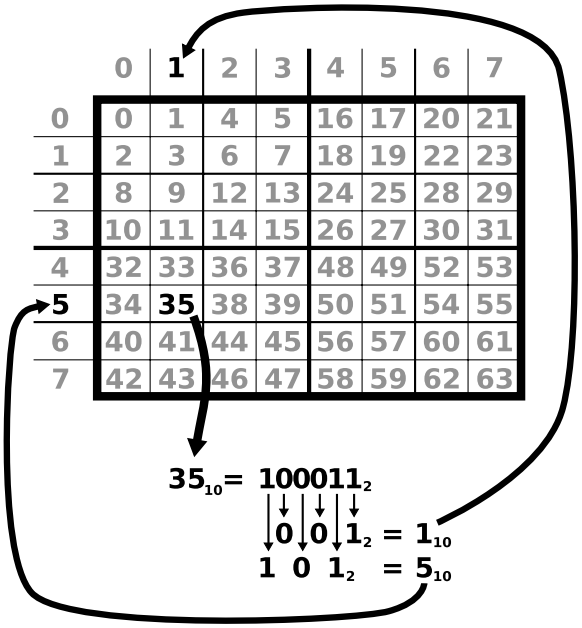



Figure 38: Quadtree coordinate calculation. Given the binary representation of a node's index, the row and column numbers are derived by combining the odd-numbered and even-numbered bits, respectively.

reference numbers and translate the slice-local numbers after each slice. A positive reference number $i$ then denotes the $i$'th frame counting from 1 in the short-term buffer, a negative number $-i$ denotes the $i$'th frame counting from 1 in the long-term buffer. The value 0 is therefore unmapped and can be used otherwise.

**Recursion.** The recursion, which is done to subdivide the quadtree, is straightforward to implement. But I want to mention the calculation of the index numbers for the subnodes. Clearly, every node with the current depth results in four subnodes in the next depth. So, when the algorithm subdivides a node with index $i$, the subnodes get $4i$, $4i+1$, $4i+2$, and $4i+3$ assigned as their indices.

**CPU specifics.** The final problem I stumbled across was a peculiarity of the x86 architecture. The H.264 decoder implementation of FFmpeg uses optimized assembler code with MMX instructions to speed up the decoding. My own code is called from within this decoder to build the motion vector quadtree at the position in decoder control flow determined previously. The problem is that my code, especially the quality loss evaluation with SSIM, uses floating point computation. But because MMX instructions trash the FPU state, I have to remember issuing the `emms` assembler instruction to recover the FPU before I use it. Fortunately, FFmpeg already provides the C Function `emms_c()`, which encapsulates this in a platform-independent way.

### 4.1.5 Optimization

One optimization I implemented early on is a rectangle of interest for the SSIM calculation and the frame replacement during the quadtree creation. This ensures that, working with one quadtree node, all computation takes place only within the area of that node, because other areas are known not to change. As discussed in 3.2.6, I also use a version of SSIM that is a bit less precise, but much faster.

Another important optimization concerns the implementation of the SSIM algorithm. As presented in Section 2.4, SSIM uses a sliding window in which it calculates the mean values, variances and covariance for the sample vectors of both images. Using the usual formulae

$$\begin{aligned}\sigma_x^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\mu_x)^2\\ \sigma_y^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(y_i-\mu_y)^2\\ \sigma_{xy} &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\mu_x)(y_i-\mu_y)\end{aligned}$$

for variances and covariance, we can see that we have to know the mean values $\mu_x$ and $\mu_y$ before calculation of the variances can start. This results in the need to iterate over each sliding window twice. Because memory accesses are generally expensive, it is beneficial to calculate all required values with one iteration only. This optimization is possible by reorganizing the covariance formula as commonly known:

$$\begin{aligned}\sigma_{xy} &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\mu_x)(y_i-\mu_y)\\ &= \frac{1}{n-1}\sum_{i=1}^{n}(x_iy_i-\mu_yx_i-\mu_xy_i+\mu_x\mu_y)\\ &= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_iy_i-\mu_y\sum_{i=1}^{n}x_i-\right.\\ &\qquad\left.\mu_x\sum_{i=1}^{n}y_i+n\mu_x\mu_y\right)\\ &= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_iy_i-\mu_yn\mu_x-\right.\\ &\qquad\left.\mu_xn\mu_y+n\mu_x\mu_y\right)\\ &= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_iy_i-n\mu_x\mu_y\right)\end{aligned}$$

$$= \frac{1}{n-1}\sum_{i=1}^{n} x_i y_i - \frac{n}{n-1}\mu_x\mu_y$$

This shows that we can move the mean values out of the summation. The variance $\sigma_x^2$ of $x$ is equal to the covariance of $x$ with itself ($\sigma_x^2 = \sigma_{xx}$), so the variance formula can be reorganized similarly:

$$\begin{aligned}\sigma_x^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)^2 \\ &= \frac{1}{n-1}\sum_{i=1}^{n} x_i^2 - \frac{n}{n-1}\mu_x^2\end{aligned}$$

Therefore, I can calculate all five values ($\mu_x$, $\mu_y$, $\sigma_x$, $\sigma_y$, $\sigma_{xy}$) in one iteration, by summing up

$$\sum_{i=1}^{n} x_i,\ \sum_{i=1}^{n} y_i,\ \sum_{i=1}^{n} x_i^2,\ \sum_{i=1}^{n} y_i^2, \text{ and } \sum_{i=1}^{n} x_i y_i$$

and combining these afterward to the final results.

Seeing these cumulative sums, which are later operands in subtractions, I thought about the required precision to avoid truncation errors. The size of a sliding window is eight by eight samples, with each sample lying in the 0 to 255 range. Two such samples are multiplied in the variance and covariance cases, yielding a worst possible sum of:

$$8^2 \cdot 255^2 < \left(2^3\right)^2 \cdot \left(2^8\right)^2 = 2^{(2\cdot 3+2\cdot 8)} = 2^{22}$$

A single precision floating point number with its 23 mantissa bits [47] is therefore sufficient, so I used C's `float` type within the local SSIM calculation. Outside the iteration over a single sliding window, I used double precision floating point variables to not loose significant bits, when the values are aggregated into the final SSIM result.

In addition to the speedup already gained by using the faster `float` instead of `double`, I implemented SSE and AltiVec versions of the summations to leverage the vector units of x86 and PowerPC. These instruction set extensions both use 128 bit registers to store vectors of four single precision floating point values and provide operations that work with such vectors in a SIMD fashion.

### 4.1.6 Error Propagation Factors

The sideband data must contain information that will allow the scheduler to estimate the effect on quality, when a slice is skipped and replaced instead of decoded. For that, the preprocessor needs to quantify error emission, which describes, how much an error in the current slice will be propagated into future slices. To calculate the error emission, the preprocessor must first establish, to what degree errors in reference frames are propagated into the current slice. This error immission data is not stored as sideband data itself, but is needed as an intermediate result.

**Error Immission:** The preprocessor extracts a matrix for every slice, which stores a coefficient for every slice of every available reference frame. This coefficient quantifies, how much an error from the reference slice will be immitted into the current slice. According to Section 3.3.2, this coefficient equals to the fraction of the reference slice that is copied to the current slice. Therefore, the coefficients are straightforward to generate by iterating over all motion vectors of the current slice and calculating, what portion of the motion vector target area falls into which slice of the reference frame. Care must be taken here, because the target area pointed to by the motion vector has a size that depends on the macroblock type due to subblocking and a slice boundary might run through it, so more than one slice has to be accounted. To decide that, the slice structure of the reference frame must be known, so metadata for the reference frame has to be kept by the preprocessor. I elaborate on that later in this section.

**Error Emission:** This error propagation is condensed for the scheduler into one value: the accumulated emission factor. Starting from the current slice, errors along all propagation paths are accumulated by multiplying along a path and summing up different paths as explained in Section 3.3.4. Unfortunately, the calculation of this value for the current slice uses information from future slices of the video. The only exploitable limitation here is the knowledge that error propagation never crosses an IDR frame, therefore the calculation of this value requires all slices up to the next IDR.

**Frame Sideband Data Buffering:** As we have seen, both the error immission and emission require knowledge of other frame's metadata. The immission needs the slice structure of past frames that are used as references, the emission needs future frames' error propagation and reference frame information. In both cases however, the frames will never be used across an IDR boundary: Reference frames get invalidated and error propagation is inhibited by IDR frames. Therefore, the clear way to solve this problem is to keep all frame's sideband data starting with the previous IDR. Always appending the current frame to the end of this list, the sideband data of any reference frame is available for the derivation of the immission factors. To easily find a frame's sideband data structure in memory when accessing one of FFmpeg's `AVFrame` structures, I populate the earlier mentioned `opaque` member of the `AVFrame` structure with a pointer to the sideband data. Once the decoding reaches the next IDR, the created frame list is traversed backward to calculate the accumulated emission factor for all kept frames, for which all necessary information is available at that point. Then the frame list is flushed and the current IDR frame becomes the new first element.

Having explained how all separate elements of sideband data are extracted, I continue by describing how to store them.

