netvc
Internet-Draft
Intended status: Informational
Expires: October 26, 2017

T. Terriberry
N. Egge
Mozilla Corporation

April 24, 2017

Coding Tools for a Next Generation Video Codec
draft-terriberry-netvc-codingtools-02

Abstract

This document proposes a number of coding tools that could be incorporated into a next-generation video codec.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 26, 2017.

Table of Contents

1. Introduction

One of the biggest contributing factors to the success of the Internet is that the underlying protocols are implementable on a royalty-free basis. This allows them to be implemented widely and easily distributed by application developers, service operators, and end users, without asking for permission. In order to produce a next-generation video codec that is competitive with the best patent-encumbered standards, yet avoids patents which are not available on an open-source compatible, royalty-free basis, we must use old coding tools in new ways and develop new coding tools. This draft documents some of the tools we have been working on for inclusion in such a codec. This is early work, and the performance of some of these tools (especially in relation to other approaches) is not yet fully known. Nevertheless, it still serves to outline some possibilities that NETVC could consider.

2. Entropy Coding

The basic theory of entropy coding was well-established by the late 1970's [Pas76]. Modern video codecs have focused on Huffman codes (or "Variable-Length Codes"/VLCs) and binary arithmetic coding.

Huffman codes are limited in the amount of compression they can provide and the design flexibility they allow, but as each code word consists of an integer number of bits, their implementation complexity is very low, so they were provided at least as an option in every video codec up through H.264. Arithmetic coding, on the other hand, uses code words that can take up fractional parts of a bit, and are more complex to implement. However, the prevalence of cheap, H.264 High Profile hardware, which requires support for arithmetic coding, shows that it is no longer so expensive that a fallback VLC-based approach is required. Having a single entropy-coding method simplifies both up-front design costs and interoperability.

However, the primary limitation of arithmetic coding is that it is an inherently serial operation. A given symbol cannot be decoded until the previous symbol is decoded, because the bits (if any) that are output depend on the exact state of the decoder at the time it is decoded. This means that a hardware implementation must run at a sufficiently high clock rate to be able to decode all of the symbols in a frame. Higher clock rates lead to increased power consumption, and in some cases the entropy coding is actually becoming the limiting factor in these designs.

As fabrication processes improve, implementers are very willing to trade increased gate count for lower clock speeds. So far, most approaches to allowing parallel entropy coding have focused on splitting the encoded symbols into multiple streams that can be decoded independently. This "independence" requirement has a non-negligible impact on compression, parallelizability, or both. For example, H.264 can split frames into "slices" which might cover only a small subset of the blocks in the frame. In order to allow decoding these slices independently, they cannot use context information from blocks in other slices (harming compression). Those contexts must adapt rapidly to account for the generally small number of symbols available for learning probabilities (also harming compression). In some cases the number of contexts must be reduced to ensure enough symbols are coded in each context to usefully learn probabilities at all (once more, harming compression). Furthermore, an encoder must specially format the stream to use multiple slices per frame to allow any parallel entropy decoding at all. Encoders rarely have enough information to evaluate this "compression efficiency" vs. "parallelizability" trade-off, since they don't generally know the limitations of the decoders for which they are encoding. That means there will be many files or streams which could have been decoded if they were encoded with different options, but which a given decoder cannot decode because of bad choices made by the encoder (at least from the perspective of that decoder). The

same set of drawbacks apply to the DCT token partitions in VP8 [RFC6386].

2.1. Non-binary Arithmetic Coding

Instead, we propose a very different approach: use non-binary arithmetic coding. In binary arithmetic coding, each decoded symbol has one of two possible values: 0 or 1. The original arithmetic coding algorithms allow a symbol to take on any number of possible values, and allow the size of that alphabet to change with each symbol coded. Reasonable values of N (for example, N <= 16) offer the potential for a decent throughput increase for a reasonable increase in gate count for hardware implementations.

Binary coding allows a number of computational simplifications. For example, for each coded symbol, the set of valid code points is partitioned in two, and the decoded value is determined by finding the partition in which the actual code point that was received lies. This can be determined by computing a single partition value (in both the encoder and decoder) and (in the decoder) doing a single comparison. A non-binary arithmetic coder partitions the set of valid code points into multiple pieces (one for each possible value of the coded symbol). This requires the encoder to compute two partition values, in general (for both the upper and lower bound of the symbol to encode). The decoder, on the other hand, must search the partitions for the one that contains the received code point. This requires computing at least O(log N) partition values.

However, coding a parameter with N possible values with a binary arithmetic coder requires O(log N) symbols in the worst case (the only case that matters for hardware design). Hence, this does not represent any actual savings (indeed, it represents an increase in the number of partition values computed by the encoder). In addition, there are a number of overheads that are per-symbol, rather than per-value. For example, renormalization (which enlarges the set of valid code points after partitioning has reduced it too much), carry propagation (to deal with the case where the high and low ends of a partition straddle a bit boundary), etc., are all performed on a symbol-by-symbol basis. Since a non-binary arithmetic coder codes a given set of values with fewer symbols than a binary one, it incurs these per-symbol overheads less often. This suggests that a non-binary arithmetic coder can actually be more efficient than a binary one.

2.2. Non-binary Context Modeling

The other aspect that binary coding simplifies is probability modeling. In arithmetic coding, the size of the sets the code points are partitioned into are (roughly) proportional to the probability of each possible symbol value. Estimating these probabilities is part of the coding process, though it can be cleanly separated from the task of actually producing the coded bits. In a binary arithmetic coder, this requires estimating the probability of only one of the two possible values (since the total probability is 1.0). This is often done with a simple table lookup that maps the old probability and the most recently decoded symbol to a new probability to use for the next symbol in the current context. The trade-off, of course, is that non-binary symbols must be "binarized" into a series of bits, and a context (with an associated probability) chosen for each one.

