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

Michael Roitzsch
Department of Computer Science
Technische Universität Dresden
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 ongoing work, we will present a technique that increases the scalability of H.264 video decoding by modifying only the encoder stage. Speedup improvements up to 4.7 times can be achieved. 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. 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.

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 companies like Intel have announced to ship quad-core consumer systems early next year. But this new technology comes with a downside: 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. 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 [1]. Therefore, this task is an obvious candidate for parallelization. Section 2 briefly elaborates, how the H.264 standard supports parallelization. However, this is not the main contribution of this work. 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 to improve scalability at virtually no cost. Section 5 concludes the paper. The reader is expected to be roughly familiar with the H.264 video coding standard [4].

While previous work optimizing either the encoder [2] or the decoder [7] for multiprocessing is available, the novelty of our approach is the modification of only the encoder to improve performance of the decoder.

2. Parallelizing H.264 Decoding

Modern video codecs such as those in the MPEG standard family allow parallel decoding through a coding feature called slices. These are sets of macroblocks within one frame that are decoded consecutively. 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.
- Other than frames, slices are the only syntactical bitstream element, whose boundaries can be found without decompressing the entropy coding layer of the video stream. This decompression would otherwise be impossible to parallelize efficiently, so it has to be avoided for the sake of good scalability.
- 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 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 [3] to decode multiple slices simultaneously in concurrent threads. 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

Starting from the source material listed in Table 1, we used the x264 encoder to encode an ensemble of test videos. Every one of the source videos was encoded with 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 slices per frame, keeping the quality constant. 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[1], 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 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 under Linux on an AMD Sempron 2200+ (1.5 GHz).

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 1a. 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 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 2a.

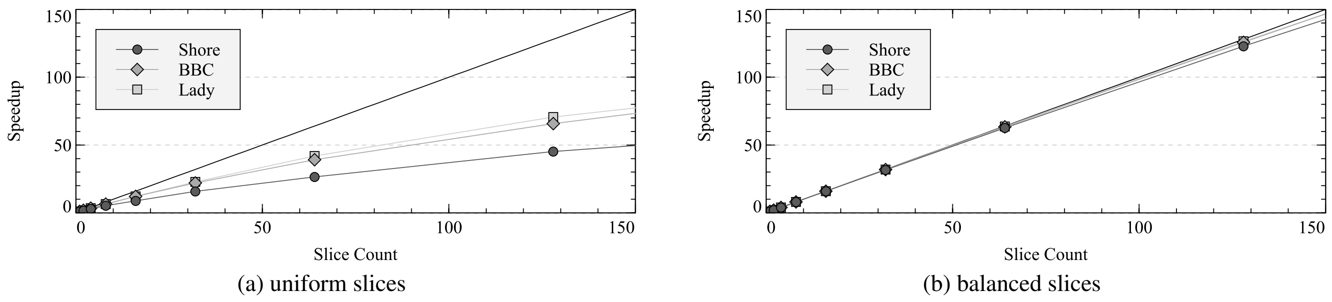
3.3. Improving Parallel Efficiency

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.

[1]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.

Name	Content	Duration	Resolution	Properties
Shore	flight over a shoreline at dawn	0:27 min	720×576	camera moving all the time
BBC	compilation of broadcast quality clips from BBC motion gallery	1:29 min	1280×720	clips with very different properties
Lady	movie trailer for “Lady In The Water”	1:44 min	1920×1080	high detail images with calm motion

Table 1. Test videos used for measurements and simulations.



(a) uniform slices (b) balanced slices

Figure 1. Scalability of parallel decoding.

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 and this 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 believe we can 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 hinder 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.