## 4.2 Sideband Data Format

Before considering how to store the sideband data, I want to summarize, what information is to be stored:

- the metrics for decoding time prediction for each slice of each frame,
- a serialized version of the motion vector quadtree for each frame,
  - a reference frame identifier for each quadtree node,
  - a motion vector for each quadtree node,
- the direct quality loss induced by replacement for each slice of each frame,
- the accumulated error emission factor for each slice of each frame,
- housekeeping data for each frame,
  - frame size,
  - slice count,
  - index of the first macroblock for each slice.

As I do not support the H.264 features flexible macroblock ordering (FMO), arbitrary slice ordering (ASO), and adaptive frame/field coding (AFF) for reasons stated earlier, the sideband data format contains no provisions to leverage those features. Instead, I tried to keep the data format as simple and straightforward as possible, because this helps in debugging and has no disadvantageous effects in the context of this thesis. Extending the format to support additional features as well as streamlining it for compression is possible, but left open for future work.

### 4.2.1 Storing in Custom NALUs

My initial plan was to store the preprocessor output in a separate file, which would have to be provided to the decoder together with the original video. But it is beneficial for usability, if the sideband data is embedded directly into the video stream. As the data is specific to H.264, it should preferably be stored in the raw H.264 stream, without relying on a specific container format like QuickTime.

Fortunately, H.264 with its built-in network abstraction layer already delivers a packetized stream of individual network abstraction layer units (NALUs). Each NALU in the bitstream is preceded by the three byte start prefix `0x00 0x00 0x01`, followed by a five bit type number in the first byte. Several types are reserved to parameter data NALUs and slice data NALUs, but types 0 and 24 to 31 are still unassigned. Therefore, I decided to store the sideband data in NALUs of a custom type. I chose type 31 (`0x1F`), in the hope that future type reservations will happen in ascending order. The sideband data NALUs are inserted into the bitstream between the existing NALUs. The latter can simply be copied, because there are no syntactical dependencies among NALUs. Without additional manipulation, this will create a valid H.264 bitstream.

As I pack the sideband data for one complete frame into one NALU, I simply insert each sideband data NALU before the first slice NALU of a frame. But the decoder, who will read the sideband data NALUs and derive its scheduling decisions from their content, needs to know the information about a frame in advance, because it may be favorable to skip the decoding of earlier slices to accommodate for a later slice with a higher decoding time. For that reason, the sideband data NALUs are not inserted before the first slice NALU of their corresponding frame, but a fixed amount of frames earlier. This provides the decoder with a lookahead window when linearly reading the stream. I chose a lookahead window of 25 frames, equaling one second of video for PAL content. The implications of this size will be discussed in Section 4.3.

Of course, sideband data for the first frames cannot be inserted earlier, so sideband data NALUs worth the lookahead window size will pile up at the beginning of the video to fill the lookahead pipeline.

Another problem is that accessing an H.264 video stream at arbitrary positions gets more difficult, when sideband lookahead is involved. The H.264 decoder can only cleanly start decoding at IDR frames. Once the decoder has found a frame close to the position requested by the user where decoding can start, the sideband data for this frame has already passed. This can happen with live streams, where users can tune in on an ongoing broadcast, or when the user seeks to a different position in the video. The possible options to remedy this are:

- Decode the first frames with no sideband data available or drop them completely. Start using sideband data once the lookahead window has filled.
- Search backward in the stream or prebuffer the sideband channel until sideband data for the intended first frame is available. When the amount of slices per frame is constant, the length of this preroll phase can be determined easily.

### 4.2.2 Bitstream Syntax

I will describe the sideband data syntax with a tabular format similar to the one used in the H.264 standard itself. The format should be clear to a reader familiar with the syntax of the C programming language. Others are invited to refer to Section 7.1 of [2], where the format is explained in more detail.

The syntax descriptors I use are:

**i(n):** signed integer value of n bits written in big-endian byte order.

**u(n):** unsigned integer value of n bits written in big-endian byte order.

**f(n):** IEEE 754 [47] floating point value of n bits written in big-endian byte order.

**ic(n), uc(n), fc(n):** the same as above, but subject to data compression presented in Section 4.2.4.

The one function I use is:

**next_bits(n):** Returns the next n bits from the bitstream without advancing the current reading position.

### NALU Syntax

| nalu() { | |
|---|---|
| **nal_unit_type** /* equal to 0x1F */ | u(8) |
| **mb_width** | uc(16) |
| **mb_height** | uc(16) |
| **slice_count** | uc(8) |
| decoding_time_metrics(slice_count) | |
| replacement_quadtree() | |
| skipping_partitions(slice_count) | |
| error_propagation_factors(slice_count) | |
| } | |

### Decoding Time Metrics Syntax

| decoding_time_metrics(slice_count) { | |
|---|---|
| for (slice = 0; slice < slice_count; slice++) { | |
| **slice_type**[slice] | uc(8) |
| **byte_count**[slice] | uc(24) |
| **intra_pcm_count**[slice] | uc(24) |
| **intra_4x4_count**[slice] | uc(24) |
| **intra_8x8_count**[slice] | uc(24) |
| **intra_16x16_count**[slice] | uc(24) |
| **inter_4x4_count**[slice] | uc(24) |
| **inter_8x8_count**[slice] | uc(24) |
| **inter_16x16_count**[slice] | uc(24) |
| **idct_4x4_count**[slice] | uc(24) |
| **idct_8x8_count**[slice] | uc(24) |
| **deblock_edges_count**[slice] | uc(24) |
| } | |
| } | |

### Replacement Partitions Syntax

| replacement_quadtree() { | |
|---|---|
| empty_quadtree = true | |
| quadtree_node(0) | |
| } | |

| quadtree_node(depth) { | |
|---|---|
| if (next_bits(8) == depth) { | |
| **node_depth** | uc(8) |
| **node_reference_number** | ic(8) |
| if (node_depth \|\| node_reference_number) { | |
| **motion_vector_x** | ic(16) |
| **motion_vector_y** | ic(16) |
| empty_quadtree = false | |
| } | |
| } else { | |
| quadtree_node(depth+1) | |
| quadtree_node(depth+1) | |
| quadtree_node(depth+1) | |
| quadtree_node(depth+1) | |
| } | |
| } | |

### Skipping Partitions Syntax

| skipping_partitions(slice_count) { | |
|---|---|
| for (slice = 0; slice < slice_count; slice++) { | |
| **slice_start_mb_index**[slice] | uc(16) |
| if (!empty_quadtree) { | |
| **direct_quality_loss**[slice] | fc(32) |
| } | |
| } | |
| } | |

### Error Propagation Syntax

| error_propagation_factors(slice_count) { | |
|---|---|
| for (slice = 0; slice < slice_count; slice++) { | |
| **emission_factor**[slice] | fc(32) |
| } | |
| } | |

The entire bitstream is subject to potential byte stuffing to avoid sequences with a special

meaning to appear inside a NALU. Those sequences are the consecutive bytes 0x00 0x00 0x0n, with n being any 4-bit value less than 4. For any such sequence, an additional byte of 0x03 is inserted between the second and third byte. This extra byte has to be removed again when reading the bitstream, by dropping any 0x03 that follows two consecutive 0x00 bytes. Fortunately, FFmpeg already processes the NALUs in that way.

### 4.2.3 Bitstream Semantics

**NALU Semantics:** One NALU encodes the sideband data for exactly one frame of video. The NALU starts with the chosen type of 0x1F (nal_unit_type), which is followed by the frame's dimensions in macroblocks (mb_width, mb_height) and the number of slices for that frame (slice_count).

**Decoding Time Metrics Semantics:** The decoding time metrics derived in Section 3.1 are stored for each slice. The pixel count metric is not stored, as it can be derived from the macroblock information in the skipping partition data.

**Replacement Partitions Semantics:** The quadtree is linearized by prepending each node's data with the depth of the node (node_depth). This information is sufficient to recover the structure of the tree. The inner nodes are not needed later, so only the leaves are stored. The invariant remains that every node must have either zero of four subnodes, as represented in the else-path of the syntax description. The data stored for each node is the number of the reference frame (node_reference_number) and the motion vector (motion_vector_x, motion_vector_y). For those frames, where replacement is completely impossible, an empty tree is stored, represented with both node_depth and node_reference_number being zero for the first and only node. The variable empty_quadtree, which is not a syntax element by itself, displays this condition.

**Skipping Partitions Semantics:** The skipping partitions are described by the index of the first macroblock of each slice (slice_start_mb_index), which is accompanied by the quality loss directly induced when skipping and replacing this slice (direct_quality_loss). How this value is derived has been discussed in Section 3.2.5.

**Error Propagation Semantics:** The required error propagation information is condensed into one value per slice: The accumulated error emission factor (emission_factor) states, how much an error in the current slice will propagate into the future video stream as a whole. See Section 3.3.4 for a detailed explanation of this value.

### 4.2.4 Bitstream Compression

Looking at the way sideband data is stored, it is clear that directly storing the bytes given in Section 4.2.2 would not be very efficient. Most of the time the values do not remotely fill up the possible range:

- The frame size is stored as a 16 bit value for each coordinate, but the higher byte is usually small.
- The slice count per frame usually is considerably less than 256. Especially, the x264 encoder only allows a minimum slice size of one complete macroblock row, so even for a frame 1080 pixels high, just 68 slices are possible.
- The decoding time metrics contain various macroblock type counts. While it is possible that all macroblocks are of the same type and thus one number would become exceptionally high, most of the time, all sorts of macroblocks are found, so each

individual count is small. Especially the higher bytes are often unused.