In a non-binary arithmetic coder, the decoder must compute at least O(log N) cumulative probabilities (one for each partition value it needs). Because these probabilities are usually not estimated directly in "cumulative" form, this can require computing (N - 1) non-cumulative probability values. Unless N is very small, these cannot be updated with a single table lookup. The normal approach is to use "frequency counts". Define the frequency of value k to be

f[k] = A*<the number of times k has been observed> + B

where A and B are parameters (usually A=2 and B=1 for a traditional Krichevsky-Trofimov estimator). The resulting probability, p[k], is given by

$$ft = \sum_{k=0}^{N-1} f[k]$$

$$p[k] = \frac{f[k]}{ft}$$

When ft grows too large, the frequencies are rescaled (e.g., halved, rounding up to prevent reduction of a probability to 0).

When ft is not a power of two, partitioning the code points requires actual divisions (see [RFC6716] Section 4.1 for one detailed example of exactly how this is done). These divisions are acceptable in an audio codec like Opus [RFC6716], which only has to code a few hundreds of these symbols per second. But video requires hundreds of

thousands of symbols per second, at a minimum, and divisions are still very expensive to implement in hardware.

There are two possible approaches to this. One is to come up with a replacement for frequency counts that produces probabilities that sum to a power of two. Some possibilities, which can be applied individually or in combination:

1. Use probabilities that are fixed for the duration of a frame. This is the approach taken by VP8, for example, even though it uses a binary arithmetic coder. In fact, it is possible to convert many of VP8's existing binary-alphabet probabilities into probabilities for non-binary alphabets, an approach that is used in the experiment presented at the end of this section.

2. Use parametric distributions. For example, DCT coefficient magnitudes usually have an approximately exponential distribution. This distribution can be characterized by a single parameter, e.g., the expected value. The expected value is trivial to update after decoding a coefficient. For example

   E[x[n+1]] = E[x[n]] + floor(C*(x[n] - E[x[n]]))

   produces an exponential moving average with a decay factor of (1 - C). For a choice of C that is a negative power of two (e.g., 1/16 or 1/32 or similar), this can be implemented with two adds and a shift. Given this expected value, the actual distribution to use can be obtained from a small set of pre-computed distributions via a lookup table. Linear interpolation between these pre-computed values can improve accuracy, at the cost of O(N) computations, but if N is kept small this is trivially parallelizable, in SIMD or otherwise.

3. Change the frequency count update mechanism so that ft is constant. This approach is described in the next section.

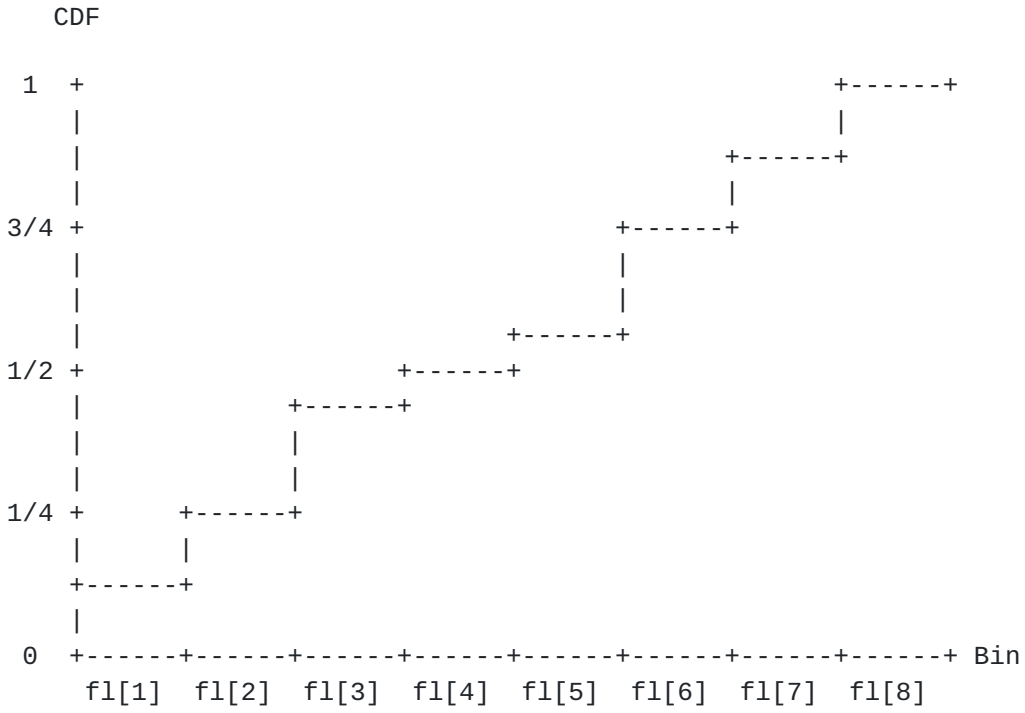
2.3. Dyadic Adaptation

The goal with context adaptation using dyadic probabilities is to maintain the invariant that the probabilities all sum to a power of two before and after adaptation. This can be achieved with a special update function that blends the cumulative probabilities of the current context with a cumulative distribution function where the coded symbol has probability 1.

