Guided Transcoding for Next-Generation Video Coding (HEVC)

Harald Nordgren
ada09hno@student.lth.se

June 12, 2017

Master's thesis work carried out at Ericsson AB

Supervisors:
Kenneth Andersson, kenneth.r.andersson@ericsson.com
Ruoyang Yu, ruoyang.yu@ericsson.com
Michael Doggett, michael.doggett@cs.lth.se

Examiner:
Jörn W. Janneck, jwj@cs.lth.se

Abstract

Video content is the dominant traffic type on mobile networks today and this portion is only expected to increase in the future. In this thesis we investigate ways of reducing bit rates for adaptive streaming applications in the latest video coding standard, H.265 / High Efficiency Video Coding (HEVC).

The current models for offering different-resolution versions of video content in a dynamic way, so called *adaptive streaming*, require either large amounts of storage capacity where full encodings of the material is kept at all times, or extremely high computational power in order to regenerate content on-demand.

Guided transcoding aims at finding a middle-ground were we can store and transmit less data, at full or near-full quality, while still keeping computational complexity low. This is achieved by shifting the computationally heavy operations to a preprocessing step where so called *side-information* is generated. The side-information can then be used to quickly reconstruct sequences on-demand – even when running on generic, non-specialized, hardware.

Two method for generating side-information, pruning and deflation, are compared on a varying set of standardized HEVC test sequences and the respective upsides and downsides of each method are discussed.

Keywords: Video Compression, Adaptive Streaming, Transcoding, H.265, HEVC

Acknowledgements

I want to thank Ruoyang Yu for his continued support with programming and video coding theory and Kenneth Andersson for his help with the report writing. I also want thank Thomas Rusert and Per Hermansson for their assistance during the project.

I want thank Markus Flierl at KTH for his lectures on images and video, and Michael Dogget and Jörn Janneck at LTH for their input on the thesis drafts.

Contents

Acronyms

ABR Adaptive Bit Rate. 10, 26, 39

CABAC Context-Adaptive Binary Arithmetic Coding. 8
CTB Coding Tree Block. 5

DCT Discrete Cosine Transform. 7

fps frames per second. 1–3, 21, 22, 27

GOP Group of Pictures. 9, 22
GT Guided Transcoding. 10, 12, 15, 21, 26, 27, 39

H.264 Video coding standard. 8, 12, 39
H.265 Video coding standard, successor to **H.264**. 12, 39
HD High-Definition. 2
HEVC High Efficiency Video Coding, synonym of **H.265**. 4, 6, 8, 18, 21, 22
HM HEVC Test Model. 22
HQ High Quality. 15–19, 22–24, 26–28

JIT Just-in-Time. 10, 11, 13, 15
JVT Joint Video Team. 21

kbit/s kilobits per second. 24

LQ Low Quality. 15–19, 22–24, 26–28

MSE Mean Squared Error. 5, 26
MV Motion Vector. 5, 6

PSNR Peak Signal-to-Noise Ratio. 21, 23, 24, 26–28

QP Quantization Parameter. 7, 9, 13, 21–23

RDO Rate-Distortion Optimization. 6, 9, 10
RDOQ Rate-Distortion Optimized Quantization. 9, 21, 27, 28

RGB Red, Green and Blue. 2

SBH Sign-Bit Hiding. 10, 13, 21, 27, 28
SHVC Scalable High Efficiency Video Coding. 12
SI Side-Information. 15–17, 19, 21–23, 26
SVC Scalable Video Coding. 10, 12, 13, 15

YCbCr Color format often used instead of **RGB**. 2, 24

Chapter 1
Introduction

Video is the dominant traffic type on mobile networks today and is expected to take up 70% of all traffic in a few years. To effectively store and transmit high-quality video the data needs to be encoded. Modern video coding algorithms can decrease the size of raw video by hundreds of times with little or no discernible loss in video quality. To achieve this, local redundancies within each frame and between neighboring pictures are utilized, and different statistical methods guide the process to achieve the lowest file size at the least distortion of picture quality.

The standard scenario for digital video is for the images first to be captured by a camera, then encoded and stored somewhere, and transmitted over a network. On the receiving end the process is reversed; the sequences is decoded and then displayed. An encoder always goes together with a decoder, together the two are known as a *codec* [1].

1.1 Image and Video Fundamentals

1.1.1 Digitally Representing Images and Video

The simplest form of image, a greyscale one, can be represented as a *light intensity function* [2]. The intensity function exists in continuous space and has infinite precision. For every x- and y-coordinate there is a light component representing the brightness of the image. To get a digital image from the light intensity function we *sample* it at certain intervals. The samples are usually confined to a rectangular area – the borders of the image. Light samples are called pixels or pels, from the word *picture element*. The number of samples in a frame are usually written in the form $N \times M$ giving the number of horizontal versus vertical samples [3], for example 1920×1080. It is also common to refer to the frame size only by its height.

What sets video apart from still images is movement. Several images shown in rapid succession creates the illusion of movement as long as the *frame rate* is at least 24 frames

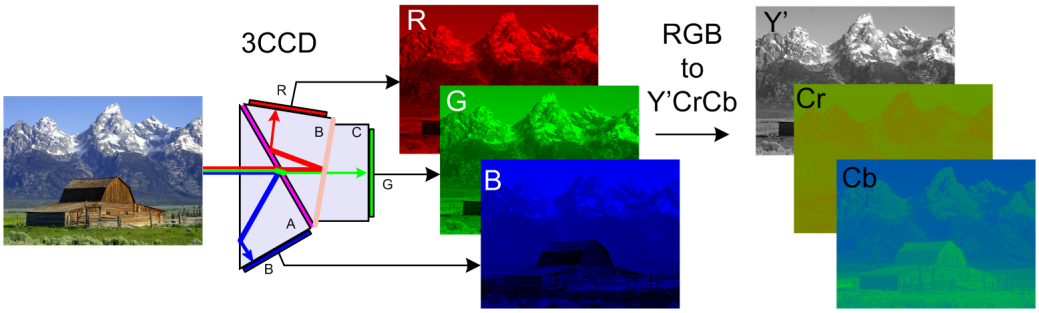

Figure 1.1: RGB to YCbCr conversion

per second (fps) [2]. Higher frame rates give a smoother-looking video, so many systems use frame rates around 50-60 fps. High-definition (HD) or Ultra HD content will often have frame rates ranging from 100 to 200 fps.

1.1.2 RGB, YCbCr and Chroma Subsampling

A light intensity function represents a greyscale image. To represent color we sample the light at different frequencies and then display the components together. Classically red, green and blue (RGB) are sampled which suffices to represent any color on the human visual spectrum [3]. However, in most modern applications, RGB is transformed to the YCbCr format. Here Y is the *luma* component, representing the overall brightness of the image, and Cb and Cr are *chroma* components, representing intensity in blue and red [2], see fig. 1.1.

Because the human eye is more sensitive to light than color [1, 2], in YCbCr the two color components are usually *subsampled* so that chroma values are averaged from a neighborhood of pixels, and subsequently less information is transmitted for the chroma than for luma [3]. This subsampling is conventionally written as a ratio between luma and chroma samples

$$Y : X_1 : X_2$$

with $Y = 4$ in most applications. X_1 describes horizontal subsampling, so $X_1 = 2$ means that 2 chroma samples are taken horizontally for every 4 luma samples. X_2 describes the vertical subsampling in relation to X_1. $X_1 = 0$ indicates that the same chroma subsampling is performed vertically as horizontally, while $X_1 = 2$ indicates no vertical subsampling is performed That is, the number of chroma samples are the same as the luma samples. See fig. 1.2. $4 : 2 : 2$ and $4 : 2 : 0$ are the most common subsampling formats [2].

In $4 : 2 : 0$, for every $2 \times 2 = 4$ luma pixels, only 1 pixel is stored in the two chroma components, meaning that a 1920×1080 video subsampled at $4 : 2 : 2$ contains only 960×540 chroma samples. Naturally, this decreases the size of the stored video. One full luma sample plus two chroma components each a one fourth of the luma size give a total of 1.5 samples, as opposed to the 3 full samples needed in RGB. Done correctly, chroma subsampling has little to no effect on the resultant video, and instead for the same file size as without subsampling the overall resolution and quality can now be increased.

But even with RGB transformed to YCbCr, storing or transmitting each pixel value is

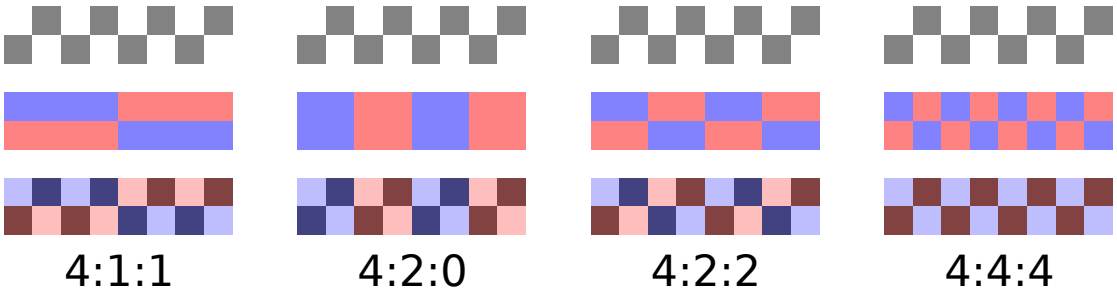

Figure 1.2: Different subsampling ratios

expensive in terms of storage space and bandwidth. 10 seconds of $4:2:0$ video content with 8 bit intensity values at 1920×1080 resolution and 50 fps will require $1.5 \times 10 \times 8 \times 1920 \times 1080 \times 50$ bits = 1.6 GB of data. A two hour movie requires 1.1 TB. This is why we need video compression.

1.1.3 Binary Numbers

To store and transmit data in any computer system, numbers are converted to *binary* format. Instead of ten digits, 0–9, binary deals exclusively with two numbers: 0 and 1. These can be more easily represented digitally. Both the decimal and binary systems gives significance to digits based on their position within the number. This is called a positional system. Take the decimal number 312, which can be re-written as

$$3 \times 100 + 1 \times 10 + 2 \times 1$$

which furthermore can be written as

$$3 \times 10^2 + 1 \times 10^1 + 2 \times 10^0$$

The internal logic of numbers is so ingrained into everyday thinking that we usually never need to worry about it, but binary conversion forces us examine the internal logic of a positional system. Because binary only has two digits, each positional exponent is now a 2 instead of a 10. 312 would in binary be represented as

$$1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

which is equal to

$$1 \times 256 + 0 \times 128 + 0 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1$$

and which we write as as 100111000_2. We use subscript to signal the number base, but this is usually left out when it is clear from context what number domain we are in. The conversion itself is non-trivial, and we won't go into detail on how to actually convert 312_{10} to 100111000_2. However, it can easily be verified that the binary sum above sum indeed adds up to the expected value, and it can furthermore be proven that every binary representation is unique.

