University of California
Los Angeles

Variable-Precision Arithmetic For Vector Quantization

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Raffi Dionysian

1994

The dissertation of Raffi Dionysian is approved.

Bruce Ho

Andrew B. Kahng

Lawrence McNamee

Kung Yao

Miloš D. Ercegovac, Committee Chair

University of California, Los Angeles

1994

Table of Contents

List of Figures

List of Tables

Symbols

b_i	$\equiv e_i - e_1$
c	index of codevector chosen in classification.
$d(\mathbf{x}, \hat{\mathbf{x}})$	distance between $\mathbf{x}$ and $\hat{\mathbf{x}}$.
$\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)$	$\equiv d(\mathbf{x}, \hat{\mathbf{x}}^i) - d(\mathbf{x}, \hat{\mathbf{x}}^h)$, discriminant of $\hat{\mathbf{x}}^h$ in contrast to $\hat{\mathbf{x}}^i$.
$\mathrm{E}[\delta(g)]$	Average necessary-precision.
e_i	$\equiv -\frac{1}{2}\lVert\hat{\mathbf{x}}^i\rVert$
$e(\mathbf{y})$	VPR's exponent, number of leftmost bits eliminated.
$g^{i,h}$	$\equiv \tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)/2^{\lceil \log_2(\max(\lvert\tilde{g}()\rvert))\rceil}$, $\tilde{g}()$ power of two normalized.
g	$\equiv g^{1,2}$ in BVPC, any $g^{i,h}$ in MVPC.
$G_j^{i,h}(\mathbf{x})$	increases $g^{i,h}(\mathbf{x})$ precision from $j-1$ to j bits.
$G_j(\mathbf{x}, \hat{\mathbf{x}})$	increments $g(\mathbf{x})$ precision from $j-1$ to j bits.
K	vector dimension.
M, N	number of codevectors (e.g., M-ary classification).
m, n	operand precision.
n'	maximum function precision ($\equiv n - \underline{\delta}$).
$max[j]$	in M-ary classification, maximum with j-bit precision.
$p_n(g)$	probability mass function of g represented with n' bits.
$p(g)$	probability density function of g, $\equiv \lim_{m\to\infty} p_m(g)2^m$.
Q	$\equiv \max_{\lvert g\rvert<2^{-n}} p(g)$
$s(\mathbf{x}, \hat{\mathbf{x}})$	similarity measure, typically $s(\mathbf{x}, \hat{\mathbf{x}}) = -d(\mathbf{x}, \hat{\mathbf{x}})$.
U	*U*ndetermined sign.
$w[j]$	$\equiv 2^j d(\mathbf{x}, \hat{\mathbf{x}}^1, \hat{\mathbf{x}}^2)$ in BVPC, in MVPC $\equiv 2^j d(\mathbf{x}, \hat{\mathbf{x}})$.
$\mathbf{x} = \{x_1, ..., x_K\}$	source vector.
$\hat{\mathbf{x}} = \{\hat{x}_1, ..., \hat{x}_K\}$	centroid codevector, the replacement vector.
$\mathbf{y} = \{y_1, ..., y_K\}$	hyperplane codevector.
$y_{k,j}^i$	j-th significant bit (i.e., weight of 2^{-j}), $0 \le j < n$; k-th element, $1 \le k \le K$; i-th vector, $1 \le i \le M$.
$\delta_n(g)$	upper bound on necessary-precision, where g is evaluated using n-bit operands.
$\delta(g)$	$\equiv \lim_{n\to\infty} \delta_n(g)$
δ	precision necessary for an instance of classification.
$\hat{\delta}$	precision sufficient for an instance of classification.
$\underline{\delta}$	minimum necessary-precision ($\underline{\delta} \le \delta \le \delta(g) \le n$).
$(\underline{\varepsilon}[j], \overline{\varepsilon}[j])$	bounds for $s(\mathbf{x}, \hat{\mathbf{x}}^c) - max[j]$.
$(\underline{\epsilon}[j], \overline{\epsilon}[j])$	bounds for $d(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^l) - w[j]$ in BVPC, and for $s(\mathbf{x}, \hat{\mathbf{x}}^c) - w[j]$ in MVPC.

ACRONYMS

AT	$Area \times Time$ complexity measure
AT^2	$Area \times Time^2$ complexity measure
BVPC	Binary VPC
CIF	Common Intermediate Format
JPEG	Joint Picture Experts Group
IP	Inner Product
LSB	Least Significant Bit
LSD	Least Significant Digit
MPEG	Moving Picture Experts Group
MAD	Mean Absolute Difference
MSB	Most Significant Bit
MSD	Most Significant Digit
MSE	Mean Square Error
PTSVQ	Pruned-Tree Structured VQ
MVPC	M-ary VPC
SB	Signed Bit $\in$ {-1, 0, 1} redundant digit set
VPR	Variable-Precision Representation
VPC	Variable-Precision Classification
VQ	Vector Quantization

Acknowledgments

I would like to thank the committee chairman Miloš D. Ercegovac for his general assistance, helpful hints, and careful revision of the manuscript. I would like to acknowledge the useful conversations I had with Joseph L. White and John C. Darragh. I also would like to thank for the encouragement of David A. Paige, Richard L. Baker, and the committee members Andrew B. Kahng, Lawrence McNamee, Bruce Ho, and Kung Yao. I also would like to note my mathematics teachers in Venice high school, Manookian high school, Alborz junior high school, and Gulbenkian elementary school, and my first mentor, Varouj, who motivated me toward serious self study. This work was supported in part by UCLA's Mangasarian scholarship fund. It was further supported by NSF Grant MIP-8813340, Composite Operations Using On-Line Arithmetic for Application-Specific Parallel Architecture: Algorithms, Design and Experimental Studies.

Vita

1987	B.Sc. in Engineering, (summa cum laude) University of California, Los Angeles
1989	M.S. in Electrical Engineering, University of California, Los Angeles

Publications

R. Dionysian and M. D. Ercegovac,
Variable-Precision Linear Classifier, VLSI Signal Processing IV, IEEE Press, pp.285-294, 1990.

R. Dionysian and M. D. Ercegovac,
Variable Precision Representation for Efficient Codebook Storage, Proceedings of IEEE Data Compression Conference, pp. 319-328, 1992.

A. Madisetti and R. Jain and R. L. Baker and R. Dionysian,
Architectures and Integrated Circuits for Real Time Vector Quantization of Images, IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 5, pp.677-681, 1992.

Abstract of the Dissertation

Variable-Precision Arithmetic For Vector Quantization

by

Raffi Dionysian
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1994
Professor Miloš D. Ercegovac, Chair

This research proposes and investigates a method for the storage and computation in Vector Quantization (VQ) – a promising technique for image/speech compression. The improvement is in the representation and arithmetic algorithm; the idea is independent of the technology and accommodates different search algorithms. Specifically, with simple lossless compression, the codebook storage in tree searched VQ is reduced more than 20%. For large codebooks, the simulations predict that the compression would be more than 40%. The compression of codevectors is achieved with Variable-Precision Representation (VPR), where we eliminate the sign extension bits. By categorizing vectors, VPR uses non-stationary nature of codevectors. Entropy measure shows that VPR compresses at least 75% as well as Huffman coding of vector elements.

In conjunction with VPR, the VQ computation complexity is reduced using Variable-Precision Classification (VPC) method. VPC evaluates bit-serially beginning with the most significant bit the codevectors which are in VPR format. When the magnitude of the error due to the unevaluated bits is less than the magnitude of the evaluated discriminant, we can classify without processing the remaining bits. We show that as operand precision increases average necessary-precision becomes asymptotically independent of the operand precision. VPC makes the complexity of L_2 norm equivalent to L_1 norm. In ensuing VQ of real images, on average, the codevector element's precision necessary for classification was less than four bits.

We implemented circuitry for binary classification based on VPR and fixed-precision representations. Both designs were implemented in LSIlogic 1.0-μ gate array, to search binary non-balanced trees and encode MPEGII format video (720×576 pixels at 30 frames per second). The comparison showed that VPR lossless compression has negative decoding cost. Moreover, when the overall execution time is important, VPC halves computation as

measured either by AT or AT^2 complexity measures.

