Video Quality and System Resources: Scheduling two Opponents

Michael Roitzsch, Martin Pohlack

Technische Universität Dresden,
Fakultät Informatik, 01062 Dresden

Abstract

In this article we present three key ideas which together form a flexible framework for maximizing *user-perceived quality* under given resources with modern video codecs (H.264). First, we present a method to predict resource usage for video decoding online. For this, we develop and discuss a video decoder model using key metadata from the video stream. Second, we explain a light-weight method for providing replacement content for a given region of a frame. We use this method for online adaptation. Third, we select a metric modeled after human image perception which we extend to quantify the consequences of available online adaptation decisions. Together, these three parts allow us, to the best of our knowledge for the first time, to maximize *user-perceived quality* in video playback under given resource constraints.

Key words: real-time, video decoding, adaptation, H.264, MPEG, prediction, decoding time prediction, visual quality, error propagation
PACS: 20.000, 20.090, 50.000, 50.010, 50.040, 50.060, 50.080

1 Introduction

Video decoding is a highly dynamic real-time problem [33]. Resource demand for a given stream can fluctuate in the order of magnitudes, with CPU time being a key resource. In the advent of high definition content and modern video codecs such as H.264 [32], resource demand does not only fluctuate greatly within a stream, but may also be very high in absolute numbers. To deal with

Email address: `[mroi,pohlack]@os.inf.tu-dresden.de` (Michael Roitzsch, Martin Pohlack).

this situation there are two principle approaches — heavy overprovisioning and adaptive decoders — which we discuss in the following.

Overprovisioning basically requires faster, more expensive hardware. It may not be suitable or possible in certain situations, such as mobile clients, where energy constraints are dominant. Also, better hardware may not yet be available (current desktop machines are barely able to play full resolution HDTV streams [13]). Furthermore, there is no easy way to define the upper bound for resource demands at the time of system design. The resource demand is highly content dependent. As a consequence, overprovisioning is a, potentially very expensive, approach that is not feasible in all situations.

An *adaptive* decoder design, on the other hand needs alternative working modes. These may, for example be: playback at a different frame rate or resolution. For some codecs it may also be possible to drop certain working steps online. The simplest form is an offline selection of appropriate content for a given platform. Such a selection does not result in optimal utilization of the platform and requires dedicated content encoding. Online adaptation is more complex and requires support in the decoder and stream format.

Currently, both approaches are applied in the real world depending on the situation. State of the art decoder implementations regard resource shortage as a rare special case and adaptation systems integrate a notion of quality only as an afterthought. A typical adaptation process for MPEG-1/2 was to skip decoding B-frames in overload situations [18], thereby dynamically reducing frame rate. This approach was feasible, because B-frames could not be used as reference pictures for future frames in MPEG-1/2. The error was therefore limited to the current frame. For the modern video codec H.264, this simple approach generally does not work anymore, as every frame type can be used as a reference. Consequently, dropping any frame could result in the loss of all frames until the next, potentially far away, instantaneous decoding refresh (IDR) frame.

With MPEG-4 pt. 2, adaptation can be done by scaling the quality of the optional post-processing step [20]. For H.264, however, the post-processing step is in-loop and thus mandatory. It can therefore not be used for adaptation.

Perspectively, online adaptation is the most promising approach, which we therefore explore for H.264. We aim at maximizing the *user-perceived quality* under given resources as the primary goal. In this article we present three key contributions which, when combined, achieve this goal.

(1) We present a method to predict resource usage for video decoding online. For this, we use a model describing decoder behavior given a small amount of metadata about a given frame's coded representation. We also describe how we obtain this metadata and how we constructed the model. Using

the model we can predict how much fully decoding a given frame *would cost*, and consequently how much resources can be saved by *not* doing so.

(2) We describe a light-weight method for providing replacement content for a given region of a frame. This method is used in online adaptation decisions.

(3) We selected an appropriate metric which is modeled after user-perception of video quality. Using this metric we can compute the grade of degradation by providing replacement content instead of fully decoding the original stream content. This also includes the degradation in further frames using the replacement content as a reference.

Combining these three contributions we can make sensible online decisions, maximizing user-perceived quality under given resource constraints.

This set of methods forms a flexible framework, which can be modified and adapted to future development in the area. Therefore, for each part of the framework, we outline the requirements that it has to fulfill for being usable in the whole system.

The remainder of this article is structured as follows: The next section describes our resource model and the prediction for decoding times. In Section 3, we continue by outlining our adaptivity approach and by discussing the visual error introduced by adaptation. Section 4 describes the interaction of the previously explained components, followed by Section 5, which evaluates our framework. Section 6 concludes the article.

2 Resource requirements

In order to make intelligent adaptation decisions at runtime, we do not only need knowledge about the current system state, that is, currently available resources, but we also need to know or need to estimate the consequences of our decisions beforehand. That may seem trivial at first but video decoding is a very complex process, where resource demand is highly dependent on the data actually processed at runtime. Therefore, static analysis alone will not solve the problem. Instead we use an approach which combines static and dynamic analysis. We developed a static decoder model derived from both an example implementation (FFmpeg [3]) and general knowledge from video standards [8,9,11,12]. We parameterize this static model with dynamic information from video streams.

Based on this model, we try to predict the consequences of future actions. These predictions should be as accurate as possible. A simple ordering of the set of possible actions according to the respective resource needs is not good

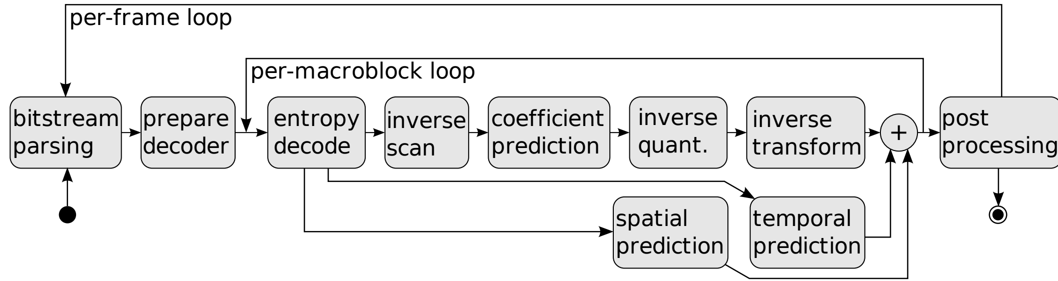


Figure 1. Generic decoder model.

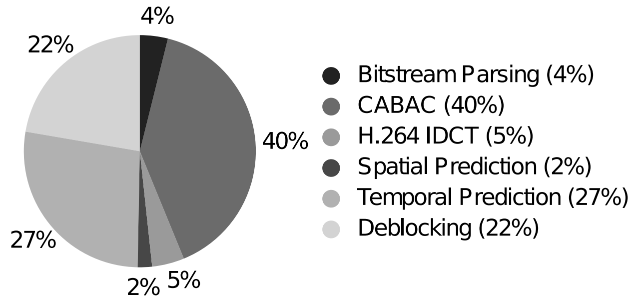


Figure 2. Execution time breakdown of H.264 decoding for BBC sequence (resolution 1280×720, Main Profile, for more details see Table 1 on page 32).

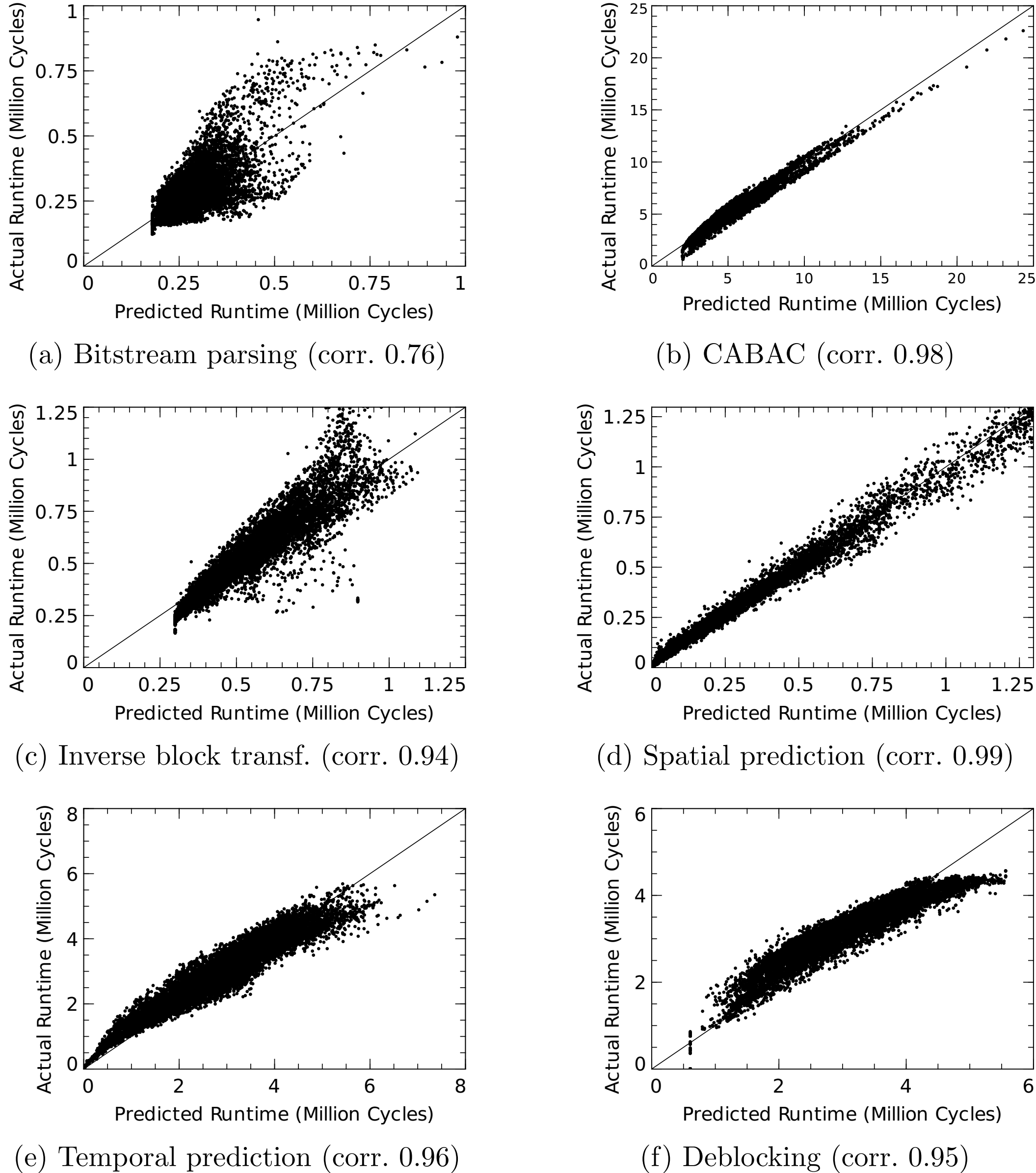
enough to make precise decision. Instead we need a metric with a quantitative notion of how much more resources will be used by a certain action in contrast to another.