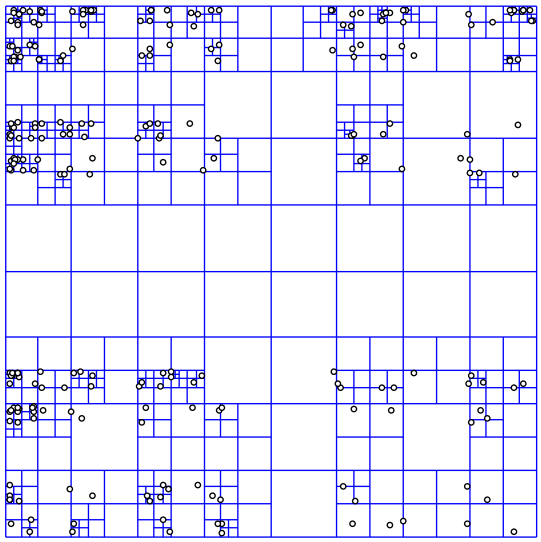

Figure 1.3: Quadtree splitting

1.2 Encoding Fundamentals

A video encoder is a program that takes some form of raw video and produces a *bitstream*, a sequence of bits, 0's and 1's. The encoder does this by going over the video frame-by-frame, often in a non-sequential order, using a multitude of algorithms to codify and store information using as few bits as possible. The decoder then turns the bitstream into pixel data again. High Efficiency Video Coding (HEVC), like most video coding standards, only specifies the behavior of its decoder. That is, a bitstream is said to conform to the HEVC standard as long as it can be decoded by its decoder, regardless of the quality of the video output. The implementer of an encoder is free to make choices about the coding as long as the resulting bitstream conforms to the standard – which naturally limits the freedom.

A certain portion of the encoding is *lossless*. That is, information is repackaged in a form so that it can be completely restored by the decoder [3]. This is done by exploiting statistical redundancies within the data. Arithmetic coding presented below is an example of lossless coding. Although the idea is obviously attractive, the potential gains from using only lossless encoding are far to small to be effective which is why lossy coding techniques are generally employed as well. Usually both are used together.

In *lossy* compression data is irreparably distorted. A good coding algorithm achieves large decreases in file size without sacrificing too much quality, and any decrease in bit rate is always weighed against the quality loss it introduces. Lossy coding techniques exploit not only statistical redundancies but also visual ones; things that look similar in a frame can be coded together to save space. The aim of the process is to remove information that in not visible to the human eye, while leaving most of what is visible [3]. Depending on the application, we may chose a smaller file size but with clear visual artifacts and distortions, and other times we want a compressed a image that looks almost identical to its original. Both are possible using lossy compression.

The first step of the encoding process is partitioning. The HEVC encoder divides the frame into slices that can be encoded in parallel, this is usually done once for the whole sequence. Then each slice goes through *quadtree* partionining. The slice is divided into

64×64 coding tree blocks (CTBs) [4], which are then recursively split into four equally sized subsections (see fig. 1.3). Both options – splitting and not splitting – are performed, the area encoded and the results are then compared to decided whether to continue splitting or to stop. For every split, this procedure is repeated for each of its four subsections, and each block can be split over and over again until minimum block size of 8×8 is reached [5]. Areas with sparse movement are easier to compress and will generally not be divided as much, forming larger continuous sheets in the quadtree structure [4].

1.2.1 Prediction

To radically cut down on the amount of information that needs to be stored, we usually try to generate a prediction of the image, which we then subtract from our frame and then we work with this residual instead. The prediction needs to be done in such a way that the decoder can reliably reverse the process and recreate the image. This means that only data that has already been encoded can be used for further predictions, because this is what will be available to the decoder at the corresponding point in the decoding process.

Although the encoder has easy access to the whole sequence, it limits its predictions to what it previously encoded to allow the content to be decoded. Of course, if the decoder cannot decode what has been encoded then the bitstream is worthless.

Intra Prediction

One prediction technique focuses on the internal relations between blocks in a image and is called *intra prediction*. Textures in a non-random image tend to look like each other [2] – if you zoom in enough on any structure in an image, many close-by pixels hold the same or similar color values – so the assumption is that a decent prediction can be generated by approximating blocks by their closest neighbors. The intra prediction concept is shared between still image and video compression.

Most of the information in the residual of an intra predicted frame will be contours of shapes in the picture [6], as those are hard to predict using this method.

Inter Prediction

A more powerful prediction technique for video content uses differences between frames and is called *inter prediction*. The basic assumption here is that objects within an image will appear in multiple frames but will move around. If an object from a previous frame can be identified and followed then this information can be used to make accurate predictions, see fig. 1.4.

Identifying an object from one frame to another means that a search has to be performed. This is a computationally complex process. While it is not impossible to search the whole frame for a best match, for any higher resolution video it is often unrealistic to do so and generally the search is confined to some nearby region of the previously known position [2]. Using a mean squared error (MSE) calculation [3], the closest match is identified, and a motion vector (MV) is stored in the bitstream to signal the displacement from one frame to the next. By storing MVs instead of pixel data, we avoid duplicating data

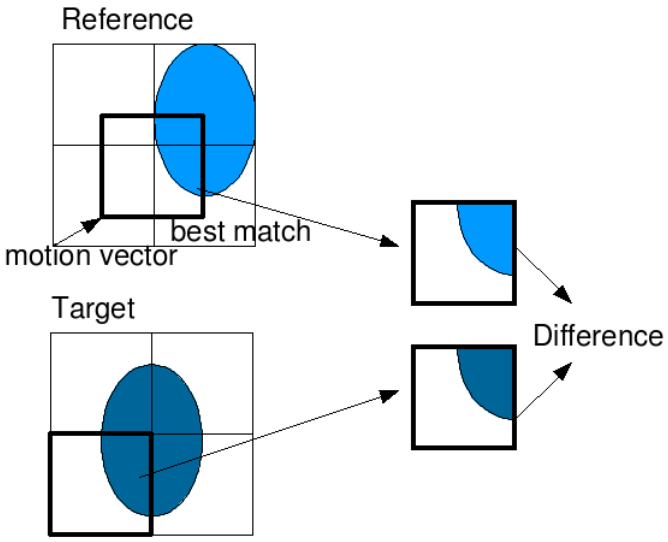

Figure 1.4: Inter prediction

from previous frames and can once again cut down the information content of the resulting file [3]. Calculating motion vectors is the most expensive part of encoding, especially because of the wealth of options offered by HEVC [5].

For both prediction techniques we won't generally achieve a perfect match, and the residual will contain our prediction error. However, our expectation is not generally to reduce the residual to nothing, only to make it as small as we possibly can. Using the residual data together with our prediction information we could still generate a perfect reconstruction of the encoded sequence. It is not until a later step that we start introducing distortion in the frame.

As opposed to intra-prediction, contours of stationary objects will mostly disappear from the inter prediction, and what is left are areas with hard-to-predict movement [6]. Accurate MVs give better prediction, but come at a certain signaling cost. MVs usually have an accuracy ranging from one pixel to a quarter pixel, so the vertical and horizontal displacements are correct up to one fourth's of a pixel's size [2, 3]. See fig. 1.5 for motion vectors superimposed on a frame.

Rate-Distortion Optimization (RDO)

It is up to the encoder to chose whether to encode each section of a frame using intra or inter prediction [2]. This is decided dynamically using techniques like rate-distortion optimization (RDO) [7], shown in eq. (1.1). D stands for distortion, R stands for bit rate and λ is a weight parameter that can be arbitrarily assigned to steer the encoding in a certain direction. We encode each option and chose the one with the lowest J, that is, the choice with the lowest distortion to relative the rate.

$$J = D * \lambda + R \tag{1.1}$$

Figure 1.5: Motion vectors superimposed on the frame

1.2.2 Frequency Transformation and Quantization

After partitioning and prediction, each block is frequency transformed using discrete cosine transform (DCT) [8], and then *quantized*. There exist a number of similar versions of DCT, but in image and video compression we generally use DCT-II, shown in eq. (1.2) [3].

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \qquad k = 0, \ldots, N-1. \tag{1.2}$$

Quantization means rounding the coefficients to some preset levels, decreasing the information content and allowing us to store the video at a decreased size. See fig. 1.6. We specify a quantization parameter (QP) to set the coarseness of the quantization, where a lower QP means less rounding. The quantization process introduces an irreversible distortion to the frame so that the original values cannot be recovered [2].

By its nature, a frequency transform divides the coefficient by their frequency content, low-frequency components are in the top-left corner and higher-frequency toward the bottom-right. The low-frequency components correspond to contours in the image and high-frequency components generally correspond to details. We utilize the fact that lower-frequency components are more visible to the human eye and thus use a higher QP for the higher-frequency components, effectively low-pass filtering the matrix [8]. Adjusting QP values allows us to specify exactly much degradation we are willing to accept in order to save bit rate – file size is traded for visual accuracy.

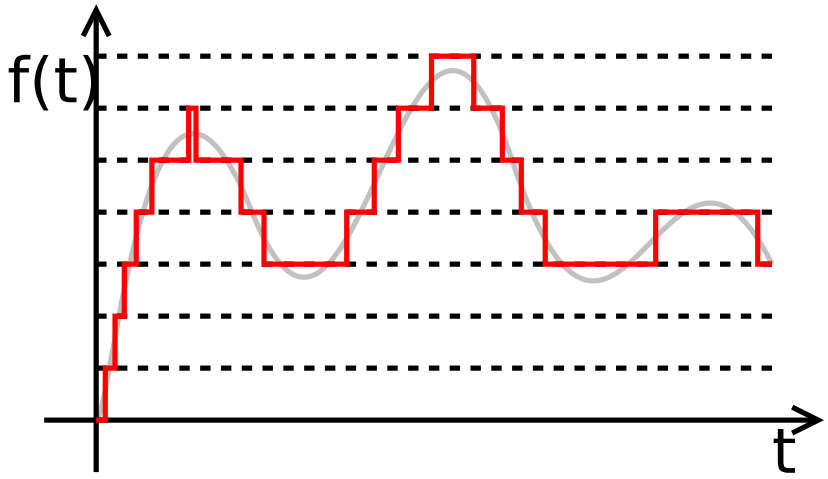

Figure 1.6: Quantization thresholds

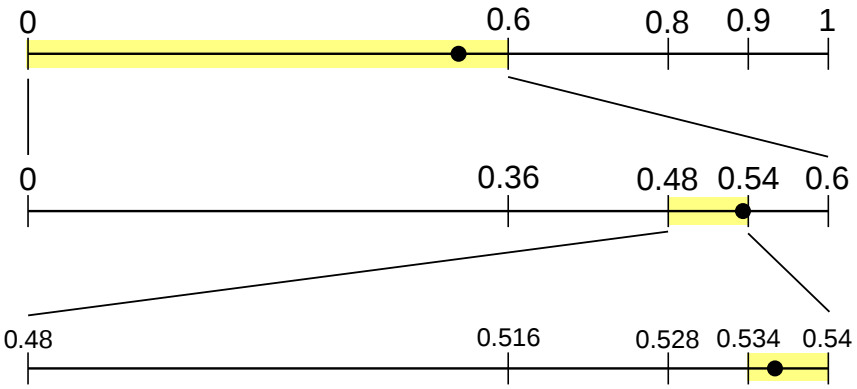

Figure 1.7: Arithmetic coding

1.2.3 Arithmetic Coding

The reason why quantization actually makes our files smaller is because we later encode sequences in a way so that shorter and more frequent numbers take up less space. In the pixel domain, using 8 bits, each value take up the same amount of bits, so 0 for example is represented as 00000000, which is wasteful.

