

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 4, 2017

A. Fuldseth
G. Bjontegaard
S. Midtskogen
T. Davies
M. Zanaty
Cisco
October 31, 2016

Thor Video Codec
draft-fuldseth-netvc-thor-03

Abstract

This document provides a high-level description of the Thor video codec. Thor is designed to achieve high compression efficiency with moderate complexity, using the well-known hybrid video coding approach of motion-compensated prediction and transform coding.

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 May 4, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Definitions	5
2.1. Requirements Language	5
2.2. Terminology	6
3. Block Structure	6
3.1. Super Blocks and Coding Blocks	6
3.2. Special Processing at Frame Boundaries	7
3.3. Transform Blocks	8
3.4. Prediction Blocks	8
4. Intra Prediction	8
5. Inter Prediction	9
5.1. Multiple Reference Frames	9
5.2. Bi-Prediction	10
5.3. Improved chroma prediction	10
5.4. Reordered Frames	10
5.5. Interpolated Reference Frames	10
5.6. Sub-Pixel Interpolation	10
5.6.1. Luma Poly-phase Filter	10
5.6.2. Luma Special Filter Position	12
5.6.3. Chroma Poly-phase Filter	13
5.7. Motion Vector Coding	13
5.7.1. Inter0 and Inter1 Modes	13
5.7.2. Inter2 and Bi-Prediction Modes	15
5.7.3. Motion Vector Direction	16
6. Transforms	16
7. Quantization	16
7.1. Quantization matrices	17
7.1.1. Quantization matrix selection	17
7.1.2. Quantization matrix design	18
8. Loop Filtering	18
8.1. Deblocking	18
8.1.1. Luma deblocking	18
8.1.2. Chroma Deblocking	19
8.2. Constrained Low Pass Filter (CLPF)	20
9. Entropy coding	20
9.1. Overview	20
9.2. Low Level Syntax	21
9.2.1. CB Level	21
9.2.2. PB Level	21
9.2.3. TB Level	22
9.2.4. Super Mode	22
9.2.5. CBP	23
9.2.6. Transform Coefficients	23

10. High Level Syntax	25
10.1. Sequence Header	25
10.2. Frame Header	26
11. IANA Considerations	27
12. Security Considerations	27
13. Normative References	27
Authors' Addresses	27

1. Introduction

This document provides a high-level description of the Thor video codec. Thor is designed to achieve high compression efficiency with moderate complexity, using the well-known hybrid video coding approach of motion-compensated prediction and transform coding.

The Thor video codec is a block-based hybrid video codec similar in structure to widespread standards. The high level encoder and decoder structures are illustrated in Figure 1 and Figure 2 respectively.

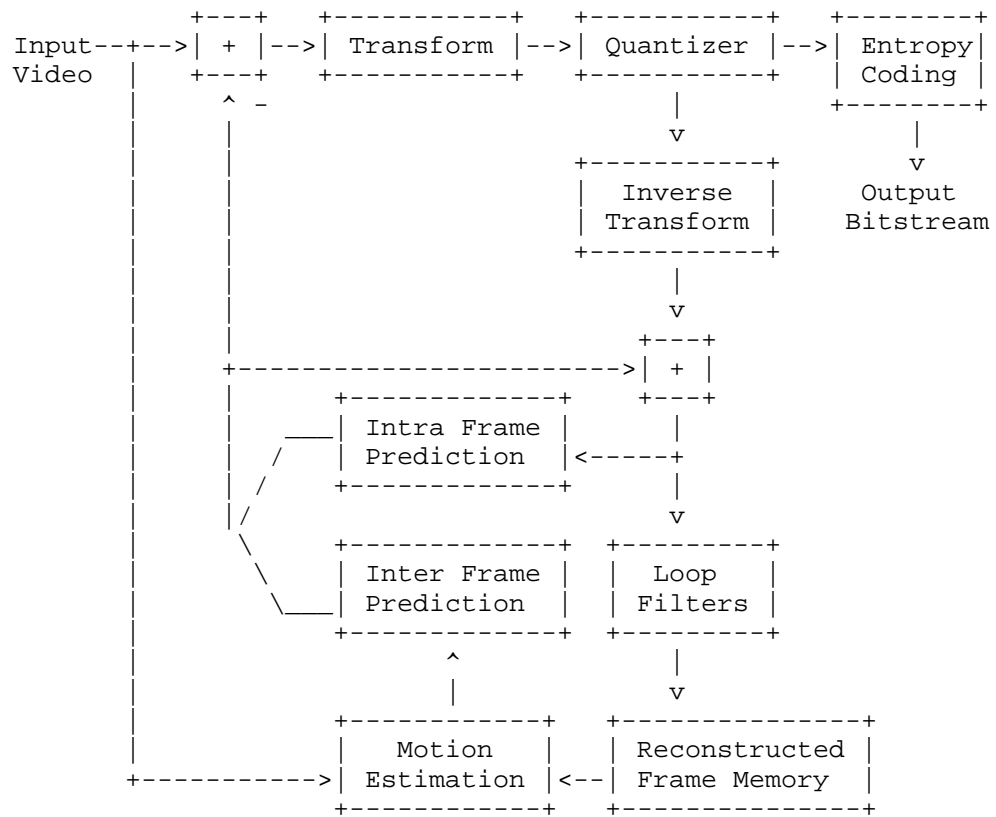


Figure 1: Encoder Structure

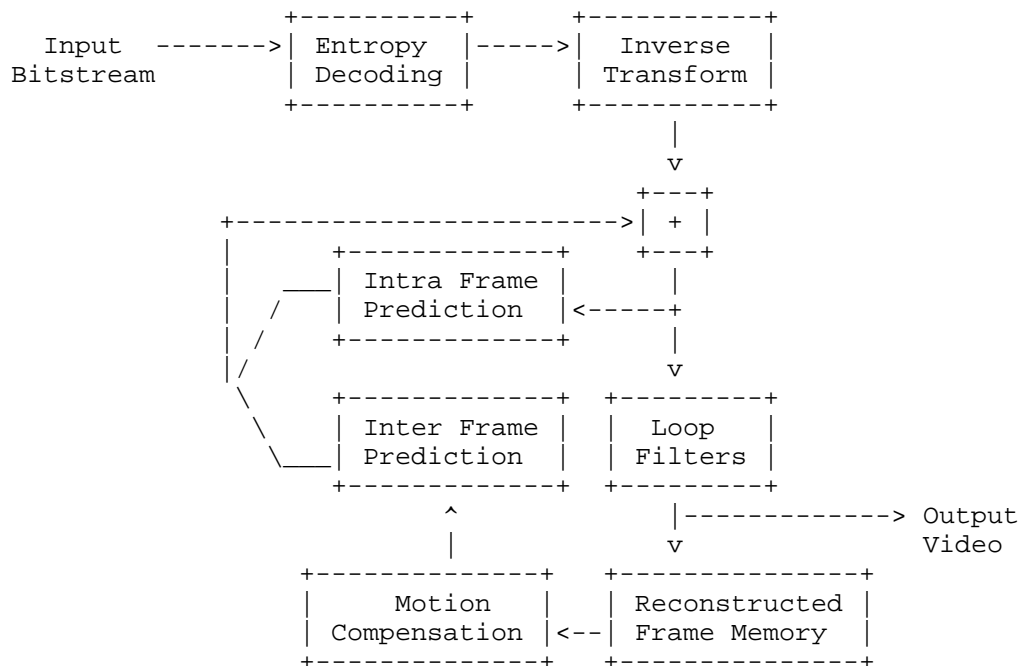


Figure 2: Decoder Structure

The remainder of this document is organized as follows. First, some requirements language and terms are defined. Block structures are described in detail, followed by intra-frame prediction techniques, inter-frame prediction techniques, transforms, quantization, loop filters, entropy coding, and finally high level syntax.

An open source reference implementation is available at github.com/cisco/thor.

2. Definitions

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2.2. Terminology

This document frequently uses the following terms.

SB: Super Block - 64x64 or 128x128 block (luma pixels) which can be divided into CBs.

CB: Coding Block - Subdivision of a SB, down to 8x8 (luma pixels).

PB: Prediction Block - Subdivision of a CB, into 1, 2 or 4 equal blocks.

TB: Transform Block - Subdivision of a CB, into 1 or 4 equal blocks.

3. Block Structure

3.1. Super Blocks and Coding Blocks

Input frames with bitdepths of 8, 10 or 12 are supported. The internal bitdepth can be 8, 10 or 12 regardless of input bitdepth. The bitdepth of the output frames always follows the input frames. Chroma can be subsampled in both directions (4:2:0) or have full resolution (4:4:4).

Each frame is divided into 64x64 or 128x128 Super Blocks (SB) which are processed in raster-scan order. The SB size is signaled in the sequence header. Each SB can be divided into Coding Blocks (CB) using a quad-tree structure. The smallest allowed CB size is 8x8 luma pixels. The four CBs of a larger block are coded/signaled in the following order; upleft, downleft, upright, and downright.

The following modes are signaled at the CB level:

- o Intra
- o Inter0 (skip): MV index, no residual information
- o Inter1 (merge): MV index, residual information
- o Inter2 (uni-pred): explicit motion information, residual information
- o Inter3 (bi-pred): explicit motion information, residual information

3.2. Special Processing at Frame Boundaries

At frame boundaries some square blocks might not be complete. For example, for 1920x1080 resolutions, the bottom row would consist of rectangular blocks of size 64x56. Rectangular blocks at frame boundaries are handled as follows. For each rectangular block, send one bit to choose between:

- o A rectangular inter0 block and
- o Further split.

For the bottom part of a 1920x1080 frame, this implies the following:

- o For each 64x56 block, transmit one bit to signal a 64x56 inter0 block or a split into two 32x32 blocks and two 32x24 blocks.
- o For each 32x24 block, transmit one bit to signal a 32x24 inter0 block or a split into two 16x16 blocks and two 16x8 blocks.
- o For each 16x8 block, transmit one bit to signal a 16x8 inter0 block or a split into two 8x8 blocks.

Two examples of handling 64x56 blocks at the bottom row of a 1920x1080 frame are shown in Figure 3 and Figure 4 respectively.