In the following we present a decoder model for the current H.264 codec and discuss this model in detail. This model is derived from a generic model we described in [23]. The model consists of a chain of execution blocks, which process the compressed video stream. We describe, which features of the video stream can reasonably be used for predicting the resource demand. We call these features *metrics*.

In this article we will only discuss H.264 (MPEG-4 pt. 10), however, this same method can also be applied to other video compression standards, such as MPEG-1/2 and MPEG-4 pt. 2. In fact, we have done so already in [23].

2.1 Decoder model for H.264

Figure 1 shows a generic video decoder architecture which is powerful enough to describe MPEG-1/2, MPEG-4 pt. 2, H.264, and potentially others as well. In the following we discuss, how an actual decoder implementation maps each of those generic model steps to H.264 functional blocks. To judge the relative relevance of the blocks, a typical breakdown of H.264 total decoding time is shown in Figure 2.



(a) Bitstream parsing (corr. 0.76)
(b) CABAC (corr. 0.98)
(c) Inverse block transf. (corr. 0.94)
(d) Spatial prediction (corr. 0.99)
(e) Temporal prediction (corr. 0.96)
(f) Deblocking (corr. 0.95)

Figure 3. Execution time estimation for individual functional blocks (BBC sequence, see Table 1). The Pearson product-moment correlation coefficient is given for each fit.

We also describe which metrics from the bitstream correlate well with the execution times of the individual function blocks. Figure 3 demonstrates the precision of the correlation by plotting actual measurements of execution time spent within a function block over the respective prediction derived from a fit of the selected metrics. The following paragraphs explain the individual blocks in detail.

All time measurements were taken using the FFmpeg H.264 decoder [3] (version SVN-r6795) on a 2 GHz AMD Opteron machine. Other than the JM

reference decoder [6], FFmpeg heavily uses hand-tuned vector assembler code and is optimized for decoding speed. FFmpeg's timing behavior should therefore match that of practically used decoders better than JM. Furthermore, FFmpeg is virtually the only H.264 decoder available as open source.

Bitstream parsing The decoder reads in and prepares the bitstream of the upcoming frame and processes any header information available. Because each pixel is represented somehow in the bitstream and the parsing depends on the bitstream length, the candidate metrics here are the pixel and bit counts. Figure 3a shows a linear fit of both matches the execution time with a correlation of 0.76 (Pearson product-moment correlation coefficient). This is not particularly accurate, but as this step only accounts for 4 % of the total decoding time, we found it to be good enough.

Decoder preparation With H.264, the preparation part consists of precomputing symbol tables to speed up the upcoming entropy decoding. Its execution time is negligible, so we chose to subsume it under the bitstream parsing step above.

Entropy decoding This function block is the first that is executed inside a per-macroblock loop. A macroblock is typically a 16×16 pixel area of the target image whose compressed representation is stored consecutively in the data stream and that is decoded in one iteration of the loop. The data needed to further decode the macroblock is stored using a lossless entropy coding technology.

The execution time breakdown (see Figure 2) shows this entropy decoding step to be the most expensive. This sets H.264 apart from other coding technologies like MPEG-4 Part 2, where the temporal prediction step was by far the most expensive [23]. The reason for this shift is that the H.264 Main Profile uses a new binary arithmetic coding (CABAC [10]), that is much harder to compute than the previous Huffman-like schemes. A less-expensive variable-length compression (CAVLC) is also available in H.264 and is used in the Baseline and Extended Profiles, where CABAC is not allowed. Both methods decode the data for the individual macroblocks. Using the same rationale as for the preceding bitstream parsing, a linear fit of pixel and bit counts predicts the execution time well. We restrict ourselves to CABAC with results shown in Figure 3b. As this step accounts for a large share (40 %) of total execution time, it is fortunate that the match is tight with a correlation of 0.98.

Inverse scan The decompressed macroblock we received from the previous stage is a one dimensional list of bytes. Those need to be rearranged as 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 diagonal pattern. H.264 decoders typically incorporate this step into the entropy decoding step above by storing the entropy decoded coefficients according to a scan pattern. Execution time of this step is thus already accounted for.

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

Inverse quantization The macroblock-coefficient quantization is reversed before decoding proceeds by multiplying with an inverse quantization matrix. As this step's individual execution time is negligible and it is tightly coupled with the upcoming block transform, we combined the two in our analysis.

Inverse block transform This decoder step transforms the macroblock matrix from the frequency domain to the spatial domain. The resulting spatial matrix corresponds to a portion of the final image and has the same dimensions as the macroblock matrix. H.264 knows two different transform block sizes of 4×4 or 8×8 pixels, which can even be applied hierarchically. Therefore, we account how often each block size is transformed and use a linear fit of these two counts to predict the execution time. Figure 3c shows the resulting correlation of 0.94. The remaining deviations are most likely caused by optimized versions of the block transform function for blocks, where only the DC coefficient is nonzero. But given the small percentage of total execution time this step contributes (5 %, see Figure 2), we refrained from trying to improve this prediction any further.

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 predict the part of the image being covered by the currently decoded macroblock. This step can potentially be executed at the same time as the inverse block transform, but we will not pursue this parallelism, because the commonly available decoder implementations perform this work in a single thread.

Spatial prediction extrapolates image data from the same frame with various patterns into the target area of the current macroblock. This prediction can use block sizes of 4×4, 8×8, or 16×16 pixels, so we account those prediction

sizes separately. A linear fit of those counts correlates well with the execution time (see Figure 3d, correlation 0.99).

Temporal prediction This step was the hardest to find a successful set of metrics for, because it is exceptionally diverse. Not only can motion compensation be used with square and rectangular blocks of different sizes, each block can also be predicted by a motion vector of full, half or quarter pixel accuracy. In addition to that, bi-predicted macroblocks use two motion vectors for each block and can apply arbitrary weighting factors to each contribution. However, motion compensation is essentially a way to copy image data from a reference frame. Thus, it is mostly memory bound.

In [23], we therefore broke this problem down for MPEG-4 Part 2 to counting the number of memory accesses required. A similar approach was used here: by consulting the H.264 standard [12] we came up with motion cost values, depending on the pixel interpolation level (full, half or quarter pixel, independently for both x- and y-direction). These values are essentially memory access counts, but we empirically took optimizations of typical decoder code like cache reuse into account.

These cost values are then accounted separately for the different block sizes of 4×4, 8×8, or 16×16 pixels. The possible rectangular block sizes of 4×8, 8×4, 8×16, or 16×8 are treated as two adjacent square blocks. Bidirectional prediction is treated as two separate motion operations. The resulting fit with a correlation of 0.96 can be seen in Figure 3e. This correlation is acceptable for a 27 % share of total decoding time.

Merging $\oplus$ The merging of the results of prediction and inverse transform ends the per-macroblock loop. Execution will continue with the entropy decoding of the next macroblock.

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. For H.264, post-processing is comprised of an edge deblocking filter. The deblocking is performed adaptively based on the calculation of a boundary filtering strength. This strength is calculated for every macroblock. Its edges are then deblocked conditionally according to a strength threshold. A correlation of 0.95 is achieved with a linear fit of pixel count and the number of edges being deblocked (see Figure 3f).

Metrics summary The metrics selected for execution time prediction are:

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

In Figure 3, we have shown prediction accuracy based on linear combinations of metrics selected specifically for single function blocks. The prediction of the entire decoding time will be more accurate than the sum of the individual predictions, because *all* metrics contribute to *all* steps of the prediction.

2.2 Numerical background

Now that we have determined a set of q metric values required for each frame of the video, we first describe how we process them by solving a linear least square problem. Following that, we explain how we actually obtain the metrics using a stripped down decoder.

We choose a linear model for two reasons: *first*, source-code inspection for FFmpeg and video coding standards suggest such a dependency already for single metrics. *Second*, our experimental validation shows both a good prediction for real experiments and a good correlation for single functional blocks of the model (see Section 2.1).

