Introduction to Video Coding
Part 1: Transform Coding

Video Compression Overview

- Most codecs use the same basic ideas

1) Motion Compensation to eliminate temporal redundancy

Input

⊖

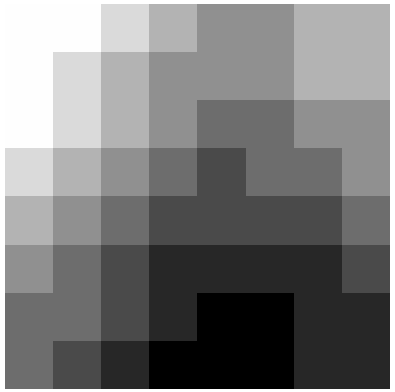
Reference frame

=

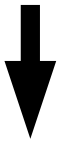
Residual

Video Compression Overview

2) A 2D transform (usually the DCT) to eliminate spatial redundancy



Input Data							
156	144	125	109	102	106	114	121
151	138	120	104	97	100	109	116
141	129	110	94	87	91	99	106
128	116	97	82	75	78	86	93
114	102	84	68	61	64	73	80
102	89	71	55	48	51	60	67
92	80	61	45	38	42	50	57
86	74	56	40	33	36	45	52



Transformed Data							
700	100	100	0	0	0	0	0
200	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

Video Compression Overview

3) Quantization to throw out unimportant details (this is the “lossy” part)

4) Lossless compression to store the result efficiently

The Ideal Linear Transform

- Karhunen-Loève Transform (KLT)
  - See also: Principal Component Analysis (PCA)
  - Just a change-of-basis (like any other linear transform)
    - Transforms, e.g., an 8×8 block of pixels into 64 coefficients in another basis
  - Goal: A *sparse* representation of the pixel data
  - Pick basis vectors one by one *minimizing* the distance of the data from the subspace they span
    - Equivalently: *maximizing* the percent of the data's variance contained in that subspace

Karhunen-Loève Transform

- Mathematically:
  - Compute the covariance matrix

$$R_{xx} = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - \mu)^T \cdot (x_i - \mu)$$

  - Compute the eigenvectors of R_{xx}
    - Sort by magnitudes of the eigenvalues
  - Project pixel data onto the eigenvectors
- Transform is *data-dependent*
  - So we need data to estimate it from
  - And would need to transmit the eigenvectors

Transforming Natural Images

- Image data is highly correlated
  - Usually modeled as a first-order *autoregressive* process (an AR(1) process)

$$x_i = \rho\, x_{i-1} + \sigma, \quad \rho = 0.95 \text{ (typically)}$$

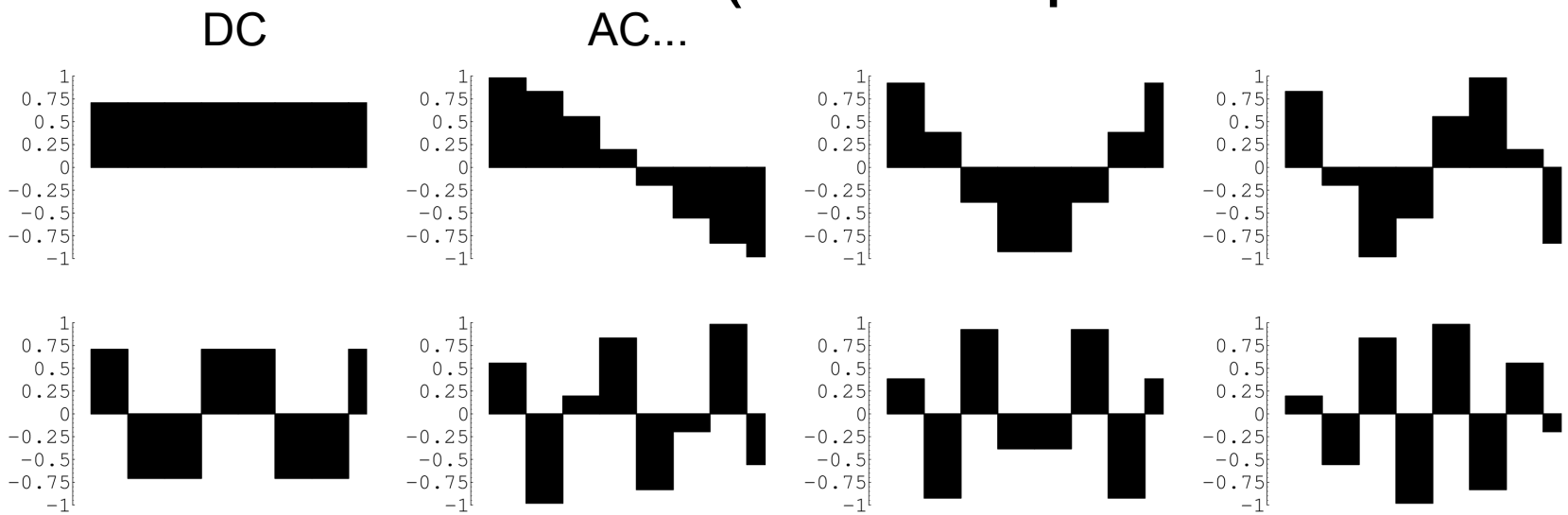
Correlation Coefficient — Gaussian Noise

  - Produces a simple cross-correlation matrix:

$$R_{xx} = \begin{bmatrix} 1 & \rho & \rho^2 & \rho^3 & \cdots \\ \rho & 1 & \rho & \rho^2 & \\ \rho^2 & \rho & 1 & \rho & \\ \rho^3 & \rho^2 & \rho & 1 & \\ \vdots & & & & \ddots \end{bmatrix}$$

The Discrete Cosine Transform

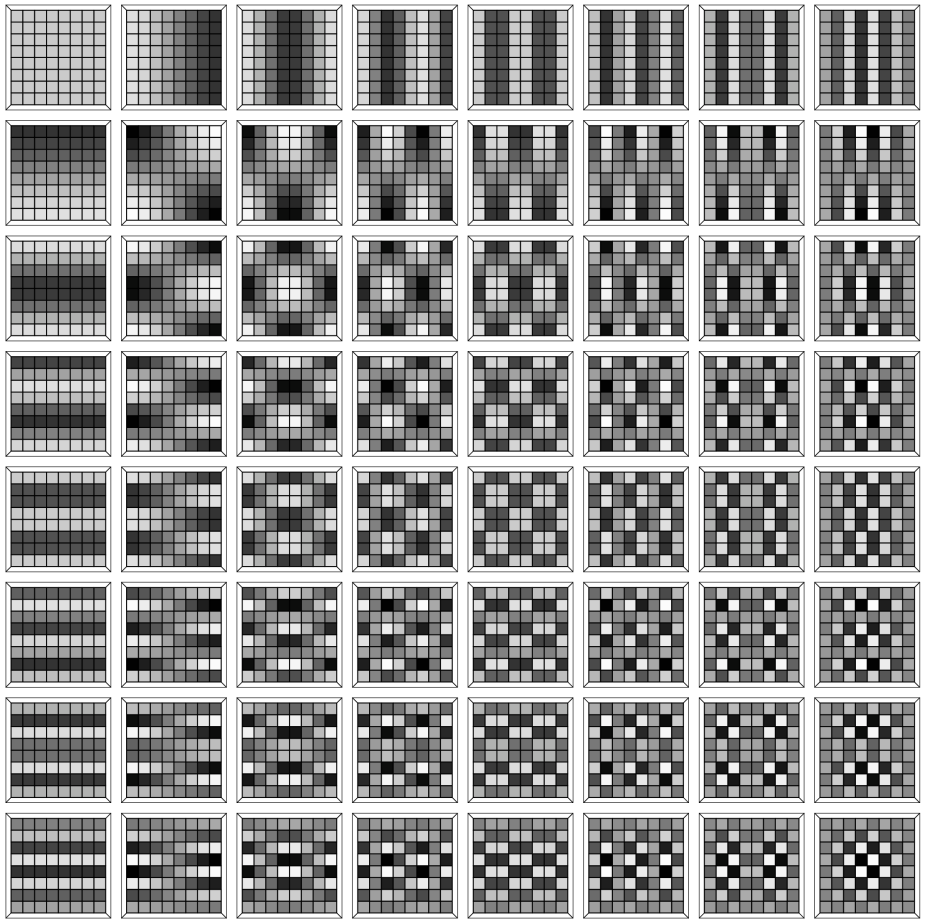
- If we assume this model holds for all image blocks, can design one transform in advance
  - This is the Discrete Cosine Transform (DCT)
- 1-D Basis Functions (for an 8-point transform):



- Orthonormal, so inverse is just the transpose

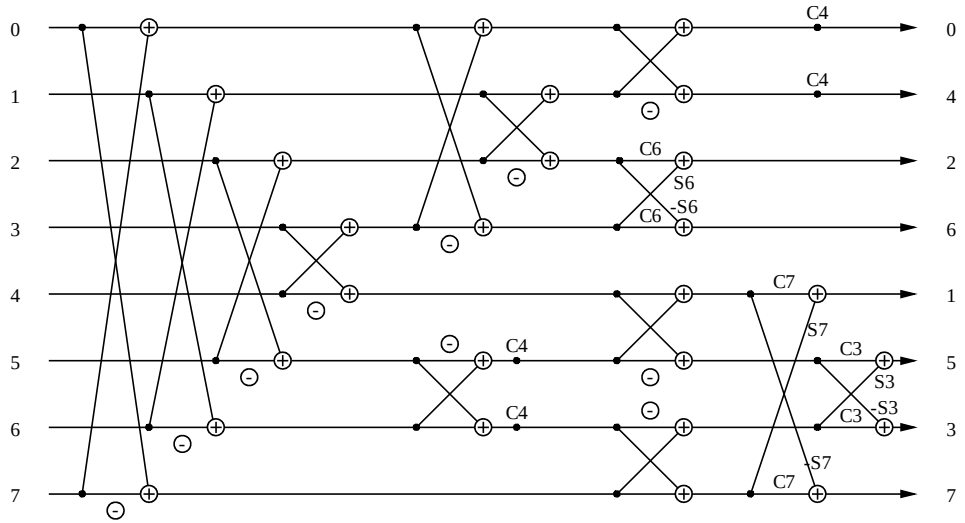
The DCT in 2D

- In 2D, first transform rows, then columns
  - $\mathbf{Y} = \mathbf{G} \cdot \mathbf{X} \cdot \mathbf{G}^T$
- Basis functions:
- Two 8x8 matrix multiplies is 1024 mults, 896 adds
  - 16 mults/pixel



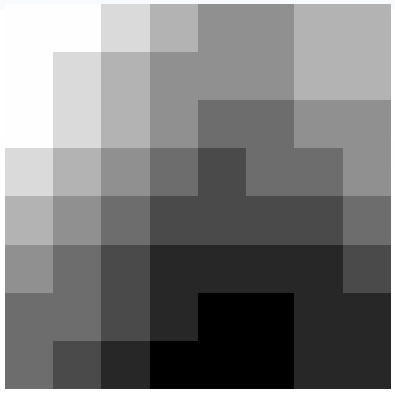
Fast DCT

- The DCT is closely related to the Fourier Transform, so there is also a fast decomposition
- 1-D: 16 mults, 26 adds



- 2-D: 256 mults, 416 adds (4 mults/pixel)

DCT Example



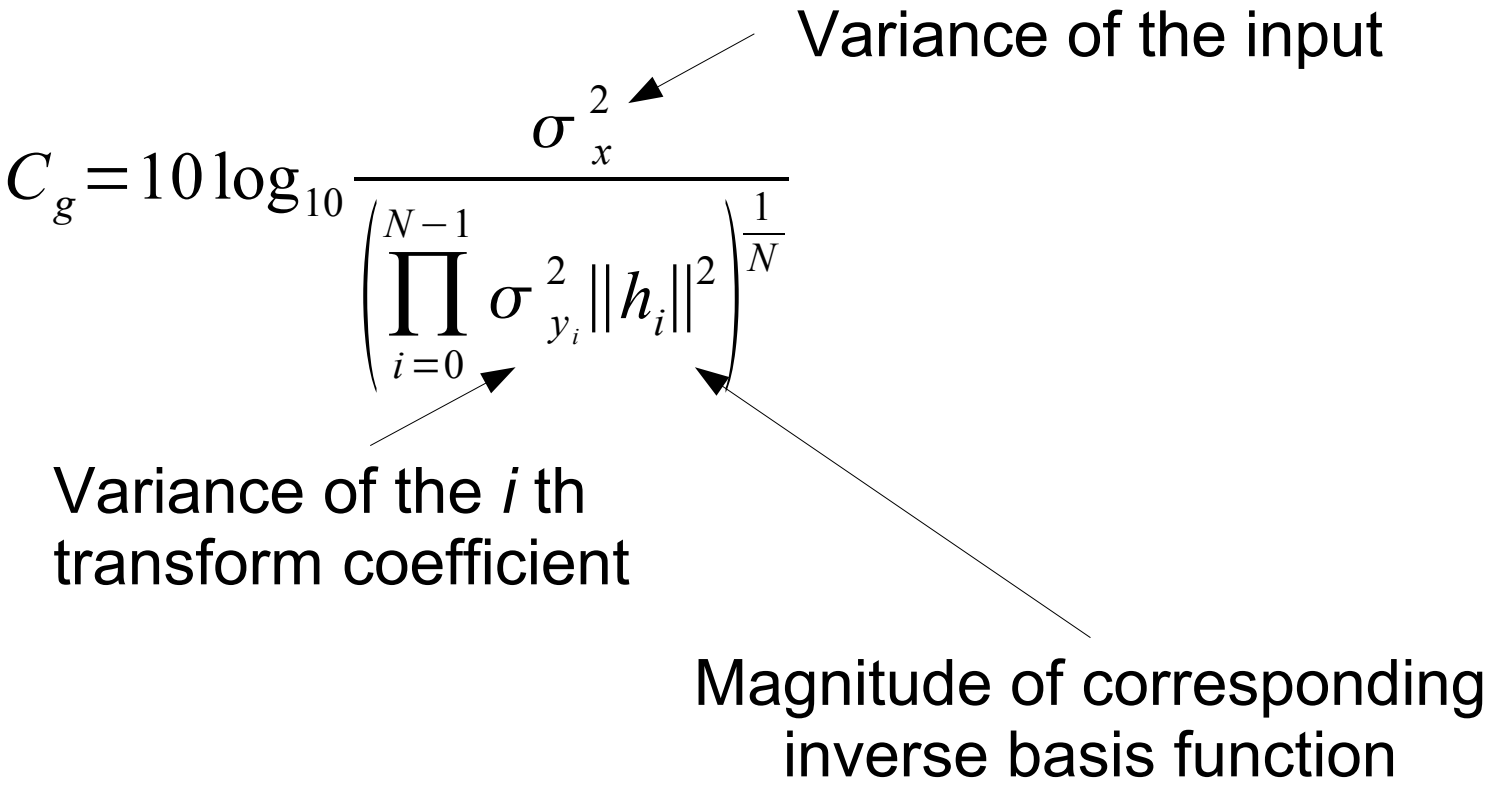
Input Data							
156	144	125	109	102	106	114	121
151	138	120	104	97	100	109	116
141	129	110	94	87	91	99	106
128	116	97	82	75	78	86	93
114	102	84	68	61	64	73	80
102	89	71	55	48	51	60	67
92	80	61	45	38	42	50	57
86	74	56	40	33	36	45	52

Transformed Data							
700	100	100	0	0	0	0	0
200	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

Shamelessly stolen from the MIT 6.837 lecture notes:
http://groups.csail.mit.edu/graphics/classes/6.837/F01/Lecture03/Slide30.html

Coding Gain

- Measures how well energy is compacted into a few coefficients



Coding Gain (cotd.)

- Calculating from a transform matrix and a cross-correlation matrix:

$$C_g = 10 \log_{10} \frac{1}{\left(\prod_{i=0}^{N-1} (G R_{xx} G^T)_{ii} \times (H^T H)_{ii} \right)^{\frac{1}{N}}}$$

- Coding gain for zero-mean, unit variance AR(1) process with $\rho = 0.95$

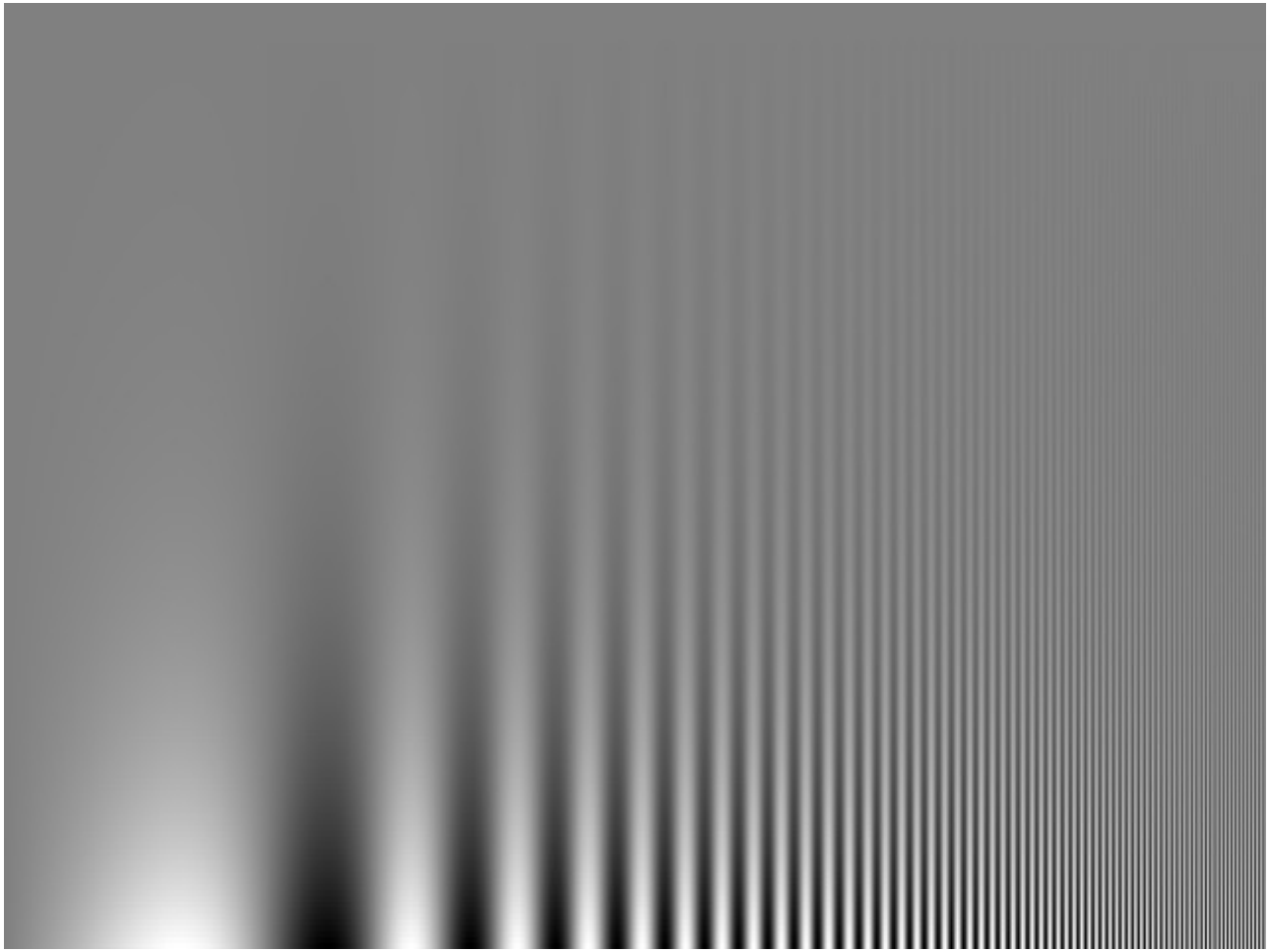
	4-point	8-point	16-point
KLT	7.5825 dB	8.8462 dB	9.4781 dB
DCT	7.5701 dB	8.8259 dB	9.4555 dB

Quantization

- Divide each coefficient by a *quantizer*
  - Integer division, with rounding
- Only required lossy step in the entire process
  - {700, 100, 200, 0, … }/32 → {22, 3, 6, 0, …}
  - Error: {22, 3, 6, 0, ...}×32 → {704, 96, 192, 0, ...}
- Resulting list has lots of zeros and small coefficients
  - That's what makes it easy to compress

The Contrast Sensitivity Function

- Contrast perception varies by spatial frequency



Quantization Matrices

- Choose quantizer for each coefficient according to the CSF
  - Example matrix:
- But that's at the visibility threshold
  - Above the threshold distribution more even
- Most codecs vary quantization by scaling a single base matrix
  - Will always be less than ideal at some rates
  - Theora has a more flexible model

Quantization Matrix							
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	58	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Blocking Artifacts

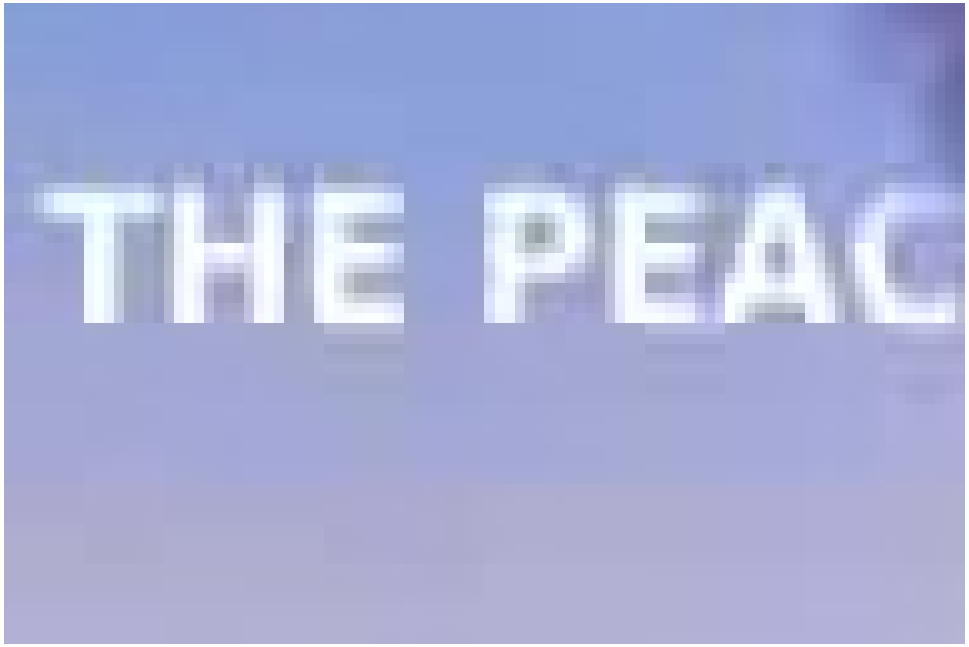
- When we have few bits, quantization errors may cause a step discontinuity between blocks
  - Error correlated along block edge → highly visible

Blocking Artifacts

- Standard solution: a “loop” filter
  - Move pixel values near block edges closer to each other

Ringing Artifacts

- Any quantization error is spread out across an entire block
- In smooth regions near edges, easily visible
- HEVC plans to add in-loop filters for this, too



Low-pass Artifacts

- HF coefficients get quantized more coarsely, and cost lots of bits to code
  - They're often omitted entirely
- Resulting image lacks details/texture
- Not often treated as a class of artifact
  - Low-pass behavior looks *good* on PSNR charts
    - This is one reason PSNR is a terrible metric

Low-pass Artifacts

- Low-passed skin tones (from original VP3 encoder)

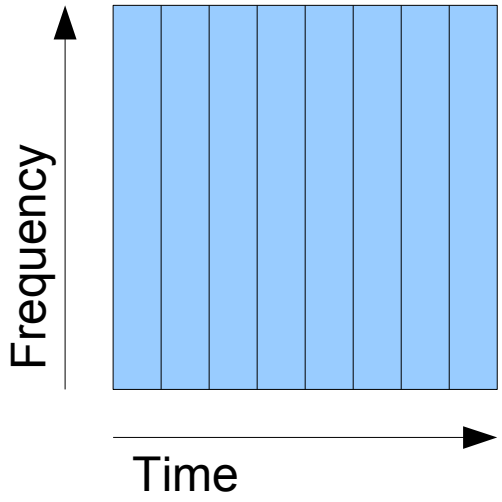
- Better retention of HF (from Thusnelda encoder)

Transform Size

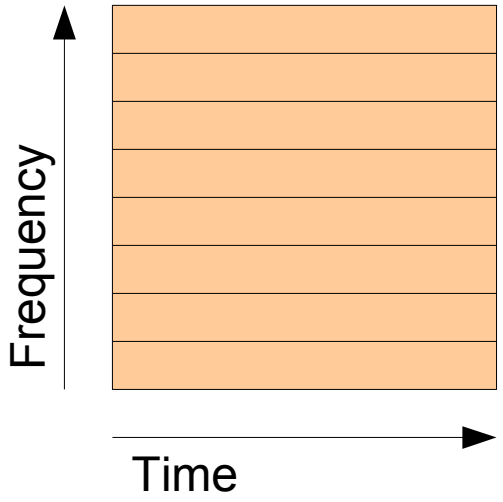
- How do we pick the size of the blocks?
  - Computation is *N* log *N*
    - Larger blocks require more operations per pixel
  - Larger blocks allow us to exploit more correlation
    - Better compression
  - Except when the data isn't correlated (edges)
    - Smaller blocks → fewer overlap edge → less ringing
- Variable block size
  - H.264: 8x8 and 4x4, HEVC: may add 16x16

The Time-Frequency Tradeoff

- Raw pixel data has good “time” (spatial) resolution



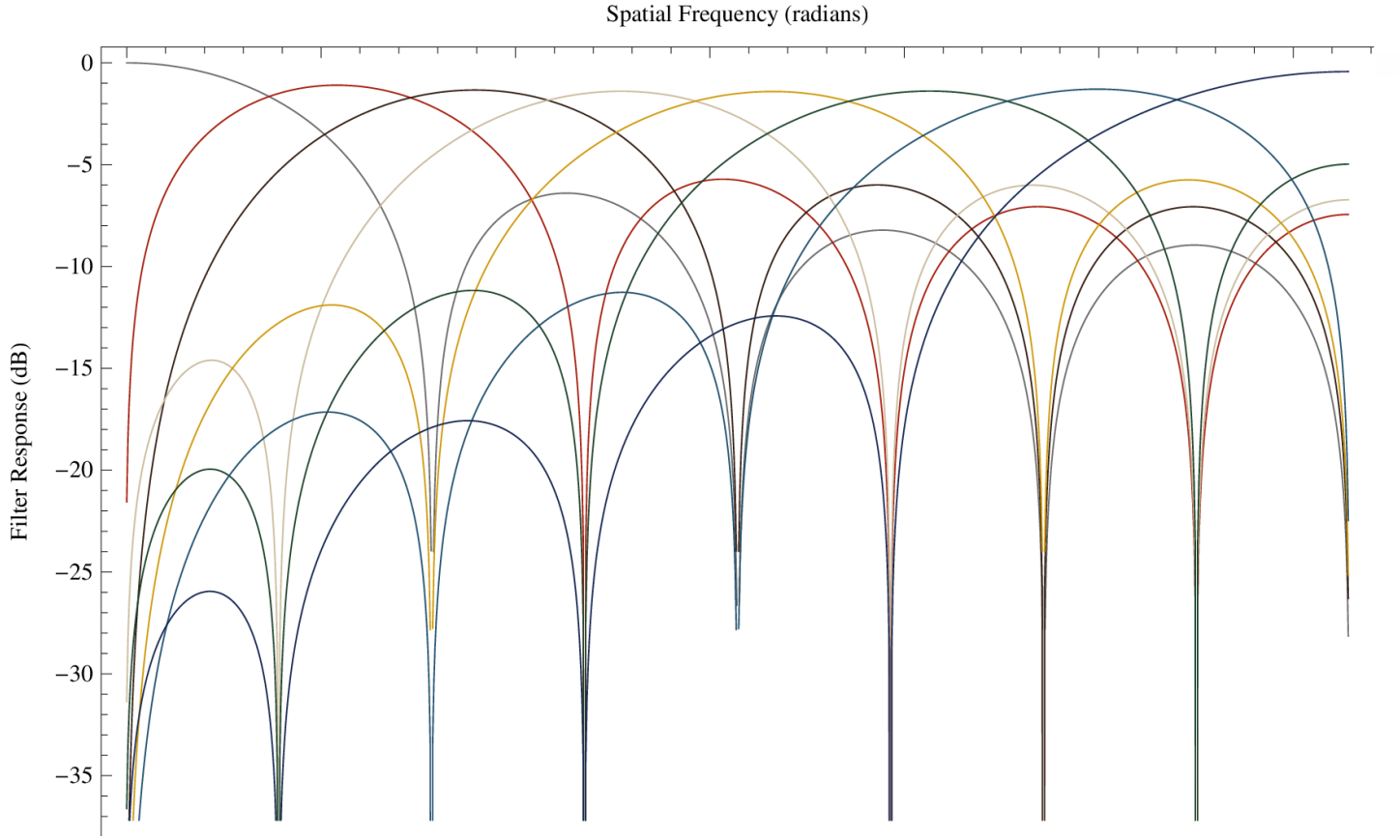
- The DCT has good frequency resolution



- Tilings of the “Time-Frequency Plane” depict this trade-off
  - $\Delta T \times \Delta F \geq$ *constant*

Leakage

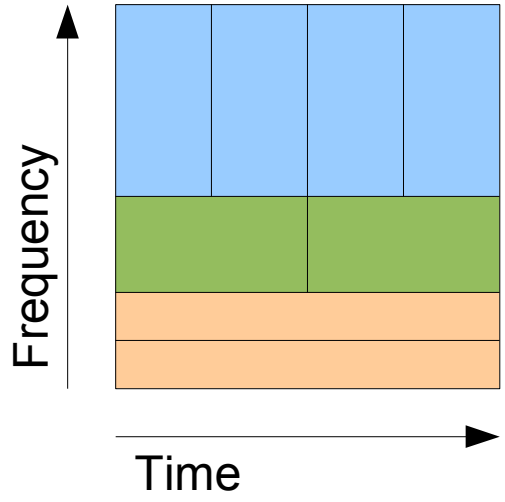
- In reality partition boundaries are not perfect



Frequency Response of 8-point DCT basis functions

Wavelets

- Good LF resolution (models correlation well)
- Better time resolution in HF (prevents ringing)
- Smooth basis functions (no blocking artifacts)



Wavelet tiling of the time-frequency plane

Wavelets (cotd.)

Original | Dirac @ 67.2 kbps | Theora @ 17.8 kbps

- Wavelets break down at low rates
  - HF “texture” requires spending *even more* bits to code it separately at every position
  - Extreme low-passing is typical
- Good for large-scale correlations (but Dirac doesn’t use them for this)

Wavelets (cotd.)