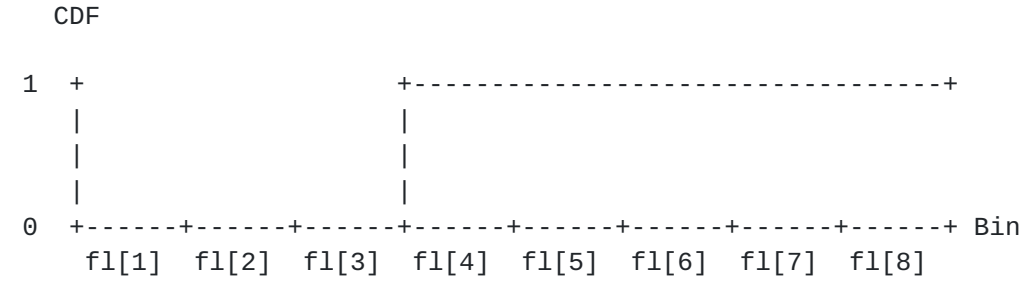
Suppose we have model for a given context that codes 8 symbols with the following probabilities:

```
+------+------+------+------+------+------+------+------+
| p[0] | p[1] | p[2] | p[3] | p[4] | p[5] | p[6] | p[7] |
+------+------+------+------+------+------+------+------+
|  1/8 |  1/8 | 3/16 | 1/16 | 1/16 | 3/16 |  1/8 |  1/8 |
+------+------+------+------+------+------+------+------+
```

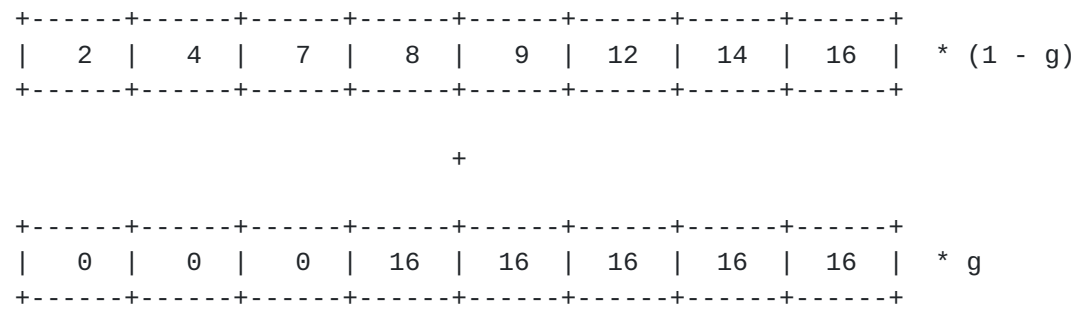
Then the cumulative distribution function is:



Suppose we code symbol 3 and wish to update the context model so that this symbol is now more likely. This can be done by blending the CDF for the current context with a CDF that has symbol 3 with likelihood 1.



Given an adaptation rate g between 0 and 1, and assuming ft = 2^4 = 16, what we are computing is:

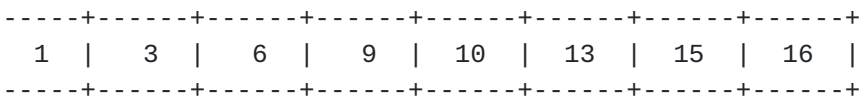


In order to prevent the probability of any one symbol from going to zero, the blending functions above and below the coded symbol are adjusted so that no adjacent cumulative probabilities are the same.

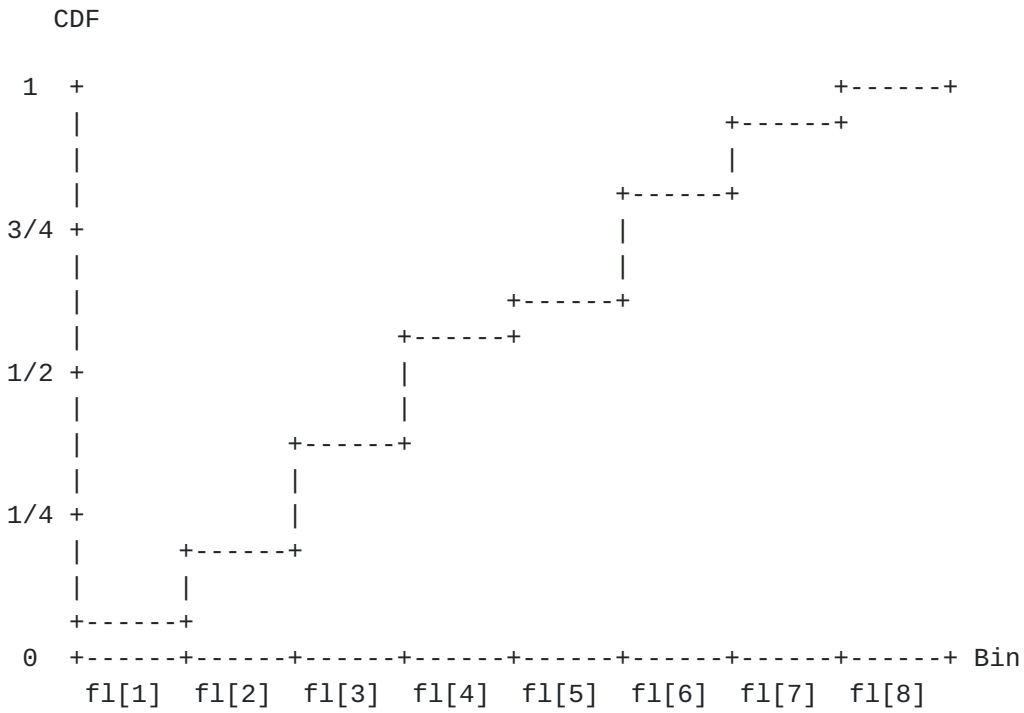
Let M be the alphabet size and 1/2^r be the adaptation rate:

```
        ( fl[i] - floor((fl[i] + 2^r - i - 1)/2^r), i <= coded symbol
fl[i] = <
        ( fl[i] - floor((fl[i] + M - i - ft)/2^r),  i > coded symbol
```

Applying these formulas to the example CDF where M = 8 with adaptation rate 1/2^16 gives the updated CDF:



Looking at the graph of the CDF we see that the likelihood for symbol 3 has gone up from 1/16 to 3/16, dropping the likelihood of all other symbols to make room.