In a learning stage, on which we will present details in Section 2.2.2 we will receive a metric vector $\underline{m}_i$ and the measured frame decoding time t_i for each of a total p frames ($i = 1, \ldots, p$). Accumulating all the metric vectors as rows of a metric matrix M and collecting the frame decoding times in a column vector $\underline{t}$, we 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 $\underline{m}\,\underline{x}$. Because the prediction coefficients $\underline{x}$ must be derived from M and $\underline{t}$ alone, we 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 measured frame decoding times $\underline{t}$ is minimized. The error is expressed by the square of the Euclidean norm of the difference-vector. Because of its insensi-

tivity against badly conditioned matrices M, we chose QR decomposition with Householder's transformation as the method to solve the LLSP. For a more detailed explanation of the involved mathematics, please refer to the literature such as [26,21].

2.2.1 Metric selection and refinement

For the general problem of metric finding we see two approaches: **(a)** First, a domain expert has to model the problem using smaller sub-steps. Then, by looking at the work done in the sub-steps, he has to guess interesting metrics which can be easily obtained from the data to be processed and which correlate with the work done in the sub-steps. These selected metrics are then verified with obtained resource usage statistics of the original problem. **(b)** Second, for more simple problems, one could get useful results without splitting up the original problem into smaller pieces and without a domain expert selecting metrics. One could just use *all* easily available metrics and try to find the relevant metrics by validating them against measured data. In both cases only those metrics are relevant for our approach which can be obtained with much less resource usage than solving the original problem.

For this article we took the first approach, as the domain is highly complex and a lot of different metrics are available. For both approaches an automatic method for metric validation is required, which we describe in the following.

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 is to make the results transferable to other videos, which might not always work when using metrics too greedily. Using too many metrics can lead to overfitting to the training material, leading to bad predictions for videos not included in the training set. A common artifact of this is negative coefficients, which make little sense in the decoder model we presented. The main cause for this is similarities of columns with linear combinations of other columns. The special case of this situation is an actual linear dependency, resulting in a rank-deficient matrix. This leads to instabilities 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 optimal fit, which might be too specific to predict other video's decoding times with the resulting coefficients. To overcome this problem, we drop metrics before solving the LLSP, deliberately making the fit less good for the training set, but more transferable to other videos outside the training set.

In the resulting R matrix of a QR decomposition, the remaining error, called residual sum of squares, for an n-column matrix is the square of the value in the nth column of the nth row. 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 goals, but results in the smallest increase of this error indicator. 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 sufficiently approximated as a linear combination of others. We propose an algorithm to eliminate such situations in [21].

2.2.2 LLSP solver

The LLSP (linear least square problem) solver and the collector support two phases of operation:

- Learning mode, in which the collector accumulates metrics and a timed and unmodified decoding step delivers real frame decoding times.
- Prediction mode, in which previously obtained LLSP coefficients are multiplied with online-collected metrics to predict frame decoding times.

During learning mode, the solver collects metric values in a matrix. If the data accumulation is finished, the coefficient vector $\underline{x}$ is calculated with an enhanced QR decomposition that we discuss in the next section. This step has a complexity of $O(pq^4)$, of which the normal QR decomposition accounts for $O(pq^2)$ and the iterative column dropping accounts for another $O(q^2)$ factor (see [21] for details). q is typically fixed and small, compared to p being unbound. Therefore, the video length has linear impact which is what one would hope for. The resulting coefficients are then stored for use in prediction mode, typically on videos other than those in the learning set.

2.2.3 Metrics extraction

In [23] we explained the metric extraction procedure for MPEG-1/2/4. In contrast to previous coding standards, the CABAC entropy decoding step dominates the total decoding time, so extraction of metrics other than compressed frame size is too expensive to do online. Instead, we extract the relevant metrics offline in a preprocessing step and embed them into the bitstream, which constitutes a size overhead of 32 Bytes per frame without any compression. This accounts for a negligible 0.2 % for a typical 4 MBit/s stream or an acceptable 6.2 % for a 100 kBit/s stream, which could be reduced significantly with a domain-specific compression.

Our approach to predicting resource requirements ahead of time combines the separation of the problem according to a decoder model with a training phase to *empirically* link the model to the actual execution environment. Various aspects of this idea have been explored in earlier work.

Szu-Wei Lee and C.-C. Jay Kuo modeled the complexity of the H.264 motion compensation step in [19]. The general approach of predicting decoding time with a linear combination of metrics extracted from the bitstream is similar. The weight coefficients for the metrics are determined using training in much the same way. But while Lee and Kuo specialize on the motion compensation step only, we extend this to cover the entire H.264 decoding process. The metrics chosen by Lee and Kuo for motion compensation are the counts of x- and y-direction interpolations, the number of motion vectors and an estimated number of cache misses. We account for motion compensation complexity primarily by block size, so integrating the proposed notion of cache behavior into our approach is an interesting direction for future improvement.

Another model specifically for the motion compensation process is presented by Yong Wang and Shih-Fu Chang in [29]. The paper explains a motion vector cost function based on subpixel interpolation complexity, which is similar to our motion cost. Wang and Chang utilize the complexity model in the *encoder* to create bitstreams with reduced decoding complexity. Their goal is static reduction of decoding effort, rather than dynamic graceful adaptation in overload situations.

An approach that addresses not only motion compensation, but the entire H.264 decoding process is presented by Horowitz et al. in [17]. The paper presents an execution time estimation for the H.264 baseline profile. It is based on a decoder model and breaks the decoding down into function blocks, similar to our approach. They also consider the different block sizes for metrics. Once the candidate metrics have been chosen, we then continue empirically by utilizing training and linear fitting. Horowitz et al. continue by translating the computational requirements of the standard into typical arithmetic operations of the target CPU. Considering superscalar execution, they derive execution times from the computational throughput of the CPU. This has the advantage of not requiring any training, but because loop overhead, flow control, memory latencies, and pipeline stalls are ignored, the estimated times are factor 2–6 below the real values. In contrast, our approach combines the decoder model idea with training to more accurately capture the behavior of real decoder code on real CPUs.

Training is also employed by van der Schaar and Andreopoulos in [27]. They

break decoding down using a generic reference machine that supports assign, add, and multiply operations. The execution time on real hardware is estimated by using the operation counts of the reference machine as metrics. This reference machine thus abstracts from the real hardware. However, van der Schaar and Andreopoulos do not consider H.264, but a custom codec. Schaar et al. do not focus on execution time prediction on real hardware. Their evaluation of prediction accuracy is not definitive. Applying the reference machine approach to our decoder model and training approach could help exploring, how weight coefficients derived on one machine can be applied to a different architecture.

Reviewing this related work, the idea of using metrics and training is common, but the abstraction level on which the modeling is complemented by training is different. Building on the previous results, we believe to have found a balance that enables both accurate results by training against real decoder implementations and transferability with a model that is independent of the hardware and the decoder implementation.

3 Quality

We have now covered the resource consumption of video decoding with a model of decoder execution times and an architecture to predict it at runtime. Resource consumption being the machine's view on the problem, we can now turn around and look at video playback from a user's perspective. This means we have to deal with visual quality under potentially constrained resources.

Current players typically react to insufficient CPU time with frame drops. This may have been acceptable with decoders prior to the H.264 standard, because the B-frames of MPEG-1/2/4 can be dropped without degrading visual quality for any frame other than the one being skipped. In fact, this approach has been proposed in the literature [18]. Losing one frame in a high-motion sequence can still be perceived as visually disrupting, but decoding can then continue normally. With H.264, however, every frame, including B-frames, can be a reference frame and might therefore be required to correctly decode future frames. This means that skipping one frame can prevent or at least degrade the decoding of all future frames until the next IDR frame resets the decoder. Such resets can be in the order of seconds apart from one another.

Another strategy to cope with insufficient resources is to briefly stall playback to recover. However, to keep audio and video synchronized, the audio has to be stopped as well, which is extremely irritating to the user. Watching a video with intermittent audio gaps can be very frustrating. This effect can be seen with web video, which can stall due to limited bandwidth.

The fundamental problem with these approaches is their underdeveloped quality-awareness, which leads to a heavily degraded user experience. Therefore, we set out to develop a way of dealing with insufficient CPU time more gracefully. Currently, the playback process regards resource shortage as a rare special case and adaptation systems integrate a notion of quality only as an afterthought. We strive to treat user-perceived quality as a first class priority and resource limitations as the common case.

3.1 The H.264 scalable extension

H.264's scalable video coding (SVC) [25] promises to be an excellent technology for implementing a fine-grained balancing algorithm between perceived visual quality and decoder resource needs. Unfortunately, the standard for this extension has not yet been ratified and as a consequence, no mature decoders or encoder toolchains are available. We are planning to look into H.264 SVC once it is ready, but to explore our ideas now, we needed a different base technology. Therefore, we decided to develop our own scalable decoding system. It provides only two decoding levels: full decoding with full resource consumption and fast fallback decoding. It will be described in full detail in the following sections. Our entire architecture, however, is modular enough to incorporate H.264 SVC with considerable reuse of research results presented here once H.264 SVC is mature. We will comment on this in Section 4.

3.2 Fallback decoding and quality

In developing our own H.264 fast fallback decoding mode to trade visual quality for decoding time, we have to answer three questions:

(1) How is the content of the fallback frame created?
(2) What is the impact on the visual quality for that frame? That is, how much does the fallback content differ visually from the original?
(3) To what degree will the quality degradation be carried over to future frames due to the degraded frame being used as a reference? How does this effect accumulate if multiple subsequent frames will be fallback-decoded?

The following sections will answer those questions.

