Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding

Michael Roitzsch
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
mroi@os.inf.tu-dresden.de

ABSTRACT

With multicore architectures being introduced to the market, the research community is revisiting problems to evaluate them under the new preconditions set by those new systems. Algorithms need to be implemented with scalability in mind. One problem that is known to be computationally demanding is video decoding. In this paper, we will present a technique that increases the scalability of H.264 video decoding by modifying only the encoder stage. In embedded scenarios, increased scalability can also enable reduced clock speeds of the individual cores, thus lowering overall power consumption.

The key idea is to equalize the potentially differing decoding times of one frame's slices by applying decoding time prediction at the encoder stage. Virtually no added penalty is inflicted on the quality or size of the encoded video. Because decoding times are predicted rather than measured, the encoder does not rely on accurate timing and can therefore run as a batch job on an encoder farm as is current practice today. In addition, apart from a decoder capable of slice-parallel decoding, no changes to the installed client systems are required, because the resulting bitstreams will still be fully compliant to the H.264 standard.

Consequently, this paper also contributes a way to accurately predict H.264 decoding times with average relative errors down to 1 %.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithms, Performance

Keywords

H.264, Video Encoding, Slices, Multicore, Scalability

1. INTRODUCTION

The industry is currently seeing the advent of multicore processor technology: Because of the well known energy consumption and heat dissipation problems with high-speed single-core CPUs, the mainstream computer market is switching to systems with lower nominal clock frequency, but with multiple CPU cores. Right now we see dual-core processors even in entry-level notebook computers and with research chips companies like Intel have proven the successful integration of 80 cores [9]. The trend towards multiple CPU cores on a single chip emerges in the world of embedded computing as well [5, 11], the major benefit being the reduced power consumption caused by distributing computations across multiple slower-clock cores and the resulting prolonged battery life of mobile devices.

But this new technology comes with a downside: In the bygone days of yearly increasing clock-speeds, algorithm developers and application programmers had to do virtually nothing to translate the technological advances into an application speed boost. Today however, to approach peak performance, algorithms have to take advantage of more than one CPU, otherwise they may even run slower than on yesterday's hardware. Never before has the continuing advancement of Moore's law relied so much on software.

Parallelizing algorithms is no easy task. And parallelizing them close to linear speedup is even harder. This paper focuses on the problem of decoding H.264 video [10]. This is known to be computationally demanding and even the latest single-core machines are just outside the recommended requirements for full HD resolution (1920×1080) H.264 playback [4]. Hence, this task is an obvious candidate for parallelization. We not only cover the problem theoretically, but also demonstrate implementations of the encoder and the decoder sides to retrieve real-life measurements and prove the practical applicability of our solution. Additionally, this work makes no assumptions on the decoder other than it being prepared for parallel decoding using slices (see next section). We deliver our solution entirely within a modified encoder, which allows end users to continue using the player application they are used to.

Section 2 briefly elaborates, how the H.264 standard supports parallelization. However, this is not the main contribution of this work, but is given to provide the reader with some insights into H.264. In Section 3, we present the scalability problems of the resulting parallelization and discuss the approaches to overcome them. Section 4 features the intended solution of applying video decoding time prediction, with Section 5 evaluating the improvement of scalability at virtually no cost. Section 6 compares against related work and Section 7 concludes the paper.

This work was presented as a work-in-progress on the 27th IEEE Real-Time Systems Symposium (RTSS 06) [12].

2. PARALLELIZING H.264 DECODING

Modern video codecs such as those in the MPEG standard family allow parallel decoding through a coding feature called slice. This is a set of macroblocks within one frame that are decoded consecutively in raster scan order. For the following reasons and solution details, slices are the most promising candidates for independent decoding by multiple cores:

- Individual frames have complex interdependencies due to the very flexible usage of reference pictures in H.264. Therefore it is hard to parallelize at frame level without limiting the encoders choice of reference frames. Such a limitation can inflict a bitrate or quality penalty.

- Other than frames, slices are the only syntactical bitstream element, whose boundaries can be found in the H.264 bitstream without decompressing the entropy coding layer. This decompression accounts for a large portion of the entire decoding process (see Figure 4), so for the sake of good scalability, it needs to be parallelized efficiently. Searching for slice boundaries and then distributing work packages to the individual cores allows for that.

- H.264 uses spatial prediction, which extrapolates already decoded parts of the final picture into yet to be decoded areas to predict their appearance. Only the residual difference between the prediction and the actual content is encoded. However, this coding feature was carefully crafted in the standard so that such predictions never cross slice boundaries and thus do not introduce dependencies among the slices of one frame.

- For global picture coding parameters (e.g., video resolution), which must be known before a slice can be decoded, the standard ensures that they do not change between different slices of the same frame.

- H.264 also uses a *mandatory* deblocking filter. This filter can operate across slice boundaries, which would defer the deblocking to the end of the decoding process of each frame, outside the slice context. If this is not desired, a deblocking mode which honors slice boundaries is available, but must be requested by the video bitstream. Therefore, it is an option that has to be enabled in the encoder. But since we plan to modify the encoder anyway, this does not pose a problem.

- Decoders usually organize the final picture and any temporary per-macroblock data storage maps as two-dimensional arrays in memory. Because the macroblocks of one slice are usually spatially compact and not scattered over the entire image, every decoder thread will operate on different memory areas when reading from or writing to such arrays. This minimizes the negative effects of false cacheline sharing. The notable exception to this is an H.264 coding feature called flexible macroblock ordering, which allows the encoder to arrange macroblocks in patterns other than the default raster scan order. But this feature is not commonly used.

In our work, we parallelized the open-source H.264 decoder from the FFmpeg project [8] to decode multiple slices simultaneously in concurrent POSIX threads. Each thread decodes a single slice. This allows us to perform measurements on real-life decoder code.

3. SCALABILITY CONCERNS

In this section, we examine the scalability problems with naively encoded slices and provide possible solutions to overcome those problems.