Arithmetic coding is a lossless compression method where a string of symbols is encoded as a fraction in the interval $I = [0, 1)$. By using the probabilities of each symbol and recursively subdividing I, we eventually get a sub-interval uniquely representing our string (see fig. 1.7). By encoding the length of our original sequence and by then calculating a binary fractional expansion residing completely within the interval, we can uniquely reverse the process to decode our message. HEVC performs arithmetic coding using an algorithm called context-adaptive binary arithmetic coding (CABAC) [4], that was first introduced in H.264.

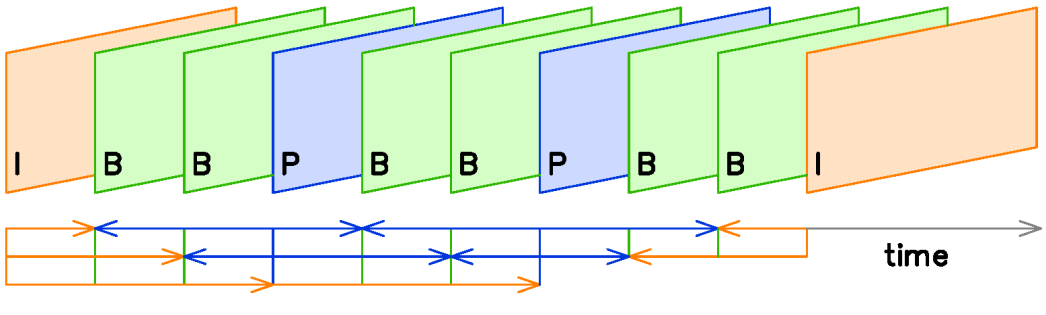

Figure 1.8: Different frame types in a sequence

1.2.4 GOPs and Temporal Layers

Encoded frames are divided into categories based on how they interact with other frames. Intra coded frames are called I-frames and they appear at the start of a sequence and at certain set points to allow the video to be decodable without having to always start the decoding from the very beginning. Most frames are predicted from other frames and are called P or B-frames. A P-frames uses one other frame for its prediction and a B-frame use two, usually one before and one after itself. See fig. 1.8.

An encoder divides the input video into groups of pictures (GOPs) when encoding. A GOP is treated as a unit by the encoder and can be usually be decoded independently from the rest of the video. This allows the viewer to skip ahead to any given time in the video without having to decode everything prior to it. While larger GOPs generally give better encoding performance [9], GOP sizes are generally chosen so that every second or half-second of video content is independently decodable.

Inter-coded video is generally not encoded in the frame order that the material was captured in. We distinguish between *output order* and the *decoding order* [2]. The GOP structure specifies a coding order which is then repeated cyclically throughout the sequence [2].

The GOP structure also divides frames into temporal layers [10]. Intra coded frames form temporal layer 0 because they depend on no other pictures. Those that reference layer 0 pictures form layer 1 and pictures that depend on layers 0 and 1 are temporal layer 2, and so on. A popular GOP structure, *Randomaccess*, has 4 temporal layers. The QP values are usually increased slightly for the higher temporal layers. The rationale is that these generally contain less data, so compressing them slightly more won't have much of an effect on the sequences as a whole.

1.3 HEVC-Specific Algorithms

1.3.1 RDOQ

Rate-distortion optimized quantization (RDOQ) is a method for improving coding efficiency by manipulating transform coefficients after quantization. Out of the available quantization levels, we try different values and perform RDO to decide the best value to re-assign each coefficient. Testing all possible combinations of quantization steps and

coefficients in for example a 16×16 matrix would very computationally heavy so instead we go through the matrix one coefficient at a time from the bottom-right and backwards, setting each on the all possible levels or just a subset of its closest neighbor values.

1.3.2 Sign-Bit Hiding

Sign-bit hiding (SBH) is a lossy technique for decreasing bit rate. To avoid transmitting the sign for the top-left element in a block of transform coefficients, it is instead inferred from the parity of the sum of all coefficients. Because transform coefficients are represented in binary form, and all even binary numbers numbers end with 0 and all odd ones end in 1, only the last bit of each coefficient needs to be taken into account.

Assuming that coefficients are equally likely to be even as odd, this parity is going yield the incorrect result half the time. To counter this, the SBH algorithm performs RDO over the transform block to find the value that would have the least impact on the resultant image if changed up or down by one, and then performs this change. That is, in order to save one bit on every transform block, a small error in introduced in half of the encoded blocks. The result is typically well worth it. Small losses in quality are traded for larger decreases in bit rate.

1.4 Adaptive Streaming

Streamed video-on-demand content is usually delivered using adaptive bit rate (ABR) [11]. Depending on device capabilities and bandwidth, different users will request different-quality content. A user that starts out watching a video when bandwidth usage is low will generally be sent high-quality video. If the available bandwidth then decreases – maybe by more users entering the network – the streaming service needs to dynamically lower quality to alleviate the network. ABR means that the transmitted quality is adapted to the receiver's needs.

Video cannot be downscaled in encoded form. Thus, to create a lower-resolution encoding of a video, it has to be decoded, downscaled and then re-encoded. This process is known as transcoding. The encoding step takes a lot of time, so to be able to do adaptive streaming, the service generally creates all encoding beforehand and stores them until they need to be transmitted.

We present here three models for adaptive streaming: simulcasting, just-in-time (JIT) transcoding and scalable video coding (SVC), and discuss their respective weaknesses. In the next chapter we will introduce guided transcoding (GT) as way to get around some of the inherent problems of the current streaming models.

1.4.1 Simulcasting

The straightforward way create adaptive video content is that whenever a new video is ingested in the *adaptation node*, we scale it to each desired size and encode (see fig. 1.9). The original video is then thrown away.

This model is known as upfront transcoding or *simulcasting* [11]. A streaming service offering 1080p, 720p, 540p and 360p versions of all of its content will have one encoding

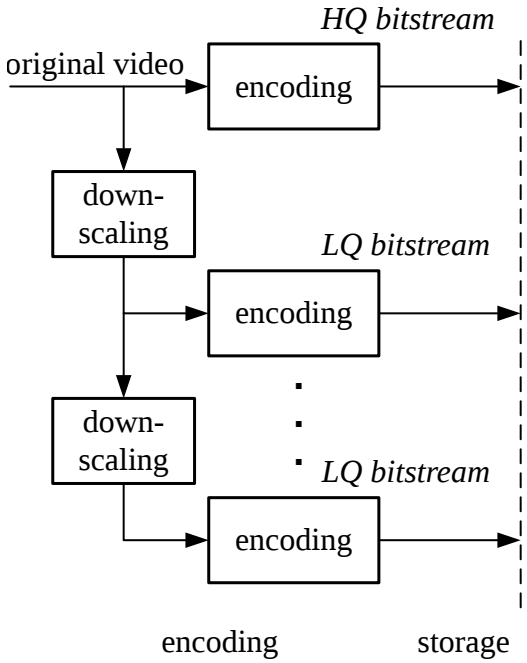

Figure 1.9: Simulcasting

for each resolution stored away and ready for transmission. This takes up a lot of space [5], and encodings are generated without prior knowledge of customer demand.

1.4.2 Just-in-Time Transcoding

The problem with simulcasting is that we may end up storing many files that are seldom or never actually requested [11]. A more attractive idea might then be to encode the original only at the highest resolution, and later transcode it to the lower resolutions on-demand (see fig. 1.10). The generated lower-resolution encodings may be kept as long as demand is high and then thrown away. This model is known as JIT transcoding.

The general problem with JIT transcoding is that encoding takes a lot of time to perform. Even with specialized hardware, the process of encoding thousands of frames is generally too slow for the idea to work in practice [11, 5]. Unless encodings can be generated within minutes, JIT transcoding is not appropriate for video streaming. For example, the idea or ordering a movie at 540p and having to wait a day for the content to be available would likely scare away most costumers in a world where proper on-demand streaming already exists.

Another problem with JIT transcoding is that the lower-resolution content will be generated from an already-degraded data source. In simulcasting, each encoding is generated directly from the original, but now the highest-resolution encoding acts as an original for the transcodings. This introduces *generation loss*; coding artifacts and high-resolution noise that are amplified by successive transcoding, and lowers the video quality [12], as well as unnecessarily increasing its size. This problem is shared with our pruning approach to GT (see section 2.1).

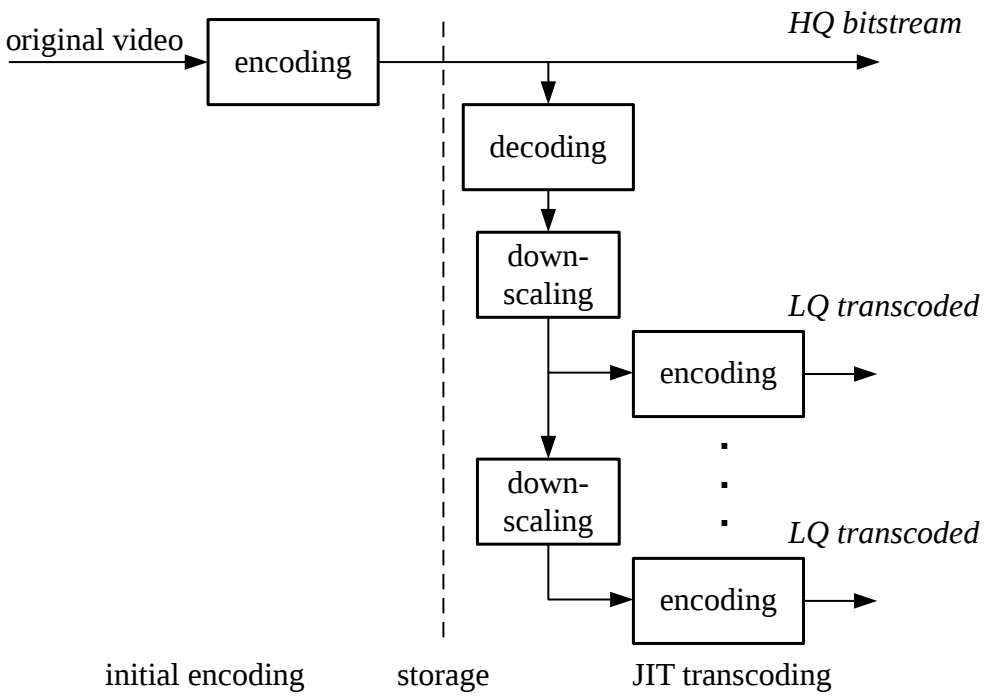

Figure 1.10: Just-in-time transcoding

1.4.3 Scalable Video Coding

Scalable video coding (SVC) uses a layered model to encode different-quality representations of the same material. Starting with a base layer, which is a regular bitstream, one or several enhancement layers are encoded with reference to the base. Enhancement layers can only be decoded after first decoding the base layer. Often, the different layers represent varying resolutions, with the base layer holding the lowest resolution and each enhancement layer representing a higher resolution.