3.3 Fallback content

Instead of dropping a frame when low on CPU time, we want to fabricate a fallback frame with replacement content. Because we want to do this as fast as possible, we have to avoid executing the expensive parts of the H.264 decoding process. Looking back at Figure 2, we can see that the CABAC step takes up a large portion of the total decoding time per frame. Therefore, avoiding this step is key to conserve CPU time. However, this means that all data of the current frame will remain in its compressed state and hence will not be directly available for creating the fallback content.

The next interesting pool of information potentially useful for crafting a replacement frame is the buffer of reference frames in the decoder. These previously decoded frames kept in memory by the decoder provide image content temporally close to the content we want to replace. Our idea is to fabricate a fallback frame by reusing portions of those previously decoded frames. This idea is especially adequate for H.264, which, with its buffer of multiple reference frames, offers a wide choice of candidate replacement regions to choose from.

Again, because we want the fallback to be fast, the image data from the reference frames should simply be copied into the replacement frame. However, copying does usually not take place from the same location of a different frame, but from different regions of different frames, leading to higher quality of the fallback because motion between the two frames can be compensated for. While motion analysis of a series of images is generally expensive, the H.264 coded video stream already provides motion vectors of good quality. But direct access to these vectors is only possible after performing CABAC decoding, which we want to avoid. Therefore, the coded bitstream of each video frame should be supplemented *offline* with another representation of the frame's motion relative to the reference frames.

However, simply extracting and redundantly storing all motion vectors would increase the bitstream size unacceptably. Therefore, we will present a more lightweight representation of the motion vectors in the next section.

3.4 Quadtree encoding

To encode the frame's motion efficiently, we use a quadtree [15] to partition the data. Starting with the root node representing the complete frame, we recursively and adaptively subdivide each node's image region into four subregions. This leads to a nonuniform subdivision of the frame, with each node having either zero or four subnodes. An example of a possible quadtree subdivision

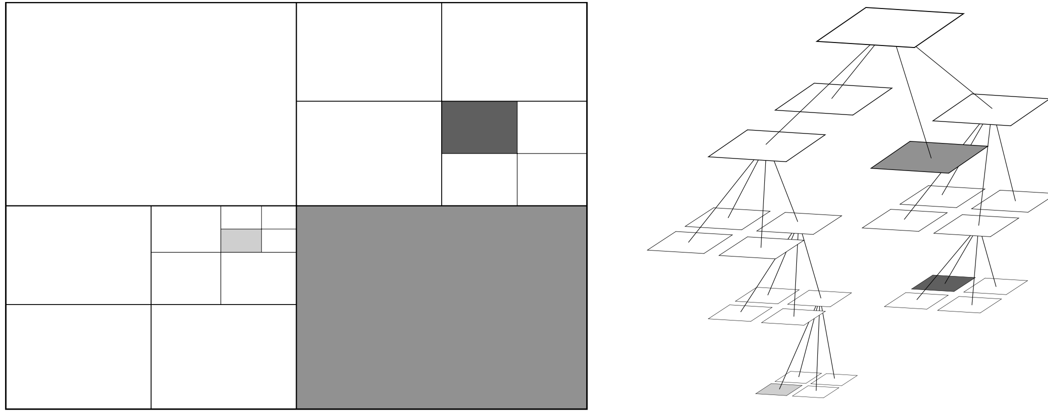


Figure 4. Quadtree subdivision example.

is given in Figure 4.

3.4.1 Bitstream assumptions

Instead of doing our own offline motion analysis, which would replicate a lot of the work already done by the H.264 encoder, our approach reuses the motion vectors already present in the bitstream. We rely on several assumptions on bitstream behavior that enable us to use these vectors:

- Areas of related motion are spatially contiguous.
- In an area of related motion, the bitstream selects the most similar reference frame.
- In an area of related motion, the motion vectors do not jump erratically, but neighboring vectors are similar in direction and length.

These assumptions are justified, because a sensible H.264 encoder tries to minimize the size of the bitstream. The coding features in H.264 have been designed such that motion vectors following a fluent pattern can be encoded with fewer bits. Therefore, the encoder will automatically prefer bitstreams encoded in favor to our assumptions.

3.4.2 Encoding algorithm

The quadtree is created as side information to already encoded macroblocks. The actual H.264 encoding process is left unchanged, our quadtree algorithm operates by processing the encoded H.264 bitstream. The following two-part algorithm associates motion vectors with quadtree nodes. The separation in two parts is purely to simplify the explanation, the overall algorithm is comprised of a serial execution of both parts. The running time of the algorithm is in the order of magnitude of an H.264 encoder run.

Algorithm 1 Fully subdividing the quadtree

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

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

    // Step 3
    averageVector = <0, 0>;
    foreach (macroblock in quadtreeNode)
        if (macroblock.refFrame == mostOftenUsedRefFrame)
            averageVector += macroblock.vector;
    quadtreeNode.vector = averageVector;

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

The first part recursively creates a fully subdivided quadtree. A pseudo-code description can be found in Algorithm 1, a textual description follows:

(1) Start the iteration with the root node of the quadtree covering the entire frame.
(2) For the region covered by the current node, determine the reference frame used most often by motion vectors. Store this reference in the current node.
(3) For the region covered by the current node, determine the average motion vector across all motion vectors using the reference frame determined in step 2. Round this vector to full pixels and store it in the current node.
(4) Subdivide the current node's region into four subregions, creating four subnodes of the current node. If the areas covered by the subnodes are each at least the size of one macroblock and contain each at least one motion vector, repeat steps 2–4 for each subnode, otherwise delete the subnodes and return.

This yields a fully subdivided quadtree with a hierarchy of reference frames and motion vectors. The algorithm continues by adaptively pruning the quadtree from the leaves towards the root node, trading quality for stream size with a quality threshold. A pseudo-code version is available in Algorithm 2 given below.

(1) Start the recursion with the root node.
(2) Return to the parent node, if the current node has no subnodes.
(3) If the current node has subnodes, recurse to prune them first.
(4) Return to the parent node, if any of the current node's subnodes is not a leaf. This ensures that cutting is not performed here if it failed on one of the subnodes.
(5) Fabricate a complete fallback frame by iterating over all leaves of the quadtree. The region covered by each leaf node is filled with an equally sized region designated by the reference frame and motion vector stored in the leaf node.
(6) Calculate the quality loss between the fully decoded frame and the fallback frame.
(7) Remove all subnodes of the current node, so the current node becomes a leaf and fabricate the fallback frame again as described in step 5. This time, the fallback frame is determined by the coarser motion representation due to the coarser subdivision of the quadtree.
(8) Calculate the quality loss between the fully decoded frame and the fallback again. How we quantify quality loss is discussed below.
(9) The coarser subdivision is expected to lead to a higher quality loss. If the resulting decrease in quality is below a certain threshold, the subnodes removed in step 7 are discarded, otherwise they are reattached. In both cases, control flow returns.

The algorithm results in a non-uniformly subdivided quadtree that approximates the motion in the frame.

The calculation of quality loss is performed using a metric we will discuss in Section 3.5. The accepted loss in step 9 provides a way to balance the size of the quadtree against the accuracy of the motion representation. More elaborate thresholds like a ratio of quality to encoded quadtree size are possible, but we did not further pursue this.

The algorithm prunes the tree in bottom-up order. We also tried a top-down approach, which turned out to be inferior in the achieved quality. The reason is that very coarse subdivisions, where nodes cover large areas, have a reverse quality behavior: The quality loss with one additional subdivision step may be higher than without, because of edges introduced by the division in the frame's interior. These edges disrupt the image structure, resulting in the observed effect on quality. While this situation basically occurs recursively

Algorithm 2 Adaptive pruning of the quadtree

```
// Step 1
rootNode = originalFrame;
pruneQuadtreeNode(rootNode);

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

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

   // Step 5
   fabricateFallback(rootNode, fallbackFrame = emptyFrame());
   // Step 6
   qualityLoss1 = compare(fallbackFrame, originalFrame);

   // Step 7
   temp = quadtreeNode.subnodes;
   quadtreeNode.subnodes = null;
   fabricateFallback(rootNode, fallbackFrame = emptyFrame());
   // Step 8
   qualityLoss2 = compare(fallbackFrame, originalFrame);

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

function fabricateFallback(quadtreeNode, fallbackFrame) {
   if (quadtreeNode.subnodes)
      foreach (subnode in quadtreeNode.subnodes)
         fabricateFallback(subnode, fallbackFrame);
   else
      fallbackFrame[quadtreeNode.area] =
        quadtreeNode.reference[quadtreeNode.area.coords +
        quadtreeNode.vector];
}
```

with every additional subdivision, the quality increase as the motion vectors become more fine grained seems to overcompensate for the negative effects of

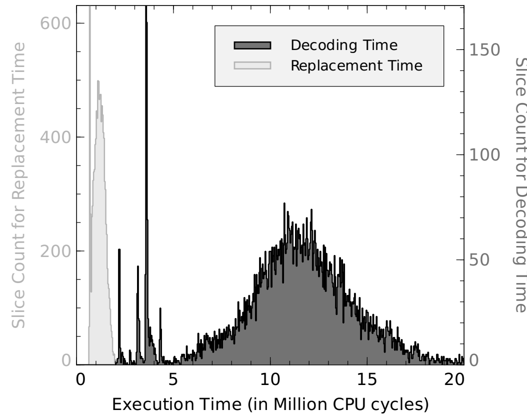


Figure 5. Decoding time and replacement (fallback) time histograms, measured over BBC video (see Table 1)

those edges.