CHAPTER 1

Introduction

We intend to investigate computer arithmetic as means of improving performance and cost of Vector Quantization (VQ) [NK88, MRG85]. Vector quantization is a recent development in signal estimation and detection. It is currently under investigation for image compression, where it outperforms the current standard by Joint Picture Experts Group (JPEG). There is also a strong interest in dedicated hardware for VQ. Several integrated circuits have been implemented specifically for VQ [DG86, FCS90, KYJ93, DJK92, DB87].

Problem: Vector Quantization

Vector quantization finds in a set of codevectors the vector most similar to the source vector, a sequence of digitized signal samples. Applications other than compression also require such a dedicated engine: VQ has been used in segmenting images [GBJM79], recognizing speaker and speech [Kav86], and recognizing alphanumeric characters [NJ85]. Many aspects of the VQ computation have been investigated in the literature. The intent of these investigations has been primarily to decrease computation time. Few have also considered the storage aspect. Dezhgosha and *et. al.* have investigated compressing codebook using conventional compression techniques [DJK92]. Our research differs in that it examines arithmetic to decrease the complexity of VQ and improve its implementation. We investigate both storage and computation aspect. Moreover, since the gains are in arithmetic of classification, our approach can be applied in variety of algorithms using classification.

We investigate VQ with inner product as the metric for real time compression of video [Gra84, MRG85]. Using the arithmetic techniques developed, we implement Pruned-Tree Structured VQ (PTSVQ). PTSVQ is the scheme with the most potential to search large codebooks. PTSVQ uses a tree structure to choose from M codevectors a replacement codevector with only $O(\log(M))$ classifications.

There is a trend toward higher precision arithmetic which is typically achieved at a corresponding increase in hardware cost. This has precluded it from high rate signal processing required for video. In classification, we believe we can attain the benefits of the high-precision without an increase in cost. Conventionally, classification is performed in two steps. First, a suitable metric measures the distance between the source vector and each codevector. Second, comparison of the distances determines the codevector closest to the source vector.

We will analyze these two steps as one single composite module. Our key insight is that in binary classification, regardless of the operand precision the output is only one bit.

Approach: Variable-Precision Arithmetic

Variable-precision arithmetic embodies two aspects. First, Variable-Precision Representation (VPR), which by eliminating sign extension bits, reduces storage and computation. In VPR, we are motivated by the reduction in dynamic range when subtracting similar codevectors. For each vector, a range indicator can store the number of Most Significant Bits (MSB) which are zero for all elements. Second part is Variable-Precision Classification (VPC). VPC avoids evaluation of Least Significant Bits (LSB) to reduce computation. VPC is easily integrated with VPR. Given the range indicator, VPC skips the initial iterations of the bit-serial evaluation.

In conventional fixed-precision design, the choice of the precision for VQ evaluation involves a tradeoff between the engine's cost and its ability to choose the best codevector. In designing a conventional VQ engine, reducing precision reduces storage, memory-processor bandwidth and computation. On the other hand, in more cases the precision will not be sufficient to distinguish which of two codevectors is closer to the source vector. Lower precision causes more frequent arbitrary decisions and higher distortion in compression. Typically, the classification is performed at the quantization precision of the samples. This varies depending on the application. A pixel of image is typically quantized at 8 bits. In future, this may increase to 10 or 12 bits. The samples of speech are quantized at 12 bits, and compact disc quality music is quantized at 16 bits. VPC goes one step further. It adjusts the precision of the arithmetic to the instance of classification.

We are motivated by an inherent property of classification: *regardless of the operand precision, a classification reduces to binary decision(s).* Binary classification evaluates the distance between the input and two codevectors and outputs one bit which indicates the smaller distance. M-ary classification breaks down to $(M-1)$ binary classification, and is evaluated as a cascade of functions with binary outputs. In turn, VPC for every binary function questions: "At what precision does this instance need to be evaluated?" In general, VPC can be used in computations whose set of possible outcomes does not increase as the operand precision increases.

VPC is similar to on-line arithmetic [EL88]: operands are processed bit-serially from the most significant bit. As a composite operator, the metric is not evaluated as a sequence of "atomic" operations (e.g., multiplications, additions). Instead, we merge these operations and rewrite the algorithm in terms of a composite operator which incrementally processes the operands. An initial estimate is formed by processing the MSBs of the operands. VPC updates the estimate by bit-serially processing the operands. The processing *terminates* when the error from the unprocessed least significant bits cannot change the output's sign.

By avoiding processing of superfluous LSBs, on average, the classification time is *reduced.* We have found out that in VPC the average evaluation precision is a constant.

Thesis Outline

- The next chapter provides an overview of vector quantization (VQ), various fast search techniques, and previous VQ implementations.
- Chapter 3 describes lossless compression of VQ codebooks using VPR. In compressing codebooks used in encoding real images, we measure the reduction in storage possible by VPR. We then analyze this and compare it to the entropy of codevectors.
- Chapter 4 develops the VPR architecture. We design a conventional architecture for comparison. Both modules are implemented using gate array technology to measure the clock period and show VPR's negative decoding cost.
- Chapter 5 describes VPC algorithm. It also proves that the average necessary-precision is asymptotically constant and independent of operand precision.
- Chapter 6 evaluates VPC for PTSVQ with the inner product measure. The average precision for classification is characterized and measured for real images. Finally, the VPC architecture is presented, and its cost and performance estimated.
- Chapter 7 summarizes the contributions of the dissertation and discusses future work.
- Additionally, there are two appendices. Appendix A describes a VQ application. Appendix B documents implementation of both VPR and conventional architectures.

CHAPTER 2

Vector Quantization (VQ)

2.1 Introduction

Vector quantization is the statistically optimum method for lossy data compression. It can yield the lowest rate-distortion. The distortion due to the compression can be made imperceptible just like in JPEG, while its compression rate can out perform JPEG standard. An example of use is in compressing video: the images are broken into blocks of K pixels (*source vectors*), $\mathbf{x}$; the encoder compares each source vector with N potential replacements (*codevectors*) $\hat{\mathbf{x}}^c, c = 1, \cdots, N$, stored in a *codebook*; it transmits the index c of the best matching (least distorting) codevector to the receiver. In decompression, the index is used to approximate $\mathbf{x}$ using $\hat{\mathbf{x}}^c$ from an identical codebook. For best results, VQ relies on large set of patterns. Survey papers by Nasrabadi *et. al.* [NK88], and Makhoul *et. al.* [MRG85] discuss a variety of techniques for building VQ codebooks.

This chapter formulates and reviews vector quantization as classification using inner product. Section 2.2 formally describes the classification problem, and the inner product similarity measure. Next, Section 2.3 shows how other similarity measures can be mapped into the inner product similarity measure. Then, Section 2.4 presents a novel algebraic transformation which reduces the number of inner products in a classification by one. Section 2.5 follows by a simple example, a geometric interpretation, and a real example. Finally, Section 2.6 reviews investigations in reducing computation and storage, and Section 2.7 summarizes previous implementations.

2.2 Classification

N-ary classification of a vector $\mathbf{x}$ is

$$c = \max_{i=1,\ldots,N}{}^{-1}\left(s(\mathbf{x}, \mathbf{x}^i)\right) \tag{2.1}$$

where $s(\mathbf{x}, \hat{\mathbf{x}})$ *measures the similarity* between $\mathbf{x}$ and $\hat{\mathbf{x}}$. And $\max^{-1}()$ denotes the index i, with the highest similarity $s(\mathbf{x}, \hat{\mathbf{x}}^i)$. In other words, it selects out of N replacement vectors (codevectors), the codevector $\hat{\mathbf{x}}^c$ which is most similar to the source vector $\mathbf{x}$.