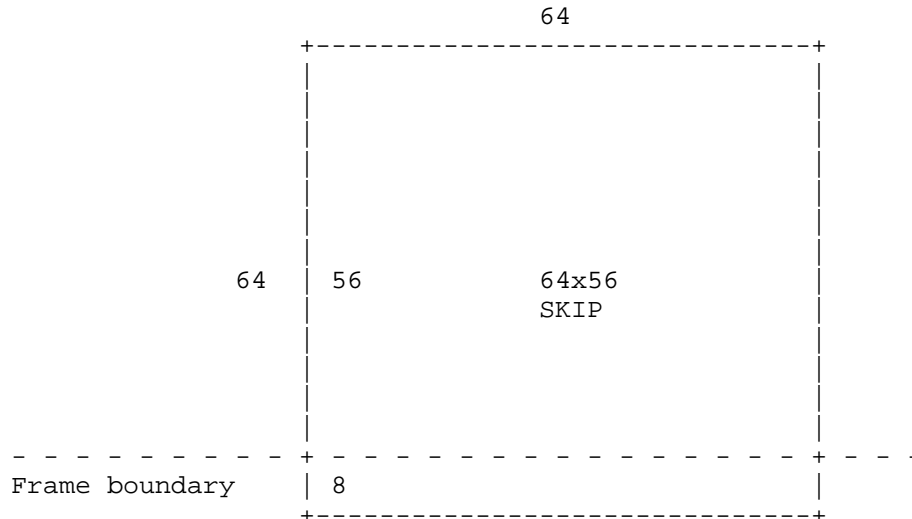


Figure 3: Super block at frame boundary

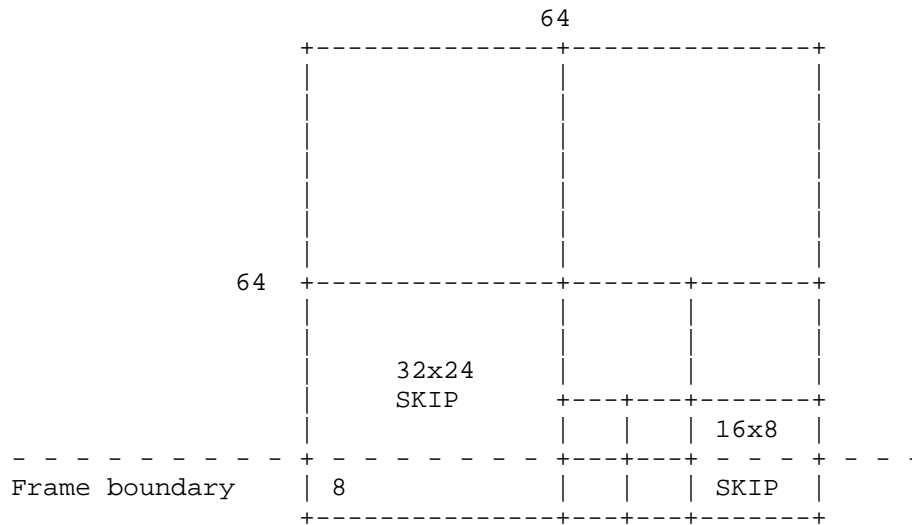


Figure 4: Coding block at frame boundary

3.3. Transform Blocks

A coding block (CB) can be divided into four smaller transform blocks (TBs).

3.4. Prediction Blocks

A coding block (CB) can also be divided into smaller prediction blocks (PBs) for the purpose of motion-compensated prediction. Horizontal, vertical and quad split are used.

4. Intra Prediction

8 intra prediction modes are used:

1. DC
2. Vertical (V)
3. Horizontal (H)
4. Upupright (north-northeast)
5. Upupleft (north-northwest)
6. Upleft (northwest)

7. Upleftleft (west-northwest)

8. Downleftleft (west-southwest)

The definition of DC, vertical, and horizontal modes are straightforward.

The upleft direction is exactly 45 degrees.

The upupright, upupleft, and upleftleft directions are equal to $\arctan(1/2)$ from the horizontal or vertical direction, since they are defined by going one pixel horizontally and two pixels vertically (or vice versa).

For the 5 angular intra modes (i.e. angle different from 90 degrees), the pixels of the neighbor blocks are filtered before they are used for prediction:

$$y(n) = (x(n-1) + 2*x(n) + x(n+1) + 2)/4$$

For the angular intra modes that are not 45 degrees, the prediction sometimes requires sample values at a half-pixel position. These sample values are determined by an additional filter:

$$z(n + 1/2) = (y(n) + y(n+1))/2$$

5. Inter Prediction

5.1. Multiple Reference Frames

Multiple reference frames are currently implemented as follows.

- o Use a sliding-window process to keep the N most recent reconstructed frames in memory. The value of N is signaled in the sequence header.
- o In the frame header, signal which of these frames shall be active for the current frame.
- o For each CB, signal which of the active frames to be used for MC.

Combined with re-ordering, this allows for MPEG-1 style B frames.

A desirable future extension is to allow long-term reference frames in addition to the short-term reference frames defined by the sliding-window process.

5.2. Bi-Prediction

In case of bi-prediction, two reference indices and two motion vectors are signaled per CB. In the current version, PB-split is not allowed in bi-prediction mode. Sub-pixel interpolation is performed for each motion vector/reference index separately before doing an average between the two predicted blocks:

$$p(x,y) = (p0(x,y) + p1(x,y))/2$$

5.3. Improved chroma prediction

If specified in the sequence header, the chroma prediction, both intra and inter, or either, is improved by using the luma reconstruction if certain criteria are met. The process is described in the separate CLPF draft [I-D.midtskogen-netvc-chromapred].

5.4. Reordered Frames

Frames may be transmitted out of order. Reference frames are selected from the sliding window buffer as normal.

5.5. Interpolated Reference Frames

A flag is sent in the sequence header indicating that interpolated reference frames may be used.

If a frame is using an interpolated reference frame, it will be the first reference in the reference list, and will be interpolated from the second and third reference in the list. It is indicated by a reference index of -1 and has a frame number equal to that of the current frame.

The interpolated reference is created by a deterministic process common to the encoder and decoder, and described in the separate IRFVC draft [I-D.davies-netvc-irfvc].

5.6. Sub-Pixel Interpolation

5.6.1. Luma Poly-phase Filter

Inter prediction uses traditional block-based motion compensated prediction with quarter pixel resolution. A separable 6-tap poly-phase filter is the basis method for doing MC with sub-pixel accuracy. The luma filter coefficients are as follows:

When bi-prediction is enabled in the sequence header:

1/4 phase: [2,-10,59,17,-5,1]/64

2/4 phase: [1,-8,39,39,-8,1]/64

3/4 phase: [1,-5,17,59,-10,2]/64

When bi-prediction is disabled in the sequence header:

1/4 phase: [1,-7,55,19,-5,1]/64

2/4 phase: [1,-7,38,38,-7,1]/64

3/4 phase: [1,-5,19,55,-7,1]/64

With reference to Figure 5, a fractional sample value, e.g. $i_{0,0}$ which has a phase of 1/4 in the horizontal dimension and a phase of 1/2 in the vertical dimension is calculated as follows:

$$a_{0,j} = 2*A_{-2,i} - 10*A_{-1,i} + 59*A_{0,i} + 17*A_{1,i} - 5*A_{2,i} + 1*A_{3,i}$$

where $j = -2, \dots, 3$

$$i_{0,0} = (1*a_{0,-2} - 8*a_{0,-1} + 39*a_{0,0} + 39*a_{0,1} - 8*a_{0,2} + 1*a_{0,3} + 2048)/4096$$

The minimum sub-block size is 8x8.

A				A	a	b	c	A
-1,-1				0,-1	0,-1	0,-1	0,-1	1,-1
A				A	a	b	c	A
-1,0				0,0	0,0	0,0	0,0	1,0
d				d	e	f	g	d
-1,0				0,0	0,0	0,0	0,0	1,0
h				h	i	j	k	h
-1,0				0,0	0,0	0,0	0,0	1,0
l				l	m	n	o	l
-1,0				0,0	0,0	0,0	0,0	1,0
A				A	a	b	c	A
-1,1				0,1	0,1	0,1	0,1	1,1

Figure 5: Sub-pixel positions

5.6.2. Luma Special Filter Position

For the fractional pixel position having exactly 2 quarter pixel offsets in each dimension, a non-separable filter is used to calculate the interpolated value. With reference to Figure 5, the center position $j0,0$ is calculated as follows:

$j0,0 =$

$[0 \cdot A_{-1,-1} + 1 \cdot A_{0,-1} + 1 \cdot A_{1,-1} + 0 \cdot A_{2,-1} +$

$1 \cdot A_{-1,0} + 2 \cdot A_{0,0} + 2 \cdot A_{1,0} + 1 \cdot A_{2,0} +$

$1 \cdot A_{-1,1} + 2 \cdot A_{0,1} + 2 \cdot A_{1,1} + 1 \cdot A_{2,1} +$

$$0*A_{-1,2} + 1*A_{0,2} + 1*A_{1,2} + 0*A_{2,2} + 8]/16$$

5.6.3. Chroma Poly-phase Filter

Chroma interpolation is performed with 1/8 pixel resolution using the following poly-phase filter.

1/8 phase: [-2, 58, 10, -2]/64

2/8 phase: [-4, 54, 16, -2]/64

3/8 phase: [-4, 44, 28, -4]/64

4/8 phase: [-4, 36, 36, -4]/64

5/8 phase: [-4, 28, 44, -4]/64

6/8 phase: [-2, 16, 54, -4]/64

7/8 phase: [-2, 10, 58, -2]/64

5.7. Motion Vector Coding

5.7.1. Inter0 and Inter1 Modes

Inter0 and inter1 modes imply signaling of a motion vector index to choose a motion vector from a list of candidate motion vectors with associated reference frame index. A list of motion vector candidates are derived from at most two different neighbor blocks, each having a unique motion vector/reference frame index. Signaling of the motion vector index uses 0 or 1 bit, dependent on the number of unique motion vector candidates. If the chosen neighbor block is coded in bi-prediction mode, the inter0 or inter1 block inherits both motion vectors, both reference indices and the bi-prediction property of the neighbor block.

For block sizes less than 64x64, inter0 has only one motion vector candidate, and its value is always zero.

Which neighbor blocks to use for motion vector candidates depends on the availability of the neighbor blocks (i.e. whether the neighbor blocks have already been coded, belong to the same slice and are not outside the frame boundaries). Four different availabilities, U, UR, L, and LL, are defined as illustrated in Figure 6. If the neighbor block is intra it is considered to be available but with a zero motion vector.

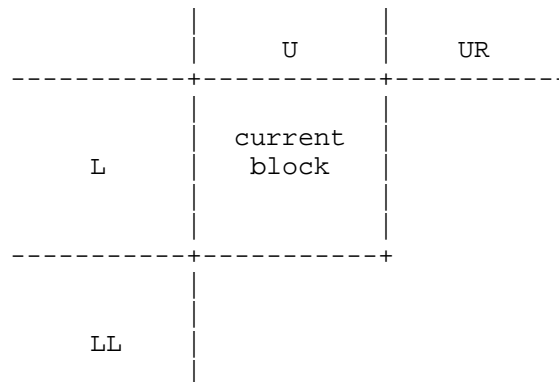


Figure 6: Availability of neighbor blocks

Based on the four availabilities defined above, each of the motion vector candidates is derived from one of the possible neighbor blocks defined in Figure 7.