3.4.3 Using and storing the quadtree

When short on resources during decoding, the fallback can be used to gain some CPU time. As illustrated in Figure 5, the fallback is in average about 9.9 times faster than full decoding. To execute the fallback, the leaves of the quadtree are required. They cover the entire frame and provide a reference frame index and a motion vector for each region. The corresponding image data pointed to by the vector is copied from the reference frame into the fallback frame. Additional decoder-internal metadata of the fallback frame like the map of macroblock type information is synthesized as well by filling with neutral values, because H.264 uses such data for prediction when decoding subsequent frames.

To do all that, fast access to the leaves of the reference tree is required. Therefore, the quadtree is created offline by a preprocessor, linearized and its leaves are stored directly in the H.264 bitstream as one custom NALU (network abstraction layer unit) per frame. Since each NALU is prefixed with a start code not otherwise appearing in the bitstream, NALU boundaries are easy to find. A decoder can therefore skip over the coded representation of a frame and use the quadtree data without spending any time on CABAC decoding. As the next frame will start at a NALU boundary, continuing regular decoding with the next frame is equally easy. By using custom NALUs, our supplemented stream can also be played back by standard compliant players that simply ignore our data.

3.5 Quality loss

As seen in the previous section, a key building block is the quantification of quality degradation as perceived by the user. Not only is this needed in the algorithm presented earlier to prune the quadtree, but it is also the foundation of our quality-driven video playback architecture we will assemble in Section 4. The basic problem is to reduce two different, but similar sections of video to a number that correlates with the decoding error the user sees.

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

3.5.1 Structural similarity index

The existing quality metrics range from simple mathematical operations to complex psychophysical models. The most widely used metric is the mean squared error (MSE), which is convenient, because it is easy to compute. Unfortunately, MSE does not always match perceived quality loss [16,28], because errors with an equal impact on the MSE can vary greatly in their visibility. A related metric is peak signal to noise ratio (PSNR) [7], but being just a logarithmically scaled version of MSE, it performs equally bad with respect to perceived quality loss.

Motivated by those deficiencies, Wang, Bovik, Sheikh, and Simoncelli developed the Structural Similarity (SSIM) Index [30], which we chose to use. The basic assumption of SSIM is that the human visual system is highly adapted to extract structural information from images. The algorithm therefore emulates the overall function of the human visual system. SSIM works by iteratively comparing aligned, limited local areas of two images. Extending SSIM to video is discussed in [31], where the authors also show SSIM to outperform all contenders of the VQEG Phase I test for video quality metrics.

SSIM fits well into our use case because of the following additional properties:

- It does not operate in the compressed domain, but on standard pixel-based image representations. This prevents dependencies between the quality metric and the decoder and thus supports the modularity of our whole approach.
- SSIM's sliding window calculation can operate locally, that is, if changes are known to be limited to a specific region of the image, SSIM computation can be accelerated by calculating over that region only.
- SSIM is symmetric. It merely calculates the visual difference between two

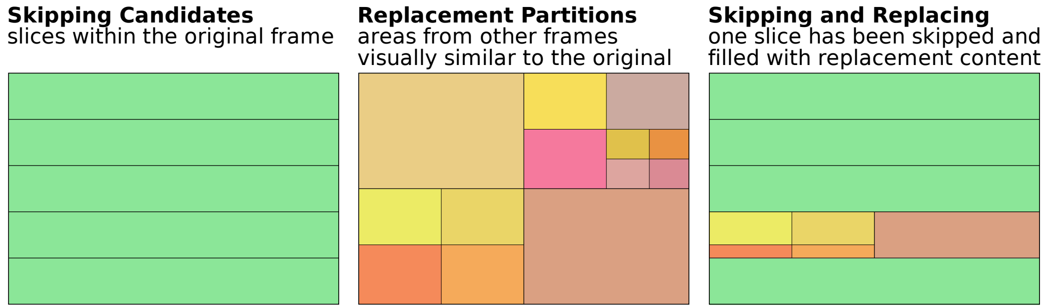


Figure 6. Skipped partitions and fallback content.

images, so it does not need any knowledge on which is the original and which is the degraded version. This is helpful when dealing with existing H.264 video that is already compressed.

3.5.2 Fallback decoding and quality loss

The quadtree generated offline by the preprocessor describes replacement partitions: areas of the frame that can be replaced with areas from previously decoded frames. What the decoder will later work with are skipped partitions: portions of the bitstream whose decoding can be skipped because the decoder can be realigned to continue decoding after the skipped partition. Those two partition types are orthogonal in our approach, but they could be unified in the future, when encoder and decoder support for H.264's built-in partitioning features, namely flexible macroblock ordering (FMO) and arbitrary slice ordering (ASO), receive more attention. Currently, these features are not implemented in common decoders or encoders. Thus, we use regular horizontal slices as our unit of skipping. It is easy to skip a slice in the bitstream and realign the decoder to the next slice by scanning for the NALU start code.

When the decoder decides to fallback-decode a skipped partition, exactly that area of the frame covered by the skipped partition is replaced. The fallback image is patched together from the quadtree's replacement partitions in that area as illustrated in Figure 6. By evaluating the motion vectors from the leaves of the quadtree, content is copied from reference frames.

Of course, when such a fallback decoding happens, the resulting image will be different from the fully decoded original. To make a sensible decision on which parts to skip, the scheduler needs information about this quality loss. The aforementioned SSIM metric provides exactly such a quantification. Once the offline preprocessor has built the quadtree, it performs a fallback decode for each slice individually and uses SSIM to calculate the error between fallback frame and original. These quality loss values are stored with the quadtree in

custom NALUs.

We have now developed a strategy for a lightweight decoding fallback to save resources. We formulated an algorithm that exploits existing motion vectors when creating a quadtree to describe the fallback content. If the decoder fallback-decodes a slice to save execution time, a quality loss metric enables the estimation of the error introduced in doing so. But so far, the effect of such a fallback has only been quantified for the frame in which it takes place. The upcoming section deals with that limitation.

3.6 Error propagation

Until now, we examined the error caused by fallback decoding within the frame directly affected. But today's decoder algorithms in general and H.264 in particular draw a large part of their compression efficiency from the exploitation of inter-frame redundancy by using temporal prediction to encode frames. This causes errors in one frame to be propagated into other frames, which then in turn cause further frames to have errors. An error introduced in one frame can affect any number of frames decoded later. In addition, H.264 uses spatial prediction to exploit intra-frame redundancy, which could lead to errors in one slice being propagated into other slices of the same frame, spreading the error over a larger portion of the current frame, which also increases the pollution of future frames.

The most accurate way to quantify the propagated error would be to measure it similarly to the error directly induced by the fallback decoder. But what was straightforward for this direct error is a lot more complex for the propagated error: Errors are potentially propagated over great distances, only an IDR frame definitely inhibits all propagations. Therefore, any slice's error can depend on the errors in every slice back to the previous IDR. The number of those slices can reach 100 and more and is generally unbounded. Every single one of those slices could be skipped or not, which would change the error inflicted on the current slice. So for a comprehensive error measurement, given a slice that is n slices away from the previous IDR, 2^n different slice skip patterns would have to be simulated and measured. This procedure would be repeated for every slice. It is quite clear that this way of measuring the error is completely infeasible. Therefore, we will *estimate* the error instead of measuring it. In the following, we first analyze a single propagation step in order to predict any propagation path later (for the interested reader: a more in-depth discussion of error propagation and our analysis of it can be found in [22]).

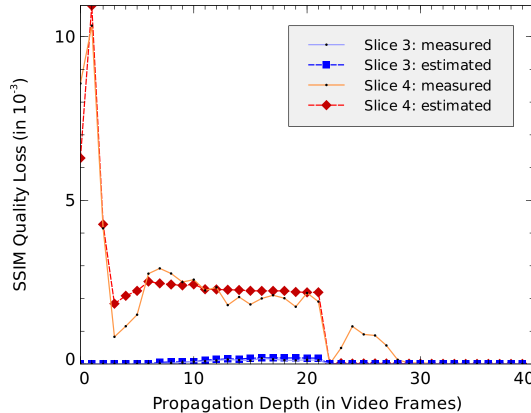


Figure 7. Measured and estimated error propagation when skipping one slice. The unmentioned slices 0, 1, and 2 of this 5-slices-per-frame video (BBC, see Table 1) showed no error in either estimation or propagation.

3.6.1 Single propagation step

The key property of referencing potentially degraded slices is that the error from that reference slice is copied to a certain degree into the current slice. The approach we pursued is to estimate that error in a straightforward way by checking motion vectors to see, what fraction of a particular reference slice's area is used by the current slice. The referenced area of one motion vector can differ in size due to motion vector subblocking, so some vectors will contribute a smaller area, others contribute larger ones. Of course, the actual temporal prediction is a lot more complex than just simple copying of reference frame data, but we will show that the simple approach of using motion vectors to determine area fractions being copied is sufficient for the intended purpose.

The result of motion vector analysis is a per-slice table that stores for each slice of each reference frame the area fraction that is copied into the current slice. The actual error estimation for a slice is then calculated by multiplying the error estimates of the reference slices with their respective area fraction and summing up these individual contributions.

3.6.2 Propagation paths

The estimation for single step error propagation can be straightforwardly composed to estimating error for entire propagation paths. Figure 7 shows an exemplary error propagation path when skipping one slice. Although the initial error is quite high, it can be observed that the error diminishes over time, which is caused by three effects:

- The decoding of a frame can use any frame currently available in the reference frame buffer. If a frame preferably uses references with smaller errors

or no error at all, it will in turn show a smaller error or even be error free.