3.1 Scalability of Uniform Slices

To demonstrate and evaluate our ideas, we obtained some of the common uncompressed high-definition test sequences available from [2, 1], namely those listed in Table 1. Using the x264 encoder [17], which has been shown to perform competitively [15], we encoded an ensemble of H.264 test sequences. Every one of the uncompressed source sequences was encoded with 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 slices per frame, keeping the quality constant at the level shown in Table 1.[1] We made sure that the slices within each frame are uniform, meaning that they are all of the same size in terms of macroblocks they contain[2], because this is what naive encoding usually yields. Using our parallelized FFmpeg encoder, we measured the decoding time for each slice when every thread runs on its own CPU core. Since CPUs with a parallelism of up to 1024 threads are not commercially available yet, we simulated the dedicated, interference-free execution by running all threads on a single CPU core, forcing sequential execution of one thread after another. This is similar to a standard decoder run on a single CPU, but it still contains the overhead caused by the code added to enable parallelization. All results presented in this paper have been obtained on a 2 GHz Intel Core Duo machine.

In the uniprocessor case, a frame is complete, when all slices of that frame are fully decoded. In the multiprocessor case, each frame's decoding is finished after the slice with the longest execution time is fully decoded. Thus, for each encoded video, the speedup can be calculated by dividing the time required on a uniprocessor by the time required on a multiprocessor. The results can be seen in Figure 1. Although the parallel efficiency is acceptable, it still offers room for improvement.

3.2 Target Clock Speed of Uniform Slices

One of the goals of multicore computing is to reduce the clock speed of the individual cores to reduce power consumption. The same idea applies to power-aware computing when systems can adapt their clock frequency on demand. Thus, it is interesting to see, what clock speed reductions are possible with the given parallelization using uniform slices. Since

[1]The exact encoder command line options were:
`x264 --qp` *quality* `--threads` *slices* `--ref 15 --mixed-refs --bframes 5 --b-pyramid --weightb --bime --8x8dct --analyse all --direct auto`

[2]Differences of one macroblock have to be tolerated, because the overall macroblock count per frame of the given video resolutions might not be integer divisible by the desired slice count.

Table 1: Test sequences used for measurements and simulations.

Name	Content	Frames	Resolution	Properties
Parkrun	man running through park	504	1280×720	steady motion, high detail
Knightshields	man points at shield on a wall	504	1280×720	steady motion, zoom at the end
Pedestrian	people walking by in a pedestrian area	375	1920×1080	lots of erratic motion
Rush Hour	cars in a rush hour traffic jam	500	1920×1080	cars moving, heat haze
BBC	reel with broadcast quality clips	2237	1280×720	clips with very different properties

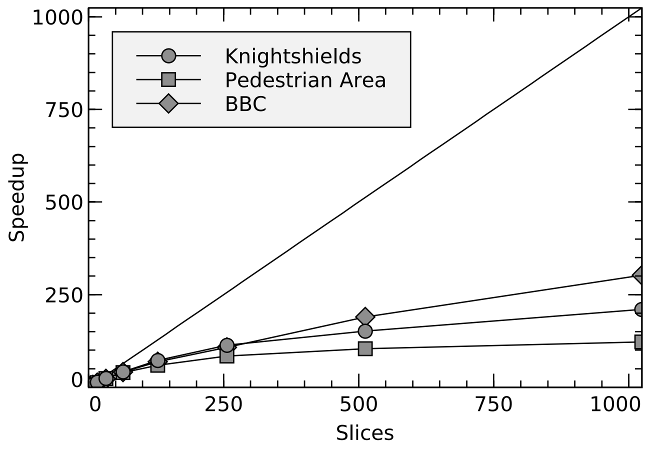


Figure 1: Speedup of parallel decoding.

every single video frame must be readily decoded within a fixed time interval, the target clock speed of the system cannot be designed for the average load of a video stream, but it must be designed for the peak load, which is the frame that takes the longest time to decode. To not catch a runaway value and also because today's video players are capable of tolerating a limited overload by buffering some decoded frames, we decided not to use the single longest per-frame decoding time, but rather the 95 % quantile of all frame decoding times. The resulting target clock speeds of the individual cores, scaled to the single-slice case, can be seen in Figure 2.

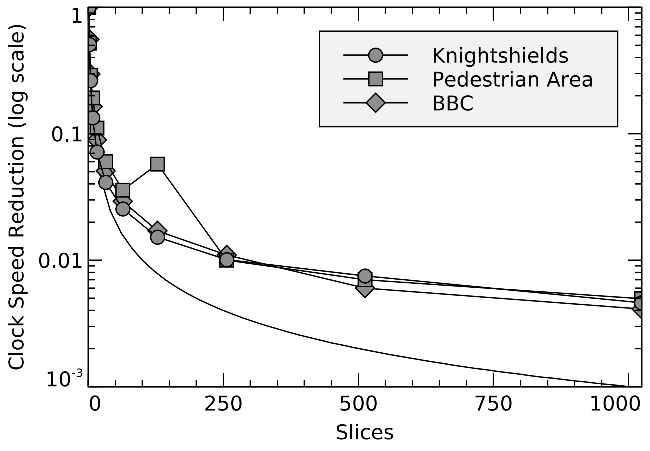


Figure 2: Clock speed envelope of parallel decoding.

3.3 Improving Parallel Efficiency

Parallel efficiency suffers because of sequential portions of the code that cannot be parallelized or because of synchronization overhead or idle time. The latter appears to be the main issue here: The frame is not fully decoded until the last of its slices is finished. The decoding of the upcoming frame cannot commence either, because inter-frame dependencies usually require the previous frame to be complete. Therefore, all threads that already finished decoding their respective slice must wait for the last thread to finish. This situation is common with uniform slices, because the time it takes to decode a slice does not depend so much on the macroblock count, but instead largely depends on the coding features that are used, which in turn are chosen by the encoder according to properties of the frame's content like speed, direction and diversity of motion in the scene.

