Thor update

High Efficiency, Moderate Complexity Video Codec using only RF IPR

draft-fuldseth-netvc-thor-01
Steinar Midtskogen (Cisco)
IETF 94 – Yokohama, JP – November 2015

IPR note

https://datatracker.ietf.org/ipr/2636/

If technology in this document is included in a standard adopted by IETF and any claims of any Cisco patents are necessary for practicing the standard, any party will have the right to use any such patent claims royalty-free under reasonable, non-discriminatory terms, including defensive suspension, to implement and fully comply with the standard.

Topics for this update

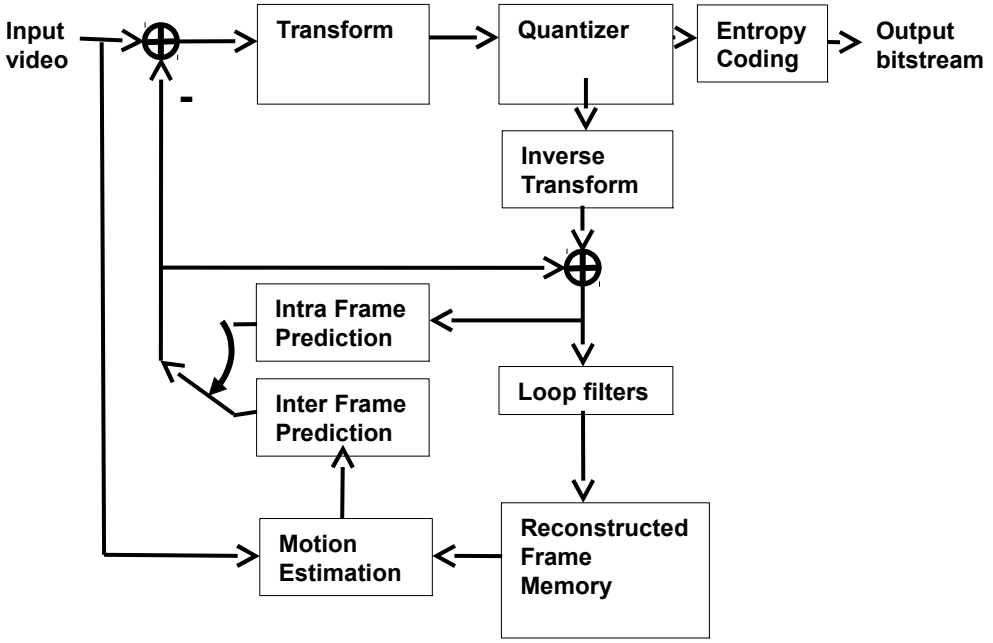
- Brief recap of the Thor design
- Changes since IETF93
  - Constrained low pass filter
  - Interpolated reference frames
- Optimisation and SIMD support
- Updated compression performance