- The reference frame buffer is of limited size. Therefore, frames are regularly pushed out of this buffer as new frames are added to it. If a frame with a high error is removed and a frame with a lower error is added, the potential for future frames to show errors is decreased.
- Even the intercoded slice types (P- and B-slices) can contain intracoded macroblocks. Because such macroblocks do not need any reference frames, they usually reduce the error.

3.6.3 Error accumulation

Now that we have analyzed single propagation steps and propagation paths, we will discuss error accumulation from multiple skipped slices in this section. So far, we have only analyzed the situation of a slice being fully decoded, but receiving an error from existing errors in reference frames. Those errors are initially caused by a slice being fallback decoded. But what happens, if the fallback decoding of a slice is further degraded by using already degraded reference frames, has not been investigated yet.

As the replacement copies data from various reference frames to fill in the skipped slice, it is obvious that we will use the same approach of using area fractions to quantify, how much the error of the reference in use will propagate into the current slice. This error estimate will only tell, how much the degraded fallback differs from the real fallback. But what is needed for error accumulation to be estimated properly is the difference between the degraded fallback and the original. During the creation of the motion vector quadtree, the preprocessor already measured the difference between the undegraded fallback and the original. To obtain the full difference, we add both difference contributions, which should work thanks to the linearity of SSIM.

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

(1) If a slice is not skipped, its error estimate is calculated by multiplying all area fractions with the error estimate of the respective reference slice.
(2) If a slice is skipped and fallback decoded, its error estimate is calculated by multiplying area fractions with the error estimate of the respective reference slice and adding the error directly induced by the fallback, which was determined by preprocessing.

Note that this method is completely agnostic to the type of the slices in question (I, P, B), only the referencing behavior which dictates the amount of pixels reused from references matters. Of course, I slices do not use any references, so they will contribute most to the diminishment of propagated errors.

In the next section, we explain how all elements of quality estimation and resource usage prediction fit together in a quality-aware playback architecture.

4 Architecture

The complete method makes use of all the modules presented above individually:

- prediction of decoding time for upcoming slices,
- quadtree-based fallback decoding, and
- prediction of error propagation in case a slice is skipped.

Existing H.264 bitstreams have to be preprocessed offline, where some additional information is embedded into the bitstream. The resulting video can then be used during playback to schedule slices based on quality and resources. After briefly summarizing, how the aforementioned modules work together in the preprocessor, we discuss the design of the online scheduler in detail.

4.1 Offline preprocessor

The preprocessor calculates the motion vector quadtree and stores it in a linearized fashion. Using the quadtree, the quality degradation between the original frame and a frame with one slice being fallback decoded is calculated. This is done for every slice of every frame. This process is costly and takes about the same time as the initial H.264 encoding. We assume an asymmetric setup with no execution time or other resource constraints on the encoder side, so a complex preprocessing is not a problem. For typical pre-encoded content, this expensive step has to be done only once to benefit a large number of consumers.

The preprocessor also traverses the video and calculates the area fractions of reference frames each slice depends on according to the error propagation model. To simplify the online use of error propagation data, these values are then combined into a single error emission factor for each slice. This factor describes the impact a degradation in the given slice will have on all future slices combined. The preprocessor calculates this by traversing the video backwards, summing up the individual error contributions and accounting them to the slices.

For fast online prediction of resource requirements, the preprocessor also stores the metrics for decoding time prediction. All the extra data are embedded into the H.264 bitstream as custom NALUs. Because the decoder will need to

make decisions based on quality and resource data of future slices, the data are embedded with a fixed look-ahead, so that linear reading of the stream will result in data of upcoming slices being available ahead of time. We used a look-ahead of 25 frames. The resulting video stream is still a standard-compliant H.264 stream, as an unmodified decoder will simply skip over the unrecognized NALUs.

Another step to be done offline is the training of the decoding time predictor as described in Section 2.2.2.

4.2 Online scheduling of slices

The decoding scheduler has to decide for each slice, whether it should be fully decoded for the sake of visual quality or fallback decoded, favoring lower execution times. In the following, we discuss what model the scheduling decision is based on and how the scheduling algorithm works.

4.2.1 Slice benefit model

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

Using predictions for the execution times of full decoding and fallback decoding we can calculate the time Δt saved by the fallback decoder. Using the directly inflicted error and the error emission factor from the bitstream, we can furthermore calculate the total quality loss Δq that would be caused by fallback decoding. The ratio $b = \frac{\Delta q}{\Delta t}$ thus expresses this slices benefit as a price-performance-ratio.

A higher benefit implies that the slice should better be decoded, whereas a lower benefit makes it a candidate for skipping.

4.2.2 Scheduler design

Video playback relies on the decoder being able to deliver decoded frames at a constant rate. The objective of the slice scheduler is therefore to maintain the natural deadlines of the frames while keeping the perceived visual quality as high as possible. Thanks to a look-ahead window, the scheduler knows the

metadata of upcoming slices ahead of time, so it can look into the future and accumulate predicted execution times to check, if a deadline is expected to be missed. If all deadlines inside the look-ahead window appear to be met, no action is required and slices are to be fully decoded. However, if a deadline will be missed, the scheduler needs to select a slice for skipping to help meeting the deadline by reducing the execution time. Because the actual execution times can still differ from the predicted ones, this deadline checking should be repeated after every slice to compensate for unexpected overtime during processing of an earlier slice.

Given a deadline expected to be missed and the execution times for full and fallback decoding of the individual slices, the core problem is to select a set of slices for fallback decoding such that the deadline is met and the loss in visual quality minimized. We can easily see that this problem is equivalent to the binary Knapsack Optimization Problem. As this problem is NP-hard, we are going to solve the problem using a greedy algorithm similar to the decreasing density greedy (DDG) algorithm discussed in [14], with the density being the benefit value b.

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

(1) Initialize the boolean skip variable with false for all slices in the sideband look-ahead window, meaning that all slices will be decoded.
(2) Calculate the current execution time budget as the difference between the current frame's deadline and the current wallclock time. This time is available for decoding the remaining slices of this frame.
(3) To exclude slices in the order of increasing benefit, the slice with the least benefit has to be remembered. The variable storing the least beneficial slice found is invalidated here, meaning that no slice has been examined yet.
(4) Iterate over all slices from the current one up to the last one in the look-ahead window.
    (a) If the skip variable for this slice is true, deplete the execution time budget by this slice's estimated replacement time.
    (b) If the skip variable for this slice is false, deplete the execution time budget by this slice's estimated decoding time. If the benefit of this slice is below the one of the currently remembered least beneficial slice, store this slice as the new least beneficial one.
    (c) If the current slice is the last one of a frame, then:
        (i) Check if the execution time budget dropped below zero, meaning we exceeded the deadline for this frame. If so, the remembered least beneficial slice's skip variable is set to true and the iteration bails out to Step 5.
        (ii) If the deadline has been met, replenish the execution time bud-

Algorithm 3 Scheduling decision

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

  do {
    deadlineMissed = false;

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

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

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

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

get with the display duration of one frame, because the deadline of the next frame will be later by this amount of time. Continue the iteration.

(5) If no deadlines have been missed or the skip variable of the current slice is set, the algorithm terminates. Otherwise the algorithm restarts at step 2.

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

The worst case runtime complexity of this algorithm is $O(n^2)$ for a look-ahead window of size n, but as the iteration bails out once no deadlines have been missed or once it has been determined that the current slice is to be skipped, the average runtime is lower. We measured an average per-slice scheduling overhead of a mere 15 μs for BBC video. Given that decoding times are in the magnitude of milliseconds, this overhead is quite acceptable.

4.3 Future integration of H.264 SVC

As mentioned earlier, we designed our architecture with the future of video coding in mind and are looking ahead to integrate H.264's scalable extension [25] into our model as seamlessly as possible. H.264 SVC, once it matures and becomes adopted by encoder and decoder vendors will provide a powerful and fine-grained way to trade decoding time for presentation quality. The decoder will have a lot more options to choose from, because H.264 SVC will allow to scale quality along three axes:

- Temporal resolution can be varied by changing the number of frames per second that are decoded.
- Spatial resolution scaling allows decoding individual frames at different resolutions.
- Quality scaling can be performed by using different quantization levels for the same frame and resolution to vary the amount of image detail that is decoded.

In our model, this scalable technology can completely replace the quadtree we introduced for exactly the same purpose. This greatly reduces the amount of processing we have to perform on the videos after encoding. The resulting data that has to be embedded as custom NALUs into the bitstream would also be reduced, thus lowering the bitstream size overhead of our technology tremendously.

At decoding time, we want to switch dynamically from one coding layer to another, so a very flexible scalability like [24] is needed. Here, the problem arises that changing quality, especially increasing it, is not possible at arbitrary points, but only when all required references are available. We plan to examine

the quality impact and error propagation behavior when switching layers, with the goal of predicting visual quality for arbitrary points in the quality space of decoding layers. The preprocessor would have to determine decoding time metrics and visual quality quantifications of the various levels and analyze and accumulate the error propagations caused by all conceivable decoder choices.

In the decoder, our idea of combining quality and resource usage would still apply, but would need to be enhanced to work with the wider choice of options. For example, in a high motion scene, the decoder should probably prefer temporal resolution over scalable resolution, as a reduction in frames per second would be perceived visually inferior to a reduction in image detail. The multiple dimensions of scalability pose some interesting research problems, as the decoder now has to find the path through all the options that leads to the highest quality under the given resource constraints. We look forward to accepting these challenges.