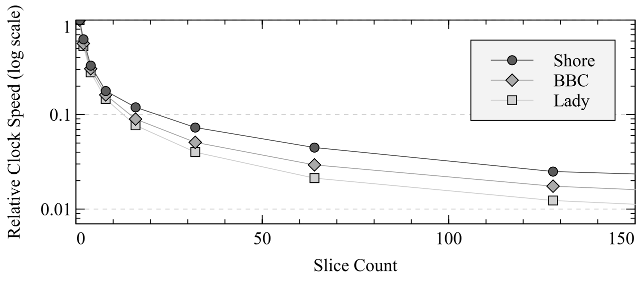
Figures 1b and 2b show the theoretical parallel efficiency of perfectly balanced slices. Although practical results will probably not be as good, the scalability and clock speed reduction can be improved considerably with this method. The speedup increases by a factor up to 1.4 for 4 slices, 2.0 for 32 slices and 4.7 for 1024 slices. The remaining gap to perfect parallel efficiency is due to code that cannot be parallelized.

4. Applying Decoding Time Prediction

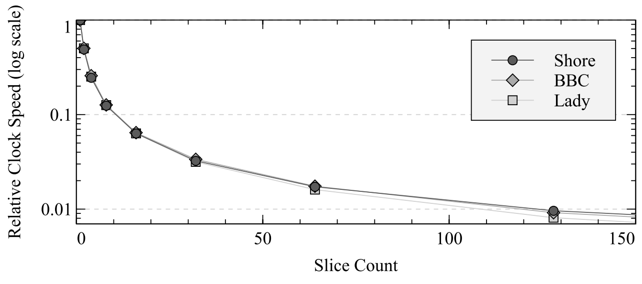
Balancing the slices according to their decoding time is possible with a feedback process: The encoding is done in a first pass with uniform slices, then information about the resulting decoding times of the slices is fed back into the encoder so it can iteratively change the slice boundaries to approach equal decoding times. High-quality encoding uses two or more passes anyway to optimize bit-budget assignment in a similar iterative way, so we plan to combine those passes to avoid inflicting a serious overhead on the encoding process. As decoding time behavior is usually monotone in the slice size[2], the iteration should converge.

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

[2]Shrinking a slice reduces decoding time, enlarging it increases decoding time, but not necessarily in a linear fashion.



(a) uniform slices



(b) balanced slices

Figure 2. Target clock speed of parallel decoding.

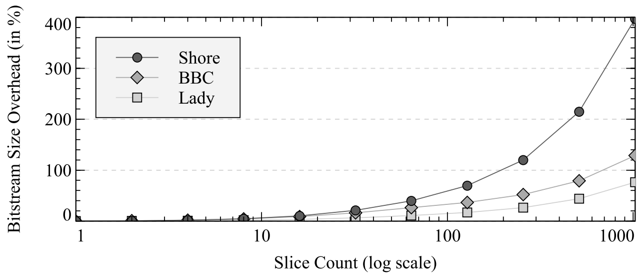


Figure 3. Bitstream size increase due to the usage of multiple slices.

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. We introduced a new technique to predict decoding times of MPEG-1/2 and MPEG-4 Part 2 video in [6]. We are confident that this method can be adapted to H.264 video and we already presented a mapping of the H.264 algorithm to the underlying generic decoder model in [5]. A preliminary decoding time prediction for a 20-slice version of the "Lady" video results in an average relative error of 0.044 at a standard deviation of 0.130. 97.9 % of all predictions are within ± 1 ms of absolute error.

5. Conclusion and Outlook

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 considerable improvements in terms of scalability and clock speed reduction. The latter is especially important on multi-core 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. 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.

In the future, we plan to improve the H.264 decoding time prediction and break it down to the macroblock level. Feeding the predictions into the encoder, we will evaluate how the actual parallel efficiency compares to the theoretical results presented here. We will also examine the impact on scalability, when a video with slices balanced for one target architecture is decoded on a different system. Finally, we hope to establish a technology leading towards a production-ready H.264 encoder capable of improving parallel efficiency for decoding on everyday systems.

References

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

[2] Y. K. Chen, X. Tian, S. Ge, and M. Girkar. 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.

[3] FFmpeg project. `http://www.ffmpeg.org/`.

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

[5] M. Roitzsch. Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video. `http://os.inf.tu-dresden.de/papers_ps/roitzsch-beleg.pdf`, 2005. Großer Beleg (Undergraduate thesis).

[6] M. Roitzsch and M. 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. To appear.

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