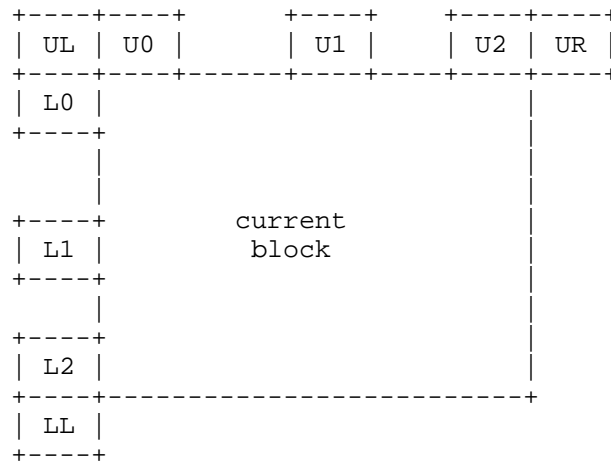


Figure 7: Motion vector candidates

The choice of motion vector candidates depends on the availability of neighbor blocks as shown in Table 1.

U	UR	L	LL	Motion vector candidates
0	0	0	0	zero vector
1	0	0	0	U2, zero vector
0	1	0	0	NA
1	1	0	0	U2, zero vector
0	0	1	0	L2, zero vector
1	0	1	0	U2,L2
0	1	1	0	NA
1	1	1	0	U2,L2
0	0	0	1	NA
1	0	0	1	NA
0	1	0	1	NA
1	1	0	1	NA
0	0	1	1	L2, zero vector
1	0	1	1	U2,L2
0	1	1	1	NA
1	1	1	1	U2,L2

Table 1: Motion vector candidates for different availability of neighbor blocks

5.7.2. Inter2 and Bi-Prediction Modes

Motion vectors are coded using motion vector prediction. The motion vector predictor is defined as the median of the motion vectors from three neighbor blocks. Definition of the motion vector predictor uses the same definition of availability and neighbors as in Figure 6 and Figure 7 respectively. The three vectors used for median filtering depends on the availability of neighbor blocks as shown in Table 2. If the neighbor block is coded in bi-prediction mode, only the first motion vector (in transmission order), MV0, is used as input to the median operator.

U	UR	L	LL	Motion vectors for median filtering
0	0	0	0	3 x zero vector
1	0	0	0	U0,U1,U2
0	1	0	0	NA
1	1	0	0	U0,U2,UR
0	0	1	0	L0,L1,L2
1	0	1	0	UL,U2,L2
0	1	1	0	NA
1	1	1	0	U0,UR,L2,L0
0	0	0	1	NA
1	0	0	1	NA
0	1	0	1	NA
1	1	0	1	NA
0	0	1	1	L0,L2,LL
1	0	1	1	U2,L0,LL
0	1	1	1	NA
1	1	1	1	U0,UR,L0

Table 2: Neighbor blocks used to define motion vector predictor through median filtering

5.7.3. Motion Vector Direction

Motion vectors referring to reference frames later in time than the current frame are stored with their sign reversed, and these reversed values are used for coding and motion vector prediction.

6. Transforms

Transforms are applied at the TB or CB level, implying that transform sizes range from 4x4 to 128x128. The transforms form an embedded structure meaning the transform matrix elements of the smaller transforms can be extracted from the larger transforms.

7. Quantization

For the 32x32, 64x64 and 128x128 transform sizes, only the 16x16 low frequency coefficients are quantized and transmitted.

The 64x64 inverse transform is defined as a 32x32 transform followed by duplicating each output sample into a 2x2 block. The 128x128 inverse transform is defined as a 32x32 transform followed by duplicating each output sample into a 4x4 block.

7.1. Quantization matrices

A flag is transmitted in the sequence header to indicate whether quantization matrices are used. If this flag is true, a 6 bit value `qmtx_offset` is transmitted in the sequence header to indicate matrix strength.

If used, then in dequantization a separate scaling factor is applied to each coefficient, so that the dequantized value of a coefficient `ci` at position `i` is:

$$(ci * d(q) * IW(i,c,s,t,q) + 2^{(k + 5)}) >> (k + 6)$$

Figure 8: Equation 1

where `IW` is the scale factor for coefficient position `i` with size `s`, frame type (inter/inter) `t`, component (Y, Cb or Cr) `c` and quantizer `q`; and `k=k(s,q)` is the dequantization shift. `IW` has scale 64, that is, a weight value of 64 is no different to unweighted dequantization.

7.1.1.1. Quantization matrix selection

The current luma `qp` value `qpY` and the offset value `qmtx_offset` determine a quantisation matrix set by the formula:

$$qmlevel = \max(0, \min(11, ((qpY + qmtx_offset) * 12) / 44))$$

Figure 9: Equation 2

This selects one of the 12 different sets of default quantization matrix, with increasing `qmlevel` indicating increasing flatness.

For a given value of `qmlevel`, different weighting matrices are provided for all combinations of transform block size, type (intra/inter), and component (Y, Cb, Cr). Matrices at low `qmlevel` are flat (constant value 64). Matrices for inter frames have unity DC gain (i.e. value 64 at position 0), whereas those for intra frames are designed such that the inverse weighting matrix has unity energy gain (i.e. normalized sum-squared of the scaling factors is 1).

7.1.2. Quantization matrix design

Further details on the quantization matrix and implementation can be found in the separate QMTX draft [I-D.davies-netvc-qmtx].

8. Loop Filtering

8.1. Deblocking

8.1.1. Luma deblocking

Luma deblocking is performed on an 8x8 grid as follows:

1. For each vertical edge between two 8x8 blocks, calculate the following for each of line 2 and line 5 respectively:

$$d = \text{abs}(a-b) + \text{abs}(c-d),$$

where a and b, are on the left hand side of the block edge and c and d are on the right hand side of the block edge:

a b | c d

2. For each line crossing the vertical edge, perform deblocking if and only if all of the following conditions are true:

- * $d2+d5 < \text{beta}(\text{QP})$

- * The edge is also a transform block edge

- * $\text{abs}(\text{mvx}(\text{left})) > 2$, or $\text{abs}(\text{mvx}(\text{right})) > 2$, or

- $\text{abs}(\text{mvy}(\text{left})) > 2$, or $\text{abs}(\text{mvy}(\text{right})) > 2$, or

One of the transform blocks on each side of the edge has non-zero coefficients, or

One of the transform blocks on each side of the edge is coded using intra mode.

3. If deblocking is performed, calculate a delta value as follows:

$$\text{delta} = \text{clip}((18*(c-b) - 6*(d-a) + 16)/32, \text{tc}, -\text{tc}),$$

where tc is a QP-dependent value.

4. Next, modify two pixels on each side of the block edge as follows:

$$a' = a + \text{delta}/2$$

$$b' = b + \text{delta}$$

$$c' = c + \text{delta}$$

$$d' = d + \text{delta}/2$$

5. The same procedure is followed for horizontal block edges.

The relative positions of the samples, a, b, c, d and the motion vectors, MV, are illustrated in Figure 10.

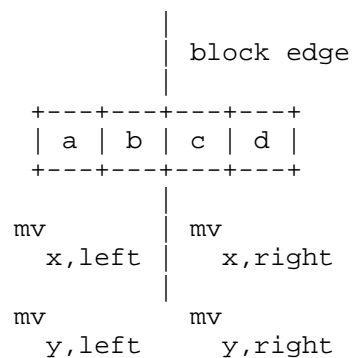


Figure 10: Deblocking filter pixel positions

8.1.2. Chroma Deblocking

Chroma deblocking is performed on a 4x4 grid as follows:

1. Deblocking of the edge between two 4x4 blocks is performed if and only if:
 - * The pixels on either side of the block edge belongs to an intra block.
 - * The block edge is also an edge between two transform blocks.
2. If deblocking is performed, calculate a delta value as follows:

$$\text{delta} = \text{clip}((4*(c-b) + (d-a) + 4)/8, \text{tc}, -\text{tc}),$$
 where tc is a QP-dependent value.

3. Next, modify one pixel on each side of the block edge as follows:

$$b' = b + \text{delta}$$

$$c' = c + \text{delta}$$

8.2. Constrained Low Pass Filter (CLPF)

A low-pass filter is applied after the deblocking filter if signaled in the sequence header. It can still be switched off for individual frames in the frame header. Also signaled in the frame header is whether to apply the filter for all qualified 128x128 blocks or to transmit a flag for each such block. A super block does not qualify if it only contains Inter0 (skip) coding block and no signal is transmitted for these blocks.

The filter is described in the separate CLPF draft [I-D.midtskogen-netvc-clpf].

9. Entropy coding

9.1. Overview

The following information is signaled at the sequence level:

- o Sequence header

The following information is signaled at the frame level:

- o Frame header

The following information is signaled at the CB level:

- o Super-mode (mode, split, reference index for uni-prediction)
- o Intra prediction mode
- o PB-split (none, hor, ver, quad)
- o TB-split (none or quad)
- o Reference frame indices for bi-prediction
- o Motion vector candidate index
- o Transform coefficients if TB-split=0

The following information is signaled at the TB level:

- o CBP (8 combinations of CBPY, CBPU, and CBPV)
- o Transform coefficients

The following information is signaled at the PB level:

- o Motion vector differences

9.2. Low Level Syntax

9.2.1. CB Level

super-mode (inter0/split/inter1/inter2-ref0/intra/inter2-ref1/inter2-ref2/inter2-ref3,...)

if (mode == inter0 || mode == inter1)

mv_idx (one of up to 2 motion vector candidates)

else if (mode == INTRA)

intra_mode (one of up to 8 intra modes)

tb_split (NONE or QUAD, coded jointly with CBP for tb_split=NONE)

else if (mode == INTER)

pb_split (NONE, VER, HOR, QUAD)

tb_split_and_cbp (NONE or QUAD and CBP)

else if (mode == BIPRED)

mvd_x0, mvd_y0 (motion vector difference for first vector)

mvd_x1, mvd_y1 (motion vector difference for second vector)

ref_idx0, ref_idx1 (two reference indices)

9.2.2. PB Level

if (mode == INTER2 || mode == BIPRED)

mvd_x, mvd_y (motion vector differences)

9.2.3. TB Level

```

    if (mode != INTER0 and tb_split == 1)

        cbp                                (8 possibilities for CBPY/CBPU/CBPV)

    if (mode != INTER0)

        transform coefficients

```

9.2.4. Super Mode

For each block of size $N \times N$ ($64 \geq N > 8$), the following mutually exclusive events are jointly encoded using a single VLC code as follows (example using 4 reference frames):

If there is no interpolated reference frame:

```

INTER0      1
SPLIT       01
INTER1      001
INTER2-REF0 0001
BIPRED      00001
INTRA       000001
INTER2-REF1 0000001
INTER2-REF2 00000001
INTER2-REF3 00000000

```

If there is an interpolated reference frame:

```

INTER0      1
SPLIT       01
INTER1      001
BIPRED      0001
INTRA       00001
INTER2-REF1 000001
INTER2-REF2 0000001
INTER2-REF3 00000001
INTER2-REF0 00000000

```

If less than 4 reference frames is used, a shorter VLC table is used. If bi-pred is not possible, or split is not possible, they are omitted from the table and shorter codes are used for subsequent elements.