Often the classification is done in a *metric space*. That is, a K-dimensional vector space with a metric $d(\mathbf{x}, \hat{\mathbf{x}})$, where $d(\mathbf{x}, \hat{\mathbf{x}})$ measures the *distance* [BB78] between the two vectors $\mathbf{x}$

and $\hat{\mathbf{x}}$. The distance metric $d(\mathbf{x}, \hat{\mathbf{x}})$ is large for dissimilar vectors and low for similar vectors. Its properties are:

1. $d(\mathbf{x}, \mathbf{x}) = 0$
2. $0 < \ d(\mathbf{x}, \hat{\mathbf{x}}),\ \mathbf{x} \neq \hat{\mathbf{x}}$
3. $d(\mathbf{x}, \hat{\mathbf{x}})\ = d(\hat{\mathbf{x}}, \mathbf{x})$
4. $d(\mathbf{x}, \hat{\mathbf{x}})\ \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \hat{\mathbf{x}}),\ \forall\ \mathbf{y}$

In the metric space, N-ary classification becomes

$$c = \min_{i=1,\ldots,N}{}^{-1} \left(d(\mathbf{x}, \hat{\mathbf{x}}^i) \right) \tag{2.2}$$

where $\min^{-1}()$ denotes the index i, with the minimum distance $d(\mathbf{x}, \hat{\mathbf{x}}^i)$. In other words, it selects out of N replacement vectors (codevectors), the codevector $\hat{\mathbf{x}}^c$ which has the least distance from the source vector $\mathbf{x}$.

Equation (2.2) can be rewritten in the form of (2.1) by using $\max^{-1}()$ instead of $\min^{-1}()$ operator. Since the smallest in a set of distances is equivalent to the largest among the negated distances:

$$c = \max_{i=1,\ldots,N}{}^{-1} \left(-d(\mathbf{x}, \hat{\mathbf{x}}^i) \right) \tag{2.3}$$

If

$$s(\mathbf{x}, \hat{\mathbf{x}}^i) \equiv -d(\mathbf{x}, \hat{\mathbf{x}}^i)$$

then equation (2.3) is identical to (2.1).

We will discuss only similarity measures which correspond to a distance metric. However, in general, $s(\mathbf{x}, \hat{\mathbf{x}}) \neq -d(\mathbf{x}, \hat{\mathbf{x}})$. So, $s(\mathbf{x}, \hat{\mathbf{x}})$ is not constrained as the distance metric is. Such measures are mentioned in [DH73], and VPC may be extended to these measures as well.

Our analysis begins with binary classification (N=2). Equation (2.1) reduces to

$$c = \begin{cases} 2 & \text{if } s(\mathbf{x}, \hat{\mathbf{x}}^2) \geq s(\mathbf{x}, \hat{\mathbf{x}}^1) \\ 1 & \text{otherwise} \end{cases} \tag{2.4}$$

In other words between the two codevectors, choose the codevector with the highest similarity to the source vector. The comparison of two similarity measures shown in (2.4) is often denoted as a discriminant evaluation. Let's rewrite the above equation with the discriminant function. A discriminant measures the similarity of $\mathbf{x}$ to $\hat{\mathbf{x}}^h$ in contrast to $\hat{\mathbf{x}}^i$:

$$\begin{aligned} \tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h) &\triangleq s(\mathbf{x}, \hat{\mathbf{x}}^h) - s(\mathbf{x}, \hat{\mathbf{x}}^i) \\ &\equiv d(\mathbf{x}, \hat{\mathbf{x}}^i) - d(\mathbf{x}, \hat{\mathbf{x}}^h) \quad \text{when } s(\mathbf{x}, \hat{\mathbf{x}}) \equiv -d(\mathbf{x}, \hat{\mathbf{x}}) \end{aligned}$$

Rewriting the binary classification (2.4) using the discriminant function,

$$c = \begin{cases} 2 & \text{if } \tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^1, \hat{\mathbf{x}}^2) \geq 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.5}$$

In other words, binary classification evaluates a binary-valued function SIGN($\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^1, \hat{\mathbf{x}}^2)$). If the discriminant is negative then $\mathbf{x}$ maps into $\hat{\mathbf{x}}^1$, otherwise $\mathbf{x}$ maps into $\hat{\mathbf{x}}^2$. As we we will elaborate in Section 2.4, the discriminant function can be effectively used to classify among N codevectors.

2.3 Inner Product Similarity Measure

Many classifications problems can be reduced to detecting sign of an inner product. The most widely use application is in Mean Square Error (MSE) classification [Wha71, Gra84]. MSE can also be generalized to weighted MSE criteria [DH73]. Moreover, there is a less apparent application: Itakura-Saito measure [IS68] which approximates a waveform with the output of a recursive linear filter. All of these three applications have been implemented with inner product similarity measure:

$$sign(\mathbf{x}^t \mathbf{y} + b) \quad = \quad sign(\sum_{i=1}^{K} x_i y_i + b)$$

The inner product of the point's coordinates with the plane's normal determines which side of a plane a point lies on. As shown next, minimizing mean square error is geometrically akin to classifying between two points separated by a plane.

Mean Square Error

Mean Square Error (MSE) [Wha71, Gra84] while theoretically tractable is simple to evaluate. For K-dimensional vectors $\mathbf{x}$ and $\hat{\mathbf{x}}$ MSE classifier is

$$c = \min_{i=1,\cdots,N}{}^{-1} \left((\mathbf{x} - \hat{\mathbf{x}}^i)^t (\mathbf{x} - \hat{\mathbf{x}}^i) \right) \tag{2.6}$$

where $\min^{-1}()$ denotes the index, c, for which the minimum Euclidean distance $||\mathbf{x} - \hat{\mathbf{x}}^c||$ is achieved. In other words, it finds the index c of the codevectors $\hat{\mathbf{x}}^c$ out of N codevectors which has the least squared error. This minimization can be carried out as a sequence of $(N-1)$ discriminant evaluations.

MSE is attractive for two reasons. First, a small distance implies small observable difference between vectors, where the two vectors are observed in the same way they were sampled.

Second, during the generation of codevectors (training), it provides a way to cluster source vectors into codevectors (replacement vectors).

We can evaluate an inner product to find the codevector which yields the least MSE, as is done in [Wha71]. Expanding (2.6), and dividing it by two:

$$c = \min_{i=1,\cdots,N}{}^{-1}\left(\frac{1}{2}||\mathbf{x}|| - \mathbf{x}^t\hat{\mathbf{x}}^i - (-\frac{1}{2}||\hat{\mathbf{x}}^i||)\right) \tag{2.7}$$

The first summation is independent of i and does not affect the minimization. It is ignored. The last summation

$$e_i \triangleq -\frac{1}{2}||\hat{\mathbf{x}}^i||^2 \tag{2.8}$$

depends only on the vector $\hat{\mathbf{x}}$. It is precomputed and stored with the codevector. Equation (2.7) becomes an inner product $\mathbf{x}^t\hat{\mathbf{x}}^i$ summed along with a constant:

$$c = \min_{i=1,\cdots,N}{}^{-1}\left(-\mathbf{x}^t\hat{\mathbf{x}}^i - e_i\right) \tag{2.9}$$

Equation (2.9) can be rewritten:

$$c = \max_{i=1,\cdots,N}{}^{-1}\left(\mathbf{x}^t\hat{\mathbf{x}}^i + e_i\right) \tag{2.10}$$

because, the smallest in a set of numbers is equivalent to the largest in the negated set of numbers.

We must note that arithmetically Equation (2.10) may not have a lower complexity than Equation (2.6). Inner product uses multiplication instead of subtraction and squaring. Squaring even with the subtraction beforehand would have lower theoretical complexity than multiplication. Additionally, the inner product metric requires storage and addition of e_i. However, as we will see later Chapter 3, the inner product evaluation can be more amenable to processing codevectors compressed with VPR. Also for classifying with small N, an algebraic transformation shown later in this chapter, makes (2.10) more attractive than (2.6).