- Can do much better... but this is hard

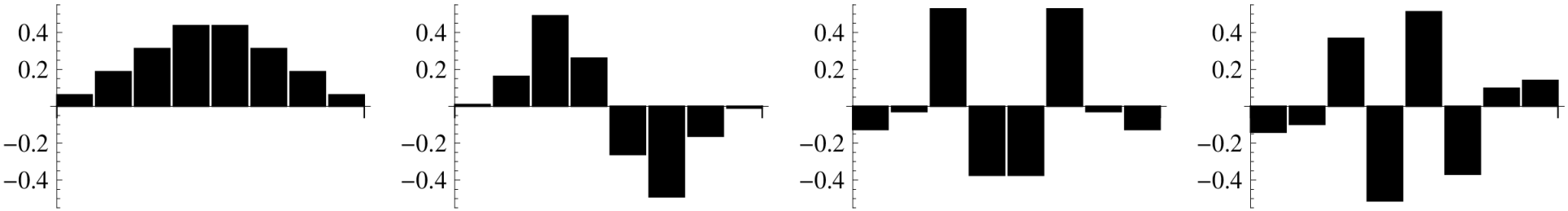
Fixed Quantizers

Contrast-based Adaptation

House at 0.1 bpp from D. M. Chandler and S. S. Hemami: “Contrast-Based Quantization and Rate Control for Wavelet-Coded Images.” In Proc. Of the 5th International Conference on Image Processing (ICIP’02), vol. 3, pp. 233–236, June 2002.

Lapped Transforms

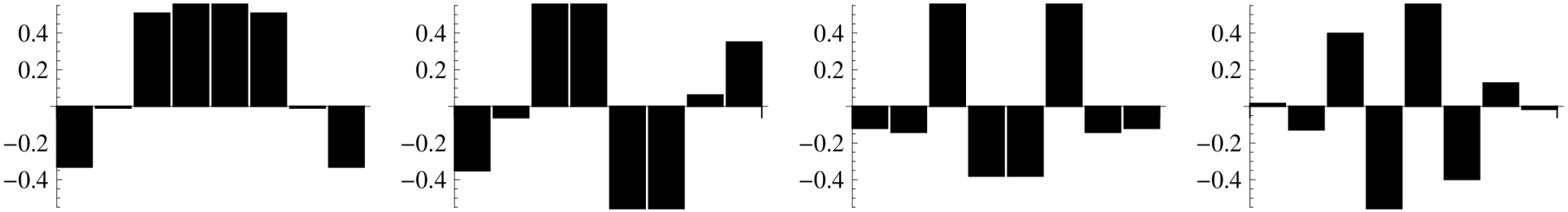
- Can eliminate blocking artifacts entirely
  - Like wavelets, but keep good DCT structure
  - Same idea has been used in audio forever
- Overlap basis functions with neighboring blocks
- Basis functions decay to zero at edges



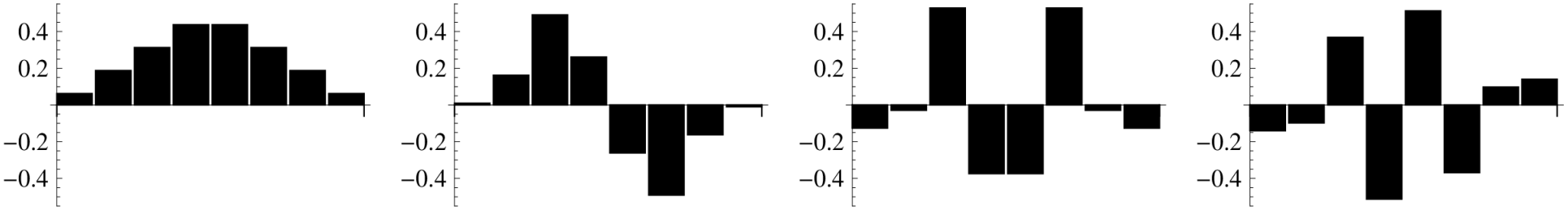
Synthesis basis functions for 4-point lapped transform with 50% overlap

Lapped Transforms

- No need for transform to be orthogonal
- Separate *analysis* and *synthesis* filters



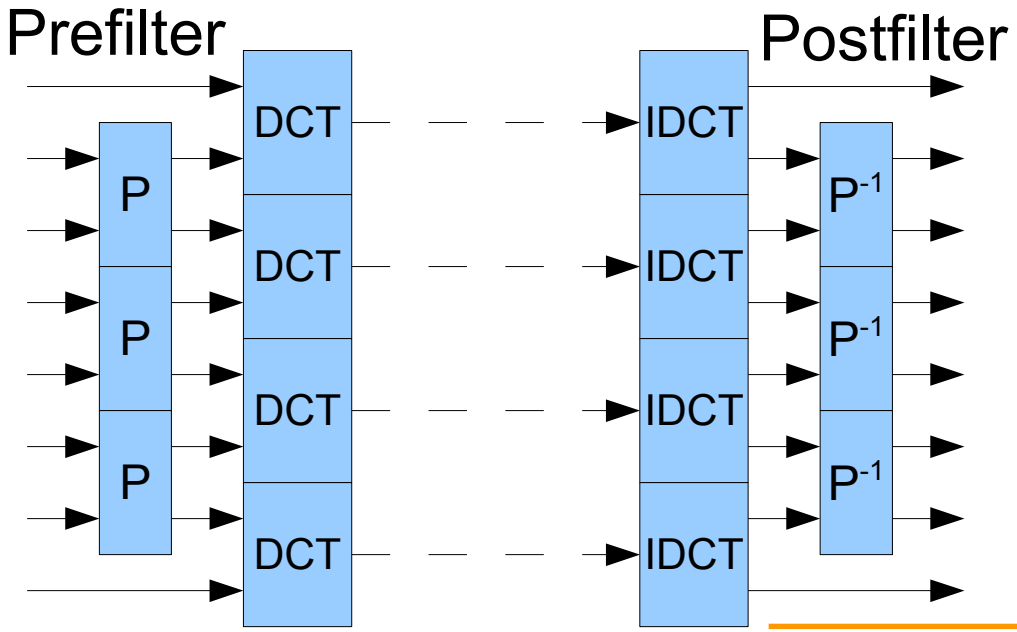
Analysis basis functions for 4-point lapped transform with 50% overlap



Synthesis basis functions for 4-point lapped transform with 50% overlap

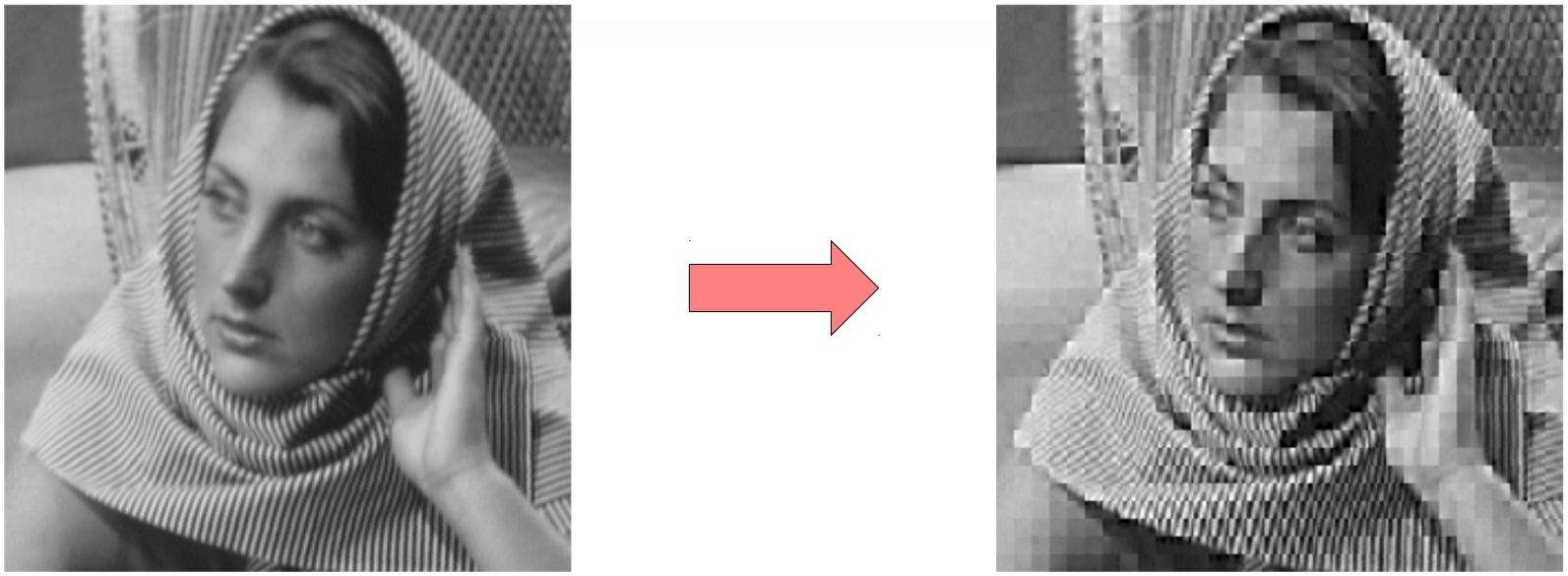
Lapped Transforms: Prefilter

- Implemented with a *prefilter* in the encoder
  - A linear transform that straddles block edges
  - Removes correlation across edge
  - Inverse applied in the decoder



Lapped Transforms: Prefilter

- Prefilter makes things blocky



- Postfilter removes blocking artifacts
  - Like loop filter, but *invertible*
    - And simpler: no conditional logic to control filter strength

Lapped Transforms

- Cheaper than wavelets
  - 8x16 LT: 3.375 muls/pixel (minus loop filter)
  - 9/7 Wavelet (3 levels): 5.25 muls/pixel
- Better compression

	4-point	8-point	16-point
KLT	7.5825 dB	8.8462 dB	9.4781 dB
DCT	7.5701 dB	8.8259 dB	9.4555 dB
LT	8.6060 dB	9.5572 dB	9.8614 dB
9/7 Wavelet		9.46 dB	

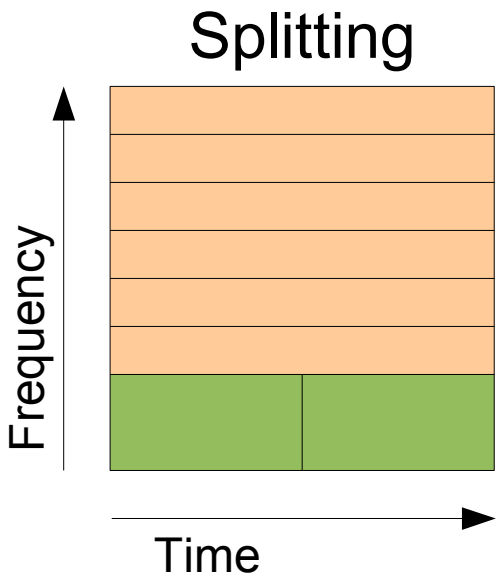
- Can keep most of block-based DCT infrastructure

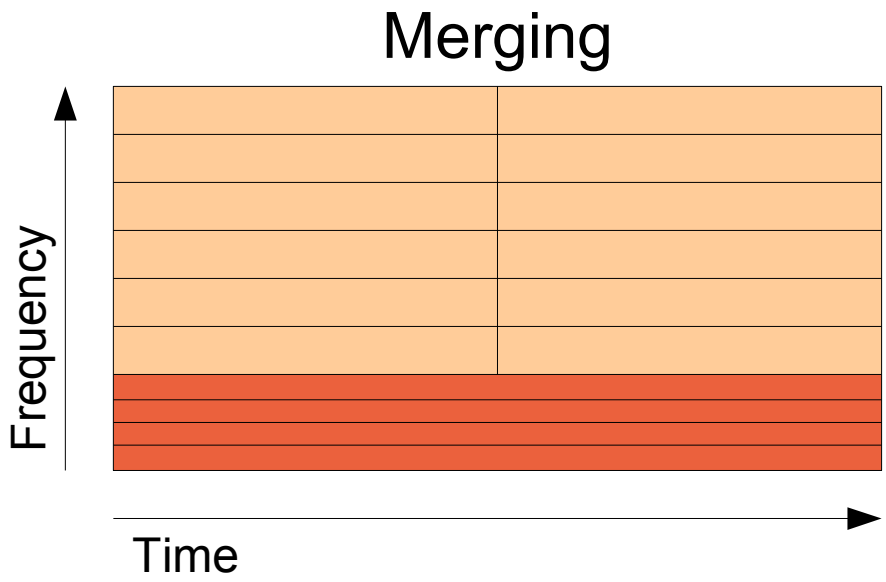
Lapped Transforms

- Early experiments (by Greg Maxwell) suggest it works!
- But others don’t think so
  - On2/Google (private communication)
  - Charles Bloom: http://cbloomrants.blogspot.com/2009/07/07-06-09-small-image-compression-notes.html
    - “Obviously in a few contrived cases it does help, such as on very smooth images at very high compression... In areas that aren't smooth, lapping actually makes artifacts like ringing worse. ”
    - This doesn’t match published examples: Barbara (heavily textured) gets much more benefit than Lena (smooth)

TF Resolution Switching

- Recursively apply DCT/IDCT to increase/decrease frequency resolution
  - Can apply to just part of the spectrum
  - Idea stolen from CELT/Opus (audio)



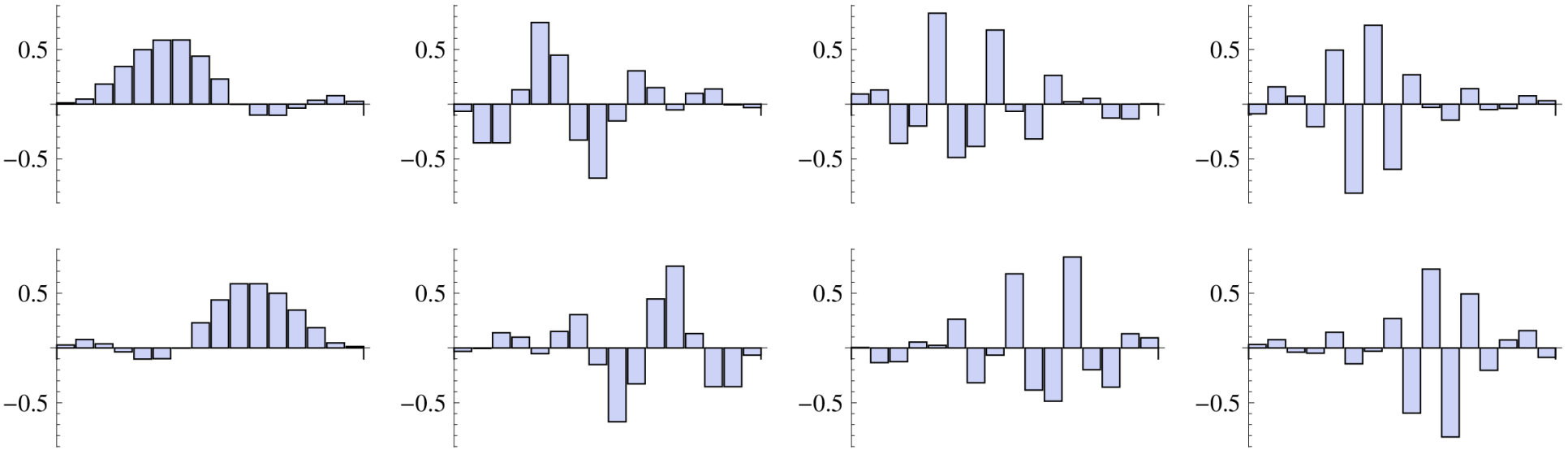


TF Resolution Switching

- Potential benefits
  - Can spread texture over larger regions
    - Only apply to HF: less complex than larger transform
  - Can capture large-scale correlation better
    - Like LF wavelet bands, but without the HF problems
  - Can reduce the number of transform sizes needed
- Cost
  - Signaling (need to code per-band decision)
  - Encoder search (more possibilities to consider)

TF Resolution: Splitting

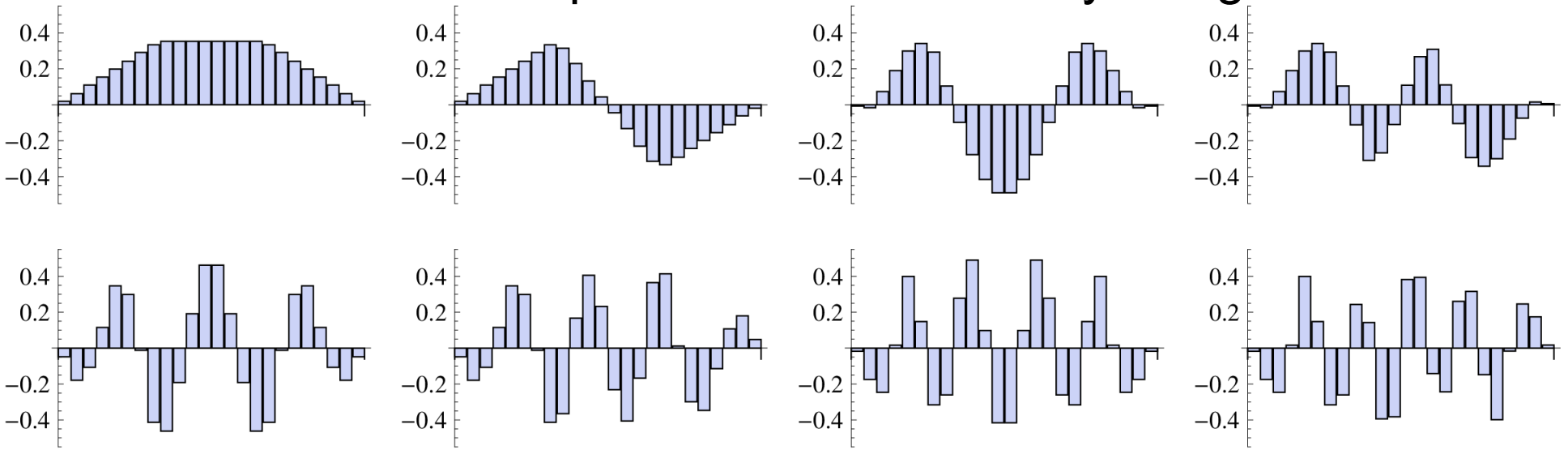
- Better time resolution → reduced ringing
  - But not as good as using a smaller transform



8-point Lapped Transform basis functions after TF splitting

TF Resolution: Merging

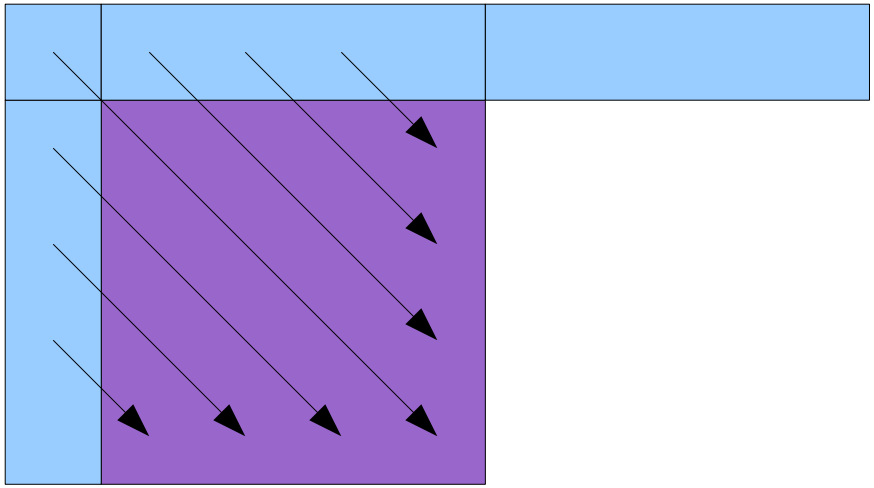
- Better frequency resolution → better coding gain
  - But not as good as a larger transform
    - 25% overlap vs. 50% → effectively using smaller window



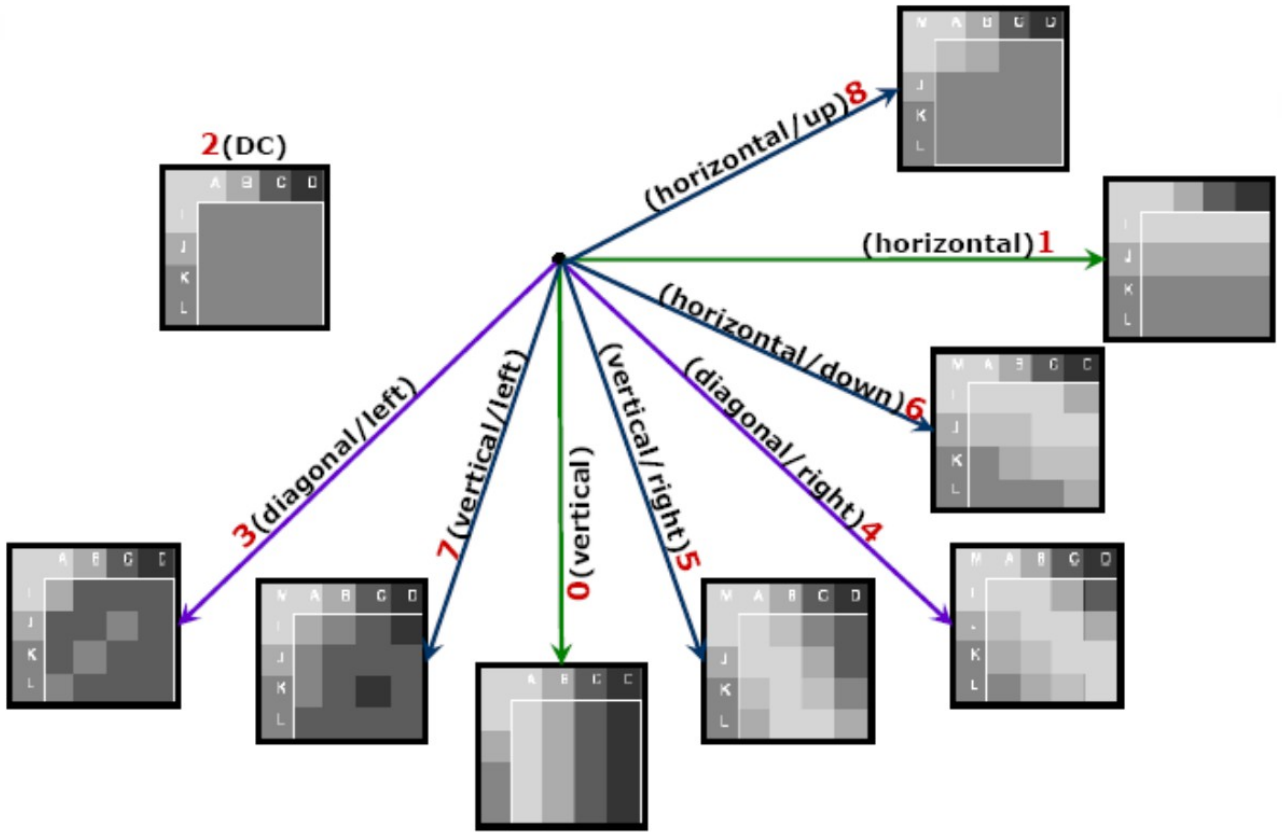
8-point Lapped Transform basis functions after TF merging
(first 8 only)

Intra Prediction

- Predict a block from its causal neighbors
- Explicitly code a *direction* along which to copy
- Extend boundary of neighbors into new block along this direction



Intra Prediction (cotd.)



The intra-prediction modes for 4×4 blocks in AVC/H.264

Intra Pred. vs. LT

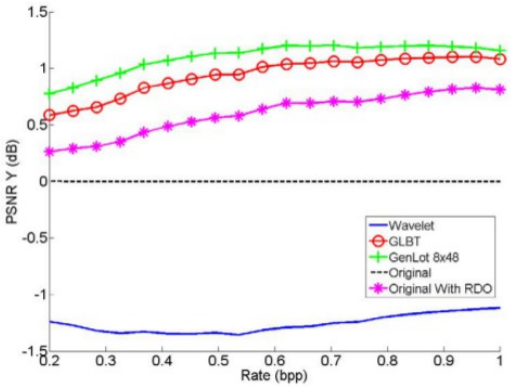


Fig. 4. Differential PSNR curves for image Barbara relative to the performance of the original H.264/AVC.

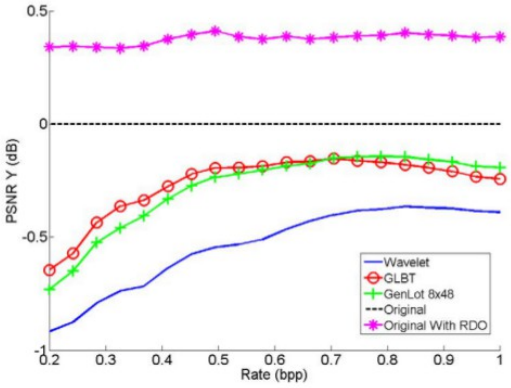


Fig. 5. Differential PSNR curves for image Lena relative to the performance of the original H.264/AVC.

- Intra pred. and LT have similar roles
  - Both exploit correlation with neighboring blocks
- But very different mechanisms
  - Best depends on image

R. G. de Oliveira and R. L. de Queiroz: "Intra Prediction versus Wavelets and Lapped Transforms in an H.264/AVC Coder." In Proc. 15th International Conference on Image Processing (ICIP'08), pp. 137–140, Oct. 2008.

Intra Pred. vs. LT

- Combine lapped transforms with intra pred.
  - Better than either alone
  - Despite predicting from farther away with only 4 of 9 modes (pixels not available for others)

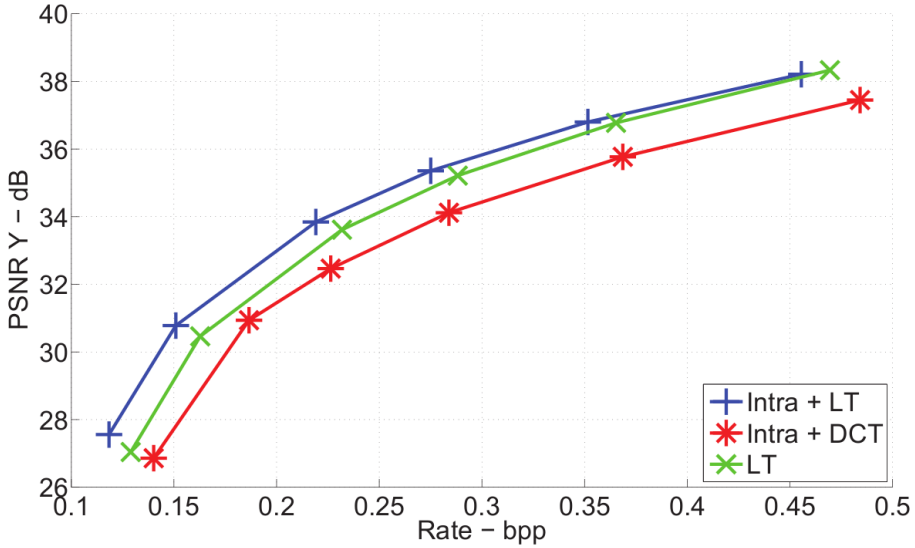
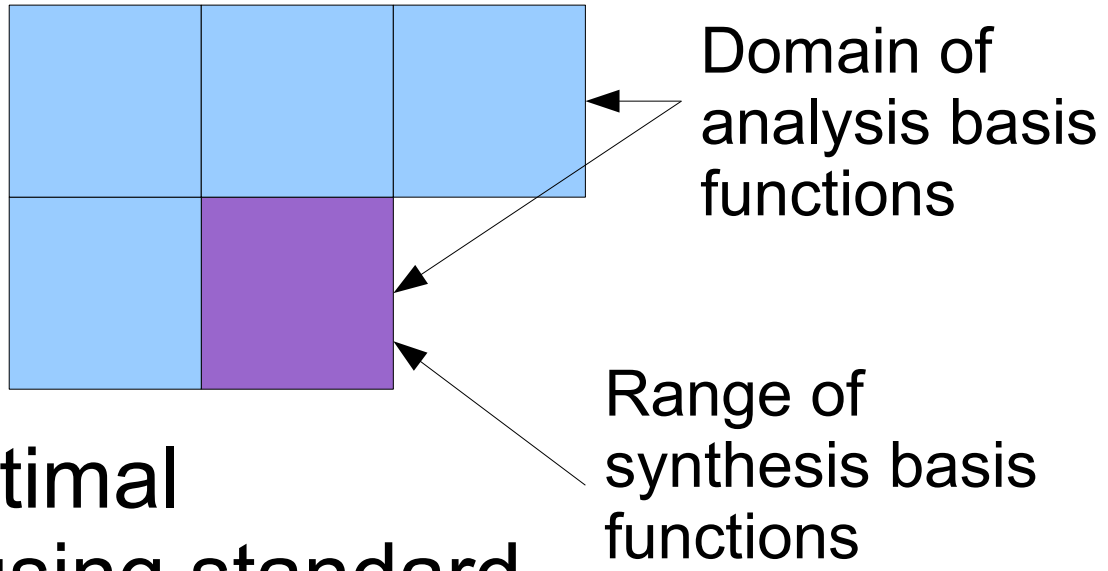


Fig. 4. Rate-distortion curves for the first frame of the full-HD sequence Riverbed

R. G. de Oliveira and B. Pesquet-Popescu: "Intra-Frame Prediction with Lapped Transforms for Image Coding." In Proc. of the 36th International Conference on Acoustics, Speech, and Signal Processing (ICASSP'11), pp. 805–808, May 2011.

Intra Prediction as Transform

- Generalize using idea of separate analysis & synthesis filters



- Can train optimal transforms using standard techniques (similar to KLT)
  - J. Xu, F. Wu, and W. Zhang: “Intra-Predictive Transforms for Block-Based Image Coding.” IEEE Transactions on Signal Processing 57(8):3030–3040, Aug. 2009.

Intra Pred. in Frequency Domain

- Optimal KLT-like transforms are not sparse
  - May not even be separable
    - E.g., 64×64 matrix multiply for 8×8 transform
- Idea: Use a standard transform, predict in frequency domain
  - Signals are sparse in frequency domain, so we should be able to enforce sparsity of predictor
- Works with lapped transforms without restrictions
  - Don’t have to worry about which pixels are available, etc.