- The serialized quadtree stores node index and node depth values, especially the latter is small.
- Motion vectors are stored as two 16 bit values, but the average motion vector is short.

I tried to keep the byte count per value as low as possible, but the potential for each number to assume larger values is there and although only rarely occurring, I have to dedicate a sufficient number of bits. I also wanted to keep things simple, so I always used an integer byte count and offloaded the handling of non-integer byte counts entirely on the compression algorithm to be used.

The compression is supposed to be lightweight and easy to implement. The compressed output cannot be written to disk directly, because it is subject to potential byte-stuffing to prevent forbidden byte sequences like the start code to appear inside a NALU. I also wanted to retain control over the implementation to be able to adapt it more easily. Therefore I decided against using a standard compression library and instead chose an algorithm to implement myself.

Because of the observation that the stored values are typically a lot smaller than the possible maximum, the algorithm should provide an elegant way to assign fewer bits to small values and more bits to larger values to achieve the desired compression effect. To simplify the code, I also decided to separate the sideband bitstream syntax and the compression completely: The compression algorithm just receives the stream of bytes marked with the syntax descriptors ic(n), uc(n), or fc(n) and compresses each byte individually, without knowing what syntax element it belongs to. Therefore, the compression only has to handle values within a fixed range.

Although it could be optimized using knowledge about the specific properties of the bitstream, designing a compression algorithm from scratch is outside the scope of this thesis. Therefore I browsed through existing coding methods and found Fibonacci Coding [48] to be quite fit for my needs: It codes small values with fewer bits, down to just two bits for the value 1 and it is elegant and lightweight.

Encoding a positive nonzero integer value into Fibonacci Coding is done by first deriving the Fibonacci decomposition of the given number: Search for the largest Fibonacci number smaller than or equal to the given number and subtract it. Do the same with the remainder. This yields a representation of the value to encode as a sum of Fibonacci numbers, for example $40 = 34 + 5 + 1$. Now, starting with 1, all Fibonacci numbers up to the highest contribution in the sum are checked in ascending order. For every Fibonacci number that is part of the sum, a 1-bit is written, for every Fibonacci number not part of the sum, a 0-bit is written. The example leads to the bitstream 10010001, written from left to right. This bitstream obviously always ends with a 1-bit, so an additional 1-bit is written to mark the end of the coded representation. This works, because two adjacent 1-bits otherwise never appear in the bitstream due to the structure of the Fibonacci numbers. The final coding for the example is therefore 100100011.

As this coding can only encode positive nonzero integers, I converted each incoming byte $b$, treated as a signed value in the range -128 to 127, to the compression input $n$ using this bijection:

$$n = \begin{cases} -2b + 1 & \text{if} \quad b < 0 \\ 1 & \text{if} \quad b = 0 \\ 2b & \text{if} \quad b > 0 \end{cases}$$

This mapping ensures, that small values $n$ are assigned to both positive and negative bytes $b$ close to zero, thus supporting their compressibility. I also tried treating the inputs $b$ as unsigned values, using them directly as compression input $n$, but this decreased the compression effect.

The resulting compression of this method was roughly by a factor of 2, reducing the sideband data overhead to 6.1 %, so the improvement over the previous 12.3 % (compare to Figure 25) is notable. The overhead introduced by the entire sideband reading process with this compression is acceptable. I measured 1.7 % for BBC video, comparing complete decoding with and without sideband decompression and parsing.

Of course this compression is not a final solution, as a domain-specific algorithm should be even more efficient, but as a proof-of-concept, it shows that the data is compressible. A quick comparison by compressing the sideband channel as a whole with common command line tools shows that gzip and bzip2 yield compression factors of 3 and 4, respectively. Although I expect both algorithms to perform worse if applied to the sideband data of each frame individually, these results indicate that additional compression can be achieved with methods more sophisticated than the easily implemented, yet simplistic Fibonacci Coding.

## 4.3 Scheduling the Slices

With the preprocessor finished, I now have available at decoding time:

- execution time metrics, from which I derive execution times,
- the motion vector quadtree, from which I construct replacements for a slice, and
- a quantification of error severity caused by skipping and replacing a slice.

Due to the sideband lookahead window, this information is even available ahead of time, before the slice in question is about to be decoded. This look into the future of the stream can be exploited to decide for each slice, whether it should be decoded for the sake of visual quality or skipped and replaced, favoring lower execution times. In the following, I discuss what model the scheduling decision is based on and how the scheduling algorithm works.

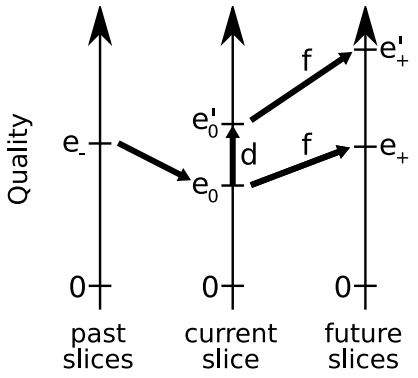



Figure 39: Quality influence of decoding versus replacing a slice. The errors of both past and future slices have each been accumulated into one value.

### 4.3.1 Slice Benefit Model

Compared to a skipped slice, the decoding of a slice has a cost in terms of additional execution time and a visual effect in terms of smaller quality loss. A higher expense in execution time makes a slice more preferable as a skipping candidate. Similarly, a higher quality should discourage the scheduler from skipping this slice. Therefore, I decided to combine both measures in a benefit value for each slice.

Whereas the execution times are conceptually clear as they do not depend on other slices, the quality impact should respect both past slices, from which this slice might inherit errors, and future slice, to which this slice might propagate errors. Figure 39 illustrates the concept for an example slice:

- The previously decoded slices show a quality loss of $e_-$, which is immitted into the current slice (see Section 3.3.2) and causes an error of $e_0$ there.
- If the slice is not skipped but regularly decoded, this error $e_0$ will be emitted into

multiple future slices, causing a total error of $e_+$.

- If the slice is skipped and replaced, the replacement induces an extra error $d$ directly into the current slice, increasing the error to $e'_0$.
- This increased error $e'_0$ causes an emitted error of $e'_+$ in future slices.

The direct error $d$ is known from measurements as explained in Section 3.2.5 and can simply be added to $e_0$ according to Section 3.3.3:

$$e'_0 = e_0 + d$$

Also known is the estimated error emission by the error emission factor $f$ received from sideband data. As I included the contribution of the current slice into $f$, I can formulate:

$$\begin{aligned} e_0 + e_+ &= f \cdot e_0 \\ e'_0 + e'_+ &= f \cdot e'_0 \end{aligned}$$

The emission factor $f$ stays the same in both cases, since the area fractions of future slices, which are responsible for this error propagation, do not depend on the absolute value of the error.

The value relevant for scheduling is the quality difference $\Delta q$ between the skipped and the fully decoded situation:

$$\Delta q = \left(e'_0 + e'_+\right) - \left(e_0 + e_+\right)$$

Using the preceding equations, this can be reformulated equivalently:

$$\begin{aligned} \Delta q &= \left(e'_0 + e'_+\right) - \left(e_0 + e_+\right) \\ &= f \cdot e'_0 - f \cdot e_0 \\ &= f \cdot d \end{aligned}$$

This allows to calculate the quality influence without knowing the error of the reference slices, which is quite beneficial, because tracking these errors would require more knowledge on error propagation to be available to the scheduler and it would make the scheduling itself more expensive.

Using the difference between the predicted execution time for decoding $t_d$ and the predicted execution time for replacement $t_r$, I model the benefit $b$ for each slice as a price-performance-ratio:

$$b = \begin{cases} \frac{\Delta q}{t_d - t_r} = \frac{f \cdot d}{t_d - t_r} & \text{if} \quad t_d > t_r \\ \infty & \text{if} \quad t_d \leq t_r \end{cases}$$

A higher benefit implies that the slice should better be decoded, whereas a lower benefit makes it a candidate for skipping. The rare cases of decoding time being smaller than replacement time are accounted with an infinite benefit, because such slices should never be skipped and replaced. Slices of fully intracoded frames where no replacement is possible are also assigned an infinite benefit.

Apart from the predicted execution times being machine-dependent, these benefit values do not use runtime information, so they allow for two optimizations:

- The values have to be calculated only once for each slice and can be cached for all subsequent scheduling decisions.
- The size of sideband data can be further reduced by storing information only for slices with a low benefit, as slices with a benefit above a certain threshold should not be skipped.
  I did not pursue this optimization, because I wanted to keep sideband data for all slices to ease testing how far the scheduler will scale.

With the benefit for a slice's decoding at hand, I can describe the actual scheduler.

### 4.3.2 Scheduler Design

Video playback relies on the decoder being able to deliver decoded frames at a constant rate.

The objective of the slice scheduler is therefore to maintain the natural deadlines of the frames while keeping the perceived visual quality as high as possible. Thanks to a lookahead window, the scheduler knows the sideband data of upcoming slices ahead of time, so it can look into the future and accumulate predicted execution times to check, if a deadline is expected to be missed. If all deadlines inside the lookahead window appear to be met, no action is required and slices are to be fully decoded. However, if a deadline will be missed, the scheduler needs to select a slice for skipping to help meeting the deadline by reducing the execution time. Because the actual execution times can still differ from the predicted ones, this deadline checking should be repeated after every slice to compensate for unexpected overtime during processing of an earlier slice.