Weighted Mean Square Error

Weighted MSE [DH73], a general form of MSE, is also used in classification. With N patterns,

$$c = \min_{i=1,\cdots,N}{}^{-1}\left((\mathbf{x} - \hat{\mathbf{x}}^i)^t\mathbf{W}(\mathbf{x} - \hat{\mathbf{x}}^i)\right) \tag{2.11}$$

where $\mathbf{x}$ and $\hat{\mathbf{x}}$ are K-dimensional column vectors and $\mathbf{W}$ is a positive definite $K \times K$ matrix. WMSE can, for example, allow some elements to be weighted more heavily than others in classification of the vector. When $\mathbf{W}$ is positive definite *and* also symmetric, (2.11) evaluates Mahalanobis distance [DH73].

In image compression, a weighted distortion attains a better subjective image quality [MS74], since the human visual system's sensitivity varies for different spatial frequencies. Until recently however, weighted MSE was not used since it requires more computation than MSE.

Davidson in [DCG88] has shown the weighted MSE metric can also be evaluated as an inner product. They expanded (2.11), and divided it by two:

$$c = \min_{i=1,\cdots,N}{}^{-1} \left(\frac{1}{2}\mathbf{x}^t\mathbf{W}\mathbf{x} - \mathbf{x}^t\mathbf{W}\hat{\mathbf{x}}^i + \frac{1}{2}\hat{\mathbf{x}}^{i^t}\mathbf{W}\hat{\mathbf{x}}^i \right)$$

If the weight is independent of codevector $\hat{\mathbf{x}}^i$, weighted MSE can be evaluated by the same architecture which evaluates the MSE metric. To do so, the following are precomputed and stored:

$$\mathbf{y}^i = \mathbf{W}\hat{\mathbf{x}}^i, \quad \text{and} \quad e_i = \frac{-||\mathbf{W}\hat{\mathbf{x}}^i||}{2}$$

so that

$$c = \max_{i=1,\cdots,N}{}^{-1} \left(\mathbf{x}^t\mathbf{y}^i + e_i \right) \tag{2.12}$$

Although the above equation is identical to its counterpart (2.9) in the previous section, the codevector precision is different. For m-bit unsigned weights, the codevector element's precision increases from n to $(n+m)$. VPC, however, will be shown to deal with the increased codevector precision without modification, as it can evaluate the codevector at any precision.

Linear Prediction's Similarity Measure

For completeness we will also discuss Itakura-Saito measure [MRG85, IS68]. It is used, when compressing speech with Linear Predictive Coding (LPC). In LPC, an all pole linear predictor estimates a segment of speech. The predictor is given as input a sinusoid whose pitch is extracted from the speech segment. The predictor's coefficients are then chosen to minimize the difference between the output and the speech samples. In VQ of the filter coefficients, we would like a vector of coefficients which minimizes MSE between the speech and the linear prediction. To do so, Itakura-Saito measure would maximize the similarity of the autocorrelation between the speech and the predictor's coefficients:

$$s(\mathbf{x}, \mathbf{y}) = R_{\mathbf{xx},0}R_{\mathbf{yy},0} + 2\sum_{k=1}^{K} R_{\mathbf{xx},k}R_{\mathbf{yy},k} \tag{2.13}$$

where $R_{\mathbf{xx}}$ is the autocorrelation of the speech samples, and $R_{\mathbf{yy}}$ is the autocorrelation of the filter coefficients.

2.4 Algebraic Transformation

The maximization in (2.10) requires N evaluations and $(N-1)$ comparisons. There is a novel way to reduce the the number of inner products by one. We subtract the first term in the maximization from the remaining $(N-1)$ terms. This will bias each term by the same amount and, therefore, it will not change the index of the maximum measure. This transformation is useful with the inner product metric because the first term is implicitly evaluated within the remaining terms. Equation (2.10) reduces to $(N-1)$ evaluations and $(N-1)$ comparisons,

$$c = \max_{i=2,\cdots,N}{}^{-1}\left(0, \mathbf{x}^t(\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^1) + (e_i - e_1)\right) \tag{2.14}$$

After precomputing $(\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^1)$ and $(e_i - e_1)$, the above equation reduces to

$$c = \max_{i=2,\cdots,N}{}^{-1}\left(0, \mathbf{x}^t\mathbf{y}^{(i)} + b_i\right) \tag{2.15}$$

The reduction is significant for small N. In binary tree searched VQ ($N = 2$) this halves the number of inner products. Exact evaluation of an element of $\mathbf{y}^i (= \hat{\mathbf{x}}^i - \hat{\mathbf{x}}^1)$, on the other hand, does require an additional bit of precision for storage and evaluation.

2.5 Classification Example

We will first show a "paper and pencil" example of VQ. We will use matrix-vector multiplications and vector additions. The codebook matrix is stack of vectors $\mathbf{y}$, where $\mathbf{y}^i = \hat{\mathbf{x}}^i - \hat{\mathbf{x}}^i$ represents the replacement vector $\hat{\mathbf{x}}^i$. A set of vectors $\mathbf{y}$ form the rows of the codebook matrix. Inner products similarity measure evaluations are grouped into matrix-vector multiplications where the codebook is multiplied by the source vector $\mathbf{x}$. The resulting vector has the measure of similarity between the source vector and the codevectors. Classification is then finding the biggest entry in the resulting vector.

In following example, the vectors are four dimensional ($K = 4$):

$$\begin{aligned}
\mathbf{x} &= (-5, -4, 4, 5)^t, & & \\
\hat{\mathbf{x}}^1 &= (0, 0, 0, 0)^t, & e_1 &= 0, \\
\hat{\mathbf{x}}^2 &= (-100, -10, 10, 100)^t, & e_2 &= -10100, \\
\hat{\mathbf{x}}^3 &= (-10, -10, 10, 10)^t, & e_3 &= -200, \\
\hat{\mathbf{x}}^4 &= (-5, -3, 3, 5)^t, & e_4 &= -34, \\
\hat{\mathbf{x}}^5 &= (0, 10, 10, 0)^t, & e_5 &= -100,
\end{aligned}$$

where e_i is defined by (2.8). Subtracting the first codevector from the rest,

$$\begin{aligned}
\mathbf{y}^2 &= (-100, -10, 10, 100)^t, & b_2 &= -10100, \\
\mathbf{y}^3 &= (-10, -10, 10, 10)^t, & b_3 &= -200, \\
\mathbf{y}^4 &= (-5, -3, 3, 5)^t, & b_4 &= -34, \\
\mathbf{y}^5 &= (0, 10, 10, 0)^t, & b_5 &= -100.
\end{aligned}$$

First, we evaluate the similarity measures in the maximization (2.15):

$$\begin{pmatrix} -9020 \\ -20 \\ 40 \\ -100 \end{pmatrix} = \begin{pmatrix} -100, & -10, & 10, & 100 \\ -10, & -10, & 10, & 10 \\ -5, & -1, & 1, & 5 \\ 0, & 10, & 10, & 0 \end{pmatrix} \times \begin{pmatrix} -5 \\ -4 \\ 4 \\ 5 \end{pmatrix} + \begin{pmatrix} -10100 \\ -200 \\ -34 \\ -100 \end{pmatrix}$$

Then, we find the index of the best entry c:

$$\begin{aligned}
c &= \max_{i=1,\cdots,N}{}^{-1} (0, -9020, -20, 24, -100) \\
&= 4
\end{aligned}$$

Geometric Interpretation

Binary classification using inner product metric partitions the space with a plane. Vectors on one side of the partition are quantized into the point 1 ($c = 1$), and on the other side into the point 2 ($c = 2$). As shown in Figure 2.1 in a plane, classification partitions with a straight line. In physical space, it partitions with a plane. In general, in a K-dimensional space, it partitions with a $(K-1)$-dimensional hyperplane.

Classification with an inner product is also referred to as a hyperplane test. It determines which side of the hyperplane the source vector is on. It performs an inner product between the source vector and the vector normal to the plane (i.e., codevector), and it adds a constant. The constant, represented by bias b is the distance of the hyperplane from origin.

In N-ary classification, the space is divided into N regions or classes. Between every two classes lies a hyperplane. Combinatorially, there are N choose 2 ($= \frac{N(N-1)}{2}$) hyperplanes. Contrary to a popular misconception fortunately, far fewer hyperplane tests suffice. Simply put, we need only $(N-1)$ tests. Each test eliminates one class, and after $(N-1)$ tests, only the best class is left.