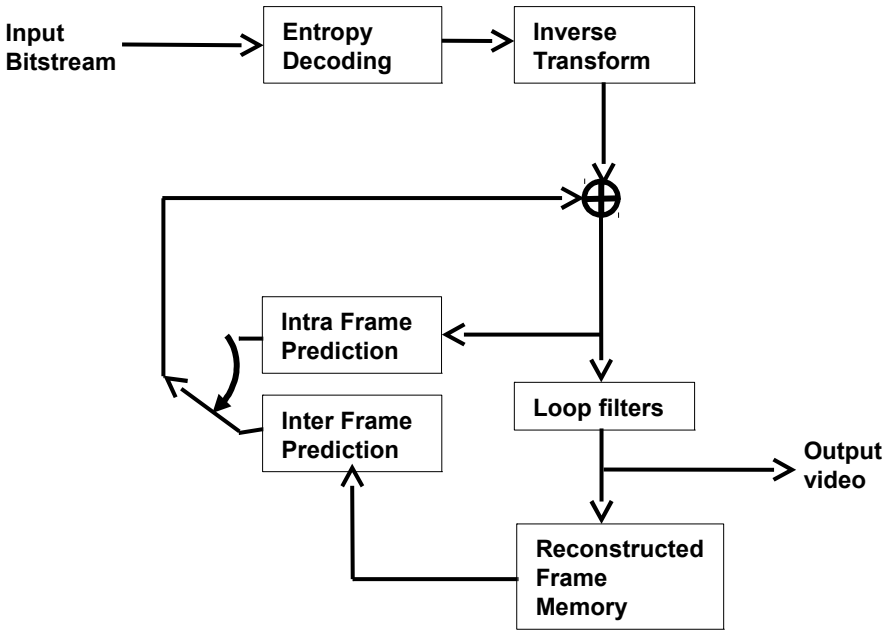
Design principles

- Moderate complexity to allow real-time implementation in software
- Favouring simplicity both in terms of computation and description
- Using techniques known to work well and improving on those
- Many similarities with H.26x
- Royalty free IPR

Encoder/decoder architecture

- The same basic architecture as H.261, H.263, H.264 and H.265





Block Structure

- Super block (SB) 64x64
- Quad-tree split into coding blocks (CB) >= 8x8
- Multiple prediction blocks (PB) per CB
  - Intra: 1 PB per CB
  - Inter: 1, 2 (rectangular) or 4 (square) PBs per CB
- 1 or 4 transform blocks (TB) per CB

Coding-block modes

- Intra
- Inter0 MV index, no residual information
- Inter1 MV index, residual information
- Inter2 Explicit motion vector information, residual information
- Bipred Explicit motion vector information (x2), residual information

Some difference from H.265

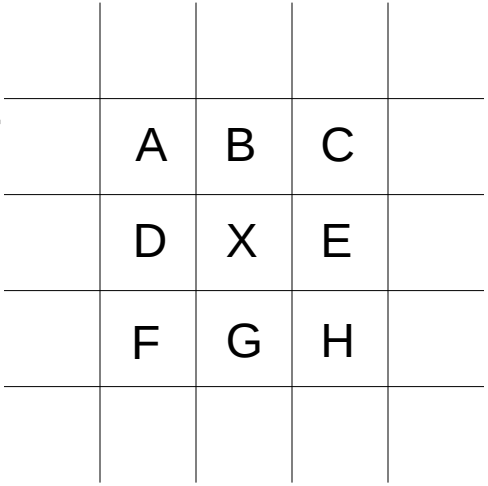
- Slightly shorter interpolation luma filter and a special non-separable filter for the (½, ½) position
- Fewer intra modes
- Simpler deblocking filter
- Simpler deringing filter
- VLC-based (non-arithmetic) entropy coding
- Temporally interpolated reference frames (never displayed)

Changes since IETF93/July 2015

- New constrained low pass filter
- Support for frame reordering
- Temporally interpolated reference frames (never displayed)
- Simplified 64x64 transform (32x32 and scaling)
- New filter coefficients
  - Different coefficients for uni-pred and bi-pred
- Various syntactic changes
- Major speed improvements (non-normative changes)
  - Motion estimation rewritten

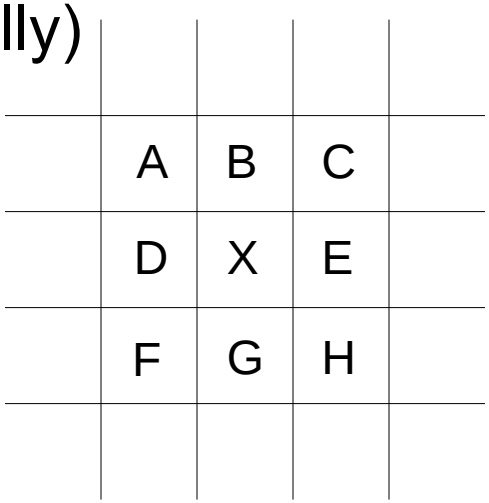
New constrained low-pass filter

- An attempt to reduce the problem into a lookup table
- Create an index using the pixel to be filtered and its neighbours
- Comparisons with 8 neighbours gives a relatively small table
- $I = (A>X)\cdot 2^0 + (B>X)\cdot 2^1 + (C>X)\cdot 2^2 + (D>X)\cdot 2^3 + (E>X)\cdot 2^4 + (F>X)\cdot 2^5 + (G>X)\cdot 2^6 + (H>X)\cdot 2^7$
- 256 entries
- Pixel weights or offsets?
  Most simple: a 0 or 1 offset



