Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video

Michael Roitzsch <mr188034@inf.tu-dresden.de>
Operating Systems Group
Computer Science Department
Technische Universität Dresden

13th June 2005

tutored by Martin Pohlack <mp26@os.inf.tu-dresden.de>,
department headed by Prof. Dr. rer. nat. Hermann Härtig
<haertig@os.inf.tu-dresden.de>

In this article, I will discuss a method to predict per-frame decoding times of modern decoder algorithms. By examining especially the MPEG-1, MPEG-2 and MPEG-4 part 2 algorithms, I will develop a generic model for these decoders, which also applies to a wide range of other decoders. From this model, I will derive a method to predict decoding times with a reasonable quality while keeping overhead low. The effectiveness of this method will be shown by an example implementation and comparing the resulting predictions with the actual decoding times. A brief outlook on future application and extension of these results concludes the paper.

Contents

1 Introduction

With increasingly demanding media applications emerging, like viewing or even creating and editing high definition content (HDTV), the immanent limitations of today's media software and underlying operating systems move into focus: Lacking realtime properties, these applications can only be dealt with by using application-specific dedicated hardware, overprovisioning of resources, or arbitrary and thus ungraceful degradation of playback quality. To allow future systems to overcome these difficulties, realtime and QoS considerations should be applied [1, 2] to allow close to average-case resource allocation and graceful degradation of quality.

For example, a video player whose decoding component cannot keep up with the display speed and therefore has to skip the decoding of frames might want to prefer skipping frames with a high decoding time. Another possible application would be in a video editing system, where the user can stack multiple layers of video and apply effects to them. This system needs to decide, which parts of the final edit can be calculated on the fly during playback and which parts need to be prerendered. The prerendering has to include all critical parts to ensure sufficient playback quality, but at the same time, prerendering should be reduced to the minimum to keep the application responsive.

As most of these concepts require previous knowledge of the resource usage of the task ahead and because the primary task in the media applications considered here is video decoding, with CPU time being the key resource, this work discusses a way to pre-determine per-frame decoding times for recent video decoder technologies. Obviously, this method must provide the decoding time with lower resource usage than the actual decoding. It should also be possible to calculate the decoding times on the fly for each frame, because users cannot be expected to provide precalculated trace data or wait for its collection. So we can deduce the following requirements for the prediction method:

- Provide per-frame decoding times considerably faster than the actual decoding takes. As little bitstream parsing and unpacking as possible should be done.
- Use data for this prediction only when it is available to the decoder at decoding time of the frame, so it becomes possible to calculate the prediction on the fly, directly preceding the actual decoding of each frame.
- Do not rely on extensive code review of decoder implementations, because it quickly becomes outdated as new decoders are released or existing decoder code is being optimized. Use statistical methods instead so our method can quickly adapt to a variety of decoders.
- If this adaptation requires sample material, it should require material only roughly related to the target application. Specifically, it should not be required to use the exact target video for adaptation.

1.1 Terminology

At first, I want to clarify the following terms, which are going to be used throughout the paper:

Decoder means the piece of software converting compressed video data into raw image data, usually in a device-dependent colorspace like YUV or, more rarely, RGB. Often, the term decoder denotes a decoding algorithm as well as its actual implementation. This article mostly deals with implementations, because the notion of decoding time is bound to actual software being executed.

Frame decoding time or just decoding time is the wallclock time it takes the decoder to parse and process, without being interrupted, enough of the compressed video data to have one video image completely

assembled. Any further processing steps on the image like enhancing its quality through the use of video filters or the presentation of the image on a graphics device do not account as decoding time. Note that the decoder may not even emit the decoded image right after its completion due to frame reordering.

Predicted frame decoding time or just predicted time or prediction is the estimate for the decoding time. The principles of deriving this value from the compressed video data is the subject of this work.

Decoder cycle time is the wallclock time that passes from control being given to the decoder until the decoder returns this control. The decoder executed one of its internal processing cycles inbetween. Mostly, one cycle directly corresponds to one video image being completely decoded, so that decoder cycle time and frame decoding time would tally. However, this might not be true for decoders with slice-based processing or frame reordering.

Predicted decoder cycle time is the estimate for the decoder cycle time. I will show ways to derive it from the predicted frame decoding time.

Video frame or just frame is used as a synonym for video image in this article.

The problem of frame reordering depends on the implementation of the decoder algorithm and will be discussed in detail in Subsection 5.4. For the following general considerations, I want to keep assumptions on actual implementations low, so I will only examine the per-frame times.

1.2 Decoder Algorithms to consider

The selection of the MPEG-1/2 [6, 7] and MPEG-4 part 2 [8] decoder algorithms as the key algorithms to analyze has been made for the following reasons:

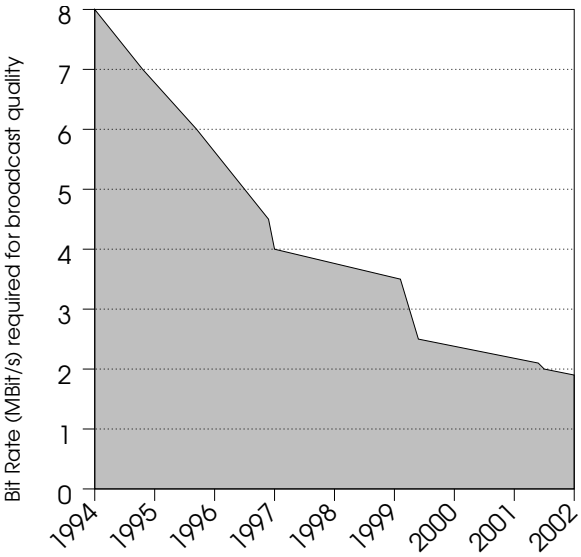

Figure 1: MPEG-2 encoder efficiency shown by bitrate need over the years [10].

- The decoder algorithms should have a wide public acceptance so the results of my work are actually applicable to real-world problems.
- The implementation should be proven and well optimized, because early stages of decoder software can have a different decoding time behavior.
- An open-source implementation should be available to allow easy insights into and reuse of existing code.
- Encoders for the algorithms should be efficient to get representative compressed material. Early stages of encoders often use only a subset of the algorithm's features, so they require higher data rates for the same perceived image quality than more advanced encoders (demonstrated exemplary on MPEG-2 in Figure 1). This behavior is not desirable, if you want future-proof results.

MPEG-2 has been chosen because it has all the preceding properties: Its presence on DVD and DVB makes it widely used. Thoroughly tested

open-source decoder software is available from the libmpeg2 project [20]. Sophisticated encoders can be found in both the commercial sector and the open-source community.

MPEG-1 is similar to MPEG-2, so it came naturally to regard MPEG-1/2 as one. Results for the H.261 and H.262[1] standards can be derived easily, because of their similarity to MPEG-1 and MPEG-2 respectively [12].

MPEG-4 part 2 is one video decoding algorithm specified in the MPEG-4 standard family. It is not as commonly used as MPEG-2, but with MPEG-4 part 10 [9] on the horizon, it seemed necessary to include a more recent algorithm than MPEG-2. MPEG-4 part 10, also known as AVC (advanced video coding) or H.264, appears to become the next generation allround format. It scales from cell phones up to high definition video, is more efficient than MPEG-2 and it uses integer transforms to allow fast implementations. Both the Blu-ray and HD-DVD groups have adopted H.264, and it is the major format in Apple's recently released QuickTime 7 [13]. However, good open-source implementations of H.264 decoders are not yet available, sample content is hard to come by and sufficiently efficient encoders are not commonly available even in the commercial market. For those reasons, H.264 is not one of the key algorithms I will examine, but I will comment on how my ideas apply to H.264 in Subsection 3.2.3. Although H.264 is much more sophisticated than MPEG-4 part 2, the latter is still closer to it than MPEG-2, so it seems wise to go at least part of the way by having a deeper look into MPEG-4 part 2. The H.263 standard is close to being a true subset of MPEG-4 part 2, so most of the findings for MPEG-4 part 2 will also apply to H.263.

Other candidate algorithms were not included for various reasons:

DV is widely used in digital camcorders, but the algorithm is not of particular interest, because every frame is self-contained, no inter-frame-redundancy is exploited [14]. This makes the algorithm well-suited for editing with frame granularity, but the compression is not exceptionally efficient.

The Windows Media algorithms, especially WMV9 are quite common today and good decoder as well as encoder implementations are available from Microsoft. Although Microsoft has standardized WMV9 under the name VC-1, no open-source implementations are available. Basically the same applies to the Sorenson SVQ3 algorithm, which is used in most downloadable QuickTime movie trailers. Efforts of reverse engineering both WMV9 and SVQ3 have shown evidence that both decoders are based on ideas from H.264 [15].

The selected MPEG-1/2 and MPEG-4 part 2 decoders exploit both intra- and inter-frame redundancy and visual perception deficiencies of the human eye to compress the video data. The common way to do that is to blockwise transform the image on encoding from the spatial domain into the frequency domain, because the perception model the encoders use can more easily exploit the redundancy there. This process has to be reversed during decoding, which requires numerous iterative calculations on the compressed data to reconstruct the image. This complexity is the main reason why the decoding is CPU intensive and scheduling decisions can benefit from knowing a prediction of the decoding times in advance.

1.3 Outline

My vision (see Figure 2) is to develop a predictor, which is fed with only the compressed video data and produces estimates for the decoder cycle time. The decoder itself should not be modified in any way. The predictor should be as independent of the actual decoder implementation as possible to allow future upgrades to

[1] The MPEG-1/2/4 standards from the Moving Picture Experts Group and the H.261/262/263/264 standards from the International Telecommunication Union (ITU) overlap.

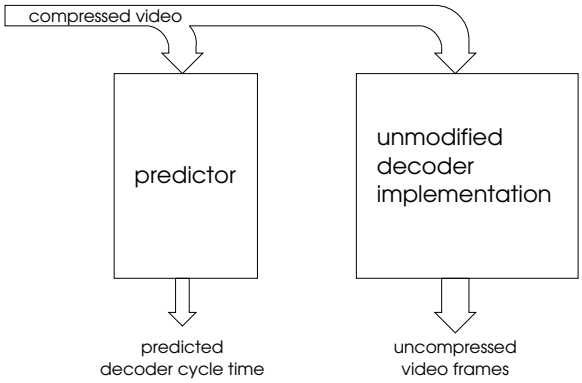

Figure 2: The target architecture of this work.

newer decoder versions without any major modifications to the predictor.

The basic idea I am going to explore is to break down the entire decoding of a frame into various sub-tasks. This will be achieved by creating a generic decoder model, into which all examined decoders and hopefully a wide range of future decoders fit (see Subsection 3.1). I will then discuss the influence of each sub-task on the decoding time in Subsection 3.3. I will point out characteristic properties of the stream that correlate positively with the decoding time. I will then extract values from the compressed video stream that serve as metrics to quantify the discussed properties. The overall per-frame decoding time will then be estimated by a linear combination of the extracted metrics. The mathematics used to calculate the coefficients of the linear combination will be explained in Section 4, followed by comments on the implementation of the given principles in Section 5. An evaluation of the results and starting points for future work conclude the article.

2 Related Work

2.1 Outside Results being used

My work is founded on previous results in the research areas of media player design, decoder algorithms, and mathematics:

I have implemented my ideas using VERNER [19], a video encoder and player developed by Carsten Rietzschel. VERNER is easily extensible by plugins and uses a multi-tier client-server architecture to cleanly model the video playback pipeline of demultiplexing, decoding, synchronizing and outputting by assigning these steps to separate tasks. This design sufficiently decouples the timing behavior of the decoder from the rest of the player, thus allowing me to use VERNER as an experimenting ground for my measurements and predictions.

VERNER itself is based on DROPS [17], the Dresden realtime operating system, which is a multiserver microkernel-based system providing the basic OS services like a filesystem or a graphical output mechanism. The Fiasco [18] L4 microkernel in use is optimized for low communication latency, which allows the VERNER client-server design to function efficiently. The real-time capability of the system is not exploited in my work, but could become vital in using the results to improve video playback behavior in resource constrained situations.

To measure the various times and display them graphically I used the RT_Mon framework, a software sensor system to gather, accumulate, distribute, and visualize arbitrary data. RT_Mon has been developed by Martin Pohlack.

Because I want to extract characterizing values from the video bitstreams, I required bitstream parsers for the selected decoder algorithms MPEG-1/2 and MPEG-4 part 2. I relied on the open-source implementations libmpeg2 [20] and XviD [21], which are already used by VERNER for the actual decoding. I stripped them down to the bitstream parsing level and also reused code from their example applications to implement the bridging between the specific decoder library API and the abstract prediction API I will provide in Section 5.

Finally, I am also building my work on results from mathematics, especially numerics. I used well known numerical techniques such as the QR decomposition as well as established algorithms

like the Householder transformation. These will be explained in greater detail in Section 4.

2.2 Comparison to other Work

Decoding time prediction has been a research subject before, so previous results exist. Altenbernd, Burchard, and Stappert present an analysis of MPEG-2 execution times in [3]. The common idea behind their work and mine is to gain metrics from the video stream that correlate well with the decoding time. However, there are considerable differences in both approaches: Altenbernd, Burchard, and Stappert divide the decoder in two phases, using data extracted in the first phase to predict the decoding time of the second phase. I want to avoid such heavy modifications to existing decoder code and rather provide a solution based on preprocessing. I also model the decoder as a sequence of different phases, but my division will be more fine grained (see Figure 3). They also examine worst-case execution times, using source code analysis, to completely avoid any underestimation. I do not want to rely on the specific source code, because the efforts of the analysis have to be repeated when the code changes due to optimizations. This simplification allows me to generalize my method to target more decoders than just MPEG-2. Because I cannot safely avoid underestimation with my approach, I have to settle for a best-effort estimation, but the results are still important because the quality assuring scheduling algorithm QAS [5] developed here at TU Dresden works on the basis of a probability distribution for execution times. It remains an area of future research to derive worst-case execution times with my method.

Another similar analysis has been conducted by Andy Bavier, Brady Montz, and Larry L. Peterson in [4], but is also limited to MPEG, focuses more on decoding times at the granularity level of network packets, and does not target transferability of the results between different videos. Yet they also follow the path of predicting decoding times by extracting metrics from the stream.

3 Decoder Analysis

3.1 Decoder Model

By looking into the inner workings and functional parts of the decoder algorithms in consideration, I established a decoder model by abstracting from the given algorithms. The model is generic enough to be applicable to a wide range of other algorithms. The simple basic structure of the model can be seen in Figure 3. It is a chain of function blocks that are executed iteratively in loops. Because I am concentrating on execution times, the edges of the graph represent control flow rather than any data flow. Control starts at the bitstream parsing block and ends implicitly when the compressed data stream ends.

Every function block but the first can have multiple alternative and completely separated execution paths in the same decoder algorithm. Those execution paths would be chosen among by context data extracted from the compressed video stream, like a frame type. As a notable special case, the function blocks can have a no-operations execution path, essentially doing nothing. I will now go through the blocks and explain each one and its input and output more closely. When it serves the understanding, I will also mention the encoder. The decoding standards, however, only specify the bitstream and the decoder, leaving the details of efficient encoder implementations to the market.

3.1.1 Bitstream Parsing

The bitstream parsing receives the compressed video stream for every frame and extracts meta- and context-information from the stream, which is required to control the following decoding steps. Because switches between the multiple execution paths of the function blocks are done based on the data extracted by this step, this

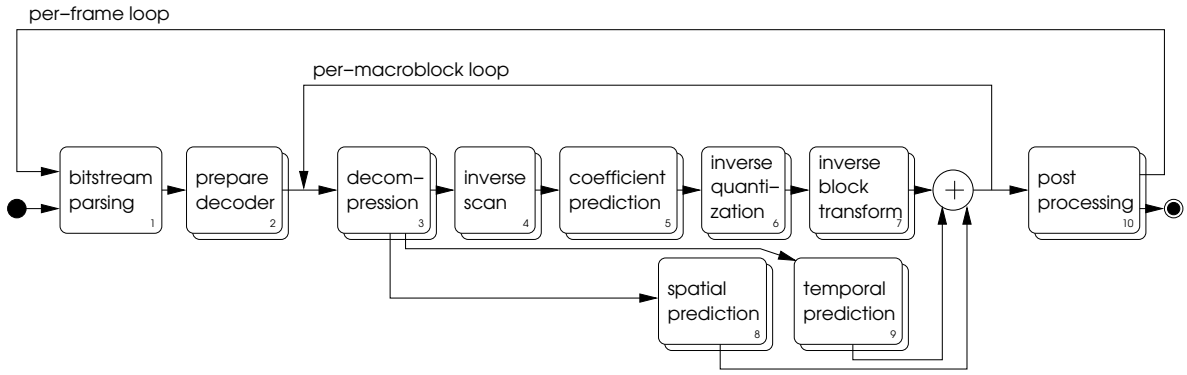

Figure 3: Chained block model for decoders

function block itself cannot have multiple, completely separate execution paths as there would be no way to switch between them. Of course the inner working of this block can still have different control flows, but those can hardly be regarded as separated.