Intra Prediction Limitations

- Intra prediction one of main innovations of H.264
- But how is it actually used?
  - Most common mode is “DC” mode, which has no orientation, and just uses one value for all pixels
  - Next is “horizontal” and “vertical”
    - These align well with the DCT basis functions, so you can fix things cheaply when it screws up
  - Diagonal modes only really useful on strong edges
    - Prediction *only* uses the edge of a block
    - Can’t extend texture, not needed for smooth regions

Intra Prediction Improvements

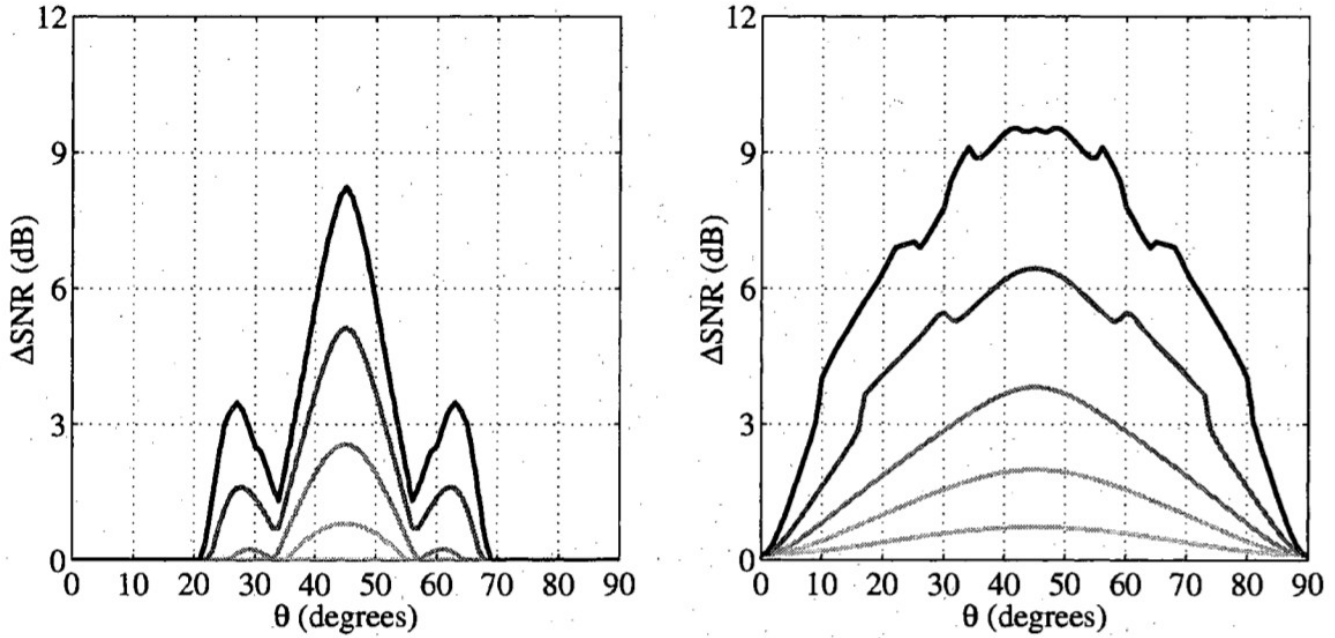
- VP8: “TM” mode: P = T + L – TL
  - Predicts gradients well, used over 50% of the time
- HEVC proposals:
  - *More* orientations (clustered around H and V)
  - Multi-stage:
    - Predict 1 out of 4 pixels
    - Decode residual for those pixels
    - Extend prediction into remaining pixels
    - Effectively restricts prediction to LF
  - Orientation-adaptive transforms for residual

Orientation-Adaptive Transforms

- Some images have many diagonally-oriented features
  - Including output of diagonal intra predictors
- Not compactly represented by DCT
  - Design custom transforms for each orientation
- Lots of research over past 6 or 7 years
  - Issues (some solved, some not): separability, data dependencies, fast implementations, coefficient ordering, etc.

Orientation Coding Gain

- Looks impressive (at least at 45°), but...



(a) DA-PBT over DCT (b) KLT over DCT

Figure 3.6: Transform coding gain improvement over the 8×8 DCT by (a) the 8×8 DA-PBT, and (b) the 8×8 KLT. The source is the elliptic random field with $\lambda_a = -\ln(0.8) f_s$, $\lambda_b = \lambda_a / k_b$ where $k_b = 2$, 4, 8, 16, and 32, and θ from 0° to 90°. A darker line corresponds to a larger k_b, i.e., stronger directionality.

Orientation Adaptive vs. Intra Pred. and Adaptive Block Size

Everything

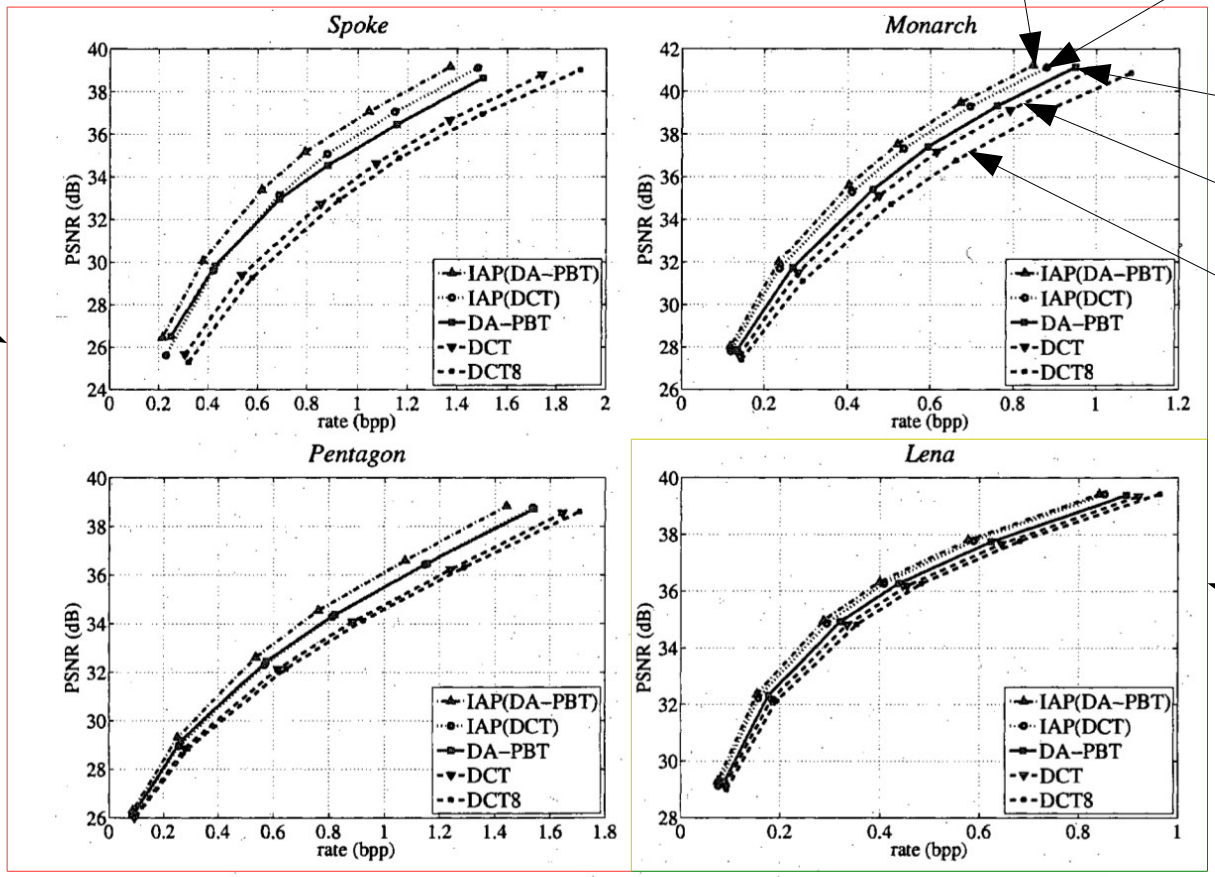
Intra prediction + Adaptive Block Size

Orientation Adaptive +Adaptive Block Size

Adaptive Block Size

Fixed Block Size

Highly-oriented images



Normal test image

Figure 3.10: Rate-distortion performance of variable blocksize transforms for image and residual image coding.

Additional References

- H.S. Malvar: “Extended Lapped Transforms: Properties, Applications, and Fast Algorithms.” IEEE Transactions on Acoustics, Speech, and Signal Processing, 40(11):2703–2714, Nov. 1992.
- T.D. Tran: “Lapped Transform via Time-Domain Pre- and Post-Filtering.” IEEE Transactions on Signal Processing 51(6):1557–1571, Jun. 2003.
- W. Dai and T.D. Tran: “Regularity-Constrained Pre- and Post-Filtering for Block DCT-based Systems.” IEEE Transactions on Signal Processing 51(10):2568–2581, Oct. 2003.
- J. Hu and J.D. Gibson: “New Rate Distortion Bounds for Natural Videos Based on a Texture Dependent Correlation Model.” In Proc. 46th Allerton Conference on Communication, Control, and Computing, pp. 996–1003, Sep. 2008.
- J. Han, A. Saxena, and V. Melkote: “Jointly Optimized Spatial Prediction and Block Transform for Video and Image Coding.” IEEE Transactions on Image Processing (pre-print), Sep. 2011.
- C.-L. Chang: “Direction-Adaptive Transforms for Image Compression.” Ph.D. Thesis, Stanford University, Jun. 2009.

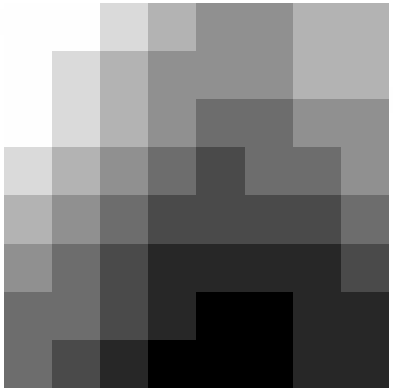
Questions?

Introduction to Video Coding
Part 2: Entropy Coding

Mozilla

Review

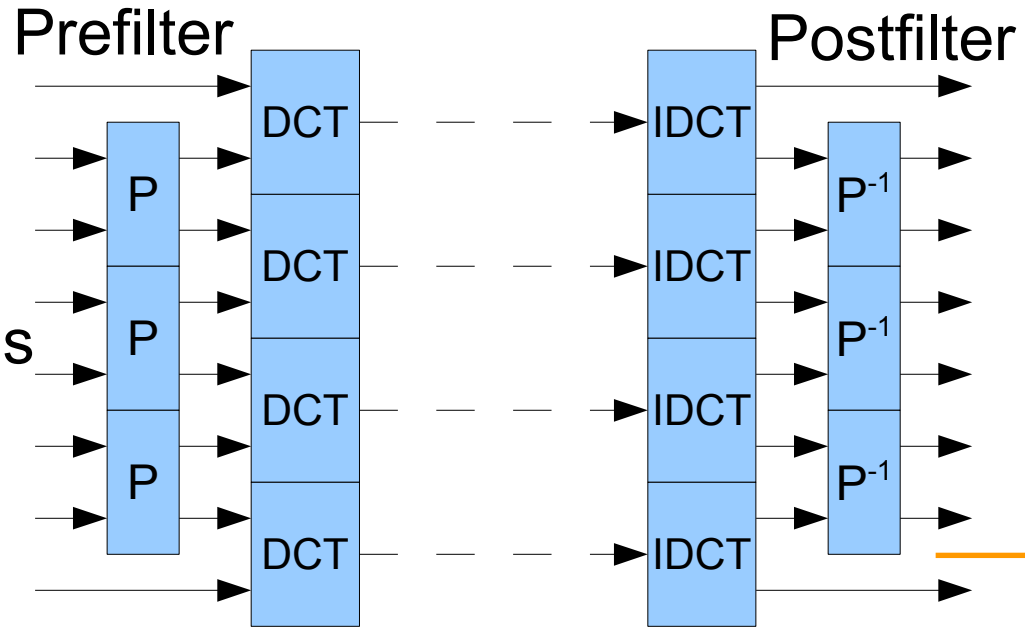
DCT

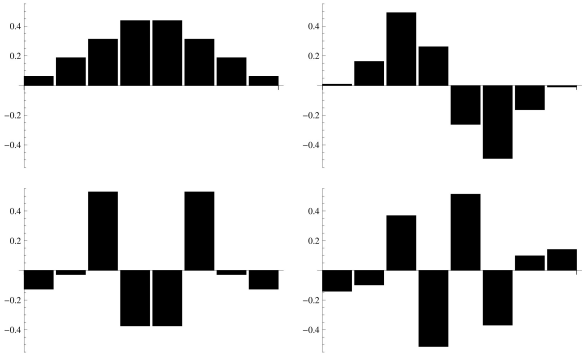


Input Data							
156	144	125	109	102	106	114	121
151	138	120	104	97	100	109	116
141	129	110	94	87	91	99	106
128	116	97	82	75	78	86	93
114	102	84	68	61	64	73	80
102	89	71	55	48	51	60	67
92	80	61	45	38	42	50	57
86	74	56	40	33	36	45	52

Transformed Data							
700	100	100	0	0	0	0	0
200	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

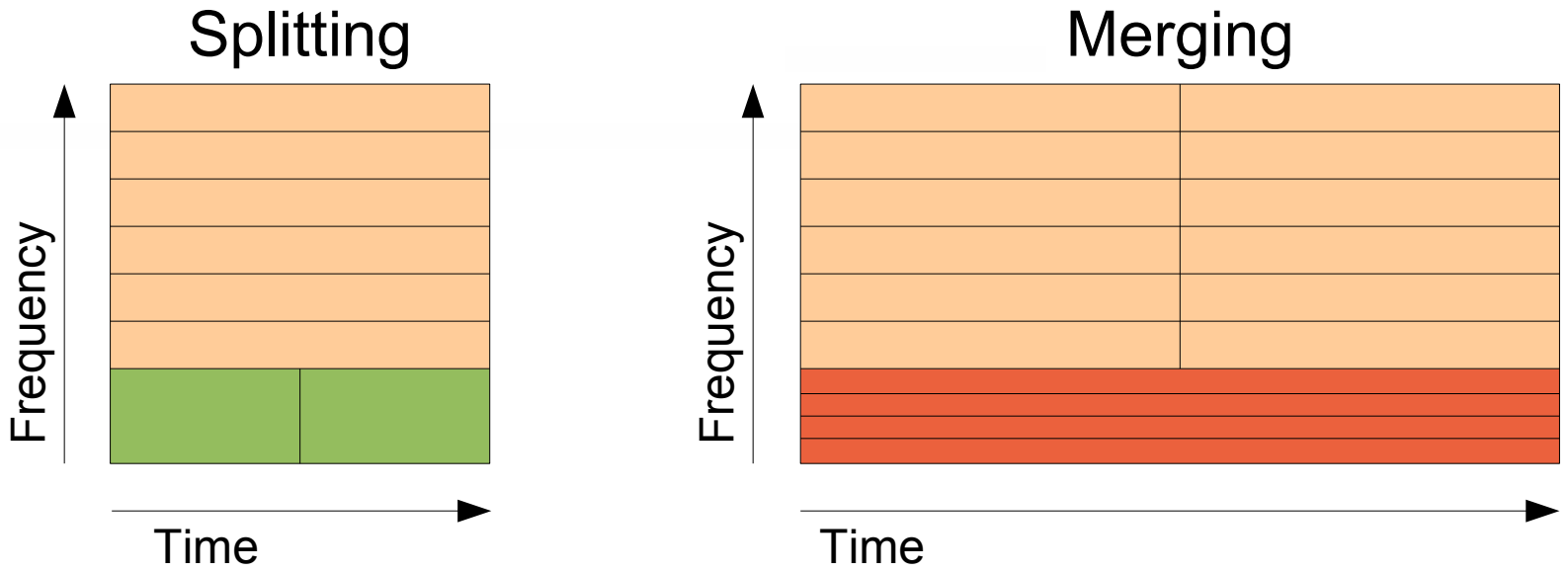
Lapped Transforms



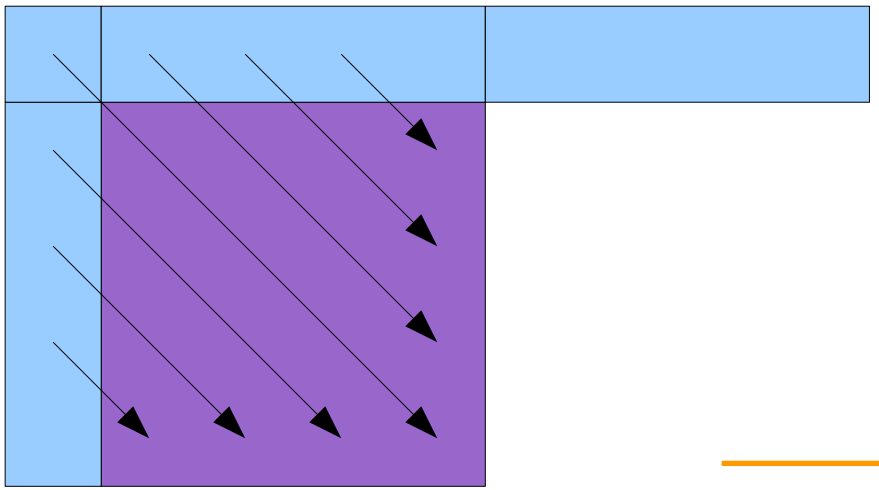


Review (cotd.)

TF Resolution Switching



Intra Prediction



Shannon Entropy

- Minimum number of bits needed to encode a message given the probability of its symbols

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2(p(x_i))$$

- Represents the limit of *lossless* compression

C. E. Shannon: “A Mathematical Theory of Communication.” *The Bell System Technical Journal*, 27(3-4): 379–423, 623–656, Jul., Oct. 1948.

Two Questions

- How do we know $p(x_i)$?
  - More on this later
- How close to this rate can we actually compress data?
  - We'll tackle this first
  - Two approaches worth discussing
    - Huffman coding (fast)
    - Arithmetic coding (better compression)

Huffman Codes

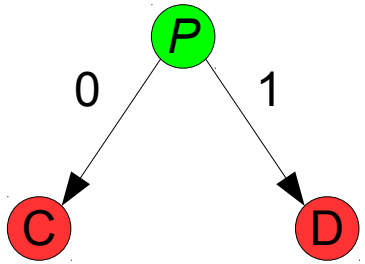
- Also known as
  - Variable-Length Codes
  - Prefix-free Codes
- Basic idea
  - Vary the number of bits per symbol
  - Use fewer bits for more frequent symbols
  - Use more bits for rare symbols

D. A. Huffman: “A Method for the Construction of Minimum-Redundancy Codes.” *Proceedings of the Institute for Radio Engineers*, 40(9): 1098–1101, Sep. 1952.

Huffman Code Construction

- Take the two least probable symbols and put them in a binary tree with “0” and “1” branches
- Set the probability of the parent equal to the sum of its childrens’, and put it back in the list
- Repeat until there’s only one item left in the list...

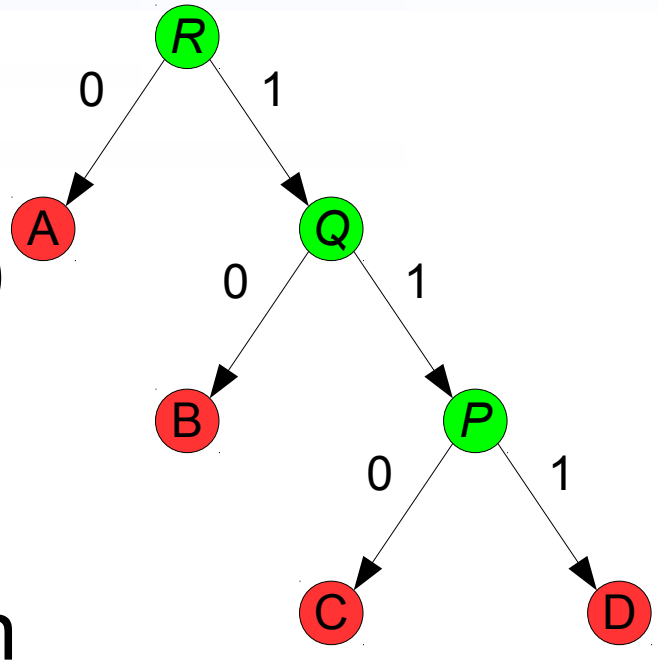
Symbol	Frequency	Probability
A	8	2/3
B	2	1/6
C	1	1/12
D	1	1/12



Symbol	Probability
A	2/3
B	1/6
P	1/6

Huffman Code Construction

- Follow the path from the root to get each leaf's codeword
- Bits per symbol: $\sum_{i=1}^{n} p(x_i)\,\text{length}(x_i)$
  - 1.5 bits (25% compression)
  - Entropy: 1.42 bits/symbol
- Huffman models the distribution
  - *As if* probs are (1/2, 1/4, 1/8, 1/8)
  - Instead of (2/3, 1/6, 1/12, 1/12)
- Called "cross-entropy"
  - Gap called Kullback-Leibler Divergence



Symbol	Codeword
A	0
B	10
C	110
D	111

Huffman Code Problems

- Only optimal when probabilities are powers of 2
  - Can't represent probabilities greater than 1/2!
  - Code multiple symbols at once to approach bound
    - But size of code grows exponentially
- Not adaptive when distribution not known
  - Code construction is slow: $O(N \log N)$
  - Need to either pick a fixed code (suboptimal) or transmit the code to use (overhead)
  - Adaptive versions exist, but still slow and complex

Huffman Coding in Video

- Some form of VLC used in every standard from H.261 (1988) to H.264-MVC (2009).
- Video is not text compression
  - There are many different kinds of symbols that need to be encoded
    - Macroblock modes, motion vectors, DCT coefficient values, signs, run lengths, etc.
  - The size of the alphabet can differ for each one
  - They have vastly different probability distributions
  - Need to change which code we're using on a symbol-by-symbol basis

Huffman Coding in Video

- Video is not text compression (cotd.)
  - We know the resolution of the video
  - Hence we know exactly when to stop reading more symbols
  - No need to reserve special values to mark the end of the stream
  - By convention, we can use zeros if we run out of input bits
    - Can strip trailing zeros from the output

Huffman Coding in Video

- Theora
  - Codes for block flags, MB modes, MVs fixed
  - Codes for DCT coefficients (the bulk of the data) transmitted in a stream header
    - Different codes used for various coefficients
    - Can pick which set to use on a frame-by-frame basis
- H.264: CAVLC (“Context Adaptive”)
  - Fixed set of codes to choose from
  - Choice is made based on previously decoded symbols (the “context”)

Huffman Decoding

- Read bits one at time, traverse tree (slow)
- Finite State Machine
  - Current state is the index of an interior node of the Huffman tree
  - Read *n* bits, use LUTs to get new state and a list of zero or more decoded symbols
  - Can choose *n* independent of code lengths
  - Fastest method, but need to know which code subsequent symbols will use
    - This decision may be very complex

Huffman Decoding: libtheora

- Traverse multiple levels of the tree with a LUT
  - Peek ahead *n* bits
  - Use a LUT to find
    - The node in the tree (up to *n* levels down)
    - How many bits to consume
  - Example: 2-bit table → 5/6 of symbols need only one lookup
    - Usually one well-predicted branch per symbol
  - *n* trades off table size (cache) and branches
    - Use a larger value for the root node

Bits	Node	Depth
00	A	1
01	A	1
10	B	2
11	*P*	2

Bits	Node	Depth
0	C	1
1	D	1

Arithmetic Coding

- Inefficiency of Huffman codes comes from using a whole number of bits per codeword
- Arithmetic coding doesn't partition the code space on bit boundaries
  - Can use fractional bits per codeword
- Gets very close to the Shannon bound
  - Limited only by the precision of the arithmetic

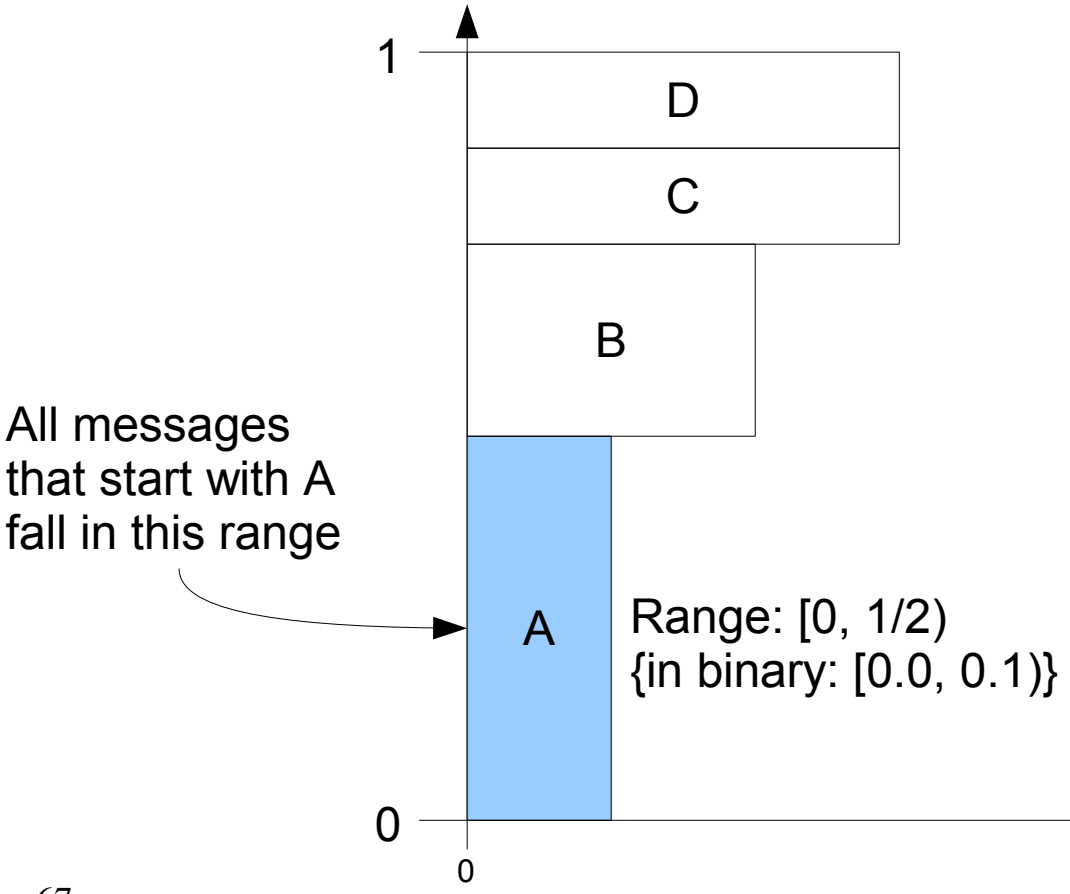
Huffman Coding Revisited

- Imagine bitstream as a binary number in [0, 1)
- Code the message “ABC”

Range: [0,1)

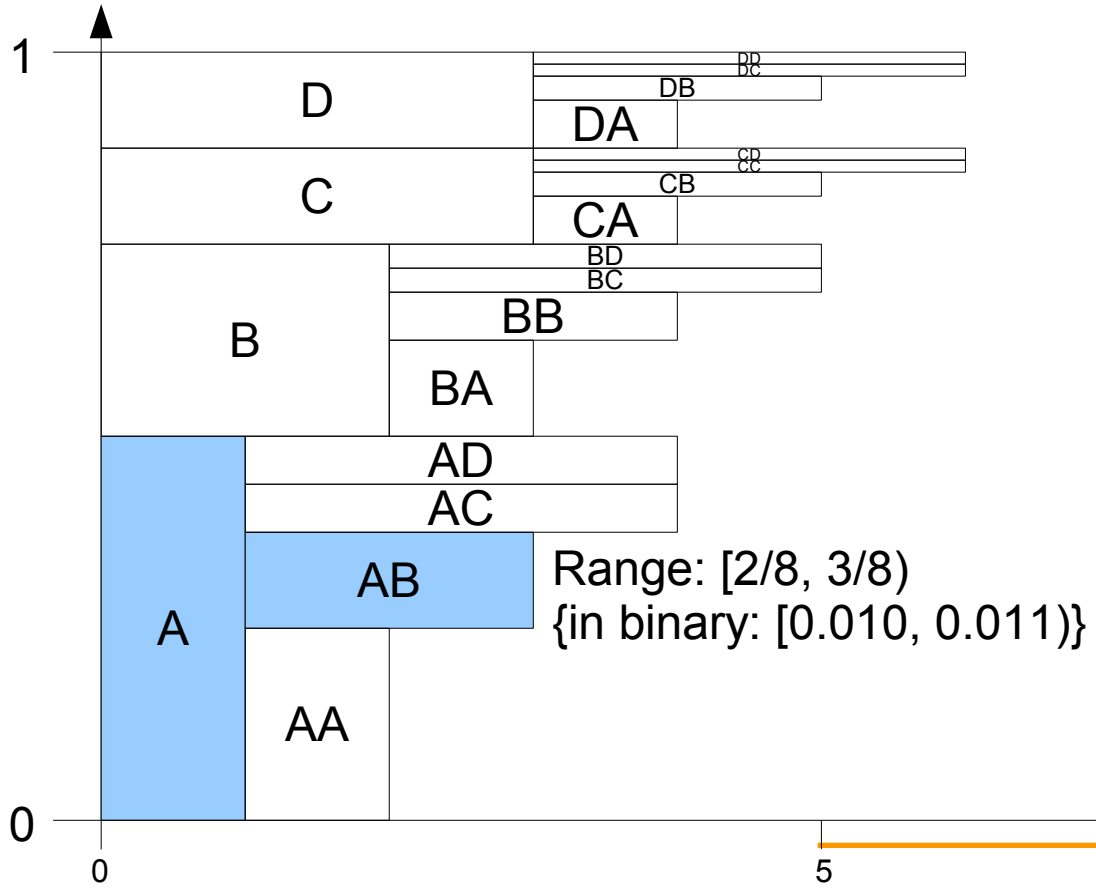
Huffman Coding Revisited

- Imagine bitstream as a binary number in [0, 1)
- Code the message “ABC”



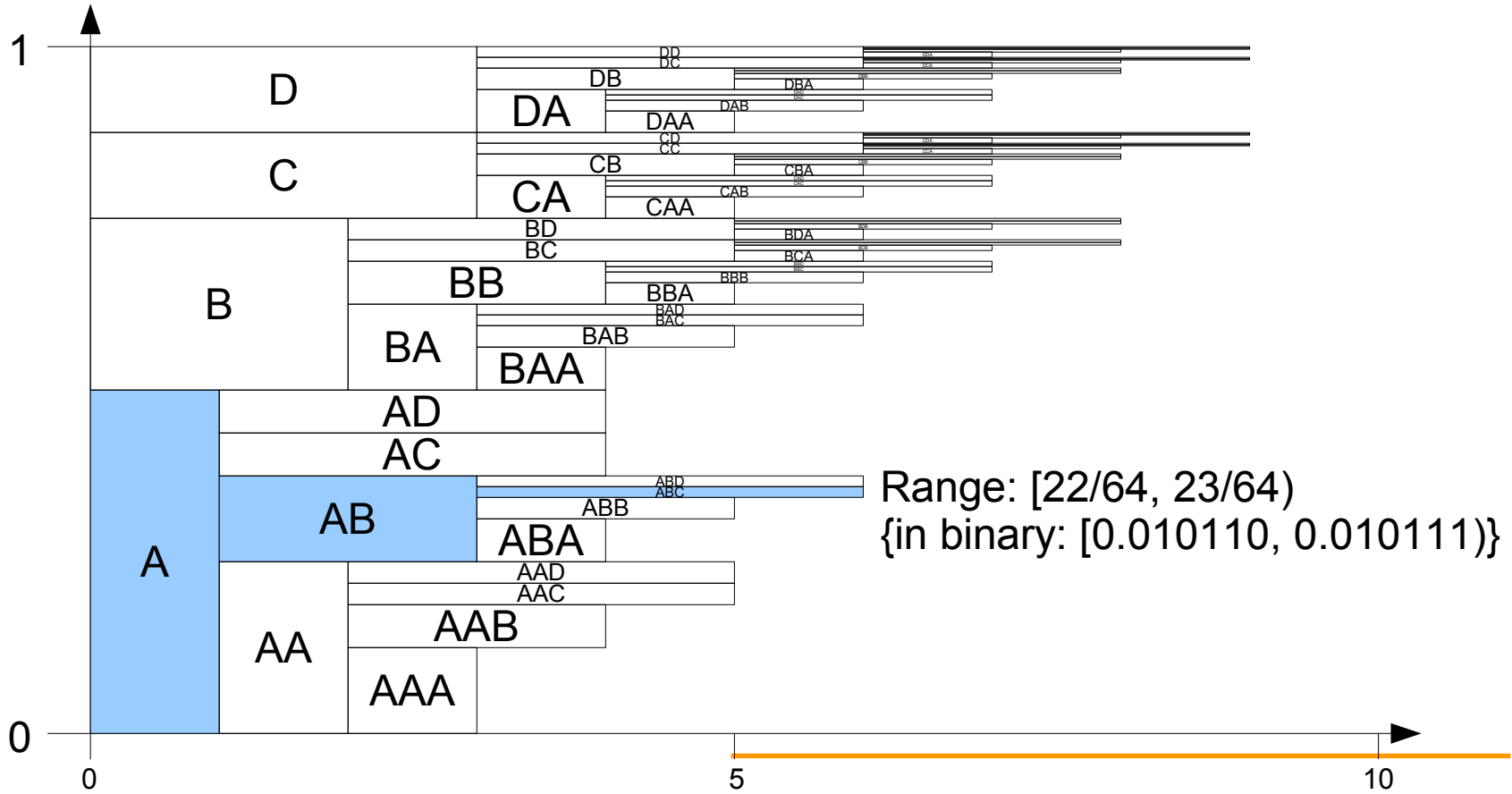
Huffman Coding Revisited

- Imagine bitstream as a binary number in [0, 1)
- Code the message “ABC”