Additionally, depending on information from the blocks to the left and above (meta data and CBP), a different sorting of the events can be used, e.g.:

SPLIT	1
INTER1	01
INTER2-REF0	001
INTER0	0001
INTRA	00001
INTER2-REF1	000001
INTER2-REF2	0000001
INTER2-REF3	00000001
BIPRED	00000000

9.2.5. CBP

Calculate code as follows:

```
if (tb-split == 0)

    N = 4*CBPV + 2*CBPU + CBPY

else

    N = 8
```

Map the value of N to code through a table lookup:

```
code = table[N]
```

where the purpose of the table lookup is the sort the different values of code according to decreasing probability (typically CBPY=1, CBPU=0, CBPV=0 having the highest probability).

Use a different table depending on the values of CBPY in neighbor blocks (left and above).

Encode the value of code using a systematic VLC code.

9.2.6. Transform Coefficients

Transform coefficient coding uses a traditional zig-zag scan pattern to convert a 2D array of quantized transform coefficients, `coeff`, to a 1D array of samples. VLC coding of quantized transform coefficients starts from the low frequency end of the 1D array using two different modes; level-mode and run-mode, starting in level-mode:

- o Level-mode
 - * Encode each coefficient, `coeff`, separately
 - * Each coefficient is encoded by:

- + The absolute value, $\text{level} = \text{abs}(\text{coeff})$, using a VLC code and
 - + If $\text{level} > 0$, the sign bit ($\text{sign}=0$ or $\text{sign}=1$ for $\text{coeff}>0$ and $\text{coeff}<0$ respectively).
 - * If coefficient N is zero, switch to run-mode, starting from coefficient $N+1$.
- o Run-mode
- * For each non-zero coefficient, encode the combined event of:
 1. Length of the zero-run, i.e. the number of zeros since the last non-zero coefficient.
 2. Whether or not $\text{level} = \text{abs}(\text{coeff})$ is greater than 1.
 3. End of block (EOB) indicating that there are no more non-zero coefficients.
 - * Additionally, if $\text{level} = 1$, code the sign bit.
 - * Additionally, if $\text{level} > 1$ define $\text{code} = 2 * (\text{level} - 2) + \text{sign}$,
 - * If the absolute value of coefficient N is larger than 1, switch to level-mode, starting from coefficient $N+1$.

Example

Figure 11 illustrates an example where 16 quantized transform coefficients are encoded.

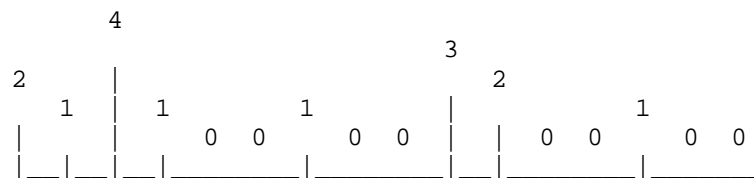


Figure 11: Coefficients to encode

Table 3 shows the mode, VLC number and symbols to be coded for each coefficient.

Index	abs(coeff)	Mode	Encoded symbols
0	2	level-mode	level=2,sign
1	1	level-mode	level=1,sign
2	4	level-mode	level=4,sign
3	1	level-mode	level=1,sign
4	0	level-mode	level=0
5	0	run-mode	
6	1	run-mode	(run=1,level=1)
7	0	run-mode	
8	0	run-mode	
9	3	run-mode	(run=1,level>1), 2*(3-2)+sign
10	2	level-mode	level=2, sign
11	0	level-mode	level=0
12	0	run-mode	
13	1	run-mode	(run=1,level=1)
14	0	run-mode	EOB
15	0	run-mode	

Table 3: Transform coefficient encoding for the example above.

10. High Level Syntax

High level syntax is currently very simple and rudimentary as the primary focus so far has been on compression performance. It is expected to evolve as functionality is added.

10.1. Sequence Header

- o Width - 16 bits
- o Height - 16 bits
- o Enable/disable PB-split - 1 bit
- o SB size - 3 bits
- o Enable/disable TB-split - 1 bit
- o Number of active reference frames (may go into frame header) - 2 bits (max 4)
- o Enable/disable interpolated reference frames - 1 bit
- o Enable/disable delta qp - 1 bit

- o Enable/disable deblocking - 1 bit
- o Constrained low-pass filter (CLPF) enable/disable - 1 bit
- o Enable/disable block context coding - 1 bit
- o Enable/disable bi-prediction - 1 bit
- o Enable/disable quantization matrices - 1 bit
- o If quantization matrices enabled: quantization matrix offset - 6 bits
- o Select 420 or 444 input - 1 bit
- o Number of reordered frames - 4 bits
- o Enable/disable chroma intra prediction from luma - 1 bit
- o Enable/disable chroma inter prediction from luma - 1 bit
- o Internal frame bitdepth (8, 10 or 12 bits) - 2 bits
- o Input video bitdepth (8, 10 or 12 bits) - 2 bits

10.2. Frame Header

- o Frame type - 1 bit
- o QP - 8 bits
- o Identification of active reference frames - num_ref*4 bits
- o Number of intra modes - 4 bits
- o Number of active reference frames - 2 bits
- o Active reference frames - number of active reference frames * 6 bits
- o Frame number - 16 bits
- o If CLPF is enabled in the sequence header: Constrained low-pass filter (CLPF) strength - 2 bits (00 = off, 01 = strength 1, 10 = strength 2, 11 = strength 4)
- o IF CLPF is enabled in the sequence header: Enable/disable CLPF signal for each qualified filter block

11. IANA Considerations

This document has no IANA considerations yet. TBD

12. Security Considerations

This document has no security considerations yet. TBD

13. Normative References

[I-D.davies-netvc-irfvc]

Davies, T., "Interpolated reference frames for video coding", draft-davies-netvc-irfvc-00 (work in progress), October 2015.

[I-D.davies-netvc-qmtx]

Davies, T., "Quantisation matrices for Thor video coding", draft-davies-netvc-qmtx-00 (work in progress), March 2016.

[I-D.midtskogen-netvc-chromapred]

Midtskogen, S., "Improved chroma prediction", draft-midtskogen-netvc-chromapred-02 (work in progress), October 2016.

[I-D.midtskogen-netvc-clpf]

Midtskogen, S., Fuldseth, A., and M. Zanaty, "Constrained Low Pass Filter", draft-midtskogen-netvc-clpf-02 (work in progress), April 2016.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

Authors' Addresses

Arild Fuldseth
Cisco
Lysaker
Norway

Email: arilfuld@cisco.com

Gisle Bjontegaard
Cisco
Lysaker
Norway

Email: gbjonteg@cisco.com

Steinar Midtskogen
Cisco
Lysaker
Norway

Email: stemidts@cisco.com

Thomas Davies
Cisco
London
UK

Email: thdavies@cisco.com

Mo Zanaty
Cisco
RTP,NC
USA

Email: mzanaty@cisco.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 3, 2020

T. Daede
Mozilla
A. Norkin
Netflix
I. Brailovski
Amazon Lab126
January 31, 2020

Video Codec Testing and Quality Measurement
draft-ietf-netvc-testing-09

Abstract

This document describes guidelines and procedures for evaluating a video codec. This covers subjective and objective tests, test conditions, and materials used for the test.

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 <https://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 August 3, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Subjective quality tests	3
2.1. Still Image Pair Comparison	3
2.2. Video Pair Comparison	4
2.3. Mean Opinion Score	4
3. Objective Metrics	5
3.1. Overall PSNR	5
3.2. Frame-averaged PSNR	5
3.3. PSNR-HVS-M	6
3.4. SSIM	6
3.5. Multi-Scale SSIM	6
3.6. CIEDE2000	6
3.7. VMAF	6
4. Comparing and Interpreting Results	7
4.1. Graphing	7
4.2. BD-Rate	7
4.3. Ranges	8
5. Test Sequences	8
5.1. Sources	8
5.2. Test Sets	8
5.2.1. regression-1	9
5.2.2. objective-2-slow	9
5.2.3. objective-2-fast	12
5.2.4. objective-1.1	14
5.2.5. objective-1-fast	17
5.3. Operating Points	19
5.3.1. Common settings	19
5.3.2. High Latency CQP	19
5.3.3. Low Latency CQP	19
5.3.4. Unconstrained High Latency	20
5.3.5. Unconstrained Low Latency	20
6. Automation	20
6.1. Regression tests	21
6.2. Objective performance tests	21
6.3. Periodic tests	22
7. IANA Considerations	22
8. Security Considerations	22
9. Informative References	22
Authors' Addresses	23

1. Introduction

When developing a video codec, changes and additions to the codec need to be decided based on their performance tradeoffs. In addition, measurements are needed to determine when the codec has met its performance goals. This document specifies how the tests are to be carried about to ensure valid comparisons when evaluating changes under consideration. Authors of features or changes should provide the results of the appropriate test when proposing codec modifications.

2. Subjective quality tests

Subjective testing uses human viewers to rate and compare the quality of videos. It is the preferable method of testing video codecs.

Subjective testing results take priority over objective testing results, when available. Subjective testing is recommended especially when taking advantage of psychovisual effects that may not be well represented by objective metrics, or when different objective metrics disagree.

Selection of a testing methodology depends on the feature being tested and the resources available. Test methodologies are presented in order of increasing accuracy and cost.

Testing relies on the resources of participants. If a participant requires a subjective test for a particular feature or improvement, they are responsible for ensuring that resources are available. This ensures that only important tests be done; in particular, the tests that are important to participants.

Subjective tests should use the same operating points as the objective tests.

2.1. Still Image Pair Comparison

A simple way to determine superiority of one compressed image is to visually compare two compressed images, and have the viewer judge which one has a higher quality. For example, this test may be suitable for an intra de-ringing filter, but not for a new inter prediction mode. For this test, the two compressed images should have similar compressed file sizes, with one image being no more than 5% larger than the other. In addition, at least 5 different images should be compared.