Bitstream parsing is nondestructive, so the data this step outputs is unchanged from its input. This makes this function block reusable for our approach of extracting certain metrics from the stream that shall give an indication of the decoding time. Because the context data extracted by the parser can change the control flow through the remaining function blocks tremendously, this decoder step is most likely required for any usable decoding time estimation.

The context data the parser creates includes information such as:

- Pixel resolution of the target video image,
- Other size information the decoder needs to allocate buffers for intermediate results in advance,
- Lookup tables used by following function blocks,
- Frame type of the upcoming compressed video data.
  Most modern decoders use different frame types to easily switch between different execution paths of following function blocks like the temporal prediction. The algorithms want to exploit temporal redundancy, so they store only changes between adjacent frames. However, not all frames can be treated this way, because to start in an arbitrary position inside a compressed video file, the decoder needs a self-contained frame without inter-frame dependencies. This makes it necessary to dismiss the temporal prediction for those frames, which is done by assigning a frame type to them.

3.1.2 Decoder Preparation

The preparation stage allocates any memory buffers needed for decoding. When decoder algorithms exploit temporal redundancy, they often work with previous images. Those may need to be copied or modified without disturbing existing images, which must not be changed, because they might not have been displayed yet. This may not be needed for all frames, so the execution path of the decoder preparation can vary based on frame type.

Because this function block only paves the way for the following blocks, we can safely drop it in extracting our metrics, unless we need to execute one of the per-macroblock functions that rely on it.

3.1.3 Decompression

This function block is the first that is executed inside a per-macroblock loop. A macroblock is an area of the target image whose compressed data is stored consecutively in the data stream and that is decoded in one iteration of the loop. For today's decoder algorithms, macroblocks are square areas of 8×8 or 16×16 pixels.

The data needed to further decode the macroblock is stored using a lossless compression technology like Huffman [22]. This compression is reversed in this step and the intermediate data is kept in a buffer for later steps to access. Because the compressed data for each macroblock usually varies in length, depending on the size and entropy of the uncompressed data, the decompression step also involves additional bitstream parsing when walking over the compressed stream from macroblock to macroblock. I decided to subsume decompression and moving forward in the compressed stream as one function block, because their implementations intermingle.

The decompression function can use tables from the bitstream parsing stage, if a Huffman table or something equivalent is embedded into the compressed video. There may even be decoder algorithms switching between completely different decompression functions, because they may be specifically adapted for certain types of video material. The compression filters in the PNG image format point in this direction [16]. They are applied to the byte stream right before the actual compression, and their effect varies with the image content.

In addition to the macroblock data itself, each macroblock may come with metadata like a macroblock type or other information for following decoder steps. As a special case, a macroblock might consist entirely of such context data and no compressed image data at all.

3.1.4 Inverse Scan

Because the resulting video image is two dimensional, the transforms used later in the compression are two dimensional as well. However, the decompressed macroblock we received from the previous stage is a one dimensional list of bytes. Those need to be rearranged in a 2D matrix. Because this would partly countereffect the preceding entropy compression step, this reordering is not done in a line-by-line fashion, but in a different, often diagonal pattern. Different patterns may be chosen based on earlier information about either the entire frame or the macroblock. The copying into the 2D matrix can be implemented together with further decompression like run length encoding.

3.1.5 Coefficient Prediction

Decoders can analyze the matrix to try to reconstruct image features that have been compressed away. One approach to do that is to define prediction functions that extrapolate additional data from existing data. The encoder will use this function to determine which information the decoder will be able to reconstruct when it is left out. Either by applying a simple accuracy tolerance or by storing prediction hints or by storing the remaining error for the decoder to apply to the prediction, the encoder decides which data can be dropped without degrading image quality too much. The encoder might even have different prediction functions or prediction patterns at its discretion.

Because the following transform function commonly uses each element of the macroblock matrix more than once, the coefficient prediction step is executed entirely before the next steps begin.

3.1.6 Inverse Quantization

During encoding, the matrix of each macroblock has been multiplied with a quantization matrix,

which, more than all other steps, makes the compression lossy. The quantization is reversed before decoding proceeds by multiplying with an inverse quantization matrix, in which every matrix element is the reciprocal of the corresponding element of the quantization matrix. The inverse quantization matrix can be a default one or it can be embedded into the bitstream. Different matrices are used for different frame types or for different types of macro blocks.

Skipping the coefficient prediction, the inverse scan and inverse quantization steps may be implemented together as a loop with just table lookups and a multiplication. However, because the different versions of inverse scan and inverse quantization can be selected independently of each other, they must be represented as two function blocks.

3.1.7 Inverse Block Transform

The macroblock matrix we dealt with in the previous steps was not necessarily a matrix in the spatial domain. So the rows and columns of the matrix are not directly mapped to the image's x and y coordinates. The algorithms usually choose a different domain, in which operations like the quantization and coefficient prediction are fairly simple matrix operations and fit the visual perception model used by the algorithm in their effect on the final image. A common domain is the frequency domain, which I will describe for the MPEG-1/2 algorithm in Subsection 3.2.1. It has the nice property that quantization in the frequency domain will gradually smooth out details in the spatial domain.

This decoder step now transforms the macroblock matrix into the spatial domain, which involves complicated mathematics, sometimes even with floating point calculations. The resulting spatial matrix corresponds to a portion of the final image and has the same dimensions as the macroblock matrix.

3.1.8 Spatial Prediction

The spatial and temporal prediction steps described now use previously decoded data of either the same frame (spatial prediction) or a different frame (temporal prediction) to reconstruct or merely guess the part of the image being covered by the macroblock currently decoded. This step can potentially be executed at the same time as the steps 4 (inverse scan) to 7 (inverse block transform), but I will not pursue this parallelity, because the commonly available decoder implementations are single threaded. The outcome of the inverse block transform is then added to or otherwise merged with this reconstruction to reduce the residual error between the prediction and the image to be encoded.

3.1.9 Temporal Prediction

Today's decoder algorithms benefit tremendously from temporal redundancy in the video. The most basic idea is to not always store entire video images, but to store the differences to the previous image. This has been extended over decoder generations to allow other reference images than just the previous one and to compensate for motion by storing translation vectors with the macroblocks. It even includes rotating and warping parts of the reference image, compressing the motion vectors by applying prediction to them or weighted interpolation between areas of multiple reference images in the most sophisticated algorithms.

Because the temporal prediction introduces inter-frame dependencies, it cannot be used on every frame to its full extent, because the algorithms need to consider that playback might start at an arbitrary position inside the stream, for which one would require a frame without any dependencies on other frames, or that a player might want to skip the decoding of upcoming frames when short on CPU time, which requires that not too many other frames depend

on the ones about to be dropped. The distinction which features of the temporal prediction are allowed is usually made with frame types again: For specific frame types, using future frames as reference images is allowed. Other frame types may only use images from the past up to a certain depth. The most restricted frame type is required to be self-contained, no reference frames can be used.

The merging of the results of prediction and inverse transform ends the per-macroblock loop. Execution will continue with the decompression of the next macroblock. It should be noted that most algorithms bundle consecutively stored macroblocks with common properties in a so called "slice". This bundling is not yet relevant for considerations about execution time, but it will become more interesting when algorithms start to associate a semantic with slices, for example marking slices containing more important parts of the image like objects in the foreground. Determining a per-slice decoding time will be interesting, because it would allow slice based scheduling. A player would be able to skip decoding of unimportant parts of the image when not enough CPU time is left. Such parts of the image would then stay unchanged compared to the previously displayed image. More details on this can be found in the discussion on H.264 in Subsection 3.2.3.

3.1.10 Post Processing

Post processing applies a set of filters to the resulting image so that compression artifacts are reduced and the perceived image quality is enhanced. However, with today's algorithms, post processing is often optional, so in realtime applications, a video decoder can skip the post processing stage if the scheduled CPU time is exceeded. Therefore, I will not analyze the execution time of the post processing stage and examine the mandatory parts of the decoding. With future algorithms, post processing will become an integral part of the decoding process. It may use hints from earlier decoding stages or improve the image so significantly that it becomes unusable without post processing. For those algorithms, researching the execution time of post processing will become necessary.

3.2 Current Decoders

I will now give a brief overview on how the algorithms this article examines fit into the developed decoder model. To widen the perspective and to further demonstrate the broadness of the model, I will also show, how the model accommodates two additional algorithms that have been touched in Subsection 1.2, but for the reasons explained there, those algorithms' execution times will not be examined.

3.2.1 MPEG-1/2

The MPEG-1 [6] and MPEG-2 [7] algorithms distinguish between three frame types: I- (intraframes), P- (predicted or interframes) and B-frames (bidirectional predicted frames). The frame type affects the selected execution path of the inverse quantization and temporal prediction function blocks. Here are the function blocks explained, as depicted in Figure 3:

1. The MPEG video elementary bitstream is packetized for easy delivery. The packets and headers are byte-aligned to allow resynchronization in case of bitstream errors, which may be caused by network outages. The various fields in the headers, however, are not byte-aligned, which makes parsing more difficult, but reduces the overall size by increasing the information density of the stream.

2. The decoder preparation is almost empty apart from acquiring the memory for the target image. Because the memory buffers needed for decoding do not change in size, there is no need for allocation or deallocation unless the video resolution changes.

3. The decompression stage is a combined Huffman/RLE algorithm, which determines patterns of nonzero coefficients and zero runs in the macroblock values. The nonzero values are stored together with an indicator for the pattern of zeros. Both are compressed according to a Huffman table.

4. The inverse scan can use two different zigzag patterns to reorder the macroblock values in the 2D matrix. The pattern is selected by a flag in the bitstream.

5. Coefficient prediction is only done by storing the difference of consecutive DC coefficients (see Step 7) instead of their actual values.

6. The inverse quantization multiplies the macroblock matrix elementwise with an inverse quantization matrix. Different matrices are available for the I-type and P/B-type frames and the matrices can also be embedded into the bitstream.

7. The heart of MPEG-1 and MPEG-2 is the IDCT, the inverse discrete cosine transform. This block-transforms the macroblock from the frequency domain into the spatial domain. The frequency domain matrix stores coefficients for a decomposition of the spatial domain into 2D cosine contributions. The 64 2D cosine functions depicted in Figure 4 form a base of the spatial domain. The left upper value is the constant offset, whose base function has a zero frequency in both directions, called "DC coefficient". The remaining values with a frequency other than zero in at least one direction are called "AC coefficients".[2] The transform takes those coefficients to scale the associated 2D cosine function. All these contributions will then be superimposed to form the final spatial domain matrix. We can see that the lower frequencies are located at the upper left corner and that they increase toward the lower right corner. While the discrete cosine transform and its inverse are lossless except for rounding errors, evaluations have shown that most images have dominant low frequency contributions and only little influence of high frequencies, so especially after quantization, a significant number of values in the frequency domain matrix will become zero during encoding. As for execution times, the IDCT is computationally expensive, but fast integer implementations with SIMD properties exist. By vectorizing loops, these implementations are able to process several values in parallel, so they make use of instruction sets like the Intel MMX instructions to speed up the calculation.

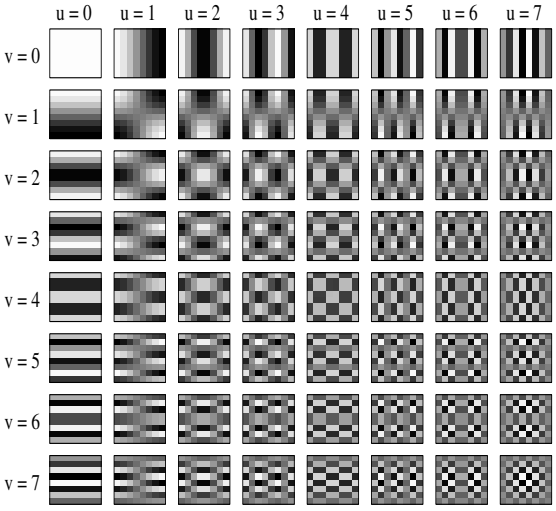

Figure 4: Base functions of the discrete cosine transform [23, 7].

8. There is no spatial prediction in MPEG-1/2.

9. The temporal prediction is only active for P- and B-type images. For I-type frames, it is empty. A P-frame stores the differences to the previous I- or P-frame, a B-frame stores the differences to the previ-

[2] I believe this to stem from the electronics world, where DC and AC stand for direct current and alternating current.

ous and the next I- or P-frame, whereas every macroblock can reference the previous, next or an average of both frames. B-frames are never used as reference frames, so the decoder has to keep only two reference frames at any time: the backward and the forward reference. Additionally, every macroblock can encompass a motion vector, which moves a certain tile of the reference image before applying the stored differences. This vector can operate with full or half-pixel granularity.

10. There is no post processing in MPEG-1/2.

The main difference between MPEG-1 and MPEG-2 is the support for interlaced material in MPEG-2 and the scalability extensions to allow decoding with a reduced resolution base layer video, complemented by enhancement layers. Other than that, only minor changes like extended resolution range and the addition of new chroma subsampling formats occurred.

3.2.2 MPEG-4 part 2

The MPEG-4 part 2 algorithm adds a new frame type to the common I-, P- and B-frames. The S-frame is a sprite frame, which uses a technology called global motion compensation (GMC) to apply an overall motion to an entire reference frame. The frame types and macroblock types are once again the deciding factors for the execution path used to decode a frame.

1. The bitstream is densely packed, so the syntax elements in it are not necessarily byte aligned, which makes parsing difficult. The stream can contain a complexity estimation header, which stores information we could use for our metrics extraction, like DCT block counts. Unfortunately this header is optional and the common encoder implementations do not use it, so existing videos would have to be reencoded with an extended encoder to benefit from this header, therefore I will not pursue this.
2. The P-, B-, and S-frames use a special treatment of the edges of the reference frames for motion vectors beyond the frame boundaries. This preparation step is computed before the reference frames are used.
3. The bitstream uses several tables for variable length coding of the coefficients and the motion vectors.
4. Different scan tables are available, which also includes scans suitable for interlaced material.
5. The coefficient prediction treats both the DC and the AC coefficients by copying rows from macroblocks above and columns from macroblocks left of the current position. Because rows represent vertical shapes and columns horizontal shapes, this will extrapolate existing image elements into the current macroblock. The stored coefficients from the bitstream are merely offsets that are added to the predicted ones.
6. The inverse quantization uses a quantization matrix to scale the coefficients. Adaptive quantization by changing the matrix within one frame is possible.
7. The block transform is the already known IDCT (see Subsection 3.2.1).
8. There is no spatial prediction in MPEG-4 part 2.
9. The temporal prediction for P- and B-frames is quite similar to the one in MPEG-2: P-frames can use up to one reference image block, selected by a motion vector, B-frames can use up to two references and interpolate between them. The motion vector precision has been doubled compared to MPEG-2 and allows for quarter pixel accuracy. One 16×16 macroblock can be divided into four 8×8 subblocks, each using an individual motion vector. The new S-frame uses one reference and shifts or warps the entire image using global motion vectors. The result of the temporal prediction

is then enhanced by adding the result of the IDCT to reduce the prediction error.

10. MPEG-4 part 2 contains an optional post processing step with different quality levels. It can reduce the block artifacts that can be seen on the macroblock boundaries and it can reduce ringing introduced by overly strong high frequency contributions in images with sharp edges.

The MPEG-4 part 2 standard has been extended to include support for nonrectangular video by storing a 1-bit alpha channel and like MPEG-2 it allows scalable decoding with a reduced resolution base layer and enhancement layers. The standard has been further polluted with concepts alien to traditional video compression, like FBA objects for face and body animation with synthesized lipsync mouth movements and fully scalable 3D meshes with compressed textures. All these features have never received a wide acceptance, but made the standard fairly big and difficult to implement.