Given a deadline expected to be missed and the execution times for decoding and replacement of the individual slices, the core problem is to select a set of slices for skipping and replacing such that the deadline is met and the loss in visual quality minimized. Considering the complementary problem of selecting a set of slices for decoding while maximizing achieved visual quality, we can easily see that this problem is equivalent to the binary Knapsack Optimization Problem [49]:

- The slices from the current one to the last one before the missed deadline are the items to be included in the knapsack.
- Each item has a weight, given by the extra time decoding requires over replacement: $t_d - t_r$.
- Each item has a value, given by the quality improvement that results from the decoding of the slice: $\Delta q = f \cdot d$
- The knapsack can hold a maximum weight given by the remaining wallclock time $t_w$ until the deadline, reduced by the sum of the replacement times of the slices chosen from: $t_w - \sum t_r$
  This ensures that for a slice left out of the knapsack and therefore not decoded, but replaced, the replacement time is still accounted against the deadline. For a slice that is included in the knapsack, an extra $t_d - t_r$ is accounted in addition to $t_r$, summing up to a total of $t_d - t_r + t_r = t_d$.
- The optimization problem is to fill the knapsack with items of maximal value, but not exceeding the maximum weight. The problem is binary, because an item can either be included once or left out of the result set. A solution for this problem therefore maximizes the quality, while the weight constraint enforces the deadline to be met.

Because the binary Knapsack Optimization Problem is known [51] to be NP-hard [50], so is our problem[2]. But scheduling is supposed to be fast and I do not require the optimal solution, just one which is good enough but fast to compute. Therefore I am going to solve the problem using a greedy algorithm similar to the Decreasing Density Greedy (DDG) algorithm discussed in [52], with the density being the benefit value $b$ explained in the previous section. However, I am not going to start with all slices skipped and then iteratively including slices, but rather the other way around: Starting with an infeasible knapsack solution that violates the deadline by decoding all slices, I will iteratively exclude slices with increasing density until the deadline is met. As I expect in average more slices to be decoded than replaced, this dual algorithm should be faster than DDG. In [53] Bank et al. have shown this dual greedy algorithm to perform sufficiently close to the optimum.

Although the decoder processes frames in decoding order, an actual player would output

[2] The correct way to show this would be a polynomial reduction of the Knapsack Problem to this problem. The required mapping would be reverse to the more intuitive one presented here and can be constructed easily.

**Algorithm 3** Scheduling decision

```
function skipCurrentSlice(lookaheadWindow) {
  // Step 1
  foreach (slice in lookaheadWindow)
    slice.skip = false;

  do {
    deadlineMissed = false;

    // Step 2
    budget = currentFrameDeadline – time();
    // Step 3
    leastBeneficialSlice = null;

    // Step 4
    foreach (slice in lookaheadWindow) {
      if (slice.skip) {
        // Step 4 a
        budget –= slice.replacementTime;
      } else {
        // Step 4 b
        budget –= slice.decodingTime;
        if (slice.benefit < leastBeneficialSlice.benefit)
          leastBeneficialSlice = slice
      }
      // Step 4 c
      if (slice.lastSliceOfFrame) {
        // Step 4 c i
        if (budget < 0) {
          deadlineMissed = true;
          leastBeneficialSlice.skip = true;
          break;
        }
        // Step 4 c ii
        budget += frameDuration;
      }
    }

  } until (!deadlineMissed ||
   lookaheadWindow.slice[0].skip);  // Step 5

  return lookaheadWindow.slice[0].skip;
}
```

frames in display order, which differs from decoding order due to frame reordering. However, my scheduler deals entirely with decoding order and therefore schedules the slices to be decoded at a given rate. Because the maximum display delay is bounded by the H.264 levels, a decoding-order deadline can be transformed to a display-order deadline with an output frame queue of fixed size.

The complete algorithm, which decides for every slice whether to decode or to skip it is given now. For a pseudo-code notation, see Algorithm 3.

1. Initialize the boolean skipping variable with false for all slices in the sideband lookahead window, meaning that all slices will be decoded.

2. Calculate the current execution time budget as the difference between the current frame's deadline and the current wallclock time. This time is available for decoding the remaining slices of this frame.

3. To exclude slices in the order of increasing benefit, the slice with the least benefit has to be remembered. The variable storing the least beneficial slice found is invalidated here, meaning that no slice has been examined yet.

4. Iterate over all slices from the current one up to the last one in the lookahead window.

   a) If the skipping variable for this slice is true, deplete the execution time budget by this slice's estimated replacement time.

   b) If the skipping variable for this slice is false, deplete the execution time budget by this slice's estimated decoding time. If the benefit of this slice is below the one of the currently remembered least beneficial slice, store this slice as the new least beneficial one.

   c) If the current slice is the last one of a frame, then:

      i. Check if the execution time budget dropped below zero, meaning we exceeded the deadline for this frame. If so, the remembered least beneficial slice's skipping variable is set to true and the iteration bails out to Step 5.

ii. If the deadline has been met, replenish the execution time budget with the display duration of one frame, because the deadline of the next frame will be later by this amount of time. Continue the iteration.

5. If no deadlines have been missed or the skipping variable of the current slice is set, the algorithm terminates. Otherwise the algorithm restarts at Step 2.

Once the algorithm terminates, the skipping variable of the current slice determines, whether this slice is skipped or decoded. As the actual execution time might differ from the predicted one, the algorithm is reran from Step 1 for the next slice.

For the execution times accounted against the budget, underestimating the actual time is far more problematic than overestimation, as the former leads to unexpected deadline misses in the future that could have been remedied by skipping more slices earlier. For that reason, I decided to add a safety margin to both the estimated decoding time and the estimated replacement time. To reduce underestimation to less than 10 % of all predictions, the decoding times need to be corrected by a factor of 1.1, the replacement times by 1.2.

The worst case runtime complexity of this algorithm is $O(n^2)$ for a lookahead window of size $n$, but as the iteration bails out once no deadlines have been missed or once it has been determined that the current slice is to be skipped, the average runtime is lower. I measured an average per-slice scheduling overhead of a mere 28 microseconds for BBC video. Given that decoding times are in the magnitude of milliseconds, this overhead is quite acceptable.

Besides the execution time, the lookahead window size also influences the achieved quality, because with a larger window, a potential future deadline miss can be detected earlier and reacted upon better, as more slices are available

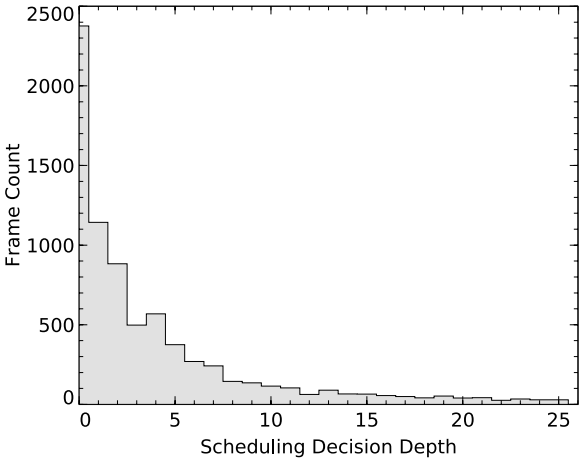



Figure 40: Lookahead depth leading to a skip decision for the current slice. Counting from the current frame, an expected deadline miss that many frames in the future will drive the scheduler to skip the current slice.

to choose a skipping candidate from. I used a lookahead window of 25 frames for all experiments. Figure 40 shows this window to provide enough room to detect and compensate deadline misses.

### 4.3.3 Scheduler Implementation

The implementation of the actual scheduler was straightforward, as all information is readily available from sideband data. The execution time predictions and the benefit value for each slice are calculated once and then stored for the scheduler, which uses the algorithm discussed previously to decide for every slice separately, whether it is to be skipped. However, the following practical problems had to be considered:

**Synthesizing Frame Metadata:** A slice that is skipped is filled with replacement image content as laid out earlier in Section 3.2. This replacement is done at the time where the slice would be decoded, so that later slices, which might use pixels of the skipped slice for spatial or temporal prediction, see image data at

least roughly similar to what they expect. However, this surrogate content is not sufficient to minimize errors in the future decoding process: With H.264, the next frame in decoding order can reuse macroblock type information, reference numbers, and motion vectors of the current frame. To provide the next frame with useful values here, this frame metadata has to be synthesized in addition to the image data. Fortunately, this synthesis is not difficult to accomplish, as motion vectors and reference numbers approximating the results of a completely decoded slice are available in the motion vector quadtree. These values are merely copied into the frame's metadata. The macroblock type is synthesized with a constant single-reference intercoded macroblock type without any subblocking. Visual examination proved the next frame to be closer to its original with this metadata synthesis.

The one difficulty with this synthesis step is that reference numbers from the motion vector quadtree are frame-global numbers, but FFmpeg expects metadata to contain slice-local reference numbers. These differ, because the per-slice reference buffer is reordered for every slice to increase compressibility. This problem has already been dealt with in the reverse direction in Section 4.1.4. Now I translate back to slice-local numbers in a similar way.

