A macroblock-tree algorithm for high-performance optimization of dependent video coding in H.264/AVC

Fiona Glaser
Department of Computer Science
Harvey Mudd College

Abstract

We present a fast heuristic algorithm for improving rate-distortion performance in dependent video coding. Previous algorithms assume a constant quantizer in each frame and rely on impractically slow approaches, such as repeatedly encoding each frame dozens of times. Our approach modifies the quantizer on a per-macroblock basis and increases encoding time by at most a few percent and actually decreasing it in some cases. We implemented it in the open-source H.264/AVC encoder x264, where it gave PSNR improvements of up to 1.2 dB and SSIM improvements of up to 2.3 dB over existing fast rate-control algorithms.

1. Introduction

Intelligent bit allocation across a sequence of frames is critical to achieving high rates of compression in video coding. The standard approach to optimizing this tradeoff is rate-distortion optimization.[1] However, finding a rate-distortion-optimal solution for bit allocation across multiple dependent frames is typically infeasible due to its high complexity.

Most existing solutions, both heuristic and optimal, have impractically high complexity or provide only a small compression improvement. Furthermore, despite their high complexity, most existing solutions still assume a constant quantizer within each frame. We propose a macroblock-tree algorithm to optimize per-block quantizer selection across multiple dependent frames at negligible computational cost.

This paper is organized as follows. Section 2 provides background for the problem of rate-distortion-optimal ratecontrol and existing heuristics. Section 3 gives a high-level overview of the macroblock-tree algorithm and its purpose. Section 4 explains the lookahead framework of x264 which was used as a basis for our implementation of the macroblock-tree algorithm. Section 5 introduces the macroblock-tree algorithm itself. Section 6 contains an analysis of the typical consequences of the algorithm. Perceptual considerations related to the macroblock-tree are discussed in section 7. Numerical quality results are presented in section 8, with performance analysis in section 9. The paper is concluded in section 10.

2. Background

The simplest possible ratecontrol method is one which, given some set of constraints, attempts to target a constant quantizer. It was realized very early in the development of video coding techniques that this was suboptimal: by varying the quantizers of frames using rate-distortion optimization techniques, one could improve quality, usually measured in the form of Peak Signal-to-Noise Ratio (PSNR) or sometimes Structural Similarity (SSIM)[13], at a given rate.

Early algorithms to optimize this problem were typically brute-force, trying many quantizers for each frame in an attempt to pick the best one. The advent of inter-frame compression complicated the matter, as the number of possible quantizer combinations grew exponentially and frames could no longer be optimized independently. This was solved using Viterbi algorithms, as in Ramchandran et al.[2]

However, while Viterbi made the optimal solution tractable, such algorithms were still very slow and in some cases still had exponential worst-case convergence time. One “fast” algorithm by Sermadevi et al had a runtime of O(Q*N*M), where Q is the number of quantizers to search, N is the number of frames, and M is the number of frames affected by a change in allocation to any given frame.[3] Even as improved in Toivonen et al, this class of algorithms still typically took dozens or hundreds of encode calls per frame[4], putting it out of the reach of most practical encoders.

Nevertheless, many simple heuristics have been derived from this research and used in practical encoders. One

described in Ramchandran et al is that the "... I-frame is the most important of the group of pictures and must not be compromised," in other words, that I-frames should be given higher quality than other frames.[2] Another common heuristic is assigning lower quality to unreferenced B-frames, as their pixels are not reused for prediction.

These heuristics are extremely common in modern video encoders. x264 in particular uses an I-frame offset of 1.4: 1.4x higher quality than P-frames, measured in linearized quantizer scale. In H.264, this maps to approximately -3 QP. Similarly, x264 uses a B-frame offset of 1.3, or 1.3x lower quality than P-frames, approximately +2 QP.

Another common heuristic is known as "quantizer curve compression", or "qcomp". qcomp attempts to compensate for the variance in RD curves among frames without the complexity of calculating the actual RD curves. It does this by leveraging the correlation between the inter residual of a frame and its importance for predicting future frames.