Profiles are used to limit the amount of features a decoder has to implement, so I will name the two most important ones [11]:

Simple Visual Profile is designed for low power and low latency encoding and decoding, so it is suited for video communication and mobile applications. It is similar to H.263. The decoder must only support I- and P-frames and half pixel motion vector accuracy.

Advanced Simple Profile (ASP) targets maximum coding efficiency for desktop and broadcast video. The decoder must support I-, P-, B- and S-frames with global motion compensation and quarter pixel accuracy.

I will not consider any of the MPEG-4 part 2 features outside the advanced simple profile, because they are not commonly used and there are no working open-source decoder or encoder implementations available.

3.2.3 H.264

H.264 is the most advanced of the codecs described here and for reasons stated before, it is not considered for the actual decoding time prediction. However, it is still interesting to see that H.264 fits into our decoder model to demonstrate its generality.

1. The H.264 bitstream is divided into two layers: The network abstraction layer and the video coding layer. The latter contains the actual video stream, whereas the former stores sideband information for network streaming. This includes data partitioning techniques and redundant slices, which allows a decoder to tolerate or hide network packet loss to a certain degree. Streaming applications have been taken into account throughout the entire design of H.264 as we will see later. But because the format also targets the high-end broadcast market, it also includes features missing in previous standards, like frame numbering and auxiliary pictures, which can be used for alpha channels.

2. The decoder preparation can require considerable resources, especially memory, because unlike the typical two reference frames in previous standards, H.264 can use up to 32 reference frames.

3. The bitstream uses a context-adaptive variable-length coding for the macroblock coefficients with coding tables changing dynamically based on bitstream statistics. The side information and syntax elements of the bitstream are compressed with a context-adaptive binary arithmetic coding. The following decoding steps are controlled by the type information, which can be one of the usual I-, P- or B-type or one of the new SI- or SP-type. But unlike older algorithms, H.264 assigns the type at the slice level, not at the frame level. To accommodate network streamed content with multiple datarate versions of the

same video, the switching intra (SI) and switching predictive (SP) slice types allow the decoder to switch to a different datarate without having to worry about consistent reference information. In addition to that, macroblocks and slices within a frame can be ordered arbitrarily to allow regions of equal visual importance to be transferred together. With the support of slice priorities and visual utilization information, future decoders can scale their performance by dynamically omitting less important parts of the image or decoding them in a lower quality.

4. The inverse scan reads the coefficients into the 16×16 sized macroblock matrix. The macroblocks are divided into 4×4 sized sub-blocks.

5. There is no coefficient prediction in H.264. It has been superseded by a complete spatial prediction, see Step 8.

6. The inverse quantization tables are fixed in H.264 and designed to have several quantization levels that correlate linearly to perceived visual quality.

7. The block transform is not the IDCT, but an integer transform with similar properties (see Figure 5). It operates on 4×4 blocks of coefficients and is an exact-match calculation that can be implemented using only shift and add operations on 16 bit integers, thus preventing rounding errors in both the inverse and forward operation. This allows H.264 to operate completely lossless, when the quantization step is skipped.

8. A spatial prediction is done on I-slices, which extrapolates image content from the edges of neighboring macroblocks using different prediction methods.

9. For P- and B-slices, temporal prediction uses motion compensation to reuse image data of previously decoded frames. The block granularity can reach from 4×4 up to 16×16 blocks and the motion vectors use quarter pixel precision. While previous standards did not allow B-frames as reference, H.264 does not have such a limitation. The large number of up to 32 reference frames can be leveraged to do long-term predictions. P-type slices select one of those frames as the reference, B-type slices can use up to two of them with an arbitrarily weighted interpolation, allowing for efficient coding of crossfades. To limit the inter-frame dependencies, selected frames are tagged as IDR-frames and function as dependency boundaries. These frames only contain I-slices and therefore do not depend on other frames. Temporal prediction must never exceed the frame interval between the previous and the next IDR-frame.

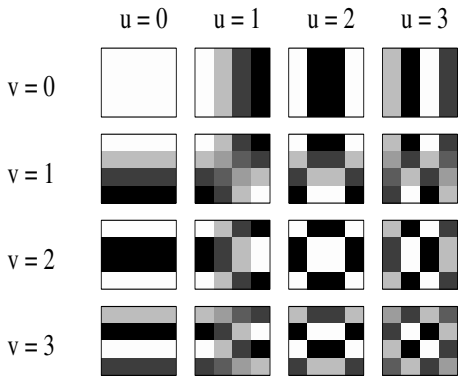

Figure 5: Base functions of the H.264 integer transform [9]

10. H.264 uses a mandatory deblocking filter that has to be applied before the final image is kept as a reference frame. This means the deblocking cannot be skipped because this would result in slightly wrong reference frames, introducing a drift between the original and the decoded images by an increasing residual error.

Despite its advancedness, I have shown that H.264 fits well into the proposed decoder model, so future application of the presented methods to predict H.264 frame decoding times should be possible.

3.2.4 DV

I only include a discussion of the DV standard to show that the decoder model can also accommodate lower complexity decoder algorithms. The DV format has not been developed for efficient compression in the sense of a high quality per byte ratio, but it is optimized for easy per-frame access to simplify video editing. As such, it does not exploit any temporal redundancy, but draws its compression effect from a lossy exploitation of spatial redundancy.

1. The DV bitstream is designed for random access. It does not consider network streaming and is of low overhead.
2. With respect to frames, the DV decoder is stateless, so no reference frame space needs to be prepared.
3. The DV format uses a JPEG-like compression for the individual frames. A variable length coding is used on the coefficients.
4. The inverse scan is a typical zig-zag scan.
5. There is no coefficient prediction.
6. Quantization is done by scaling the coefficients.
7. The block transform is the IDCT.
8. There is no spatial prediction.
9. There is no temporal prediction.
10. There is no post processing.

3.3 Metrics for Decoding Time

Using the XviD [21] implementation of the MPEG-4 part 2 algorithm and a selection of example videos ranging from highly compressed low resolution video up to high definition content, I measured the per-frame execution time spent inside the various function blocks on a 400 MHz Pentium II machine. The example videos listed in Table 1 naturally span a variety of stream parameters, which is necessary to evaluate their influence on the decoding time. If, for example, all videos had been of the same resolution, the influence of the pixel count on decoding time could not be seen.

The profiling facility included in the XviD implementation has been a good starting point for a logical splitting of the decoder to wrap the right functions for time measurement. It is interesting to see in Figure 6 how much time is spent in each function block for XviD on average. (This average has been taken over all sample videos from Table 1.) I will now go through the various function blocks of the decoder model and explain how their decoding time can be estimated using values derivable from the stream.

3.3.1 Bitstream Parsing

Because the bitstream is not always byte aligned and parts of it may be compressed, the parsing takes time. In my XviD measurements, the bitstream parsing also encompasses all the parsing inside the macroblocks and the glue code binding the function blocks together. Because most of this should take place at the macroblock level, we can expect the execution time for this step to be closely related to the amount of pixels per frame, because the amount of macroblocks to parse correlates well with the amount of pixels. This works well for I-frames as can be seen in Figure 7. (The samples appear in clusters because the pixel count is constant throughout each video.)

However, the P-, B- and S-frames show a different behavior. I chose the B-frames to demon-

	video name	duration	resolution	size	profile	properties
○	Voyager	1:20 min	352×288	13 MB	SP	low in movement, noisy image (TV recording)
◇	IBM commercial	1:50 min	352×240	2.8 MB	SP	low in movement, many monochrome shapes
◇	IBM commercial	1:50 min	352×240	508 KB	ASP	see previous, highly compressed
+	movie trailer	1:13 min	720×576	63 MB	SP	dynamic, fast cuts, DVD quality material
+	movie trailer	1:13 min	720×576	8.6 MB	ASP	dynamic, fast cuts
×	Amazon documentary	1:37 min	1440×1080	97 MB	ASP	high quality HD content, detailed images

profile names: SP=simple visual profile, ASP=advanced simple profile
The first column shows the symbol with which this stream is represented in the diagrams throughout this section.

Table 1: The sample videos used in the decoding time analysis.

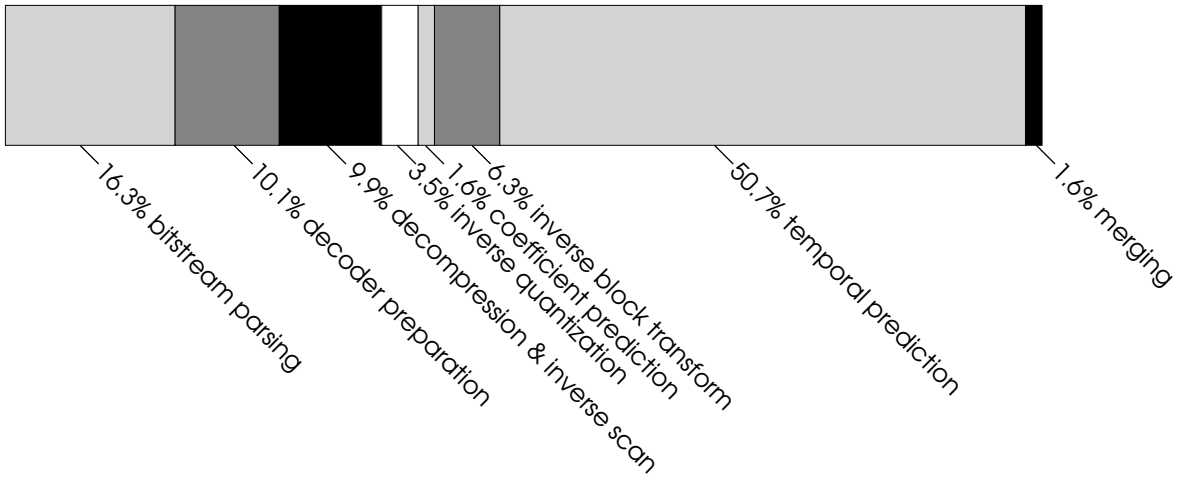

Figure 6: Average execution time profile for MPEG-4 part 2.

strate this in Figure 8, the P- and S-frames behave similar. Because these frame types allow a greater variety of coding options for each macroblock than the I-frames, which naturally consist entirely of intracoded macroblocks, the amount of bytes per frame has an influence, too. As can be seen in Figure 9, the result improves when taking this into account by calculating a linear combination of pixel count and bytes per frame that matches the execution time best. This idea also works for P- and S-frames.

I conclude that the pixel count as well as the amount of bytes per frame need to be included in the metrics to predict decoding time.

3.3.2 Decoder Preparation

The MPEG-4 part 2 algorithm extends the edges of reference frames before they are used for temporal prediction. The XviD implementation does this once for the entire reference frame before the macroblocks are evaluated. Because the length of the image edges correlates with the square root of the pixel count, I would expect a square root match between the pixel count and the execution time for this step. Figure 10 shows exemplary for P-frames that this assumption is correct.

The B- and S-frames behave the same and for I-frames, there are no reference frames to be modified. In earlier tests I experienced stray samples on the abscissae with S-frames, but this turned out to be a problem in XviD: The MPEG-

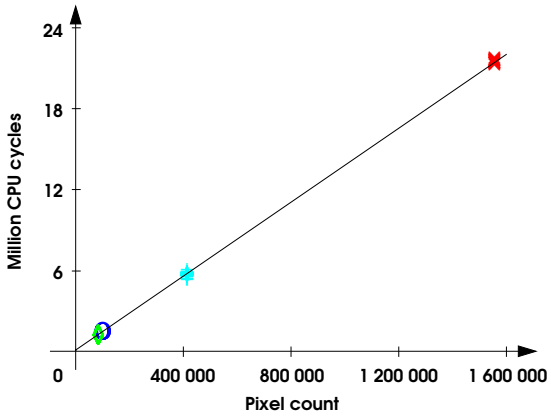

Figure 7: Estimating the bitstream parsing time for I-frames.

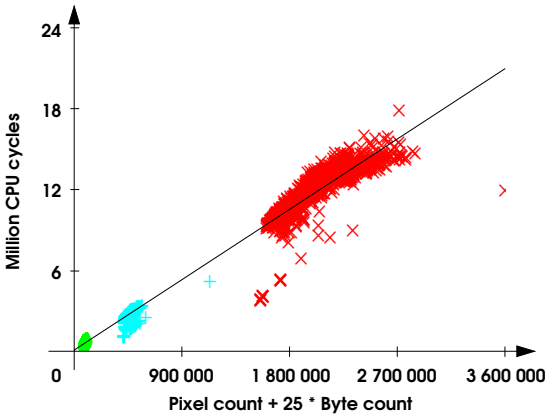

Figure 9: Estimating the bitstream parsing time for B-frames (2).

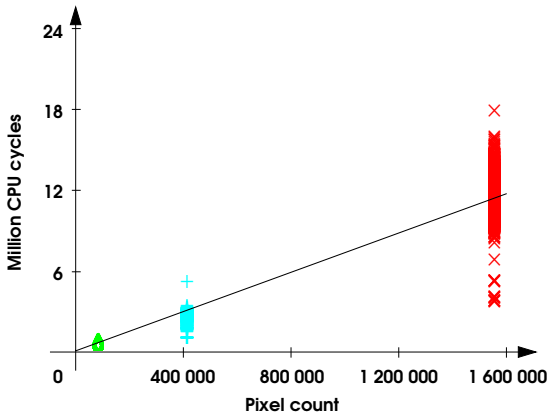

Figure 8: Estimating the bitstream parsing time for B-frames (1).

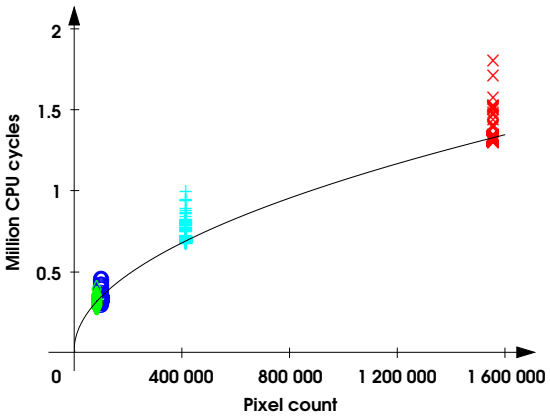

Figure 10: Estimating the decoder preparation time for P-frames.

4 part 2 standard allows for entire frames to be not coded, so they use up almost no CPU time. This could be regarded as a separate frame type (N-frame), which is what I am doing in my implementation of the prediction. XviD was erroneously reporting these frames as S-frames.

3.3.3 Decompression and Inverse Scan

The decompression should be the dominant task here, because the inverse scan is as easy as selecting a scan table and then using one table lookup for every decompressed coefficient. The decompression however is a variable length decompression, so the syntax elements are not byte aligned. The execution time correlates well across all frame types with the per-frame length of the bitstream, because most of the bitstream is spent on storing the macroblocks, header overhead is minimal. Figure 11 shows the match.

3.3.4 Inverse Quantization

The inverse quantization is done once for every macroblock, so it correlates well with the to-

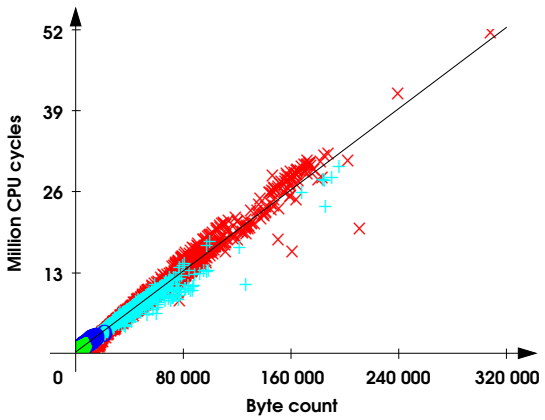

Figure 11: Estimating the decompression time for I-, P-, B- and S-frames.