Decoding a higher-resolution SVC layer requires access to the base layer together with every intermediate enhancement layer. Decoding an enhancement layer means the previous layer is decoded, upscaled to the size of the enhancement layer and used for prediction. The process in repeated for every subsequent layer until the target resolution is reached. The term SVC stems from the H.264 standard and is often known as SHVC (Scalable high efficiency video coding) in H.265, but we will use the terms interchangeably and refer to both as SVC.

The major problem with SVC is that the dependent structure causes a size overhead when encoding the enhancement layers. While storage space can be saved on the adaptation node compared to simulcasting, any higher-resolution videos transmitted to the customer will typically take up more space in SVC than in a regular encoding [5]. SVC generally performs better at the lower resolutions – for example, adding a new lowest-resolution encoding to the scheme moves every enhancement layer further away from the base and actually lowers coding efficiency. As such, the whole idea of SVC is in many ways out of step with the modern usage were the demand is on HD, Ultra HD, and beyond.

1.5 My Contributions

The idea of guided transcoding is not a novel one. [5] explores pruning as an idea and compares it to simulcasting, JIT transcoding and SVC for a single lower-quality representation. No mention, however, is made of any partial techniques.

We expanded these experiments to a large scale with our simulation environment, where we test a wide array of downscaled sizes and QP values, as well as using a workaround to be able to simulate the effects of SBH. Deflation as an idea is presented for the first time in this thesis report.

Partial pruning and deflation were presented to me on the conceptual level by my Ericsson supervisors. I then implemented the bit manipulations in low-level C and also wrote thousands of lines of Python to run simulations and to automatically export the results to Excel sheets.

Chapter 2
Guided Transcoding

Guided transcoding (GT) aims to get around some of the problems presented in the earlier models of adaptive streaming (see section 1.4). Instead of working with full encodings of the low-quality (LQ) representations, we generate side-information (SI) which is cheaper to store, and then use our highest-resolution video together with the SI to quickly regenerate the sequences when needed.

"Quick" is this case means that the computational complexity of the regeneration is comparable to that of regular decoding, several orders of magnitude faster than encoding. Because the process is fast – it can be done in real-time under certain conditions – we can keep our lower-resolution videos stored only as side-information, and regenerate the full sequence on-demand when requested, thus achieving what is generally not possible in the JIT transcoding scheme.

So GT achieves the virtues of both upfront and JIT transcoding; we save space by allowing us not to store full encodings of every resolution, but the process is fast so we can generate content on short notice. Guided transcoding can also be seen as a reversal of SVC in that we save the high-quality (HQ) video and use that to regenerate LQ representations, instead of the other way around with a base layer and enhancement layers.

2.1 Pruning

Our first model for generating SI is referred to as *pruning* and was first described in [5]. Pruning a bitstream means passing it through a modified decoder that removes transform coefficients and replaces them with sparse dummy matrices that are cheaper to encode. What we are left with is the *mode information*; the frame partitionings and prediction modes chosen for each block, together with motion vectors for every inter coded block.

See fig. 2.1 for a pruning scenario with two lower resolutions. We generate LQ encodings just like in a simulcast case, prune them and then store the pruned bitstreams to save space. The blocks on the left of the dashed line represent the steps that are performed

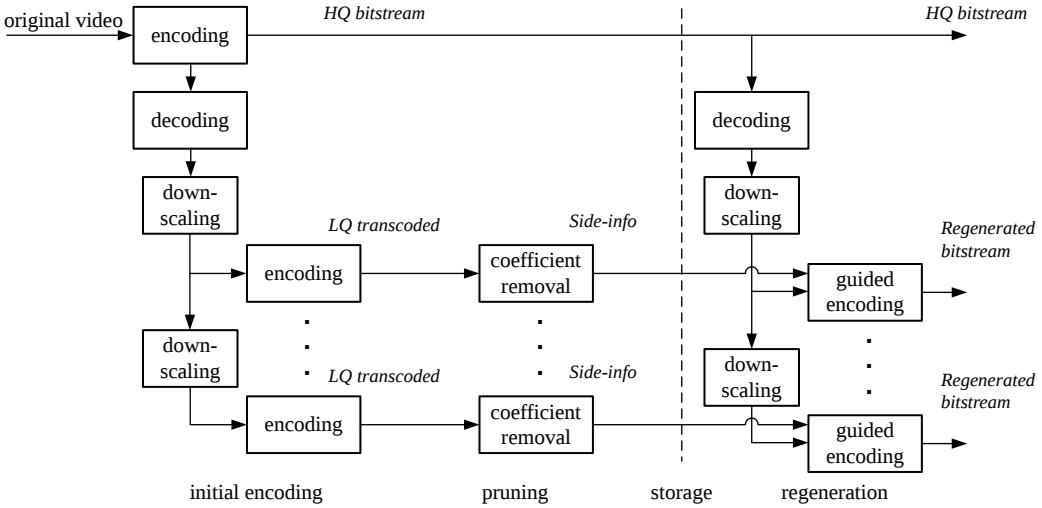

Figure 2.1: Guided transcoding in the pruning scenario

without time constraints. At a later stage, we use the mode information to recreate the transform coefficients and give a perfect reconstruction of our pruned video. The vertical dots represent repetition. We show two LQ representations, but the idea can easily be expanded two an arbitrary amount of representations.

To achieve this, we replicate the process that was used to create the LQ video in the first place. The HQ bitstream is decoded and downscaled and its pixel data is encoded. Because we have all partitionings and mode decisions stored in the SI, the process is now much less complex. Even without the SI, we could encode the downscaled HQ video and get our LQ bitstream back. Encoding the same video twice leads the encoder to make all the same mode decisions to create the same bitstream. What that SI allows us to do is to create the LQ encoding without having to calculate all the encoder decisions. This is why it is so much faster, the SI *guides* the encoding process, thus giving the method its name.

2.1.1 Generating Side-Information

Because we won't have access to the original video when regenerating sequences – it is generally discarded after being used to create the HQ bitstream, and takes far too much space to store long term – we treat our HQ bitstream as an "original" and transcode it to generate LQ representations.

That is, we decode the HQ bitstream, downscale it to each desired LQ resolution and re-encode. Downscaling is performed by passing a decoded video to a downscaler program. This decreases the number of total pixels in the image, lowering its quality and file size while still preserving the overall "look" of each frame. To achieve this, groups of pixels are mapped to single values. Downscaling by a factor of 2 means that each 2×2 block is replaced by a single pixel. The downscaler looks at a number of adjacent values in the input and calculates a weighted average that it assigns to the pixel value in the output video. Each of the three components in a color image are downscaled separately and the subsampling ratio (see section 1.1.2) is always preserved.

We pass the LQ representation to the pruner and then store the SI instead of the full

Figure 2.2: A decoded frame from a pruned bitstream

encoding. The pruned bitstream is still decodable, but because all transform coefficients have been removed – most of them set to 0 – the video will look far from normal. The screen shows contours of objects, with what looks like very heavy-motion blur, and colors distorted toward pink and purple (see fig. 2.2).

Because we transcode an already degraded bitstream to generate the LQ representation – the HQ representation is a lossy encoding of the original – we introduce generation loss. This requires us encode the LQ bitstreams at a higher bit rate than we would in the simulcast scenario, to achieve the same picture quality.

2.1.2 Regenerating a Pruned Sequence

As mentioned, regenerating a pruned bitstream shares many similarities with regular encoding. The main differences is that we now have two input files instead of one.

The higher-resolution video must be in decoding order, meaning that the frames are output in the order in which they were encoded, not in the order they are meant to be displayed. Non-sequential GOP structures cause the decoder to buffer certain images, use them for predictions, and then output them later when it is their turn to be displayed. We suspend this buffering using a specialized decoder to allow the HQ video to sync up with the SI, which is in encoded form, and thus naturally in decoding order.

Regenerating the pruned video means encoding the pixel data from the HQ representation using the mode information from the SI to steer the process. Both the pixels and the mode info are the same as were used generate our LQ bitstream in the first place, so this process will yield a perfect reconstruction of the sequence.

2.1.3 Partial Pruning

Even with the SI to guide the transcoding process and speed up the process, depending on the hardware that the guided transcoding is running on, it may still be too complex to perform in real-time. This is why we introduced the concept of *partial pruning*. Instead of removing all transform coefficients from a sequence, we prune only certain frames and

Figure 2.3: A decoded frame from a partially pruned bitstream

thus save complexity in the reconstruction.

Our partial pruning scheme is based on temporal layers, and we designate different *pruning levels* based on which layers are excluded from the pruning. Partial pruning level 1 means that the highest temporal layer is excluded, level 2 excludes the top two layers, and so on. The higher temporal layers consist of P-frames and B-frames (see section 1.2.4). They reference other frames for their predictions, and will generally contain fewer transform coefficients. Thus, our expectation is that partial pruning level 1 or 2 will have a minimal effect of bit rate while still saving a significant amount of complexity in the reconstruction. Then the effectiveness tapers off as we increase the pruning level.

A partially pruned frame at level 3 can be seen in fig. 2.3. It still has the characteristic look of pruning, but certain pixel values are now retained to hint at the appearance of the original frame.

2.2 Deflation

A second guided transcoding idea is introduced in this thesis and we call it *deflation*. In this scenario we encode our LQ representations directly from the original video, so there is no transcoding loss like for pruning. See fig. 2.4.

2.2.1 Generating Side-Information

To deflate a video, we use two inputs. The mode information from the LQ bitstream together with the pixel data from the HQ video are used to generate predictions and calculate a residual. This residual is frequency transformed and quantized (see section 1.2.2) and then subtracted from the coefficients in the LQ bitstream. Because the HQ and LQ videos emanate from the same original, this difference is expected to be small and therefore cheap to encode.

In fact, some transform block in the deflated bitstream are likely to be all-zeros, and HEVC does not allow that. So we identify all matrices of the form shown in eq. (2.1) and increase the k coefficient by one to create a compliant bitstream.