Typically inter prediction is less useful in sections of video with high inter residual, and thus the value of a higher quality reference frame is lower. As such, qcomp adjusts the quality of frames in inverse proportion to their inter residual. This algorithm was originally invented for use in libavcodec's MPEG video encoder. x264's implementation of qcomp measures the Sum of Absolute Hadamard-Transformed Differences (SATD) residuals of frames, performs a Gaussian blur over the residuals to limit local variation, then multiplies the quality of all frames by $(\text{SATD residual})^{0.4}$. Combined with heuristics such as constant I-frame and B-frame offsets, qcomp helps approximate the effect of a much slower RD-optimal ratecontrol algorithm with negligible computational cost.[5]

In 2006, the algorithm described by Toivonen (colloquially dubbed RDRC, or Rate-Distortion Rate Control) was implemented in x264.[6] Testing showed significant quality improvements at constant rate, in some cases upwards of 1db PSNR. This demonstrated that there was still a large gap between the qcomp fast approximation and the optimal solution.

One weakness common to every algorithm mentioned so far is that they all operate on a per-frame level as opposed to a per-block level, ignoring variation within a frame. Ramchandran et al briefly mentioned a possible extension of their algorithm to this, but even when ignoring inter-block dependencies within a frame, their modification introduces an additional $O(4^M)$ complexity factor.

3. High-level overview of macroblock-tree

The purpose of the macroblock-tree algorithm is to estimate the amount of information that each macroblock contributes to the prediction of future frames. This information allows macroblock-tree to weight the quality of each macroblock based on its contribution. To do this, macroblock-tree works in the opposite direction of prediction, propagating information from future frames back to the current frame to be encoded.

In order to do this, macroblock-tree needs to know various pieces of information, or at least approximations thereof. First, it must know the frame types of the future frames to be analyzed. Second, it must know the motion vectors of these frames. Third, it must know how much information is to be propagated at each step, which will be calculated based on the inter and intra costs. The x264 lookahead, described next, is how macroblock-tree gets this information.

4. The x264 lookahead

x264 has a complex lookahead module designed to estimate the coding cost of frames that have not yet been analyzed by the main encoder module. It uses these estimations to make a variety of decisions, such as adaptive B-frame placement, explicit weighted prediction, and bit allocation for buffer-constrained ratecontrol. For performance reasons, it operates on a half-resolution version of the frame and calculates SATD residuals only, doing no quantization or reconstruction.[5]

The core of the lookahead is the *x264_slicetype_frame_cost* function, which is called repeatedly to calculate the cost of a frame given p_0, p_1, and b values. p_0 is the list-0 (past) reference frame of the frame to be analyzed. p_1 is the list-1 (future) reference frame of the frame to be analyzed. b is the frame to be analyzed. If p_1 is equal to b, the frame is inferred to be a P-frame. If *p0* is equal to b, the frame is inferred to be an I-frame. As *x264_slicetype_frame_cost* may be called repeatedly on the same arguments as part of the algorithms that use it, the results of each call are cached for future usage.[7]

x264_slicetype_frame_cost operates by calling *x264_slicetype_mb_cost* for each macroblock in the frame. As the frame is half-resolution, each "macroblock" is 8x8 pixels instead of 16x16. *x264_slicetype_mb_cost* performs a motion search for each reference frame (past for P-frames, past and future for B-frames). This motion search is typically a hexagon

motion search with subpel refinement, as described in Zhu et al.[8]

For B-frames it also checks a few possible bidirectional modes: a mode similar to H.264/AVC's "temporal direct", the zero motion vector, and a mode using the motion vectors resulting from the list0 and list1 motion searches. *x264_slicetype_mb_cost* also calculates an approximate intra cost. All of these costs are stored for potential future usage. This is important for macroblock-tree, which will need this information for its calculations.