2.4. Simplified Partition Function

Let the range of valid code points in the current arithmetic coder state be [L, L + R), where L is the lower bound of the range and R is the number of valid code points. The goal of the arithmetic coder is to partition this interval proportional to the probability of each symbol. When using dyadic probabilities, the partition point in the range corresponding to a given CDF value can be determined via

$$u[k] = \text{floor}\left(\frac{fl[k]*R}{ft}\right)$$

Since ft is a power of two, this may be implemented using a right shift by T bits in place of the division:

$$u[k] = (fl[k]*R) >> T$$

The latency of the multiply still dominates the hardware timing. However, we can reduce this latency by using a smaller multiply, at the cost of some accuracy in the partition. We cannot, in general, reduce the size of fl[k], since this might send a probability to zero (i.e., cause u[k] to have the same value as u[k+1]). On the other hand, we know that the top bit of R is always 1, since it gets renormalized with every symbol that is encoded. Suppose R contains 16 bits and that T is at least 8. Then we can greatly reduce the size of the multiply by using the formula

```
                ( (fl[k]*(R >> 8)) >> (T - 8), 0 <= k < M
        u[k] = <
                ( R,                           k == M
```

The special case for k == M is required because, with the general formula, u[M] no longer exactly equals R. Without the special case we would waste some amount of code space and require the decoder to check for invalid streams. This special case slightly inflates the probability of the last symbol. Unfortunately, in codecs the usual convention is that the last symbol is the least probable, while the first symbol (e.g., 0) is the most probable. That maximizes the coding overhead introduced by this approximation error. To minimize it, we instead add all of the accumulated error to the first symbol by using a variation of the above update formula:

```
          ( 0,                                        k == 0
    u[k] = <
          ( R - (((ft - fl[k])*(R >> 8)) >> (T - 8)), 0 < k <= M
```

This also aids the software decoder search, since it can prime the search loop with the special case, instead of needing to check for it on every iteration of the loop. It is easier to incorporate into a SIMD search as well. It does, however, add two subtractions. Since the encoder always operates on the difference between two partition points, the first subtraction (involving R) can be eliminated. Similar optimizations can eliminate this subtraction in the decoder by flipping its internal state (measuring the distance of the encoder output from the top of the range instead of the bottom). To avoid the other subtraction, we can simply use "inverse CDFs" that natively store ifl[k] = (ft - fl[k]) instead of fl[k]. This produces the following partition function:

```
                  ( R,                             k == 0
       R - u[k] = <
                  ( (ifl[k]*(R >> 8)) >> (T - 8), 0 < k <= M
```

The reduction in hardware latency can be as much as 20%, and the impact on area is even larger. The overall software complexity overhead is minimal, and the coding efficiency overhead due to the approximation is about 0.02%. We could have achieved the same efficiency by leaving the special case on the last symbol and reversing the alphabet instead of inverting the probabilities. However, reversing the alphabet at runtime would have required an extra subtraction (or more general re-ordering requires a table lookup). That may be avoidable in some cases, but only by propagating the reordering alphabet outside of the entropy coding machinery, requiring changes to every coding tool and potentially leading to confusion. CDFs, on the other hand, are already a

somewhat abstract representation of the underlying probabilities used for computational efficiency reasons. Generalizing these to "inverse CDFs" is a straightforward change that only affects probability initialization and adaptation, without impacting the design of other coding tools.

2.5. Context Adaptation

The dyadic adaptation scheme described in Section 2.3 implements a low-complexity IIR filter for the steady-state case where we only want to adapt the context CDF as fast as the 1/2^r adaptation rate. In many cases, for example when coding symbols at the start of a video frame, only a limited number of symbols have been seen per context. Using this steady-state adaptation scheme risks adapting too slowly and spending too many bits to code symbols with incorrect probability estimates. In other video codecs, this problem is reduced by either implicitly or explicitly allowing for mechanisms to set the initial probability models for a given context.

2.5.1. Implicit Adaptation

One implicit way to use default probabilities is to simply require as a normative part of the decoder that some specific CDFs are used to initialize each context. A representative set of inputs is run through the encoder and a frequency based probability model is computed and reloaded at the start of every frame. This has the advantage of having zero bitstream overhead and is optimal for certain stationary symbols. However for other non-stationary symbols, or highly content dependent contexts where the sample input is not representative, this can be worse than starting with a flat distribution as it now takes even longer to adapt to the steady-state. Moreover the amount of hardware area required to store initial probability tables for each context goes up with the number of contexts in the codec.

Another implicit way to deal with poor initial probabilities is through backward adaptation based on the probability estimates from the previous frame. After decoding a frame, the adapted CDFs for each context are simply kept as-is and not reset to their defaults. This has the advantage of having no bitstream overhead, and tracking to certain content types closely as we expect frames with similar content at similar rates, to have well correlated CDFs. However, this only works when we know there will be no bitstream errors due to the transport layer, e.g., TCP or HTTP. In low delay use cases (video on demand, live streaming, video conferencing), implicit backwards adaptation is avoided as it risks desynchronizing the entropy decoder state and permanently losing the video stream.

2.5.2. Explicit Adaptation

For codecs that include the ability to update the probability models in the bitstream, it is possible to explicitly signal a starting CDF. The previously described implicit backwards adaptation is now possible by simply explicitly coding a probability update for each frame. However, the cost of signaling the updated CDF must be overcome by the savings from coding with the updated CDF. Blindly updating all contexts per frame may work at high rates where the size of the CDFs is small relative to the coded symbol data. However at low rates, the benefit of using more accurate CDFs is quickly overcome by the cost of coding them, which increases with the number of contexts.