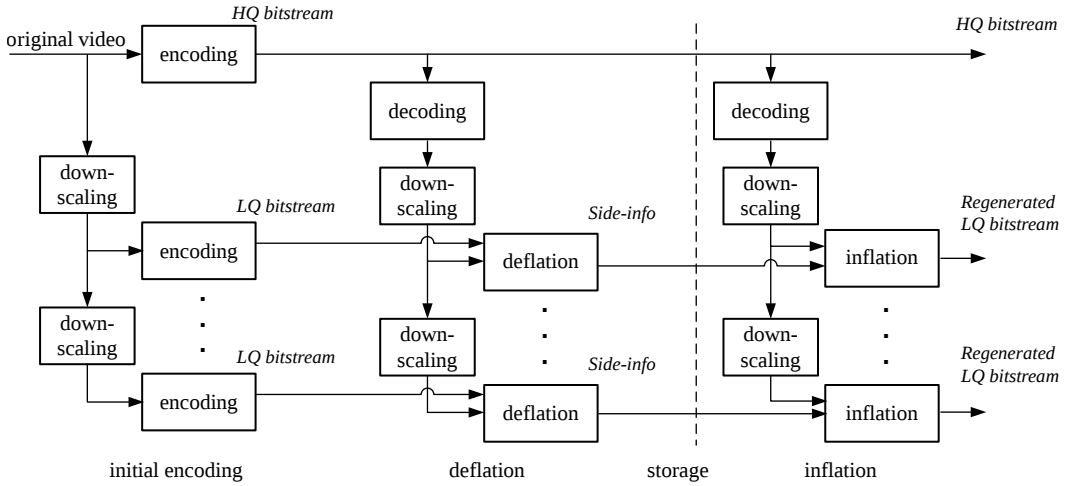

Figure 2.4: Guided transcoding in the deflation scenario

This scheme maps the highest possible value – `SHRT_MAX` because we are storing transform coefficients in variables of type `short` – and the one just below it to the same position, potentially introducing a small distortion. However, most values will actually residue around 0, especially this lowest-frequency component, and this distortion has never appeared once in any of our simulations.

$$\begin{pmatrix} k & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}, \quad 0 \leq k < \texttt{SHRT\_MAX} \tag{2.1}$$

2.2.2 Regenerating a Deflated Sequence

Reversing the deflation is called *inflation* and replicates many of the steps of the deflation process. We reverse the last step from the deflation by looking for matrices of the form eq. (2.1), but with $k \geq 1$, and subtract one.

Because the mode information from the LQ is untouched by the deflation, we can once again downscale the HQ video and generate a residual. After frequency transformation and quantization we re-add the transform coefficients to the deflated bitstream and thus restore our LQ video. Just like in the pruning scenario, the SI guides the encoding of the pixels from the HQ video, so the process is much faster than a full encoding.

Chapter 3
Evaluation

3.1 The Guided Transcoding Chain

Our test environment for running GT simulations has many components. A full simulation chain entails several encodings at multiple QP values, decoding and downscaling each encoding to a number of different sizes, re-encoding, generating SI and reconstructing the sequences. Additionally, we want to measure bit rate and peak signal-to-noise ratio (PSNR) at several points in the simulation, and be able to access all the data in a structured manner to try to make sense of it and to compare different simulations.

We wrote a test environment in Python that allows us to easily specify sets of QP values and downscaled sizes, together with GT schemes; pruning, partial pruning or deflation, and options like SBH and RDOQ.

3.1.1 Encoding the Original

The pruning and deflation scenarios differ in the way side-information is generated but many parts of the chain works the same way, using most of the same programs. We first describe the GT approach common to both methods and then elaborate on the details of each case.

The first step is always to encode the original test sequences. Our suite uses five HEVC sequences that belong to the Joint Video Team (JVT) common test conditions, class B: Kimono, ParkScene, Cactus, BasketballDrive and BQTerrace. Still frames can be seen in fig. 3.1. They represent a wide variety of possible use-cases for video encoding: varying degrees of movement, static camera and panning movements, and frame rates varying from 24 to 60 fps. All sequences have a bit depth of 8 bits and 1920×1080 resolution [2]. Each clip is ten seconds long, giving frame count between 240 and 600 which JVT considers sufficient to get a qualified assessment of video quality, while still keeping the required encoding complexity sufficiently low [2].


Figure 3.1: Still frames from our HEVC standard sequences

Some of our sequences have a copyright frame as the last one, giving a total of 601 frames for a 60 fps video. This frame frame is not expected to used in simulations, and we exclude it from all of our encodings and references, and it should therefor have no effect on our results.

All the videos we work with are stored in progressive scan, meaning simply that each individual frame contains information about all of its own pixels, as opposed to interlaced scan where a frame is split over two successive frames, each one holding half. We will follow the convention regarding scan modes and refer our resolutions as 1080p, 720p, etc., thus omitting the frame width.

For each original video we create four encoding with QP values 22, 26, 30 and 34, and refer to these as QP_{base}. All encoding and later re-encoding are performed using the HEVC Test Model (HM) encoder with a configuration file specifying *HEVC Main profile* and the *randomaccess* GOP structure [2]. Randomaccess has a GOP size of 8 and coding order 0-8-4-2-1-3-6-5-7, after which it skips to frame 16, repeats the same pattern, and then continues like that for the whole sequence [2].

All of the outputs from the initial encoding step are referred to as HQ bitstreams. Generating these is by far the most expensive part of any simulation in terms of computational complexity, taking around 15 hours on our cluster environment (see section 3.2), so we make sure to store them and always try to re-use bitstreams for any simulation where all the applicable test parameters are the same.

3.1.2 Re-Encoding and Pruning

For the pruning scenario, the next step is to decode the HQ bitstreams and downscale from 1080p to 720p, 540p and 360p representations which we refer to as the LQ resolutions. Our downscaler only supports downscaling to 1/2 or 2/3 size, so to generate a 360p video from 1080p we always have to get there via an intermediate 720p step.

We re-encode with the same QP value and with $QP + 2$. That is, for an HQ bitstream generated with $QP = 22$ we create re-encodings with QP values 22 and 24, for HQ bitstreams generated with $QP = 26$ we use 26 and 28, and so on. This gives us a wide array of test cases to accurately quantify the effect on QP value on video quality, bit rate and reconstruction time.

After re-encoding, we prune the LQ bitsteams to generate SI. We either do partial or full pruning, the difference at this stage is just an input parameter. In an real real-world application we would then store the SI instead of the LQ bitsteam, saving a certain amount

of storage space. Of course, for the sake of the simulation, we keep both. We sometimes refer to the SI and the bitstreams as *uplink data*.

We also downscale the original sequences to the LQ resolutions for use as reference data when calculating PSNR, and then encode them to act as LQ bitstreams in the deflation scenario. For this we use all applicable QP values; 22, 24, 26, 28, 30, 32, 34 and 36, which we refer to as $QP_{extended}$.

3.1.3 Regenerating Sequences

The HQ bitstreams together with the SI is used to reconstruct videos. For each test case, we reconstruct the video to make sure everything works as expected, then measure bit rate and PSNR.

3.2 Cluster Simulations

For any simulation of five tests sequences, four QP values, three LQ resolutions and two additional QP values, we get a total of $5 \times 4 \times 3 \times 2 = 120$ test cases.

Each combination is submitted to a cluster system as a self-contained *job*. The cluster allows for faster calculations than running locally, and allows hundreds of jobs to run in parallel without affecting performance. The cluster is shared among many users and a has its own scheduler. One job will for example correspond to BasketballDrive, QP 22, 540p and QP 24, and will then only be concerned with the creation of the specific files needed for that test case.

A separate meta-script is responsible for iterating over simulation parameters and starting each of the 120 jobs that make up one simulation. Jobs will then work in parallel, sharing many of the same files, and together generate every combination of files needed to evaluate a full simulation. This script also creates a test_data file to keep track of the locations of all data files holding bit rate and PSNR information.

To allow jobs to work in parallel and read and write from the same directory structure without destroying data, we implemented a locking system. Because cluster jobs cannot directly communicate with each other we could not utilize traditional threading. So we represented locks as empty files named as the target file plus the extra file extension ".lock", it is then up to each script to respect the lock. All the jobs share the same storage area, but also have private tmp areas where files are created. Whenever a job wants to create a file, it attempts to lock it in the storage area. If the lock creation fails because the lock file already exists then we assume that some other job is busy creating that file, and we enter a sleep loop that regularly checks the existence of the lock and only exists when the lock has been removed, at which point our desired file must exist in the storage area.

The execution works as follows; if two jobs both have parameters BasketballDrive and QP 22, but one has LQ resolution 720p and the other 540p, then both will want to create an HQ bitstream of BasketballDrive with QP 22. Assuming that this file does not already exist from a previous simulation – if it does then both scripts move on the their next step – both jobs will try to acquire the lock but only one will succeed. The job that grabs the lock starts creating the bitstream in its tmp area while the other jobs sleeps. After the file has been created, the job moves the file over to the public storage area and then removes

the lock. This way, no in-progress or incomplete files will ever exist in the storage area, and furthermore we won't have to worry about race conditions in the code. If the file is not locked – and it exists in the storage area – it is guaranteed to be complete.

Especially early on in a simulation chain, many jobs will share the same files. Only the LQ encoding is actually unique to a specific job. For example, out of 120 jobs, evenly divided groups of 24 jobs each share the exact same HQ bitstream file.

Every file in the simulation chain follows the same pattern of creation; the job checks for the existence of the file in the storage area and whenever it find it there, it swiftly moves on to the next step. If all files are already created, the jobs runs through the whole chain without actually creating any files and then exits cleanly. This structure allows for maximum file reuse whenever we want to test new simulation parameters. If we run a simulation where a parameter that only affects the latter part our chain has been changed, the jobs will find many of its files already present and won't have to spend any to recreate them. In many simulations we thus can avoid the encoding the HQ bitstreams, for example, cutting down the total simulation time by many hours.

Figure 3.2 shows an excerpt from the directory structure of one our or simulations. Notice how the folder names contain all the test parameters so that each file can be uniquely addressed. `Bin` files are bitstreams and `yuv` is code for YCbCr so these are decoded files. Through clever naming of files and folders, each job will always know which files already exist and which ones it needs to create. We also store test data in the same directory structure. We extract bit rate and PSNR into txt files during execution, and using the information written to test_data when starting the simulation we can later navigate the directory structure to locate all data files.

3.3 Simulation Data

3.3.1 Measuring Bit Rate

Bit rate is an absolute measure of file size per time unit of video content. It is often presented in kilobits per second (kbit/s). We use the size of each output file to calculate an average across the whole sequence. Because file sizes in most operating systems are presented in bytes, we get the number of bits as in eq. (3.1). To calculate the bit rate (in kbit/s) we then use eq. (3.2).

$$\text{bits} = \text{file size} \cdot 8 \tag{3.1}$$

$$\text{bit rate} = \frac{\text{bits} \cdot \text{framerate}}{\text{frames} \cdot 1000} \tag{3.2}$$

3.3.2 Measuring Video Quality

The common way to measure video quality objectively is to use PSNR [2]. For two sequences emanating from the same source both with the same resolution $n \times m$, referred to as I and K, usually the original and some encoded version of it, we take the difference