New constrained low-pass filter

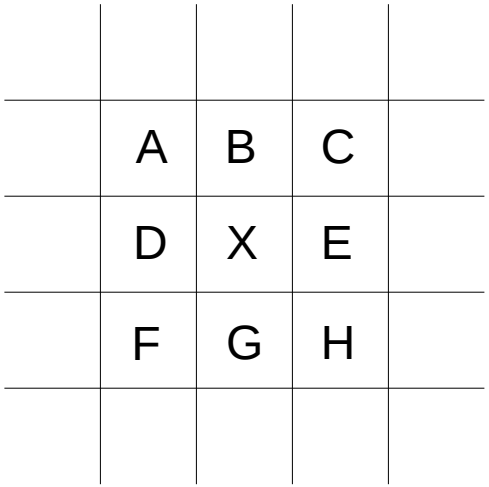
- An overnight script can create the table:
  Make all tables consisting of 255 0's and one 1, and record all 1's that give an improvement.
- It turns out that B, D, E and G are important.
  Comparing with A, C, F and H (diagonally) only give very small gains.
- Initial experiments using both > and >= operators to create an index were not convincing, but not fully explored.
- Signalled offsets higher than 1 give small gains.



New constrained low-pass filter

- A table with few 1's still giving most of the gain:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



- Equivalent to:
  X' = X + ((B>X)+(D>X)+(E>X)+(G>X) > 2)
- Increase by one if at least three of the four neighbours (up, left, right, down) are larger.
- Symmetry: X' = X + ((B>X)+(D>X)+(E>X)+(G>X) > 2) -
  ((B<X)+(D<X)+(E<X)+(G<X) > 2)

New constrained low-pass filter

- Pixels outside frame or block border: Give X's value
- Input pixels are always unfiltered to allow parallelism
- A very simple filter with little memory footprint and very well suited for SIMD instructions.
- Does not work well with bi-prediction. Probably because the bi-predictive averaging itself is a low-pass filter.

New constrained low-pass filter

- We don't want to filter everything!
- Flag at superblock (64x64) level indicates whether to filter the block or not
- Test using squared sum of differences
- Sub-blocks with no residual are not filtered
- Sub-blocks with bi-prediction are not filtered
- Superblocks with no residual or fully bi-predictive are implicitly unfiltered – no need spend a bit for the flag

New constrained low-pass filter

- Performs better than the previous filter and gives more consistent gains
- Subjective gains larger than objective gains
- The objective gains at low bitrates are small, so there is still room for improvements.

New constrained low-pass filter

Results with only uni-prediction:

Sequence	BDR	BDR (low br)	BDR (high br)
Kimono	-1.5%	-0.8%	-2.5%
BasketballDrive	-2.9%	-1.6%	-4.5%
BQTerrace	-6.6%	-3.8%	-8.0%
FourPeople	-4.5%	-2.3%	-8.0%
Johnny	-3.6%	-1.5%	-7.0%
ChangeSeats	-4.7%	-1.9%	-8.3%
HeadAndShoulder	-6.7%	-0.7%	-14.9%
TelePresence	-2.9%	-1.0%	-5.8%
Average	**-4.2%**	**-1.7%**	**-7.4%**

New constrained low-pass filter

Results with bi-prediction enabled:

Sequence	BDR	BDR (low br)	BDR (high br)
Kimono	-0.9%	-0.4%	-1.5%
BasketballDrive	-1.2%	-0.8%	-1.5%
BQTerrace	-1.6%	-1.1%	-2.0%
FourPeople	-2.5%	-1.5%	-3.7%
Johnny	-2.1%	-1.1%	-3.6%
ChangeSeats	-2.5%	-1.2%	-4.1%
HeadAndShoulder	-2.4%	-0.9%	-4.5%
TelePresence	-1.5%	-0.2%	-3.5%
Average	**-1.8%**	**-0.9%**	**-3.1%**