More sophisticated encoders can compute the cost of coding a probability update for a given context, and compare it to the size reduction achieved by coding symbols with this context. Here all symbols for a given frame (or tile) are buffered and not serialized by the entropy coder until the end of the frame (or tile) is reached. Once the end of the entropy segment has been reached, the cost in bits for coding symbols with both the default probabilities and the proposed updated probabilities can be measured and compared. However, note that with the symbols already buffered, rather than consider the context probabilities from the previous frame, a simple frequency based probability model can be computed and measured. Because this probability model is computed based on the symbols we are about to code this technique is called forward adaptation. If the cost in bits to signal and code with this new probability model is less than that of using the default then it is used. This has the advantage of only ever coding a probability update if it is an improvement and producing a bitstream that is robust to errors, but requires an entire entropy segments worth of symbols be cached.

2.5.3. Early Adaptation

We would like to take advantage of the low-cost multi-symbol CDF adaptation described in Section 2.3 without in the broadest set of use cases. This means the initial probability adaptation scheme should support low-delay, error-resilient streams that efficiently implemented in both hardware and software. We propose an early adaptation scheme that supports this goal.

At the beginning of a frame (or tile), all CDFs are initialized to a flat distribution. For a given multi-symbol context with M potential symbols, assume that the initial dyadic CDF is initialized so that each symbol has probability 1/M. For the first M coded symbols, the CDF is updated as follows:

```
a[c,M] = ft/(M + c)

        ( fl[i] - floor((fl[i] - i)*a/ft),          i <= coded symbol
fl[i] = <
        ( fl[i] - floor((fl[i] + M - i - ft)*a/ft), i > coded symbol
```

where c goes from 0 to M-1 and is the running count of the number of symbols coded with this CDF. Note that for a fixed CDF precision (ft is always a power of two) and a maximum number of possible symbols M, the values of a[c,M] can be stored in a M*(M+1)/2 element table, which is 136 entries when M = 16.

2.6. Simple Experiment

As a simple experiment to validate the non-binary approach, we compared a non-binary arithmetic coder to the VP8 (binary) entropy coder. This was done by instrumenting vp8_treed_read() in libvpx to dump out the symbol decoded and the associated probabilities used to decode it. This data only includes macroblock mode and motion vector information, as the DCT token data is decoded with custom inline functions, and not vp8_treed_read(). This data is available at [1]. It includes 1,019,670 values encode using 2,125,995 binary symbols (or 2.08 symbols per value). We expect that with a conscious effort to group symbols during the codec design, this average could easily be increased.

We then implemented both the regular VP8 entropy decoder (in plain C, using all of the optimizations available in libvpx at the time) and a multisymbol entropy decoder (also in plain C, using similar optimizations), which encodes each value with a single symbol. For the decoder partition search in the non-binary decoder, we used a simple for loop (O(N) worst-case), even though this could be made constant-time and branchless with a few SIMD instructions such as (on x86) PCMPGTW, PACKUSWB, and PMOVMASKB followed by BSR. The source code for both implementations is available at [2] (compile with -DEC_BINARY for the binary version and -DEC_MULTISYM for the non-binary version).

The test simply loads the tokens, and then loops 1024 times encoding them using the probabilities provided, and then decoding them. The loop was added to reduce the impact of the overhead of loading the data, which is implemented very inefficiently. The total runtime on a Core i7 from 2010 is 53.735 seconds for the binary version, and 27.937 seconds for the non-binary version, or a 1.92x improvement. This is very nearly equal to the number of symbols per value in the binary coder, suggesting that the per-symbol overheads account for the vast majority of the computation time in this implementation.

3. Reversible Integer Transforms

Integer transforms in image and video coding date back to at least 1969 [PKA69]. Although standards such as MPEG2 and MPEG4 Part 2 allow some flexibility in the transform implementation, implementations were subject to drift and error accumulation, and encoders had to impose special macroblock refresh requirements to avoid these problems, not always successfully. As transforms in modern codecs only account for on the order of 10% of the total decoder complexity, and, with the use of weighted prediction with gains greater than unity and intra prediction, are far more susceptible to drift and error accumulation, it no longer makes sense to allow a non-exact transform specification.

However, it is also possible to make such transforms "reversible", in the sense that applying the inverse transform to the result of the forward transform gives back the original input values, exactly. This gives a lossy codec, which normally quantizes the coefficients before feeding them into the inverse transform, the ability to scale all the way to lossless compression without requiring any new coding tools. This approach has been used successfully by JPEG XR, for example [TSSRM08].

Such reversible transforms can be constructed using "lifting steps", a series of shear operations that can represent any set of plane rotations, and thus any orthogonal transform. This approach dates back to at least 1992 [BE92], which used it to implement a four-point 1-D Discrete Cosine Transform (DCT). Their implementation requires 6 multiplications, 10 additions, 2 shifts, and 2 negations, and produces output that is a factor of sqrt(2) larger than the orthonormal version of the transform. The expansion of the dynamic range directly translates into more bits to code for lossless compression. Because the least significant bits are usually very nearly random noise, this scaling increases the coding cost by approximately half a bit per sample.

3.1. Lifting Steps

To demonstrate the idea of lifting steps, consider the two-point transform

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

This can be implemented up to scale via

```
y0 = x0 + x1

y1 = 2*x1 - y0
```

and reversed via

```
x1 = (y0 + y1) >> 1

x0 = y0 - x1
```

Both y0 and y1 are too large by a factor of sqrt(2), however.