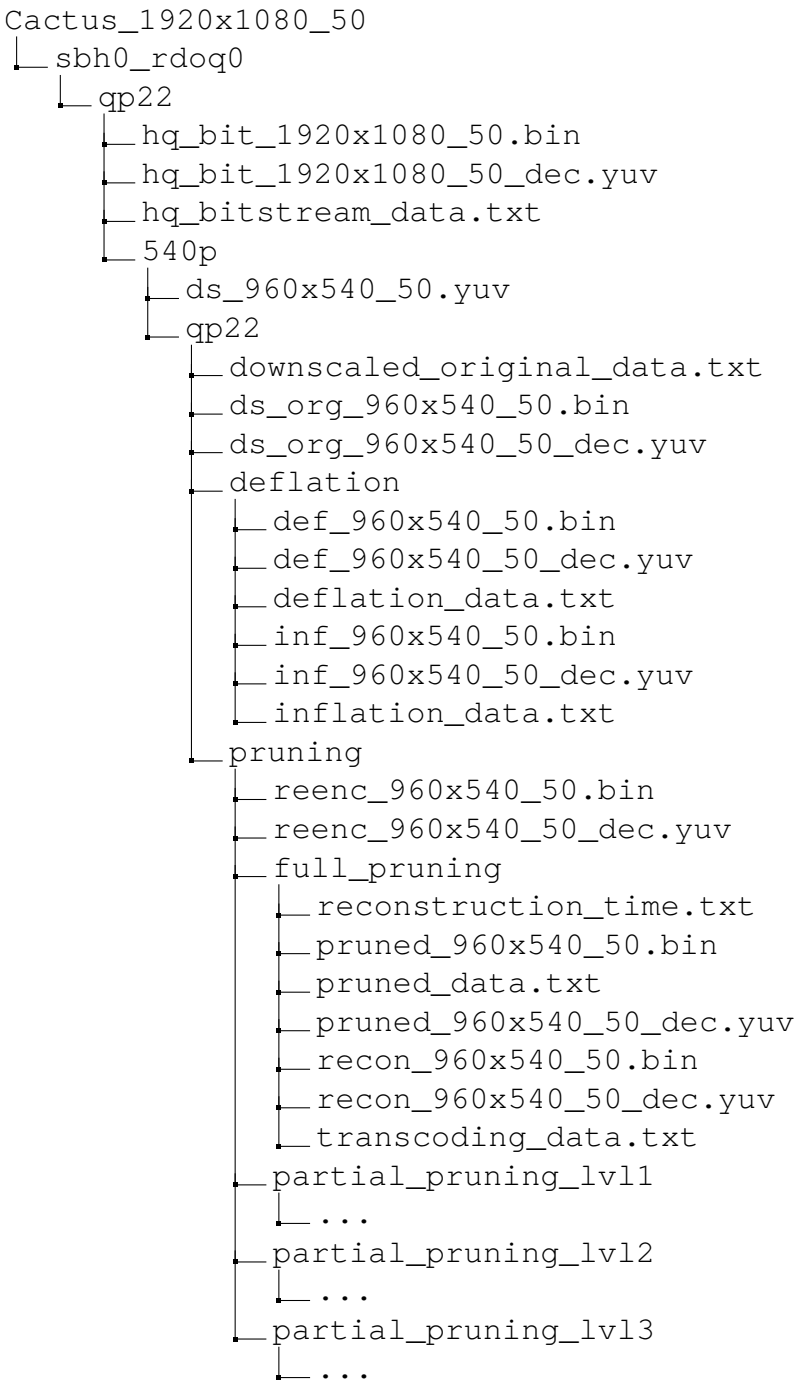

Figure 3.2: A small excerpt of the storage tree

per pixel and calculate the sum across the whole frame to get the MSE. This is shown in eq. (3.3).

$$MSE = \frac{1}{m\,n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2 \tag{3.3}$$

The PSNR per frame is then calculated as in eq. (3.4), where MAX_I is the maximum value of the intensity function; $2^8 - 1 = 255$ for an 8 bit image. To get get the PSNR for the whole sequence we take the average over all frames. PSNR is calculated separately for the luma and two chroma components. We save the data for all three, but we are generally only interested in the luma PSNR.

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \tag{3.4}$$

The more similar two sequences are – the less distorted the encoded version is – the higher the PSNR value will be. PSNR as a number carries no significance in itself, but a higher PSNR is always better, so the relative difference between two PSNR values is meaningful. We always measure the PSNR against the original sequence, which means that each HQ bitstream is decoded and compared, and the LQ encodings are compared to downscaled versions of the original.

3.3.3 Gains

We refer to bit rate savings as gains and calculate the *GT gain* using eq. (3.5) which represents the disk space we can save for all ABR resolutions by storing the SI instead of the LQ resolutions.

Here R_{HQ}, R_{LQ} and R_{SI} represent the bit rates for the HQ, LQ and SI bitstreams respectively. For the pruning scenario, R_{LQ} always refers to an interpolated bit rate, see section 3.3.4.

$$\text{GT gain} = \frac{(R_{HQ} + R_{LQ}) - (R_{HQ} + R_{SI})}{R_{HQ} + R_{LQ}} = \frac{R_{LQ} - R_{SI}}{R_{HQ} + R_{LQ}} \tag{3.5}$$

The *max gain* represents the theoretical upper limit of any guided transcoding application where the SI is decreased to nothing, and is calculated using eq. (3.6). The ratio between the GT gain and the theoretical maximum gives a good indication of how effective the method is.

$$\text{Max gain} = \frac{R_{LQ}}{R_{HQ} + R_{LQ}} \tag{3.6}$$

We also measure *rate reductions* showing us how much bit rate we can save per sequence by storing the SI instead of the LQ bitstream, calculated using eq. (3.7).

$$\text{Rate reduction} = \frac{R_{LQ} - R_{SI}}{R_{LQ}} \tag{3.7}$$

3.3.4 Costs

Using QP_{base}, we calculate a third degree polynomial to interpolate the bit rate required for the LQ re-encodings, in the pruning case, in order to achieve the same PSNR after transcoding as the simulcast case has [13]. We refer to the increase in bit rate caused by transcoding as *cost* and it is defined in eq. (3.8), where $R_{LQ_{transcoded}}$ is the bitrate of the transcoded bitstream and R_{LQ} is the bit rate of the directly encoded bitstream.

$$\text{Cost} = \frac{R_{LQ_{transcoded}} - R_{LQ}}{R_{LQ}} \tag{3.8}$$

R_{LQ} is interpolated to match the PSNR of the transcoded bitstream. We calculate interpolation coefficients using the bit rates from the simulcast reference data together with the PSNR values for the different LQ representations amd then plug the pruned bit rates in to get an interpolation. In the data in tables presented below, the *Average bitrates* sections contain our interpolated bit rates. In the deflation scenario the cost is always 0.

3.4 Results

3.4.1 Excel Sheets

One full simulation means running the chain of 120 test cases for a given set of test parameters; pruning, partial pruning or deflation, together with options like SBH and RDOQ.

To compare different simulations we wrote a big Python script to extract all the test data into an Excel sheet using the *openpyxl* library. This allows us to automatically calculate bit rate reductions, GT gains and losses introduced by re-encoding data. We average all the data to get values per sequence and size.

3.4.2 Pruning

We simulated full and partial pruning levels 1-3 and measured reconstruction time, which can be seen in tables 3.1 to 3.4.

To measure regeneration time, three operations need to be performed; decoding the HQ bitstream, downscaling it, and regenerating transform coefficients. We used an Intel Core i7 3.3 GHz processor that we forced to run on a single execution thread. To get accurate timing data, these three steps have to be continually re-done and the intermediate files thrown away between iterations. For the GT scheme to work realistically, the three steps should to be done is real-time or at least close to this, so we want to regenerate frames at a higher fps than that of the video.

Our chain cannot fully handle RDOQ and SBH. In both the pruning and deflation scenarios, at least one program malfunctions or gives worse results with either option turned on. However, both method generally give better results, so we would like to include them in our simulations. For the pruning case it is only the regeneration causing problems with SBH, so utilizing the fact that the reconstruction is a perfect process – this is asserted programmatically every time a pruned file is reconstructed without SBH – we can use the

bit rates and PSNR data calculated for the LQ bitstreams instead. This way, we can present data for a SBH simulation that we were not actually able to run, see tables 3.5 to 3.8.

Naturally, these have no time measurements. RDOQ is always turned off in the pruning simulations.

3.4.3 Deflation

The deflation simulations are shown in tables 3.9 and 3.10. We did not have time to implement RDOQ in the deflator so having it turned on for the HQ encoding introduces a discrepancy between the two videos, lowering the effectiveness of the scheme. In this scenario neither the deflator or inflator work with SBH. Thus it is turned off, and we cannot use the work-around from the pruning case. Given all of this, the only fair comparisons between the two methods are given for SBH and RDOQ turned off.

We have no timing data from the deflation scenario because the algorithm was never optimized for efficiency. While our inverse pruner heavily utilized parallelization, the inflator was built for correctness first, efficiency second. However, simple simulations confirm that the complexity is somewhere in the range of inverse pruning, much closer to decoding than actual encoding.

Table 3.1: Full pruning (SBH off, RDOQ off)

Average bitrates									
	QP_{base}				QP_{base} + 2			**Totals**	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2419	1043	760	475	835	593	362	33.9%	62.7%
ParkScene	3552	1341	888	472	1053	678	354	26.9%	57.7%
Cactus	8283	2799	1933	1118	2242	1522	868	28.3%	58.9%
BasketballDrive	8058	2852	2031	1207	2300	1605	938	26.7%	60.0%
BQTerrace	13605	2585	1718	889	1949	1295	664	25.2%	48.6%
Averages	**7183**	**2124**	**1466**	**832**	**1676**	**1139**	**637**	**28.2%**	**57.6%**

Rate reductions							
	QP_{base}			QP_{base} + 2			**Totals**
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	55.7%	54.0%	52.6%	55.2%	52.7%	50.2%	53.4%
ParkScene	46.0%	47.5%	50.6%	45.3%	46.0%	47.7%	47.2%
Cactus	48.4%	49.4%	52.3%	47.2%	47.4%	49.1%	48.9%
BasketballDrive	44.9%	45.5%	48.1%	43.8%	43.6%	44.9%	45.1%
BQTerrace	52.7%	53.6%	56.1%	52.2%	52.3%	53.8%	53.5%
Averages	**49.5%**	**50.0%**	**51.9%**	**48.8%**	**48.4%**	**49.1%**	**49.6%**

Average costs							
	QP_{base}			QP_{base} + 2			**Totals**
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.5%	10.1%	4.5%	11.4%	6.7%	2.9%	8.8%
ParkScene	15.6%	8.5%	4.0%	10.1%	5.4%	2.5%	7.7%
Cactus	14.8%	8.0%	3.6%	9.4%	5.1%	2.3%	7.2%
BasketballDrive	16.3%	9.2%	4.4%	10.3%	5.9%	2.7%	8.1%
BQTerrace	17.2%	9.1%	3.9%	11.3%	5.9%	2.6%	8.3%
Averages	**16.3%**	**9.0%**	**4.1%**	**10.5%**	**5.8%**	**2.6%**	**8.0%**

Average reconstruction speed (frames per second)							
	720p		**540p**		**360p**		**Totals**
	QP	**QP+2**	**QP**	**QP+2**	**QP**	**QP+2**	
qp 22/24	12.5	13.0	15.0	15.4	14.7	15.0	14.3
qp 26/28	16.7	17.0	20.1	20.6	19.2	19.4	18.8
qp 30/32	19.8	19.9	23.9	24.3	22.2	22.2	22.1
qp 34/36	21.7	22.3	26.7	26.8	24.2	24.2	24.3
Averages	**17.7**	**18.1**	**21.4**	**21.8**	**20.1**	**20.2**	**19.9**

Table 3.2: Partial pruning lvl 1 (SBH off, RDOQ off)