One obvious way to overcome this problem is to replace the static mapping of slices to threads with a dynamic one: When the video is encoded with more slices than the intended parallelism, the slices can be scheduled to threads dynamically. For example, each thread that has finished decoding one slice can start to decode the next unassigned slice until all slices are decoded. Since the individual slices will take less time to decode, the waiting times for the longest running thread to finish up are also reduced.

However, this implies using more slices than strictly required, which does not come for free. Every slice starts with a slice header and due to the requirement of no dependencies to other slices of the same frame, all predictions like spatial prediction and motion vector prediction H.264 applies to reduce bitstream size are disrupted by slice boundaries. Consequently, to encode a video with more slices while maintaining the same quality level, one has to dedicate a larger bit budget to the encoder. Figure 3 shows the bitstream growth at constant quality level. Of course this penalty cannot be eliminated completely, because if a parallelism of n is intended, the video has to be encoded with at least n slices. What can be avoided is the extra price to be paid, when even more slices are used to increase parallel efficiency. In some applications this extra size increase may be unacceptable, especially since we provide a way to achieve the same result without this size overhead.

3.4 Balanced Slices

Our idea is to considerably reduce waiting times by encoding the slices for balanced decoding time: The slice boundaries shall no longer be placed in a uniform fashion, but they are placed so that, for each frame, the decoding times of all slices of that frame are equal. This invariably means that slice boundaries in adjacent frames will generally not be at the same position, but this does not pose a problem, since the H.264 standard allows different slice boundaries for each frame without any penalty. It also does not hin-

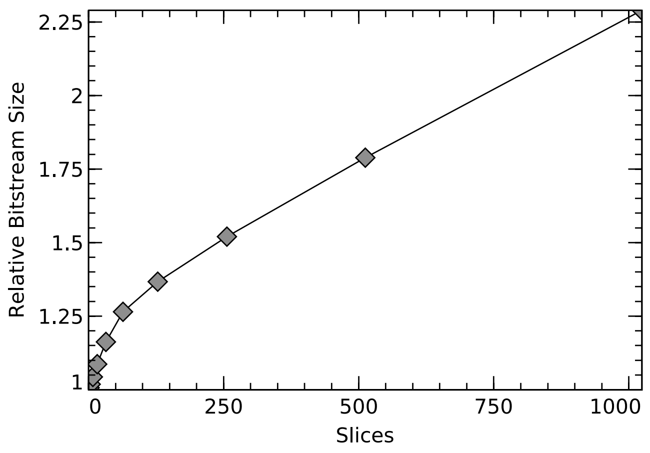


Figure 3: Bitstream size increase for BBC sequence due to the usage of multiple slices.

der parallelization, because the slice header always contains the position of the slice's first macroblock, so the slice decoder threads will know where to write the decoded data to. Further, this method is compatible to H.264's advanced reordering feature called flexible macroblock ordering, which organizes arbitrary macroblock patters in slice groups. As these are in turn subdivided into slices, the same balancing can be applied to the slices of these slice groups.

4. APPLYING DECODING TIME PREDICTION

Balancing the slices according to their decoding time is possible with a feedback process: The encoding is done in a first pass with uniform slices, then information about the resulting decoding times of the slices is fed back into the encoder so it can iteratively change the slice boundaries to approach equal decoding times.

The decoding times in this feedback loop could be determined by simple measurement: Running the encoded video through a decoder yields exact decoding times. However, this may not be applicable, since encoding jobs might run on hardware that differs from the systems targeted for end-user decoding. In addition, the encoding could be running in a distributed environment (encoder farm) or it might share one machine with other computation tasks, so exact measures cannot be determined. Furthermore, it would be very helpful to not only have decoding time information on the slice level, but for individual macroblocks. This would allow much faster convergence of the feedback loop towards balanced decoding times. But measurements on such a small scale might be subject to imprecisions due to measurement overhead. For those reasons, we propose to use decoding time prediction instead of actual measurement to determine the decoding times.

4.1 H.264 Decoder Model

We introduced a new technique to predict decoding times of MPEG-1/2 and MPEG-4 Part 2 video in [13]. The overall idea is to find a vector of metrics extractable from the bitstream for each frame. This vector's dot product with a vector of fixed coefficients gives an estimate of the decoding time. The coefficients are determined by the predictor automatically in a training phase. To ease finding the set of metrics to use, decoding is broken down into small subtasks. The metrics chosen for each subtask have to provide a good linear fit with the execution time of this subtask. Given such metrics and actual, measured decoding times, a linear least square problem solver calculates the coefficient vector that estimates the decoding time with the smallest error. The solver has been enhanced to avoid negative coefficients and to provide numerically stable and transferable results. The resulting coefficient vector is then stored and used for subsequent predictions.

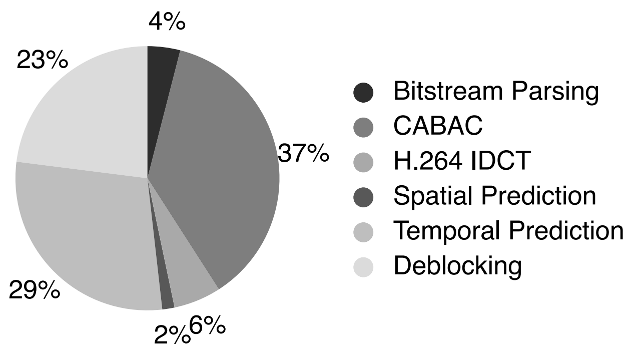


Figure 4: Execution time breakdown by functional block for BBC sequence.

We will not reiterate the entire method here, but explain the steps needed to apply the technique to H.264, which involve:

- mapping the functional blocks of H.264 to those of the general decoder model reproduced in Figure 5 and
- finding metrics to extract from the bitstream that correlate well with the execution times of the individual functional blocks.