The results of this analysis are used primarily in a Viterbi algorithm for adaptive B-frame placement. The output of this Viterbi algorithm is not merely the next frame-type to use, but also a plan for the frame types to use for the next N frames, where N is the size of the lookahead. This plan is effectively a queue: it changes over time as frames are pulled from one end and encoded using the specified frame types, frames are added to the other end as new frames enter the encoder, and the plan is recalculated. The existence of this plan is important for macroblock-tree: it means that any algorithm that needs to know frame types for future frames has a reasonably accurate estimation of what they will be, even if the GOP structure isn't constant.

As a result of this, macroblock-tree is aware of the frame types of the next N frames, approximate motion vectors and mode decisions, and inter/intra mode costs (in the form of SATD scores). The computational cost of this is effectively zero, as this data was already being calculated for other purposes within the encoder. Even so, with respect to total encoding time, the computational cost of the lookahead is low.

5. The Macroblock-tree algorithm

In order to perform the aforementioned estimation of information propagation, macroblock-tree keeps track of a *propagate_cost* for each macroblock in each lookahead frame: a numerical estimate, in units of SATD residual, of how much future residual depends on that macroblock. This is initialized to zero for all frames in the lookahead.

Macroblock-tree begins its operation in the last (futuremost) minigop in the lookahead, first operating on B-frames, then P-frames. It then works its way backwards to the first frame in the lookahead (the next frame to be encoded). In this way, macroblock-tree works backwards: it "propagates" dependencies backwards in time. Thus, when we "propagate" a dependency, we are operating in the opposite direction of the actual dependency itself: moving information from a frame to its reference frames.

For each frame, we run the *propagate* step on all macroblocks. The *propagate* step of macroblock-tree operates as follows:

1. For the current macroblock, we load the following variables:
    - *intra_cost*: the estimated SATD cost of the intra mode for this macroblock.
    - *inter_cost*: the estimated SATD cost of the inter mode for this macroblock. If this value is greater than *intra_cost*, it should be set to *intra_cost*.
    - *propagate_in*: the *propagate_cost* for the current macroblock. It is intentional that *propagate_cost* is zero for the first frame that *propagate* is run on, as no information has been collected yet for that frame.

2. We calculate the fraction of information from this macroblock to be propagated to macroblocks in its reference frame(s), called *propagate_fraction*. This is approximated by the formula *1 – intra_cost / inter_cost*. As an example, if the inter cost for a macroblock is 80% of the intra cost for that macroblock, we say that 20% of the information in that macroblock is sourced from its reference frame(s). This is a clearly a very rough approximation, but is fast and simple.

3. The total amount of information that depends on this macroblock is equal to *(intra_cost + propagate_in)*. This is the sum of all future dependencies (up to the edge of the lookahead) and the intra cost of the current macroblock. We multiply this by *propagate_fraction*, resulting in the approximate amount of information that should be propagated to this macroblock's reference frames, *propagate_amount*.

4. We split *propagate_amount* among the macroblocks in its reference frame that are used to predict the current block. The splitting is weighted based on the number of pixels used from each macroblock to predict the current macroblock. This can be calculated based on the motion vector of the current macroblock. If the block has two reference frames, as in the case of biprediction, the *propagate_amount* is split between the two equally, or if weighted B-frame prediction is enabled, according to the biprediction weight. The properly split portions of

propagate_amount are then added to the *propagate_cost* of each of the macroblocks used for prediction. Note that, if we ignore the effects of interpolation filters for simplicity, at most 4 macroblocks in each frame can be used for prediction of the current macroblock.

The result of this process is that the *propagate_cost* values of the references of the current frame have been increased based on the contents of the current frame. By repeating this in reverse order for all the frames in the lookahead, we approximate the contribution of each macroblock in the next to the quality of the rest of the frames in the lookahead.

Finally, the *finish* step is applied to the macroblocks in any frame we wish to acquire final quantizer deltas for. This is typically the next few frames to be encoded. The *finish* step of macroblock-tree operates with the same inputs as *propagate*, but instead outputs an H.264 quantizer delta:

$$\textit{Macroblock QP Delta} = -\textit{strength} * \log_2(1 + \textit{propagate\_cost} / \textit{intra\_cost})$$

where *strength* is an arbitrary factor derived from experimentation. Testing suggests that 2 is a near-optimal value for most videos. As the H.264 quantizer scale doubles precision every 6 Qps, this means that optimal quantizer precision appears to scale roughly with the cube-root of the resulting *(1 + propagate_cost / intra_cost)* value used in the *finish* step.