Average bitrates

	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2419	1043	760	475	835	593	362	32.3%	62.7%
ParkScene	3552	1341	888	472	1053	678	354	26.3%	57.7%
Cactus	8283	2799	1933	1118	2242	1522	868	26.9%	58.9%
BasketballDrive	8058	2852	2031	1207	2300	1605	938	24.5%	60.0%
BQTerrace	13605	2585	1718	889	1949	1295	664	25.0%	48.6%
Averages	**7183**	**2124**	**1466**	**832**	**1676**	**1139**	**637**	**27.0%**	**57.6%**

Rate reductions

	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	52.1%	51.4%	50.9%	52.8%	51.0%	49.2%	51.2%
ParkScene	44.7%	46.4%	49.6%	44.5%	45.2%	47.0%	46.2%
Cactus	45.3%	46.6%	49.6%	44.9%	45.3%	47.2%	46.5%
BasketballDrive	39.8%	41.4%	44.7%	40.4%	40.9%	42.7%	41.6%
BQTerrace	51.9%	52.7%	55.3%	51.8%	51.8%	53.4%	52.8%
Averages	**46.8%**	**47.7%**	**50.0%**	**46.9%**	**46.8%**	**47.9%**	**47.7%**

Average costs

	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.5%	10.1%	4.5%	11.4%	6.7%	2.9%	8.8%
ParkScene	15.6%	8.5%	4.0%	10.1%	5.4%	2.5%	7.7%
Cactus	14.8%	8.0%	3.6%	9.4%	5.1%	2.3%	7.2%
BasketballDrive	16.3%	9.2%	4.4%	10.3%	5.9%	2.7%	8.1%
BQTerrace	17.2%	9.1%	3.9%	11.3%	5.9%	2.6%	8.3%
Averages	**16.3%**	**9.0%**	**4.1%**	**10.5%**	**5.8%**	**2.6%**	**8.0%**

Average reconstruction speed (frames per second)

	720p		**540p**		**360p**		**Totals**
	QP	**QP+2**	**QP**	**QP+2**	**QP**	**QP+2**	
qp 22/24	19.0	19.8	22.8	22.8	23.5	23.1	21.8
qp 26/28	26.3	27.3	32.3	32.1	32.0	31.5	30.3
qp 30/32	32.4	33.7	39.1	39.2	38.4	38.3	36.9
qp 34/36	37.6	38.6	45.0	45.7	43.5	43.6	42.3
Averages	**28.8**	**29.9**	**34.8**	**35.0**	**34.3**	**34.1**	**32.8**

Table 3.3: Partial pruning lvl 2 (SBH off, RDOQ off)

Average bitrates

	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2419	1043	760	475	835	593	362	26.9%	62.7%
ParkScene	3552	1341	888	472	1053	678	354	24.2%	57.7%
Cactus	8283	2799	1933	1118	2242	1522	868	23.3%	58.9%
BasketballDrive	8058	2852	2031	1207	2300	1605	938	18.6%	60.0%
BQTerrace	13605	2585	1718	889	1949	1295	664	23.8%	48.6%
Averages	**7183**	**2124**	**1466**	**832**	**1676**	**1139**	**637**	**23.4%**	**57.6%**

Rate reductions

	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	41.2%	42.6%	43.9%	43.9%	44.0%	43.8%	43.2%
ParkScene	40.1%	42.5%	45.9%	41.1%	42.4%	44.5%	42.7%
Cactus	38.2%	40.1%	43.3%	39.1%	40.0%	42.1%	40.4%
BasketballDrive	27.9%	31.1%	35.2%	30.5%	32.2%	35.0%	32.0%
BQTerrace	48.0%	49.1%	51.9%	49.5%	49.5%	51.3%	49.9%
Averages	**39.1%**	**41.1%**	**44.0%**	**40.8%**	**41.6%**	**43.3%**	**41.7%**

Average costs

	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.5%	10.1%	4.5%	11.4%	6.7%	2.9%	8.8%
ParkScene	15.6%	8.5%	4.0%	10.1%	5.4%	2.5%	7.7%
Cactus	14.8%	8.0%	3.6%	9.4%	5.1%	2.3%	7.2%
BasketballDrive	16.3%	9.2%	4.4%	10.3%	5.9%	2.7%	8.1%
BQTerrace	17.2%	9.1%	3.9%	11.3%	5.9%	2.6%	8.3%
Averages	**16.3%**	**9.0%**	**4.1%**	**10.5%**	**5.8%**	**2.6%**	**8.0%**

Average reconstruction speed (frames per second)

	720p		**540p**		**360p**		**Totals**
	QP	**QP+2**	**QP**	**QP+2**	**QP**	**QP+2**	
qp 22/24	30.4	31.5	35.9	36.7	37.4	37.8	35.0
qp 26/28	41.3	41.9	49.2	50.1	50.5	51.3	47.4
qp 30/32	48.8	52.3	60.3	62.2	61.5	61.6	57.8
qp 34/36	59.1	61.8	72.1	72.2	71.6	71.7	68.1
Averages	**44.9**	**46.9**	**54.4**	**55.3**	**55.2**	**55.6**	**52.1**

Table 3.4: Partial pruning lvl 3 (SBH off, RDOQ off)

Average bitrates

	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2419	1043	760	475	835	593	362	20.8%	62.7%
ParkScene	3552	1341	888	472	1053	678	354	21.4%	57.7%
Cactus	8283	2799	1933	1118	2242	1522	868	19.2%	58.9%
BasketballDrive	8058	2852	2031	1207	2300	1605	938	12.6%	60.0%
BQTerrace	13605	2585	1718	889	1949	1295	664	22.2%	48.6%
Averages	**7183**	**2124**	**1466**	**832**	**1676**	**1139**	**637**	**19.3%**	**57.6%**

Rate reductions

	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	29.4%	32.7%	35.6%	33.8%	35.4%	36.8%	34.0%
ParkScene	34.4%	37.4%	41.0%	36.7%	38.4%	40.7%	38.1%
Cactus	30.0%	32.8%	36.2%	32.3%	33.8%	36.1%	33.5%
BasketballDrive	16.5%	21.0%	25.4%	20.4%	23.2%	26.3%	22.1%
BQTerrace	43.2%	44.8%	48.0%	46.3%	46.5%	48.7%	46.3%
Averages	**30.7%**	**33.7%**	**37.2%**	**33.9%**	**35.5%**	**37.7%**	**34.8%**

Average costs

	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.5%	10.1%	4.5%	11.4%	6.7%	2.9%	8.8%
ParkScene	15.6%	8.5%	4.0%	10.1%	5.4%	2.5%	7.7%
Cactus	14.8%	8.0%	3.6%	9.4%	5.1%	2.3%	7.2%
BasketballDrive	16.3%	9.2%	4.4%	10.3%	5.9%	2.7%	8.1%
BQTerrace	17.2%	9.1%	3.9%	11.3%	5.9%	2.6%	8.3%
Averages	**16.3%**	**9.0%**	**4.1%**	**10.5%**	**5.8%**	**2.6%**	**8.0%**

Average reconstruction speed (frames per second)

	720p		**540p**		**360p**		**Totals**
	QP	**QP+2**	**QP**	**QP+2**	**QP**	**QP+2**	
qp 22/24	49.4	50.3	57.9	58.4	61.0	62.0	56.5
qp 26/28	63.4	65.0	75.6	76.4	78.9	79.2	73.1
qp 30/32	76.9	78.9	92.4	93.6	94.3	96.1	88.7
qp 34/36	90.9	94.3	108.9	111.7	111.0	112.3	104.9
Averages	**70.2**	**72.1**	**83.7**	**85.0**	**86.3**	**87.4**	**80.8**

Table 3.5: Full pruning (SBH on, RDOQ off)

Average bitrates									
	QP_{base}				QP_{base} + 2			**Totals**	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2409	1058	769	477	846	599	363	35.5%	63.0%
ParkScene	3530	1353	893	472	1061	682	354	28.1%	58.0%
Cactus	8275	2823	1944	1119	2259	1527	868	29.4%	59.2%
BasketballDrive	8078	2886	2048	1212	2323	1615	941	27.8%	60.2%
BQTerrace	13791	2626	1728	887	1971	1297	661	25.8%	48.6%
Averages	**7217**	**2149**	**1476**	**834**	**1692**	**1144**	**638**	**29.3%**	**57.8%**

Rate reductions							
	QP_{base}			QP_{base} + 2			**Totals**
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	58.2%	56.4%	55.0%	57.4%	55.1%	52.5%	55.8%
ParkScene	48.1%	49.1%	51.7%	47.4%	47.5%	48.8%	48.8%
Cactus	50.4%	51.1%	53.4%	49.1%	49.0%	50.4%	50.6%
BasketballDrive	46.8%	47.2%	49.4%	45.7%	45.3%	46.2%	46.8%
BQTerrace	54.0%	54.7%	56.8%	53.6%	53.2%	54.3%	54.4%
Averages	**51.5%**	**51.7%**	**53.2%**	**50.6%**	**50.0%**	**50.4%**	**51.3%**

Average costs							
	QP_{base}			QP_{base} + 2			**Totals**
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.0%	9.7%	4.4%	11.0%	6.3%	2.6%	8.5%
ParkScene	15.3%	8.2%	3.8%	9.7%	5.1%	2.4%	7.4%
Cactus	14.3%	7.6%	3.5%	9.0%	4.8%	2.1%	6.9%
BasketballDrive	15.8%	8.8%	4.1%	9.9%	5.5%	2.5%	7.8%
BQTerrace	16.9%	8.8%	3.7%	10.8%	5.7%	2.4%	8.1%
Averages	**15.9%**	**8.6%**	**3.9%**	**10.1%**	**5.5%**	**2.4%**	**7.7%**

Table 3.6: Partial pruning lvl 1 (SBH on, RDOQ off)

Average bitrates									
	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2409	1058	769	477	846	599	363	33.9%	63.0%
ParkScene	3530	1353	893	472	1061	682	354	27.5%	58.0%
Cactus	8275	2823	1944	1119	2259	1527	868	27.9%	59.2%
BasketballDrive	8078	2886	2048	1212	2323	1615	941	25.6%	60.2%
BQTerrace	13791	2626	1728	887	1971	1297	661	25.5%	48.6%
Averages	**7217**	**2149**	**1476**	**834**	**1692**	**1144**	**638**	**28.1%**	**57.8%**

Rate reductions							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	54.5%	53.7%	53.1%	54.9%	53.3%	51.4%	53.5%
ParkScene	46.7%	47.9%	50.6%	46.5%	46.8%	48.1%	47.8%
Cactus	47.3%	48.2%	50.6%	46.7%	46.9%	48.3%	48.0%
BasketballDrive	41.7%	43.0%	45.9%	42.2%	42.4%	44.0%	43.2%
BQTerrace	53.2%	53.7%	55.9%	53.1%	52.7%	54.0%	53.8%
Averages	**48.7%**	**49.3%**	**51.2%**	**48.7%**	**48.4%**	**49.2%**	**49.2%**