To judge the relative contribution of the individual parts to the total decoding time, an execution time breakdown can be seen in Figure 4. In the following, we will discuss the modeling and metrics selection by functional blocks.

4.1.1 Bitstream Parsing and Decoder Preparation

The decoder reads in and prepares the bitstream of the upcoming frame and processes any header information available. The preparation part mainly consists of precomputing symbol tables to speed up the upcoming decompression. Its execution time is negligible, so we chose to treat these two steps as one. Because each pixel is represented somehow in the bitstream and the parsing depends on the bitstream length, the candidate metrics here are the pixel and bit counts. Figure 6a shows that a linear fit of both actually matches the execution time.

4.1.2 Decompression and Inverse Scan

The execution time breakdown (see Figure 4) shows the decompression step to be the most expensive. This sets H.264 apart from other coding technologies like MPEG-4 Part 2, where the temporal prediction step was by far the most expensive [13]. The reason for this shift is that the H.264 Main Profile uses a new binary arithmetic coding (CABAC) for compression, that is much harder to compute

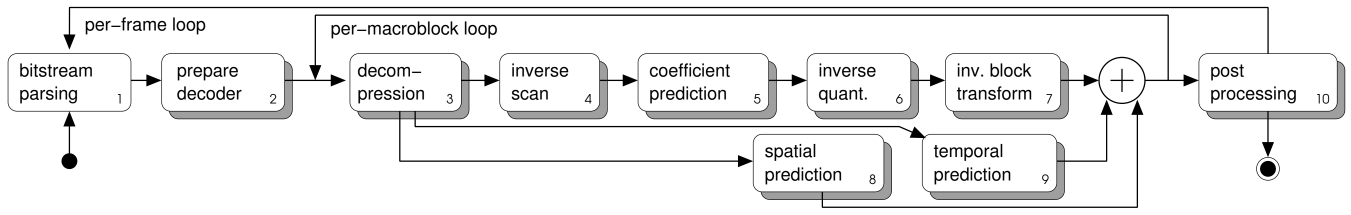
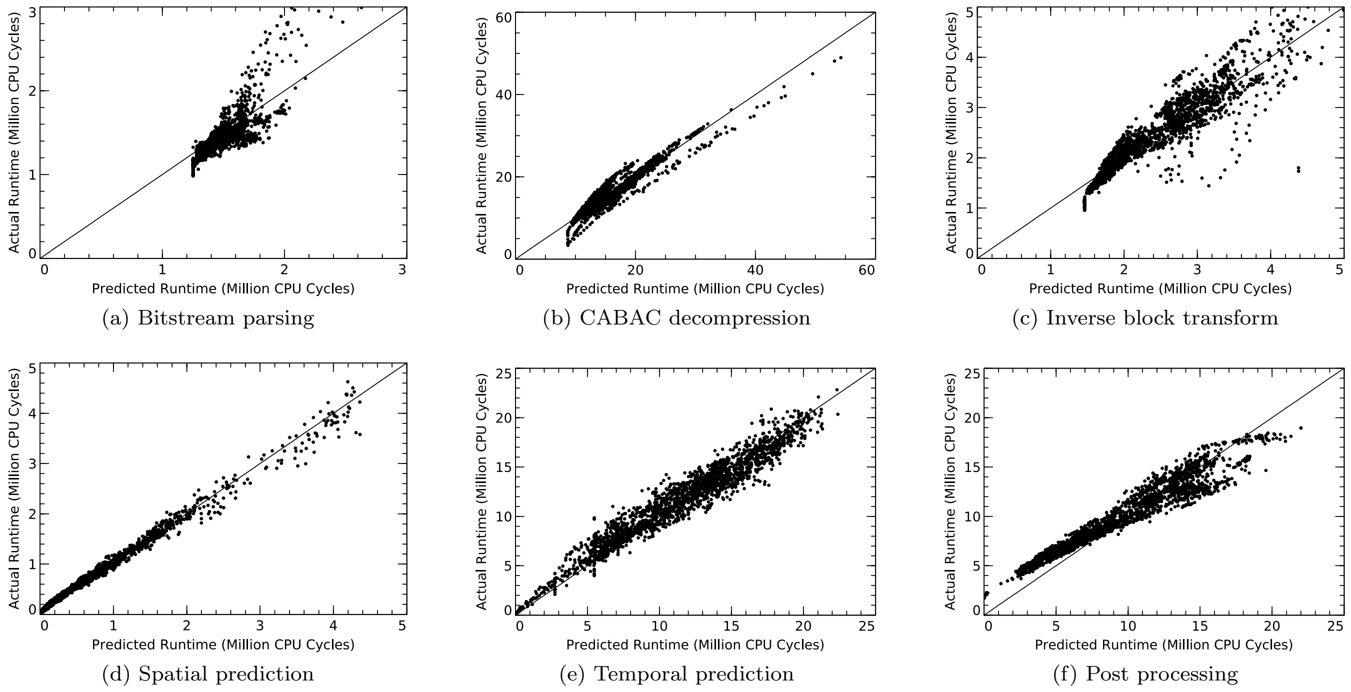


Figure 5: Decoder model.



(a) Bitstream parsing (b) CABAC decompression (c) Inverse block transform (d) Spatial prediction (e) Temporal prediction (f) Post processing

Figure 6: Execution time estimation for individual functional blocks (BBC sequence).

than the previous Huffman-like schemes. A less expensive variable length compression (CAVLC) is also available in H.264 and is used in the Baseline and Extended Profiles, where CABAC is not allowed. Both methods decompress the data for the individual macroblocks and already sort the data according to a scan pattern, so the inverse scan is a part of this step. Using the same rationale as for the preceding bitstream parsing, a linear fit of pixel and bit counts predicts the execution time well. We restrict ourselves to CABAC with results shown in Figure 6b. As this step accounts for a large share of total execution time, it is fortunate that the match is tight.