It should be noted that this algorithm results in unreferenced frames having QP deltas entirely of zero, as nothing is ever propagated to them:

$$\textit{Macroblock QP Delta} = -\textit{strength} * \log_2((\textit{intra\_cost} + 0) / \textit{intra\_cost}) = -\textit{strength} * \log_2(1) = 0$$

This is intentional: by the logic of macroblock-tree, all unreferenced macroblocks have equal (and thus minimal) value.

Macroblock-tree derives its name from the fact that its operation can be represented as a tree structure over the macroblocks of a video where the nodes are macroblocks and the edges are prediction dependencies. The weights of the edges map to the *propagate_fraction* value. Despite the name, as seen above, macroblock-tree can be implemented without mapping the lookahead into an explicit tree structure – the motion vectors and costs are sufficient to implicitly store the information for the tree.

It is important to note that x264 already has a variance-based adaptive quantization (VAQ) algorithm implemented. Macroblock-tree adds quantizer deltas *on top of* the effects of the existing adaptive quantization algorithm. While it might be tempting to consider adaptive quantization a part of macroblock-tree to inflate its SSIM gain, they are two separate algorithms and VAQ will not be covered or benchmarked in this paper. This is important when comparing to other ratecontrol work that optimizes for SSIM.

6. Analysis

Macroblock-tree has a number of consistent effects with regard to bit distribution. One of these is the effect on B-frame quantizers. As mentioned in the introduction, B-frame quantizers are typically derived via an offset from the P-frame quantizer. Macroblock-tree is effectively the opposite: unreferenced B-frame quantizers are always the same, whereas the neighboring P-frame quantizers vary.

The result of this is effectively an adaptive B-frame quantizer offset. In areas of high motion, B-frames tend to get quantizers not much higher than that of nearby P-frames. In areas of low motion, the quantizer difference is much higher: +4-6 QP or more in some cases. It is important to note that, in x264, the regular B-frame quantizer offsets are disabled when macroblock-tree is on, since they serve the same role.

One might assume similarly that macroblock-tree can replace keyframe quantizer offsets. However, testing suggested this was not the case: macroblock-tree typically lowered quantizers for an entire scene, not solely the first frame in the scene. Keyframe quantizer offsets remained useful for PSNR, and so were kept in x264. Algorithmic derivation of keyframe quantizer offsets likely requires some form of lookahead quantization, as in Schmitsch et al.[9]

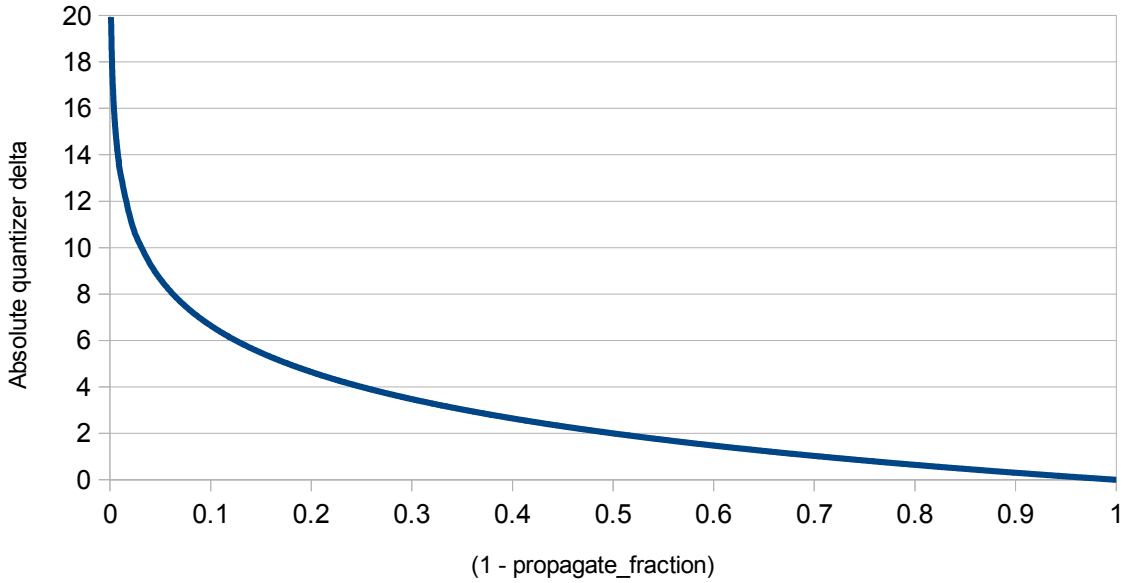
Though macroblock-tree is too complex to allow a closed-form general solution, some insight can be derived from a solution for a special case input: all-zero motion vectors, no B-frames, constant *intra_cost*, and constant *propagate_fraction*. Though such an input is entirely unrealistic, it gives insight into the way that macroblock-tree scales as a function of *propagate_cost*.