Interpolated reference frames

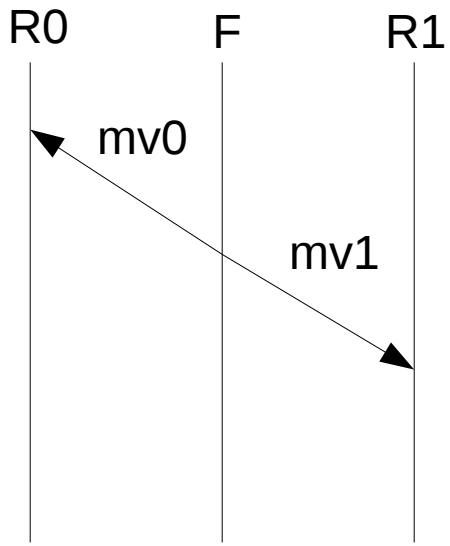
- Uses motion estimation between two frames to create a new reference frame
- For prediction only, never displayed (unless used to code a frame with no residual and no vectors).
- Motion estimation must be done in both the encoder and decoder
- Generally speeds up encoding (but not in the worst case), because we get a lot of skip blocks
- But adds complexity to the decoder

Interpolated reference frames

- Can be used for extrapolation (motion estimation between two past frames) and interpolation (between past and future frame, requires frame reordering)
- Only interpolation seems to give useful results
- Since the decoder has to perform the same motion estimation as the encoder, we need a fast and simple algorithm!

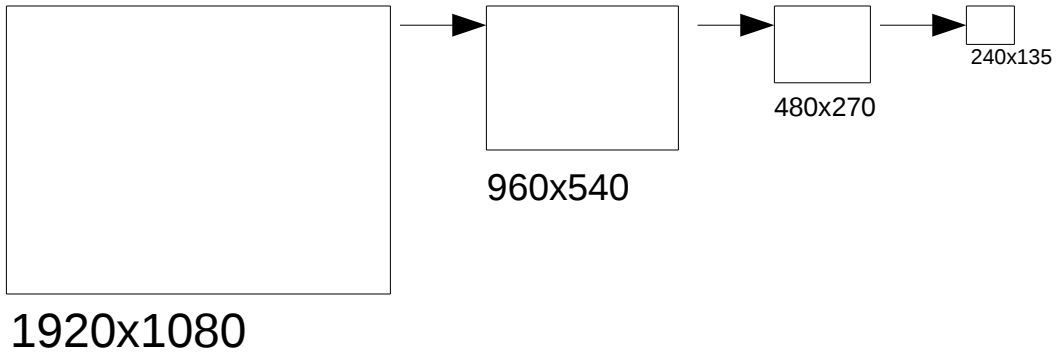
Interpolated reference frames

- The typical case: Two frames R0 and R1 and a frame F equidistant in time between them to be interpolated



Interpolated reference frames

- Both reference frames are repeatedly scaled down by a factor of ½ vertically and horisontally using the filter (½,½) up to 4 times (or until the frame cannot hold a 16x16 block)



Interpolated reference frames

- Start ME for the smallest frames and use motion vectors found as search candidates for the higher layer
- For each layer, the stages are as follows:
  - For each 16x16 in raster order
    - Check if ME can be bypassed
    - If not, get candidates from lower layer and neighbour blocks
    - Perform an adaptive cross search around each candidate vector and detemine the best vector. Up to 16 steps at lowest layer, else just 2.
  - For each 8x8 in raster order, find the best merge candidate, i.e. use the original 16x16 block vector or one of the neighbouring block vectors

Interpolated reference frames

- Bypass prediction is used to stabilise the mv field (i.e. prevent accidental matches) and reduce complexity.
- mv1 (and its derived mv0) are computed from neighbouring blocks (like a candidate vector)
- For each 8x8 block S calculate the SAD between S+mv0 in R0 and S+mv1 in R1 (luma and chroma).
- If all SADs are below a given threshold, further ME is bypassed
- Corresponds to early skip techniques used in encoders

Interpolated reference frames

- Adaptive cross search examines in each step 4 positions (left, right, up, down) with a displacement D.
- If none of them is better, divide D by two and try again. Otherwise, search again around the best position.
- D is 1 and the number of matches allowed is 8 (two steps), except at the lowest level where it is 64.
- The matching criterion (420 video):

  SAD(B0, B1) + 4*(SAD(U0, U1) + SAD(V0, V1)) + λ*mv_cost
  (B0 = b + mv0 in R0.luma, B1 = B + mv1 in R1.luma, etc)
- mv_cost is a measure of the disparity between the mv and neighbour vectors. λ is fixed for each layer.