4.1.3 Coefficient Prediction

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

4.1.4 Inverse Quantization and Inverse Block Transform

These two steps convert the macroblock coefficients from the frequency domain to spatial domain, similarly to the IDCT in previous standards. However, H.264 knows two different transform block sizes of 4×4 or 8×8 pixels, which can even be applied hierarchically. 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 6c shows that this works. The remaining deviations are most likely caused by optimized versions of the block transform function for blocks, where only the DC coefficient is nonzero. But given the small percentage of total execution time this step contributes, we refrained from trying to improve this prediction any further.

4.1.5 Spatial Prediction

In this step, already decoded image data from the same frame is extrapolated with various patterns into the target area of the current macroblock. This prediction can use block sizes of 4×4, 8×8, or 16×16 pixels, so we account those prediction sizes separately. A linear fit of those counts adequately predicts the execution time (see Figure 6d).

4.1.6 Temporal Prediction

This step was the hardest to find a successful set of metrics for, because it is exceptionally diverse. Not only can motion compensation be used with square and rectangular blocks of different sizes, each block can also be predicted by a motion vector of full, half or quarter pixel accuracy. In addition to that, bi-predicted macroblocks use two motion vectors for each block and can apply arbitrary weighting factors to each contribution. In [13], we 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 [10] and some empirical improvements 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 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 can be seen in Figure 6e.

4.1.7 Post Processing

The mandatory post processing step tries to reduce block artifacts by selective blurring of macroblock edges. A sufficiently precise execution time prediction is possible by just counting the number of edges being treated (see Figure 6f).

4.1.8 Metrics Summary

The metrics selected for execution time prediction therefore 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.

4.2 Decoding Time Prediction and Balanced Slices

To balance the slices of one frame for equalized decoding times, we have to pass decoding time information to the encoder. Therefore, the decoding time prediction is trained according to [13] on the hardware end-users will decode the resulting videos on. Even if a single hardware platform cannot be pinpointed, there may be a typical embedded or even mobile target, for which the vendor wants to optimize power consumption and thus battery life. For example, a 3G network provider might want to optimize broadcast feeds for its common brand of cell phones. Videos and TV shows encoded for Apple's iTunes Store could be optimized for the iPod. In addition to that, content optimized for one platform will likely show improved scalability on other multicore platforms as well, unless their architecture differs radically.

The encoder can then use the training data obtained on the target hardware to balance the slices' decoding time in the resulting H.264 video. This is done in a way that supports the current practice of encoder use in the industry:

- *The encoding uses no time measurements, but decoding time prediction only.* No actual execution of decoder code and wall-clock sampling is performed. This allows setups that would interfere with timing behavior, like encoders running as background jobs or distributed on an encoder farm. Additionally, the predictor runs faster than the actual decoding.
- *Decoding time prediction is trained on separate hardware.* This enables the encoder to run on hardware entirely different from the end-user decoding hardware. Even custom silicon for H.264 encoding can be used, if it can adhere to slice boundaries from our balancing algorithm.
- *The prediction can be applied on the macroblock level.* This results in accurate decoding times for each individual macroblock. With such information available, balancing does not require many encoder iterations with boundaries for the balanced slices guessed from coarse timing information.

In the following section, we will validate the above claims.

Practically, the slice balancing works as follows: The video is first encoded traditionally, resulting in uniform slices. For each frame of the resulting video, decoding time prediction is applied to each macroblock. Ignoring not parallelizable leading and trailing housekeeping, the total decoding time t of a frame is the sum of its per-macroblock decoding times. If that frame should be divided into n balanced slices, each slice has to contain so many macroblocks that their cumulative decoding time is as close to $\frac{t}{n}$ as possible. This idea is easily implemented by iterating over all macroblocks of one frame in raster-scan order and accounting their decoding time.

5. EVALUATION

We will start by evaluating the decoding time prediction with both frame and macroblock granularity. After that, we demonstrate the scalability improvements and clock speed reductions of balanced slices. Unless noted otherwise, all results have been obtained on a 2 GHz Intel Core Duo machine.

5.1 Accuracy of Decoding Time Prediction

The predictor was trained [13] with the sequences BBC and Pedestrian (see Table 1), each in the single-slice variant. Applying the prediction to all test videos at frame level yields the results shown in Table 2. With average relative errors between -4.54 % and +4.55 %, the frame-level prediction is very accurate. Figures 7 and 8 present detailed results for the BBC sequence. You can see that the prediction does not only work in average, but closely follows decoding time fluctuations of individual frames.

Table 2: Frame-level decoding time prediction.

Name	Avg. Relative Error	Std. Deviation
Parkrun	3.98 %	6.68 %
Knightshields	4.55 %	3.41 %
Pedestrian	-1.25 %	3.34 %
Rush Hour	-4.54 %	3.00 %
BBC	1.69 %	5.67 %

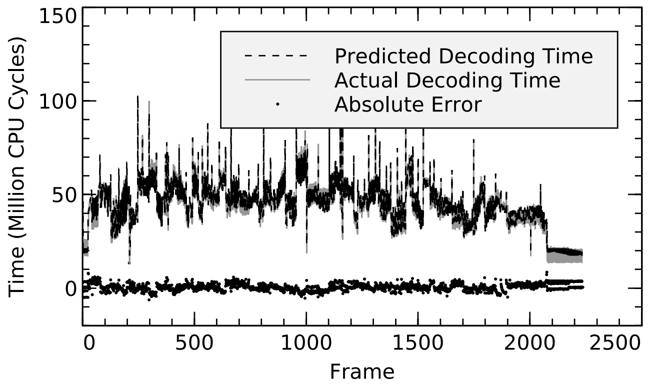


Figure 7: Actual decoding time, predicted decoding time and absolute error plotted over the runtime of the BBC sequence.