We define macroblock-tree as follows:

Let TREE[N] be the propagate_cost after propagating through N frames. Let Y be the constant *intra_cost* and Z be the constant *propagate_fraction,* range 0-1.

$$TREE[0] = 0$$
$$TREE[N] = (TREE[N-1] + Y)*Z$$

This can trivially be shown to converge to Y * (Z/(1-Z)) for large N. Accordingly, the quantizer delta of macroblock-tree in this case scales as follows:



7. Perceptual considerations

As macroblock-tree redistributes bits both within frames and across the frames of a video, it is particularly likely to have perceptual consequences, both positive and negative. There are two primary categories of these impacts that we have noticed during visual comparisons: motion-adaptive quantization and pre-echo.

Decreased visual quality in higher-motion sections of a video is the perceptual interpretation of qcompress and other similar algorithms. Macroblock-tree performs the same role, except localized within each frame. Thus, unlike qcompress, macroblock-tree will not lower the quality of static sections of a frame merely because other portions of the frame are temporally complex. This is particularly important in the case of static backgrounds and overlaid graphics. In this sense, macroblock-tree is a motion-adaptive quantization algorithm focused on coding efficiency.

“Pre-echo” is a natural consequence of the design of macroblock-tree. A macroblock's quality depends on how much it is referenced in the future; if that macroblock will soon be occluded (as in the case of a moving object) or completely replaced (as in the case of a scene change), macroblock-tree will reduce its quality.

Typically this “pre-echo” is only visible in the couple frames immediately prior to the occlusion/scene change and is almost entirely hidden by inter prediction. Furthermore, prior work by Lee et al suggests that scene changes cause a backwards temporal masking effect that helps hide such artifacts.[10] This backwards temporal masking effect lasts a few tens of milliseconds, enough to cover one or two frames, and is believed to be caused by the processing delay in the human visual system.[11] This agrees with our own informal visual testing of macroblock-tree, which also suggests that such artifacts are visually invisible. As such, the bits saved in such macroblocks are free to be used elsewhere in the video.

There is one particular case where pre-echo is visible: that of forced keyframes. The naïve macroblock-tree

algorithm treats keyframes as all-intra frames, even if the keyframes are not scene changes. This causes the few frames prior to the keyframe to be of reduced quality, which can be visually noticeable as a tiny pulse in quality in the case that the keyframe is not actually a scene change.

In x264, this problem is avoided by treating forced keyframes as P-frames for the purpose of macroblock-tree. Do note that this optimization is only performed in x264 when perceptual optimizations are on, and thus does not apply to the results below. The omission of this perceptual optimization likely has a small positive effect on PSNR and SSIM.

8. Quality results

All tests were performed with r1924 (git hash 08d04a4d30b452faed3b763528611737d994b30b) of x264 [12].