Once testing is complete, a p-value can be computed using the binomial test. A significant result should have a resulting p-value less than or equal to 0.5. For example:

```
p_value = binom_test(a,a+b)
```

where a is the number of votes for one video, b is the number of votes for the second video, and `binom_test(x,y)` returns the binomial PMF (probability mass function) with x observed tests, y total tests, and expected probability 0.5.

If ties are allowed to be reported, then the equation is modified:

```
p_value = binom_test(a+floor(t/2),a+b+t)
```

where t is the number of tie votes.

Still image pair comparison is used for rapid comparisons during development – the viewer may be either a developer or user, for example. As the results are only relative, it is effective even with an inconsistent viewing environment. Because this test only uses still images (keyframes), this is only suitable for changes with similar or no effect on inter frames.

2.2. Video Pair Comparison

The still image pair comparison method can be modified to also compare videos. This is necessary when making changes with temporal effects, such as changes to inter-frame prediction. Video pair comparisons follow the same procedure as still images. Videos used for testing should be limited to 10 seconds in length, and can be rewatched an unlimited number of times.

2.3. Mean Opinion Score

A Mean Opinion Score (MOS) viewing test is the preferred method of evaluating the quality. The subjective test should be performed as either consecutively showing the video sequences on one screen or on two screens located side-by-side. The testing procedure should normally follow rules described in [BT500] and be performed with non-expert test subjects. The result of the test will be (depending on the test procedure) mean opinion scores (MOS) or differential mean opinion scores (DMOS). Confidence intervals are also calculated to judge whether the difference between two encodings is statistically significant. In certain cases, a viewing test with expert test subjects can be performed, for example if a test should evaluate technologies with similar performance with respect to a particular artifact (e.g. loop filters or motion prediction). Unlike pair

comparisons, a MOS test requires a consistent testing environment. This means that for large scale or distributed tests, pair comparisons are preferred.

3. Objective Metrics

Objective metrics are used in place of subjective metrics for easy and repeatable experiments. Most objective metrics have been designed to correlate with subjective scores.

The following descriptions give an overview of the operation of each of the metrics. Because implementation details can sometimes vary, the exact implementation is specified in C in the Daala tools repository [DAALA-GIT]. Implementations of metrics must directly support the input's resolution, bit depth, and sampling format.

Unless otherwise specified, all of the metrics described below only apply to the luma plane, individually by frame. When applied to the video, the scores of each frame are averaged to create the final score.

Codecs must output the same resolution, bit depth, and sampling format as the input.

3.1. Overall PSNR

PSNR is a traditional signal quality metric, measured in decibels. It is directly derived from mean square error (MSE), or its square root (RMSE). The formula used is:

$$20 * \log_{10} (\text{MAX} / \text{RMSE})$$

or, equivalently:

$$10 * \log_{10} (\text{MAX}^2 / \text{MSE})$$

where the error is computed over all the pixels in the video, which is the method used in the `dump_psnr.c` reference implementation.

This metric may be applied to both the luma and chroma planes, with all planes reported separately.

3.2. Frame-averaged PSNR

PSNR can also be calculated per-frame, and then the values averaged together. This is reported in the same way as overall PSNR.

3.3. PSNR-HVS-M

The PSNR-HVS [PSNRHVS] metric performs a DCT transform of 8x8 blocks of the image, weights the coefficients, and then calculates the PSNR of those coefficients. Several different sets of weights have been considered. The weights used by the `dump_pnsrhvs.c` tool in the Daala repository have been found to be the best match to real MOS scores.

3.4. SSIM

SSIM (Structural Similarity Image Metric) is a still image quality metric introduced in 2004 [SSIM]. It computes a score for each individual pixel, using a window of neighboring pixels. These scores can then be averaged to produce a global score for the entire image. The original paper produces scores ranging between 0 and 1.

To linearize the metric for BD-Rate computation, the score is converted into a nonlinear decibel scale:

$$-10 * \log_{10} (1 - \text{SSIM})$$

3.5. Multi-Scale SSIM

Multi-Scale SSIM is SSIM extended to multiple window sizes [MSSSIM]. The metric score is converted to decibels in the same way as SSIM.

3.6. CIEDE2000

CIEDE2000 is a metric based on CIEDE color distances [CIEDE2000]. It generates a single score taking into account all three chroma planes. It does not take into consideration any structural similarity or other psychovisual effects.

3.7. VMAF

Video Multi-method Assessment Fusion (VMAF) is a full-reference perceptual video quality metric that aims to approximate human perception of video quality [VMAF]. This metric is focused on quality degradation due to compression and rescaling. VMAF estimates the perceived quality score by computing scores from multiple quality assessment algorithms, and fusing them using a support vector machine (SVM). Currently, three image fidelity metrics and one temporal signal have been chosen as features to the SVM, namely Anti-noise SNR (ANSNR), Detail Loss Measure (DLM), Visual Information Fidelity (VIF), and the mean co-located pixel difference of a frame with respect to the previous frame.

The quality score from VMAF is used directly to calculate BD-Rate, without any conversions.

4. Comparing and Interpreting Results

4.1. Graphing

When displayed on a graph, bitrate is shown on the X axis, and the quality metric is on the Y axis. For publication, the X axis should be linear. The Y axis metric should be plotted in decibels. If the quality metric does not natively report quality in decibels, it should be converted as described in the previous section.

4.2. BD-Rate

The Bjontegaard rate difference, also known as BD-rate, allows the measurement of the bitrate reduction offered by a codec or codec feature, while maintaining the same quality as measured by objective metrics. The rate change is computed as the average percent difference in rate over a range of qualities. Metric score ranges are not static - they are calculated either from a range of bitrates of the reference codec, or from quantizers of a third, anchor codec. Given a reference codec and test codec, BD-rate values are calculated as follows:

- o Rate/distortion points are calculated for the reference and test codec.
 - * At least four points must be computed. These points should be the same quantizers when comparing two versions of the same codec.
 - * Additional points outside of the range should be discarded.
- o The rates are converted into log-rates.
- o A piecewise cubic hermite interpolating polynomial is fit to the points for each codec to produce functions of log-rate in terms of distortion.
- o Metric score ranges are computed:
 - * If comparing two versions of the same codec, the overlap is the intersection of the two curves, bound by the chosen quantizer points.
 - * If comparing dissimilar codecs, a third anchor codec's metric scores at fixed quantizers are used directly as the bounds.

- o The log-rate is numerically integrated over the metric range for each curve, using at least 1000 samples and trapezoidal integration.
- o The resulting integrated log-rates are converted back into linear rate, and then the percent difference is calculated from the reference to the test codec.

4.3. Ranges

For individual feature changes in libaom or libvpx, the overlap BD-Rate method with quantizers 20, 32, 43, and 55 must be used.

For the final evaluation described in [I-D.ietf-netvc-requirements], the quantizers used are 20, 24, 28, 32, 36, 39, 43, 47, 51, and 55.

5. Test Sequences

5.1. Sources

Lossless test clips are preferred for most tests, because the structure of compression artifacts in already-compressed clips may introduce extra noise in the test results. However, a large amount of content on the internet needs to be recompressed at least once, so some sources of this nature are useful. The encoder should run at the same bit depth as the original source. In addition, metrics need to support operation at high bit depth. If one or more codecs in a comparison do not support high bit depth, sources need to be converted once before entering the encoder.

5.2. Test Sets

Sources are divided into several categories to test different scenarios the codec will be required to operate in. For easier comparison, all videos in each set should have the same color subsampling, same resolution, and same number of frames. In addition, all test videos must be publicly available for testing use, to allow for reproducibility of results. All current test sets are available for download [TESTSEQUENCES].

Test sequences should be downloaded in whole. They should not be recreated from the original sources.

Each clip is labeled with its resolution, bit depth, color subsampling, and length.

5.2.1. regression-1

This test set is used for basic regression testing. It contains a very small number of clips.

- o kirlandvga (640x360, 8bit, 4:2:0, 300 frames)
- o FourPeople (1280x720, 8bit, 4:2:0, 60 frames)
- o Narrator (4096x2160, 10bit, 4:2:0, 15 frames)
- o CSGO (1920x1080, 8bit, 4:4:4 60 frames)

5.2.2. objective-2-slow

This test set is a comprehensive test set, grouped by resolution. These test clips were created from originals at [TESTSEQUENCES]. They have been scaled and cropped to match the resolution of their category. This test set requires a codec that supports both 8 and 10 bit video.

4096x2160, 4:2:0, 60 frames:

- o Netflix_BarScene_4096x2160_60fps_10bit_420_60f
- o Netflix_BoxingPractice_4096x2160_60fps_10bit_420_60f
- o Netflix_Dancers_4096x2160_60fps_10bit_420_60f
- o Netflix_Narrator_4096x2160_60fps_10bit_420_60f
- o Netflix_RitualDance_4096x2160_60fps_10bit_420_60f
- o Netflix_ToddlerFountain_4096x2160_60fps_10bit_420_60f
- o Netflix_WindAndNature_4096x2160_60fps_10bit_420_60f
- o street_hdr_amazon_2160p

1920x1080, 4:2:0, 60 frames:

- o aspen_1080p_60f
- o crowd_run_1080p50_60f
- o ducks_take_off_1080p50_60f
- o guitar_hdr_amazon_1080p

- o life_1080p30_60f
- o Netflix_Aerial_1920x1080_60fps_8bit_420_60f
- o Netflix_Boat_1920x1080_60fps_8bit_420_60f
- o Netflix_Crosswalk_1920x1080_60fps_8bit_420_60f
- o Netflix_FoodMarket_1920x1080_60fps_8bit_420_60f
- o Netflix_PierSeaside_1920x1080_60fps_8bit_420_60f
- o Netflix_SquareAndTimelapse_1920x1080_60fps_8bit_420_60f
- o Netflix_TunnelFlag_1920x1080_60fps_8bit_420_60f
- o old_town_cross_1080p50_60f
- o pan_hdr_amazon_1080p
- o park_joy_1080p50_60f
- o pedestrian_area_1080p25_60f
- o rush_field_cuts_1080p_60f
- o rush_hour_1080p25_60f
- o seaplane_hdr_amazon_1080p
- o station2_1080p25_60f
- o touchdown_pass_1080p_60f

1280x720, 4:2:0, 120 frames:

- o boat_hdr_amazon_720p
- o dark720p_120f
- o FourPeople_1280x720_60_120f
- o gipsrestat720p_120f
- o Johnny_1280x720_60_120f
- o KristenAndSara_1280x720_60_120f