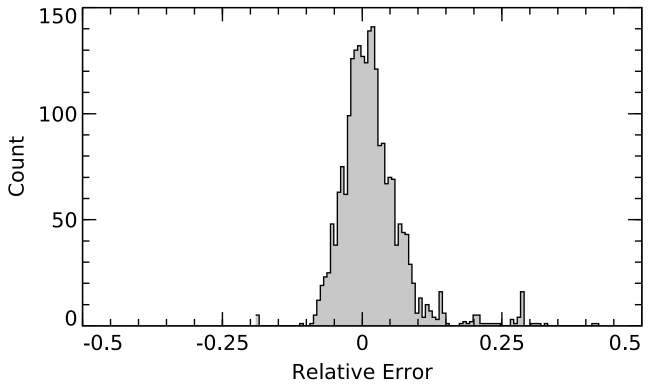


Figure 8: Histogram of the frame-level relative prediction error for BBC sequence.

However, as we plan to apply the prediction to individual macroblocks, it has to work with an even finer granularity. Figure 9 demonstrates this for BBC sequence, while Table 3 shows the results for all videos. With average relative errors for macroblock-level prediction as low as 0.86 %, the results are promising. Unfortunately, the standard deviation is higher than for frame-level prediction, which is most likely due to the noisier behavior on the macroblock-level caused by effects like cache misses.

Table 3: Macroblock-level decoding time prediction.

Name	Avg. Relative Error	Std. Deviation
Parkrun	0.86 %	11.13 %
Knightshields	0.91 %	9.56 %
Pedestrian	-5.42 %	10.84 %
Rush Hour	-8.77 %	8.70 %
BBC	-1.04 %	10.70 %

5.2 Speedup of Balanced Slices

To assess the increase in scalability, we first demonstrate the effect of the balancing encoding. Using decoding time prediction, we reencoded a balanced 2-slices version of the Parkrun sequence. Figure 10 visualizes slice boundaries and per-slice decoding times before and after balancing. You

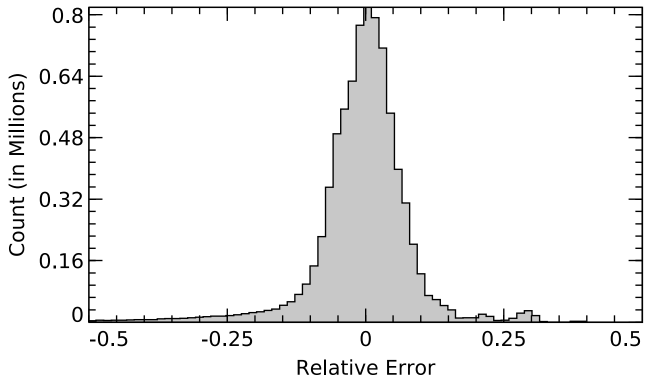


Figure 9: Histogram of the macroblock-level relative prediction error for BBC sequence.

can see that the slice boundaries move between subsequent frames, resulting in more equalized decoding times.

The resulting increase in speedup can be seen in Figure 11 for a selection of test sequences. The plots show practically achieved speedup with uniform slices and balanced slices as well as the hypothetical speedup with perfectly balanced slices, that experience only the penalty caused by not parallelizable code [3]. As CPUs with the shown number of cores are not yet available, measurements have been made with a single CPU as discussed in Section 3.1: Measuring the decoding times per slice allows estimates of the behavior on multiple cores, since parallel decoding of H.264 slices is largely interference-free.

5.3 Clock Speed Reduction

As introduced in Section 3.2, scalability improvements also offer the potential of reducing the clock speed of the individual cores. Because the cores must still be fast enough to decode the frame with the longest decoding time, the 95 % quantile of the decoding times is an interesting indicator (see Figure 12).

5.4 Bitstream Size Considerations

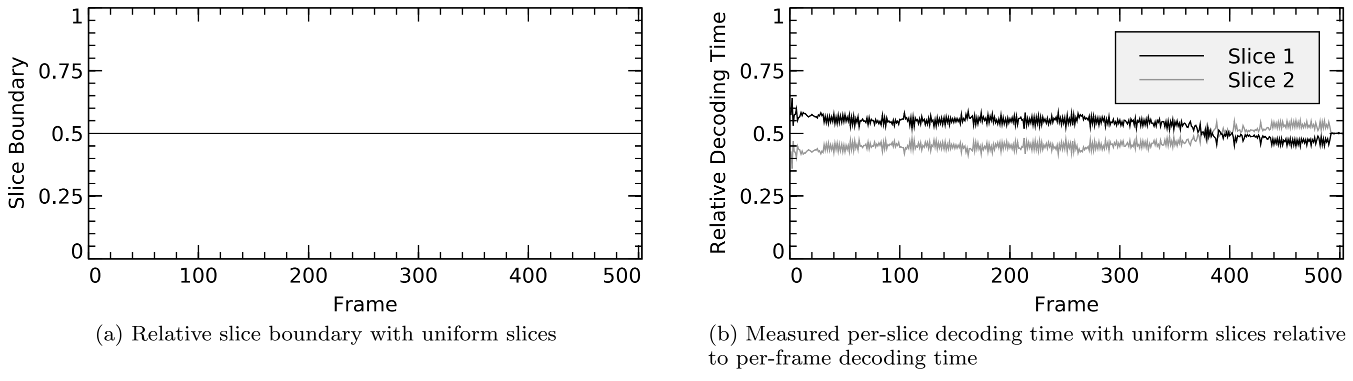
If quality is kept constant, slice balancing has negligible influence on bitstream sizes as can be seen in Table 4. Analogously, if average bitrate and thus bitstream size is kept constant, as is commonly done when given bit budget or storage constraints apply, the quality will not change visibly when using balanced slices.

Table 4: Bitstream size impact of balanced slices. Shown are the sizes in bytes for the four-slice versions.