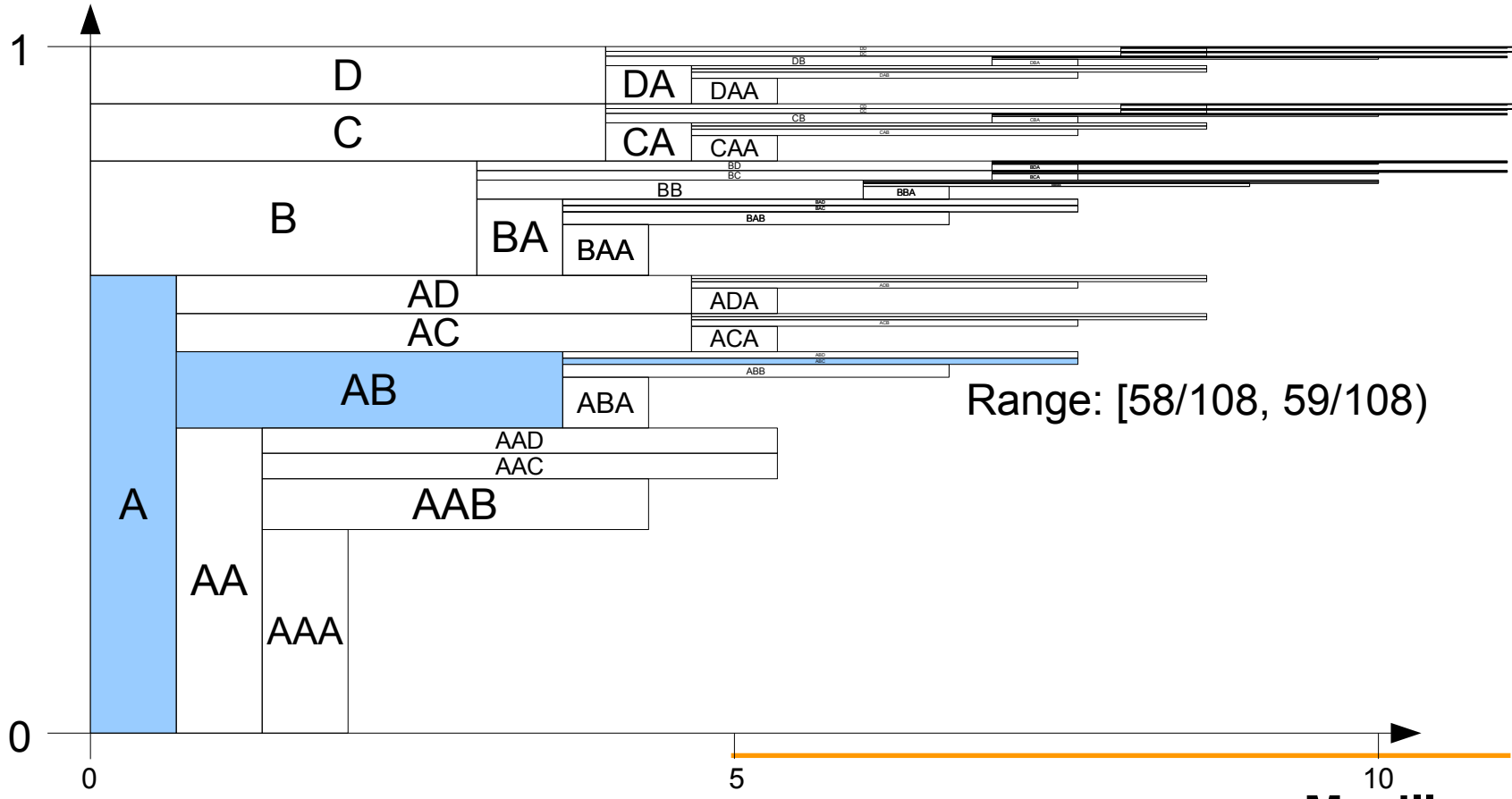
Huffman Coding Revisited

- Encode message with shortest number in range
- Number of bits ≤ $\lceil -\log_2(\text{Range}) \rceil = -\log_2(1/64) = 6$



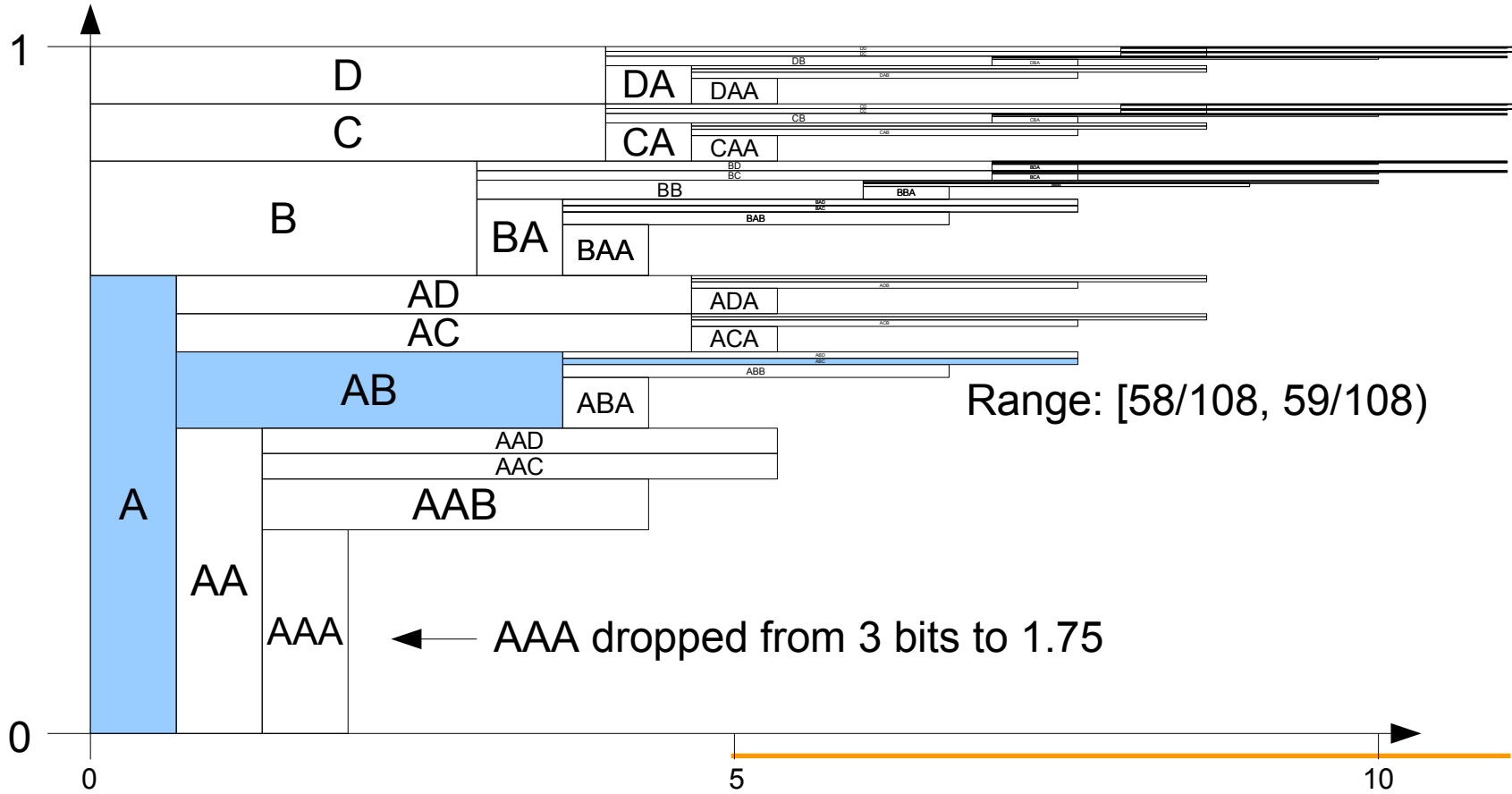
Arithmetic Coding

- No need for intervals to be powers of two
- Number of bits ≤ $\lceil -\log_2(1/108) \rceil = \lceil 6.75 \rceil = 7$



Arithmetic Coding

- No need for intervals to be powers of two
- Number of bits ≤ $\lceil -\log_2(1/108) \rceil = \lceil 6.75 \rceil = 7$



Mozilla

Arithmetic Coding

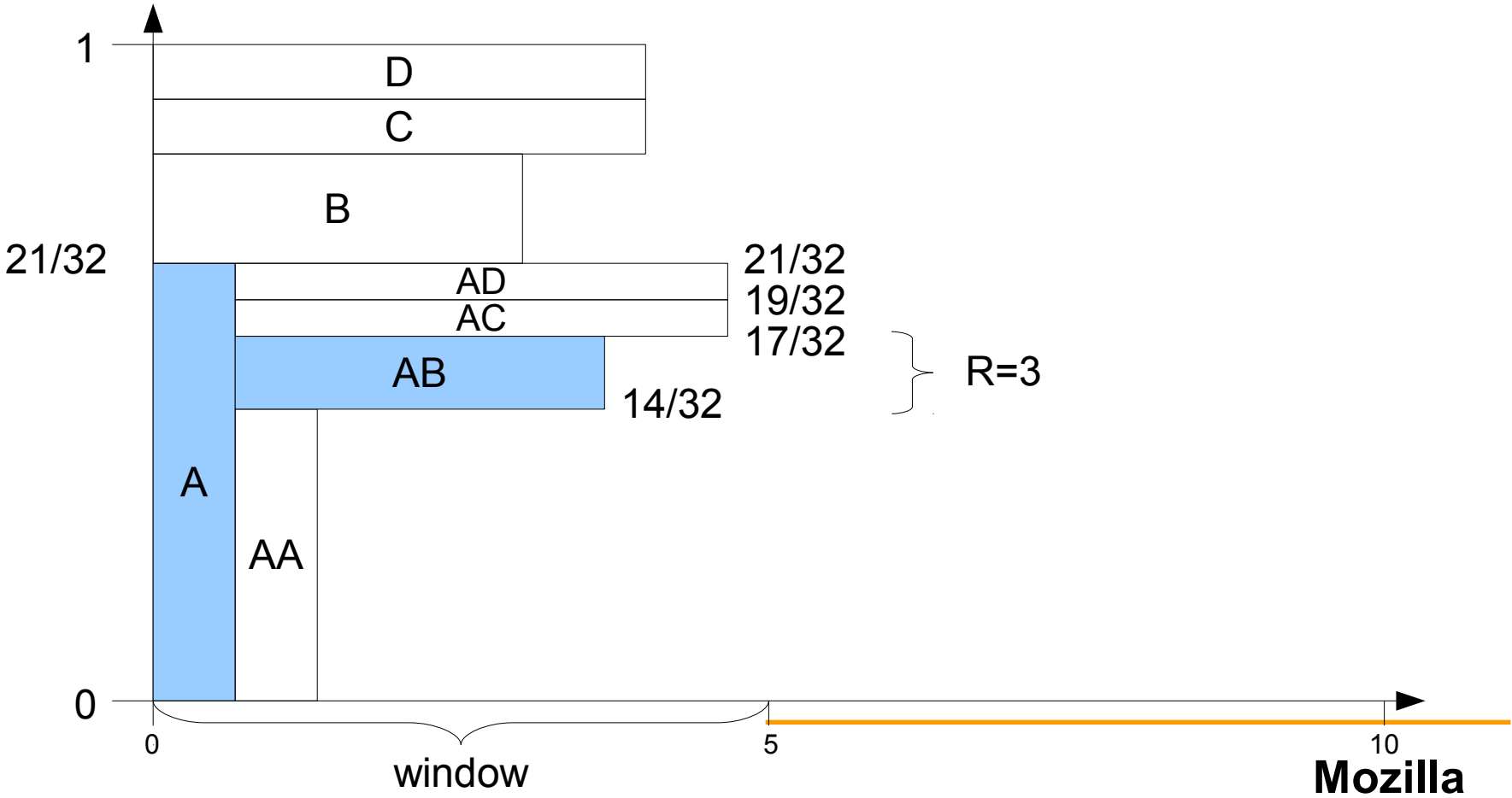
- Can represent entire message with shortest number (most trailing zeros) that lies in range
- Infinite precision arithmetic called “Elias Coding”
  - Gets within 1 bit of Shannon bound
- Real arithmetic coding uses finite-precision
  - Use a “sliding window” that gets rescaled
    - L = lower bound of range, R = size of range

5-bit Window (Encoding)

- Example: L = 0, R = 21, want to code B

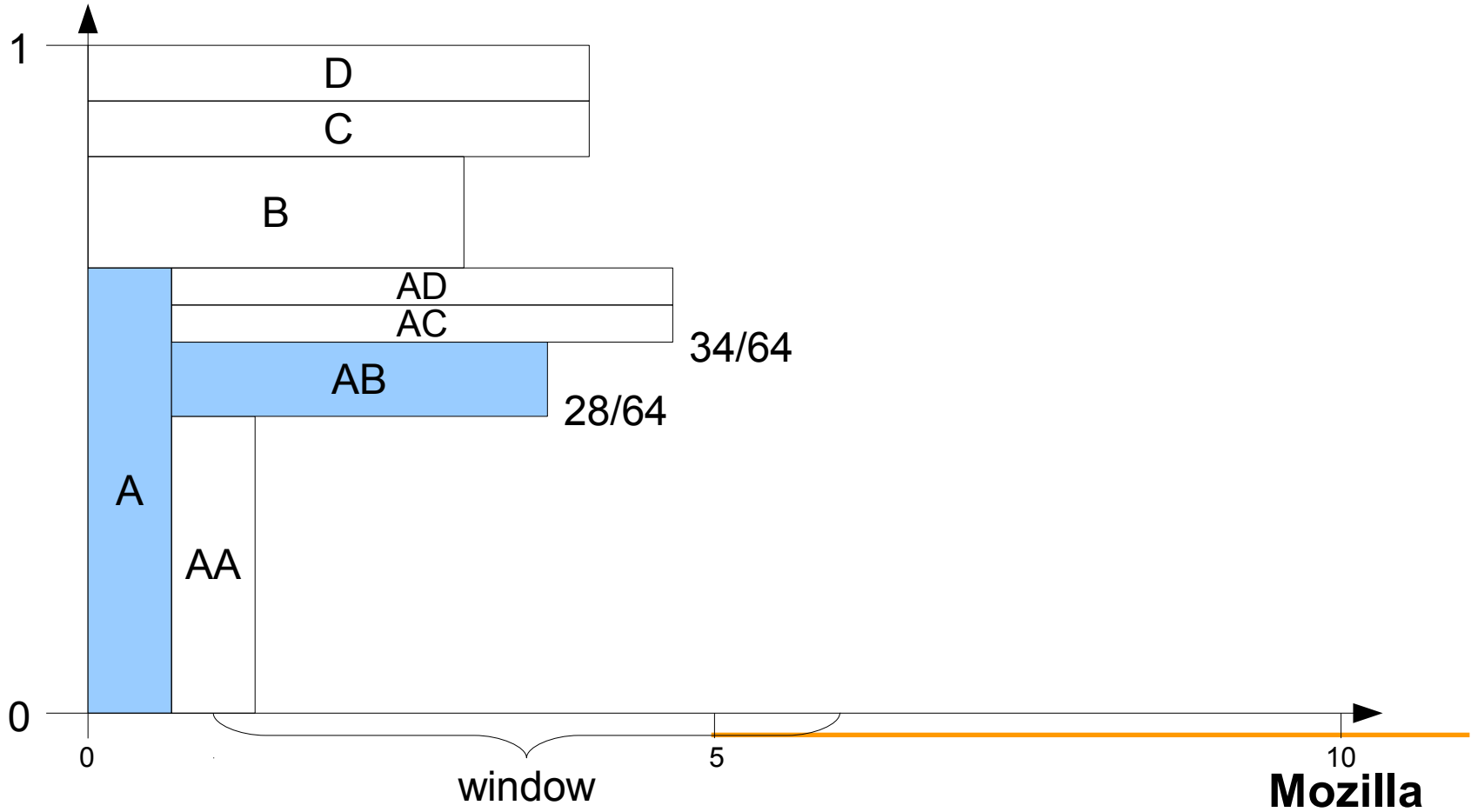
5-bit Window (Encoding)

- Partition R proportional to probabilities
- Update: L=14, R=3



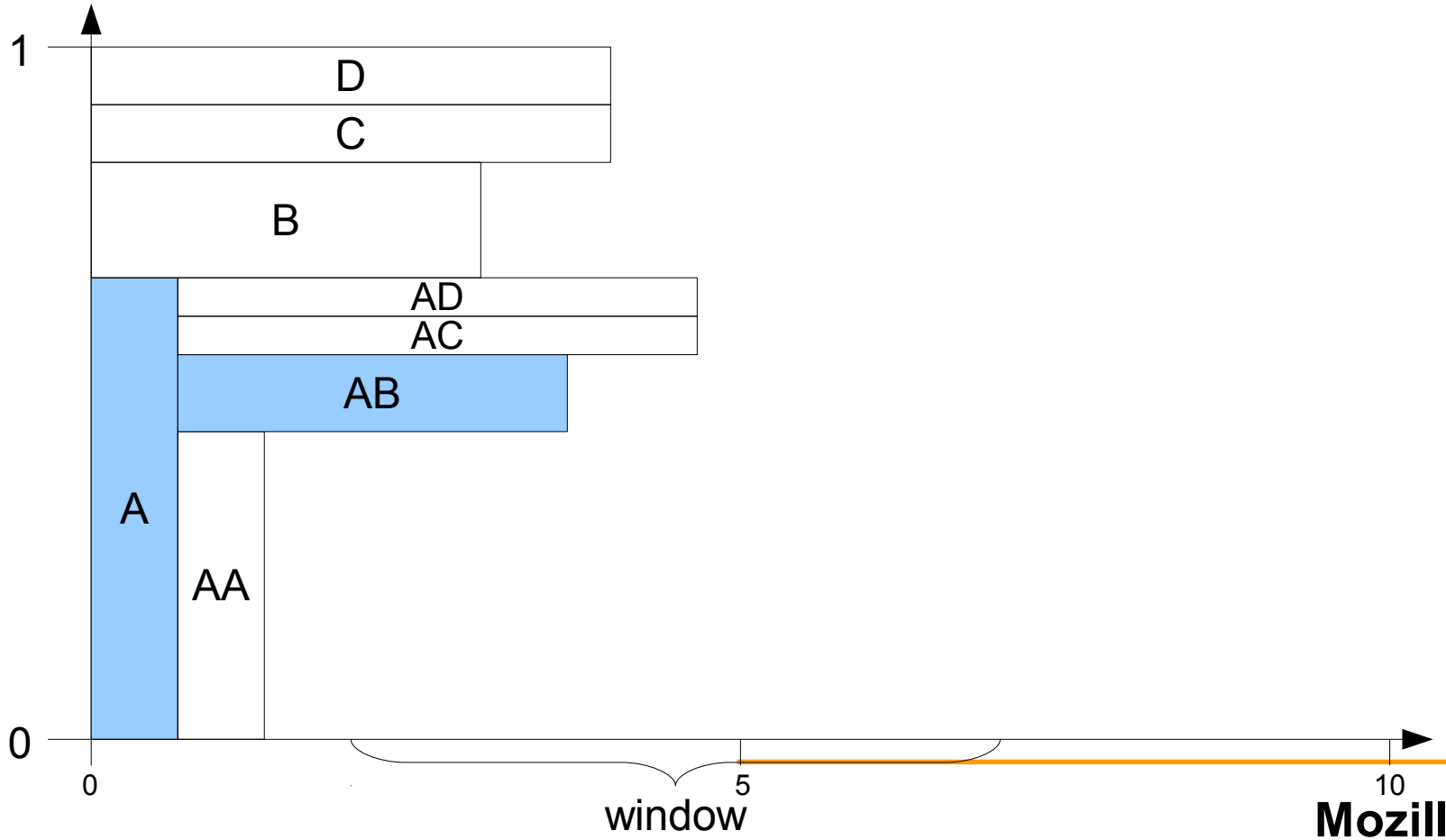
5-bit Window (Encoding)

- Double L and R until R ≥ 16
- Renormalize: output: 0 ← L=28, R=6



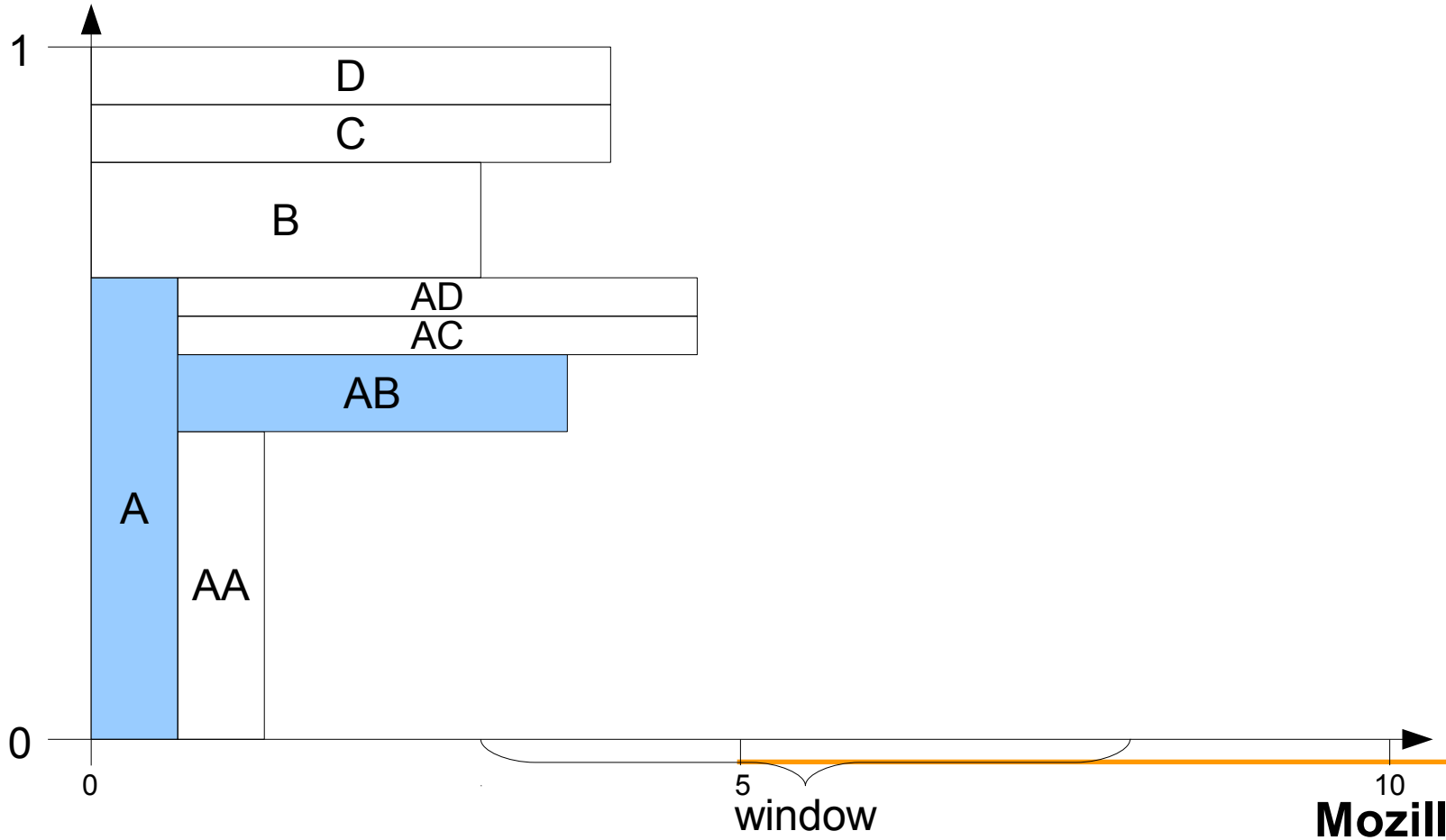
5-bit Window (Encoding)

- Double L and R until R ≥ 16
- Renormalize: output: 01 ← L=24, R=12



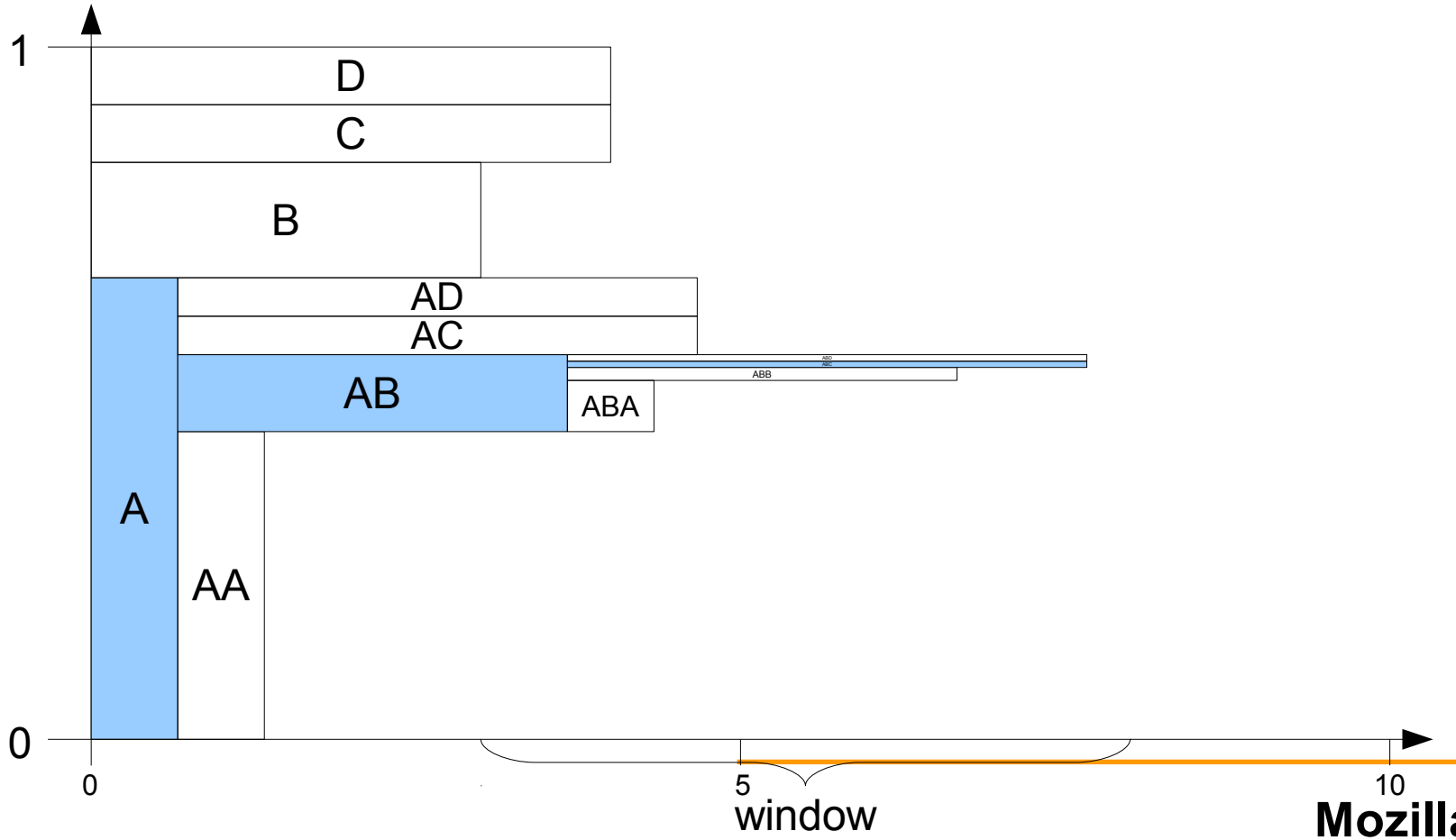
5-bit Window (Encoding)