- o Netflix_DinnerScene_1280x720_60fps_8bit_420_120f
- o Netflix_DrivingPOV_1280x720_60fps_8bit_420_120f
- o Netflix_FoodMarket2_1280x720_60fps_8bit_420_120f
- o Netflix_RollerCoaster_1280x720_60fps_8bit_420_120f
- o Netflix_Tango_1280x720_60fps_8bit_420_120f
- o rain_hdr_amazon_720p
- o vidyo1_720p_60fps_120f
- o vidyo3_720p_60fps_120f
- o vidyo4_720p_60fps_120f

640x360, 4:2:0, 120 frames:

- o blue_sky_360p_120f
- o controlled_burn_640x360_120f
- o desktop2360p_120f
- o kirland360p_120f
- o mmstationary360p_120f
- o niklas360p_120f
- o rain2_hdr_amazon_360p
- o red_kayak_360p_120f
- o riverbed_360p25_120f
- o shields2_640x360_120f
- o snow_mnt_640x360_120f
- o speed_bag_640x360_120f
- o stockholm_640x360_120f
- o tacomanarrows360p_120f

- o thaloundeskg360p_120f
 - o water_hdr_amazon_360p
- 426x240, 4:2:0, 120 frames:

- o bqfree_240p_120f
- o bqhighway_240p_120f
- o bqzoom_240p_120f
- o chairlift_240p_120f
- o dirtbike_240p_120f
- o mozzoom_240p_120f

1920x1080, 4:4:4 or 4:2:0, 60 frames:

- o CSGO_60f.y4m
- o DOTA2_60f_420.y4m
- o MINECRAFT_60f_420.y4m
- o STARCRAFT_60f_420.y4m
- o EuroTruckSimulator2_60f.y4m
- o Hearthstone_60f.y4m
- o wikipedia_420.y4m
- o pvq_slideshow.y4m

5.2.3. objective-2-fast

This test set is a strict subset of objective-2-slow. It is designed for faster runtime. This test set requires compiling with high bit depth support.

1920x1080, 4:2:0, 60 frames:

- o aspen_1080p_60f
- o ducks_take_off_1080p50_60f

- o life_1080p30_60f
- o Netflix_Aerial_1920x1080_60fps_8bit_420_60f
- o Netflix_Boat_1920x1080_60fps_8bit_420_60f
- o Netflix_FoodMarket_1920x1080_60fps_8bit_420_60f
- o Netflix_PierSeaside_1920x1080_60fps_8bit_420_60f
- o Netflix_SquareAndTimelapse_1920x1080_60fps_8bit_420_60f
- o Netflix_TunnelFlag_1920x1080_60fps_8bit_420_60f
- o rush_hour_1080p25_60f
- o seaplane_hdr_amazon_1080p
- o touchdown_pass_1080p_60f

1280x720, 4:2:0, 120 frames:

- o boat_hdr_amazon_720p
- o dark720p_120f
- o gipsrestat720p_120f
- o KristenAndSara_1280x720_60_120f
- o Netflix_DrivingPOV_1280x720_60fps_8bit_420_60f
- o Netflix_RollerCoaster_1280x720_60fps_8bit_420_60f
- o vidyo1_720p_60fps_120f
- o vidyo4_720p_60fps_120f

640x360, 4:2:0, 120 frames:

- o blue_sky_360p_120f
- o controlled_burn_640x360_120f
- o kirland360p_120f
- o niklas360p_120f

- o rain2_hdr_amazon_360p
- o red_kayak_360p_120f
- o riverbed_360p25_120f
- o shields2_640x360_120f
- o speed_bag_640x360_120f
- o thaloundesgmtg360p_120f

426x240, 4:2:0, 120 frames:

- o bqfree_240p_120f
- o bqzoom_240p_120f
- o dirtbike_240p_120f

1290x1080, 4:2:0, 60 frames:

- o DOTA2_60f_420.y4m
- o MINECRAFT_60f_420.y4m
- o STARCRAFT_60f_420.y4m
- o wikipedia_420.y4m

5.2.4. objective-1.1

This test set is an old version of objective-2-slow.

4096x2160, 10bit, 4:2:0, 60 frames:

- o Aerial (start frame 600)
- o BarScene (start frame 120)
- o Boat (start frame 0)
- o BoxingPractice (start frame 0)
- o Crosswalk (start frame 0)
- o Dancers (start frame 120)

- o FoodMarket
- o Narrator
- o PierSeaside
- o RitualDance
- o SquareAndTimelapse
- o ToddlerFountain (start frame 120)
- o TunnelFlag
- o WindAndNature (start frame 120)

1920x1080, 8bit, 4:4:4, 60 frames:

- o CSGO
- o DOTA2
- o EuroTruckSimulator2
- o Hearthstone
- o MINECRAFT
- o STARCRAFT
- o wikipedia
- o pvq_slideshow

1920x1080, 8bit, 4:2:0, 60 frames:

- o ducks_take_off
- o life
- o aspen
- o crowd_run
- o old_town_cross
- o park_joy

- o pedestrian_area
- o rush_field_cuts
- o rush_hour
- o station2
- o touchdown_pass

1280x720, 8bit, 4:2:0, 60 frames:

- o Netflix_FoodMarket2
- o Netflix_Tango
- o DrivingPOV (start frame 120)
- o DinnerScene (start frame 120)
- o RollerCoaster (start frame 600)
- o FourPeople
- o Johnny
- o KristenAndSara
- o vidyo1
- o vidyo3
- o vidyo4
- o dark720p
- o gipsrecmotion720p
- o gipsrestat720p
- o controlled_burn
- o stockholm
- o speed_bag
- o snow_mnt

- o shields

640x360, 8bit, 4:2:0, 60 frames:

- o red_kayak
- o blue_sky
- o riverbed
- o thaloundeskmvgvga
- o kirlandvga
- o tacomanarrowsvga
- o tacomascmvvga
- o desktop2360p
- o mmmovingvga
- o mmstationaryvga
- o niklasvga

5.2.5. objective-1-fast

This is an old version of objective-2-fast.

1920x1080, 8bit, 4:2:0, 60 frames:

- o Aerial (start frame 600)
- o Boat (start frame 0)
- o Crosswalk (start frame 0)
- o FoodMarket
- o PierSeaside
- o SquareAndTimelapse
- o TunnelFlag

1920x1080, 8bit, 4:2:0, 60 frames:

- o CSGO
- o EuroTruckSimulator2
- o MINECRAFT

- o wikipedia

1920x1080, 8bit, 4:2:0, 60 frames:

- o ducks_take_off
- o aspen
- o old_town_cross
- o pedestrian_area
- o rush_hour
- o touchdown_pass

1280x720, 8bit, 4:2:0, 60 frames:

- o Netflix_FoodMarket2
- o DrivingPOV (start frame 120)
- o RollerCoaster (start frame 600)
- o Johnny
- o vidyo1
- o vidyo4
- o gipsrecmotion720p
- o speed_bag
- o shields

640x360, 8bit, 4:2:0, 60 frames:

- o red_kayak
- o riverbed

- o kirlandvga
- o tacomascmvvga
- o mmmovingvga
- o niklasvga

5.3. Operating Points

Four operating modes are defined. High latency is intended for on demand streaming, one-to-many live streaming, and stored video. Low latency is intended for videoconferencing and remote access. Both of these modes come in CQP (constant quantizer parameter) and unconstrained variants. When testing still image sets, such as subset1, high latency CQP mode should be used.

5.3.1. Common settings

Encoders should be configured to their best settings when being compared against each other:

- o av1: -codec=av1 -ivf -frame-parallel=0 -tile-columns=0 -cpu-used=0 -threads=1

5.3.2. High Latency CQP

High Latency CQP is used for evaluating incremental changes to a codec. This method is well suited to compare codecs with similar coding tools. It allows codec features with intrinsic frame delay.

- o daala: -v=x -b 2
- o vp9: -end-usage=q -cq-level=x -lag-in-frames=25 -auto-alt-ref=2
- o av1: -end-usage=q -cq-level=x -auto-alt-ref=2

5.3.3. Low Latency CQP

Low Latency CQP is used for evaluating incremental changes to a codec. This method is well suited to compare codecs with similar coding tools. It requires the codec to be set for zero intrinsic frame delay.

- o daala: -v=x
- o av1: -end-usage=q -cq-level=x -lag-in-frames=0

5.3.4. Unconstrained High Latency

The encoder should be run at the best quality mode available, using the mode that will provide the best quality per bitrate (VBR or constant quality mode). Lookahead and/or two-pass are allowed, if supported. One parameter is provided to adjust bitrate, but the units are arbitrary. Example configurations follow:

- o x264: -crf=x
- o x265: -crf=x
- o daala: -v=x -b 2
- o av1: -end-usage=q -cq-level=x -lag-in-frames=25 -auto-alt-ref=2

5.3.5. Unconstrained Low Latency

The encoder should be run at the best quality mode available, using the mode that will provide the best quality per bitrate (VBR or constant quality mode), but no frame delay, buffering, or lookahead is allowed. One parameter is provided to adjust bitrate, but the units are arbitrary. Example configurations follow:

- o x264: -crf=x -tune zerolatency
- o x265: -crf=x -tune zerolatency
- o daala: -v=x
- o av1: -end-usage=q -cq-level=x -lag-in-frames=0

6. Automation

Frequent objective comparisons are extremely beneficial while developing a new codec. Several tools exist in order to automate the process of objective comparisons. The Compare-Codecs tool allows BD-rate curves to be generated for a wide variety of codecs [COMPARECODECS]. The Daala source repository contains a set of scripts that can be used to automate the various metrics used. In addition, these scripts can be run automatically utilizing distributed computers for fast results, with rd_tool [RD_TOOL]. This tool can be run via a web interface called AreWeCompressedYet [AWCY], or locally.

Because of computational constraints, several levels of testing are specified.

6.1. Regression tests

Regression tests run on a small number of short sequences - regression-test-1. The regression tests should include a number of various test conditions. The purpose of regression tests is to ensure bug fixes (and similar patches) do not negatively affect the performance. The anchor in regression tests is the previous revision of the codec in source control. Regression tests are run on both high and low latency CQP modes

6.2. Objective performance tests

Changes that are expected to affect the quality of encode or bitstream should run an objective performance test. The performance tests should be run on a wider number of sequences. The following data should be reported:

- o Identifying information for the encoder used, such as the git commit hash.
- o Command line options to the encoder, configure script, and anything else necessary to replicate the experiment.
- o The name of the test set run (objective-1-fast)
- o For both high and low latency CQP modes, and for each objective metric:
 - * The BD-Rate score, in percent, for each clip.
 - * The average of all BD-Rate scores, equally weighted, for each resolution category in the test set.
 - * The average of all BD-Rate scores for all videos in all categories.