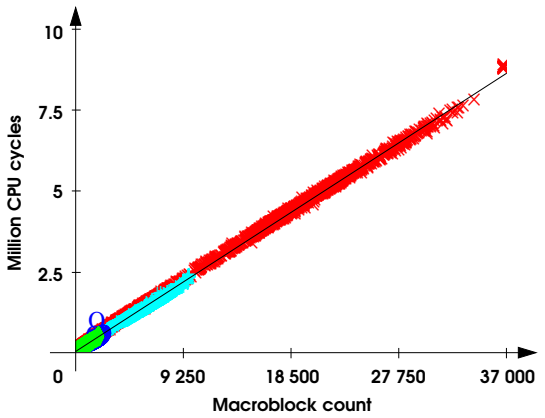

Figure 12: Estimating the inverse quantization time for I-, P-, B- and S-frames.

tal macroblock count. The macroblock count should not be mistaken for a scaled pixel count. Because of not coded macroblocks, the pixel count only yields an upper bound for the macroblock count. The diagram for all frame types can be seen in Figure 12.

3.3.5 Coefficient Prediction

The coefficient prediction is only done for intracoded macroblocks, so the execution time of this step correlates with the count of this macroblock type. Intracoded macroblocks can occur within every frame type, but the estimation works equally well for all types as shown in Figure 13.

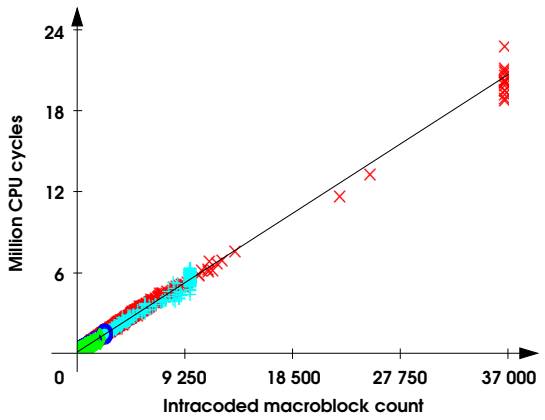

Figure 13: Estimating the coefficient prediction time for I-, P-, B- and S-frames.

3.3.6 Inverse Block Transform

As with the inverse quantization, the total macroblock count gives a good estimate, as Figure 14 shows. There is still deviation from the linear match, which might stem from either cache effects or different amounts of zero values in the macroblocks. Depending on the implementation, algorithms can be faster when more coefficients are zero. However, looking into that has the disadvantage of increased prediction overhead, and because the inverse block transform only accounts for 3.9 % of the total decoding time, I decided not to count zero values separately.

3.3.7 Temporal Prediction

This step is the most time consuming. Unfortunately, its execution time is also the hardest to predict. Because motion compensation is only done for intercoded macroblocks, one might be tempted to derive the execution time from the count of intercoded macroblocks. Figure 15 shows that this fails.

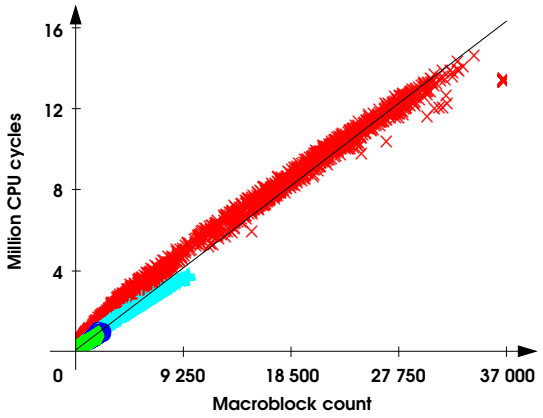

Figure 14: Estimating the inverse block transform time for I-, P-, B- and S-frames.

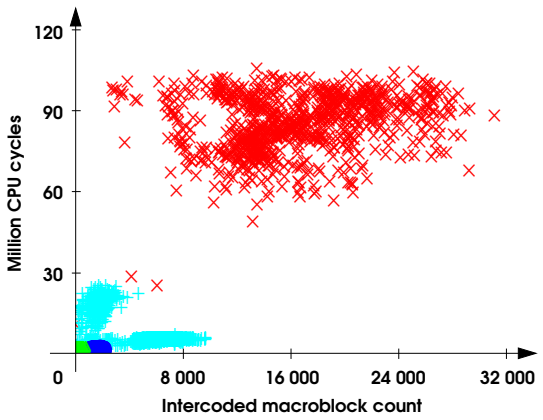

Figure 15: Estimating the temporal prediction time for P-frames (1).

The reason is that there are various different methods of motion compensation due to macroblock subblocking and different storage types for motion vectors. These options are independent of the macroblock type. Searching for a more adequate metric, I first tried to distinguish between all these methods and account for them individually, but this strategy is problematic, because there are too many combinations and covering them all with sample videos is difficult. But when looking at the bigger picture, I realized that motion compensation is basically just copying pixels from a reference image into the

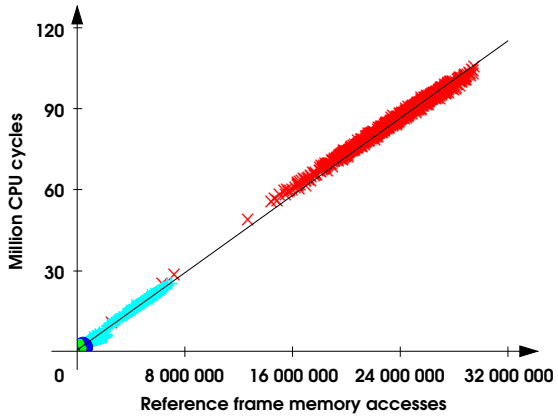

Figure 16: Estimating the temporal prediction time for P-frames (2).

current frame. Because of half pixel or quarter pixel accuracy and the necessary interpolation and filtering, one pixel copy operation can range from 1 to 20 memory accesses. Therefore, the number of memory accesses into the reference frame should result in a far better prediction. Of course this number cannot be measured directly, but looking at the code for motion compensation, we can easily count the memory accesses. So depending on the lower bits of the motion vectors, which differentiate between full, half or quarter pixel references, I created a formula to calculate the number of memory accesses. For that, the motion vectors need to be decoded completely, which takes time and increases prediction overhead. However, because the temporal prediction accounts for a big portion of the overall decoding time, I think this step is necessary. I will show the effect on the overhead in Subsection 6.1.1. The promising results for P-frames can be seen in Figure 16. This works equally well for B-frames.

It may be a bit surprising that memory accesses alone estimate the execution time so well, given that different interpolation and filtering is done for full, half and quarter pixel accesses. I assume that with today's processors, these additional operations are covered up by the memory accesses because of parallel execution.

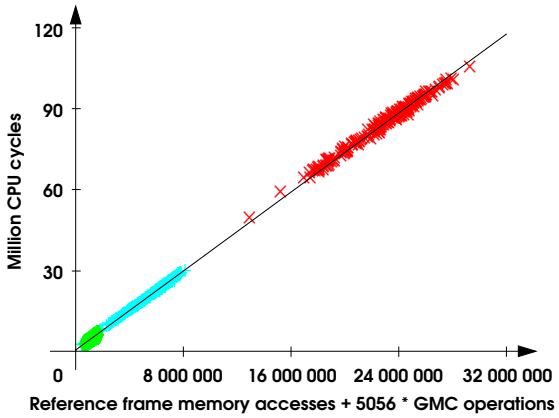

Figure 17: Estimating the temporal prediction time for S-frames.

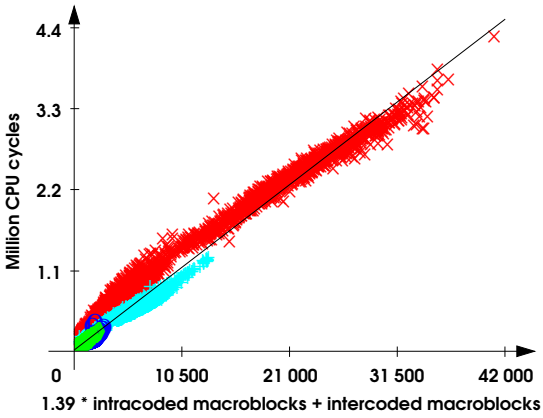

Figure 18: Estimating the merging time for P-, B- and S-frames.

S-frames, however, encompass global motion compensation (GMC), so a different approach is required here. Because I did not want to look into the more complicated warping algorithm, I just counted the number of macroblocks using global motion compensation and calculated a linear combination of the memory accesses from non-GMC macroblocks and the amount of GMC macroblocks to match the execution time. The good results shown in Figure 17 indicate that the execution time per GMC macroblock is fairly constant.

3.3.8 Merging

The merging step combines the results of the temporal prediction with the decoded macroblocks, so a linear match of intra- and intercoded macroblock counts should estimate the execution time well enough. Because this step has only a small influence on the total decoding time, we can tolerate the deviations seen in Figure 18, which are possibly caused by cache effects or saturation artifacts. (The saturation of the 16 bit integers used throughout the decoding process to 8 bit integers in the final image also takes place here.)

3.3.9 Summary

I have now established which metrics are useful to get reasonable estimations for the execution time of the various stages. Because the time required for the entire decoding process is the sum of the separate execution times, these metrics will also allow predictions for the frame decoding times. The metrics are:

- pixel count
- square root of pixel count
- byte count
- intracoded macroblock count
- intercoded macroblock count
- motion compensation reference frame memory accesses
- global motion compensation macroblock count

With similar considerations as presented for MPEG-4 part 2, I also selected the necessary metrics for MPEG-1/2, which I will name here:

- pixel count

- byte count
- intracoded macroblock count
- intercoded macroblocks without IDCT shortcut
- intercoded macroblocks with IDCT shortcut
- motion compensation reference frame memory accesses

The only notable difference to the MPEG-4 part 2 metrics is the differentiation of the intercoded macroblocks. The libmpeg2 implementation of the IDCT uses a shortcut for macroblocks where all AC coefficients are zero. This situation is easy to detect during macroblock decompression, so the overhead introduced by this step is negligible.

How to extract the metrics from the stream will be covered in Section 5. This leaves the problem of automatically determining the factors to apply to the metrics to get a decoding time prediction. This will be discussed in the following section.

4 Numerical Background

I have now extracted a set of metric values for each frame of the video. In a learning stage, of which I will present details in Section 5, I will therefore receive a metric vector and the measured frame decoding time for each frame. Accumulating all the metric vectors as rows of a metric matrix M and collecting the frame decoding times in a column vector $\underline{t}$, I now want to derive a column vector of coefficients $\underline{x}$, which will, given any metric row vector $\underline{m}$, yield a predicted frame decoding time $\underline{m}\,\underline{x}$.

I will first summarize the used variables:

m	the number of metric vectors accumulated during learning
n	the number of metrics extracted per frame
M	$m \times n$ matrix of accumulated metric vectors
$\underline{m}_i$	the i'th row of M
$\underline{t}$	m-dimensional column vector holding the frame decoding times
t_i	the i'th component of $\underline{t}$
$\underline{x}$	n-dimensional column vector holding the prediction coefficients

Because the prediction coefficients $\underline{x}$ must be derived from M and $\underline{t}$ alone, I model the situation as a linear least square problem (LLSP):

$$\|M\underline{x} - \underline{t}\|_e^2 \rightarrow \min_{\underline{x}}$$

That means the accumulated error between the prediction $M\underline{x}$ and the actual frame decoding times $\underline{t}$ must be minimal. The error is expressed by the square of the Euclidean norm, which is the sum of the squared error per row as we can easily see:

$$\begin{aligned} \|M\underline{x} - \underline{t}\|_e^2 &= (M\underline{x} - \underline{t})^T (M\underline{x} - \underline{t}) \\ &= \sum_{i=1}^{m} (\underline{m}_i\,\underline{x} - t_i)^2 \end{aligned}$$

In general, the error will not disappear, because this would be equivalent to solving the linear system $(M, \underline{t})$:

$$\|M\underline{x} - \underline{t}\|_e^2 = 0 \quad \text{iff} \quad M\underline{x} = \underline{t}$$

But solving this equation is not possible in general, because we are examining a large number of frames, thus $m > n$. Thus, the linear equation $M\underline{x} = \underline{t}$ is overdetermined and the solution $\underline{x}$ need not exist.

For the original linear least square problem, two common solving methods exist: The normal equation and the QR decomposition [24, 25].

The normal equation derives $\underline{x}$ by solving the linear equation $M^T M\underline{x} = M^T\underline{t}$. I will not explain why this solves the original problem, because I am not going to pursue this solution. The normal equations have benefits for large m, because the matrix $M^T M$ is only of size $n \times n$, and if M can be assumed to have full rank, there are fast solvers like Cholesky factorization [26], but normal equations have the disadvantage to be numerically unstable [27]. In comparison, the QR decomposition is more insensitive against badly conditioned matrices M, and because the calculation of the coefficients $\underline{x}$ is not time critical, we value a stable result higher than calculation speed.

However, we can see from the normal equation that, given a full rank matrix M, the matrix $M^T M$ is symmetric and positive definite, so there is exactly one solution $\underline{x}$. The full rank property will be dealt with later, and the QR decomposition algorithm we are going to use detects matrices with rank deficiencies as we will see in Subsection 4.2.

4.1 QR Decomposition

The solution of the linear least square problem $\|M\underline{x} - \underline{t}\|_e^2 \to \min$ discussed previously is calculated by a QR decomposition of M. This transformation creates two matrices Q and R, for which $M = QR$ holds, whereby the matrices have these additional properties:

- Q is a $m \times m$ real matrix,
- R is a $m \times n$ real matrix,
- Q is orthogonal, so $Q^{-1} = Q^T$,
- R is in the form

$$R = \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} \begin{matrix} \}n \\ \}m-n \end{matrix}$$

  with $\hat{R}$ being a real $n \times n$ upper triangular real matrix, the remaining rows of R are filled with zeros.

- If M has full rank, $\hat{R}$ is regular, because the diagonal elements $\hat{r}_{ii}$ are not zero.

Using these properties of Q and R, we can reformulate the original problem:

$$\begin{aligned}
\|M\underline{x} - \underline{t}\|_e^2 &= \|QR\underline{x} - \underline{t}\|_e^2 \\
&= \left\|QR\underline{x} - QQ^{-1}\underline{t}\right\|_e^2 \\
&= \left\|QR\underline{x} - QQ^T\underline{t}\right\|_e^2 \\
&= \left\|Q\left(R\underline{x} - Q^T\underline{t}\right)\right\|_e^2 \\
&= \left(Q\left(R\underline{x} - Q^T\underline{t}\right)\right)^T\left(Q\left(R\underline{x} - Q^T\underline{t}\right)\right) \\
&= \left(R\underline{x} - Q^T\underline{t}\right)^T Q^T Q\left(R\underline{x} - Q^T\underline{t}\right) \\
&= \left(R\underline{x} - Q^T\underline{t}\right)^T Q^{-1} Q\left(R\underline{x} - Q^T\underline{t}\right) \\
&= \left(R\underline{x} - Q^T\underline{t}\right)^T\left(R\underline{x} - Q^T\underline{t}\right) \\
&= \left\|R\underline{x} - Q^T\underline{t}\right\|_e^2 \\
&= \left\|\begin{pmatrix} \hat{R} \\ 0 \end{pmatrix}\underline{x} - \begin{pmatrix} \underline{\hat{c}} \\ \underline{c_R} \end{pmatrix}\right\|_e^2 \\
&= \left\|\begin{pmatrix} \hat{R}\underline{x} \\ 0 \end{pmatrix} - \begin{pmatrix} \underline{\hat{c}} \\ \underline{c_R} \end{pmatrix}\right\|_e^2 \\
&= \left\|\begin{pmatrix} \hat{R}\underline{x} - \underline{\hat{c}} \\ 0 - \underline{c_R} \end{pmatrix}\right\|_e^2 \\
&= \left\|\hat{R}\underline{x} - \underline{\hat{c}}\right\|_e^2 + \left\|-\underline{c_R}\right\|_e^2 \\
&= \left\|\hat{R}\underline{x} - \underline{\hat{c}}\right\|_e^2 + \left\|\underline{c_R}\right\|_e^2
\end{aligned}$$

$Q^T\underline{t}$ has been divided similar to R:

$$\underline{c} := Q^T\underline{t} = \begin{pmatrix} \underline{\hat{c}} \\ \underline{c_R} \end{pmatrix} \begin{matrix} \}n \\ \}m-n \end{matrix}$$

Because only $\left\|\hat{R}\underline{x} - \underline{\hat{c}}\right\|_e^2$ depends on $\underline{x}$ and squares can never be negative, we can reach our solution by

$$\|M\underline{x} - \underline{t}\|_e^2 = \left\|\hat{R}\underline{x} - \underline{\hat{c}}\right\|_e^2 + \left\|\underline{c_R}\right\|_e^2 \to \min_{\underline{x}}$$

iff

$$\hat{R}\underline{x} - \underline{\hat{c}} = 0$$

$$\text{iff}$$

$$\hat{R}\underline{x} = \underline{\hat{c}}$$

This equation is easy to solve, because $\hat{R}$ is upper triangular and always gives a result if M has full rank, because then, $\hat{R}$ is regular.