Name	Unbalanced	Balanced	Rel. Difference
Parkrun	87172446	87164298	-0.009 %
Knightshields	45549457	45552631	+0.007 %
Pedestrian	23850582	23617081	-0.979 %
Rush Hour	34148349	33807077	-0.999 %
BBC	47386673	47441590	+0.116 %

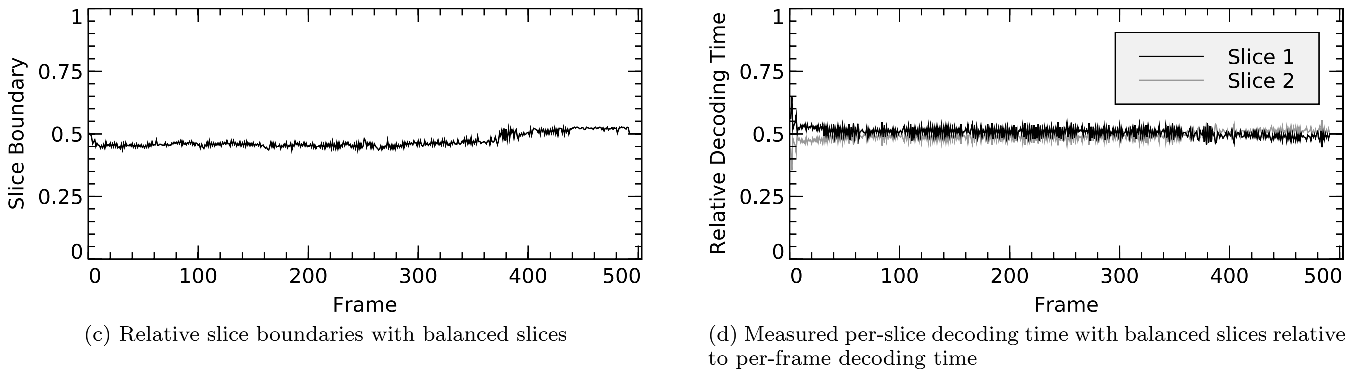
6. RELATED WORK

The idea to use slices to parallelize H.264 decoding is not new. Wiegand et al. formulated it in [16] for H.264. For



(a) Relative slice boundary with uniform slices

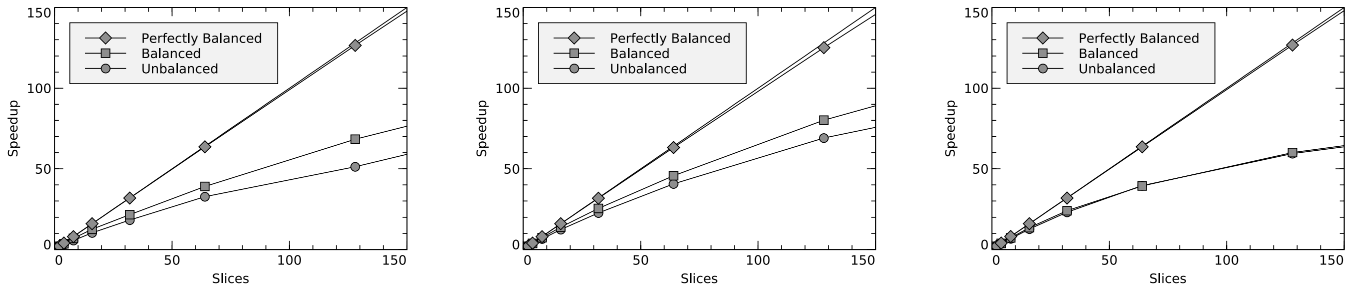
(b) Measured per-slice decoding time with uniform slices relative to per-frame decoding time



(c) Relative slice boundaries with balanced slices

(d) Measured per-slice decoding time with balanced slices relative to per-frame decoding time

Figure 10: The effect of slice balancing.



(a) Parkrun sequence

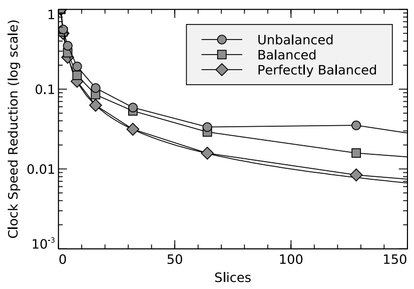
(b) BBC sequence

(c) Pedestrian Area sequence

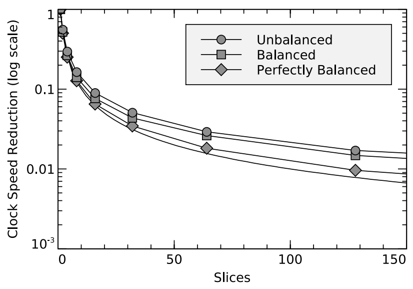
Figure 11: Speedup of parallel decoding with balanced slices.

preceding video decoding standards, the potential of slices for parallel decoding was evaluated even earlier. Bilas et al. analyzed parallel decoding of MPEG-2 in [6] and came up with two alternative approaches: GOP-level parallelism and slice-level parallelism. The former dispatches very large chunks of data to the individual processing units as GOPs are independent groups of pictures, separated by fully intracoded frames (I-frames). With MPEG-2, GOPs are typically 15 frames long. However, this idea is not suited for H.264, because I-frames in H.264 are more sparsely distributed, which is one source of H.264's increased coding efficiency compared to MPEG-2. In addition, to allow long-term prediction, an I-frame does not necessarily separate the stream into independently decodable units. Only IDR-frames (internal decoder reset frames) completely inhibit all inter-frame dependencies. As these can be seconds apart, using IDR-separated-GOPs as parallelizable workloads would introduce large delays until the decoder has received enough data to fully utilize the multicore CPU. Users would experience this as longer player response times and increased latency for live streams.