Normally, the encoder should always be run at the slowest, highest quality speed setting (cpu-used=0 in the case of AV1 and VP9). However, in the case of computation time, both the reference and changed encoder can be built with some options disabled. For AV1, -disable-ext_partition and -disable-ext_partition_types can be passed to the configure script to substantially speed up encoding, but the usage of these options must be reported in the test results.

6.3. Periodic tests

Periodic tests are run on a wide range of bitrates in order to gauge progress over time, as well as detect potential regressions missed by other tests.

7. IANA Considerations

This document does not require any IANA actions.

8. Security Considerations

This document describes the methodologies and procedures for qualitative testing, therefore does not itself have implications for network of decoder security.

9. Informative References

- [AWCY] Xiph.Org, "Are We Compressed Yet?", 2016, <<https://arewecompressedyet.com/>>.
- [BT500] ITU-R, "Recommendation ITU-R BT.500-13", 2012, <https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.500-13-201201-I!!PDF-E.pdf>.
- [CIEDE2000] Yang, Y., Ming, J., and N. Yu, "Color Image Quality Assessment Based on CIEDE2000", 2012, <<http://dx.doi.org/10.1155/2012/273723>>.
- [COMPARECODECS] Alvestrand, H., "Compare Codecs", 2015, <<http://compare-codecs.appspot.com/>>.
- [DAALA-GIT] Xiph.Org, "Daala Git Repository", 2015, <<http://git.xiph.org/?p=daala.git;a=summary>>.
- [I-D.ietf-netvc-requirements] Filippov, A., Norkin, A., and j. jose.roberto.alvarez@huawei.com, "Video Codec Requirements and Evaluation Methodology", draft-ietf-netvc-requirements-10 (work in progress), November 2019.
- [MSSSIM] Wang, Z., Simoncelli, E., and A. Bovik, "Multi-Scale Structural Similarity for Image Quality Assessment", n.d., <<http://www.cns.nyu.edu/~zwang/files/papers/msssim.pdf>>.

- [PSNRHVS] Egiazarian, K., Astola, J., Ponomarenko, N., Lukin, V., Battisti, F., and M. Carli, "A New Full-Reference Quality Metrics Based on HVS", 2002.
- [RD_TOOL] Xiph.Org, "rd_tool", 2016,
<https://github.com/tdaede/rd_tool>.
- [SSIM] Wang, Z., Bovik, A., Sheikh, H., and E. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity", 2004,
<<http://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf>>.
- [TESTSEQUENCES] Daede, T., "Test Sets", n.d.,
<<https://people.xiph.org/~tdaede/sets/>>.
- [VMAF] Aaron, A., Li, Z., Manohara, M., Lin, J., Wu, E., and C. Kuo, "VMAF - Video Multi-Method Assessment Fusion", 2015,
<<https://github.com/Netflix/vmaf>>.

Authors' Addresses

Thomas Daede
Mozilla

Email: tdaede@mozilla.com

Andrey Norkin
Netflix

Email: anorkin@netflix.com

Ilya Brailovskiy
Amazon Lab126

Email: brailovs@lab126.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 11, 2017

S. Midtskogen
A. Fuldseth
M. Zanaty
Cisco
March 10, 2017

Constrained Low Pass Filter
draft-midtskogen-netvc-clpf-04

Abstract

This document describes a low complexity filtering technique which is being used as a low pass loop filter in the Thor 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 September 11, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Definitions	2
2.1. Requirements Language	2
2.2. Terminology	2
3. Filtering Process	3
4. Further complexity considerations	6
5. Performance	6
6. IANA Considerations	7
7. Security Considerations	7
8. Acknowledgements	7
9. References	8
9.1. Normative References	8
9.2. Informative References	8
Authors' Addresses	8

1. Introduction

Modern video coding standards such as Thor [I-D.fuldseth-netvc-thor] include in-loop filters which correct artifacts introduced in the encoding process. Thor includes a deblocking filter which corrects artifacts introduced by the block based nature of the encoding process, and a low pass filter correcting artifacts not corrected by the deblocking filter, in particular artifacts introduced by quantisation errors of transform coefficients and by the interpolation filter. Since in-loop filters have to be applied in both the encoder and decoder, it is highly desirable that these filters have low computational complexity.

2. Definitions

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2.2. Terminology

This document will refer to a pixel X and eight of its neighbouring pixels A - H ordered in the following pattern.

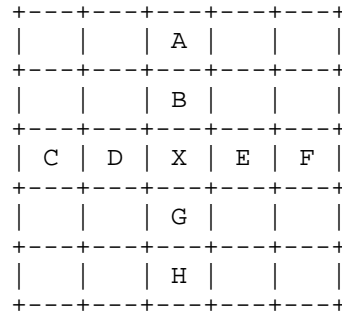


Figure 1: Filter pixel positions

In Thor the frames are divided into filter blocks (FB) of 128x128, 64x64 or 32x32 pixels, which is signalled for each frame to be filtered. Also, each frame is divided into coding blocks (CB) which range from 8x8 to 128x128 independent of the FB size. The filter described in this draft can be switched on or off for the entire frame or optionally on or off for each FB. CB's that have been coded using the skip mode are not filtered, and if a FB only contains CB's that have been coded in skip mode, the FB will not be filtered and no signal will be transmitted for this FB.

If the frame can't fit a whole number of FB's, the FB's at the right and bottom edges are clipped to fit. For instance, if the frame resolution is 1920x1080 and the FB size is 128x128, the size of the FB's at the bottom of the frame becomes 128x56.

3. Filtering Process

Given a pixel X and its neighbouring pixels described above we can define a general non-linear filter as:

$$\begin{aligned}
 X' = X &+ a*\text{constrain}(A-X) + b*\text{constrain}(B-X) + \\
 &c*\text{constrain}(C-X) + d*\text{constrain}(D-X) + \\
 &e*\text{constrain}(E-X) + f*\text{constrain}(F-X) + \\
 &g*\text{constrain}(G-X) + h*\text{constrain}(H-X)
 \end{aligned}$$

Figure 2: Equation 1

where constrain(x) is a function limiting the range of x.

If a neighbour pixel is outside the image frame, it is given the same value as the closest pixel within the frame. To avoid dependencies

prohibiting parallel processing, all neighbour pixels must be the unfiltered pixels of the frame being filtered.

Experiments in Thor have shown that a good compromise between complexity and performance is $a=c=f=h=1/16$, $b=d=e=g=3/16$, and a good constrain function has been found to be:

```
constrain(x, s, d) =
    sign(x) * max(0, abs(x) - max(0, abs(x) - s +
                                (abs(x) >> (d - log2(s))))))
```

Figure 3: Equation 2

where $\text{sign}(x)$ returns 1 or -1 if x is positive or negative respectively, s denotes the strength of the filter by which x will be clipped, and d further constrains the range of x so that the output of the function will linearly approach 0 as $\text{abs}(x)$ approaches 2^d . d depends on the frame quality (qp) and is computed as

$$d = \text{bitdepth} - 4 + qp/16$$

Figure 4: Equation 4

for the luma plane and

$$d = \text{bitdepth} - 5 + qp/16$$

Figure 5: Equation 5

for the chroma planes.

The constrain function can be visualised as follows

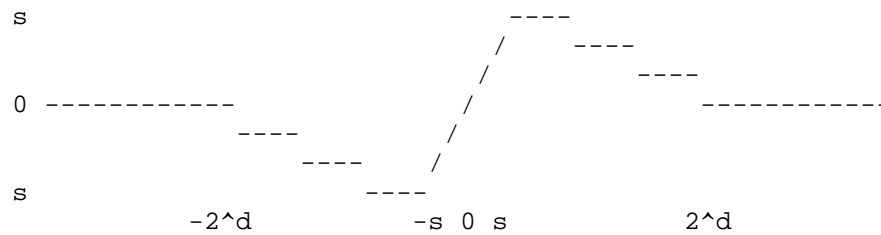


Figure 6: Graph 1

The filter strength s can be 1, 2 or 4 signalled at frame level when the bitdepth is 8. The strengths are scaled according to the bitdepth, so they become 4, 8 and 16 when the bitdepth is 10, and 16, 32 and 64 when the bitdepth is 12. The rounding is to the nearest integer.

This gives us the equation:

$$X' = X + (1 \cdot \text{constrain}(A-X, s, d) + 3 \cdot \text{constrain}(B-X, s, d) + 1 \cdot \text{constrain}(C-X, s, d) + 3 \cdot \text{constrain}(D-X, s, d) + 3 \cdot \text{constrain}(E-X, s, d) + 1 \cdot \text{constrain}(F-X, s, d) + 3 \cdot \text{constrain}(G-X, s, d) + 1 \cdot \text{constrain}(H-X, s, d))$$

Figure 7: Equation 6

The filter leaves the encoder 13 different choices for a frame. The filter can be disabled for the entire frame, or the frame is filtered using all distinct combinations of strength (1, 2 or 4 scaled for bitdepth), non-skip FB signal (enabled/disabled) and FB size (32x32, 64x64 or 128x128). Note that the FB size only matters when FB signalling is in use.

The decisions at both frame level and FB level may be based on rate-distortion optimisation (RDO), but an encoder running in a low-complexity mode, or possibly a low-delay mode, may instead assume that a fixed mode will be beneficial. In general, using $s=2$, a QP dependent FB size and RDO only at the FB level gives good results.

However, because of the low complexity of the filter, fully RDO based decisions are not costly. The distortion of the 13 configurations of the filter can easily be computed in a single pass by keeping track of the distortions of the three different strengths and the bit costs for different FB sizes.

The filter is applied after the deblocking filter.

4. Further complexity considerations

The filter has been designed to offer the best compromise between low complexity and performance. All operations are easily vectorised with SIMD instructions and if the video input is 8 bit, all SIMD operations can have 8 bit lanes in architectures such as x86/SSE4 and ARM/NEON. Clipping at frame borders can be implemented using shuffle instructions.

5. Performance

The table below shows filters effect on the bandwidth for a selection of 10 second video sequences encoded in Thor with uni-prediction only. The numbers have been computed using the Bjontegaard Delta Rate (BDR). BDR-low and BDR-high indicate the effect at low and high bitrates, respectively, as described in BDR [BDR].

The effect of the filter was tested in two encoder low-delay configurations: high complexity in which the encoder strongly favours compression efficiency over CPU usage, and medium complexity which is more suited for real-time applications. The bandwidth reduction is somewhat less in the high complexity configuration.