- Double L and R until R ≥ 16
- Renormalize: output: 011 ← L=16, R=24



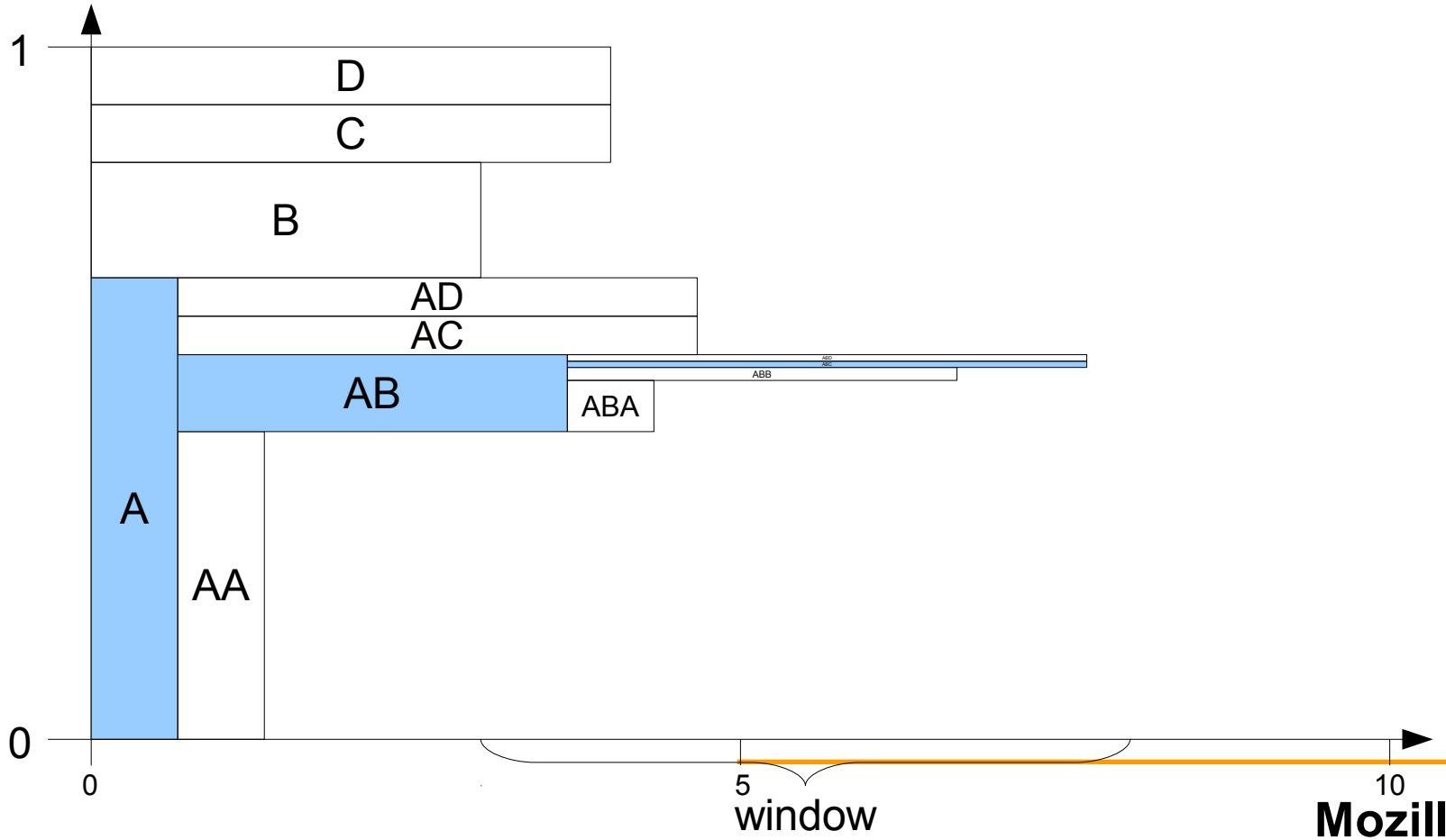
5-bit Window (Encoding)

- Carry propagation: L can exceed 32
- Update: output: 011, L=36, R=2



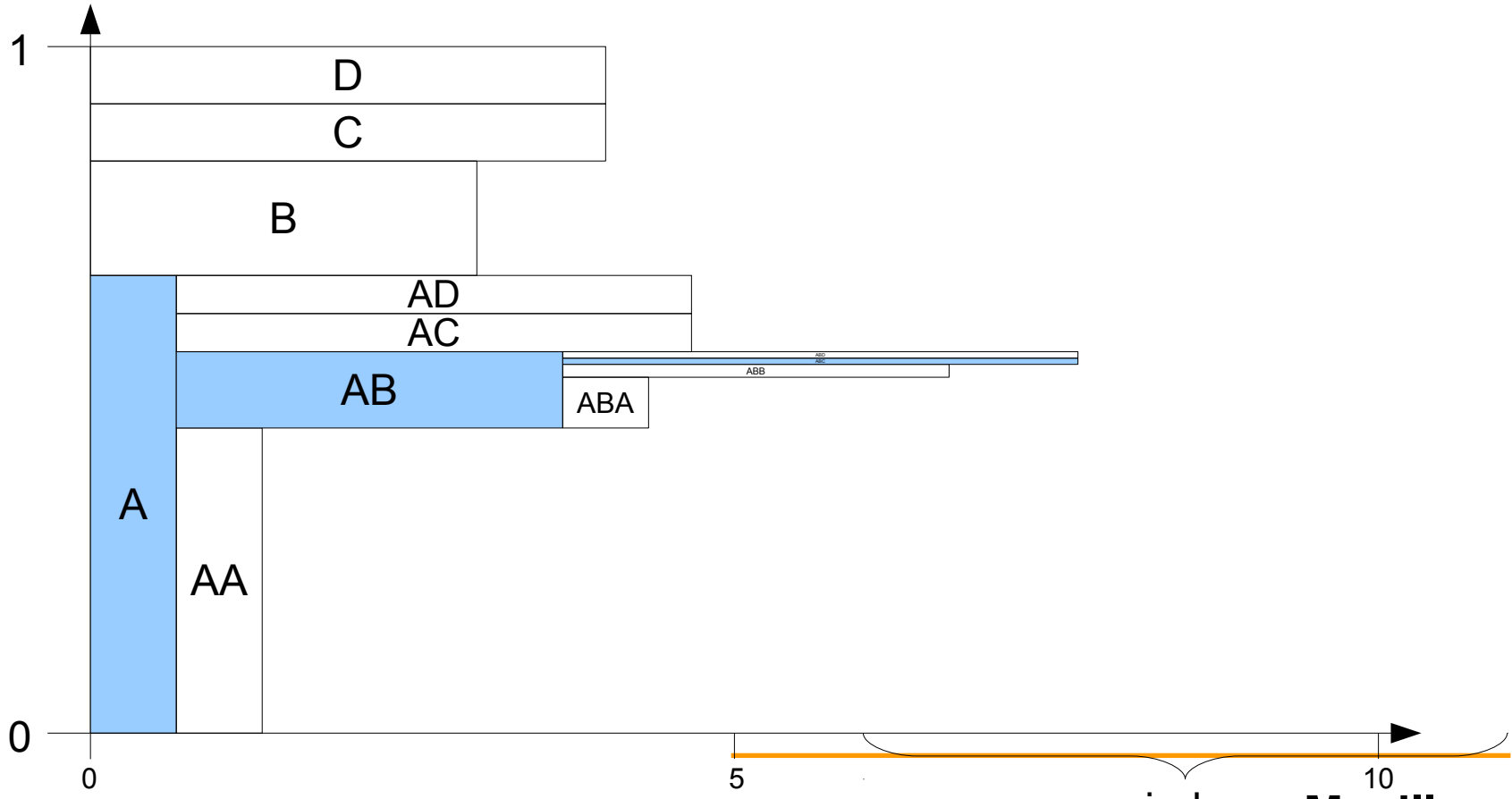
5-bit Window (Encoding)

- Carry propagation: L can exceed 32
- Update: output: 100, L=4, R=2



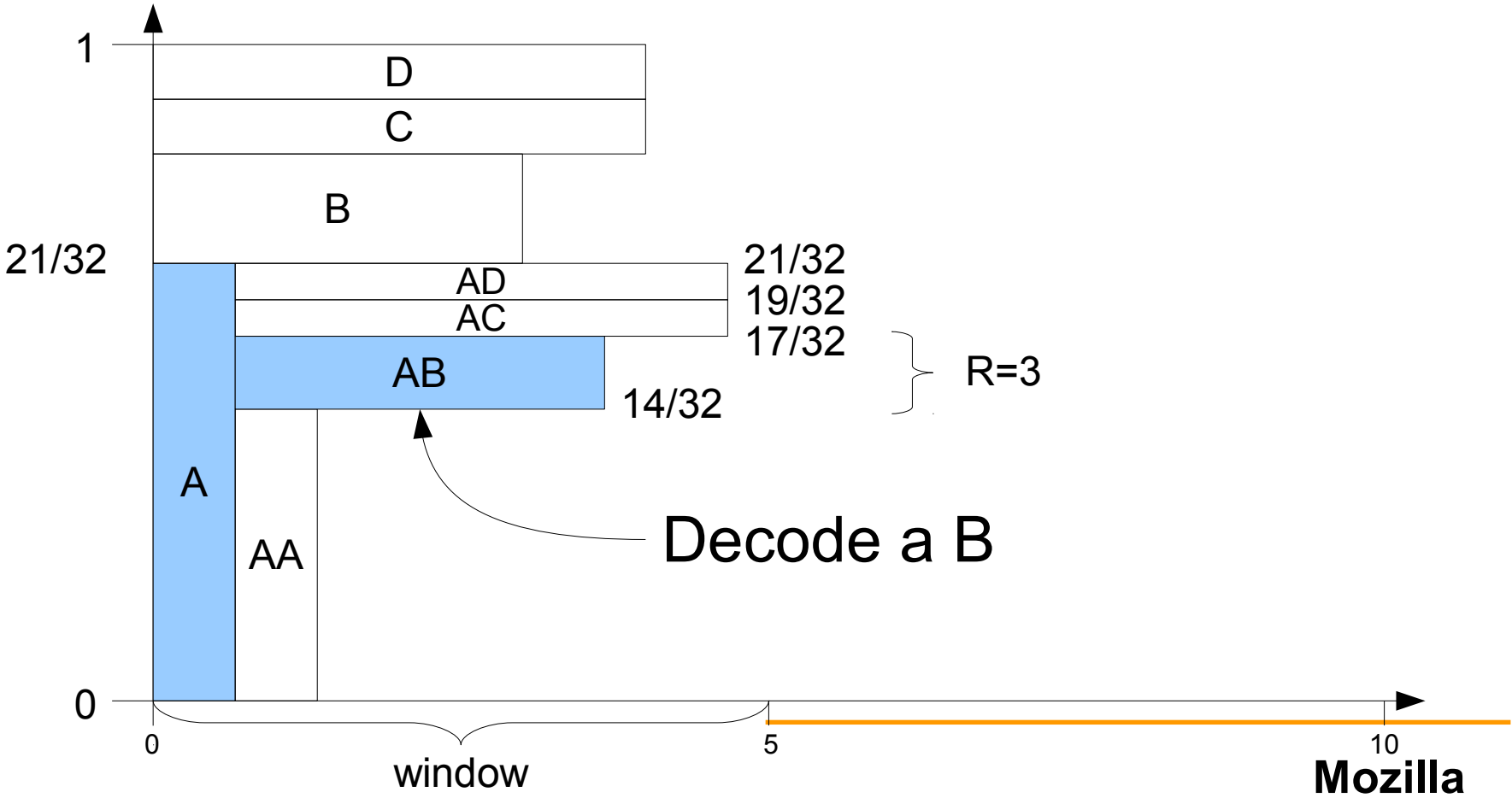
5-bit Window (Encoding)

- Then renormalize like normal
- Renormalize: output: 100001 ← L=0, R=16



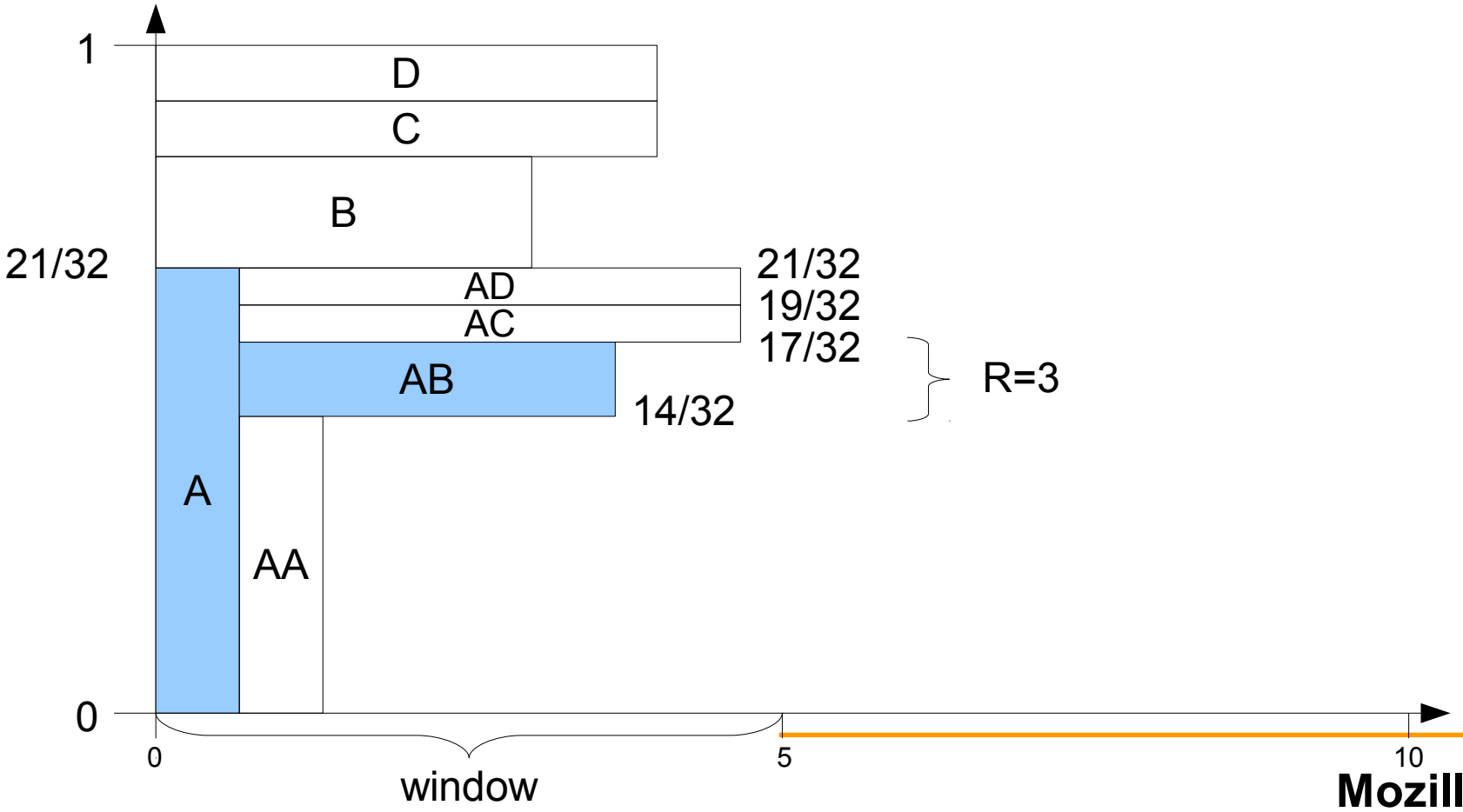
5-bit Window (Decoding)

- Decoding: Read bits into C, find partition it's in
- L=0, R=21, C=16 ← input: 100001



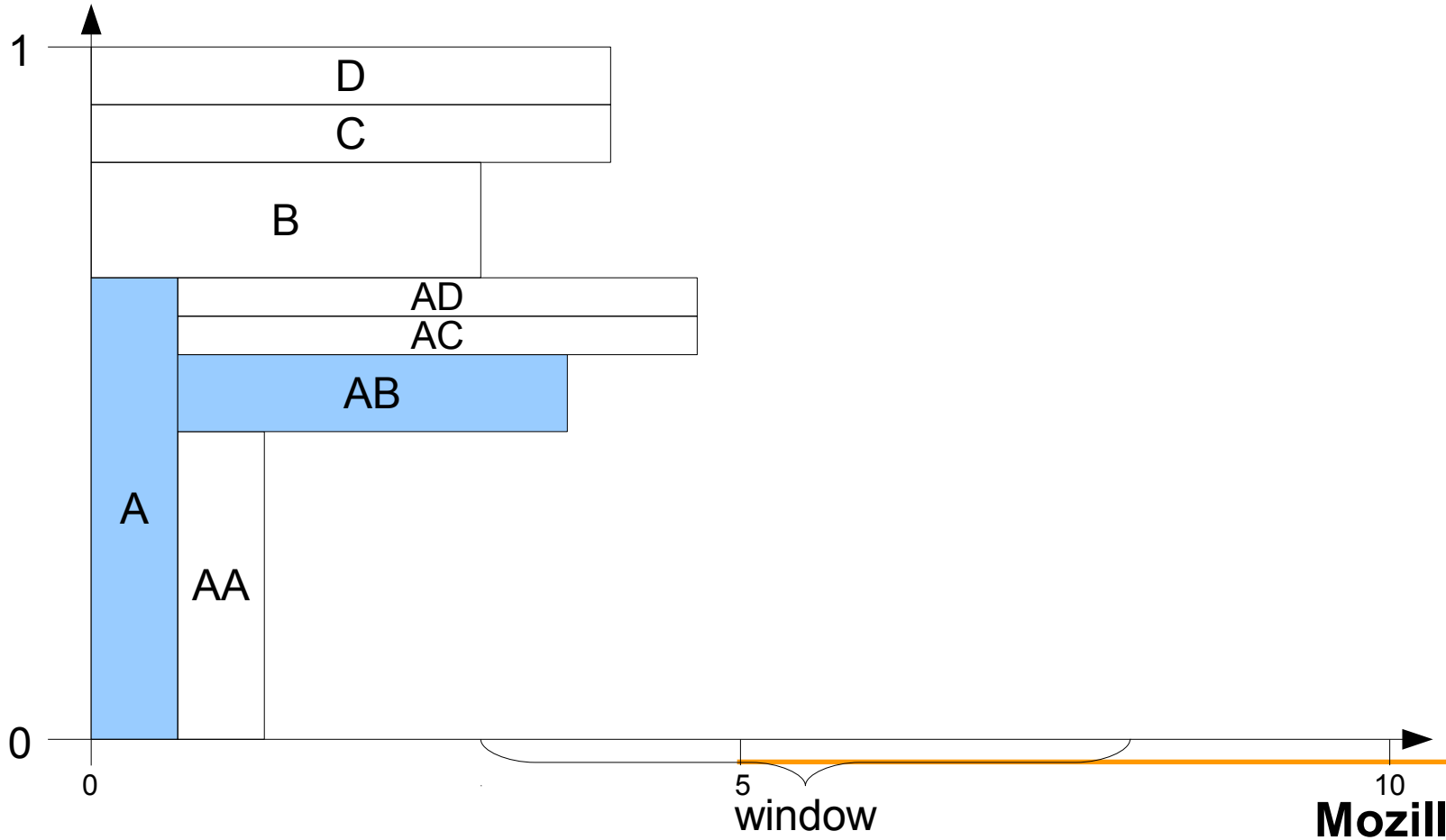
5-bit Window (Decoding)

- Update: Same calculations as encoder
- L=14, R=3, C=16, input: 100001



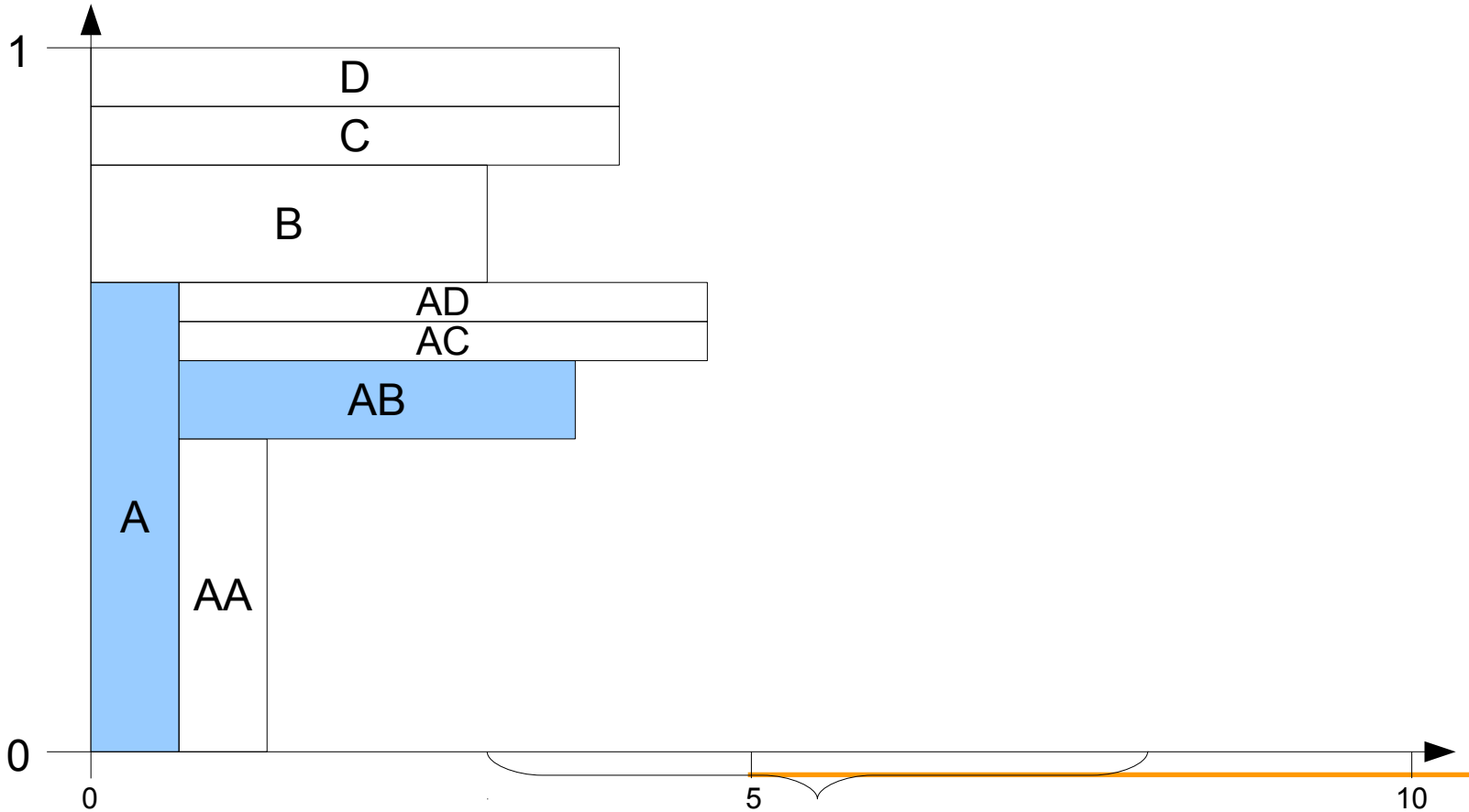
5-bit Window (Decoding)

- Renormalize: Shift more bits into C
- L=16, R=24, C=4 ← input: 10000100



5-bit Window (Decoding)

- If C isn't in [L, L+R), borrow (inverse of carry)
- L=16, R=24, C=36 ← input: 10000100



Arithmetic Coding: That's it!

- Variations
  - Renormalize 8 bits at a time ("Range Coding")
    - Byte-wise processing is faster in software
  - Binary: Less arithmetic, no search in decoder
    - Lots of optimizations and approximations only work with binary alphabets
    - This is what all current video codecs use
- Partition functions...

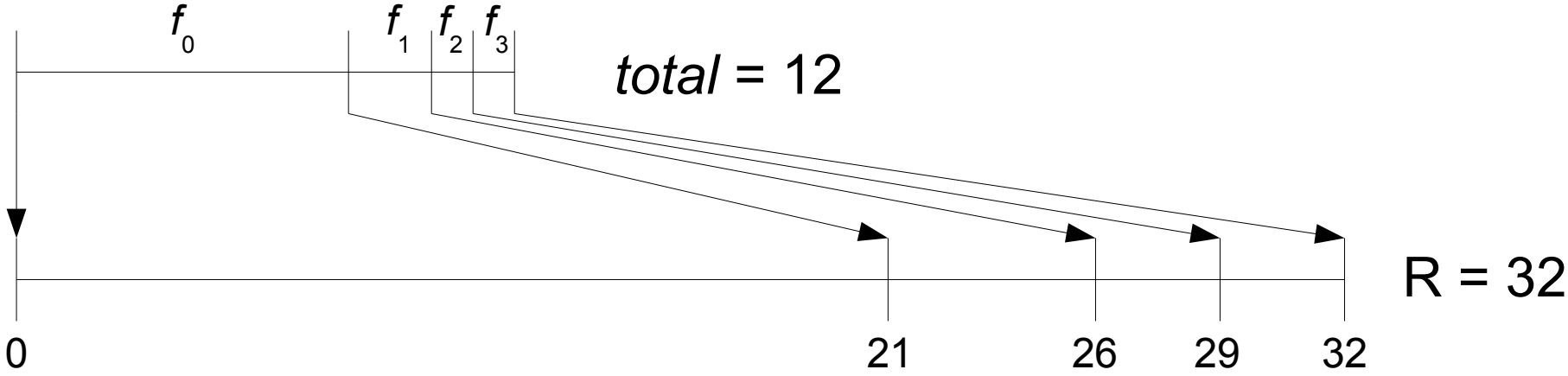
Partition Functions

- Splits R according to *cumulative* frequency counts:

$$f_i = \text{frequency of } i \text{ th symbol}$$

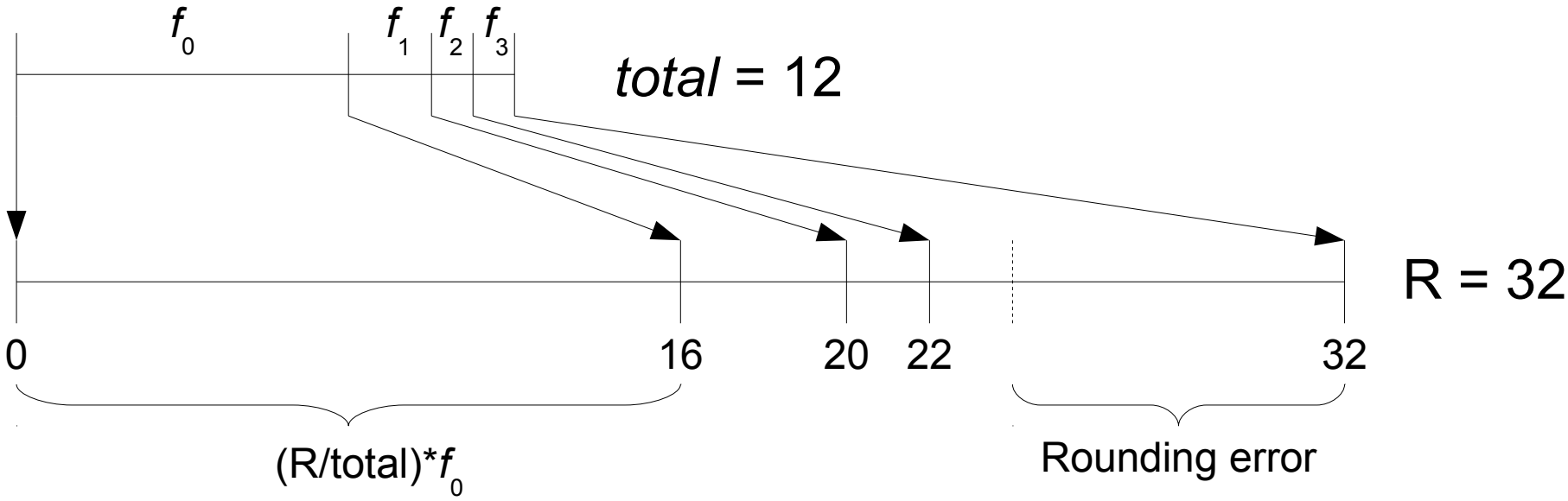
$$c_i = \sum_{k=0}^{i-1} f_k \qquad total = c_N$$

- $R \rightarrow R * c_i / total$ ("CACM" coder, Witten et al. 1987)



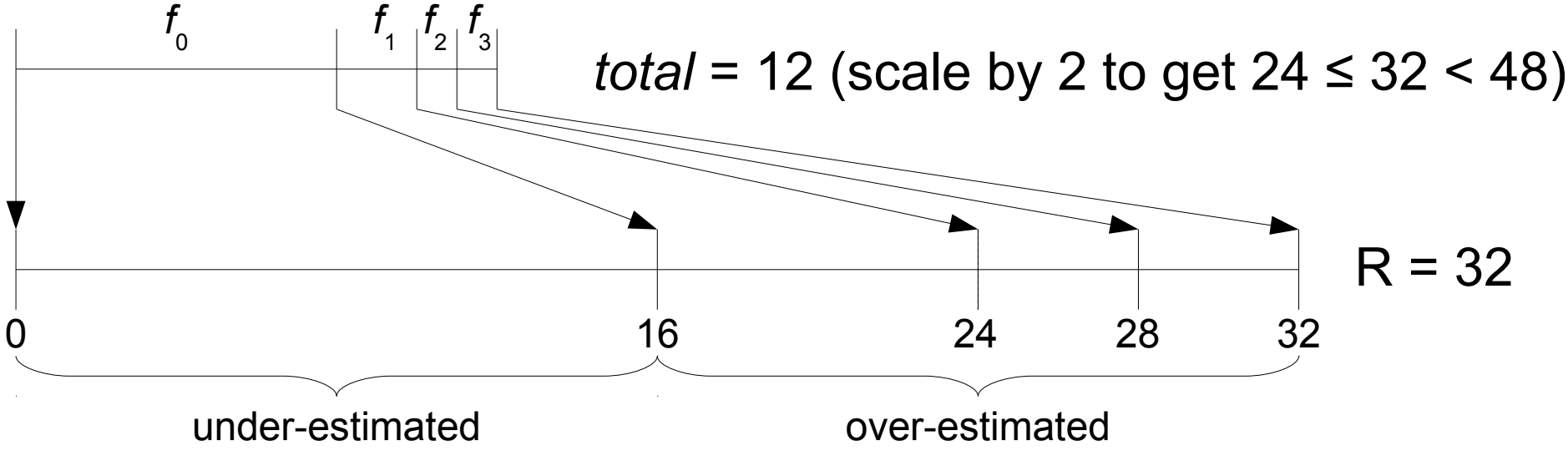
Partition Functions (cotd.)

- $R \rightarrow (R / total) * c$ (Moffat et al. 1998)
  - Better accuracy for fixed register size, one less div
  - Over-estimates the probability of the last symbol
    - The error is small as long as *total* << R (less than 1% given 6 bits of headroom)



Partition Functions (cotd.)

- $\mathrm{R} \rightarrow \max(c, 2*(c-total)+\mathrm{R})$
  - Stuiver and Moffat 1998
  - Requires *total* ≤ R < 2**total* (shift up if not)
  - Distributes rounding error more evenly
    - But always has a significant amount



Partition Functions (cotd.)