The entire metadata synthesis introduces a slowdown on the replacement, but the execution time discussion in Section 3.2.5 already included these extra calculations.

**Unit Stride Copying:** The replacement of a slice is created by iterating over all macroblocks of that slice, looking up the motion vector and reference number in the motion vector quadtree and then copying the macroblock from the reference image into the current one. But if done strictly that way, copying would always take place in blocks of 16×16 pixels, meaning that only 16 consecutive bytes are copied with unit stride access. Unit stride memory accesses are the fastest because DRAM can use burst transfers to serve them. Therefore I combine macroblocks, if they fall into the same quadtree node and copy from adjacent areas of the reference image. This results in longer unit stride transfers, which speeds up the replacement by about 10 %.

**Output Queue Simulation:** When running the scheduler outside an actual video player to simulate results, I have to ensure that the execution time budget does not accumulate unboundedly. If the decoder gets too far ahead of time, the output frame queue of an actual player would fill up, which would block the decoder. I simulate this by limiting the execution time budget to the time equivalent to 10 frames.

## 4.4 Integration into Verner

Verner [54] is a real-time-capable video player built on top of the DROPS [55] operating system. To integrate my scheduler into Verner, I reused work done for [20], namely the handling of the coefficients file for the execution time predictor. To get H.264 support in Verner, I had to update the included copy of FFmpeg's libavcodec to the latest version. The integration of my own scheduler code was then rather straightforward.

In the learning phase, Verner measures the actual execution times for decoding and replacement for every slice using the `get_thread_time_microsec()` function. These times are fed into the LLSP solver, which derives the prediction coefficients from them and stores them in a file. These coefficients are then used by the actual scheduler to predict the execution times. From those times and the benefit values derived from sideband data, the scheduler decides for each slice, whether it should be skipped to ensure that deadlines are met.

# 5 Evaluation and Conclusion

I now present the results of my thesis. Starting with preliminary evaluation in Section 5.1, I finally compare my method to other slice classification and error concealment approaches in Section 5.2. Remarks on the achieved flexibility and an outlook into future work conclude my work.

All results have been obtained under Linux on an AMD Sempron 2200+ (1.5 GHz) machine.

## 5.1 Preliminary Results

The slice scheduler builds primarily upon three accomplishments: decoding time prediction, slice replacement and error propagation estimation. I will briefly present these results in the following sections. The time required to preprocess the videos to extract this data as well as the resulting bitstream size overhead can be seen in Table 2.

### 5.1.1 Decoding Time Prediction

I used the videos Freeway, Golf, Shore, and BBC as the training set to calculate the prediction coefficients. The prediction results can be found in Table 3. Especially the results of the Lady1 video, which was not part of the training set are remarkable. Figure 41 illustrates the relative error of that prediction. A detail plot in Figure 42 shows, that the prediction follows the decoding time variations of different slices of the same frame as well as the same slice of different frames.

| Video | Runtime (h:mm:ss) | Size Overhead |
|---|---|---|
| Freeway | 0:02:46 | 4.4 % |
| Golf | 0:00:10 | 26.4 % |
| Shore | 0:01:28 | 17.7 % |
| BBC | 0:54:47 | 6.1 % |
| Lady1 | 3:57:04 | 3.0 % |
| Lady2 | 3:49:00 | 1.5 % |

Table 2: Preprocessor runtime and bitstream size overhead caused by sideband data

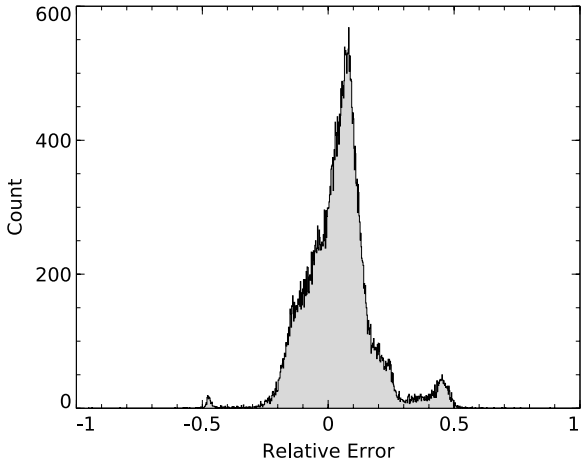



Figure 41: Relative error histogram of decoding time prediction for Lady1 video

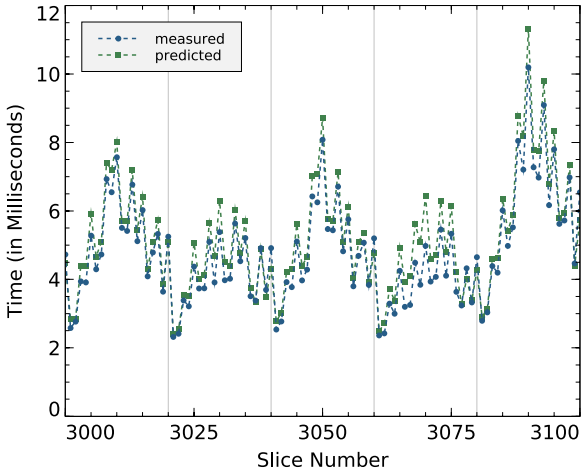



Figure 42: Prediction detail of Lady1 video. Vertical lines show frame boundaries.

### 5.1.2 Slice Replacement

The slice replacement is best evaluated by visual inspection. Figure 43 tries to illustrate, that the quadtree subdivision follows the areas of motion in the frame. The replacements are generally of high quality.

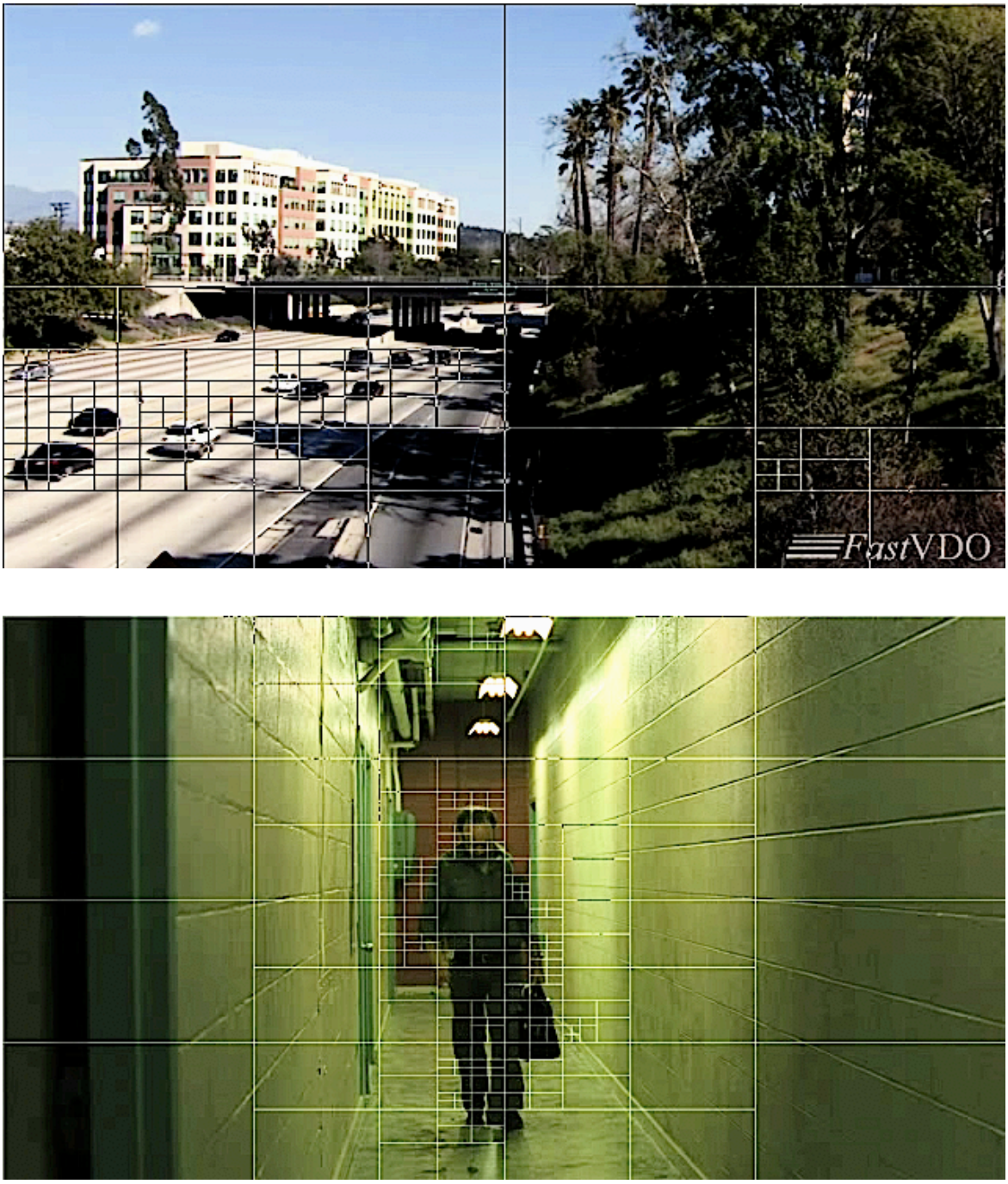



Figure 43: Quadtree subdivisions overlaid on their corresponding replacement frames from Freeway (top) and Lady1 (bottom) videos. Note that the subdivision appears finer in the Lady1 video due to its higher resolution.

| Video | Avg. Relative Error (Std. Deviation) | Avg. Absolute Error (Std. Deviation) | Values within ±0.2 Rel. Error | Values within ±1 ms Abs. Error | 99 % Quantile* |
|---|---|---|---|---|---|
| Freeway | 0.128 (0.016) | 1.67 ms (0.60 ms) | 99.6 % | 0.0 % | -1.327 ms |
| Golf | 0.222 (0.052) | 0.09 ms (0.03 ms) | 34.8 % | 100.0 % | -0.049 ms |
| Shore | 0.158 (0.045) | 0.47 ms (0.22 ms) | 89.4 % | 99.7 % | -0.184 ms |
| BBC | 0.107 (0.093) | 0.84 ms (0.76 ms) | 89.0 % | 58.1 % | 0.931 ms |
| Lady1 | 0.044 (0.130) | 0.12 ms (0.39 ms) | 90.2 % | 97.9 % | 0.766 ms |

* Increasing the predictions by this value results in 99 % overestimation

Table 3: Per-slice decoding time prediction for various videos. The Lady1 video was not part of the training set.