It is also possible to implement any rotation by an angle t, including the orthonormal scale factor, by decomposing it into three steps:

$$u0 = x0 + \frac{\cos(t) - 1}{\sin(t)} * x1$$

$$y1 = x1 + \sin(t)*u0$$

$$y0 = u0 + \frac{\cos(t) - 1}{\sin(t)} * y1$$

By letting t=-pi/4, we get an implementation of the first transform that includes the scaling factor. To get an integer approximation of this transform, we need only replace the transcendental constants by fixed-point approximations:

```
u0 = x0 + ((27*x1 + 32) >> 6)

y1 = x1 - ((45*u0 + 32) >> 6)

y0 = u0 + ((27*y1 + 32) >> 6)
```

This approximation is still perfectly reversible:

```
u0 = y0 - ((27*y1 + 32) >> 6)

x1 = y1 + ((45*u0 + 32) >> 6)

x0 = u0 - ((27*x1 + 32) >> 6)
```

Each of the three steps can be implemented using just two ARM instructions, with constants that have up to 14 bits of precision (though using fewer bits allows more efficient hardware

implementations, at a small cost in coding gain). However, it is still much more complex than the first approach.

We can get a compromise with a slight modification:

```
y0 = x0 + x1

y1 = x1 - (y0 >> 1)
```

This still only implements the original orthonormal transform up to scale. The y0 coefficient is too large by a factor of sqrt(2) as before, but y1 is now too small by a factor of sqrt(2). If our goal is simply to (optionally quantize) and code the result, this is good enough. The different scale factors can be incorporated into the quantization matrix in the lossy case, and the total expansion is roughly equivalent to that of the orthonormal transform in the lossless case. Plus, we can perform each step with just one ARM instruction.

However, if instead we want to apply additional transformations to the data, or use the result to predict other data, it becomes much more convenient to have uniformly scaled outputs. For a two-point transform, there is little we can do to improve on the three-multiplications approach above. However, for a four-point transform, we can use the last approach and arrange multiple transform stages such that the "too large" and "too small" scaling factors cancel out, producing a result that has the true, uniform, orthonormal scaling. To do this, we need one more tool, which implements the following transform:

```
                ___
[ y0 ]     / 1  [ cos(t) -sin(t) ] [ 1  0 ] [ x0 ]
[    ] =  / --- [                ] [      ] [    ]
[ y1 ]   v   2  [ sin(t)  cos(t) ] [ 0  2 ] [ x1 ]
```

This takes unevenly scaled inputs, rescales them, and then rotates them. Like an ordinary rotation, it can be reduced to three lifting steps:

$$u0 = x0 + \frac{2\cos(t) - \sqrt{v2}}{\sin(t)} * x1$$

$$y1 = x1 + \sqrt{\frac{1}{v} - \frac{1}{2}} * \sin(t) * u0$$

$$y0 = u0 + \frac{\cos(t) - \sqrt{v2}}{\sin(t)} * y1$$

As before, the transcendental constants may be replaced by fixed-point approximations without harming the reversibility property.

3.2. 4-Point Transform

Using the tools from the previous section, we can design a reversible integer four-point DCT approximation with uniform, orthonormal scaling. This requires 3 multiplies, 9 additions, and 2 shifts (not counting the shift and rounding offset used in the fixed-point multiplies, as these are built into the multiplier). This is significantly cheaper than the [BE92] approach, and the output scaling is smaller by a factor of sqrt(2), saving half a bit per sample in the lossless case. By comparison, the four-point forward DCT approximation used in VP9, which is not reversible, uses 6 multiplies, 6 additions, and 2 shifts (counting shifts and rounding offsets which cannot be merged into a single multiply instruction on ARM). Four of its multipliers also require 28-bit accumulators, whereas this proposal can use much smaller multipliers without giving up the reversibility property. The total dynamic range expansion is 1 bit: inputs in the range [-256,255) produce transformed values in the range [-512,510). This is the smallest dynamic range expansion possible for any reversible transform constructed from mostly-linear operations. It is possible to make reversible orthogonal transforms with no dynamic range expansion by using "piecewise-linear" rotations [SLD04], but each step requires a large number of operations in a software implementation.

Pseudo-code for the forward transform follows:

```
Input:  x0, x1, x2, x3
Output: y0, y1, y2, y3
/* Rotate (x3, x0) by -pi/4, asymmetrically scaled output. */
t3  = x0 - x3
t0  = x0 - (t3 >> 1)
/* Rotate (x1, x2) by pi/4, asymmetrically scaled output. */
t2  = x1 + x2
t2h = t2 >> 1
t1  = t2h - x2
/* Rotate (t2, t0) by -pi/4, asymmetrically scaled input. */
y0  = t0 + t2h
y2  = y0 - t2
/* Rotate (t3, t1) by 3*pi/8, asymmetrically scaled input. */
t3  = t3 - (45*t1 + 32 >> 6)
y1  = t1 + (21*t3 + 16 >> 5)
y3  = t3 - (71*y1 + 32 >> 6)
```

Even though there are three asymmetrically scaled rotations by pi/4, by careful arrangement we can share one of the shift operations (to help software implementations: shifts by a constant are basically free in hardware). This technique can be used to even greater effect in larger transforms.

The inverse transform is constructed by simply undoing each step in turn:

```
Input:  y0, y1, y2, y3
Output: x0, x1, x2, x3
/* Rotate (y3, y1) by -3*pi/8, asymmetrically scaled output. */
t3  = y3 + (71*y1 + 32 >> 6)
t1  = y1 - (21*t3 + 16 >> 5)
t3  = t3 + (45*t1 + 32 >> 6)
/* Rotate (y2, y0) by pi/4, asymmetrically scaled output. */
t2  = y0 - y2
t2h = t2 >> 1
t0  = y0 - t2h
/* Rotate (t1, t2) by -pi/4, asymmetrically scaled input. */
x2  = t2h - t1
x1  = t2 - x2
/* Rotate (x3, x0) by pi/4, asymmetrically scaled input. */
x0  = t0 - (t3 >> 1)
x3  = x0 - t3
```