The remaining error can be expressed as

$$\min_{\underline{x}} \|M\underline{x} - \underline{t}\|_e^2 = \left\|\underline{c_R}\right\|_e^2$$

The calculation of R and $\underline{c}$ (and thus of $\hat{R}$ and $\underline{\hat{c}}$) can be unified into one step:

$$\begin{aligned} R &= Q^T M \\ \underline{c} &= Q^T \underline{t} \\ &\text{iff} \\ (R \mid \underline{c}) &= Q^T (M \mid \underline{t}) \end{aligned}$$

So we can construct the algorithm to reach the final result:

1. Add the vector $\underline{t}$ as an additional column to the right of matrix M.
2. Use a QR decomposition algorithm to decompose $(M \mid \underline{t})$, resulting in $(R \mid \underline{c})$, which is of dimension $m \times (n+1)$.
3. Take the upper n rows of R and $\underline{c}$, resulting in $\hat{R}$ and $\underline{\hat{c}}$, which have the necessary properties, because the upper $n+1$ rows of $(R \mid \underline{c})$ are upper triangular, so the upper n rows of R are upper triangular too.
4. Solve the staggered system $\hat{R}\underline{x} = \underline{\hat{c}}$ to gain the solution $\underline{x}$.

The following interesting facts should be noted:

- As a result of the QR decomposition, the lower $m-(n+1)$ rows of $(R \mid \underline{c})$ are filled with zeros, which makes every element of $\underline{c}$ but the first $n+1$ disappear. Because the lower $m-n$ elements of $\underline{c}$ make up for $\underline{c_R}$, every element in $\underline{c_R}$ but the first is zero. So the remaining error easily calculates as

$$\min_{\underline{x}} \|M\underline{x} - \underline{t}\|_e^2 = \left\|\underline{c_R}\right\|_e^2 = c_{R_1}^2 = c_{n+1}^2 \tag{1}$$

  with c_{R_1} being the first element of $\underline{c_R}$ and c_{n+1} being element $n+1$ of $\underline{c}$. This error is called the residual sum of squares.

- The subdiagonal elements and the zero-filled rows of $(R \mid \underline{c})$ do not need to be touched again, so implementations can be optimized by not actually zero-filling them.
- The value of Q is never used explicitly, only R is needed. So the QR decomposition algorithm can benefit from that by not actually calculating Q.

4.2 Householder's Algorithm

Several other algorithms for the QR decomposition exist, like the Gram-Schmidt [28] or Givens [29] algorithms, but Householder [28] is the most commonly used, because of its numeric stability and simplicity. It is also suitable for the given task, because it directly calculates R, but not Q. The principle of Householder's algorithm is iteratively zeroing the subdiagonal elements of the initial matrix $A^{(1)}$. This transformation is done on a per column basis. In Householder iteration i, the columns of $A^{(i)}$ are treated in this fashion:

1. Given a matrix $A^{(i)}$ of dimension $k_i \times l_i$ (with $k_i > l_i$ and $l_i > 0$), we divide the first column $\underline{a}_1^{(i)}$ of $A^{(i)}$ like this:

$$\underline{a}_1^{(i)} = \begin{pmatrix} p_i \\ \underline{q}^{(i)} \end{pmatrix} \begin{matrix} \}1 \\ \}k_i - 1 \end{matrix}$$

   and we construct an orthogonal matrix $Q^{(i)}$, so that

$$Q^{(i)}\underline{a}_1^{(i)} = \begin{pmatrix} \sigma_i \\ 0 \end{pmatrix} \begin{matrix} \}1 \\ \}k_i - 1 \end{matrix}$$

2. Householder's solution is:

$$\begin{aligned}
\sigma_i &= -\operatorname{sgn}(p_i) * \sqrt{p_i^2 + \left\|\underline{q}^{(i)}\right\|_e^2} \\
&= -\operatorname{sgn}(p_i) * \sqrt{p_i^2 + \underline{q}^{(i)^T}\underline{q}^{(i)}} \\
&\quad \text{with} \quad \operatorname{sgn}(p) = \begin{cases} 1 & \text{for } p \geq 0 \\ -1 & \text{for } p < 0 \end{cases} \\
\underline{v}^{(i)} &= \begin{pmatrix} p_i - \sigma_i \\ \underline{q}^{(i)} \end{pmatrix} \\
Q^{(i)} &= I + \frac{1}{\sigma_i (p_i - \sigma_i)} \underline{v}^{(i)} \underline{v}^{(i)^T}
\end{aligned}$$

by showing $Q^{(i)}Q^{(i)^T} = I$, $Q^{(i)}$ can be proven orthogonal. (I is an identity matrix of a fitting size.) But because $Q^{(i)}$ is not needed explicitly to proceed, it does not have to be calculated.

3. The vector $Q^{(i)}\underline{a}_1^{(i)} = \begin{pmatrix} \sigma_i \\ 0 \end{pmatrix}$ replaces the first column of $A^{(i)}$.

4. The remaining columns $\underline{a}_j^{(i)}$ $(j = 2, \ldots, l_i)$ of $A^{(i)}$ are each replaced with

$$Q^{(i)}\underline{a}_j^{(i)} = \underline{a}_j^{(i)} + \frac{\underline{v}^{(i)^T}\underline{a}_j^{(i)}}{\sigma_i (p_i - \sigma_i)} \underline{v}^{(i)}$$

5. Take everything but the first row and the first column of $A^{(i)}$ and call this $A^{(i+1)}$. Set $k_{i+1} = k_i - 1$ and $l_{i+1} = l_i - 1$. If $l_{i+1} > 0$ (that is: $A^{(i+1)}$ has at least one column), repeat at step 1, otherwise the algorithm has finished.

The final result R is gained by unnesting the $A^{(i)}$, as if the transformation of $A^{(i+1)}$ had been done inside $A^{(i)}$:

$$R = \begin{pmatrix}
\sigma_1 & a_2^{(1)} & a_3^{(1)} & a_4^{(1)} & \cdots & a_{l_1}^{(1)} \\
0 & \sigma_2 & a_2^{(2)} & a_3^{(2)} & \cdots & a_{l_2}^{(2)} \\
0 & 0 & \sigma_3 & a_2^{(3)} & \cdots & a_{l_3}^{(3)} \\
0 & 0 & 0 & \sigma_4 & \cdots & a_{l_4}^{(4)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & \cdots & \sigma_{l_1} \\
0 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & \cdots & 0
\end{pmatrix}$$

with $a_j^{(i)}$ being the first element of column vector $\underline{a}_j^{(i)}$.

I am not going to show how Q can be derived from the $Q^{(i)}$ and why Q is orthogonal. This can be found in literature [28].

It can be seen that a subproblem $\widetilde{A}$ of $A := A^{(1)}$, created by deleting columns from the right hand side of A, will have a solution $\widetilde{R}$, which can be derived from the solution R to A by deleting the same amount of columns from the right hand side of R. The reason is that the individual iterations work from left to right through the matrix, so every column in the result is not influenced by columns further to the right. Therefore, an entire Householder QR decomposition will solve not only the problem A, but also all subproblems $\widetilde{A}$. As we have seen in Equation 1 on the preceding page, the residual sum of squares of a $m \times n$ linear least square problem is the square of the matrix element $r_{n,n}$ in R. This value can be found as the final element in the σ-diagonal. The same applies to the subproblems, so all σ_i are residual sums of squares of their related subproblems. Whenever any of these values becomes zero, we know there is a subproblem inside A with a residual error of zero. This means there is a set of coefficients, with which one column can be described exactly as a linear combination of other columns, so there is a linear dependency in A and A does not have full rank. Because every combination of columns is checked during an entire QR decomposition,

every matrix without full rank will show up as a disappearing σ value. Using this information to drop one of the dependent columns as will be discussed in Subsection 4.3, the algorithm can ensure its own prerequisites, because a full rank is required for obtaining a solution.

The numerical stability of the Householder algorithm can be further improved by using column pivoting: At the beginning of each iteration i, select the column with the largest euclidean norm and swap it with the leading column. Of course this will give permutated results.

For our initial problem, the Householder algorithm starts with $A^{(1)} = (M \mid \underline{t})$ and $k_1 = m$, $l_1 = n$.

4.3 Refining the Result by Column Dropping

To ease implementations of prediction for new decoder algorithms, we want to simplify the process of metrics selection for the decoding time estimation. In general, it should be possible to feed the LLSP solver with sensible metrics and it should figure out, which ones to use and which ones to drop by itself. Of course, the best result for the linear least square problem is always achieved by using as many metrics as possible, but one of the design goals was to make the results transferable to other videos, which might not always work when using metrics too greedily.

I will first examine why results might not be transferable. The primary reason for this are similarities of columns inside the metric matrix with linear combinations of other columns. The special case of this situation is an actual linear dependency. This will lead to ambiguities in the resulting coefficients, such that we can increase certain coefficients and compensate by decreasing others with little or no influence on the prediction results. The barebone LLSP solver will always search for the optimum, which might have such unwanted properties as negative coefficients or coefficients being otherwise way off.[3] Those coefficients will of course still obtain good results for the video they have been calculated from, but the results for other videos may be completely wrong. The goals to reduce such effects must therefore be:

- Eliminate any negative coefficients, because they clearly indicate something going wrong. Our decoder model is a chain of function blocks with varying execution times summing up to a total execution time. But no such function block can have a negative execution time.
- Eliminate any further linear dependencies or situations close to a linear dependency that make the coefficients unstable.

The strategy we are going to use to make the result better transferable is iteratively dropping columns until all coefficients are positive and inter-column dependencies have been reduced. This leads to a reduction in the columns while still keeping enough columns to maintain the quality of the prediction.

As we have seen in Equation 1 on page 25, the remaining error, called residual sum of squares, for an n-column matrix is the square of the value in the n'th column of the n'th row after the QR decomposition. This value indicates the quality of the prediction. The smaller, the better. If we have to drop columns for transferability, we want to do so without too much degradation on the quality of the result. Therefore, we iteratively drop columns and then choose the one that best fits our two goals, but results in the smallest increase of this error indicator.

[3] Example:

$$\begin{array}{rcrcr} x_1 & + & x_2 & = & 2 \\ x_1 & + & 1.0001x_2 & = & 2.0001 \end{array}$$

leads to $x_1 = x_2 = 1$, but if the right hand side of the second equation had been 1.9999 due to a measurement error, the result suddenly becomes $x_1 = 3$, $x_2 = -1$, because of the badly conditioned system, which almost has a linear dependency in it.

A linear dependency or a situation close to it can also be detected with this indicator: If we drop a column and there is only a minor increase in the residual sum of squares, the dropped column had little to no influence on the result, so the column can be approximated as a linear combination of others:

Given columns $\underline{c}_1$ to $\underline{c}_{n+1}$ and an error vector $\underline{e}$ with $\|\underline{e}\|_e^2 = \underline{e}^T\underline{e} = \left\|\underline{c}_R\right\|_e^2$, we have this solution to the LLSP:

$$x_1\underline{c}_1 + x_2\underline{c}_2 + \cdots + x_i\underline{c}_i + \cdots + x_n\underline{c}_n = \underline{c}_{n+1} + \underline{e} \tag{2}$$

If we drop column $\underline{c}_n$, we get a new solution by QR decomposing the remaining columns:

$$x'_1\underline{c}_1 + x'_2\underline{c}_2 + \cdots + x'_i\underline{c}_i + \cdots + x'_{n-1}\underline{c}_{n-1} = \underline{c}_{n+1} + \underline{e}' \tag{3}$$

By subtracting Equation 2 from Equation 3, we can express $\underline{c}_n$:

$$\left(x'_1 - x_1\right)\underline{c}_1 + \cdots + \left(x'_{n-1} - x_{n-1}\right)\underline{c}_{n-1} = x_n\underline{c}_n + \left(\underline{e}' - \underline{e}\right)$$

$$\frac{x'_1 - x_1}{x_n}\underline{c}_1 + \cdots + \frac{x'_{n-1} - x_{n-1}}{x_n}\underline{c}_{n-1} + \frac{1}{x_n}\left(\underline{e} - \underline{e}'\right) = \underline{c}_n$$

If the residual sum of squares is small and changes only marginally, $\|\underline{e}\|_e^2 \approx \|\underline{e}'\|_e^2$ holds.

$$\begin{aligned} \left\|\underline{e} - \underline{e}'\right\| &= \left\|\underline{e} + \left(-\underline{e}'\right)\right\| \\ &\leq \|\underline{e}\| + \left\|-\underline{e}'\right\| \\ &= \|\underline{e}\| + \left\|\underline{e}'\right\| \\ &\approx 2\,\|\underline{e}\| \end{aligned}$$

Thus, if $\|\underline{e}\|_e^2$ has been sufficiently small, $\left\|\frac{1}{x_n}\left(\underline{e} - \underline{e}'\right)\right\|_e^2$ is small and the column $\underline{c}_n$ can be approximated using the other columns, resulting in a badly conditioned system.

I propose this algorithm to eliminate such situations:

1. Solve the linear least square problem.
2. Count the negative coefficients in the result.
3. Try dropping every column but the final one and compute the individual reduced linear least square problems. It is obvious that the final column must not be dropped, because it holds the measured decoding times, the target of our entire prediction.
   If no columns can be dropped, because the problem does not have enough columns left, we terminate and output the set of columns dropped so far.
4. For all solutions of the reduced problems, count the negative coefficients in the results.
5. Select a reduced problem based on the following criteria:
   a) Among the problems with fewer negative coefficients than the original unreduced one, select the one with the fewest negative coefficients. If several problems have an equally low amount of negative coefficients, select the one with the smallest increase on the error indicator.
      Go to step 6, if a problem has been chosen.
   b) Among the problems with a residual sum of squares less than 0.01 % (explanation follows) higher than the residual sum of the original unreduced problem, select the one with the lowest increase[4] on the error.
      Go to step 6, if a problem has been chosen.
   c) If no problem has been selected yet, the algorithm terminates outputting

[4] Although the residual sum of squares should never decrease when dropping a column, this can happen due to rounding errors. So a decrease should be treated like a negative and thus small increase here.

the cumulative set of columns that have been dropped earlier.

6. Remember the column that has been dropped to form the selected reduced problem. Restart the algorithm at step 1 with the selected reduced problem.

The 0.01 % tolerance used has been chosen empirically. Higher values will obviously make the algorithm more robust against numerical instabilities, but they might also result in columns being dropped that are significant to the result. Remember that the coefficients are only derived from a small set of learning videos. Columns with only little influence on this set may be important for predicting other video's decoding times. I received good results with the 0.01 %. When the metrics are selected wisely, even lower values down to 0 should work. If one is unsure about the metrics, for example when implementing support for a new algorithm, it might help to increase this tolerance level until more refined metrics are available.

This algorithm always terminates by construction. It is also correct according to our goals, because it will only drop a column, when this either reduces the amount of negative coefficients or results in only a minor increase on the residual sum of squares, in which case we have shown that a linear approximation for the dropped column existed. These situations are exactly what we wanted to avoid. Unfortunately the algorithm cannot guarantee any properties on the resulting final solution, because we can construct a linear least square problem that always results in negative coefficients.[5] However, such problems never appear within our application, because we can assume any sensible metrics selection to contain a number of columns that correlate positively with the measured decoding time.

[5] All columns but the final one being filled with negative values, the final one being positive will do. No matter what columns you drop, the result will always contain negative coefficients.

5 Implementation

The implementation is threefold: At the beginning stands the basic solver for the linear least square problem with the additional column dropping. This solver is fed with the extracted metrics from the video stream. All that is then integrated into VERNER [19] with considerations on frame reordering. The entire programming interface together with additional documentation for each function can be found in the file `predict.h`.