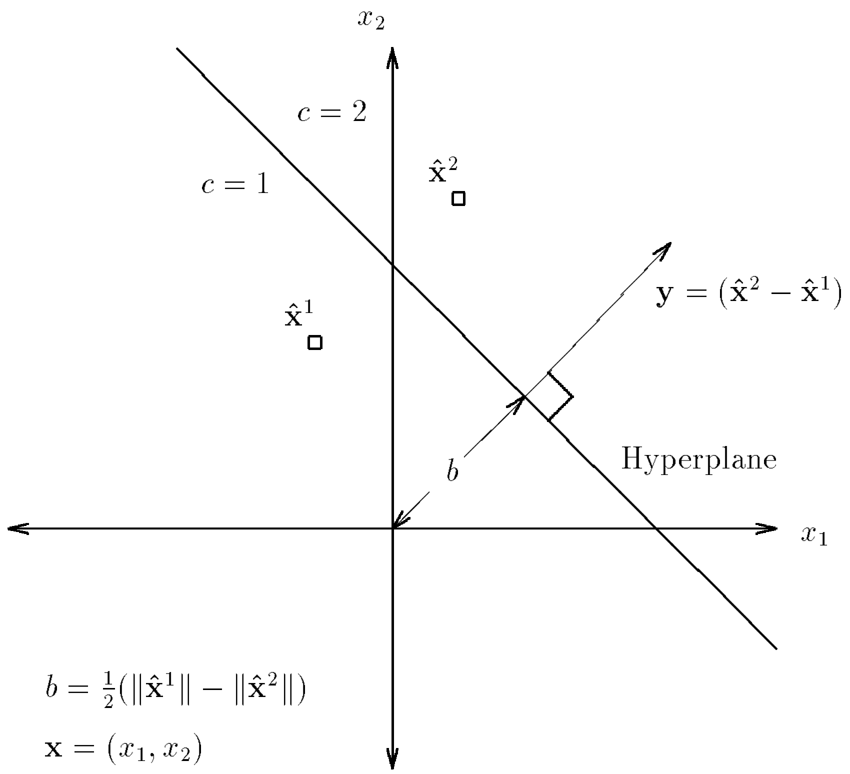


Figure 2.1: The inner product metric partitioning the plane with a straight line.

VQ Example

Now, we show an application of VQ. For image compression, we encoded with mean-residual VQ [Bak84] which by removing the mean does a simple form of preprocessing. The codebook was generated using k-means clustering [LBG80]. The data set used for training was 4×4 blocks (vector dimension $K = 16$) from 13 256×256 pixel images. The codebook which had about 2000 codevectors was organized into a binary-tree shown in Figure 2.7. The tree-structure allowed finding replacement codevector in less than twelve classifications.

We will show the inputs and output data set pictorially. Figure 2.2(a) shows a halftoned image of *Lena*. A popular picture for testing image processing algorithms, it is an 256×256 pixel image, whose grey levels were halftoned for display purposes. Figure 2.3 is the residual image after mean of each 4x4 blocks was removed. Subtracting the mean causes some pixels to become negative. For display purposes the pixels were biased up, such that the most negative value was displayed as pitch black and zero value was displayed with a gray value. Also since preprocessing has removed most of the image energy, the residual was magnified four times ($\times$ 4). As can be seen, even after the magnification most pixels still have low intensity. This implies that their most significant bits are zero. This was an intuitive motivation for variable-precision representation of the vectors. Finally, we examine the result of vector quantization. Figure 2.2(b) shows the coded-decoded residual image with the mean added. Some of the distortion due to coding is apparent even on the halftoned image. With more sophisticated preprocessing methods, this distortion would become imperceptible without decreasing the compression rate.

For a better illustration, we look at Lena's eye and a magnified section of the codebook. Figure 2.4(a) shows Lena's eye. Figure 2.5 shows a page from the codebook used for compressing the image. It has 64 codevectors, where each codevector is arranged as 4×4 pixels. The last picture, Figure 2.4(b) is the coded-decoded image, where codevectors have replaced 4×4 block of pixels. Again we may note the block artifacts inherent in all compression techniques which compress one block of the image at a time.

2.6 Fast Search Methods

There has been a variety of approaches to decrease the computation time of VQ. In VQ, a full codebook search would inspect every codevector for the best matching one. Such an exhaustive search of codebooks is similar to the operation of a content associative memory. It has to perform $O(\ KN)$ operations per source vector. Number of entries is large: N typically ranges from 2^8 to 2^{20}. It was shown both experimentally and theoretically, that increasing N and K increases the performance. Correspondingly, techniques are developed which reduce the amount of computation. They can be categorized as follows:

(a)

(b)

Figure 2.2: (a) Lena, (b) Lena coded at one bit per pixel.

Figure 2.3: Lena's residual.

1. The distance evaluation and comparing against the maximum can be done in tandem. The evaluation of the metric then can be stopped when the metric is greater than the previously computed distance [BG85]. In some ways this is similar to VPC. It, however, cannot be used with similarity measures.

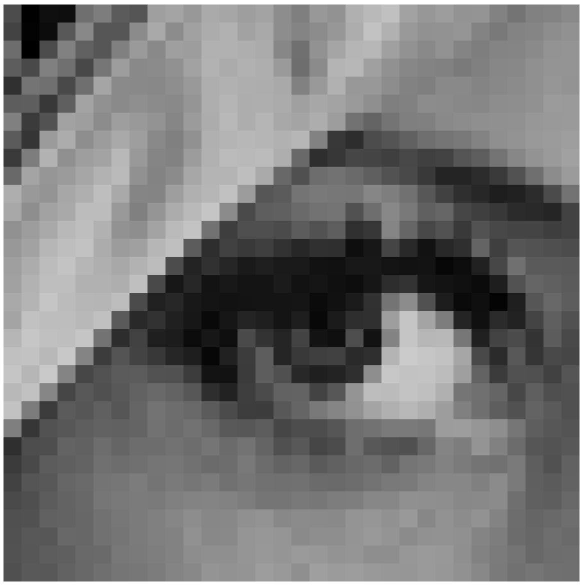
2. Techniques similar to hashing can be used for accessing the codevector with less computation. Similar to hashing each codevector has a precomputed tag. Unlike hashing using these tags, we can exclude large number of codevectors from the ensuing exhaustive search. Although they reduce computation, these techniques still require $O(KN)$ operations. These tags also require additional storage.

3. Techniques which are variations of Tree-Structured VQ (TSVQ) [Gra84, BGGM80] are most attractive. They reduce the computational complexity to $O(K\log(N))$ operations per search.

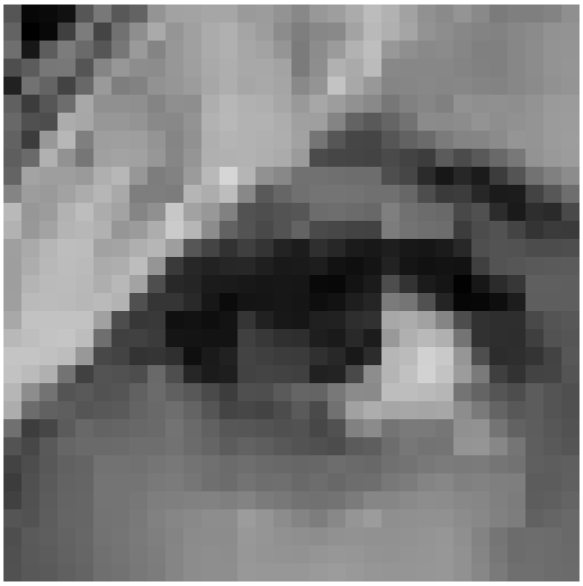
Partial Distance

Bei and Gray [BG85] showed an optimization applicable in sequential evaluation of the distance metric. They would stop the evaluation as soon as the partial distance became greater than the minimum. As shown in Figure 2.6, the evaluation of metric and comparison are effectively merged together into one composite computation.