Although the right shifts make this transform non-linear, we can compute "basis functions" for it by sending a vector through it with a single value set to a large constant (256 was used here), and the rest of the values set to zero. The true basis functions for a four-point DCT (up to five digits) are

```
[ y0 ]   [ 0.50000  0.50000  0.50000  0.50000 ] [ x0 ]
[ y1 ] = [ 0.65625  0.26953 -0.26953 -0.65625 ] [ x1 ]
[ y2 ]   [ 0.50000 -0.50000 -0.50000  0.50000 ] [ x2 ]
[ y3 ]   [ 0.27344 -0.65234  0.65234 -0.27344 ] [ x3 ]
```

The corresponding basis functions for our reversible, integer DCT, computed using the approximation described above, are

```
[ y0 ]   [ 0.50000  0.50000  0.50000  0.50000 ] [ x0 ]
[ y1 ] = [ 0.65328  0.27060 -0.27060 -0.65328 ] [ x1 ]
[ y2 ]   [ 0.50000 -0.50000 -0.50000  0.50000 ] [ x2 ]
[ y3 ]   [ 0.27060 -0.65328  0.65328 -0.27060 ] [ x3 ]
```

The mean squared error (MSE) of the output, compared to a true DCT, can be computed with some assumptions about the input signal. Let G be the true DCT basis and G' be the basis for our integer approximation (computed as described above). Then the error in the transformed results is

```
e = G.x - G'.x = (G - G').x = D.x
```

where D = (G - G') . The MSE is then [Que98]

$$\frac{1}{N} * E[e^T.e] = \frac{1}{N} * E[x^T.D^T.D.x]$$

$$= \frac{1}{N} * E[tr(D.x.x^T.D^T)]$$

$$= \frac{1}{N} * E[tr(D.Rxx.D^T)]$$

where Rxx is the autocorrelation matrix of the input signal. Assuming the input is a zero-mean, first-order autoregressive (AR(1)) process gives an autocorrelation matrix of

$$Rxx[i,j] = rho^{|i - j|}$$

for some correlation coefficient rho. A value of rho = 0.95 is typical for image compression applications. Smaller values are more normal for motion-compensated frame differences, but this makes surprisingly little difference in transform design. Using the above procedure, the theoretical MSE of this approximation is 1.230E-6, which is below the level of the truncation error introduced by the

right shift operations. This suggests the dynamic range of the input would have to be more than 20 bits before it became worthwhile to increase the precision of the constants used in the multiplications to improve accuracy, though it may be worth using more precision to reduce bias.

3.3. Larger Transforms

The same techniques can be applied to construct a reversible eight-point DCT approximation with uniform, orthonormal scaling using 15 multiplies, 31 additions, and 5 shifts. It is possible to reduce this to 11 multiplies and 29 additions, which is the minimum number of multiplies possible for an eight-point DCT with uniform scaling [LLM89], by introducing a scaling factor of sqrt(2), but this harms lossless performance. The dynamic range expansion is 1.5 bits (again the smallest possible), and the MSE is 1.592E-06. By comparison, the eight-point transform in VP9 uses 12 multiplications, 32 additions, and 6 shifts.

Similarly, we have constructed a reversible sixteen-point DCT approximation with uniform, orthonormal scaling using 33 multiplies, 83 additions, and 16 shifts. This is just 2 multiplies and 2 additions more than the (non-reversible, non-integer, but uniformly scaled) factorization in [LLM89]. By comparison, the sixteen-point transform in VP9 uses 44 multiplies, 88 additions, and 18 shifts. The dynamic range expansion is only 2 bits (again the smallest possible), and the MSE is 1.495E-5.

We also have a reversible 32-point DCT approximation with uniform, orthonormal scaling using 87 multiplies, 215 additions, and 38 shifts. By comparison, the 32-point transform in VP9 uses 116 multiplies, 194 additions, and 66 shifts. Our dynamic range expansion is still the minimal 2.5 bits, and the MSE is 8.006E-05

Code for all of these transforms is available in the development repository listed in Section 4.

3.4. Walsh-Hadamard Transforms

These techniques can also be applied to constructing Walsh-Hadamard Transforms, another useful transform family that is cheaper to implement than the DCT (since it requires no multiplications at all). The WHT has many applications as a cheap way to approximately change the time and frequency resolution of a set of data (either individual bands, as in the Opus audio codec, or whole blocks). VP9 uses it as a reversible transform with uniform, orthonormal scaling for lossless coding in place of its DCT, which does not have these properties.

Applying a 2x2 WHT to a block of 2x2 inputs involves running a 2-point WHT on the rows, and then another 2-point WHT on the columns. The basis functions for the 2-point WHT are, up to scaling, [1, 1] and [1, -1]. The four variations of a two-step lifer given in Section 3.1 are exactly the lifting steps needed to implement a 2x2 WHT: two stages that produce asymmetrically scaled outputs followed by two stages that consume asymmetrically scaled inputs.

```
Input:  x00, x01, x10, x11
Output: y00, y01, y10, y11
/* Transform rows */
t1 = x00 - x01
t0 = x00 - (t1 >> 1) /* == (x00 + x01)/2 */
t2 = x10 + x11
t3 = (t2 >> 1) - x11 /* == (x10 - x11)/2 */
/* Transform columns */
y00 = t0 + (t2 >> 1) /* == (x00 + x01 + x10 + x11)/2 */
y10 = y00 - t2       /* == (x00 + x01 - x10 - x11)/2 */
y11 = (t1 >> 1) - t3 /* == (x00 - x01 - x10 + x11)/2 */
y01 = t1 - y11       /* == (x00 - x01 + x10 - x11)/2 */
```