5.1 Metrics Accumulation and LLSP Solver

The LLSP (linear least square problem) solver and predictor supports two phases of operation:

- Learning mode, in which the solver accumulates metrics and frame decoding times
- Prediction mode, in which previously obtained coefficients are used to calculate predicted frame decoding times

During learning mode, the solver collects metric values in a matrix by calling `llsp_accumulate()` repeatedly. This function takes the extracted metrics and the measured frame decoding time and appends it as a new row to an internally kept matrix. This matrix is stored in the `learning_t` structure as a linked list of blocks of matrix rows. Because the final size of the matrix is unknown, new blocks will be allocated as needed. Newly allocated blocks will have a random size to hold between 1024 and 2048 rows. The randomization is done to avoid any possible aliasing of the allocation overhead and properties of the stream influencing decoding time. The allocation of a new block transiently changes timing and cache behavior, which might have unforeseeable and unwanted influences on the measured decoding times. If, for example, allocation always happens at an I-frame, this

influence could unfavorably alter the calculated coefficients for I-frames. The randomization is hoped to level off such influences. If no allocation happens, the accumulation function is fast and has $O(n)$ complexity with n being the number of metric values to collect, so video players can use it during regular playback.

If the data accumulation is finished, calling `llsp_finalize()` will run the actual solver. The solver implements column dropping using a bitfield: Every bit of an `unsigned long long` variable represents one column of the matrix. If the bit is set, the column is used, otherwise dropped. Unfortunately, this limits the possible amount of metric columns to 64 on current architectures, but this amount is sufficient for the algorithms I examined.

The solver iteratively calculates a QR decomposition, checks all remaining columns for drop candidates, and then decides, whether a column should be dropped. If this applies, the next iteration starts, otherwise the loop exits. A column is dropped if it either results in negative coefficients or has an influence below 0.01 % on the prediction result. For enhanced numerical stability on rank-deficient matrices, the QR decomposition uses column pivoting during the iterations. The numerical background has been explained in Subsection 4.3. Because the QR decomposition is performed with Householder's algorithm, we require efficient column-wise access to the matrix. Therefore the matrix is transposed when copied from the accumulated blocks, which allows for better cache use when walking the columns. It also makes the column swapping needed for the pivoting easier, because this can be achieved by simple pointer swapping. Of course the implementation exploits that we do not need the matrix Q. Householder calculates only R directly. Q would need to be derived from intermediate results by expensive matrix multiplications. We also do not fill the subdiagonal elements of R with zeros, because they are never touched again.

One Householder step as well as the norm calculation for the pivoting make use of a scalar product function. Because the scalar product can suffer from truncation errors when small contributions are added to an already large sum[6], this function uses `long double` format for additional accuracy.

The entire finalization has a complexity of $O\left(m * n^4\right)$, with m being the number of accumulated rows and n being the number of metrics. Every householder step is $O(m * n)$ and there are $O(n)$ steps needed for an entire QR decomposition. Each column is then checked for dropping, which accounts for $O(n)$ QR decompositions per dropped column. At most n columns can be dropped, resulting in a total of $O(m * n) * O(n) * O(n) * O(n) = O\left(m * n^4\right)$. This may look scary, but n is typically fixed and small compared to m being unbound: While n is in the magnitude of 10, 40 seconds of video at 25 frames per second already make up for $m = 1000$. Given that the video length has only linear influence on execution time and that speed is not critical in the final calculations, the algorithm is well suited for our needs.

The prediction mode can then calculate predicted frame decoding times based on the determined coordinates. The function `llsp_predict()` provides such an estimate, given the extracted metrics. Additional functions allow loading and storing the resulting prediction coefficients, so that the learning stage does not need to be repeated all the time. The file format of the coefficient storage is simple and optimized for easy access: One file contains multiple blocks of coefficients, each of which is able to hold the maximum number of coefficients supported. The blocks can be accessed by using the identification number, which every LLSP solver gets assigned at initialization, as an index. The individual coefficients are just dumped `double` values. Because the format is not too sophisticated, this should be regarded as a demonstrational feature only. A final file

[6] Example: with standard IEEE-754 64-bit floating point (usually C's `double` type), the condition $\left(10^{10} + 10^{-10}\right) = 10^{10}$ holds. The 10^{-10} is truncated.

format should contain versioning features and the values should be stored in an architecture independent, endian-safe way.

The loading, storing, finalizing, and predicting is nondestructive, so it is possible to accumulate metrics, finalize and store the determined coefficients and then accumulate additional metrics, which will be appended to the existing ones. However, the functions are currently not reentrant and it is not recommended to run the finalization stage in a low-priority background thread. This might be helpful to keep a player application responsive while calculating the coefficients, but the necessary locking is not implemented.

5.2 Metrics Extraction

The extraction wraps the generic LLSP solver described previously in an even easier API specific to video decoding. Unlike the LLSP solver, this functionality depends on the actual decoder algorithm. Every supported decoder is represented by six functions, two of which only provide the usual create and destroy infrastructure. The remaining functions pass a block of compressed video for metrics extraction, pass a decoding time for learning, trigger the final evaluation of the learned data, and reset the metrics extractor. Currently, a combined MPEG-1/2 and a MPEG-4 part 2 predictor are implemented. Each prediction context stores multiple LLSP solvers internally, one for each frame type the decoder algorithm uses. This allows for metrics of different frame types to be accumulated and evaluated simultaneously, but independently.

The metrics extraction itself is done similarly to the actual decoding. I extended an open-source decoder for the algorithm with additional instructions to determine the desired metrics and report them to the outside. After that, all actual decoding code has been taken out, so that only the bitstream parsing remained. This was further stripped to get a minimalistic "decoder" that does not do any decoding any more, but only parses the bitstream as much as necessary to retrieve the metrics. This code is called in much the same way the original decoder has been called. Example code from the open-source projects has been used as a starting point: The decoders are passed a block of compressed video data, which they process until an internal decoding cycle has been completed. Then the decoder returns control to the caller, awaiting further data blocks.

For MPEG-1/2, the libmpeg2 decoder library [20] in version 0.4.0 and for MPEG-4 part 2, the XviD library [21] in version 1.0.3 have been used. Both are available under the terms of the GPL [32], so they can be combined with VERNER without any licensing issues.

I wanted the prediction to be as independent of the actual decoder implementation as possible. However, there is one area where this could not be accomplished entirely: Both the predictor and the decoder have to stay in sync. That is, their stream parsers have to be in the same state. This requires a compatible handling of data blocks by both. XviD always expects to receive a full frame, which VERNER honors, so there is no problem here. But libmpeg2 can receive arbitrary chunks of data, which it will copy in an internal buffer. The predictor has to behave the same way. The problem here is that the frames can be scattered across multiple chunks and decoding will already start after the first chunk is being received. The predictor, however, can only estimate the total decoding time after it received the entire frame. This leads to situations where the decoder has already spent CPU time on decoding a good part of the frame before a prediction is available. Several solutions could be applied to solve this:

- A complete frame is accumulated before data is being passed to the decoder. This requires either an additional copying of the compressed data or modifications to libmpeg2.
- VERNER's demultiplexers are changed so that they always deliver complete frames

to libmpeg2. This requires the demultiplexers to know the concept of frames, which should better be encapsulated inside the decoder.

- The prediction is modified to work with a granularity smaller than frames, for example a slice-based prediction.

As this work targets the prediction itself, I did not implement any of these strategies, but I would favor the last one.

Based on the extracted metrics, the predictor calculates a predicted frame decoding time and stores the metrics internally. Learning decoding times is now possible by measuring the actual time consumed by a decoding cycle, which usually directly follows the metrics extraction and prediction to benefit from caching. This measured decoder cycle time is then appended to the LLSP matrix using `llsp_accumulate()`, together with the previously extracted metrics stored in the predictor. If an error has occurred during extraction, the predictor is invalid and learning will silently fail.

The evaluation can be made at any point in the learning process. It will finalize all the LLSP solvers internal to the predictor and store the resulting coefficients on file. Evaluation is non-destructive, so learning can continue afterward.

The discontinuation function resets the stripped-down decoder internal to the predictor, but it does not reset the accumulated matrices with the metrics and the measured decoding times. This function should be called between different video streams. It allows to learn prediction coefficients from more than one video, which is extremely helpful, because the influence of metrics like pixel resolution can only be calculated if it is not constant.

Implementing support for a new decoder algorithm would simply require writing these six functions for the new decoder.

5.3 Integration into VERNER

I extended the VERNER video player to accept three new command line arguments:

`--learn` to enable learning mode and pass the file name to store newly learned coefficients to.

`--predict` to enable prediction mode and pass the file name to load the coefficients for the prediction from

`--silent` to disable any output and synchronization to process the compressed video data as fast as possible

The file names are handed over via IPC to the vcore process, where the entire learning and prediction will take place. Although audio decoding time is not considered here, parts of the infrastructure have been implemented for audio decoding, too, to keep the audio and video parts of VERNER symmetric. The metrics extraction and prediction is done directly preceding the actual decoding. The exact same data block processed in the decoding is also used for the prediction. The wallclock time for the prediction and decoding are derived from the timestamp counter of the CPU. The decoder cycle time is used for learning, which directly follows the decoding, whereas the time the prediction took is interesting to verify the overhead introduced by the metrics extraction. We will see figures in Subsection 6.1. Optional RT_Mon sensors can give online plots of prediction error and overhead. Of course it is beneficial to use the time for which the thread was actually running instead of the wallclock time to take the scheduling out of the equation. Therefore, I used Fiasco's fast userspace thread time measurement to get more accurate execution times.

The evaluation of the predictor in learning mode is done on video boundaries. Everytime a video finishes playback and a new video starts, the current predictor is evaluated, which will store the learned coefficients. If the new video uses

a different decoder than the previous one, the current predictor is replaced with a new one, otherwise the current predictor is appended to. This limitation is because VERNER only stores one predictor for now. This could be extended in the future to have one predictor per decoder, but it is sufficient for our purposes, because it allows to use multiple files for learning.

Unlike `--silent`, all the new features are optional and can be disabled in the VERNER compile-time configuration. The silent mode is always available and adds a new dropping playmode to VERNER's synchronization component, which just throws away all decoded video frames instead of displaying them. For symmetry, decoded audio samples are also thrown away. For usability reasons, I did not want the video window to be displayed in silent mode, so I delayed the initialization of that window until it is first used. Because it is never used in silent mode, it will not appear. Again, for symmetry, the same has been done for the audio driver.

5.4 Frame Reordering

One problem left to explain is the discrepancy between the frame decoding time and the decoder cycle time. As you may have noticed, I was talking about decoder cycle times in this section, whereas previous sections dealt with frame decoding times. The problem is that one decoder cycle might not always decode exactly one frame. Dependencies amongst frames can make it necessary that a frame A requires the availability of the decoded frame B for reference. Although frame A is to be displayed before frame B, frame B has to be decoded before A. Frame B is called a forward reference. Because decoders output the frames in display order, this can lead to one decoder cycle decoding two frames, first B and then A in this example. The next decoder cycle can then either decode nothing and just flush out B or keep one reference in storage, outputting one frame per cycle from now on, if the decoder can make due with one forward reference.

This has different effects on the learning and prediction modes. In learning mode, a decoder cycle decoding more than one frame is not useful, because the cumulated time for the cycle cannot be separated into the individual per-frame decoding times needed for the LLSP solver. Modifying the LLSP solver to handle such cumulated times would be possible, but depending on the amount of frames per cycle that should be supported, this would at least double the amount of columns in the accumulated matrix, which in turn would increase the execution time of the LLSP finalization by factor $2^4 = 16$. With the decoder implementations used here, this situation occurs only once at the beginning of every video, so it is too rare to justify such a high price. I decided to simply ignore decoder cycles decoding more than one frame by invalidating the predictor, so the learning silently fails for this cycle.

In prediction mode, the predicted frame decoding times for the multiple frames can simply be added to estimate the decoder cycle time. The other case of a decoder cycle not decoding anything is not interesting at all, because the execution time for this step is negligible.

However, one decoder cycle outputting one frame is still the common case. Otherwise, the decoder would not be well-suited for real time applications, which would want to schedule the decoding of each frame individually.

6 Results

6.1 Prediction Accuracy and Overhead

I will now present the resulting decoding time predictions on real-life video material and also discuss the prediction overhead. Because MPEG-4 part 2 was the primary focus of my work, I will begin with that, but will also add remarks on MPEG-1/2.

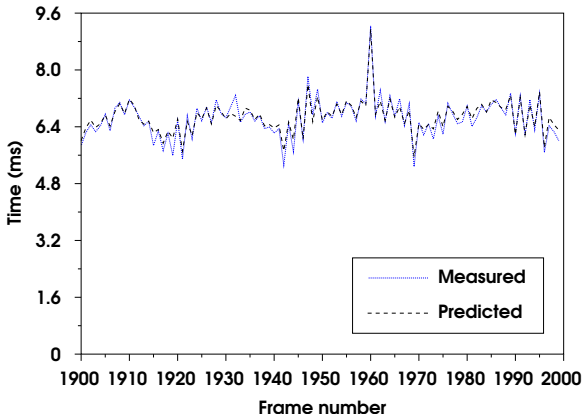

Figure 19: Prediction of stochastical load fluctuations for Voyager video.

6.1.1 MPEG-4 part 2

I used small snippets of four MPEG-4 part 2 videos to train the coefficients. The videos are the two IBM commercials and the two movie trailers from Table 1 on page 18. They differ in their resolution, which is necessary because otherwise the influence of the pixel count cannot be calculated properly. They also cover both the simple and advanced simple visual profiles of MPEG-4 part 2 and different data rates, so the feature combinations of the algorithm are sufficiently represented. I did not use the entire videos but always the first 500 frames of each. This ensures an equal length so that no video would outweigh the others. In addition to that, the videos for learning and for the actual prediction need to fit into memory to play them back in VERNER.

I played the four clips in VERNER's learning mode on a 700 MHz AMD Athlon machine to calculate the coefficients. In another VERNER instance, this time in prediction mode, I used the derived coefficients to predict the frame decoding times for the Voyager simple visual profile video. The results can be seen in Figures 20 to 23.

With relative errors [30] from -0.1050538 to 0.1978610 with an average of 0.0283729 at a standard deviation of 0.0443204 and absolute errors [31] from -0.746 ms to 0.909 ms with an average of 0.1546128 ms at a standard deviation of 0.2575455 ms, the prediction is quite accurate. In particular, I can predict not only the long-term behavior of the decoding time, but also the short-term fluctuations as can be seen enlarged in Figure 19. [1] uses the terms "structural load fluctuation" and "stochastical load fluctuation" for these long-term and short-term variations respectively. I believe, now that they can both be predicted, the term "stochastical" should be reconsidered.

I also wanted to test the prediction quality under different conditions, so I used small command line utilities to take the learning and predicting functionality out of VERNER. I used these tools on a PowerPC system (PowerBook G4 1.33 GHz) running Mac OS X 10.4, training the coefficients with the same set of short learning clips and then predicting the decoding times for the Amazon advanced simple profile high definition video. I think the results in Figures 24 to 27 are outstanding. Remember that this configuration is a completely different operating environment, with a different machine architecture, including different cache behavior and endianness, as well as a different compiler (gcc-4.0 as opposed to gcc-3.2 used for VERNER). Further, the coefficients have been derived from standard definition content and are now applied to a high definition video with about four times the frame size, but the prediction still works. The relative error ranges from -0.4312217 to 0.1645416 with an average of 0.0324660 at a standard deviation of 0.0448016. The absolute error spans -25.1316 ms to 17.533 ms with an average of 3.3054763 ms at a standard deviation of 4.6638761 ms. The two individual peaks in the histograms are most likely due to a slight nonlinearity between one or more of the metrics and the execution time. We have seen such nonlinear behavior in Subsection 3.3. Now, with the high resolution of the video, this nonlinearity is scaled up and is reflected in the long-term fluctuations of the absolute error.

It should be noted that systems with a more chaotic scheduling due to running background

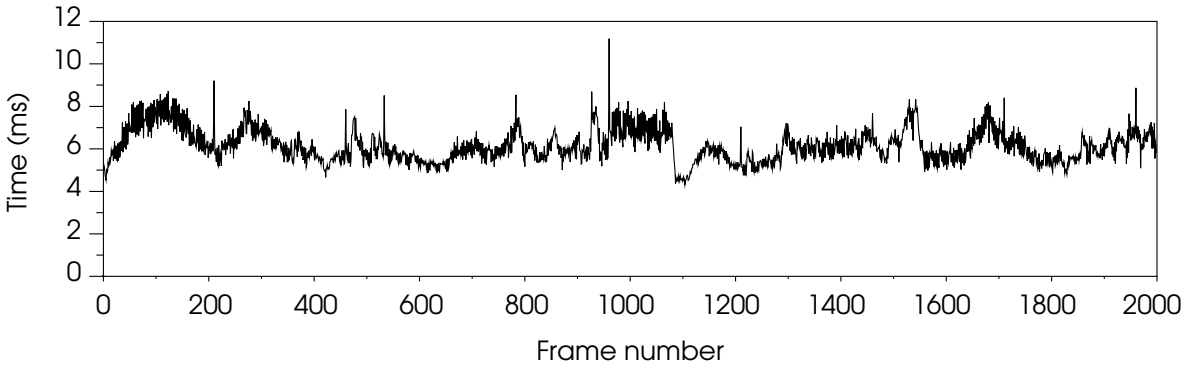

Figure 20: Measured frame decoding times for Voyager video.