The partial evaluation is applicable for any distance metric. That is, for any measure where every term of the summation is non-negative. During the metric evaluation, if a par-



(a)



(b)

Figure 2.4: (a)Lena's eye, (b) Eye coded at one bit per pixel.

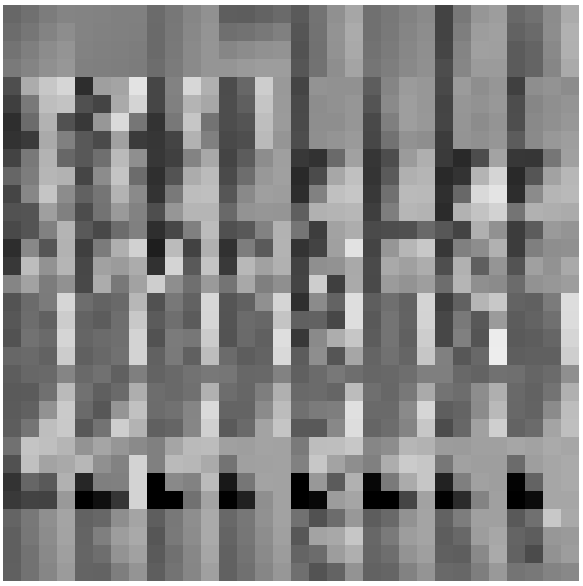


Figure 2.5: A page of codebook.

tially evaluated metric is greater than the minimum, the evaluation can output the outcome early. Bei and Gray [BG85] evaluated the performance in classification of auto-regressive sequences and real images. For binary classification, the improvement was more than 25%. Since the first distance is always evaluated, the improvement would be less than 50%. A more significant drawback is the inability to apply this idea to similarity measures such as inner product, where each term can be either positive or negative. This makes this technique unattractive in binary classification, where the two distance evaluations can be replaced by one inner product similarity measure.

$partial\_distance[0] = 0$;
for $i = 1, \cdots, K$
    if $partial\_distance[i] > min$ then
        terminate;
    else
        $partial\_distance[i] = partial\_distance[i-1] + (x_i - y_i)^2$;
if $partial\_distance[i] < min$ then
    $min = partial\_distance[i]$;

Figure 2.6: Partial evaluation of metric.

Excluding Codevectors

For completeness, we present methods for eliminating codevectors without metric evaluations. These techniques require additional storage for the associated index tables. We are not aware of custom hardware for any of these techniques. The following is a brief summary.

We can index the codevectors using some of the elements of codevector [SM87]. Based on corresponding source vector's element, we can then limit the search to codevectors within a hypercube around the source vector. An extension of this idea is K-d trees [Ben75]. K-d tree is a tree-structure which for branching uses hyperplanes which are normal to one of the K axis. Since all but one of the elements of the corresponding codevector are zero, the inner products reduce to element-by-element comparisons. K-d tree VQ has a performance less than TSVQ, since the orientation of its hyperplanes at the branch nodes is restricted to be normal to one of the principle axis.

We can bound the search area with a hypersphere instead of a hypercube. In such an approach, codevectors are ordered and listed based on their norm. Then depending on source vector's norm, we index into the list and limit the search to a spherical shell containing codevectors with similar distances from origin [Mad90]. A more sophisticated variant of this method uses the first singular value of the codevectors instead of the vector norm[AC87] for indexing. It can offer better performance, although it requires some form of singular value decomposition.

Tree-Structured Codebook

Tree-structured VQ (TSVQ) [Gra84, BGGM80] is fast. Using the tree-structured codebook decreases the computational complexity of VQ from $O(KN)$ to $O(K\log(N))$. As shown in Figure 2.7, a tree-structure permits finding the best codevector with a series of classifications. Each classification chooses the best branch out of a set of branches. Trees with binary branches result in the least number of classifications. In binary TSVQ moreover, two MSE evaluations can be replaced with one hyperplane test. This halves the computation and storage.

TSVQ has higher distortion than exhaustive VQ. After all to get the tree-structure, the codevectors are clustered together to form the parent codevector, approximating piecewise linear partition with one linear partition. This results in non-optimal codevector selection. Experiments reveal that tree structured codebooks of size N designed by the generalized Lloyd algorithm (LBG) typically yield a distortion-rate performance comparable to full search codebooks of size $N/2$, or alternatively one dB more noise in data compression [Bak84].

Trees with 4, 8, 16, or in general M classifications at every node would make the performance of the tree-structured search comparable to the exhaustive search. The generalization is simple. Let $\{\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \cdots, \hat{\mathbf{x}}_M\}$ be the set of all children of the current node. At each branch-

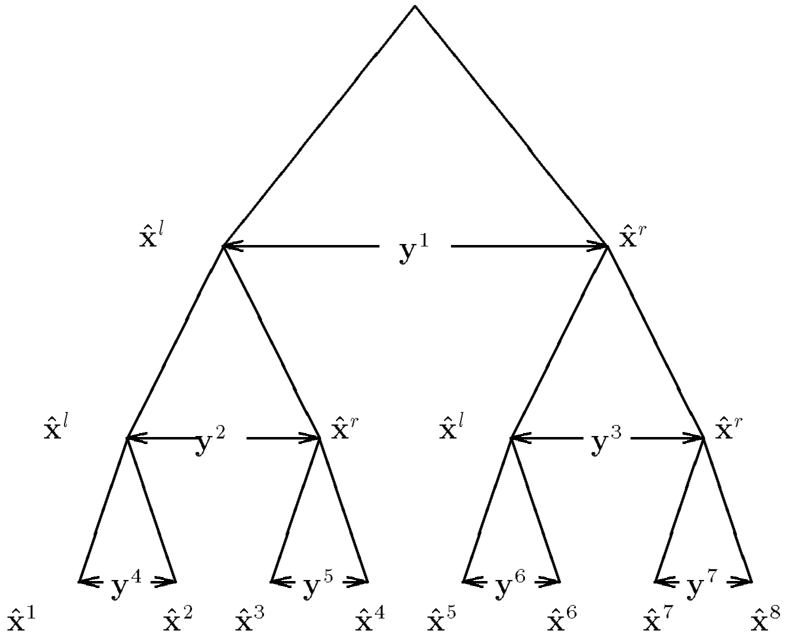


Figure 2.7: A codebook structured as a binary tree.

ing node, we then descend to node $\hat{\mathbf{x}}^c$ which is closest to the source vector $\mathbf{x}$.

More recently Chou *et. al.* [CLG88] developed a pruned tree search algorithm which greatly improves the distortion rate performance of TSVQ. Kiang *et.al.* [KBSC92] improve the algorithm by allowing a finer degrees of tree pruning. The key idea is as follows. Beginning with a binary tree structured codebook of rate 2R, the algorithm prunes away least frequently used subtrees. While increasing the distortion, eliminating the sub-tree decreases the rate. This is continued until the desired average rate R is obtained. Entropy Pruned Tree-Structured VQ (PTSVQ) [CLG88], reduces the average codevector index length by removing the infrequently used codevectors. By allowing control on the depth in searching the tree, PTSVQ permits the user to vary the compression rate. A drawback of PTSVQ is its variable index rate and variable search time. On the positive side, the variable search time conveniently masks the variable execution time of VPC. TSVQ has also been extended to multi-path TSVQ [CCW91], which searches for the four best candidates in the tree. Overall, PTSVQ seems the most viable fast search technique.

2.7 Previous Implementations

Table 2.1 summarizes previous VQ implementations. Most of the early VQ implementation were for speech compression. Since speech has orders of magnitude lower data rate than video, a VQ encoder is easier to realize. The implementation's precision depends largely

Table 2.1: Summary of previous implementations.