Interpolated reference frames

Sequence	QP 22,27,32,37	QP 32,36,40,44
Kimono	-3.5%	-6.6%
ParkScene	-3.1%	-7.0%
Cactus	-4.9%	-8.9%
BasketballDrive	-2.1%	-5.5%
BQTerrace	-1.9%	-4.7%
ChangeSeats	-5.8%	-12.1%
HeadAndShoulder	-6.6%	-10.1%
TelePresence	-6.6%	-11.0%
WhiteBoard	-7.5%	-12.4%
FourPeople	-7.0%	-9.1%
Johnny	-6.2%	-8.0%
KristenAndSara	-7.0%	-9.9%
Average	**-5.2%**	**-8.8%**

SIMD optimisations

- We need to verify that Thor is “SIMD friendly” and can compete with other optimised codecs
- Supported by modern CPU's (x86: SSE2, SSE3, etc and ARM: NEON)
- Single instruction, multiple data
- Very useful for video processing
- Compilers are not (yet) good at redesigning code to match the instruction set
- Can we avoid having to maintain a separate set of function for different architectures?

SIMD optimisations

- Thor's solution: An abstraction layer for intrinsics
- Most compilers offer intrinsics to support SIMD instructions in the C code. Let the compiler do the register allocation!
- The most used instructions in different architectures such as x86 and ARM are identical
- So the abstraction layer is mostly an instruction name translator
- Support for 64 and 128 bit wide operands
- Does not always give optimal code, but close enough

SIMD optimisations

- An example: Add 16 pairs of bytes with a single instruction
  - x86/SSE2: `_mm_add_epi8(a, b)`
  - ARM/NEON: `vaddq_u8(a, b)`
- Thor: `v128_add_8(a, b)`
- Thor supports many instructions, but not everything
- Support for x86 and ARM, and C implementations to ease porting to new architectures
- Kernels in both SIMD and plain C as fallback. Bitexact.

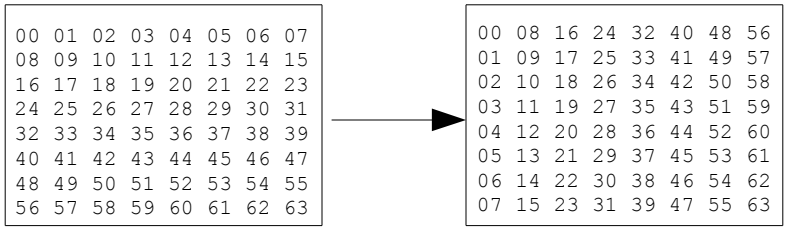
SIMD optimisations

```
void transpose8x8(const int16_t *src, int sstride, int16_t *dst, int dstride)
{
  v128 i0 = v128_load_aligned(src + sstride*0);
  v128 i1 = v128_load_aligned(src + sstride*1);
  v128 i2 = v128_load_aligned(src + sstride*2);
  v128 i3 = v128_load_aligned(src + sstride*3);
  v128 i4 = v128_load_aligned(src + sstride*4);
  v128 i5 = v128_load_aligned(src + sstride*5);
  v128 i6 = v128_load_aligned(src + sstride*6);
  v128 i7 = v128_load_aligned(src + sstride*7);

  v128 t0 = v128_ziplo_16(i1, i0);
  v128 t1 = v128_ziplo_16(i3, i2);
  v128 t2 = v128_ziplo_16(i5, i4);
  v128 t3 = v128_ziplo_16(i7, i6);
  v128 t4 = v128_ziphi_16(i1, i0);
  v128 t5 = v128_ziphi_16(i3, i2);
  v128 t6 = v128_ziphi_16(i5, i4);
  v128 t7 = v128_ziphi_16(i7, i6);

  i0 = v128_ziplo_32(t1, t0);
  i1 = v128_ziplo_32(t3, t2);
  i2 = v128_ziplo_32(t5, t4);
  i3 = v128_ziplo_32(t7, t6);
  i4 = v128_ziphi_32(t1, t0);
  i5 = v128_ziphi_32(t3, t2);
  i6 = v128_ziphi_32(t5, t4);
  i7 = v128_ziphi_32(t7, t6);

  v128_store_aligned(dst + dstride*0, v128_ziplo_64(i1, i0));
  v128_store_aligned(dst + dstride*1, v128_ziphi_64(i1, i0));
  v128_store_aligned(dst + dstride*2, v128_ziplo_64(i5, i4));
  v128_store_aligned(dst + dstride*3, v128_ziphi_64(i5, i4));
  v128_store_aligned(dst + dstride*4, v128_ziplo_64(i3, i2));
  v128_store_aligned(dst + dstride*5, v128_ziphi_64(i3, i2));
  v128_store_aligned(dst + dstride*6, v128_ziplo_64(i7, i6));
  v128_store_aligned(dst + dstride*7, v128_ziphi_64(i7, i6));
}
```