By simply re-ordering the operations, we can see that there are two shifts that may be shared between the two stages:

```
Input:  x00, x01, x10, x11
Output: y00, y01, y10, y11
t1 = x00 - x01
t2 = x10 + x11
t0 = x00 - (t1 >> 1) /* == (x00 + x01)/2 */
y00 = t0 + (t2 >> 1) /* == (x00 + x01 + x10 + x11)/2 */
t3 = (t2 >> 1) - x11 /* == (x10 - x11)/2 */
y11 = (t1 >> 1) - t3 /* == (x00 - x01 - x10 + x11)/2 */
y10 = y00 - t2       /* == (x00 + x01 - x10 - x11)/2 */
y01 = t1 - y11       /* == (x00 - x01 + x10 - x11)/2 */
```

By eliminating the double-negation of x11 and re-ordering the additions to it, we can see even more operations in common:

```
Input:  x00, x01, x10, x11
Output: y00, y01, y10, y11
t1 = x00 - x01
t2 = x10 + x11
t0 = x00 - (t1 >> 1) /* == (x00 + x01)/2 */
y00 = t0 + (t2 >> 1) /* == (x00 + x01 + x10 + x11)/2 */
t3 = x11 + (t1 >> 1) /* == x11 + (x00 - x01)/2 */
y11 = t3 - (t2 >> 1) /* == (x00 - x01 - x10 + x11)/2 */
y10 = y00 - t2       /* == (x00 + x01 - x10 - x11)/2 */
y01 = t1 - y11       /* == (x00 - x01 + x10 - x11)/2 */
```

Simplifying further, the whole transform may be computed with just 7 additions and 1 shift:

```
Input:  x00, x01, x10, x11
Output: y00, y01, y10, y11
t1 = x00 - x01
t2 = x10 + x11
t4 = (t2 - t1) >> 1 /* == (-x00 + x01 + x10 + x11)/2 */
y00 = x00 + t4      /* ==  (x00 + x01 + x10 + x11)/2 */
y11 = x11 - t4      /* ==  (x00 - x01 - x10 + x11)/2 */
y10 = y00 - t2      /* ==  (x00 + x01 - x10 - x11)/2 */
y01 = t1 - y11      /* ==  (x00 - x01 + x10 - x11)/2 */
```

This is a significant savings over other approaches described in the literature, which require 8 additions, 2 shifts, and 1 negation [FOIK99] (37.5% more operations), or 10 additions, 1 shift, and 2 negations [TSSRM08] (62.5% more operations). The same operations can be applied to compute a 4-point WHT in one dimension. This implementation is used in this way in VP9's lossless mode. Since larger WHTs may be trivially factored into multiple smaller WHTs, the same approach can implement a reversible, orthonormally scaled WHT of any size (2**N)x(2**M), so long as (N + M) is even.

4. Development Repository

The tools presented here were developed as part of Xiph.Org's Daala project. They are available, along with many others in greater and lesser states of maturity, in the Daala git repository at [3]. See [4] for more information.

5. IANA Considerations

This document has no actions for IANA.

6. Acknowledgments

Thanks to Nathan Egge, Gregory Maxwell, and Jean-Marc Valin for their assistance in the implementation and experimentation, and in preparing this draft.

7. References

7.1. Informative References

[RFC6386] Bankoski, J., Koleszar, J., Quillio, L., Salonen, J., Wilkins, P., and Y. Xu, "VP8 Data Format and Decoding Guide", RFC 6386, November 2011.

[RFC6716] Valin, JM., Vos, K., and T. Terriberry, "Definition of the Opus Audio Codec", RFC 6716, September 2012.

[BE92] Bruekers, F. and A. van den Enden, "New Networks for Perfect Inversion and Perfect Reconstruction", IEEE Journal on Selected Areas in Communication 10(1):129--137, January 1992.

[FOIK99] Fukuma, S., Oyama, K., Iwahashi, M., and N. Kambayashi, "Lossless 8-point Fast Discrete Cosine Transform Using Lossless Hadamard Transform", Technical Report The Institute of Electronics, Information, and Communication Engineers of Japan, October 1999.

[LLM89] Loeffler, C., Ligtenberg, A., and G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", Proc. Acoustics, Speech, and Signal Processing (ICASSP'89) vol. 2, pp. 988--991, May 1989.

[Pas76] Pasco, R., "Source Coding Algorithms for Fast Data Compression", Ph.D. Thesis Dept. of Electrical Engineering, Stanford University, May 1976.

[PKA69] Pratt, W., Kane, J., and H. Andrews, "Hadamard Transform Image Coding", Proc. IEEE 57(1):58--68, Jan 1969.

[Que98] de Queiroz, R., "On Unitary Transform Approximations", IEEE Signal Processing Letters 5(2):46--47, Feb 1998.

[SLD04] Senecal, J., Lindstrom, P., and M. Duchaineau, "An Improved N-Bit to N-Bit Reversible Haar-Like Transform", Proc. of the 12th Pacific Conference on Computer Graphics and Applications (PG'04) pp. 371--380, October 2004.

[TSSRM08] Tu, C., Srinivasan, S., Sullivan, G., Regunathan, S., and H. Malvar, "Low-complexity Hierarchical Lapped Transform for Lossy-to-Lossless Image Coding in JPEG XR/HD Photo", Applications of Digital Image Processing XXXI vol 7073, August 2008.

7.2. URIs

[1] https://people.xiph.org/~tterribe/daala/ec_test0/ec_tokens.txt

[2] https://people.xiph.org/~tterribe/daala/ec_test0/ec_test.c

[3] https://git.xiph.org/daala.git

[4] https://xiph.org/daala/

Authors' Addresses

Timothy B. Terriberry
Mozilla Corporation
331 E. Evelyn Avenue
Mountain View, CA 94041
USA

Phone: +1 650 903-0800
Email: tterribe@xiph.org

Nathan E. Egge
Mozilla Corporation
331 E. Evelyn Avenue
Mountain View, CA 94041
USA

Phone: +1 650 903-0800
Email: negge@xiph.org