Year	First Author [Reference]	Precision (bits)	Measure	Operator	Algorithm
1984	Tao [TAG84]	8	MSE	parallel	VQ
1986	Davidson [DG86]	12	MSE	parallel	VQ
1986	Nelson [NR86]	9	MSE	serial	VQ
1987	Abut [ATS87]	8	MSE	parallel	VQ
1987	Dionysian[DB87]	7	IP	parallel	VQ
1988	Davidson[DCG88]	12	IP	parallel	VQ
1989	Ramamoorthy [RPT89]	9	MSE	serial	VQ
1990	Fang [FCS90]	8	IP	parallel	TSVQ
1992	Dezhgosha [DJK92]	9	MAD	parallel	VQ
1993	Kolagotla [KYJ93]	8	IP	parallel	TSVQ

on the application. Image pixels used in VQ compression are 8 bits. Depending on the preprocessing, its precision could increase to 9 or 10 bits. Meanwhile, speech samples are 12 bits.

Several chips directly evaluate sum of square of difference distance. A VLSI implementation by Davidson [DG86] has parallel subtractors, squaring circuits, and adders. It operates on 12-bit operands. There are similar VQ architectures which use 8-bit operands [ATS87, TAG84]. Architectures which have bit-serial subtract, square and addition operators have also been designed [NR86, RPT89]. These use 9-bit operands.

More recent chips evaluate Inner Product (IP). In binary hyperplane test, one IP similarity measure replaces evaluation of two MSE distance measures. Inner product has only two type of operators: multiply and accumulate, which makes it easier to implement than subtract-square-add circuitry needed for distance evaluation. A Systolic architecture proposed by Davidson [DCG88] uses parallel multipliers and adders with 12-bit operands. Variations of this Systolic architecture can trade between the number of multiplier-adder units used and throughput. It can also fully search the codebook or search through balanced trees. Another chip [KYJ93] uses the same architecture with intent to cascade it with a linear predictor to form Predictive VQ.

A recent chip by Fang *et.al.* [FCS90] can search 8-level balanced trees. It contains eight inner product processor and eight adjoining codebooks. Each processor, is fed 8-bit operands, has a multiplier, an accumulator and a comparator. Each adjoining codebook stores hyperplanes for a given level of the tree.

Another recent chip designed by Dezhgosha *et.al.* [DJK92], uses Mean Absolute Distance (MAD) for the classification. Built using subtractors and adders, it searches exhaustively. Also, the pixels and the codevectors are non-uniformly quantized at 5-bit to reduce codebook storage.

We, Dianysian (the prior name of Dionysian :-) and Baker, designed a chip [DB87] at UCLA to evaluate the inner product metric, with 7-bit operands. Each pipelined chip contained five parallel multipliers and adders with each multiply-add unit forming one pipeline stage. In addition it had registers to store five source vectors. The chips would have been cascaded to permit processing of vectors with up to 64 elements.

2.8 Summary

Vector quantization (VQ) approximates a source vector by a codevector, representing the image by codevector indices instead of pixels. Many VQ techniques are derived from the classification field, and the key computation is identical. We summarized the previous investigations in reducing computation, and highlighted the one we believe is the most promising one: tree structured VQ (TSVQ). TSVQ reduces the computational complexity logarithmically. M-ary trees allow a tradeoff between better codevector selection and quicker search time. We developed a generalization of binary hyperplane tests to M-ary hyperplane tests. This novel technique permits reducing the number of inner products in a classification by one. Finally, we tabulated the previous hardware designs. A recently published design incorporated codebook compression. Another design, a VLSI implementation used a large portion of the chip for codebook storage. These highlight codevector storage as an important issue.

CHAPTER 3

Variable-Precision Representation (VPR)

3.1 Introduction

In vector quantization (VQ) as the codebook size increases, the noise in the compression decreases. This has been shown both theoretically and experimentally. High performance VQ systems have become viable with the rapid progress in memory technology. Successful proprietary video transmission systems using VQ have been built [Mok89]. VQ is also a candidate for the next generation of image/speech compression standards. In this chapter, we present a scheme for reducing the codebook storage needed in vector quantization. With storage reduction, we can fit larger codebooks in a given amount of memory. This can improve the cost-performance of the system.

The memory structure of VQ depends on the search technique employed. In real-time VQ of video and speech with large codebooks, exhaustive search of the replacement vector $\hat{\mathbf{x}}^c$ is computationally prohibitive. A variety of codebook structures [NK88], [MRG85] have been devised to reduce the search time, sometimes at cost of additional storage. A popular one is Tree Structured VQ (TSVQ) which uses a tree structure to search N codevectors in $O(\log(N))$ time. Normally, TSVQ requires twice the number of codevectors. In addition to the leaves of the codebook tree $\hat{\mathbf{x}}^c, c = 1, \cdots, N$, it needs the codevectors on the tree branches. If the leaf codevectors are used only in decoding, binary TSVQ can avoid their storage. In binary TSVQ, in every tree branch, the two mean square error evaluations can be replaced with an inner product with the vector normal to the plane separating the two leaves [DH73]. This reduces the computation and averts an increase in number of codevectors for a given codebook.

In the next section, we describe previous work in compression and introduce a novel way for vector prediction. In Section 3, we present VPR and discuss storing the codebook as difference of codevectors. In the ensuing section, we show the reduction in storage of actual codebooks using VPR; we follow with a quantitative analysis of the factors affecting the performance using a pruned TSVQ [CLG88]. Finally, in Section 6, we extend the scheme used for binary tree to M-ary tree.

3.2 Codebook Compression

Source coding schemes [JN84] are applicable to compressing codebook. For example, a scalar quantizer can be used. While it introduces noise, it is simple to implement. In specific, μ-law code [JN84] with $\mu = 255$ quantizes a 13-bit magnitude of a speech sample into a floating point number with 3-bit exponent and 4-bit mantissa (7 bits). Originally used in local telephone stations, this code has been used for compressing the elements of codevectors trained on speech samples. For better performance, one can train the scalar quantizer – the precursor to vector quantizer, on codevector elements.

Dezhgosha and *et. al.* [DJK92] present a non-conventional VQ system. It combines evaluation of L_1 distance metric (mean absolute difference) with non-uniform scalar quantization of the codevector elements. The quantizer is trained on elements which exhibit a 2-sided exponential *pdf.* The element's magnitude are quantized into 9 different values (4 bits). For decoding, look up tables are placed between the storage and the processor. (The paper does not emphasize these tables. Without them, however, conventional weighted representation arithmetic which they use would not operate.) The paper also states that the loss due to the 4-bit scalar quantization to be less than 0.2 (i.e., 36.41 vs. 36.28). These SNR numbers are not in dB units (log of mean square error). Since their design minimizes L_1 (mean of absolute difference) in quantization, they similarly used log of L_1 a recent criterion for error measurement. The reported SNR cannot be used for general comparison, although it does seems marginal. VPR, on the other hand, focuses on noiseless compression which does not affect the distortion in VQ. Moreover, it is arithmetically benign. It decreases the amount of computation.

We use codevector prediction to improve compression. We note an inherent property of codevectors: *the difference between two neighboring codevectors is small.* By encoding the difference between codevectors' centroid, the energy of the stored codevectors reduces. This lowers the entropy. This also can be realized just by pairwise subtraction of centroid codevectors. Even a simple scheme of coding the difference – the hyperplane codevector – can provide significant compression.

In storing codevector difference, VPR is an effective vector compression scheme. By classifying codevectors based on the largest element of the vector in magnitude ($\max |x_i|$), it accounts for the non-stationary nature of codevector whose characteristic varies from one codevector to another. For each vector, VPR chooses the precision to accommodate the largest element in magnitude. For many vectors, all the vector elements are small enough not to need one or more of the leading bits in their representation. This significantly reduces the number of bits for codebook storage.

We will compare VPR with two alternatives: zeroth- and first-order entropy coders. Zeroth-order entropy coding is the best noiseless compression when vector elements are mutually independent. First-order entropy coding is preferred when autocorrelation of vector elements is large. It compresses the difference between an element and its prediction using the adjacent element. In either case, the decoding is somewhat complex. For example, Huffman decoder requires a table (e.g., programmable logic array) to convert the code into a representation more permissive to computation. It also needs shifters for unpacking of vector elements. More importantly, Huffman codes are variable length. Its length cannot be determined prior to decoding. As a result, the decoding is serial. In VQ decoders, where the codevector elements are accessed serially, the marginal gain offered by this method may become attractive; specially so, as transistors per chip increase and chip design becomes simpler. VPR, meanwhile, does not require decoding and is not restricted to serial codevector access. Howard and Vitter [HV92] did present a parallel variant of Huffman coding. It, however, needs a re-allocation network. The re-allocation network distributes the space unused by the short codes to the space needed by the long codes. The re-allocation network requires even more inter-connectivity, incurring further cost. In addition as we will discuss later, difference entropy coding has other implementation difficulties.