### 5.1.3 Error Propagation Estimation

I estimated the resulting error propagation for the skipping of randomly selected slices at a rate of about one slice out of ten. Comparing this estimate to the real error yields the differences listed in Table 4. The values obtained for the two versions of the Lady video allow the conclusion, that more slices lead to a more accurate error estimation. Although practically irrelevant, each macroblock could in the extreme be a slice of its own. This could move the error estimation closer to the accuracy of the ROPE algorithm [23], which accounts for error propagation on single pixel level. For practical uses, I would expect the potential for better estimates using slices that follow the structure of the frame, like combining areas of coherent motion in one slice. Even checkerboard-like patterns should be beneficial, because such slices would cover a smaller number of different objects in the picture. But this requires encoder support for flexible macroblock ordering (FMO), which is not available yet, so the slices in these videos are merely horizontal stripes.

| Video | Slices/ Frame | Avg. Difference (Std. Deviation) | Max. Error Value |
|---|---|---|---|
| Freeway | 1 | 0.0645 (0.0514) | 0.128 |
| Golf | 1 | 0.0097 (0.0113) | 0.048 |
| Shore | 1 | 0.0662 (0.0830) | 0.106 |
| BBC | 5 | -0.0020 (0.0042) | 0.067 |
| Lady1 | 20 | -0.0005 (0.0013) | 0.019 |
| Lady2 | 4 | -0.0020 (0.0039) | 0.046 |

Table 4: Error propagation estimation for various videos. For each video, randomly selected slices have been skipped at an average rate of one slice out of ten. The resulting errors were predicted for each slice and compared to the measured error.

## 5.2 Comparison to Other Methods

My slice scheduling method selects slices for skipping based on their benefit value. Slices with the lowest benefit are skipped first. A skipped slice is replaced with content copied from reference frames using the motion vector quadtree. To evaluate this method in its entirety, I am going to compare it to various other slice selection strategies, some of which have already been mentioned in Section 2.3, and to one alternative slice replacement strategy:

**No Skip:** Slices are not skipped at all. Instead, when frames miss their deadline, the previous frame remains visible until playback recovers at an IDR frame. Current video players behave similarly.

**Highest Cost Skip:** The slice with the highest cost, that is: the highest decoding time, is skipped first. The reasoning behind this idea is that skipping the slices of the highest cost, a minimal amount of slices is skipped to meet the deadlines.

**Highest Cost, FFmpeg-Concealed:** Like the preceding strategy, but replacement is done using FFmpeg's built-in error concealment, which is related to the error concealment in the H.264 test model. (see [56] on page 13 f.)

**Least Direct Error Skip:** The slice with the least directly introduced error, disregarding any error propagation, is skipped first. This method is similar to the frame-level distortion calculation suggested in [25]. The assumption behind this method is that minimizing the first order error will also reduce the propagated error.

**Lifetime-Based Skip:** Slices are skipped according to a benefit value, but instead of multiplying the directly induced error with the error emission factor from sideband data, it is multiplied with the frame's reference lifetime. This lifetime is the number of future frames, which can access the current frame in the reference buffer. It is current practice for MPEG-2 to skip B-frames first, because they are never used as references. This scheduling method extends this idea to H.264 as has been suggested in [26].

**Least Benefit Skip:** Finally, this is the method developed in this thesis.

To compare the different approaches, I wanted to simulate playback on a machine incapable of decoding the video completely without missing deadlines. This could have been done by actually using a slower machine and then scheduling the videos to their native framerate of 25 fps. However, the much easier and more flexible way is to use one machine of fixed speed and adjust the target framerate of the scheduler. A higher framerate thus allows to simulate a slower machine and vice versa. For each video, I took the 95 % quantile of all slice's decoding times as the basis for calibration. This means: a machine capable of decoding 95 % of all slices in time is considered just fast enough. A fraction of that quantile was then used to determine the speed of the simulated machine. I used rather low fractions down to 50 %, meaning that a machine with only half the required CPU power is simulated. This was done to make the errors induced by the different schedulers more distinctive. The scheduling with higher fractions will skip fewer slices, so the results will be better. The target framerates I chose can be seen in Table 5. All the scheduling methods had to compete at those same framerates.

| Video | Fraction | Framerate |
|---|---|---|
| Freeway | 80 % | 90 fps |
| Golf | 80 % | 2446 fps |
| Shore | 80 % | 334 fps |
| BBC | 50 % | 35 fps |
| Lady1 | 50 % | 19 fps |

Table 5: Target framerates for the scheduling

I evaluated the different strategies by running each video through each contender algorithm at the selected framerate. The resulting video was compared to the original using SSIM over the entire video. These quality differences can be seen in Figure 44.

Except for the Golf video, were the results are too close together to be conclusive, my method outperforms or is on par with the contenders for all videos. It is also visible that FFmpeg's error concealment always yields the worst or second worst result, proving that my error concealment method has considerable benefit. On the other hand, the FFmpeg concealment and the No Skip strategy require only little to no preprocessing. But as preprocessing needs to be done only once for each video and the size overhead of the sideband data is acceptable, this extra work seems a reasonable price for the achieved quality on resource-limited systems. The other contenders, Highest Cost, Least Direct Error, and Lifetime-Based Skip, all used the motion vector quadtree to replace slices, so they all require the same preprocessing as my Least Benefit Skip scheduler. It is also interesting, that the performance comparison of the Least Direct Error method, which ignores error propagation completely, and the

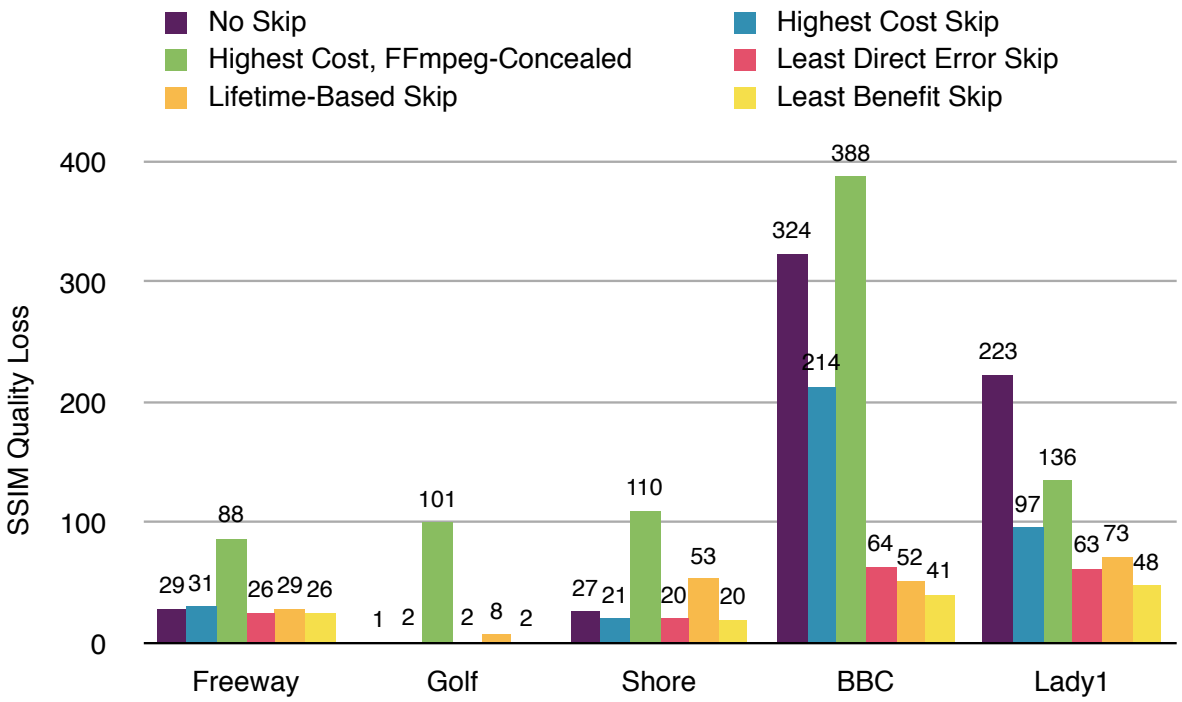



Figure 44: SSIM quality losses introduced by the various scheduler methods. The smaller the value the higher the quality of the video.