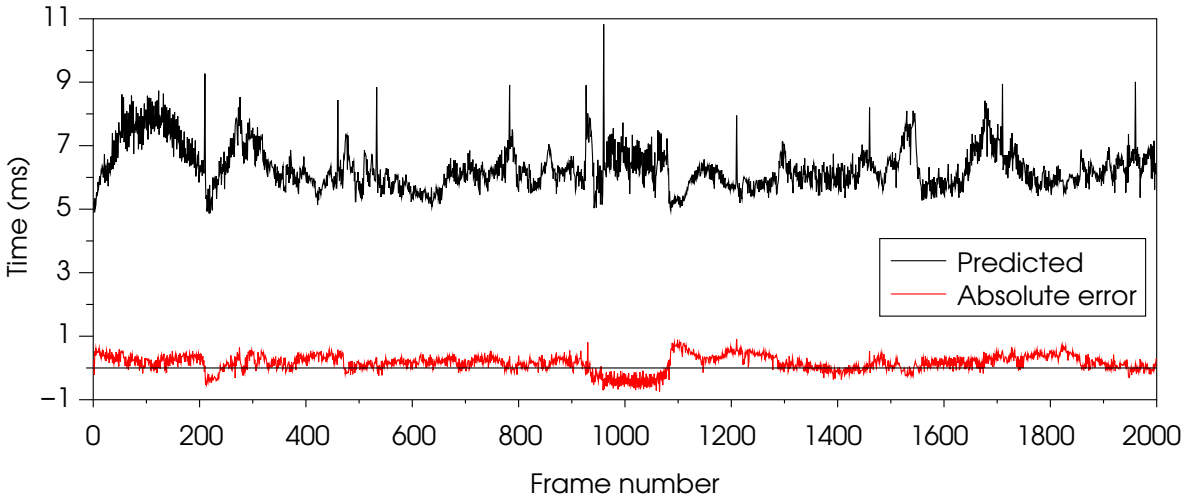

Figure 21: Predicted frame decoding times and absolute errors for Voyager video.

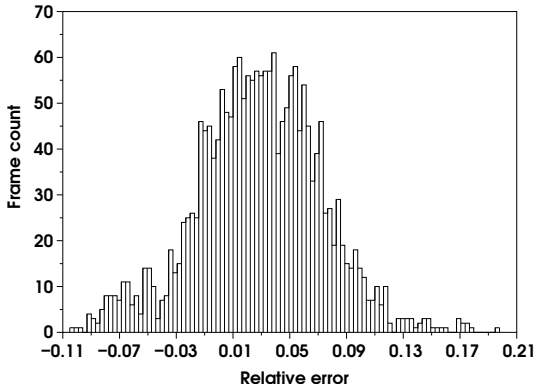

Figure 22: Histogram for the relative error of Voyager video decoding time prediction.

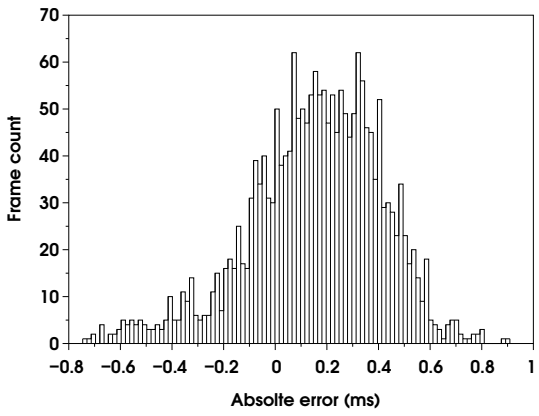

Figure 23: Histogram for the absolute error of Voyager video decoding time prediction.

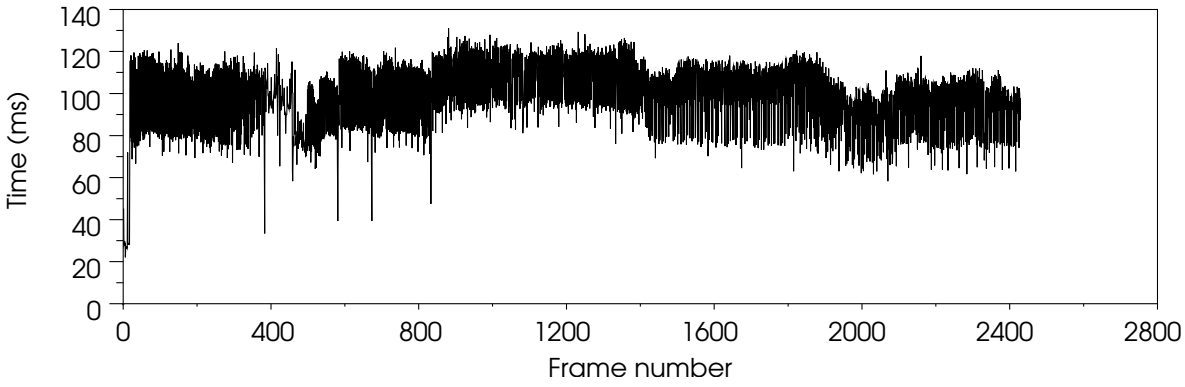

Figure 24: Measured frame decoding times for Amazon video.

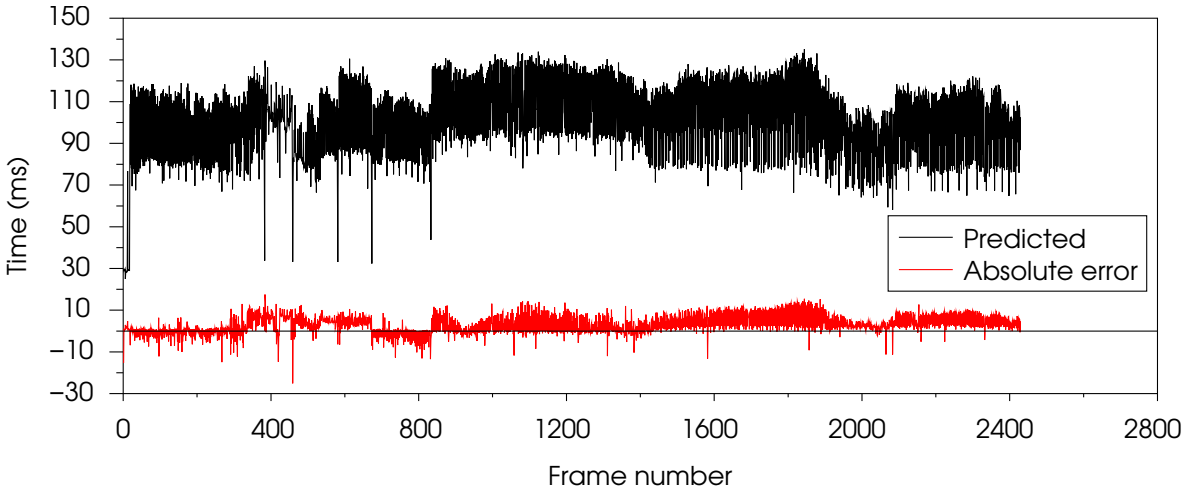

Figure 25: Predicted frame decoding times and absolute errors for Amazon video.

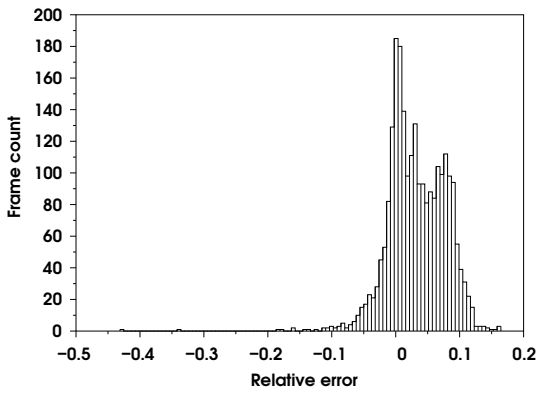

Figure 26: Histogram for the relative error of Amazon video decoding time prediction.

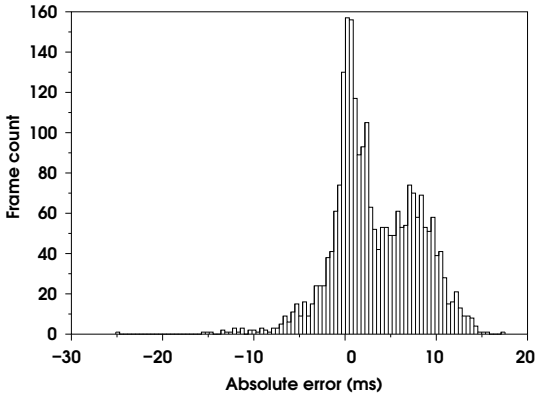

Figure 27: Histogram for the absolute error of Amazon video decoding time prediction.

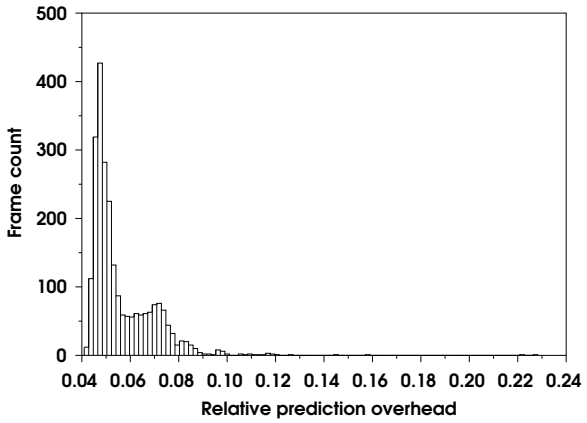

Figure 28: Histogram for the overhead of Amazon video decoding time prediction.

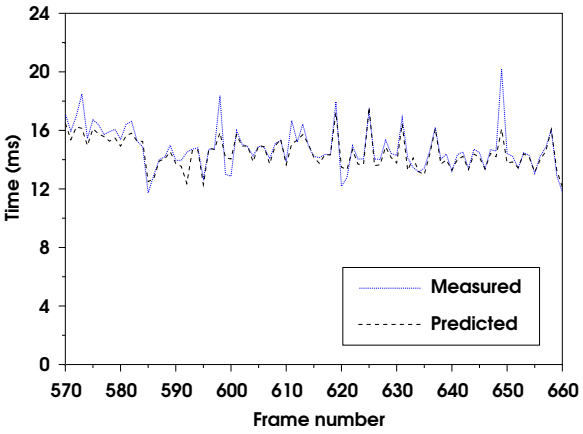

Figure 29: Prediction of stochastical load fluctuations for movie trailer.

processes are not suitable for learning or prediction, because the measured times will show scheduling artifacts when the predictor or decoder are preempted. This can be solved by either enhancing the operating system with realtime properties or using a timebase that does compensate for the time where the prediction process is interrupted.

The overhead introduced by the metrics extraction and the prediction can be seen in Figure 28. The average overhead is 5.6 %. The main source for the overhead is the bitstream parsing and macroblock decompression. As we have seen in Figure 6 on page 18, an additional bitstream parser would account for an average overhead of 16 %, so optimization by omitting unnecessary code has already been done. Unfortunately, further reduction of the overhead is difficult, because to extract metrics like macroblock counts, we have to look at each individual macroblock. However, skipping over single macroblocks is only possible by completely decompressing them. Because they are coded according to a variable length table, their size cannot be determined in advance. A possible approach is to split the decoder in half and parse and decompress the bitstream in the first half. The metrics can then be extracted from the preprocessed data and the second decoder phase would do the rest of the decoding, using the already decompressed macroblocks as an input. I wanted to avoid such constructions because they usually require heavy modifications to decoder code, so new decoder implementations would be difficult to deploy. Altenbernd, Burchard, and Stappert have taken this approach in [3], and they also ended up with overheads of 4-9 %, depending on the video, so there may not be much benefit in pursuing this.

Another way to reduce the overhead would be to not decode the motion vectors that we use to predict the temporal prediction step as discussed in Subsection 3.3.7 on page 20. This would lower the overhead from 5.6 % to 4.1 %, but also reduces the quality and transferability of the prediction coefficients.

6.1.2 MPEG-1/2

From the available MPEG videos in Table 2, I used the Sokolovski clip and the first 500 frames of the Antz scene to train the predictor. These two videos have different resolutions and cover both MPEG-1 and MPEG-2.The prediction for the MPEG-2 movie trailer in VERNER is shown in Figures 29 to 33. The average relative error is -0.0498824 at a standard deviation of 0.0426241, the average absolute error -0.7198970 ms at a standard deviation of 0.5429565 ms. I had to shorten the video to fit into the testing machine's memory.

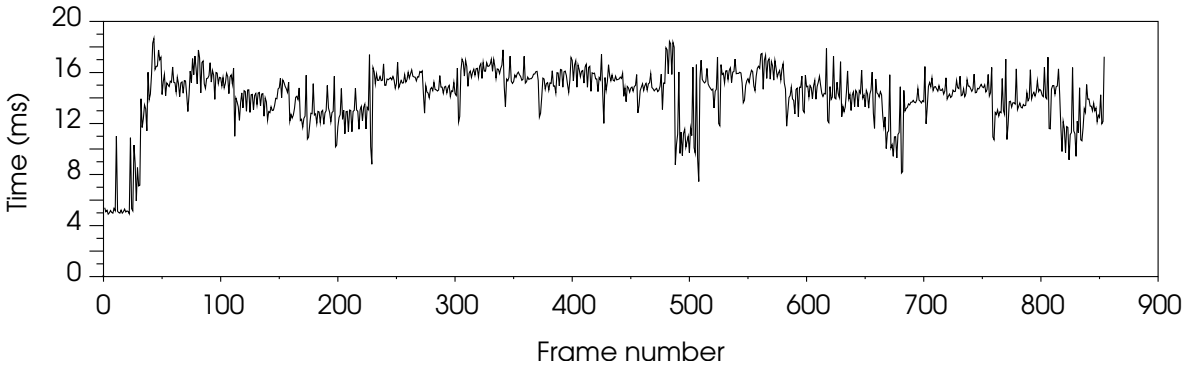

Figure 30: Measured frame decoding times for movie trailer.

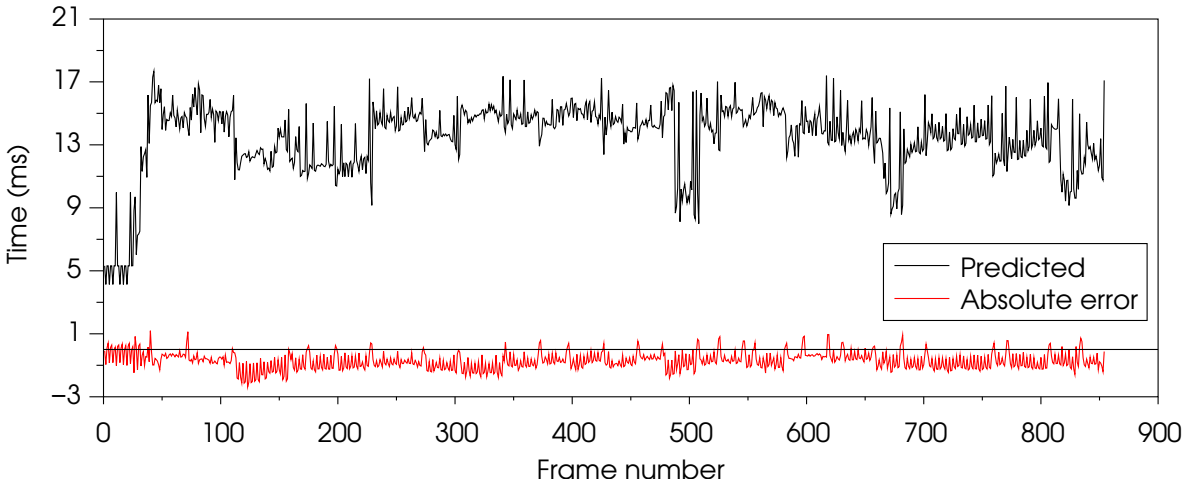

Figure 31: Predicted frame decoding times and absolute errors for movie trailer.