5 Evaluation

Our evaluation is twofold. First we present results of the building blocks of our technology individually. After that, we will evaluate the architecture as a whole. All timing-related results have been obtained under Linux on an AMD Opteron 2 GHz machine using the FFmpeg H.264 decoder [3]. The videos from Table 1 were used for the evaluation.

5.1 Individual building blocks

The architecture builds upon three accomplishments: decoding time prediction, fallback decoding and error propagation estimation.

5.1.1 Decoding time prediction

We used the videos Shore, BBC and Pedestrian as the training set to calculate the prediction coefficients (see Table 2). The prediction results can be found in Table 3. Especially the results of the Knightshields and Rush Hour sequences, which were not part of the training set are remarkable. Figure 8 illustrates the relative error of the Rush Hour prediction. A detail plot in Figure 9 shows, that the prediction follows the decoding time variations of different frames.

Table 1
Test videos (all progressive) used throughout the evaluation.

Video	Content	Frames	Resolution (Framerate)	Bitrate	Slices/ Frame	IDR Frames	Properties
Freeway [2]*	cars on a freeway	232	704×576 (24p)	1.73 Mbit/s	1	1	fixed camera scene
Golf [2]*	golfer making a swing	311	176×144 (24p)	30 kbit/s	1	1	fixed camera scene, very little motion
Shore [2]*	flight over a shoreline at dawn	682	352×288 (24p)	210 kbit/s	1	1	camera moving all the time
BBC [1]**	various combined broadcast quality clips from BBC motion gallery	2237	1280×720 (24p)	4.92 Mbit/s	5	43	clips with very different properties
Parkrun [4]**	man running in a park with an umbrella, trees, snow and water	504	1280×720 (50p)	8.66 Mbit/s	2	3	very detailed
Knight-shields [4]**	man walking in front of a wall of knight shields	504	1280×720 (50p)	9.68 Mbit/s	4	3	very detailed, zoom at the end
Pedestrian [5]**	shot of a pedestrian area, people pass by very close to the camera	375	1920×1080 (25p)	7.70 Mbit/s	8	2	low camera angle, large foreground motion objects
Rush Hour [5]**	rush-hour in Munich city with heat haze	500	1920×1080 (25p)	7.74 Mbit/s	8	2	high depth of focus, fixed camera, lots of motion

* videos have not been reencoded, the original FastVDO encoded material was used
** videos have been encoded using x264 (open GOPs, dynamic IDR interval of 25–250 frames with scene cut detection)

5.1.2 Fallback decoder

Each test sequence was preprocessed to calculate and embed the quadtree information, resulting in an increased bitstream size. We then randomly selected 10% of the slices. Using SSIM over the entire video, we objectively measured the quality when dropping the selected frames and replacing them with the previous frame. We also measured the SSIM quality when using our fallback decoding on the selected frames. Both SSIM quality values include errors propagated through reference frames. The results can be seen in Table 4. The average per-frame speedup of the fallback compared to full decoding ranges from 6.8 to 10.5. With quality improvement factors up to 5.1 the algorithm

Table 2
Prediction coefficient vector to calculate decoding time in milliseconds. Note that the transform metrics have been ignored by automatic column dropping.

Metric	Coefficient
pixels	3.602×10^{-3}
bits	2.860×10^{-5}
intra 4×4	1.559×10^{-3}
intra 8×8	1.589×10^{-3}
intra 16×16	1.665×10^{-3}
motion 4×4	3.700×10^{-5}
motion 8×8	3.797×10^{-5}
motion 16×16	1.935×10^{-4}
transform 4×4	0
transform 8×8	0
deblock edges	5.962×10^{-4}

Table 3
Per-slice decoding time prediction for various videos.

Video	Avg. Rel. Error (Std. Deviation)	Avg. Abs. Error (Std. Deviation)	Values within ±0.2 Rel. Err.	Values within ±1 ms Abs. Err.	99 % Quantile*
Freeway	0.189 (0.012)	1.52 ms (0.18 ms)	81.8 %	0.0 %	-1.386 ms
Golf	0.548 (0.099)	0.11 ms (0.01 ms)	0.3 %	100.0 %	-0.093 ms
Shore	0.286 (0.107)	0.44 ms (0.07 ms)	23.1 %	100.0 %	-0.207 ms
BBC	0.024 (0.108)	0.03 ms (0.45 ms)	93.1 %	98.7 %	0.960 ms
Parkrun	0.186 (0.146)	2.44 ms (2.08 ms)	64.2 %	14.2 %	1.236 ms
Knightsh.	0.036 (0.099)	0.20 ms (0.93 ms)	92.5 %	87.0 %	1.469 ms
Pedestrian	-0.021 (0.066)	-0.15 ms (0.73 ms)	98.1 %	96.0 %	1.396 ms
Rush Hour	-0.044 (0.043)	-0.37 ms (0.66 ms)	100.0 %	97.5 %	1.102 ms

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

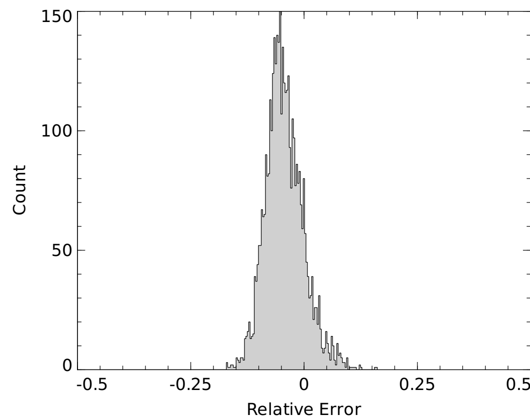


Figure 8. Relative error histogram of decoding time prediction for Rush Hour video

provides a better quality than frame drops with a reasonable size overhead of the bitstream. Only for the low-bitrate Golf video, frame drops resulted in

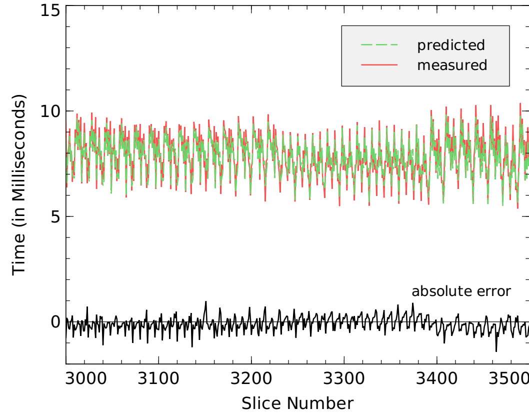


Figure 9. Prediction detail of Rush Hour video.

Table 4
Fallback decoding evaluation. The SSIM metric is 1.0 for equal frames and decreases with increasing perceptual difference. To calculate the improvement factor, both SSIM results were transformed to linear scale (QIF = $(1 - \text{SSIM}_{\text{Drop}}) / (1 - \text{SSIM}_{\text{Fallback}})$).

Test Sequence	Compressed Size Overhead	Average Speedup	SSIM of a Drop	SSIM of Fallback	Quality Improvement Factor (QIF)
Freeway	2.2 %	7.264	0.912	0.922	1.134
Golf	13.2 %	6.849	0.993	0.993	0.964
Shore	8.8 %	10.21	0.937	0.950	1.253
BBC	3.0 %	9.449	0.975	0.995	5.137
Parkrun	5.0 %	7.806	0.835	0.939	2.704
Knightshields	5.3 %	10.502	0.908	0.962	2.432
Pedestrian	3.6 %	7.887	0.965	0.990	3.639
Rush Hour	3.7 %	8.515	0.977	0.994	4.082

better visual quality than our fallback decoder. But remember that in both cases slices were dropped randomly and not based on their quality impact. Because low-bitrate video is not our main focus, we do not further investigate this anomaly in this work. We experimentally applied bzip2 compression to our metadata to reduce the size overhead. In an implementation for practical use, a domain specific compression would have to be devised.

5.1.3 Error propagation estimation

We estimated the resulting error propagation for skipping randomly selected slices at a rate of one slice out of ten. Comparing this estimate to the real error yields the differences listed in Table 5. As the difference between estimated and actual error is about an order of magnitude smaller than the absolute value of the propagated errors, this estimation will prove to be sufficiently accurate for our use.

Table 5
Error propagation estimation for various videos. For each video, randomly selected slices have been skipped at an average rate of one slice out of ten. The resulting errors were predicted for each slice and compared to the measured error.

Video	Slices/Frame	Avg. Difference (Std. Deviation)	Max. Error Value
Freeway	1	0.0645 (0.0514)	0.128
Golf	1	0.0097 (0.0113)	0.048
Shore	1	0.0662 (0.0830)	0.106
BBC	5	-0.0020 (0.0042)	0.067
Parkrun	2	0.0768 (0.0710)	0.168
Knightshields	4	-0.0156 (0.0136)	0.104
Pedestrian	8	-0.0072 (0.0051)	0.036
Rush Hour	8	-0.0045 (0.0037)	0.020

5.2 Architecture as a whole

The slice scheduling method selects slices for fallback decoding based on their benefit value. Slices with the lowest benefit are selected first. To evaluate this method in its entirety, we compared it to various other decoding strategies:

No fallback: Slices are not fallback decoded at all. Instead, when frames miss their deadline, the previous frame remains visible until playback recovers. Most current video players behave similarly.

Highest cost: The slice with the highest cost, that is: the highest decoding time, is skipped first. The reasoning behind this idea is that skipping the slices with the highest cost, a minimal amount of slices is skipped to meet the deadlines. This method uses decoding time prediction to estimate future deadline misses and to decide, which slices to fallback decode. Fallback decoding uses the quadtree metadata embedded in the bitstream.