- Table-driven (e.g., CABAC)
  - No multiplies or divides
  - Binary alphabets only
  - Small set of allowed c_i's
  - R restricted to 256...511, only bits 6 and 7 used

$$\mathrm{R} \rightarrow \begin{cases} 0, & c_i = 0 \\ \mathrm{rangeTabLPS}\left[\mathrm{indexOf}\left(c_i\right)\right]\left[\left\lfloor \mathrm{R}/64 \right\rfloor mod\ 4\right], & 0 < c_i < total \\ R & c_i = total \end{cases}$$

- Truncation error can be as large as 25%
  - All added to the MPS, can waste 0.32 bits for LPS

Arithmetic Coding References

- J.J. Rissanen: “Generalized Kraft Inequality and Arithmetic Coding.” *IBM Journal of Research and Development,* 20(3): 198–203, May 1976.
- I.H. Witten, R.M. Neal, and J.G. Cleary: “Arithmetic Coding for Data Compression.” *Communications of the ACM,* 30(6): 520–540, Jun. 1987.
- A. Moffat, R.M. Neal, and I.H. Witten: “Arithmetic Coding Revisited.” *ACM Transactions on Information Systems*, 16(3): 256–294, Jul. 1998.
- L. Stuiver and A. Moffat: “Piecewise Integer Mapping for Arithmetic Coding.” In Proc. *8th Data Compression Conference (DCC ‘98)*, pp. 3–12, Mar. 1998.
- D. Marpe, H. Schwarz, and T. Wiegand: “Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard.” *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7): 620–636, Jul. 2003.
- http://people.xiph.org/~tterribe/notes/range.html

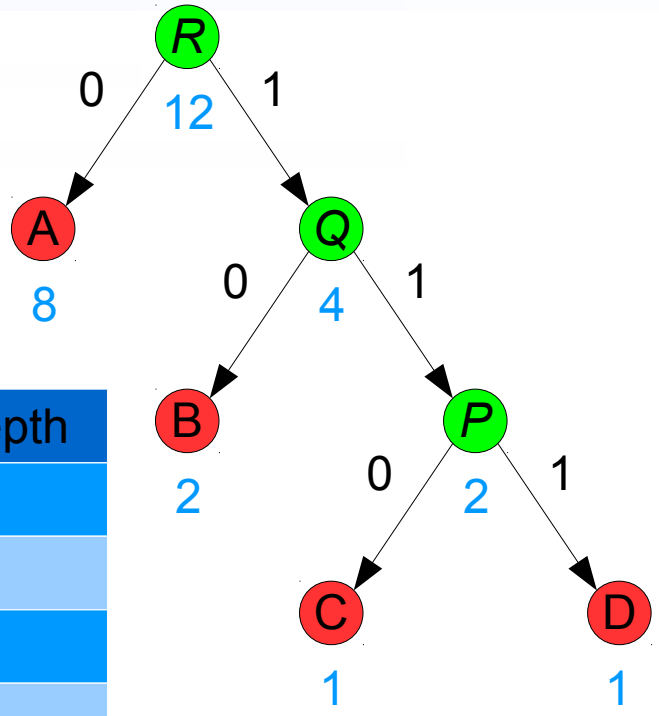
Questions?

Introduction to Video Coding
Part 3: Probability Modeling

Mozilla

Review: Huffman Coding

- Relatively fast, but probabilities limited to power of 2
- LUTs can traverse multiple levels of tree at a time for decoder
- FSM even faster, but limited to a single codebook

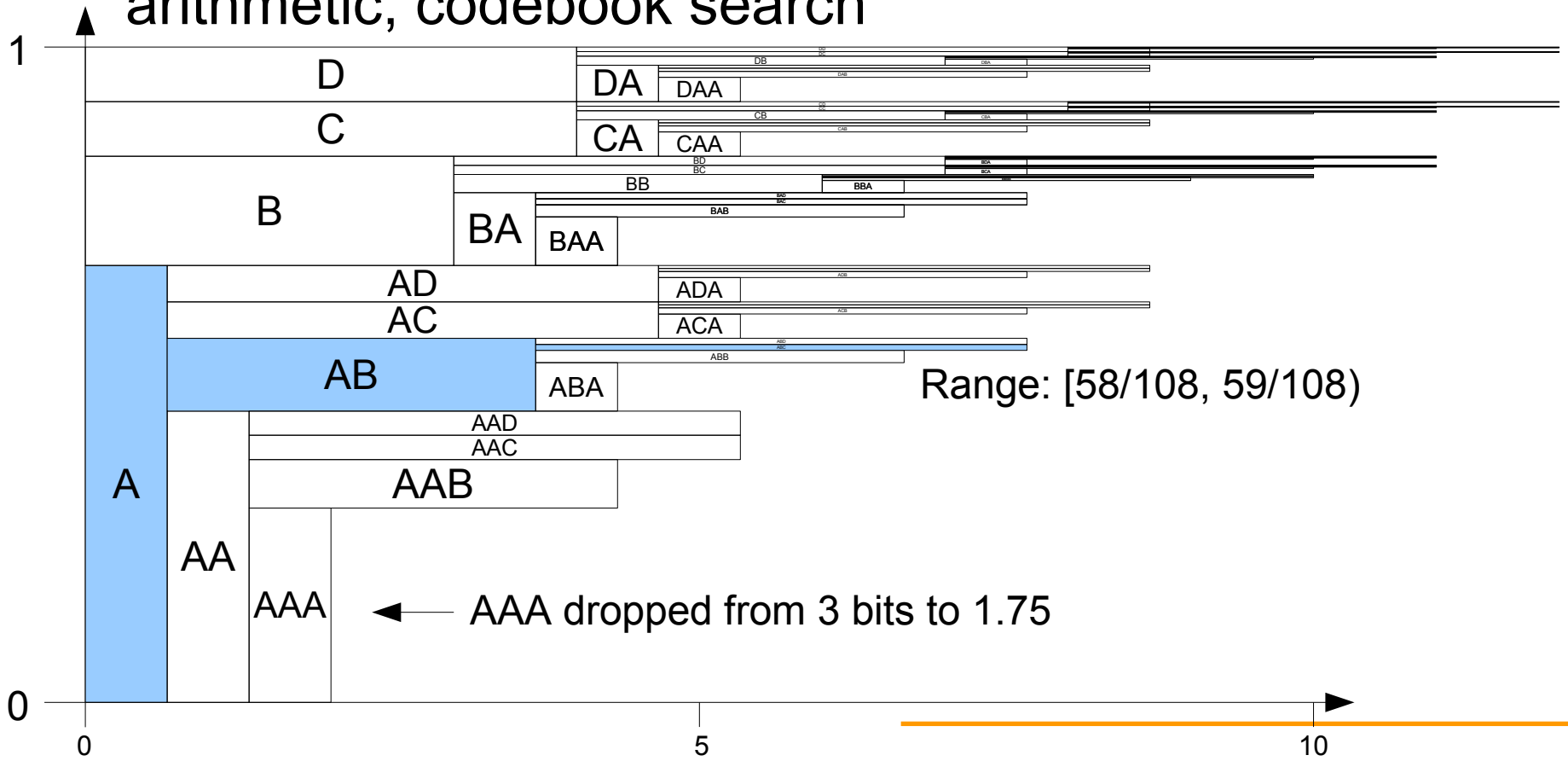


Bits	Node	Depth
00	A	1
01	A	1
10	B	2
11	*P*	2

Bits	Node	Depth
0	C	1
1	D	1

Review: Arithmetic Coding

- Efficient for any probabilities, but somewhat slow
  - Binary coders simplest, larger alphabets require more arithmetic, codebook search



Modeling

- "Modeling" assigns probabilities to each symbol to be coded
- Arithmetic coding completely separates modeling from the coding itself
  - This is its biggest advantage
- Basic approach: partition symbols into "contexts"
  - Can switch from context to context on every symbol
- Model the distribution within a context as if it were *independently, identically distributed* (i.i.d.)

Context Modeling

- How do we get (approximately) i.i.d. Data?
  - Partition it into “contexts” based on the observed values of a symbol’s neighbors
  - Each context has its own probability estimator: $P(x|y_1,...,y_k)$
- Having a separate context for each possible set of values of $y_1,...,y_k$ models the dependence of x on $y_1,...,y_k$

Context Modeling

- Example: A “skip macroblock” flag
  - Skipping is more likely if neighbors were also skipped
- Let $c = 2 \times \text{skip}_{\text{above}} + \text{skip}_{\text{left}}$ be the context index
  - Use four different probability estimators, one for each c
  - All macroblocks with the same value of c code their skip flag using the same probability estimator
- But... how do we estimate probabilities?

Frequency Counts

- Most basic adaptive model
  - Initialize $f(x_i) = 1$ for all x_i
  - Each time x_i is decoded, add 2 to $f(x_i)$
  - $p(x_i) = f(x_i) / total$, where $total = \sum f(x_j)$
- Why add 2 instead of 1?
  - Minimizes worst-case inefficiency
    - Is that the right thing to minimize?
  - Called a Krichevsky-Trofimov estimator (Krichevsky and Trofimov '81)

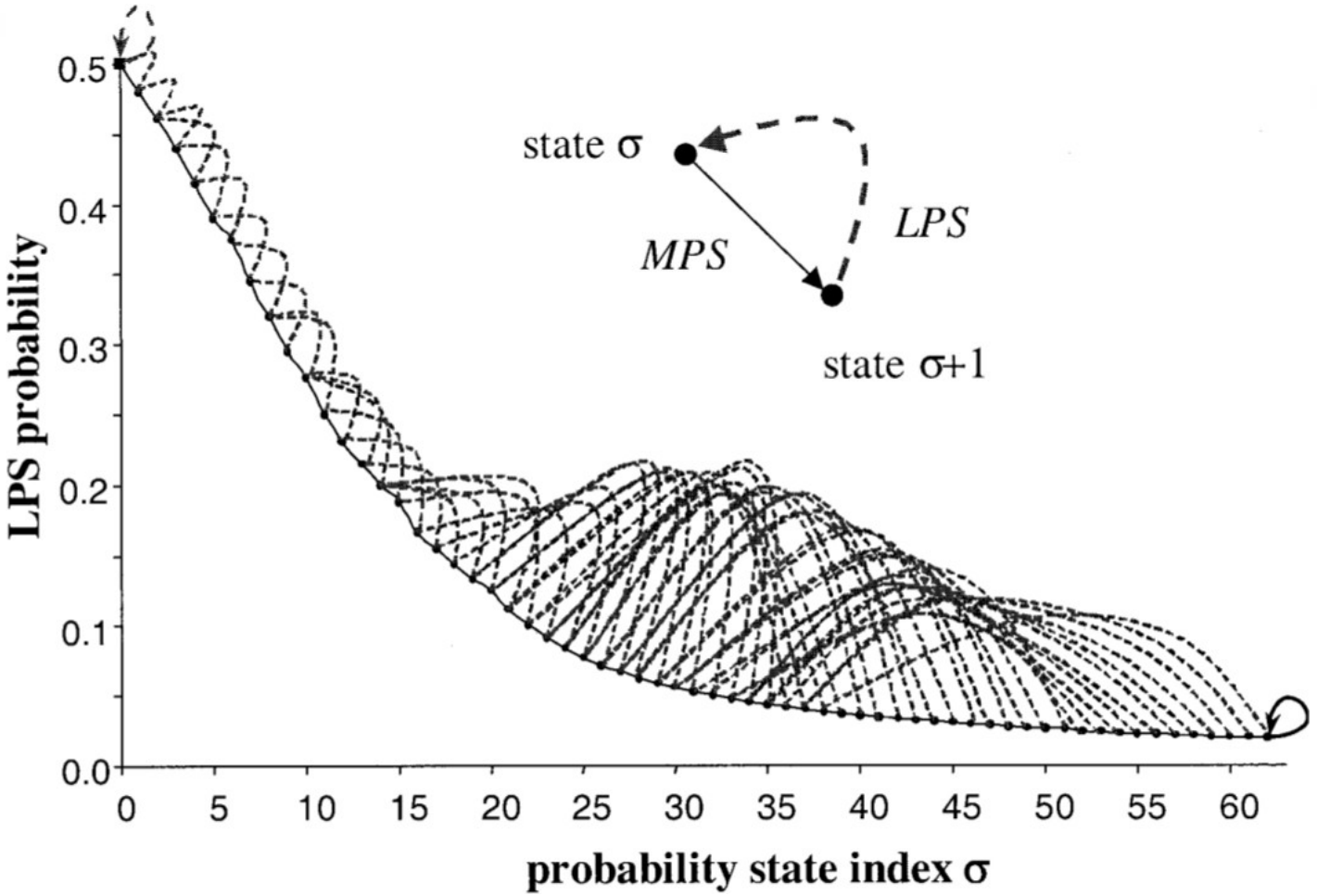
Frequency Counts in Practice

- Rescaling
  - When *total* grows too large, scale all counts by ½ (rounding up to keep them non-zero)
- Distributions not exactly i.i.d.
  - Can imply the need for a faster learning rate
  - Faster learning implies more frequent rescaling → Faster forgetting
- Optimal learning rate varies greatly by context
- Biggest drawback: *total* not a power of 2 (divisions!)

Finite State Machine

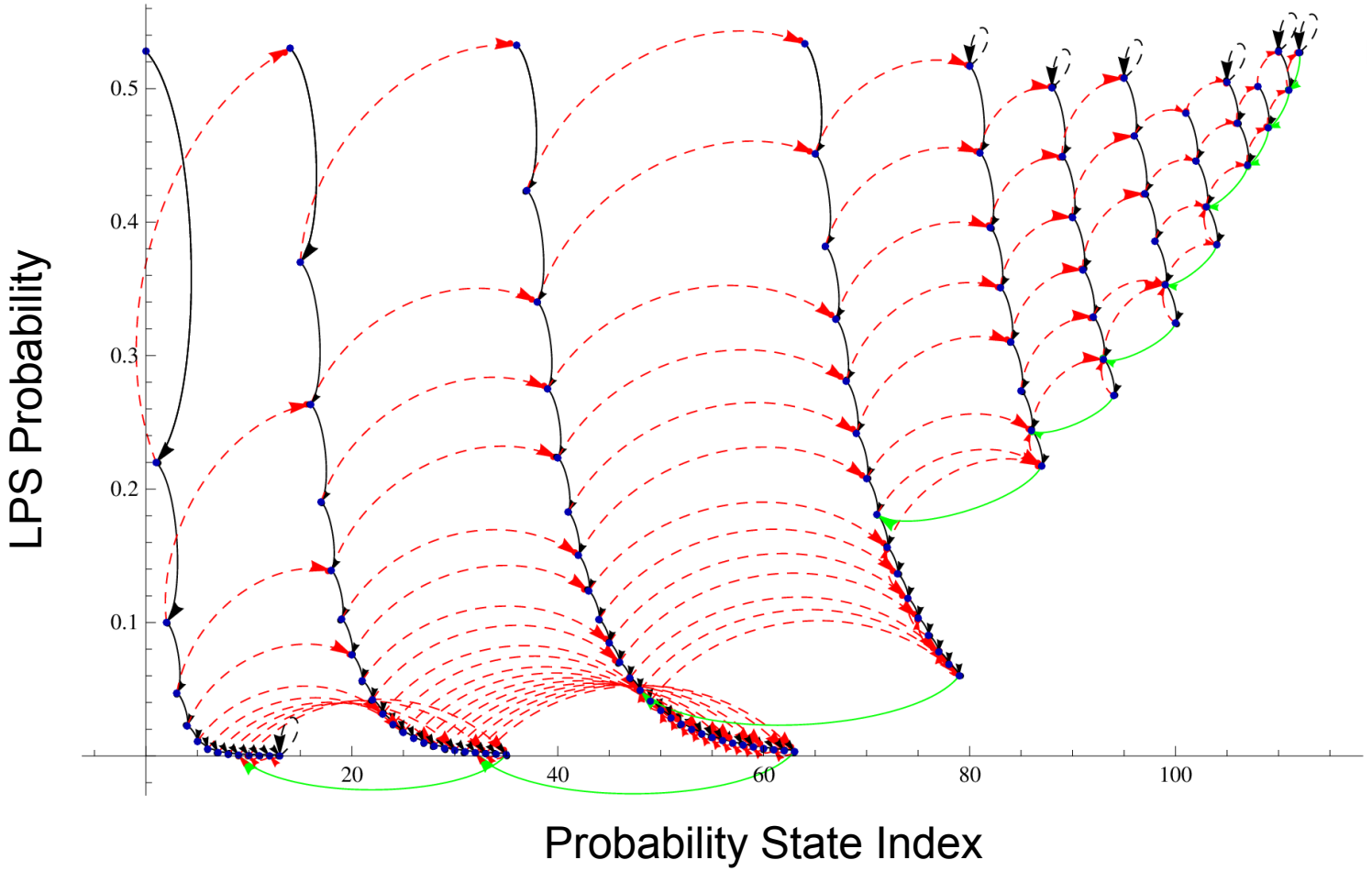
- Only used for binary coders
- Each state has associated probabilities and transition rules
  - LUT maps state and coded symbol to new state
- Example: CABAC (64 states plus reflections)
  - $p_0 = 0.5$, $p_\sigma = \alpha p_{\sigma-1}$, $\alpha = (3/80)^{1/63} \approx 0.949217$
    - Defines probability of "Least Probable Symbol"
    - Probability of "Most Probable Symbol" is $(1 - p_\sigma)$
  - Transitions: $p_{\text{new}} = \begin{cases} \text{Quantize}(\alpha\, p_{\text{old}}), & \text{Coded MPS} \\ \text{Quantize}(1 - \alpha(1 - p_{\text{old}})), & \text{Coded LPS} \end{cases}$

CABAC Transitions



QM Coder (JPEG) Transitions

- "State" can include a notion of learning rate



Designing the FSM

- FSMs are popular (JPEG, JBIG, H.264, Dirac, OMS), but very little research on their design
- Existing methods based on theory, not data
- But it looks much like a classic machine-learning problem: Hidden Markov Models
  - *Hidden* (unobserved) state (e.g., the learning rate)
  - Each state has an unknown probability distribution over the possible (observed) outputs
  - Good training algorithms exist
    - Baum-Welch (1970)
    - Baldi-Chauvin (1994)

Binarization

- All existing video standards use binary alphabets
  - But we want to code non-binary values
- Binarization: converting an integer to bits
  - Want to code as few symbols as possible, on average (for speed and modeling efficiency)
  - Compression comes from arithmetic coder
    - Binarization doesn't need to be perfect

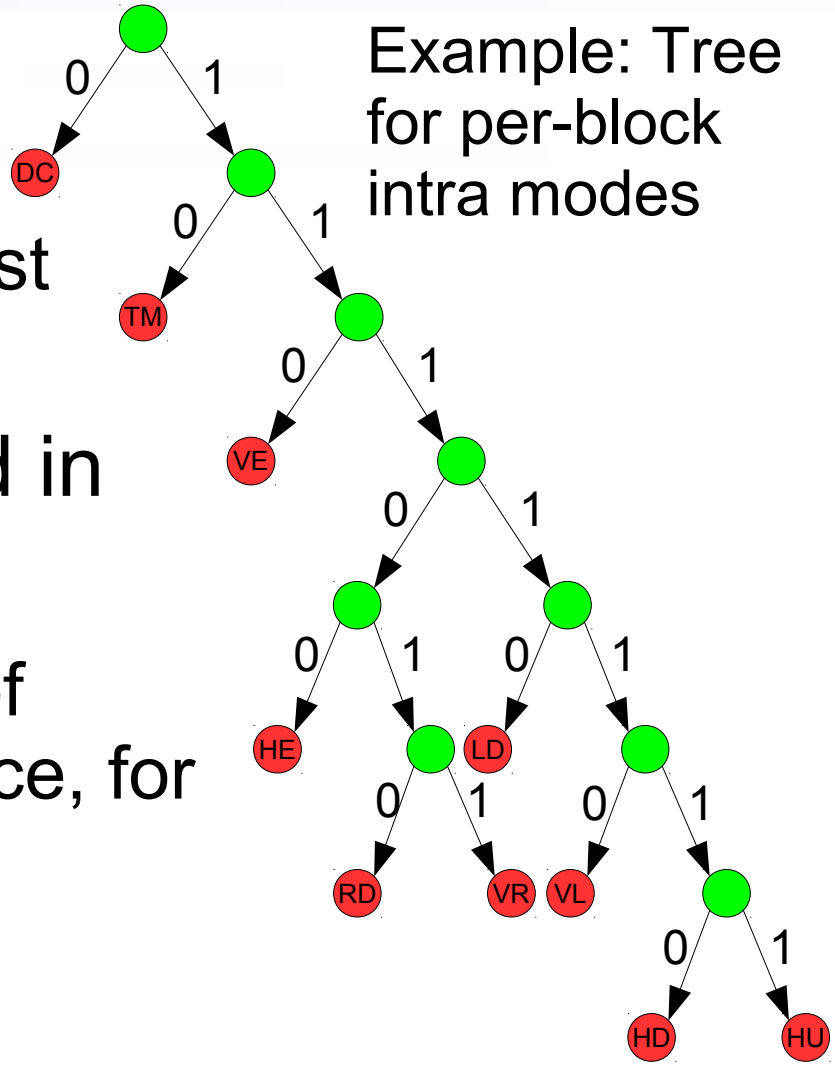
Exp-Golomb Codes

- Binarization method for many syntax elements
  - Four types: unsigned, signed, “coded_block_pattern”, and “truncated”
- Not optimal, but relatively simple
  - Strongly prefers 0’s (coding a 1 takes 3 symbols!)
- Also uses unary, truncated unary, fixed-length, and for MB/sub-MB type a custom code

Bit String	Value(s)
0	0
10x	1...2
110xx	3...6
1110xxx	7...14
11110xxxx	15...30
111110xxxxx	31...62

VP8 “Tree Coder”

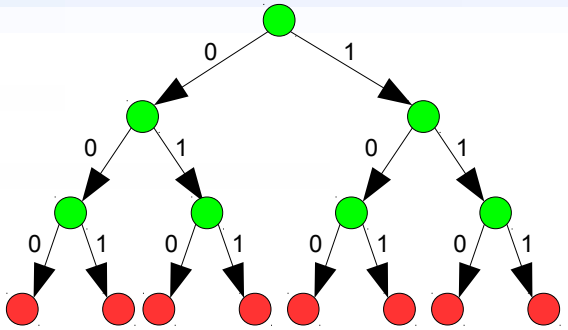
- Binarization is the Huffman coding problem!
  - Uses Huffman trees for most syntax elements
- Each internal node is coded in its own context
  - But decision on which set of contexts to use is made once, for the whole tree



Example: Tree for per-block intra modes

Binary FSM vs. Freq. Counts

- If a symbol can have *N* values, its distribution has *N*-1 degrees of freedom (DOFs)



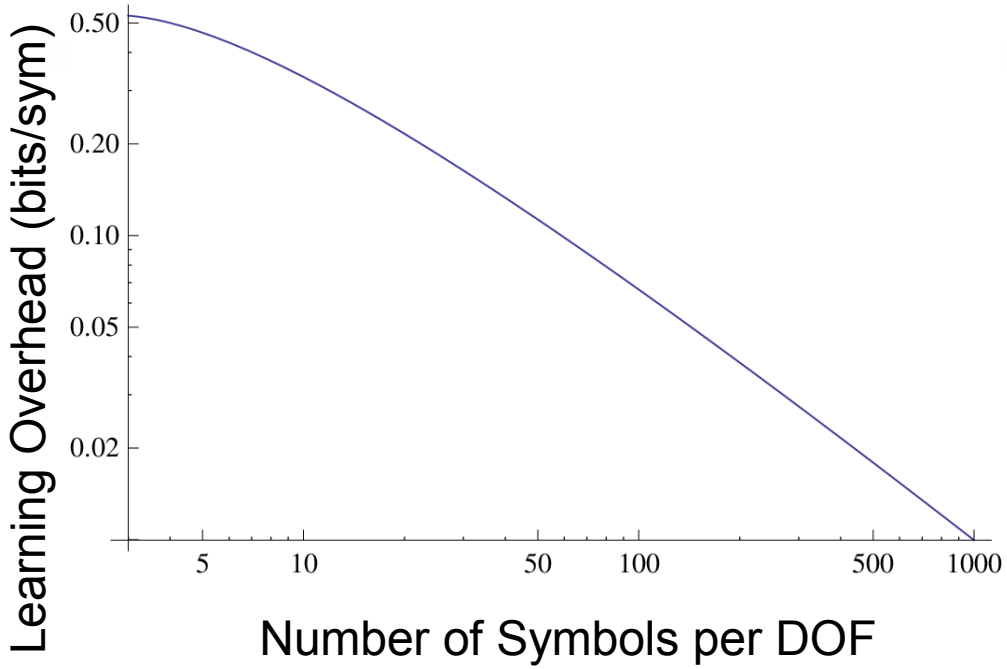
  - Binary contexts: one per binary tree internal node
  - Frequency counts: one per value, minus one constraint (probabilities sum to 1)
- When values are correlated, binary contexts adapt more quickly
  - Taking a branch raises the probability of everything on that side of the tree
  - Example: seeing a MV pointing up means it's more likely to see other MVs that point up

Context Dilution

- What is the cost of adaptation?
  - Learning probabilities adds (roughly) log(*N*) bits of overhead per DOF to code *N* symbols
    - Assumes a static distribution... doesn't count the cost of re-learning if the distribution changes
- How does the number of contexts impact this?
  - Lots of dependencies means lots of contexts
  - Lots of contexts means few symbols per context: *context dilution*

Context Dilution Overhead

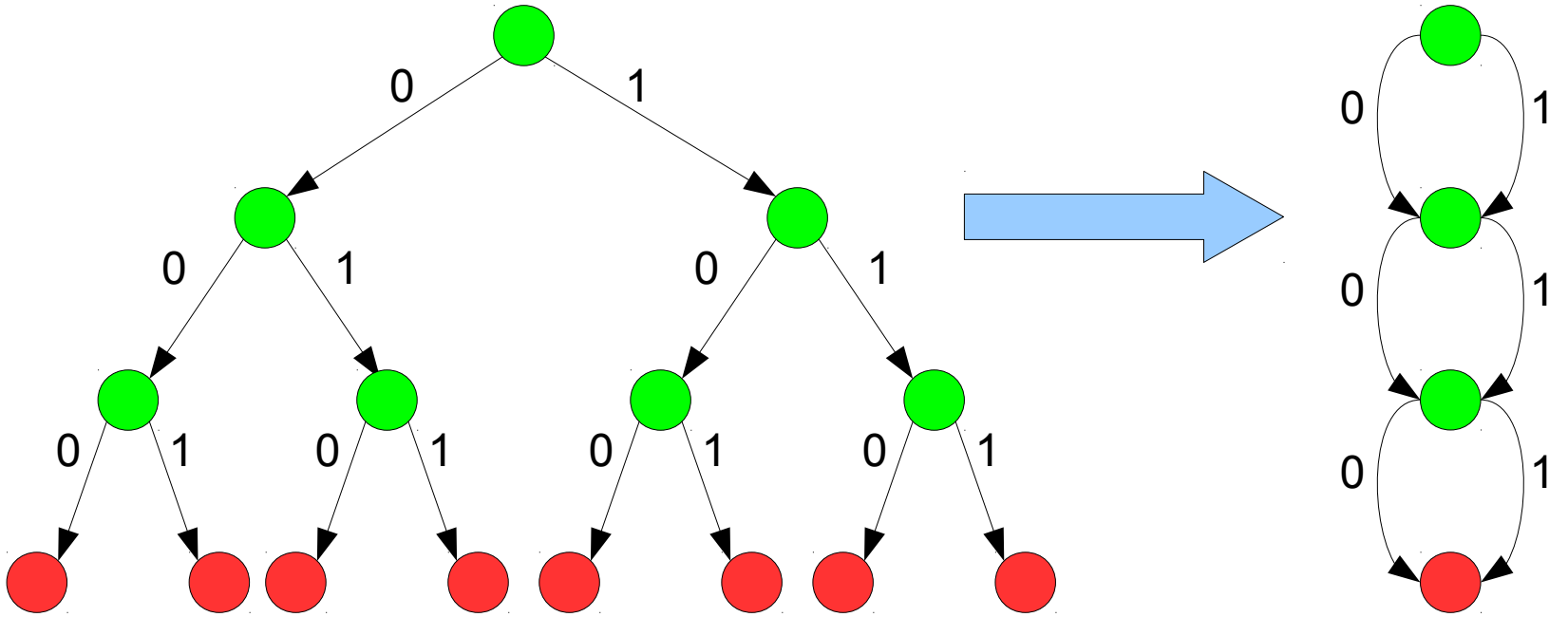
- Larger contexts amortize learning overhead



  - Need 1000 symbols per DOF to get under 1%
- Gain from better modeling must offset overhead

Reducing Contexts

- Merge contexts: $c = \text{skip}_{\text{above}} + \text{skip}_{\text{left}}$
  - 4 contexts reduced to 3
- Replace binary tree with DAG:



  - 7 contexts reduced to 3

Parallelism

- Arithmetic coding is inherently serial
  - State depends on a complicated sequence of rounding and truncation operations
  - Now the bottleneck in many hardware decoders
- Lots of ways to address this
  - All of them have drawbacks

Parallelism: Partitions

- Code different parts of the frame independently
  - Allowed by both H.264 and VP8
- Each uses an independent arithmetic coder
- Can all be decoded in parallel
- BUT
  - Must encode files specially (not on all the time)
  - Can’t predict across partition boundaries
  - Have to re-learn statistics in each partition
  - Non-trivial bitrate impact

Parallelism: HEVC Approach

- Quantize the probabilities (just like with an FSM)
- Map each probability to a *separate* stream
  - Encodes symbols with just that one probability
  - Allows efficient Huffman code (cheap!)
  - Allows FSM decoding (no codebook switching)
- Decode all streams in parallel and buffer output
  - Complicated scheduling problem: not evenly split
- Software implementation half CABAC's speed
  - (Loren Merritt, Personal Communication, 2010)

Parallelism: Paired Coders

- Idea due to David Schleef
- Code pairs of symbols at a time with independent coders
- Carefully schedule when bits are written to/read from the bitstream
  - So they get sent to the right coder
- Drawbacks:
  - If you only want to code one symbol, must clock a “null” (prob=1.0) symbol through other coder
  - Small speed-up

Parallelism: Non-Binary Coder

- Coding a symbol from an alphabet of size 16 is equivalent to coding 4 bits at once
  - This is just the libtheora Huffman decoding approach applied to the VP8 tree coder idea
  - Most coding operations are per-symbol
    - Context selection, state update, renormalization check, probability estimator
  - Exception: codebook search
    - Parallel search needs *N* multipliers for log(*N*) speed-up
    - Can keep it serial for speed/gate-count trade-off
  - Can’t use FSM to estimate probabilities

Non-Binary Coder (cotd.)

- Tested with VP8 tree coder
  - Used mode and motion vector data (not DCT)
  - Binary decoder vs. multi-symbol decoder (pure C implementation):
    - Includes all overheads for non-binary arithmetic coding
    - Serial codebook search (no SIMD)
    - Doesn’t include overheads for probability estimation (VP8 uses static probabilities)
  - Speed-up almost directly proportional to reduction in symbols coded (over a factor of 2 for VP8)

Non-Binary Coder (cotd.)