	MEDIUM COMPLEXITY			HIGH COMPLEXITY		
Sequence	BDR	BDR-low	BDR-high	BDR	BDR-low	BDR-high
Kimono	-2.6%	-2.3%	-3.2%	-1.7%	-1.7%	-1.9%
BasketballDrive	-3.9%	-3.1%	-5.0%	-2.8%	-2.4%	-3.5%
BQTerrace	-7.4%	-4.0%	-10.2%	-4.8%	-2.4%	-6.8%
FourPeople	-5.5%	-3.9%	-8.2%	-3.8%	-2.9%	-5.1%
Johnny	-5.2%	-3.5%	-8.2%	-3.3%	-2.7%	-4.5%
ChangeSeats	-6.4%	-3.9%	-10.2%	-4.7%	-2.6%	-6.9%
HeadAndShoulder	-9.3%	-3.4%	-19.6%	-6.2%	-2.6%	-11.8%
TelePresence	-5.8%	-3.6%	-10.0%	-4.5%	-3.3%	-6.6%
Average	-5.8%	-3.5%	-9.3%	-4.0%	-2.6%	-5.9%

Figure 8: Compression Performance without Biprediction

While the filter objectively performs better at relatively high bitrates, the subjective effect seems better at relatively low bitrates, and overall the subjective effect seems better than what the objective numbers suggest.

If biprediction is allowed, there is generally less bandwidth reduction as the table below shows. These results reflect low-delay biprediction without frame reordering.

	MEDIUM COMPLEXITY			HIGH COMPLEXITY		
Sequence	BDR	BDR- low	BDR- high	BDR	BDR- low	BDR- high
Kimono	-2.2%	-2.0%	-2.7%	-1.4%	-1.3%	-1.5%
BasketballDrive	-3.1%	-3.0%	-3.3%	-1.9%	-2.0%	-1.7%
BQTerrace	-5.4%	-4.3%	-6.5%	-3.9%	-3.6%	-3.8%
FourPeople	-3.8%	-2.8%	-5.2%	-2.4%	-1.8%	-3.0%
Johnny	-3.8%	-3.1%	-4.8%	-2.4%	-2.2%	-2.7%
ChangeSeats	-4.4%	-3.1%	-6.5%	-3.2%	-2.6%	-3.9%
HeadAndShoulder	-4.8%	-3.0%	-8.1%	-3.0%	-2.7%	-3.7%
TelePresence	-3.4%	-2.3%	-5.5%	-2.2%	-1.7%	-3.1%
Average	-3.9%	-2.9%	-5.3%	-2.5%	-2.2%	-2.9%

Figure 9: Compression Performance with Biprediction

6. IANA Considerations

This document has no IANA considerations yet. TBD

7. Security Considerations

This document has no security considerations yet. TBD

8. Acknowledgements

The authors would like to thank Gisle Bjontegaard for reviewing this document and design, and providing constructive feedback and direction.

9. References

9.1. Normative References

- [I-D.fuldseth-netvc-thor]
Fuldseth, A., Bjontegaard, G., Midtskogen, S., Davies, T.,
and M. Zanaty, "Thor Video Codec", draft-fuldseth-netvc-
thor-03 (work in progress), October 2016.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,
<<http://www.rfc-editor.org/info/rfc2119>>.

9.2. Informative References

- [BDR] Bjontegaard, G., "Calculation of average PSNR differences
between RD-curves", ITU-T SG16 Q6 VCEG-M33 , April 2001.

Authors' Addresses

Steinar Midtskogen
Cisco
Lysaker
Norway

Email: stemidts@cisco.com

Arild Fuldseth
Cisco
Lysaker
Norway

Email: arilfuld@cisco.com

Mo Zanaty
Cisco
RTP, NC
USA

Email: mzanaty@cisco.com

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.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Entropy Coding	2
2.1. Non-binary Arithmetic Coding	4
2.2. Non-binary Context Modeling	5
2.3. Dyadic Adaptation	6
2.4. Simplified Partition Function	9
2.5. Context Adaptation	11
2.5.1. Implicit Adaptation	11
2.5.2. Explicit Adaptation	12
2.5.3. Early Adaptation	12
2.6. Simple Experiment	13
3. Reversible Integer Transforms	14
3.1. Lifting Steps	14
3.2. 4-Point Transform	17
3.3. Larger Transforms	20
3.4. Walsh-Hadamard Transforms	20
4. Development Repository	22
5. IANA Considerations	22
6. Acknowledgments	22
7. References	22
7.1. Informative References	22
7.2. URIs	23
Authors' Addresses	24

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, $N \leq 16$) 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

$$f[k] = A * \langle \text{the number of times } k \text{ has been observed} \rangle + B$$

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

$$E[x[n+1]] = E[x[n]] + \text{floor}(C*(x[n] - E[x[n]]))$$

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.

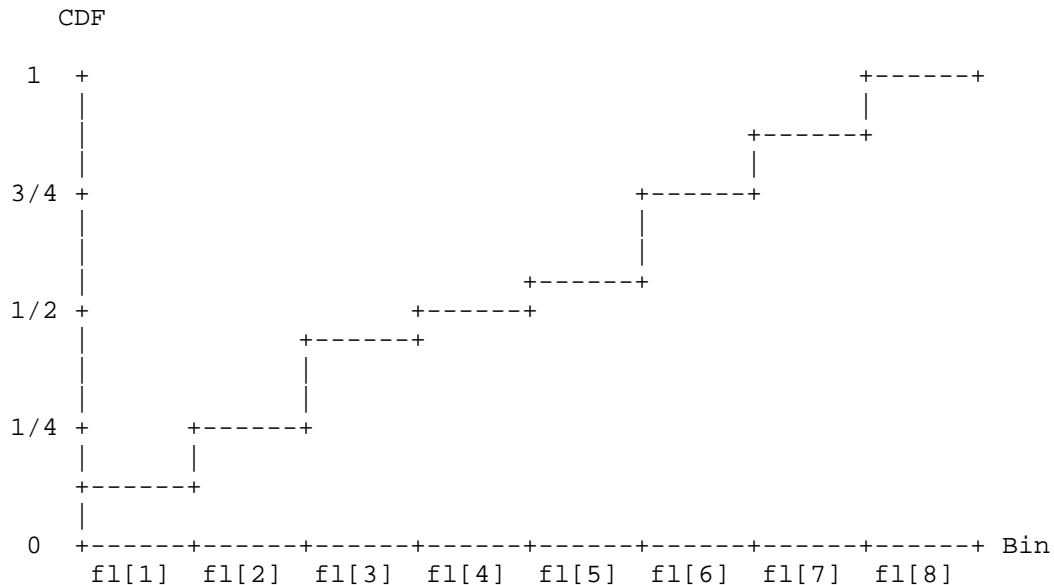
2.3. Dyadic Adaptation

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.

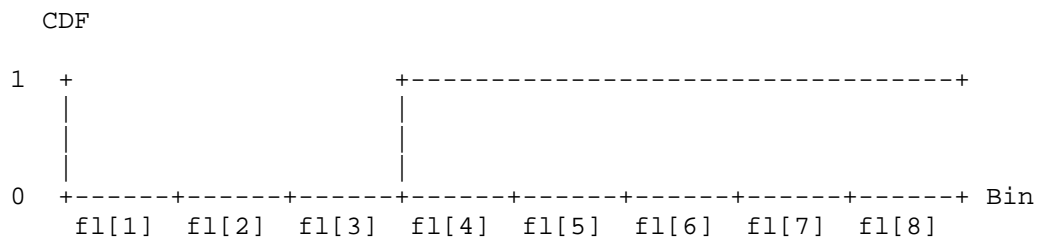
Suppose we have model for a given context that codes 8 symbols with the following probabilities:

+	-----+	-----+	-----+	-----+	-----+	-----+	-----+	-----+	+								
	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 $f_t = 2^4 = 16$, what we are computing is:

+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	
2	4	7	8	9	12	14	16		* (1 - g)
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	
+									
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	
0	0	0	16	16	16	16	16		* g
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	

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:

```

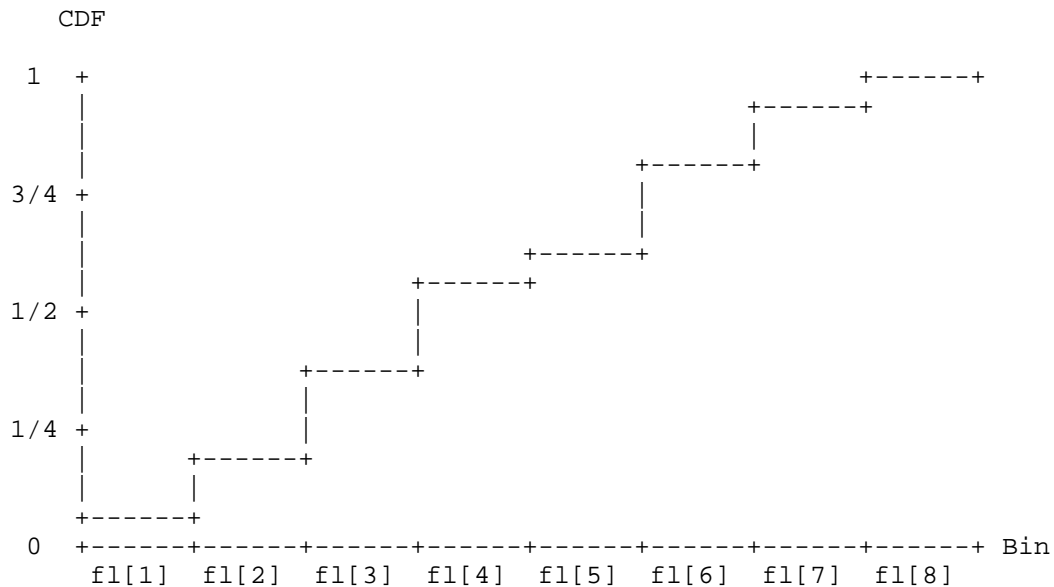
      ( fl[i] - floor((fl[i] + 2^r - i - 1)/2^r), i <= coded symbol
fl[i] = <
      ( fl[i] - floor((fl[i] + M - i - ft)/2^r), i > coded symbol

```

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

+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	
1	3	6	9	10	13	15	16		
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	

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:

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

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:

$$\begin{bmatrix} y0 \\ y1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x0 \\ x1 \end{bmatrix}$$

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

$$\begin{aligned}
 u_0 &= x_0 + \frac{2\cos(t) - \sqrt{v_2}}{\sin(t)} * x_1 \\
 y_1 &= x_1 + \frac{1}{v_2} * \sin(t) * u_0 \\
 y_0 &= u_0 + \frac{\cos(t) - \sqrt{v_2}}{\sin(t)} * y_1
 \end{aligned}$$

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]

$$\begin{aligned} \frac{1}{N} * E[e^T.e] &= \frac{1}{N} * E[x^T.D^T.D.x] \\ &= \frac{1}{N} * E[\text{tr}(D.x.x^T.D^T)] \\ &= \frac{1}{N} * E[\text{tr}(D.Rxx.D^T)] \end{aligned}$$

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 ρ . 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