Least direct error: The slice with the least directly introduced error, disregarding any error propagation, is skipped first. The assumption behind this method is that minimizing the first order error will also reduce the propagated error. Again, the decoding time prediction is used to estimate deadline misses. Fallback decoding is decided based on the stored error values for each slice.

Lifetime: Slices are skipped according to the ratio of decoding time and the frame's reference lifetime. This lifetime is the number of future frames, which can access the current frame in the reference buffer. It is current practice for MPEG-2 to skip B-frames first [18], because they are never used as references. This scheduling method extends this idea to H.264. Decoding time prediction and quadtree metadata are used as in the previously described approaches. The stored values for the direct error are multiplied with the frame's lifetime in the reference buffer.

Least benefit: Finally, this is the method we developed. Least Benefit re-

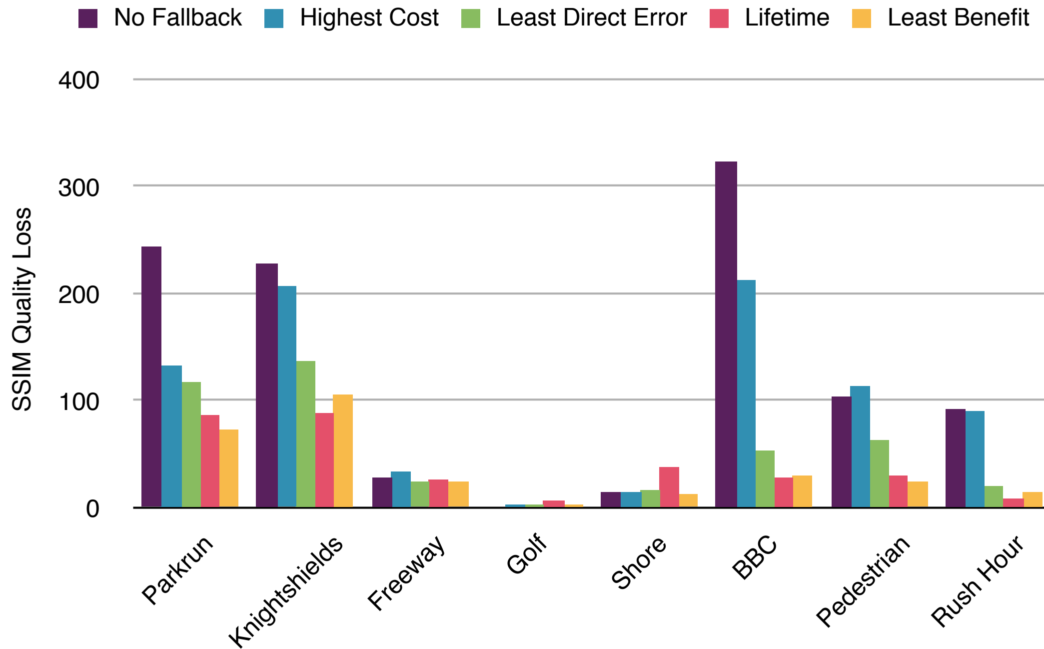


Figure 10. SSIM quality losses introduced by the various scheduler methods. The smaller the value the higher the quality of the video.

fines Lifetime by additionally using the error emission factor instead of the simplifying lifetime.

To compare the different approaches, we simulated playback on a machine incapable of decoding the video completely without missing deadlines. For the Freeway, Golf and Shore videos, we simulated a machine that could only decode 80 % of the frames within their deadline. For all other sequences, we reduced this fraction to only 50 % of the frames, meaning that a machine with only half the required CPU power is simulated. We evaluated the different strategies by running each video through each contender algorithm at the selected frame rate. The resulting video was compared to the original objectively using SSIM over the entire video. These quality differences can be seen in Figure 10.

We observe that, from no fallback to least benefit, using more building blocks of our architecture generally improves the visual quality under the given resource constraints. Lifetime and least benefit often perform similarly. However, in the Shore example, lifetime's simplification of the propagated error is shown to be insufficient. Our method may require preprocessing and enlarge the bitstream, but we hope to remedy these disadvantages by using H.264 SVC in the future.

6 Conclusion

In this article we described a framework for maximizing *user-perceived* video playback quality under constrained resources. This framework consists of the following three key parts:

(1) A method for predicting slice decoding time online. The prediction is based on metrics extracted from the bitstream, according to a decoder model. This allows us to estimate the costs associated with decoding each slice beforehand.
(2) We describe a fallback decoder for providing replacement content for a slice. This method is used in online adaptation decisions.
(3) We used the SSIM metric [30] for quantifying visual error introduced by the fallback decoder. We also presented a model for error propagation into future frames.

These three building blocks combined allow us to make sensible online decisions based on quality *and* resource usage with a runtime overhead of merely 15 μs per scheduling decision for an example video. To the best of our knowledge this is the first time that resource usage estimation, adaptivity, and a model for visual error including propagation are combined into one framework. According to our evaluation in Figure 10, our solution outperforms all competing approaches, supporting the idea of combining the described components.

We would like to emphasize here, that our approach is modular in the sense that the individual components can be improved independently. This will simplify the adoption of H.264 scalable video coding. Further research could also target extending our approach to other resource types than CPU, namely, network bandwidth and disk bandwidth.

Acknowledgments

This work was partially funded by the European Commission through the ROBIN FP6 project. Further thanks go to the anonymous reviewers for their comments.

References

[1] BBC Motion Gallery Reel. `http://www.apple.com/quicktime/guide/hd/bbcmotiongalleryreel.html`.

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

[3] FFmpeg project. http://ffmpeg.sourceforge.net/.

[4] High-Definition Test Sequences by SVT. http://www.ldv.ei.tum.de/lehrstuhl/team/Members/tobias/sequences/svt.

[5] High-Definition Test Sequences by Taurus Media Technik. http://www.ldv.ei.tum.de/lehrstuhl/team/Members/tobias/sequences/tmt.

[6] JM reference software. http://iphome.hhi.de/suehring/tml/.

[7] Peak Signal to Noise Ratio. http://en.wikipedia.org/wiki/PSNR.

[8] *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.* 1993.

[9] *ISO/IEC 13818-2: Generic coding of moving pictures and associated audio information — Part 2: Video.* 2000.

[10] Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 2003.

[11] *ISO/IEC 14496-2: Coding of audio-visual objects — Part 2: Visual.* 2004.

[12] *ISO/IEC 14496-10: Coding of audio-visual objects — Part 10: Advanced Video Coding.* 2005.

[13] Apple Inc. QuickTime HD Gallery System Recommendations. http://www.apple.com/quicktime/guide/hd/recommendations.html.

[14] James M. Calvin and Joseph Y-T. Leung. Average-case analysis of a greedy algorithm for the 0/1 knapsack problem. *Operations Research Letters*, 31(3):202–210, May 2003.

[15] Raphael A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica 4*, pages 1–9, 1974.

[16] Bernd Girod. What's wrong with mean-squared error? In *Digital Images and Human Vision*, pages 207–220. MIT Press, 1993.

[17] Michael Horowitz, Anthony Joch, Faouzi Kossentini, and Antti Hallapuro. H.264/AVC baseline profile decoder complexity analysis. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7):704–716, 2003.

[18] Damir Isović and Gerhard Fohler. Quality aware MPEG-2 Stream Adaptation in Resource Constrained Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2004.

[19] S. W. Lee and C. C. J. Kuo. Complexity Modeling for Motion Compensation in H.264/AVC Decoder. In *Proceedings of the IEEE International Conference on Image Processing (ICIP07)*, pages V: 313–316, 2007.

[20] Carsten Rietzschel. VERNER – ein Video EnkodeR uNd playER für DROPS, 2003. Master's thesis, Technische Universität Dresden.

[21] Michael Roitzsch. Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video, 2005. Großer Beleg (Undergraduate thesis).

[22] Michael Roitzsch. Slice-Level Trading of Quality and Performance in Decoding H.264 Video, 2006. Diplomarbeit (Master's thesis).

[23] Michael Roitzsch and Martin Pohlack. Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 06)*, Rio de Janeiro, Brazil, December 2006. IEEE.

[24] Heiko Schwarz, Detlev Marpe, Thomas Schierl, and Thomas Wiegand. Combined Scalability Support for the Scalable Extension of H.264/AVC. *IEEE International Conference on Multimedia and Expo*, July 2005.

[25] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable H.264/MPEG4-AVC extension. *IEEE International Conference on Image Processing*, pages 161–164, 2006.

[26] J. Stör and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1980.

[27] Mihaela van der Schaar and Yiannis Andreopoulos. Rate-distortion-complexity modeling for network and receiver aware adaptation. *IEEE Transactions on Multimedia*, 7(3):471–479, 2005.

[28] Dmitriy Vatolin, Alexander Parshin, Oleg Petrov, and Artem Titarenko. Subjective Comparison of Modern Video Codecs. Technical report, CS MSU Graphics and Media Lab Video Group, January 2006.

[29] Yong Wang and Shih-Fu Chang. Complexity Adaptive H.264 Encoding for Light Weight Streams. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Toulouse, France, 2006.

[30] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[31] Zhou Wang, Ligang Lu, and Alan C. Bovik. Video quality assessment based on structural distortion measurement. *Signal Processing: Image Communication*, pages 121–132, February 2004.

[32] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[33] Clemens C. Wüst, Liesbeth Steffens, Reinder J. Bril, and Wim F.J. Verhaegh. QoS Control Strategies for High-Quality Video Processing. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.