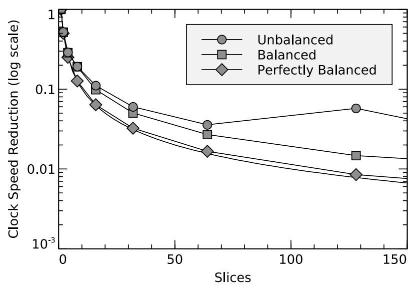
But [6] also analyzes slice-level parallelism for MPEG-2 and also recognizes speedup penalties caused by imbalances in the workload. However, they use a dynamic assignment of slices to threads and propose to start decoding slices from the next frame when cores are waiting. With MPEG-2, this approach may be viable, because most frames in typical MPEG-2 streams are B-frames, which are never used as references. Thus, decoding slices from the next frame becomes possible, whenever the current frame is a B-frame, as the next frame does not depend on the current frame in this case. Again, this idea is not suited for H.264, because



(a) Parkrun sequence



(b) BBC sequence



(c) Pedestrian Area sequence

Figure 12: Clock speed envelope of parallel decoding with balanced slices.

any frame can be a reference frame. Limiting the encoder in its choice of reference frames to allow this optimization is unwise, because it would prevent usage of the preceding frame, which is regularly the most effective one.

Therefore, due to the advances of H.264, work on parallelizing MPEG-2 does not directly apply. Some work on multicore H.264 decoding is available, like [14]. The authors also conclude that data partitioning is the enabling method. They also dismissed frame-level parallelism, but went even beyond slices to exploit the parallelism of individual macroblocks by decomposing their dependencies and selecting groups of independent macroblocks for concurrent decoding. While this is an intriguing idea and does not require special encoding, it requires modifications to decoders. The author's evaluation focuses more on memory load instead of scalability, so it is difficult to project, how far this concept scales. We could imagine inter-macroblock dependencies and inter-core cacheline transfers caused by the fine-grained workload dispatching to impede speedup for large numbers of cores.

In summary, while previous work optimizing either the encoder [7] or the decoder [6, 14] for multiprocessing is available, the novelty of our approach is the modification of only the encoder to improve performance of the decoder.

7. OUTLOOK AND CONCLUSION

We presented a new technique to improve parallel efficiency of multithreaded H.264 decoding. By using slices balanced for decoding time, this method can achieve improvements in terms of scalability or clock speed reduction. The latter is especially important on multicore systems and in power-aware computing since it allows to run the cores at lower clock speeds, which can help conserving energy. Our idea imposes virtually no overhead on encoding workload or video bitstream size. The current practice of using encoders as background jobs or in distributed encoder farms is supported. No modifications to the decoder other than enabling it for parallel decoding are necessary, so for example out-of-the-box QuickTime installations, which are capable of multithreaded decoding, should work.

The results are not dramatic, but as the improvement comes for free, we find the results still interesting. However, the first and foremost task for future work is to improve the balancing even further to push the speedup closer to the theoretical maximum for perfectly balanced slices. For this, we will evaluate the quality of the decoding time prediction to assess, whether it is accurate enough to achieve the scalability level we desire. Maybe an iterative approach with multiple balancing steps can help improving scalability. To counteract the resulting overhead, we will consider integrating the balancing steps with the multiple runs of a traditional multipass encoder. We also intend to evaluate, how a video balanced for one specific platform scales on different hardware to analyze the degree of architecture dependencies of the solution.

The implementation is not yet fully integrated into the encoding. Instead of two separate encoding passes, it would be beneficial to reencode on the frame level: Every frame is encoded first with uniform slices, balanced slice boundaries are determined and the frame is reencoded with balanced slices right away. This would speedup the encoding because of warm caches, but has no effect on the results presented here.

A potential improvement for the decoding is to have the encoder embed core affinity hints in the video bitstream: Depending on what reference frame the decoder needs to access, some slices can be decoded more efficiently on cores, where a certain reference slice has been decoded earlier, because the reference image data will still be in a cache close to that core. If the encoder has such intimate knowledge on the target hardware, it can anticipate such effects and advise the decoder with affinity hints it embeds in the H.264 bitstream.

Despite these opportunities for future work, we think we have helped to establish a technology leading towards a production-ready H.264 encoder capable of improving parallel efficiency for decoding on everyday systems.

8. REFERENCES

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

[2] High-Definition Test Sequences. `http://www.ldv.ei.tum.de/liquid.php?page=70`.

[3] Amdahl, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Conference* (1967), pp. 483–485.

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

[5] ARM. ARM11 MPCore. `http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html`.

[6] Bilas, A., Fritts, J., and Singh, J. P. Real-Time Parallel MPEG-2 Decoding in Software. In

Proceedings of the 11th International Parallel Processing Symposium (1997), pp. 197–203.

[7] Chen, Y. K., Tian, X., Ge, S., and Girkar, M. Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (2004).

[8] FFmpeg Project. `http://www.ffmpeg.org/`.

[9] Intel News Release. Intel Develops Tera-Scale Research Chips. `http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm`.

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

[11] Raytheon Company. MONARCH Processor Enables Next-Generation Integrated Sensors. `http://wwwxt.raytheon.com/technology_today/2006_i2/eye_on_tech_processing.html`.

[12] Roitzsch, M. Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 06)* (Rio de Janeiro, Brazil, December 2006), IEEE, pp. 77–80.

[13] Roitzsch, M., and Pohlack, M. 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, pp. 271–280.

[14] van der Tol, E. B., Jaspers, E. G., and Gelderblom, R. H. Mapping of H.264 decoding on a multiprocessor architecture. In *Proceedings of the SPIE* (May 2003), pp. 707–718.

[15] Vatolin, D., Parshin, A., Petrov, O., and Titarenko, A. Subjective Comparison of Modern Video Codecs. Tech. rep., CS MSU Graphics and Media Lab Video Group, January 2006.

[16] Wiegand, T., Sullivan, G. J., Bjøntegaard, G., and Luthra, A. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology 13*, 7 (July 2003), 560–576.

[17] x264 Project. `http://www.videolan.org/developers/x264.html`.