All tests were performed with the "slow" preset in 1-pass constant quality mode. This uses a 50-frame lookahead as well as Viterbi adaptive B-frame placement. Options used:

All tests:	*--preset slow --crf q* (q is varied from 1 to 51 to create the curves)
PSNR tests:	*--tune psnr –psnr*
SSIM tests:	*--tune ssim –ssim*
mb-tree strength 1:	*--qcomp 0.8*
mb-tree strength 2:	*--qcomp 0.6*
mb-tree strength 3:	*--qcomp 0.4*
qcompress:	*--no-mbtree*
nominal constant qp:	*--no-mbtree --qcomp 1*

In addition, all inputs were set to 25fps before being passed to x264 to eliminate the effects of x264's psychovisual optimizations based on framerate. The bitrates shown below were calculated based on the real framerate of the video.

The tests here can be divided into three categories based on their purpose. The first category is that from which the least gains are expected: short standard test clips. These have very little variation in content. The more variation in complexity within a video, the more quality there is to gain from redistributing bits throughout the video. Thus, standard test sequences typically gain less from macroblock-tree than real-world content.

The second category of tests are those intended to represent real-world content. These are much longer videos with dramatically varying complexity from scene to scene, allowing the full benefit of macroblock-tree to appear. The third category of tests are special-purpose videos that illustrate particular types of content where macroblock-tree gives unusually large benefits. All graphs use log bitrate scales.

For the first category of test, both PSNR and SSIM results are provided.

Average PSNR scores tend to be less meaningful for videos with dramatically varying complexity. Accordingly, for the second category of tests, OPSNR has been used instead. Average PSNR is also included for reference. “Mix”, the first test, is a concatenation of Mobile, Akiyo, Husky, and Foreman, respectively. The high improvement on this video in comparison to the individual test clips demonstrates the effect of the temporal bit distribution properties of macroblock-tree.

The last two videos represent some types of content that gain an unusual amount from macroblock-tree. The first is a section from an anime DVD, with a raw YUV md5 hash of b473157131044a818914ceefba148271. The second is footage from a video game, with a raw YUV md5 hash of 33cec0cc3a68be633cd04e83d89e0977.

9. Performance results

In addition to testing quality, we also tested the performance of macroblock-tree. The test was performed on a 1.866Ghz Core i7 system running x86_64 Gentoo Linux using gcc 4.6 on foreman CIF. To make the results more consistent, a single thread was used. The options used were *--threads 1 --quiet --preset slow*.

Due to the effect of bitrate on compression speed, the quality level was set for the no-mb-tree test such that the resulting bitrate between the two encodes was as close to identical as possible: 456.99kbps vs 457.16kbps with B-frames and 456.96kbps vs 456.84kbps without B-frames.

	Original	Macroblock-tree	Change
With B-frames	29.770 +/- 0.018 fps	30.293 +/- 0.015 fps	1.80%
Without B-frames	29.363 +/- 0.018 fps	28.705 +/- 0.011 fps	-2.20%

With B-frames enabled, macroblock-tree actually improves overall performance: this is due to the fact that it tends to use higher quantizers on B-frames (as explained in section 5), resulting in faster encoding of B-frames. If we remove this effect by disabling B-frames, the total performance impact of macroblock-tree is about 2.2%. This clearly demonstrates that the computational cost of macroblock-tree is small enough for practical real-time operation.

This cost could be further reduced at the cost of some accuracy by only re-running macroblock-tree for every dozen or so frames, instead of for each frame, and caching results from previous runs. The lookahead size could also be reduced (from 50 frames) for a linear complexity improvement. The cost of macroblock-tree in our implementation is approximately 28 clock cycles per macroblock per lookahead frame. Even with a lookahead of size 50 frames, this is less than half the cost of a single rate-distortion mode analysis in x264, making macroblock-tree's computational complexity practically negligible.

In a naïve implementation, the division in step 2 (see section 4) is by far the most costly portion of the algorithm. On a typical x86 CPU, an integer division takes approximately 40 cycles, and no SIMD integer division operation exists on most architectures. Using floating point division improved performance (allowing the use of *divps*) while providing sufficient accuracy for the algorithm. The final implementation used two iterations of Newton's Method to avoid *divps* altogether, further improving performance at negligible cost in accuracy.