Lifetime-Based method, which uses a simplistic error propagation model, seems to depend on the video, but most of the time, the Lifetime-Based method performs worse. This indicates that no notion of error propagation can be better than a simple one. It also shows that my detailed analysis of error propagation can successfully improve visual performance over the propagation model presented in [26].

To confirm the results from the objective quality measurements, I also performed a test screening with six human test viewers. Having them watch the same 13 s clip from the BBC video for all scheduling methods, they were asked to judge the quality of each clip. Of course the test was made blindly. I included the undegraded clip in the test set, which acts as a plausibility check for the results. Because all viewers voted the original best and the FFmpeg-concealed Highest Cost Skip clip worst, I was able to scale the votes of each viewer to a common scale. The aggregated results can be seen in Figure 45 and although six viewers are not representative, they confirm the objective values from SSIM. My method is rated to be of best quality after the original. This test also shows that the video quality is visually acceptable even with a simulated machine of only half the required CPU power (see Table 5).

## 5.3 Flexibility

The slice scheduling method presented is not a monolithic algorithm, but rather a framework that draws on many individual concepts, which can be improved upon separately.

- SSIM can be replaced by a different quality loss metric. Only little, well-documented assumptions were made on the inner workings of this metric. Improving SSIM with foveation [57] or brightness weighing (suggested in [33]) is easily possible.
- The video is partitioned in skipping and replacement partitions. This separation

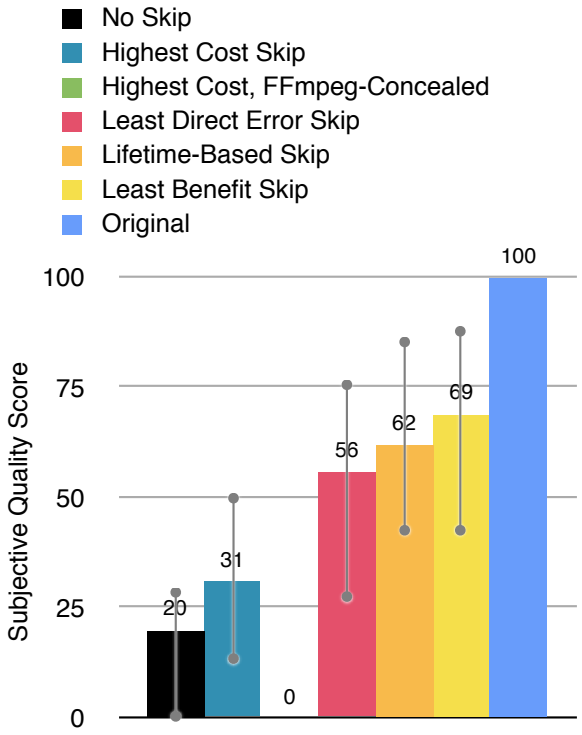



Figure 45: Subjective quality for BBC video clip. The bar shows the average score, the two auxiliary marks give the minimum and maximum score for each scheduler strategy.

was necessary because the available encoders did not support flexible macroblock ordering (FMO) or arbitrary slice ordering (ASO). Once these features are supported, partition shapes can follow structural properties of the frame like areas of interest. It should be easy to adapt my method, because I made no assumptions on the shape of the partitions.

- ASO allows to order slices of one frame in decreasing benefit order, so the most important slices can be decoded first while the deadline is further away. As the deadline draws closer, only slices of lower benefit remain, which can be skipped should time demand it. ASO is easy to exploit as my method imposes no constraints on slice order.

- The partition size is arbitrary. Skipping partitions can be reduced down to individual macroblocks or increased to full frames. Table 4 indicated that smaller partitions can improve the error propagation estimation and the scheduler comparison showed that a good propagation estimate can improve the achieved quality.

- The error concealment using the motion vector quadtree is orthogonal to the actual scheduling using benefit values. It is easy to integrate improved concealment methods. Only the concealment information itself and the measured directly induced error change in sideband data.

- The error propagation estimate can be enhanced separately. This would only change the error emission factor in sideband data. Error concealment and scheduling could be reused.

- Finally, most of the concepts can be applied to coding standards other than H.264.

## 5.4 Future Work

The flexibility listed in the previous section provides an interesting space of options to explore. I consider the improvement of perceptional properties without sacrificing the lightweight approach of scheduling according to a single benefit value to be most interesting. Improving the quality loss metric from the frame-based SSIM toward incorporating temporal structure is a promising idea. This would allow to better quantify errors caused by frame drops and other sudden disruptions to the movement in the video, which SSIM might not penalize adequately. The error propagation estimation would need considerable improvement to predict this type of errors. Such a three-dimensional quality loss metric could even allow to extend my currently strictly frame-based approach beyond frame borders, defining video partitions for skipping or replacement that stretch not only in

space, but also in time. Ultimately, this could lead to the elimination of the assumption that video is discretized into frames along the time axis, which would bring the method even closer to the initial model of a video as a continuous three-dimensional function.

Another area for follow-up work is to make the approach more practical. Despite the low invasiveness already achieved, reducing the required preprocessing would make the method more appealing for incorporation into future decoding standards. One idea to accomplish this is to use a quality loss metric like VQM [36] operating in the compressed domain. Together with encoders using ASO and slice priorities to distinguish slices of interest from less important background areas, this can allow for a method that preprocesses the video much faster, leading to a method that works online. Unfortunately, I expect a trade-off between speed and perceptionality, because the most expensive part of the preprocessing is the creation of the motion vector quadtree with all its SSIM comparisons to achieve a visually similar slice replacement.

If the preprocessing approach is kept, the compression and layout of the sideband data could be enhanced to allow the preprocessor output to be more tightly integrated into the bitstream. This may allow a more fine grained error concealment or a scalable decoding with visual quality tunable on demand. H.264 already provides a way of choosing different quality levels using multiple versions of the same stream. Switching to a different stream during playback is possible using switching slices. It would be interesting and I think not difficult to include this switching decision into my scheduler. The stream of lower quality would be handled as an alternate set of slices with shorter decoding times, but a continuous quality loss to the high quality stream. If the scheduler receives this quality loss from the preprocessor via sideband data, it can decide, whether switching to a lower quality stream actually leads to a higher quality, because slice skipping is prevented. However, switching slices are currently targeted for streaming video, because the increased storage size makes this impractical for prerecorded media.

Lastly, it should be possible to apply the scheduling according to visual benefit not only to the CPU resource, but to network bandwidth as well. What would change in the determination of the benefit value is the cost component, which would then be measured in network bandwidth instead of processing time. But as the replacement information for each slice is much smaller than the slice data itself, basically the same scheduling ideas can be used: Slices are sent in decreasing benefit order. If the network saturates, slices of low benefit are not transmitted, but their replacement information is sent instead.

## 5.5 Summary

In this thesis, I developed a lightweight metric quantifying the perceptional importance of H.264 slices. This metric is based on a slice replacement scheme that provides a visually similar replacement for each slice, using lower computation time and bitstream size than the original slice. I measured the error induced by this replacement with an algorithm modeled after the human vision system, not with the otherwise often employed but unsuitable mean squared error. I analyzed error propagation and integrated it into the metric as an error emission factor, resulting in a single importance value for each slice. This value is complemented by an estimate of the computational cost, derived using methods I developed earlier. I combined cost and perceptional importance to formulate a scheduling algorithm, which uses preprocessor-provided sideband data of acceptable size to admit the individual slices for decoding or to select them for skipping. This allows video playback of acceptable quality on machines otherwise severely underpowered for the task. Both subjective and objective evaluation have shown my method to outperform all competing approaches. These results can be used to improve the behavior of current video playback systems,

which currently present the user with stuttering video on CPU shortage. My method allows for a much more graceful degradation of playback quality.