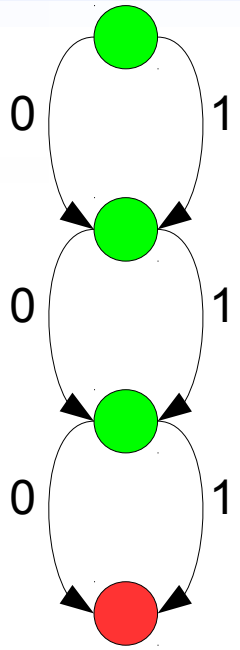
- Estimating probabilities is harder
  - Can use frequency counts, but no easy way to ensure *total* is a power of two
    - Could use approximate coder (e.g., Stuiver's)
    - Could transmit explicit probabilities (à la VP8)
- Partitioning into contexts is harder
  - 2 neighbors with a binary value is $2^2 = 4$ contexts
  - With 16 possible values that's $16^2 = 256$ contexts
    - And each one has 15 parameters to estimate
  - Need other ways to condition contexts

Reducing Freq. Count DOFs

- Frequency counts in large contexts
  - Only boost prob. of one value per symbol
  - Neighboring values often correlated
- Idea: add something to the count of the coded symbol's neighbors, too
  - SIMD makes it as cheap as normal frequency count updates
  - Can simulate any binary model (1st order approx.)
  - A binary model with reduced DOFs yield reduced DOF frequency count models

Reducing Freq. Count DOFs

- Example: Update after coding a 0 from an 8-value context given model at right



$$
\begin{aligned}
&\{1,1,1,1,0,0,0,0\} \\
+\ &\{1,1,0,0,1,1,0,0\} \\
+\ &\{1,0,1,0,1,0,1,0\} \\
\hline
&\{3,2,2,1,2,1,1,0\}
\end{aligned}
$$

- All frequency counts are linear combinations of these 3 vectors or their inverses
  - 3 DOF instead of 7

Beyond Binary Models

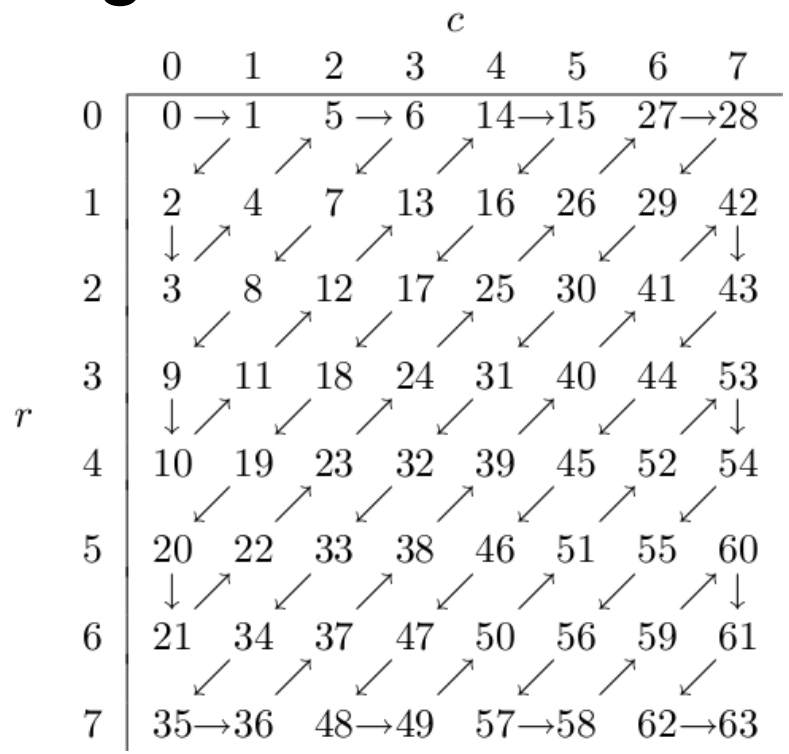
- No need to ape a binary context model
  - How do we train a model from data?
    - Standard dimensionality reduction techniques (PCA, ICA, NMF, etc.)
- No need to use frequency counts
  - Some contexts well-approximated by parametric distributions (Laplace, etc.) with 1 or 2 DOF
  - Must re-compute distribution after each update
    - Can be combined with codebook search, early termination, etc.

Other Improvements

- Per-context learning rate
  - Optimal rate varies drastically
  - Easy to measure from real data (saves 3% bitrate)
  - FSM prob. estimator: expensive (lots of tables)
  - Freq. counts: cheap (1 shift/symbol)
- Rate-dependent context merging
  - Lower rates code fewer symbols
  - Should use fewer contexts (context dilution)
  - No existing codecs do this

Practical Examples: Run-Level Coding

- JPEG and MPEG standards prior to H.264 use (run,level) coding
- Coefficients scanned in zig-zag order
  - “run”: Number of zero coefficients before the next non-zero coefficient
  - “level”: The value of the non-zero coefficient



Practical Examples: Run-Level Coding (cotd.)

- Huffman code built for most common (run,level) pairs
- Additional “End Of Block” symbol indicates no more non-zero coefficients
- Escape code signals uncommon (run,level) pair
- Fast: Lots of coefficients can be decoded from one symbol
- Huffman code is static, so some codewords can produce more coefficients than remain in the block (wasted code space)

Concrete Example: H.264 DCT Coefficients

- Codes locations of non-zero coefficients first, then their values second
- First pass (forwards):
  - coded_block_flag: 0 → skip whole block
    - significant_coeff_flag[i]: one per coeff (except last)
      - last_significant_coeff_flag[i]: one per significant coeff. (except last coeff.): 1 → skip the rest of the block
- Second pass (backwards):
  - coeff_abs_level_minus1[i]: one per sig. coeff.
  - coeff_sign_flag[i]: one per significant coeff.

H.264 DCT Coefficients (cotd.)

- 6 “categories”, each with separate contexts

#	Type	#	Type
0	Luma DC coefficients from Intra 16×16 MB	3	Chroma DC coefficients
1	Luma AC coefficients from Intra 16×16 MB	4	Chroma AC coefficients
2	Luma coefficients from any other 4×4 block	5	Luma coefficients from an 8×8 block

- Context model
  - coded_block_flag
    - Separate contexts for each combination of coded_block_flag values from two neighboring blocks

H.264 DCT Coefficients (cotd.)

- Context model (cotd.)
  - significant_coeff_flag, last_significant_coeff_flag just use position in the list for most categories
    - Chroma DC and 8×8 luma coeffs. merge some contexts
    - No dependence on values in other blocks at all!
- coeff_abs_level_minus1
  - Values less than 14 use unary (0, 10, 110, etc.)
  - Values 14 and larger code an Exp-Golomb “suffix”
    - Codes (coeff_abs_level_minus1-14) with Exp-Golomb

H.264 DCT Coefficients (cotd.)

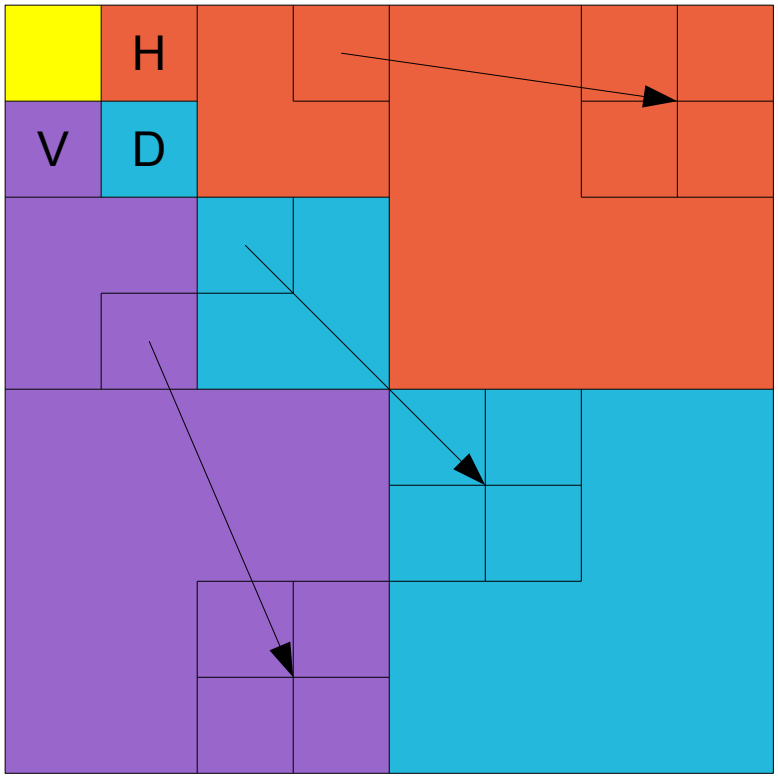
- coeff_abs_level_minus1 context model:
  - First bit
    - 1 context used if there's a value greater than 1 in the block
    - 4 more contexts indexed by the number of 1's seen so far
  - All remaining bits in unary prefix
    - 5 contexts (except for chroma DC, which only uses 2)
    - Indexed by the number of values greater than 1
  - Exp-Golomb suffix: no entropy coding (p=0.5)
- coeff_sign_flag: No entropy coding (p=0.5)

Alternate Design: SPIHT-based

- Lots of research on entropy coding in the 90's for wavelet codecs
- Major examples
  - EZW: Embedded Zerotrees of Wavelets (Shapiro '93)
  - SPIHT: Set Partitioning In Hierarchical Trees (Said and Pearlman '96)
  - EBCOT: Embedded Block Coding with Optimal Truncation (Taubman '98)
- Most also applicable to DCT coefficients

Set Partitioning in Hierarchical Trees

- Arrange coefficients in hierarchical trees
  - Basis functions with similar orientation grouped into the same tree



"Horizontal", "Vertical", and "Diagonal" groups with example parent/child relationships

Other groupings possible, can even switch between them for each block

SPIHT Coding

- "Bit-plane"/"Embedded" coding method
  - Coefficients converted to binary sign-magnitude
  - A coefficient is "significant" at level *n* if its magnitude is at least 2^n.
  - Scan each bitplane
    - Identify coefficients that become significant at this level
      - Code the sign when the coefficient becomes significant
    - Code another bit of already-significant coefficients
- Called "embedded" because you can stop coding at any point to achieve a desired rate

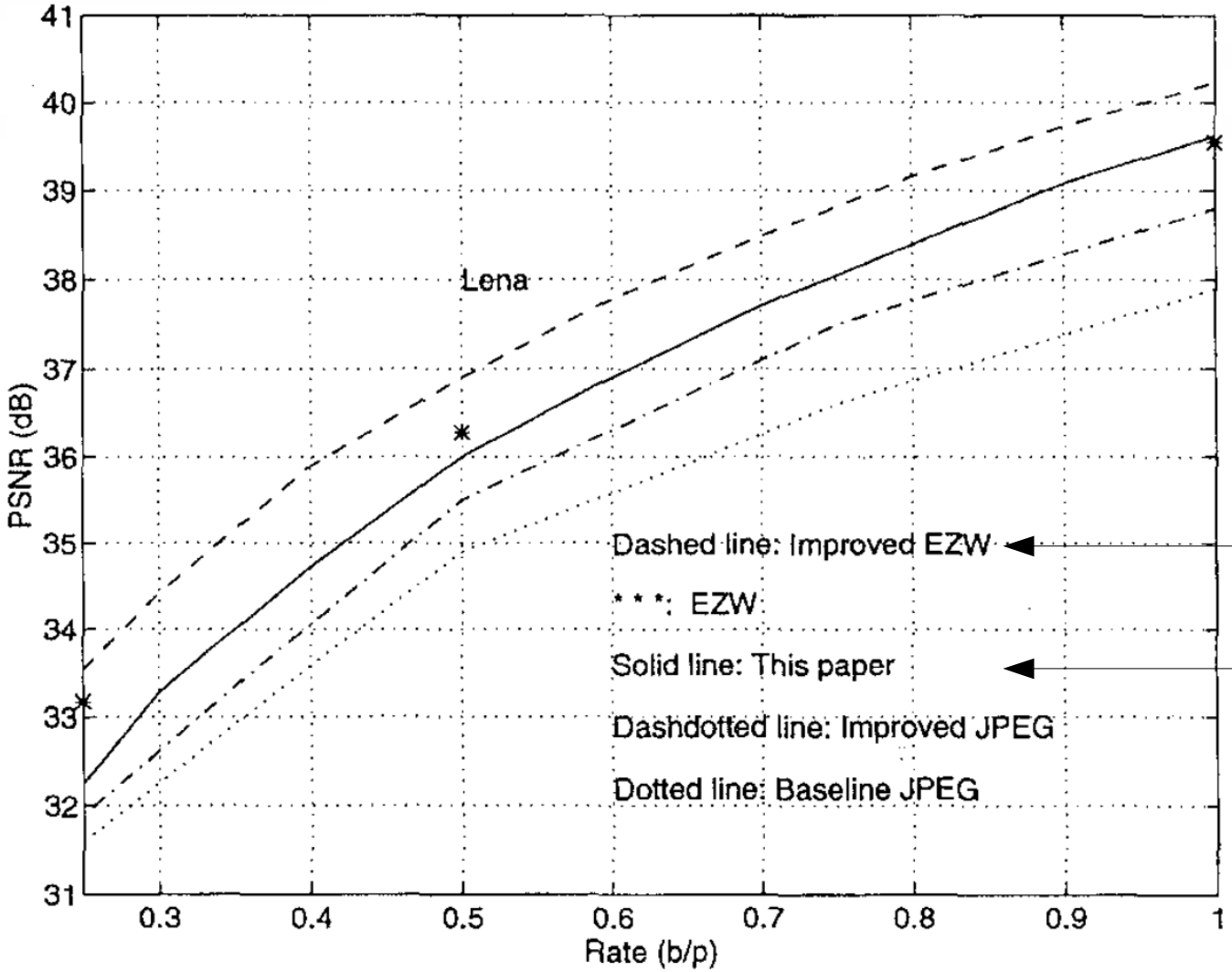
Identifying Significant Coefficients

- "Partition" into three sets using hierarchical trees:
  - List of Significant Pixels (LSP)
  - List of Insignificant Pixels (LIP)
  - List of Insignificant Sets (LIS)
    - Two types of entries
      - Type A, denoted $D(i,j)$, all descendants of node (i,j)
      - Type B, denoted $L(i,j)$, all descendants excluding immediate children

Identifying Significant Coefficients: Algorithm

- Set LSP={}, LIP={(0,0)}, LIS={*D*(0,0)} in top plane
- For (*i*,*j*) in LIP, output one bit to indicate significance
  - If significant, output sign bit, move (*i*,*j*) to LSP
- For each entry in LIS, remove it, and then:
  - Type A: output one bit to indicate if any *D*(*i*,*j*) significant
    - If so, output four bits to indicate significance of each child, move them to LSP, and code a sign
    - Then add *L*(*i*,*j*) to LIS
  - Type B: output one bit to indicate if any *L*(*i*,*j*) significant
    - If so, add all four children to LIS as Type A entries

SPIHT/DCT Results



Z. Xiong, O.G. Guleryuz, M.T. Orchard: "A DCT-Based Embedded Image Coder." IEEE Signal Processing Letters 3(11):289–290, Nov. 1996.

SPIHT

DCT+EZW

SPHIT (cotd.)

- Gives good performance even without an entropy coder
  - Arithmetic coding adds another 0.25-0.3 dB
- But very slow
  - Repeated scanning, list manipulation, branches, etc.
- However...

Non-Embedded SPIHT

- Can make a non-embedded version of SPIHT just by re-arranging the bits
  - Noticed independently by Guo et al. 2006, Charles Bloom 2011: http://cbloomrants.blogspot.com/2011/02/02-11-11-some-notes-on-ez-trees.html
    - Ironically Said and Pearlman made a non-embedded coder strictly inferior to SPIHT (Cho et al. 2005)
- Details for “Backwards” version in Guo et al. 2006
  - Don’t need “Backwards” part (can buffer one block)
  - Single-pass, no lists

Non-Embedded SPIHT: Reducing Symbols/Block

- Code difference in level where $D(i,j)$ and $L(i,j)$ or children become significant
  - Unary code: equivalent to SPIHT
  - But we can use multi-symbol arithmetic coder
- When $D(i,j)$ and $L(i,j)$ are significant at different levels, must have at least one significant child
  - Code top bitplane for all four children at once
- After a coefficient becomes significant, can output rest of bits immediately

Additional References

- R.E. Krichevsky and V.K. Trofimov: “The Performance of Universal Encoding.” IEEE Transactions on Information Theory, IT-27(2):199–207, Mar. 1981.
- J.M. Shapiro: “Embedded Image Coding Using Zerotrees of Wavelet Coefficients.” IEEE Transactions on Signal Processing, 41(12):3445–3462, Dec. 1993.
- A. Said and W. A. Pearlman: “A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees.” IEEE Transactions on Circuits and Systems for Video Technology, 6(3):243–250, Jun. 1996.
- D. Taubman: “High Performance Scalable Image Compression with EBCOT.” IEEE Transactions on Image Processing, 9(7):1158–1170, Jul. 2000.
- Y. Cho, W.A. Pearlman, and A. Said: “Low Complexity Resolution Progressive Image Coding Algorithm: PROGRES (Progressive Resolution Decompression).” In Proc. 12th International Conference on Image Processing (ICIP’05), vol. 3, pp. 49–52, Sep. 2005.
- J. Guo, S. Mitra, B. Nutter, and T. Karp: “A Fast and Low Complexity Image Codec Based on Backwards Coding of Wavelet Trees.” In Proc. 16th Data Compression Conference (DCC’06), pp. 292–301, Mar. 2006

Questions?

Introduction to Video Coding
Part 4: Motion Compensation

Mozilla

Main Idea


Input

⊖


Reference frame

=


Residual

- "Block Matching Algorithms" (BMA)
  - Code an offset for each block (the "motion vector")
  - Copy pixel data from previous frame relative to that offset
- If the match is good, result is mostly zero

Motion Search: Exhaustive

- Check all offsets within some range (±16 pixels)
- Pick one with smallest Sum of Absolute Differences (SAD)

$$\sum_{x=0}^{N-1}\sum_{y=0}^{M-1}\left|I_k(x,y)-I_{k-1}(x+\mathrm{MV}_x,y+\mathrm{MV}_y)\right|$$

  - Error function not smooth in general
    - Global optimum could be anywhere
  - Successive Elimination: algorithm that lets you skip many candidates due to overlap
    - M. Yang, H. Cui, and K. Tang: “Efficient Tree Structured Motion Estimation Using Successive Elimination.” IEE Proceedings – Vision, Image, and Signal Processing, 151(5):369–377, Oct. 2004.

Aperture Problem

- In flat regions, most MVs give approximately the same error
  - Random noise decides which one you choose
- Along an edge, still ambiguous in 1-D subset

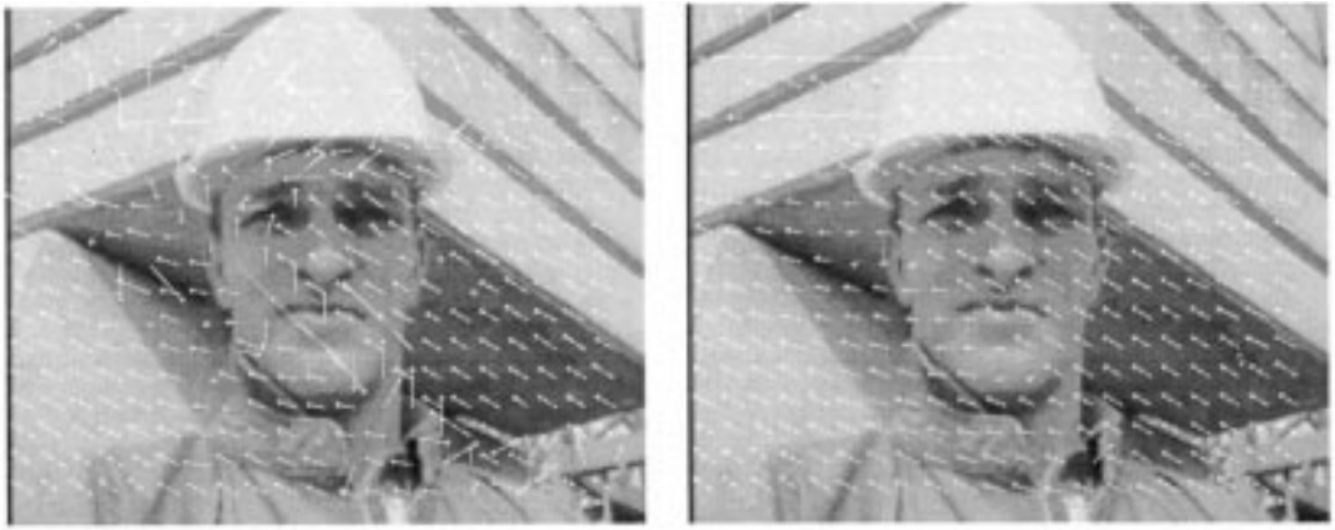


Image from Chen and Willson, 2000.

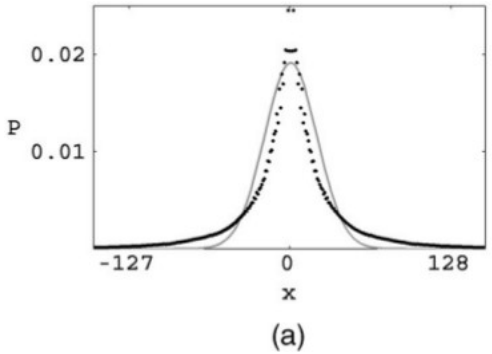
- Solution: account for coding cost of MV: D+λR

Motion Search: Pattern-Based

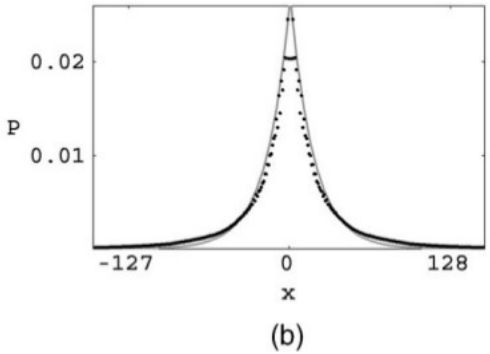
- Start from (0,0)
  - Search, e.g., 4 neighbors (diamond search)
    - Move to the best result
    - Stop when there's no improvement
  - Fast: usually only have to check a few candidates
  - Lots of research into best patterns
    - "Hex" search (6-points): slower, but better quality
- "Zonal Search" (Tourapis et al. 2001, 2002)
  - Predict several MVs from neighbors, stop if under threshold, pattern search from best otherwise

Why SAD?

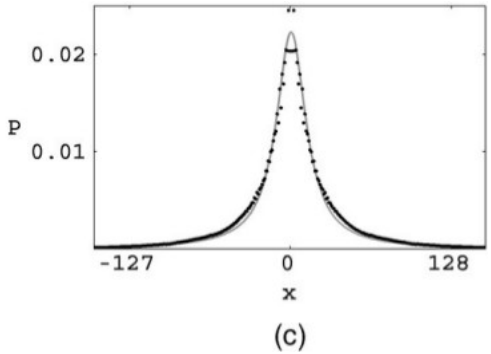
- It's fast
  - No multiplies like Sum of Squared Differences (SSD)
  - SAD of 2×8 pixels in one x86 instruction
- It fits the statistics of the residual better than SSD



Gaussian error (SSD metric)



Laplacian error (SAD metric)

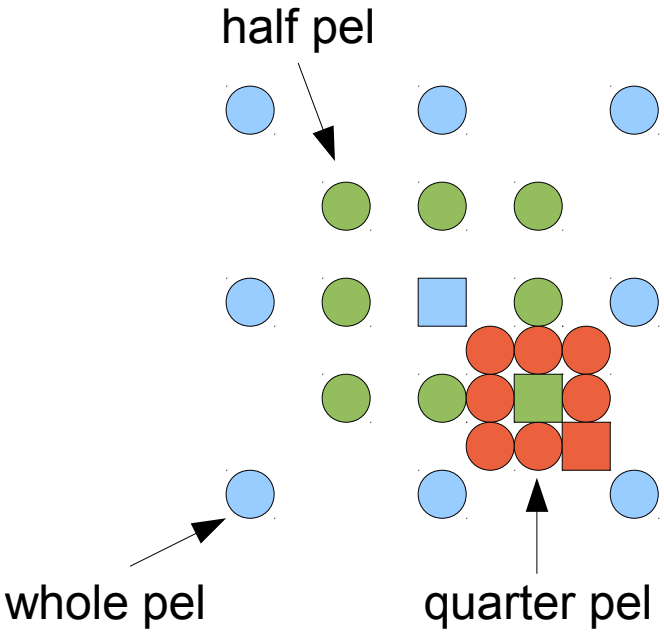


Cauchy error ($\log(1+(x/a)^2)$ metric)

N. Sebe, M.S. Lew, and D.P. Huijsmans: "Toward Improved Ranking Metrics." IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(10):1132–1143, Oct. 2000.

Subpel Refinement

- Motion often isn't aligned with the pixel grid
- After initial search, "refine" motion vector to subpel precision



  - Error well-behaved between pixel locations
  - No need for exhaustive search at subpel resolution
    - ½ pel alone would require 4× as many search points

Subpel Interpolation

- Need to interpolate between pixels in the reference frame for subpel
  - Why is this hard?
  - Aliasing:

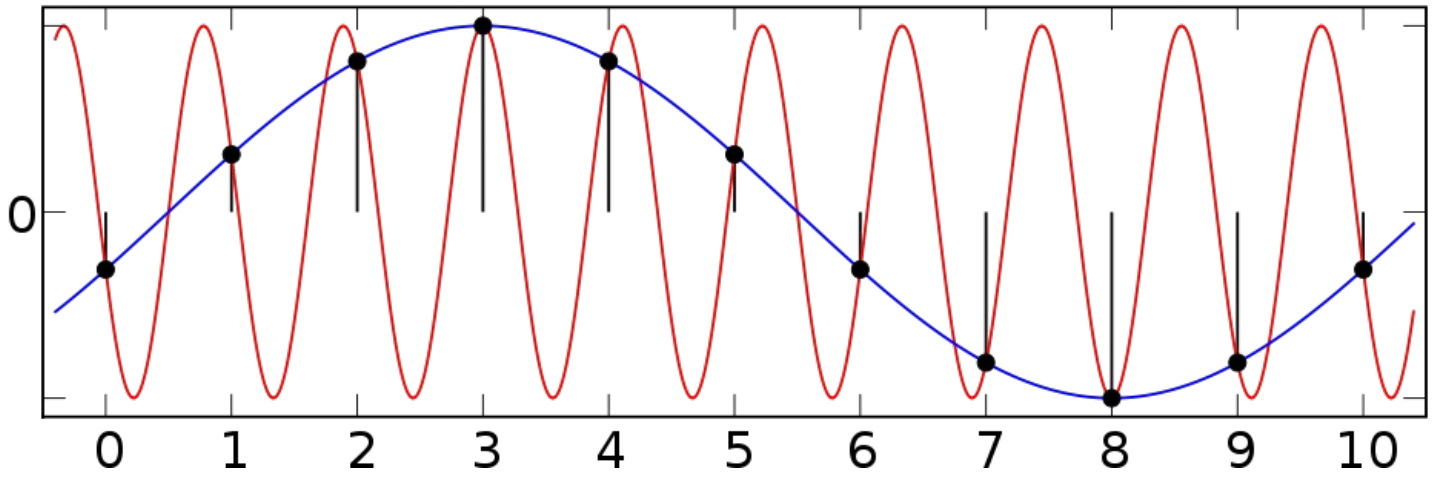
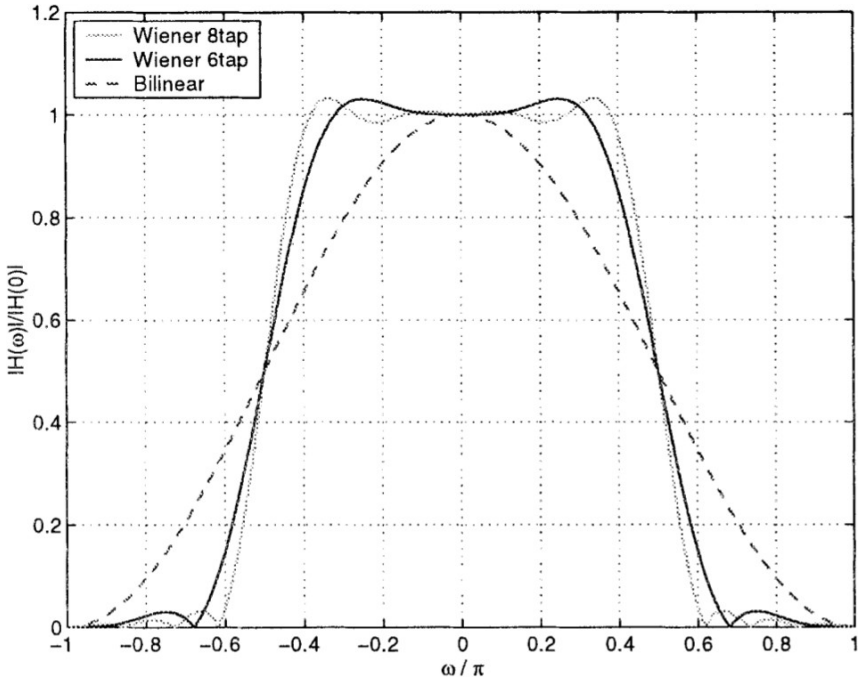


Image from http://en.wikipedia.org/wiki/Aliasing

Subpel Interpolation (cotd.)

- Real signals are not *bandlimited*
  - Hard edges have energy at all frequencies
- But even if they were, practical interpolation filters can't preserve spectrum perfectly



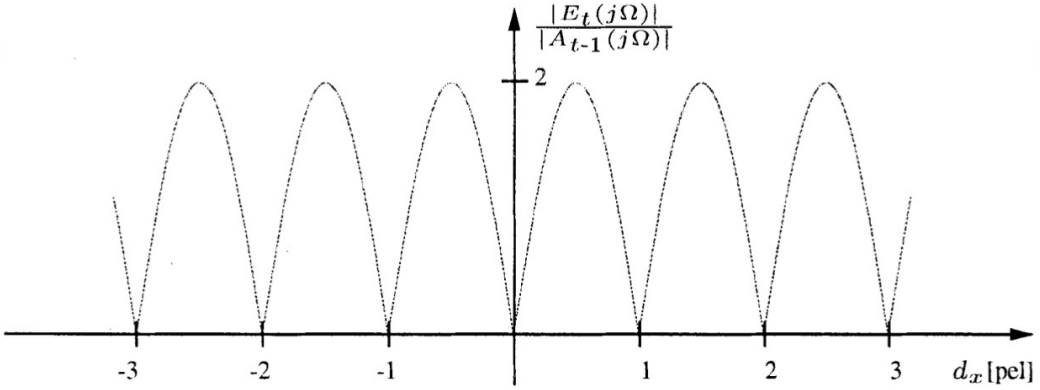
  - MPEG1...4 use bilinear
  - Theora uses linear
  - H.264, VP8 use 6-tap
  - HEVC proposes 12-tap, non-separable filters

Adaptive Interpolation Filters (HEVC proposals)

- Compute “optimal” filters for each frame
  - Transmit one filter per subpel location
    - Requires multiple passes in encoder (slow)
    - Lots of coefficients, significant amount of bitrate
      - Need R-D optimization to decide which to update
    - Some subpel locations only used a few times
      - So many free parameters you’re basically fitting noise
  - Alternative: backwards-adaptive
    - Computationally expensive in the decoder
    - No original to compare to, won’t be as good
- No reason to expect same filter to be good over a whole frame

Subpel Interpolation (cotd.)