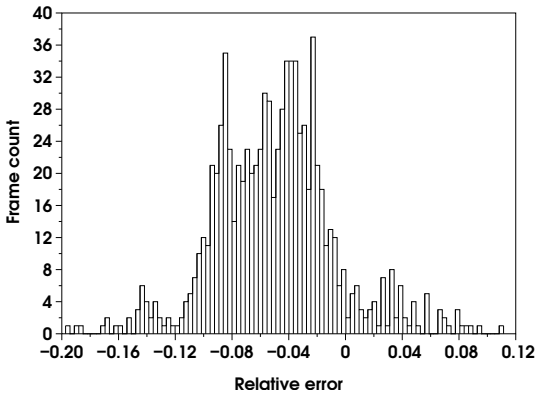

Figure 32: Histogram for the relative error of movie trailer decoding time prediction.

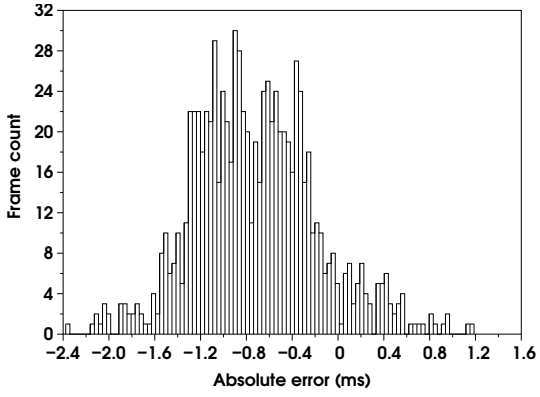

Figure 33: Histogram for the absolute error of movie trailer decoding time prediction.

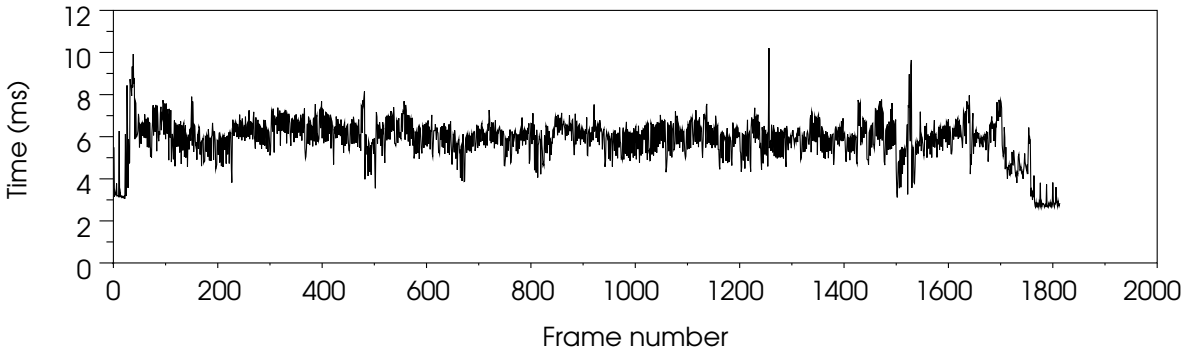

Figure 34: Measured frame decoding times for movie trailer.

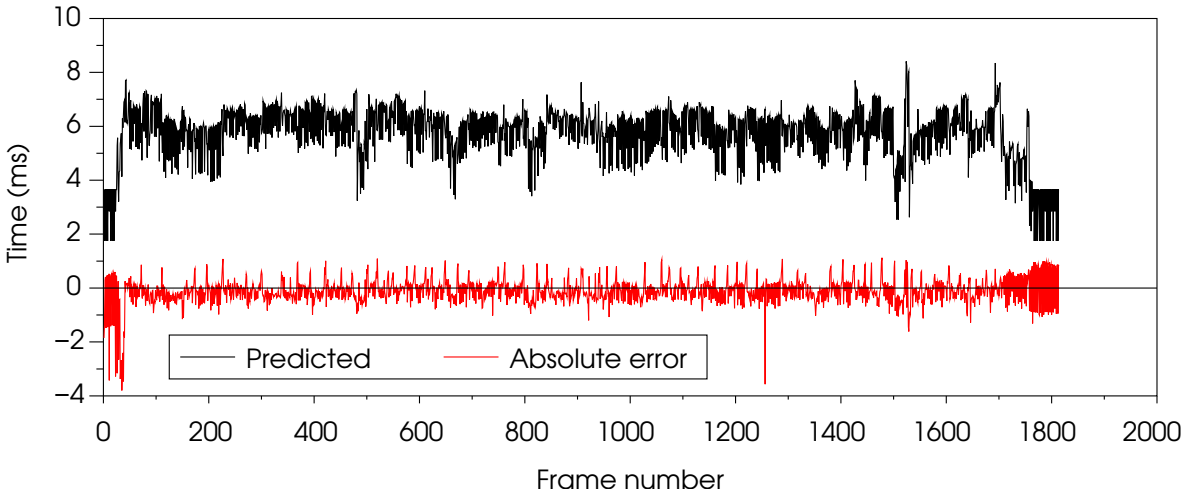

Figure 35: Predicted frame decoding times and absolute errors for movie trailer.

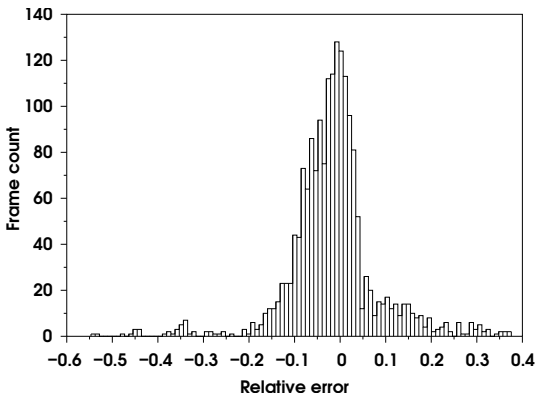

Figure 36: Histogram for the relative error of movie trailer decoding time prediction.

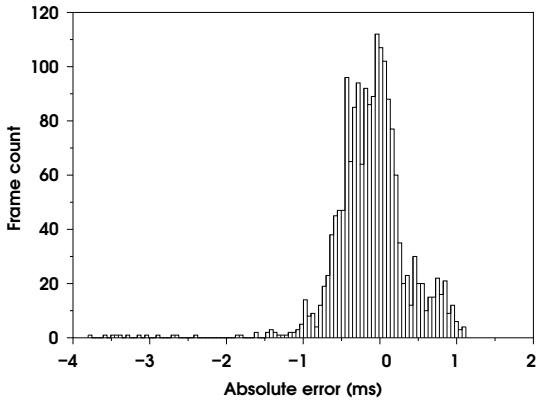

Figure 37: Histogram for the absolute error of movie trailer decoding time prediction.

video name	duration	resolution	size	codec	properties
Sokolovski	0:13 min	192×144	636 KB	MPEG-1	highly compressed thumbnail video
Antz	1:20 min	720×576	64 MB	MPEG-2	DVD movie credit scene, low in movement
movie trailer	1:13 min	720×576	43 MB	MPEG-2	dynamic, fast cuts, DVD quality material

Table 2: The MPEG-1/2 sample videos used to test the prediction.

Again, I cross-checked my method on the PowerPC CPU, also predicting the MPEG-2 movie trailer (Figures 34 to 37). The average relative error here is -0.0155866 at a standard deviation of 0.0985084, the average absolute error -0.1121871 ms at a standard deviation of 0.4853568 ms. Unfortunately, the average overhead introduced by the prediction is 18.6 %, but profiling of the metrics extraction code showed that 83 % of the execution time is spent on bitstream parsing and decompression, which is required for reasons presented previously. Motion vector decoding and glue code accounts for the rest. Not accounting for the IDCT shortcut (see Subsection 3.3.9 on page 22) would lower the overhead to 17.2 %, but also degrade the prediction quality. I can only explain the higher overhead with the lower complexity of MPEG-1/2 compared to MPEG-4 part 2, resulting in a higher portion of time used for parsing than for actual calculation.

Although small parts still could use improvement, I think the overall goal of accurately predicting decoding times for MPEG-1/2 and MPEG-4 part 2 in both the simple and advanced simple profiles has been accomplished. The prediction coefficients derived from one set of learning videos can be applied to a wide range of content, which is critical to future applications' usability, because it reduces the amount of learning necessary for good results.

6.2 Remaining Issues

There are several implementation details that could be improved upon:

- The prediction overhead could possibly be reduced by applying further optimization.
- VERNER's data delivery could be modified to not spread the data of one frame across multiple chunks.
- VERNER's user interface should provide better feedback on the learning mode.
- The on disk storage format of the coefficients is inefficient. A more sophisticated file format needs versioning and should be more dense.
- It should be possible to run the finalization stage of the LLSP solver in a background thread. The required data should be properly locked.
- The amount of metrics in use is currently limited.

Another problem is decoder specific and might become more pressing as more complex decoders appear: frame reordering. Decoder implementations where the frame decoding time and the decoder cycle time become even more independent of one another pose difficult problems. If the decoder does not return control after decoding a frame, the measuring of decoding times for individual frames and also the entire per-frame scheduling of a realtime application would require additional work or modifications to the decoder implementation. Current decoder implementations decode 0 to 2 frames per decoder cycle, because the algorithms examined here use one forward reference frame at maximum. Future decoders with heavy inter-frame dependencies will do even more frame reordering, so to output one frame, the decoder might need to decode an arbitrary number of frames. Decoder implementations should return

control after every decoded frame instead of every frame ready for display.

Future decoders' complexity itself can also be a problem. The MPEG-4 standard itself is huge, ranging from audio and video compression over container formats to interactive 3D scenery. The concepts are still separated into individual compression techniques, but future decoders might want to unify here. At some point, there could just be too much bitstream parsing required to obtain the metric values, increasing prediction overhead beyond sensible limits. At this point, it will be interesting to pre-determine the metrics already during encoding and embed them at prominent positions inside the bitstream. The MPEG-4 part 2 bitstream already contains this concept in a complexity estimation header, which stores information like macroblock and DCT coefficient counts. Unfortunately this header is optional and the common encoder implementations do not make use of it, so almost no publicly available MPEG-4 part 2 video stream includes this header. Future video bitstreams and container formats should consider this and assign header space for mandatory metrics storage.

6.3 Conclusion

Several areas are still open for future research. First of all, when H.264 has matured, it would be interesting to see, how it fits into the decoder model and prediction system presented here. Because H.264 is so complex, it will be difficult to cover all features the algorithm provides with appropriate metrics. Another challenge is to further automate the implementation of new algorithms like it has already been done with the column dropping. One could think of automatic derivation of metrics from profiling runs and function call frequency. This might even lead to results for codecs only available in binary form.

On the other hand, the prediction could be made more precise and it would be an interesting research subject to include source code analysis into my method to completely avoid any underestimation in the predicted decoding times.

Transferability is another area that could be improved upon. The coefficients calculated from learned decoding times on one architecture could be used on another architecture by compensating architectural features like processor speed and cache sizes. Even on the same architecture, further analysis of caching patters might improve the prediction, when coefficients learned from small videos are being applied to high definition content.

Another approach would be to redesign decoder implementations to return control after each decoded slice. I am confident that my prediction method can be adapted to predict execution times on a per-slice basis. If compression algorithms assign priorities to slices, a slice-based scheduling for the decoder is worth investigating. This would also solve the problem with decoding time prediction for libmpeg2.

Despite these issues, a system to predict decoding times with reasonable accuracy and acceptable overhead has been designed and implemented. The prediction relies on preprocessing and statistical evaluation of training runs rather than requiring heavy source code analysis or decoder modifications. This ensures that the presented results will not be obsoleted by decoder development such as code optimizations, because the method is independent of the specific decoder code. The integration into VERNER makes the software easy to use by allowing coefficient learning with a few mouseclicks. The prediction results can be examined as comprehensive RT_Mon online diagrams. When DROPS in general and VERNER in particular grow further, the usability of the current solution can be improved with background threads for the calculation work to not interrupt the user.

While the prediction of decoding times itself is already interesting, the results presented here should be regarded in a larger context. It is my firm belief that future operating systems will have realtime properties, because only such a

development can solve problems like guaranteed datarates to a DVD recorder or click-free audio playback cleanly and reliably. The schedulers of these operating systems will benefit from knowing resource usages in advance to make favorable scheduling decisions. My work can help to provide that and can be nicely complemented by research on perception models, which can help assign benefit values to video frames. This would allow video frames to compete for the CPU resource on the basis of a true price-performance ratio, resulting in optimal video presentation even in high load situations to really improve the user's experience.

References

[1] Clemens C. Wüst, Liesbeth Steffens, Reinder J. Bril, Wim F.J. Verhaegh: QoS Control Strategies for High-Quality Video Processing. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*.

[2] Damir Isović, Gerhard Fohler: Quality aware MPEG-2 Stream Adaptation in Resource Constrained Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*.

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

[4] Andy Bavier, Brady Montz, Larry L. Peterson: Predicting MPEG Execution Times. In *SIGMETRICS, 1998. Proceedings of the joint international conference on measurement and modeling of computer systems*.

[5] Claude-Joachim Hamann, Jork Loeser, Lars Reuther, Sebastian Schönberg, Jean Wolter, Hermann Härtig: Quality Assuring Scheduling – Deploying Stochastic Behavior to Improve Resource Utilization. In *Proceedings of the 22th IEEE Real-Time Systems Symposium (RTSS-XXII)*.

[6] ISO/IEC 11172-2: Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 2: Video

[7] ISO/IEC 13818-2: Generic coding of moving pictures and associated audio information – Part 2: Video

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

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

[10] TandbergTV diagram on MPEG-2 coding efficiency `http://www.m4if.org/public/documents/vault/m4-out-20027.pdf`, page 8

[11] MPEG-4 part 2 visual profiles `http://www.m4if.org/public/documents/vault/m4-out-30037.pdf`

[12] `http://www.uni-koeln.de/WRAKG/Berichte/Klassifikation/node203.html`

[13] Apple H.264 FAQ `http://www.apple.com/mpeg4/h264faq.html`

[14] Digital Video `http://de.wikipedia.org/wiki/Digital_Video`

[15] SVQ3 and WMV9 are related to H.264 `http://de.wikipedia.org/wiki/H.264#Verwandte_Verfahren`

[16] PNG compression filters `http://www.libpng.org/pub/png/pngintro.html#filters`

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

[18] Fiasco microkernel `http://os.inf.tu-dresden.de/fiasco/`

[19] Carsten Rietzschel: VERNER – ein Video EnkodeR uNd playER für DROPS http://os.inf.tu-dresden.de/papers_ps/rietzschel-diplom.pdf

[20] libmpeg2 project http://libmpeg2.sourceforge.net/

[21] XviD project http://www.xvid.org/

[22] David A. Huffman: A method for the construction of minimum redundancy codes. In *Proceedings of the IRE, 1952*.

[23] formula for the discrete cosine transform http://www.bretl.com/mpeghtml/DCxfrm.HTM

[24] J. Stör, R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, 1980. Subsection 4.8.1: Linear Least Squares. The Normal Equations

[25] J. Stör, R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, 1980. Subsection 4.8.2: The Use of Orthogonalization in Solving Linear Least-Squares Problems

[26] J. Stör, R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, 1980. Subsection 4.3: The Cholesky Decomposition

[27] J. Stör, R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, 1980. Subsection 4.8.3: The Conditioning of the Linear Least-Squares Problem

[28] J. Stör, R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, 1980. Subsection 4.7: Orthogonalization Techniques of Householder and Gram-Schmidt

[29] J. Stör, R. Bulirsch, Introduction to Numerical Analysis, Springer-Verlag, 1980. Subsection 4.9: Modification Techniques for Matrix Decompositions

[30] Relative Error http://mathworld.wolfram.com/RelativeError.html

[31] Absolute Error http://mathworld.wolfram.com/AbsoluteError.html

[32] GNU general public license http://www.gnu.org/licenses/gpl.txt