## References


[1] Apple HD Gallery – System Recommendations http://www.apple.com/quicktime/guide/hd/recommendations.html

[2] ISO/IEC 14496-10: Coding of audio-visual objects – Part 10: Advanced Video Coding

[3] Microsoft VC-1 http://www.microsoft.com/windows/windowsmedia/forpros/events/NAB2005/VC-1.aspx

[4] On2 Technologies Truemotion VP7 http://www.on2.com/technology/vp7/

[5] Blu-ray Disc Association http://www.blu-raydiscassociation.com/

[6] HD DVD Promotion Group http://www.hddvdprg.com/

[7] Apple iPod Technical Specifications http://www.apple.com/ipod/specs.html

[8] Apple QuickTime H.264 Technology http://www.apple.com/quicktime/technologies/h264/

[9] Specification for the use of Video and Audio Coding in DVB services delivered directly over IP Protocols http://www.dvb-h.org/PDF/a084r1.tm2821r9.dTs102005.V1.2.1.pdf

[10] iChat AV uses H.264 http://www.apple.com/macosx/features/ichat/

[11] PREMIERE HD http://info.premiere.de/inhalt/eng/medienzentrum_news_uk_05122005.jsp

[12] Tobias Oelbaum, Vittorio Baroncini, Thiow Keng Tan, Charles Fenimore: Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard, *International Broadcasting Conference (IBC)*, September 2004, http://www.itl.nist.gov/div895/papers/IBC-Paper-AVC%20VerifTestResults.pdf

[13] Gary J. Sullivan, Pankaj Topiwala, Ajay Luthra: The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions, *SPIE Conference on Applications of Digital Image Processing XXVII*, August 2004, http://www.cdt.luth.se/~peppar/kurs/smd151/spie04-h264OverviewPaper.pdf

[14] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, Ajay Luthra: Overview of the H.264/AVC Video Coding Standard, *IEEE Transactions on Circuits and Systems for Video Technology*, July 2003, http://iphome.hhi.de/wiegand/assets/pdfs/csvt_overview_0305.pdf

[15] MPEG Overview http://www.fh-friedberg.de/fachbereiche/e2/telekom-labor/zinke/mk/mpeg2beg/beginnzi.htm

[16] Peter List, Anthony Joch, Jani Lainema, Gisle Bjøntegaard, Marta Karczewicz: Adaptive Deblocking Filter, *IEEE Transactions on Circuits and Systems for Video Technology*, July 2003, http://vc.cs.nthu.edu.tw/home/paper/codfiles/shihyu/200408061426/Adaptive_Deblocking_Filter.pdf

[17] Y. Zhong, I. Richardson, A. Miller, Y. Zhao: Perceptual Quality of H.264/AVC Deblocking Filter, http://www.rgu.ac.uk/files/Perceptual%20quality%20of%20H264AVC%20deblocking%20filter_final.pdf

[18] Detlev Marpe, Heiko Schwarz, Thomas Wiegand: Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard, *IEEE Transactions on Circuits and Systems for Video Technology*, 2003, http://ip.hhi.de/imagecom_G1/assets/pdfs/csvt_cabac_0305.pdf

[19] ISO/IEC 14496-2: Coding of audio-visual objects – Part 2: Visual

[20] Michael Roitzsch: Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video, undergraduate thesis, 2005, http://os.inf.tu-dresden.de/papers_ps/roitzsch-beleg.pdf

[21] Peter Altenbernd, Lars-Olof Burchard, Friedhelm Stappert: Worst-Case Execution Times Analysis of MPEG-2 Decoding, *Proceedings of the 12th Euromicro Conference on Real Time Systems (ECRTS)*

[22] Damir Isović, Gerhard Fohler: Quality aware MPEG-2 Stream Adaptation in Resource Constrained Systems, *Proceedings of the 16th Euromicro Conference on Real Time Systems (ECRTS)*, 2004, http://www.mrtc.mdh.se/publications/0679.pdf

[23] Rui Zhang, Shankar L. Regunathan, Kenneth Rose: Video Coding with Optimal Inter/Intra-Mode Switching for Packet Loss Resilience, *IEEE Journal on Selected Areas in Communication*, June 2000, http://scl.ece.ucsb.edu/pubs/pubs_A/a00_5.pdf

[24] Enrico Masala, Davide Quaglia, Juan Carlos De Martin: Adaptive Picture Slicing for Distortion-based Classification of Video Packets, *IEEE Fourth Workshop on Multimedia Signal Processing*, 2001, http://www.cercom.polito.it/Publication/Pdf/92.pdf

[25] Fabio De Vito, Laura Farinetti, Juan Carlos De Martin: Perceptual Classification of MPEG Video for Differentiated-Services Communications, *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo*, 2002, http://demartin.polito.it/papers/DeVito_ICME2002.pdf

[26] Paolo Bucciol, Enrico Masala, Juan Carlos De Martin: Perceptual ARQ for H.264 Video Streaming over 3G Wireless Networks, *IEEE International Conference on Communications*, 2004, http://media.polito.it/papers/masala_icc2004.pdf

[27] Fabio De Vito, Davide Quaglia, Juan Carlos De Martin: Model-based Distortion Estimation for Perceptual Classification of Video Packets, *Proceedings of the IEEE International Workshop on Multimedia Signal Processing (MMSP)*, 2004, http://media.polito.it/papers/devito_mmsp2004.pdf

[28] Bernd Girod: What's wrong with mean-squared error?, *Digital Images and Human Vision*, MIT Press, 1993, pp. 207-220

[29] Dmitriy Vatolin, Alexander Parshin, Oleg Petrov, Artem Titarenko: Subjective Comparison of Modern Video Codecs, CS MSU Graphics & Media Lab Video Group, January 2006, http://www.compression.ru/video/codec_comparison/pdf/msu_subjective_codecs_comparison_en.pdf

[30] Peak Signal to Noise Ratio http://en.wikipedia.org/w/index.php?title=Peak_signal-to-noise_ratio&oldid=54382448

[31] Michael P. Eckert, Andrew P. Bradley: Perceptual quality metrics applied to still image compression, *Signal Processing 70*, 1998, pp. 177-200, http://www.itee.uq.edu.au/~bradley/Papers/APB%20IQ79.pdf

[32] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, Eero P. Simoncelli: Image Quality Assessment: From Error Visibility to Structural Similarity, *IEEE Transactions on Image Processing*, April 2004, http://www.cns.nyu.edu/~zwang/files/papers/ssim.pdf

[33] Zhou Wang, Ligang Lu, Alan C. Bovik: Video Quality Assessment Based on Structural Distortion Measurement, *Signal Processing: Image Communication*, February

2004, pp. 121-132, http://www.cns.nyu.edu/~zwang/files/papers/vssim.pdf

[34] Charles Poynton: Frequently Asked Questions about Color http://www.poynton.com/PDFs/ColorFAQ.pdf

[35] Final Report from the Video Quality Experts Group on the Validation of Objective Models of Video Quality Assessment, June 2000 ftp://ftp.its.bldrdoc.gov/dist/ituvidq/phase1_final_report/COM-80E.pdf

[36] Feng Xiao: DCT-based Video Quality Evaluation, Winter 2000, http://www-ise.stanford.edu/class/ee392j/projects/projects/xiao_report.pdf

[37] FastVDO Test Videos http://www.fastvdo.com/H.264.html

[38] BBC Test Video http://www.apple.com/quicktime/guide/hd/bbcmotiongalleryreel.html

[39] "Lady In The Water" Test Video http://www.apple.com/trailers/wb/ladyinthewater/hd/

[40] x264 http://developers.videolan.org/x264.html

[41] FFmpeg http://www.ffmpeg.org/

[42] Marcelo Bertalmío, Guillermo Sapiro, Vicent Caselles, Coloma Ballester: Image Inpainting, *Proceedings of SIGGRAPH 2000*, July 2000, http://www.iua.upf.es/~mbertalmio/bertalmi.pdf

[43] Raphael A. Finkel, J. L. Bentley: Quad Trees: A Data Structure for Retrieval on Composite Keys, *Acta Informatica 4*, 1974, pp. 1-9

[44] M. Dowell, P. Jarrat: The "Pegasus" method for computing the root of an equation, *BIT Numerical Mathematics*, Springer Netherlands, December 1972

[45] Matlab implementation of SSIM http://www.cns.nyu.edu/~lcv/ssim/

[46] GNU General Public License http://www.gnu.org/copyleft/gpl.html

[47] IEEE Standard for Binary Floating-Point Arithmetic, 1985 http://754r.ucbtest.org/standards/754.pdf

[48] Aviezri S. Fraenkel, Shmuel T. Klein: Robust Universal Complete Codes for Transmission and Compression, Discrete Applied Mathematics, January 1996, pp. 31-55, http://www.cs.biu.ac.il/~tomi/Postscripts/robust.ps

[49] Michael R. Garey, David S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co. New York, 1979, p. 247

[50] Michael Sipser: Introduction to the Theory of Computation, PWS Publishing, 1997, Section 7.4

[51] Richard Manning Karp: Reducibility among combinatorial problems, *Complexity of computer computations*, 1972, pp. 85-103, Slides: http://dclab.cs.nthu.edu.tw/~kwc/20030808/karp.pdf

[52] James M. Calvin, Joseph Y-T. Leung: Average-case analysis of a greedy algorithm for the 0/1 Knapsack Problem, February 2003, http://www.cis.njit.edu/~calvin/knapsack.pdf

[53] B. Bank, G. Diubin, A. Korbut, I. Sigal: The average behaviour of greedy algorithms for the knapsack problem: Computational experiments, 2004, http://edoc.hu-berlin.de/series/mathematik-preprints/2004-6/PDF/6.pdf

[54] Carsten Rietzschel: VERNER – ein Video EnkodeR uNd playER für DROPS, Master's Thesis, http://os.inf.tu-dresden.de/papers_ps/rietzschel-diplom.pdf

[55] The Dresden Real-Time Operating Systems Project http://os.inf.tu-dresden.de/drops/overview.html

[56] Thomas Stockhammer, Miska M. Hannuksela, Thomas Wiegand: H.264/AVC in Wireless Environments, *IEEE Transactions on Circuits and Systems for Video Technology*, July 2003, http://ip.hhi.de/imagecom_G1/assets/pdfs/csvt_wireless_0305.pdf

[57] Zhou Wang, Ligang Lu, Alan C. Bovik: Foveation Scalable Video Coding with Automatic Fixation Selection, *IEEE Transactions on Image Processing*, February 2003, http://www.cns.nyu.edu/~zwang/files/papers/fsvc.pdf