Average costs							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.0%	9.7%	4.4%	11.0%	6.3%	2.6%	8.5%
ParkScene	15.3%	8.2%	3.8%	9.7%	5.1%	2.4%	7.4%
Cactus	14.3%	7.6%	3.5%	9.0%	4.8%	2.1%	6.9%
BasketballDrive	15.8%	8.8%	4.1%	9.9%	5.5%	2.5%	7.8%
BQTerrace	16.9%	8.8%	3.7%	10.8%	5.7%	2.4%	8.1%
Averages	**15.9%**	**8.6%**	**3.9%**	**10.1%**	**5.5%**	**2.4%**	**7.7%**

Table 3.7: Partial pruning lvl 2 (SBH on, RDOQ off)

Average bitrates									
	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2409	1058	769	477	846	599	363	28.3%	63.0%
ParkScene	3530	1353	893	472	1061	682	354	25.2%	58.0%
Cactus	8275	2823	1944	1119	2259	1527	868	24.3%	59.2%
BasketballDrive	8078	2886	2048	1212	2323	1615	941	19.4%	60.2%
BQTerrace	13791	2626	1728	887	1971	1297	661	24.3%	48.6%
Averages	**7217**	**2149**	**1476**	**834**	**1692**	**1144**	**638**	**24.3%**	**57.8%**

Rate reductions							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	43.2%	44.6%	45.8%	45.7%	45.9%	45.7%	45.1%
ParkScene	42.0%	43.9%	46.9%	43.0%	43.8%	45.4%	44.2%
Cactus	39.9%	41.5%	44.2%	40.8%	41.3%	43.1%	41.8%
BasketballDrive	29.4%	32.4%	36.1%	31.9%	33.5%	35.9%	33.2%
BQTerrace	49.2%	50.0%	52.5%	50.8%	50.4%	51.8%	50.8%
Averages	**40.7%**	**42.5%**	**45.1%**	**42.4%**	**43.0%**	**44.4%**	**43.0%**

Average costs							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.0%	9.7%	4.4%	11.0%	6.3%	2.6%	8.5%
ParkScene	15.3%	8.2%	3.8%	9.7%	5.1%	2.4%	7.4%
Cactus	14.3%	7.6%	3.5%	9.0%	4.8%	2.1%	6.9%
BasketballDrive	15.8%	8.8%	4.1%	9.9%	5.5%	2.5%	7.8%
BQTerrace	16.9%	8.8%	3.7%	10.8%	5.7%	2.4%	8.1%
Averages	**15.9%**	**8.6%**	**3.9%**	**10.1%**	**5.5%**	**2.4%**	**7.7%**

Table 3.8: Partial pruning lvl 3 (SBH on, RDOQ off)

Average bitrates									
	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2409	1058	769	477	846	599	363	21.9%	63.0%
ParkScene	3530	1353	893	472	1061	682	354	22.3%	58.0%
Cactus	8275	2823	1944	1119	2259	1527	868	20.0%	59.2%
BasketballDrive	8078	2886	2048	1212	2323	1615	941	13.3%	60.2%
BQTerrace	13791	2626	1728	887	1971	1297	661	22.7%	48.6%
Averages	**7217**	**2149**	**1476**	**834**	**1692**	**1144**	**638**	**20.0%**	**57.8%**

Rate reductions							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	31.0%	34.3%	37.1%	35.2%	37.0%	38.5%	35.5%
ParkScene	36.0%	38.7%	41.8%	38.4%	39.7%	41.6%	39.4%
Cactus	31.4%	34.0%	36.9%	33.7%	34.9%	37.0%	34.7%
BasketballDrive	17.7%	21.9%	26.1%	21.6%	24.2%	27.1%	23.1%
BQTerrace	44.2%	45.6%	48.5%	47.5%	47.3%	49.2%	47.1%
Averages	**32.1%**	**34.9%**	**38.1%**	**35.3%**	**36.6%**	**38.7%**	**35.9%**

Average costs							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	17.0%	9.7%	4.4%	11.0%	6.3%	2.6%	8.5%
ParkScene	15.3%	8.2%	3.8%	9.7%	5.1%	2.4%	7.4%
Cactus	14.3%	7.6%	3.5%	9.0%	4.8%	2.1%	6.9%
BasketballDrive	15.8%	8.8%	4.1%	9.9%	5.5%	2.5%	7.8%
BQTerrace	16.9%	8.8%	3.7%	10.8%	5.7%	2.4%	8.1%
Averages	**15.9%**	**8.6%**	**3.9%**	**10.1%**	**5.5%**	**2.4%**	**7.7%**

Table 3.9: Deflation (SBH off, RDOQ off)

Average bitrates									
	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2419	1407	927	536	1045	690	397	25.3%	67.5%
ParkScene	3552	1761	1056	518	1288	771	380	21.4%	62.4%
Cactus	8283	3557	2240	1212	2662	1694	920	22.2%	62.8%
BasketballDrive	8058	3647	2370	1315	2738	1791	997	20.8%	63.8%
BQTerrace	13605	3395	2000	955	2342	1439	699	20.7%	52.7%
Averages	**7183**	**2753**	**1719**	**907**	**2015**	**1277**	**679**	**22.1%**	**61.8%**

Rate reductions							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	34.9%	38.5%	40.1%	36.7%	39.2%	39.6%	38.2%
ParkScene	31.5%	35.8%	40.8%	32.5%	35.8%	39.6%	36.0%
Cactus	32.6%	37.1%	41.8%	33.5%	36.9%	40.5%	37.1%
BasketballDrive	30.1%	34.1%	38.3%	30.9%	33.8%	36.8%	34.0%
BQTerrace	36.7%	41.0%	46.0%	38.7%	41.6%	45.4%	41.6%
Averages	**33.2%**	**37.3%**	**41.4%**	**34.5%**	**37.5%**	**40.4%**	**37.4%**

Average costs							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ParkScene	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Cactus	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
BasketballDrive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
BQTerrace	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Averages	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**

Table 3.10: Deflation (SBH off, RDOQ on)

Average bitrates									
	QP_{base}				QP_{base} + 2			Totals	
	1080p	**720p**	**540p**	**360p**	**720p**	**540p**	**360p**	**GT gain**	**Max gain**
Kimono	2412	1423	940	545	1055	699	402	22.3%	67.8%
ParkScene	3630	1804	1082	533	1316	789	390	19.1%	62.4%
Cactus	8454	3571	2248	1224	2660	1696	927	19.0%	62.8%
BasketballDrive	8025	3603	2346	1314	2693	1768	993	16.5%	64.0%
BQTerrace	15280	3524	2038	969	2411	1463	709	18.1%	51.6%
Averages	**7560**	**2785**	**1731**	**917**	**2027**	**1283**	**684**	**19.0%**	**61.7%**

Rate reductions							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	32.3%	34.0%	33.8%	32.5%	33.2%	32.4%	33.0%
ParkScene	29.3%	31.7%	34.5%	29.3%	30.9%	32.7%	31.4%
Cactus	29.5%	31.9%	34.2%	29.2%	30.7%	32.3%	31.3%
BasketballDrive	25.0%	27.4%	29.4%	24.6%	25.9%	27.2%	26.6%
BQTerrace	33.8%	36.1%	38.7%	34.9%	36.1%	37.9%	36.3%
Averages	**30.0%**	**32.2%**	**34.1%**	**30.1%**	**31.4%**	**32.5%**	**31.7%**

Average costs							
	QP_{base}			QP_{base} + 2			Totals
	720p	**540p**	**360p**	**720p**	**540p**	**360p**	
Kimono	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ParkScene	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Cactus	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
BasketballDrive	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
BQTerrace	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Averages	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.0%**

Chapter 4
Conclusions

Guided transcoding is an excellent method for saving bit rates on the transmitting side in any ABR application. We trade a small amount of computational complexity for gains around 20–30% which is actually quite astonishing. We ran our simulations using H.265, but there nothing about either pruning or deflation that says it cannot be used in H.264, or any other standard like VP8 or VP9. As long as there is a division between mode information and transform coefficients then GT works.

Pruning has higher rate reductions than deflation, close to 30% although but at the cost of some degradation for transcoding that needs to be handled by increasing the bit rate.

We investigated partial pruning as method for bringing the complexity down further. In our simulations, partial pruning level 2 seems to have a very good balance between bit rate gains and decrease in computational complexity. The scheme could be deployed dynamically, so that we use partial pruning on the higher resolutions where complexity is more critical, but prune fully the smaller resolutions were regeneration is already less complex. Finding a sweet spot where gains are high enough, but computational complexity is still decreased is up to each provider to decide based on their available hardware and how much they need strict real-time regeneration. If strict real-time regeneration is not necessary then full pruning is still probably the best option.

For the time measurements, the double downscaling for 360p (see section 3.1.2) takes extra time, and most likely explains why regenerating a 360p video takes longer than 540p, although it contains fewer coefficients are should actually be faster. In a real scenario we would use a downscaler that perform this is a single step.

Deflation has a gains just above 20%, but with no quality loss. This is a very attractive quality because there is no need to tinker with the bit rate to find out how much to increase it in order to keep the quality at the same level as for simulcasting. You just encode the sequences and deflate, and when the video is inflated it will be same as it was in the simulcast case. As for the complexity of the deflation scenario, we suspect that regeneration will be slower than for pruning because more steps are involved. At the very least they should be of the same complexity.

We ran our simulations on seven principal resolutions and qualities. 1080p encoded at QP, and 720p, 540p and 360p encoded at QP and QP+2. A realisitic scenario could offer a lot more versions than that and this would make guided transcoding even more effective.

Bibliography

[1] Iain Richardson. Introduction to Video Coding.

[2] Mathias Wien. *High Efficiency Video Coding*. Springer, 2015.

[3] Markus Flierl. Image and Video Processing, KTH EQ2330. Lecture slides, 2015.

[4] Iain Richardson. HEVC: An Introduction to High Efficiency Video Coding.

[5] Glenn Van Wallendael, Jan De Cock, and Rik Van de Walle. Fast Transcoding For Video Delivery by Means of a Control Stream. In *19th IEEE International Conference on Image Processing (ICIP 2012)*. Ghent University – IBBT, ELIS Department - Multimedia Lab, 2012.

[6] Iain Richardson. HEVC Walkthrough, Jun 2013.

[7] Gary J. Sullivan and Thomas Wiegand. Rate-distortion optimization for video compression. *IEEE Signal Processing Magazine*, Nov 1998.

[8] Timothy Sauer. *Numerical Analysis*. Pearson, 2nd edition, 2012.

[9] Johan Bartelmess. Video Compression – Evaluation of Picture Coding Structures in HEVC for Improved Compression Efficiency, 2016.

[10] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Hierarchical b pictures. In *JVT-P014*, Poznan, Poland, Jul 2005.

[11] Thomas Rusert, Kenneth Andersson, Ruoyang Yu, and Harald Nordgren. Guided Just-in-Time Transcoding for Cloud-Based Video Platforms. Submitted to ICIP 2016, Jan 2016.

[12] Anthony Vetro, Charilaos Christopoulos, and Huifang Sun. Video Transcoding Architechtures and Techniques: An Overview. *IEEE Signal Processing Magazine*, March 2003.

[13] Gisle Bjøntegaard. Calculation of average psnr differences between rd curves. In *ITU-T SG 16 WP 3, doc. VCEG-M33*, Austin, TX, USA, Apr 2001.