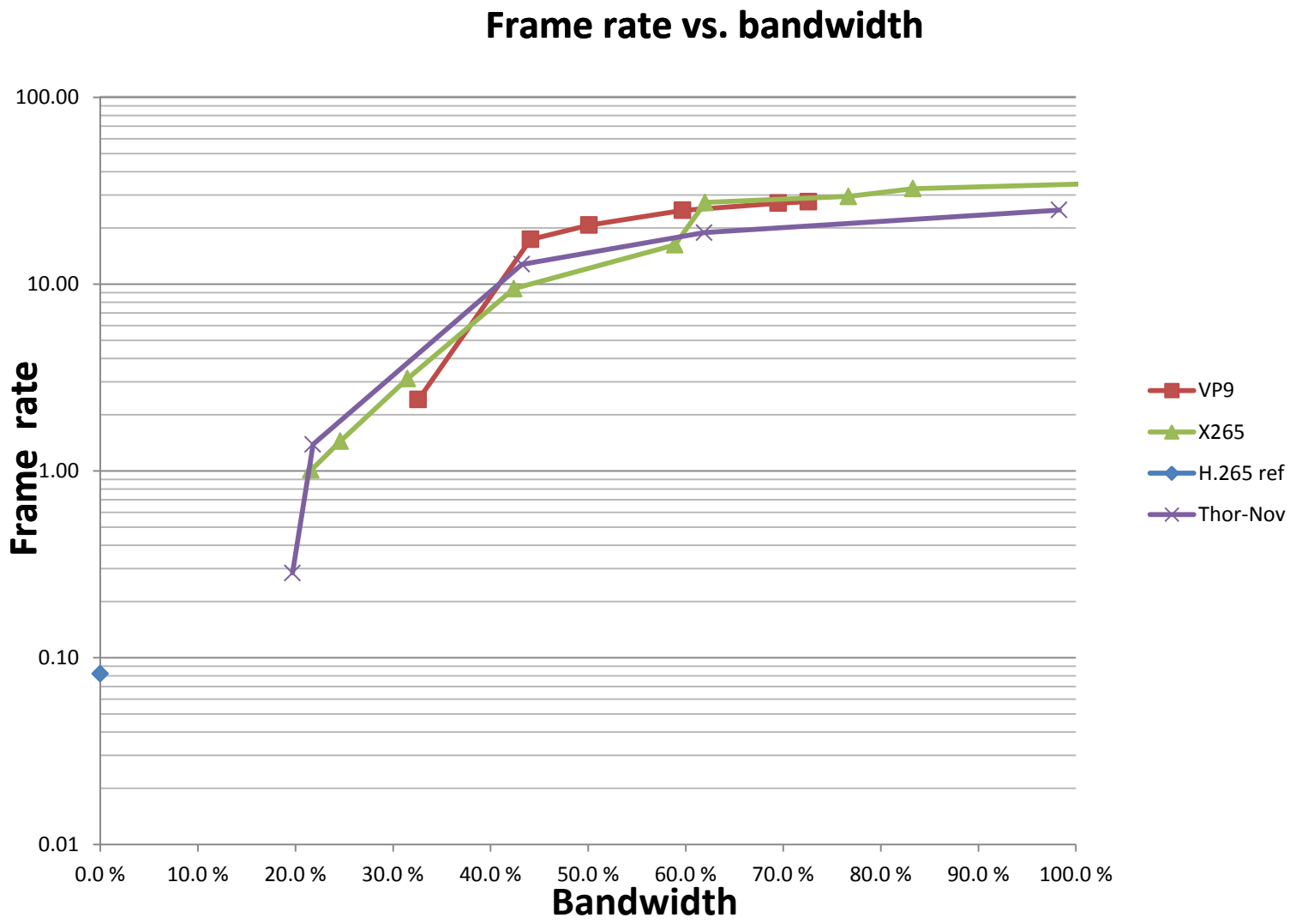
Performance, high delay

- Anchor:
  - HM13.0 (HEVC reference software)
  - Random access without periodic I frames
- Thor:
  - Same constraints as the anchor
- VP9: `-p 1 --cpu-used=0 --end-usage=q -cq-level=$q --auto-alt-ref=1 --disable-kf -y`
- x265: `-I -1 --no-wpp --tune psnr -p veryslow --qp $q`
- Complexity: FourPeople at QP 32 on a single core

Note: HM and Thor have fixed QP variation, x265 and VP9 adapt dynamically.

Performance, high delay

Class	Sequence	Thor	VP9	x265
Class B	Kimono	24.5%	49.3%	20.3%
	ParkScene	23.2%	45.4%	26.5%
	Cactus	17.5%	34.5%	17.2%
	BasketballDrive	31.3%	46.1%	13.3%
	BQTerrace	35.4%	51.5%	19.7%
Class E	FourPeople	8.8%	13.8%	26.7%
	Johnny	16.2%	38.6%	28.4%
	KristenAndSara	7.3%	16.8%	23.0%
Internal	ChangeSeats	20.9%	29.1%	18.3%
	HeadAndShoulder	10.9%	6.0%	21.0%
	TelePresence	22.6%	45.1%	20.0%
	WhiteBoard	15.6%	22.0%	24.9%
	Average	**19.5%**	**33.2%**	**21.6%**


Frame rate vs. bandwidth
100.00
10.00
1.00
0.10
0.01
Frame rate
0.0 %
10.0 %
20.0 %
30.0 %
40.0 %
50.0 %
60.0 %
70.0 %
80.0 %
90.0 %
100.0 %
Bandwidth
VP9
X265
H.265 ref
Thor-Nov