- Aliasing error maximal at halfpel locations

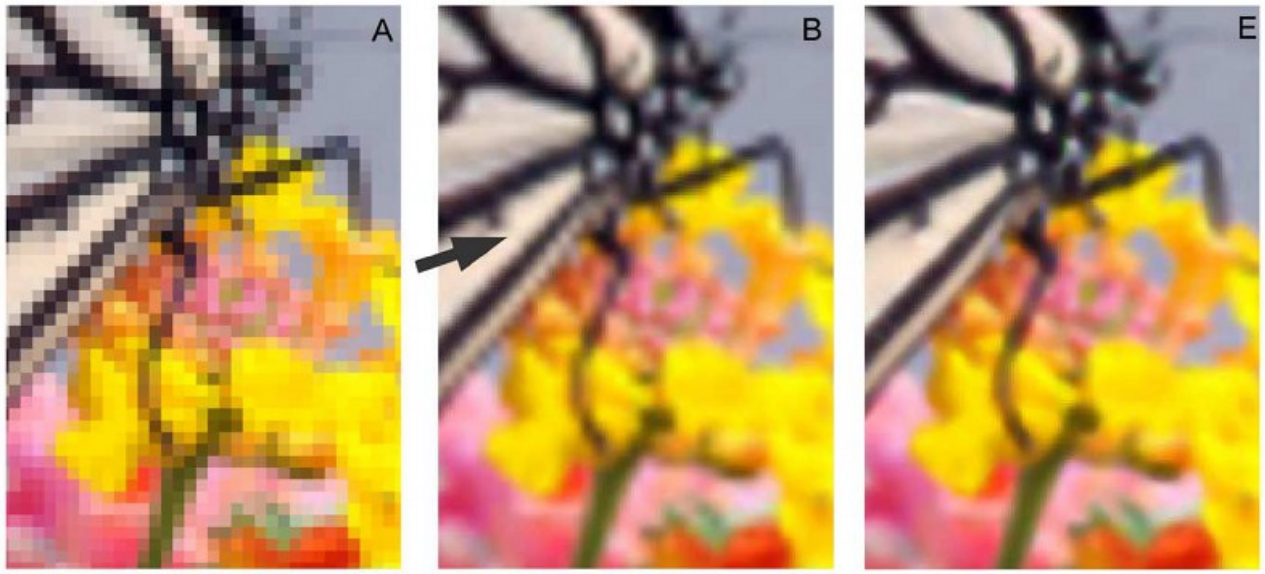


From T. Wedi and H.G. Musmann: "Motion- and Aliasing-Compensated Prediction for Hybrid Video Coding." IEEE Transactions on Circuits and Systems for Video Technology, 13(7):577–586, Jul. 2003.

- Worth spending time to do high-quality upsample by a factor of 2
  - Further subpel refinement can use cheaper interpolation
- But reference image already corrupted by aliasing...

Subpel: Edge-Directed Interpolation

- Lots of recent research into nonlinear filters
  - Can do better than linear ones, but expensive
  - Based on assumptions about natural images

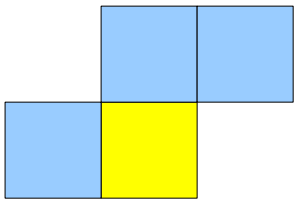


From A. Giachetti and N. Asuni: “Real-Time Artifact-Free Image Upscaling.” IEEE Transactions on Image Processing, 20(10):2760–2768, Oct. 2011.

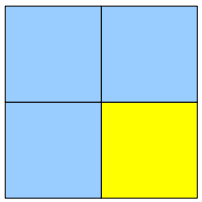
- All we need is something fast enough for video

Coding MVs

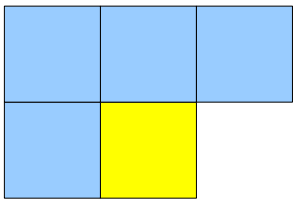
- Compute a predictor from neighboring blocks
  - Usually uses a median to avoid outliers



Median of 3
(MPEG codecs)



Median of 3 (Dirac)



Median of 4 (VC1)

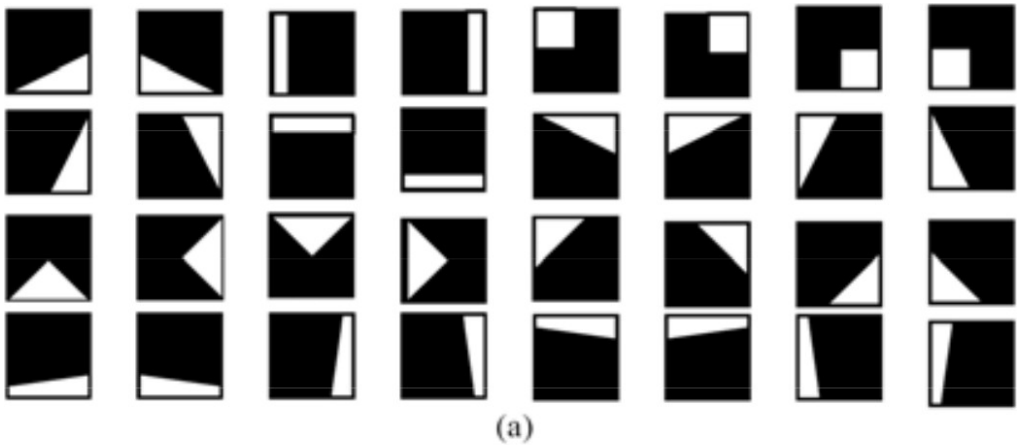
- Subtract predictor, code offset
- Creates a non-trivial dependency between MVs
  - Usually ignored

Variable Block Size

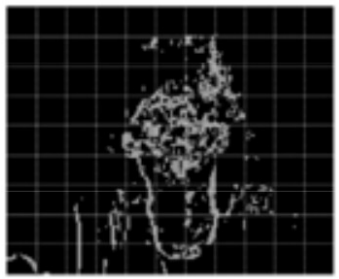
- Can give large gains (0.9 to 1.7 dB) at low rates
- Most codecs support at least two sizes:
  - 16×16 (the default)
  - 8×8 (called “4MV”)
- H.264/VP8 add 8×16, 16×8, 4×4 partition sizes
  - 4×4 isn’t that useful according to Jason Garrett-Glaser
- HEVC expands this to 32×32 or even larger
  - Not necessary with good MV prediction, and good searches in the encoder, but makes things easier

Non-Rectangular Partitions

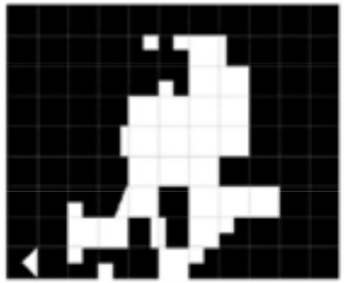
- Several HEVC proposals for 4×16, 16×4, L-shaped, split on arbitary line, etc.
- One good collection of partition shapes:



(a)



(b)

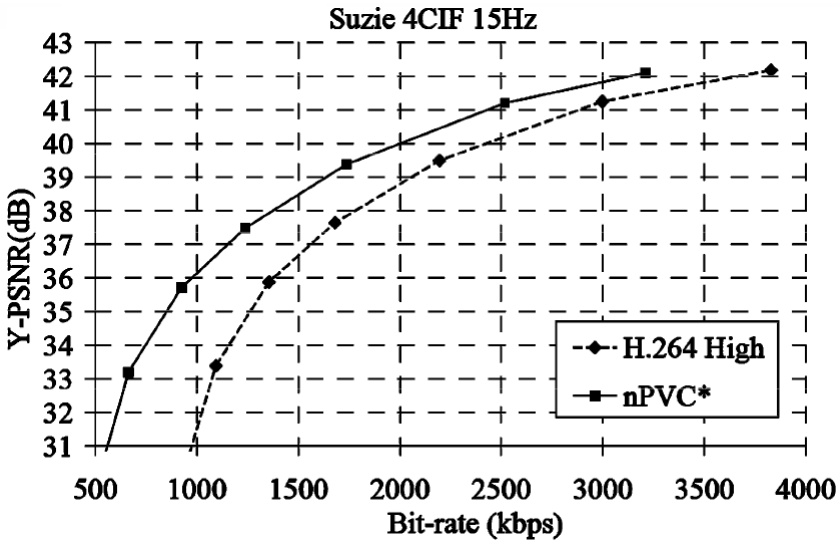


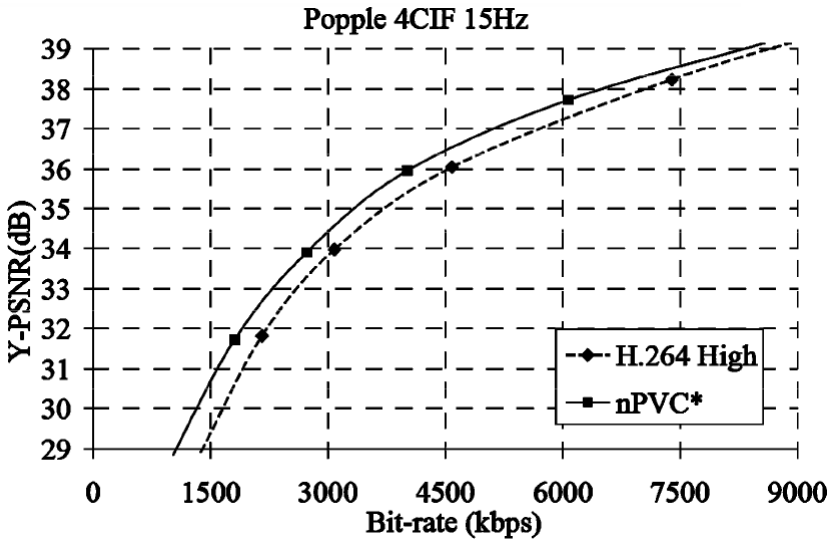
(c)

From M. Paul and M. Murshed: "Video Coding Focusing on Block Partitioning and Occlusion." IEEE Transactions on Image Processing, 19(3):691–701, Mar. 2010.

Non-Rectangular Partitions

- Most performance gain at low bitrates





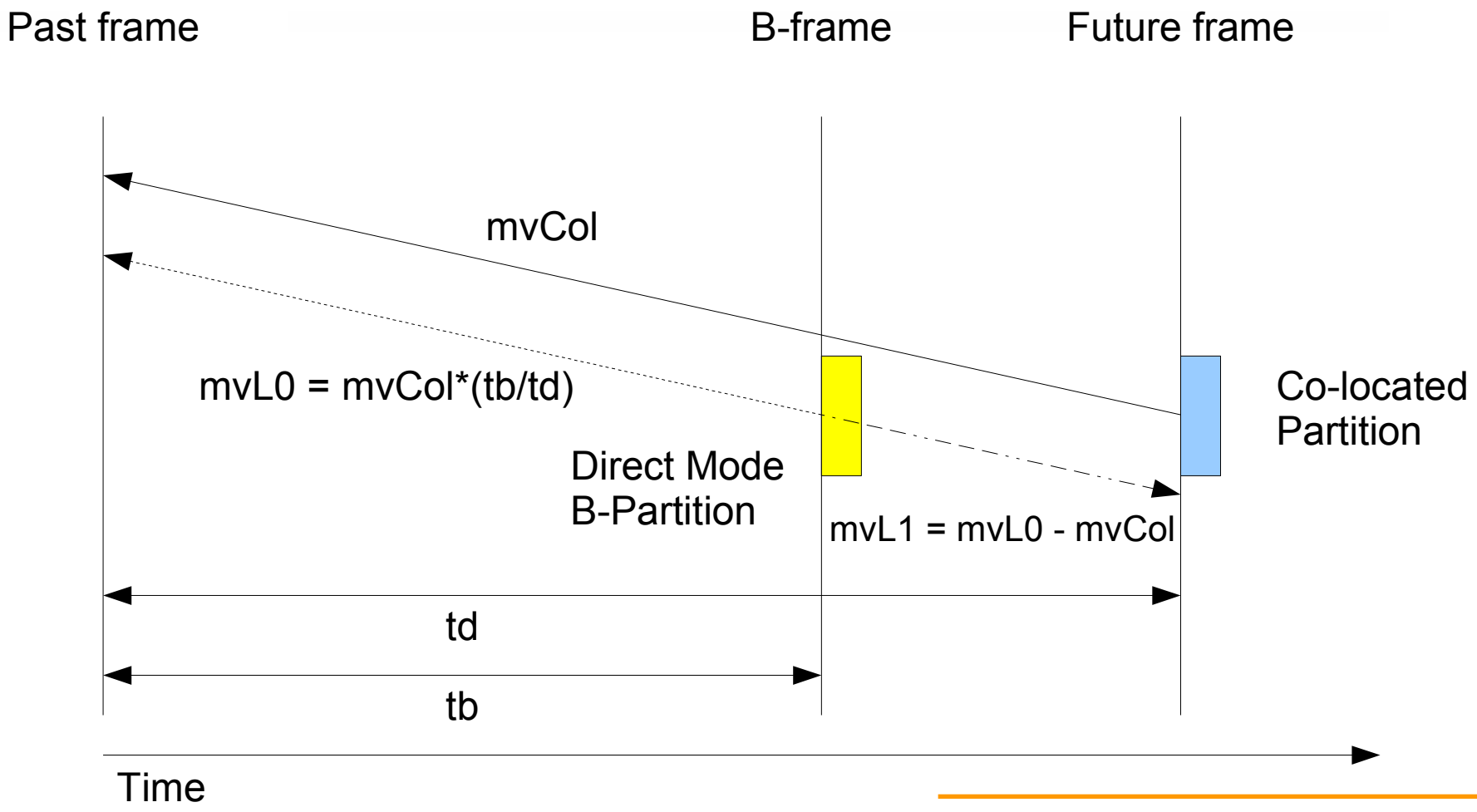
  - Odd partition shapes allow you to
    - Spend fewer bits coding split
    - Code fewer MVs on a split
      - Not sure how important this is given dynamic programming in the encoder

Reference Frame Structure

- Types of frames:
  - I-Frames (Intra Frames, Key Frames)
    - No motion compensation
  - P-Frames (“Predicted”, Inter Frames)
    - MPEG1...4: Single reference frame (last I or P frame)
    - H.264: Choose from multiple reference frames, all from the past
  - B-Frames (“Bi-Predicted”)
    - Up to two reference frames per block (averaged)
    - MPEG1...4: Closest I- or P-frame from past *and future*
    - H.264: Basically arbitrary choice (“Generalized B-frames”)

B-Frames: Direct Mode

- Bi-prediction without signaling



“Multihypothesis” Motion Compensation

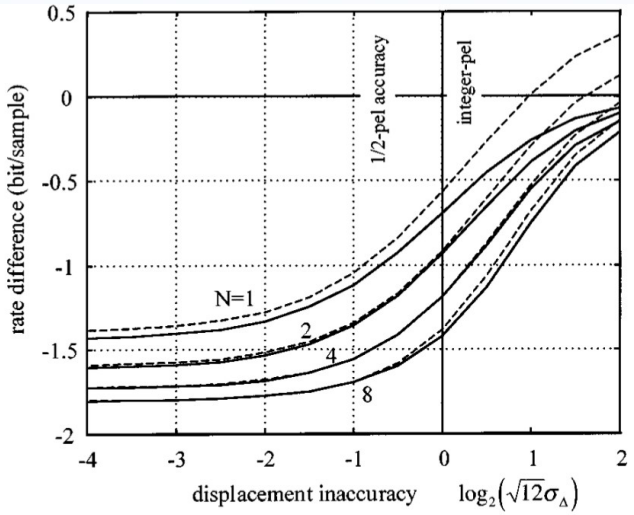


Fig. 8. Rate difference compared to optimum intraframe coding due to multihypothesis motion-compensated prediction as a function of displacement error variance for combining different numbers of hypotheses N. Residual noise level RNL $= -24$ dB. Solid lines assume an optimum filter F, dashed curves show the case where hypotheses are simply averaged.

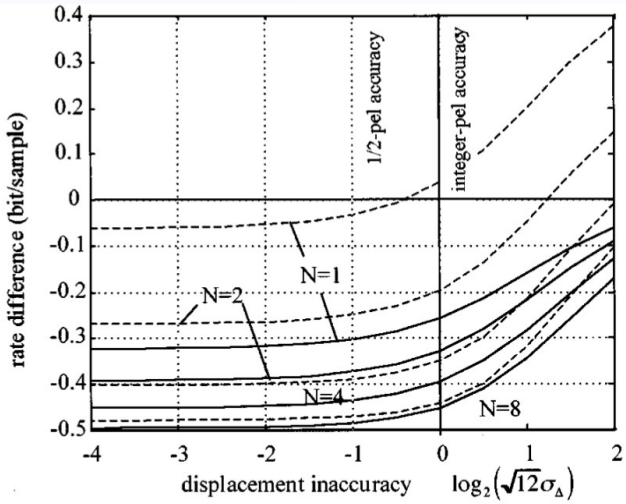


Fig. 9. Rate difference compared to optimum intraframe coding due to multihypothesis motion-compensated prediction as a function of displacement error variance for combining different numbers of hypotheses N. Residual noise level RNL $= -12$ dB. Solid lines assume an optimum filter F, dashed curves show the case where hypotheses are simply averaged.

From B. Girod: “Efficiency Analysis of Multihypothesis Motion-Compensated Prediction for Video Coding.” IEEE Transactions on Image Processing, 9(2):173–183, Feb. 2000.

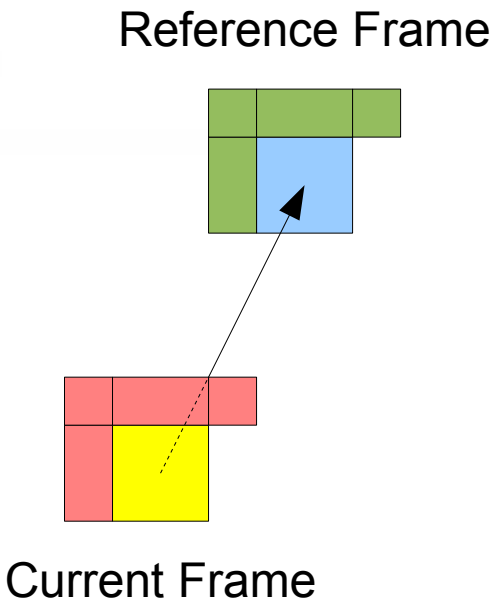
- Diminishing returns for more references
- Optimal MV precision depends on noise level
- Half-pel is a form of multihypothesis prediction!

Weighted Prediction

- Fades between scenes are common, but very expensive to code
- Prior to H.264, B-frames could only average two frames
- H.264 added weighted prediction
  - Each reference gets one weight for entire frame
    - Can range from -128.0 to 127.0!
  - No way to disable on block-by-block basis
    - Can’t handle shadows in illumination changes that only affect part of a frame

Template Matching Weighted Prediction

- HEVC proposal
  - Look at area around the block you're copying from in the reference frame
  - Compare it to area you've already coded in the current frame
  - Compute the optimal prediction weight for the overlap
- Weight varies block by block, no extra signaling required

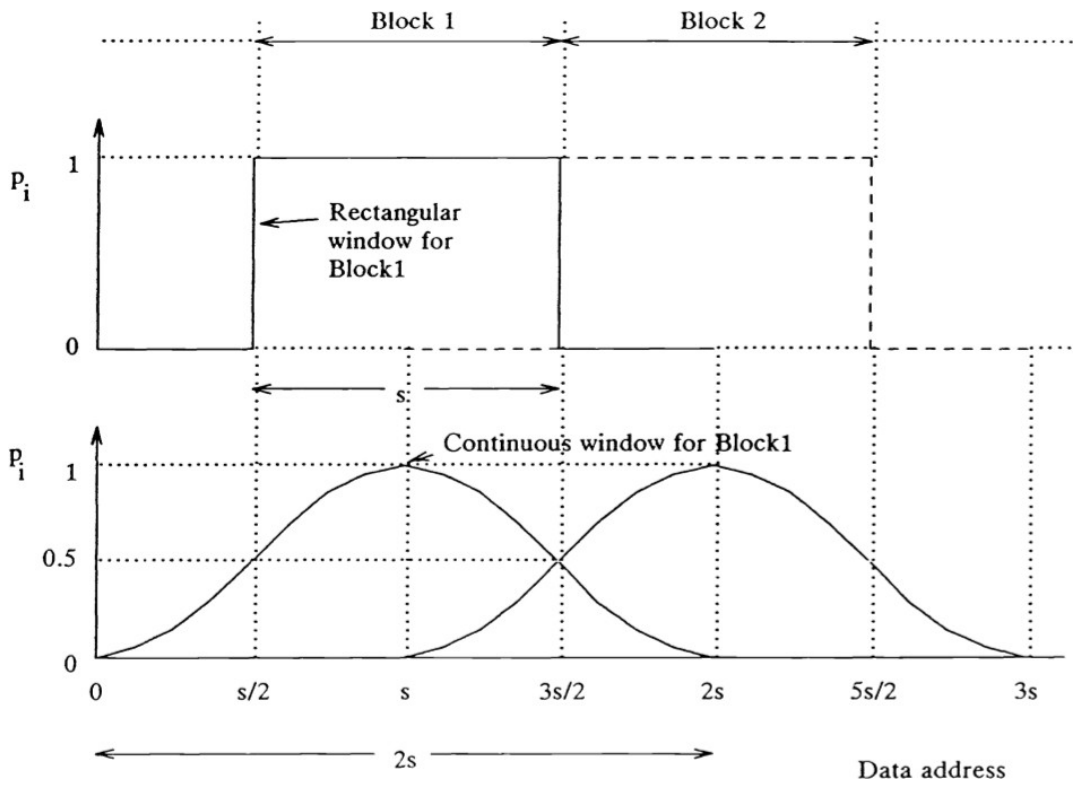


Blocking Artifacts

- Lapped transforms eliminate blocking artifacts from the residual
- But block-based motion compensation *adds* more discontinuities on block boundaries
  - Hard to code with a lapped transform
- Two approaches to avoid them
  - Overlapped Block Motion Compensation (OBMC)
  - Control Grid Interpolation (CGI)

Overlapped Block Motion Compensation (OBMC)

- Overlap the predictions from multiple nearby MVs, and blend them with a window



  - Also a form of multihypothesis prediction

From H. Watanabe and S. Singhal: “Windowed Motion Compensation.” In Proc. SPIE Visual Communications and Image Processing ‘91, vol. 1605, pp. 582–589, Nov. 1991.

OBMC (cotd.)

- Used by Dirac
  - Also want to avoid blocking artifacts with wavelets
- PSNR improvements as much as 1 dB
- Issues
  - Motion vectors no longer independent
    - Can use iterative refinement, dynamic programming (Chen and Willson, 2000), bigger cost of ignoring this
  - Low-pass behavior
    - Blurs sharp features
  - Handling multiple block sizes (2-D window switching)

Multiresolution Blending

- Technique due to Burt and Adelson 1983
  - Decompose predictor into low-pass and high-pass subbands LL, HL, LH, HH
    - Just like a wavelet transform
  - Blend with small window in high-pass bands
  - Recursively decompose LL band
- Prorposed simplification
  - One level of Haar decomposition
  - Blend LL band like OBMC, copy rest like BMA
  - Reduces OBMC multiplies by 75%

Control Grid Interpolation (CGI)

- Instead of blending predictors like OBMC, blend MVs (like texture mapping)
  - More expensive: Can’t use large loads
  - Need subpel interpolation at much finer resolution
  - Can’t model motion discontinuities, multiple reference frames
- Harder to estimate MVs
  - Can’t ignore dependencies anymore
  - Little PSNR improvement over BMA without iterative refinement, dynamic programming

Switching Between OBMC and CGI

- Various published research suggests using both techniques is better than either one alone
  - Switch on block-by-block basis
  - In the range of 0.5 dB better
- None of them avoid blocking artifacts at switch
- Alternate approach
  - Choose which method to use on the *edges* of blocks
  - Pick interpolation formulas that can achieve desired behavior on an edge

Adaptive Switching

- VVVV $I(w_0\boldsymbol{m}_0+w_1\boldsymbol{m}_1+w_2\boldsymbol{m}_2+w_3\boldsymbol{m}_3)$

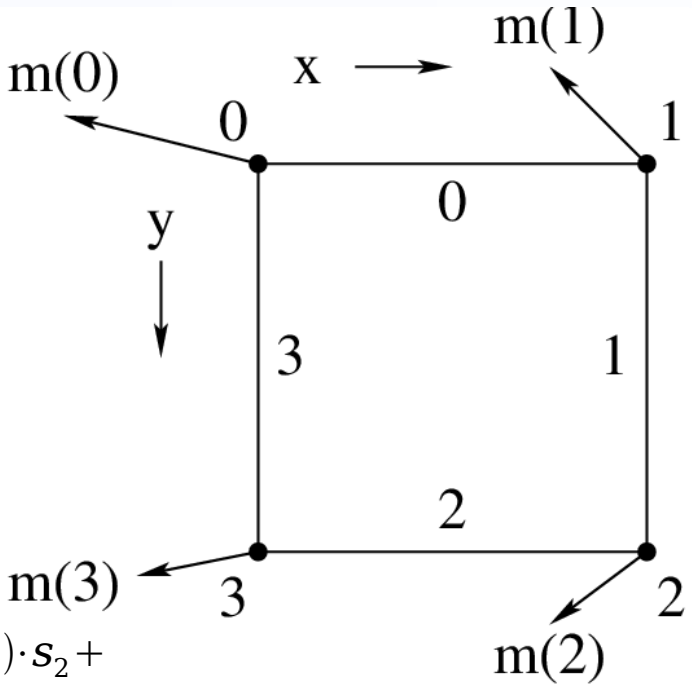
- BVVV $I((w_0+w_1)\boldsymbol{m}_0+w_2\boldsymbol{m}_2+w_3\boldsymbol{m}_3)\cdot s_0+$
  $I((w_0+w_1)\boldsymbol{m}_1+w_2\boldsymbol{m}_2+w_3\boldsymbol{m}_3)\cdot s_1+$
  $I(w_0\boldsymbol{m}_0+w_1\boldsymbol{m}_1+w_2\boldsymbol{m}_2+w_3\boldsymbol{m}_3)\cdot(s_2+s_3)$

- BVBV $I((w_0+w_1)\boldsymbol{m}_0+(w_2+w_3)\boldsymbol{m}_3)\cdot(s_0+s_3)+$
  $I((w_0+w_1)\boldsymbol{m}_1+(w_2+w_3)\boldsymbol{m}_2)\cdot(s_1+s_2)$

- VVBB $I((1-w_1)\boldsymbol{m}_0+w_1\boldsymbol{m}_1)\cdot s_0+I(w_1\boldsymbol{m}_1+(1-w_1)\boldsymbol{m}_2)\cdot s_2+$
  $I(w_0\boldsymbol{m}_0+w_1\boldsymbol{m}_1+w_2\boldsymbol{m}_2+w_3\boldsymbol{m}_3)\cdot s_1+I(\boldsymbol{m}_3)\cdot s_3$

- VBBB $I((1-w_1)\boldsymbol{m}_0+w_1\boldsymbol{m}_1)\cdot s_0+I(\boldsymbol{m}_2)\cdot s_2+$
  $I(w_0\boldsymbol{m}_0+(1-w_0)\boldsymbol{m}_1)\cdot s_1+I(\boldsymbol{m}_3)\cdot s_3$

- BBBB $I(\boldsymbol{m}_0)\cdot s_0+I(\boldsymbol{m}_1)\cdot s_1+I(\boldsymbol{m}_2)\cdot s_2+I(\boldsymbol{m}_3)\cdot s_3$



w_i : bilinear vector weights

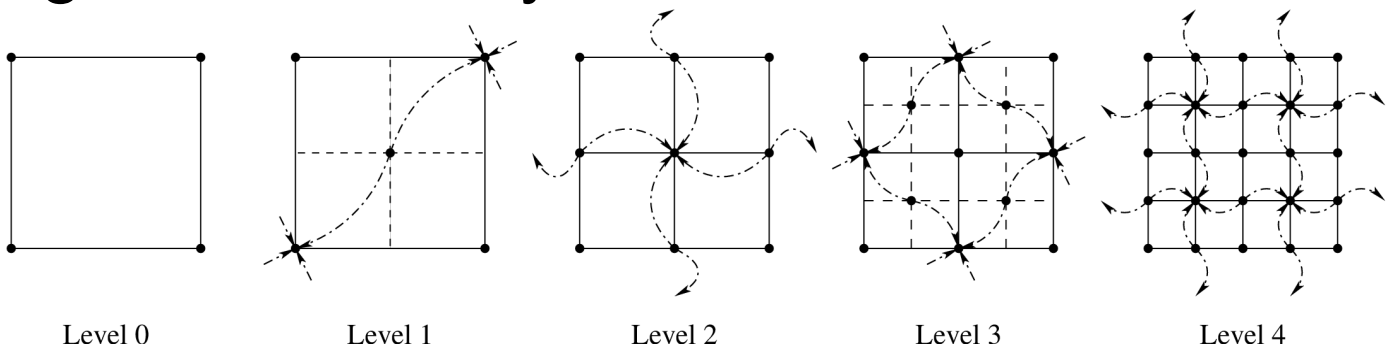
s_j : bilinear image weights

Variable Block Size

- Need a way change block size that doesn't create blocking artifacts
- Dirac subdivides all blocks to the smallest level and copies MVs
  - Lots of setup overhead for smaller blocks
  - Redundant computations for adjacent blocks with same MV
  - Only works for OBMC, not CGI

Adaptive Subdivision

- Slight modifications to the previous formulas allow artifact-free subdivision in a 4-8 mesh
  - Neighbors differ by at most 1 level of subdivision



  - Fine-grained control (MV rate doubles each level)
  - Efficient R-D optimization methods (Balmelli 2001)
    - Developed for compressing triangle mesh/terrain data
  - Larger interpolation kernels, less setup overhead, fewer redundant calculations

Demo

References

- F. Dufaux and F. Moscheni: “Motion Estimation Techniques for Digital TV: A Review and a New Contribution.” Proceedings of the IEEE, 83(6):858–876, Jun. 1995.
- A.M. Tourapis: “Enhanced Predictive Zonal Search for Single and Multiple Frame Motion Estimation.” Proc. SPIE Visual Communications and Image Processing, vol. 4671, pp. 1069–1079, Jan. 2002.
- A.M. Tourapis, O.C. Au, M.L. Liou: “Highly Efficient Predictive Zonal Algorithms for Fast Block-Matching Motion Estimation.” IEEE Transactions on Circuits and Systems for Video Technology, 12(10):934–947, Oct. 2002.
- M.C. Chen and A.N. Willson, Jr.: “Motion-Vector Optimization of Control Grid Interpolation and Overlapped Block Motion Compensation Using Iterated Dynamic Programming.” IEEE Transactions on Image Processing, 9(7):1145–1157, Jul. 2000.
- K.-C. Hui and W.-C. Siu: “Extended Analysis of Motion-Compensation Frame Difference for Block-Based Motion Prediction Error.” IEEE Transactions on Image Processing, 16(5):1232–1245, May 2007.
- W. Zheng, Y. Shishikui, M. Naemura, Y. Kanatsugu, and S. Itoh: “Analysis of Space-Dependent Characteristics of Motion-Compensated Frame Differences Based on a Statistical Motion Distribution Model.” IEEE Transactions on Image Processing, 11(4):377–389, Apr. 2002.
- P.J. Burt and E.H. Adelson: “A Multiresolution Spline with Application to Image Mosaics.” ACM Transactions on Graphics, 2(4):217–236, Oct. 1983.
- L. Balmelli: “Rate-Distortion Optimal Mesh Simplification for Communications.” Ph.D. Thesis, École Polytechnique Fédérale de Lausanne, 2000.

Questions?