10. Conclusions

Macroblock-tree clearly achieves its goal, providing a large compression improvement at low computational cost, both when compared to the naïve constant quantizer algorithm and qcomp. As can be seen in section 9, macroblock-tree tends to improve SSIM significantly more than PSNR. As SSIM is believed to be a significantly more perceptually accurate metric than PSNR[13], this suggests that macroblock-tree is particularly beneficial perceptually.

It should be noted that macroblock-tree requires a significant lookahead, making it appear to be infeasible in low-latency applications. However, it is possible to implement macroblock-tree without a lookahead, by propagating the *propagate_cost* from past frames to the current frame (i.e. in reverse). This is not as effective as a future-based macroblock-tree, and has the problem of “resetting” its history on every scenecut, but may be applicable in some applications. This has also been implemented in x264, though a full analysis is outside the scope of this paper.

Future improvements to macroblock-tree may want to consider better methods of modeling information propagation than a mere ratio of inter and intra cost. Additionally, it might benefit perceptual quality to take into account temporal effects of perceptual masking, to utilize the relationship between how long an object stays on the screen and its perceptual value. Macroblock-tree could also be improved by making the lookahead a more accurate approximation of a real encode (e.g. by adding multiple reference frame support or partition support) or running it on data generated by a real encode. This would reduce performance, but could improve compression.

1. Sullivan, G; Wiegand, T.; , "Rate-Distortion Optimization for Video Compression," *IEEE Signal Processing Magazine*, vol. 15, pp.74-90, 1998

2. Ramchandran, K.; Ortega, A.; Vetterli, M.; , "Bit allocation for dependent quantization with applications to multiresolution and MPEG video coders," *Image Processing, IEEE Transactions on Image Processing*, vol.3, no.5, pp.533-545, Sep 1994

3. Sermadevi, Y.; Hemami, S.S.; , "Efficient bit allocation for dependent video coding," *Data Compression Coference, 2004. Proceedings. DCC 2004* , vol., no., pp. 232- 241, 23-25 March 2004

4. Toivonen, T.; Merritt, L.; Ojansivu, V.; Heikkila, J.; , "A New Rotation Search for Dependent Rate-Distortion Optimization in Video Coding," *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on* , vol.1, no., pp.I-1165-I-1168, 15-20 April 2007

5. Merritt, L.; Vanam, R.; , "x264: a high performance H.264/AVC encoder,"

6. http://akuvian.org/src/x264/x264_rdrc.5.diff

7. x264, available at http://www.videolan.org/developers/x264.html

8. Ce Zhu; Xiao Lin; Lap-Pui Chau; , "Hexagon-based search pattern for fast block motion estimation," *Circuits and Systems for Video Technology, IEEE Transactions on* , vol.12, no.5, pp.349-355, May 2002

9. Schmuitsch Brad; Schwarz Heiko; Wiegand Thomas; , “Optimization of transform coefficient selection and motion vector estimation considering inter-picture dependencies in hybrid video coding,” *Proceedings of SPIE*, 2005

10. Jungwoo Lee; Dickinson, B.W.; , "Temporally adaptive motion interpolation exploiting temporal masking in visual perception," *Image Processing, IEEE Transactions on* , vol.3, no.5, pp.513-526, Sep 1994

11. Jungwoo Lee; Dickinson, B.W.; , "Subband video coding with scene-adaptive hierarchical motion estimation," *Circuits and Systems for Video Technology, IEEE Transactions on* , vol.9, no.3, pp.459-466, Apr 1999

12. http://git.videolan.org/?p=x264.git;a=commit;h=08d04a4d30b452faed3b763528611737d994b30b

13. Zhou Wang; Bovik, A.C.; Sheikh, H.R.; Simoncelli, E.P.; , "Image quality assessment: from error visibility to structural similarity," *Image Processing, IEEE Transactions on Image Processing*, vol.13, no.4, pp.600-612, April 2004