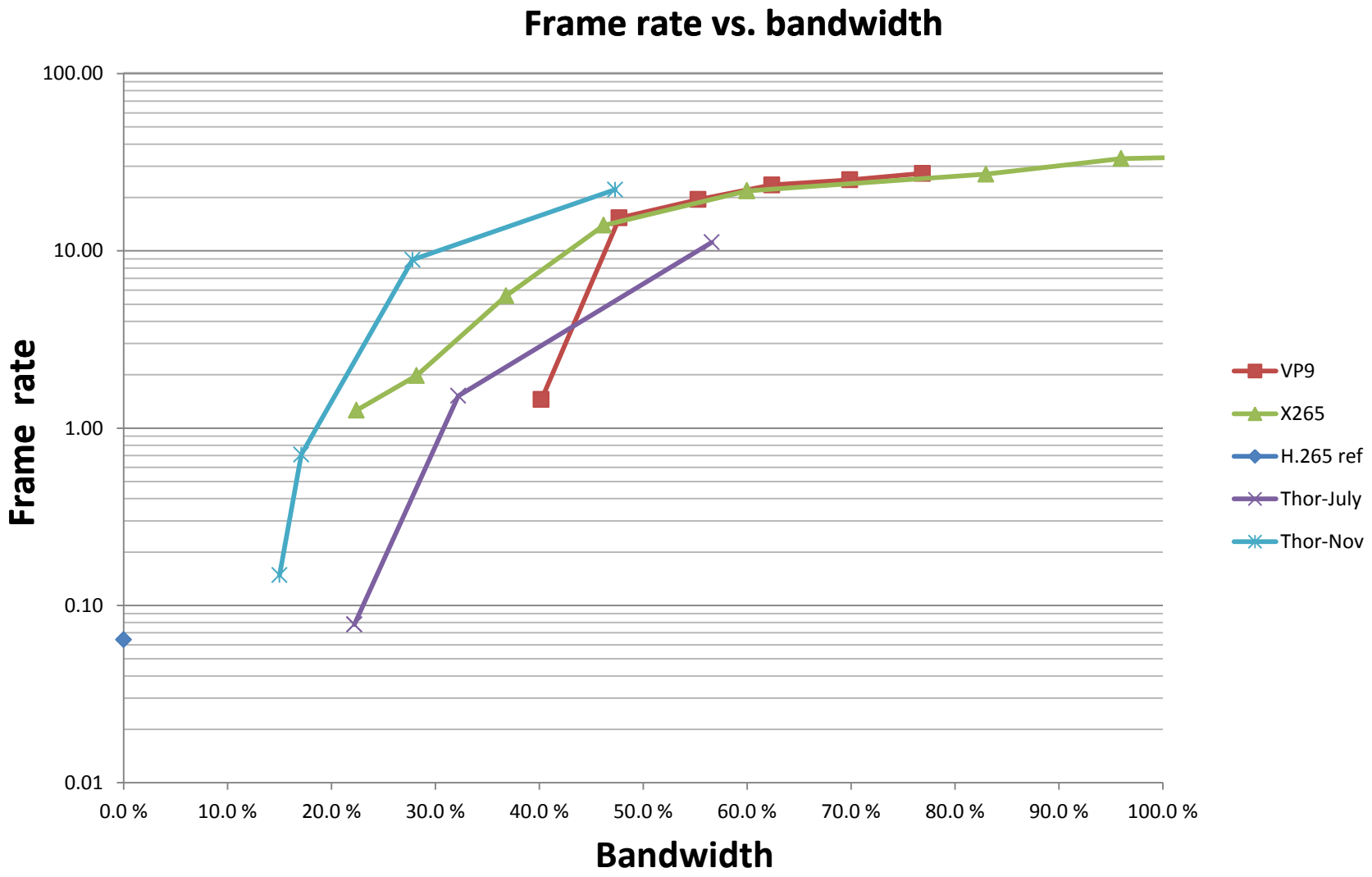
Performance, low delay

- Anchor:
  - HM13.0 (HEVC reference software)
  - Low-delay B configuration
- Thor:
  - Same constraints as the anchor
- VP9: `-p 1 --cpu-used=0 --end-usage=q --cq-level=$q -auto-alt-ref=0 --lag-in-frames=0 --disable-kf -y`
- x265: `-I -1 --no-wpp --bframes 0 --tune psnr -p veryslow --qp $q --qpfile $q.txt`
- Complexity: FourPeople at QP 32 on a single core

Note: HM, Thor and x265 have fixed QP variation, VP9 adapts dynamically.

Performance, low delay

Class	Sequence	Thor	VP9	x265
Class B	Kimono	16.1%	21.7%	14.1%
	ParkScene	19.5%	31.4%	16.4%
	Cactus	16.6%	26.6%	21.5%
	BasketballDrive	26.9%	32.9%	14.0%
	BQTerrace	31.8%	84.1%	44.9%
Class E	FourPeople	6.6%	35.5%	22.5%
	Johnny	12.0%	66.9%	30.8%
	KristenAndSara	4.7%	36.9%	20.3%
Internal	ChangeSeats	14.4%	20.5%	12.8%
	HeadAndShoulder	2.5%	59.8%	34.8%
	TelePresence	15.1%	25.3%	11.9%
	WhiteBoard	11.1%	43.8%	24.3%
	Average	**14.8%**	**40.5%**	**22.4%**


Frame rate vs. bandwidth
100.00
10.00
1.00
0.10
0.01
Frame rate
0.0 %
10.0 %
20.0 %
30.0 %
40.0 %
50.0 %
60.0 %
70.0 %
80.0 %
90.0 %
100.0 %
Bandwidth
VP9
X265
H.265 ref
Thor-July
Thor-Nov

Source Code

- Available at: github.com/cisco/thor