3.3 Variable-Precision Representation (VPR)

Variable-precision representation is an extension of Block Floating Point (BFP) [Opp70]. BFP is a floating point system with a single exponent for all the elements of a vector $\mathbf{y}$. Originally used in digital filters [Opp70], BFP represents K n-bit 2's complement integers as a vector of n-bit fractions and a single exponent:

$$\mathbf{y} \;=\; (y_1, y_2, \cdots, y_K)^t \, 2^{e_0}, |y_i| \leq 1 \;\; \forall i$$

BFP has the advantages of floating point numbers, while by having a common exponent for all the vector elements, it requires less storage for the exponents.

BFP, through normalization, avoids overflow (or underflow) [Opp70]. VPR uses BFP representation where the vector is normalized. It views normalization as compression. Through normalization, VPR reduces vector precision. This eliminates e bits per element.

In VPR, leading bits which are zero in all the elements are eliminated and the number of eliminated bits is stored in e:

$$(y_1^{'}, y_2^{'}, ..., y_K^{'})^t \;=\; (y_1 2^e, y_2 2^e, \cdots, y_K 2^e)^t \, 2^{-e}$$

where $\mathbf{y}$ is before and $\mathbf{y}^{'}$ is after block normalization; both $\mathbf{y}$ and $\mathbf{y}^{'}$ are in fractional 2's complement form. The floating point radix is two. This maximizes the bit reduction possible by VPR. In vector normalization, the element with the least number of zeros determines the normalization. Or equivalently the smallest e is chosen such that,

$$|y_i| < 2^{-e} \;\; \forall i$$

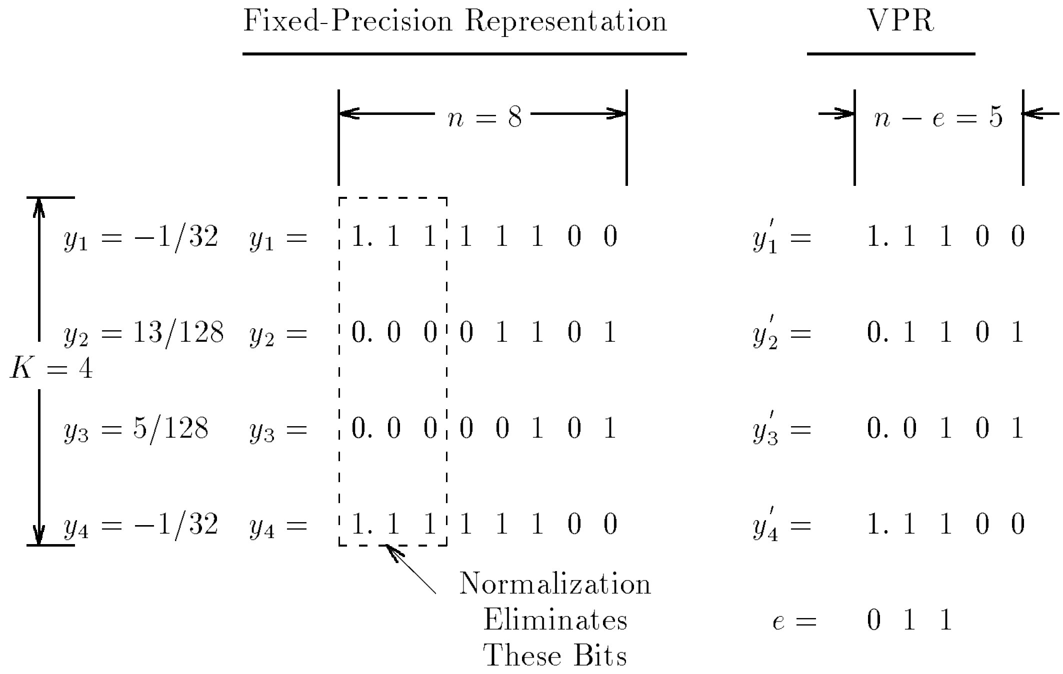


Figure 3.1: Reduction in storage using variable-precision representation (VPR).

This will normalize the fractional representation y',

$$\frac{1}{2} \leq |y_i'| \ \exists i$$

In other words, e designates the smallest interval which contains all the vector elements:

$$e(\mathbf{y}) \quad = \quad \lfloor -\log_2(\max_{\forall i} |y_i|) \rfloor$$

Let's show an example. Figure 3.1 shows the two different ways to store 8-bit precision ($n = 8$) elements of
$\mathbf{y}$ = (-1/32, 13/128, 5/128, -1/32 $)^t$. Note that when a negative number is converted into 2's complement, the left most bits that were zero become extension of the sign. Fixed-precision 2's complement representation needs $4 \times 8 = 32$ bits. VPR eliminates three leading bits ($e = 3$), it needs $4 \times (8 - 3) + 3 = 23$ bits. VPR needs nine less bits.

In general, the n-bit fixed-precision requires nK bits for the K element vector $\mathbf{y}$. In comparison, VPR needs $(n - e(\mathbf{y}))K$ bits for vector elements and $\lceil \log_2(n) \rceil$ bits for vector range e. When $K \geq 3$, $e(\mathbf{y})$ is one or greater, VPR uses fewer bits.

Computation

Let's review the computation the codevectors participate in. It selects one of the two branches in the binary TSVQ:

$$\mathrm{sign}(\mathbf{x}^t\mathbf{y} + b)$$

where $\mathbf{y} = (\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^h)$ is the hyperplane and $b = (\|\hat{\mathbf{x}}^h\| - \|\hat{\mathbf{x}}^i\|)/2$ is the bias term. It is also referred to as a hyperplane test [DH73]. In VPR, we evaluate

$$\mathrm{sign}(\mathbf{x}^t\mathbf{y}' + b')$$

where y_i' is $(10 - e(\mathbf{y}))$ bits, and $b' = b/2^{2^e}$.

To avoid overflow, the dynamic range of the elements is increased in preprocessing. To start with, a pixel is 8-bit. An element of $\mathbf{x}$, x_i, is 9-bits – it is the difference of an 8-bit pixel from the estimate in preprocessing. An element of $\mathbf{y}$, y_i is 10-bits – it is the difference of two 9-bit elements.

3.4 Experimental Results

We generated three codebooks with k-means clustering [LBG80]. Each codebook used a different set of images for training. As shown as an example in Chapter 2, the first codebook was a mean-residual [Bak84] codebook with 16 element vectors ($K = 16$). It was trained on a data set made of 4×4 blocks of pixels ($K = 16$) from 13 256×256 pixel images. The codebook had 2^{11} codevectors. The second mean-residual codebook was trained on a data set made of 4×8 pixel blocks ($K = 32$) from 10 256×256 pixel images. It contained 2^{10} codevectors. The third codebook was for interpolative VQ [HH88] with 32 element codevectors ($K = 32$). Its 2^{10} codevectors were trained on 10 256×256 images. All codebooks used binary TSVQ. The trees were pruned using an algorithm developed by Kiang *et. al.* [KBSC92].

Figure 3.2 uses the entropy function [JN84] as an upper bound in compressing the centroid codevectors $\hat{\mathbf{x}}$, and the hyperplane codevectors $\mathbf{y}$. The zeroth-order entropy H of Random Variable (RV) Y is defined as,

$$H(Y) \;=\; \sum_i -p(Y = i)\log_2(p(Y = i))$$

where Y is an element from one of the codevectors $\mathbf{y}$. Zeroth order entropy measures the information, in bits, required for representing elements of the vector independent of each other.

As shown in Figure 3.2, hyperplane codevector is a good representation for compressed storage of codebook. The hyperplane codevector is a prediction of one centroid codevector

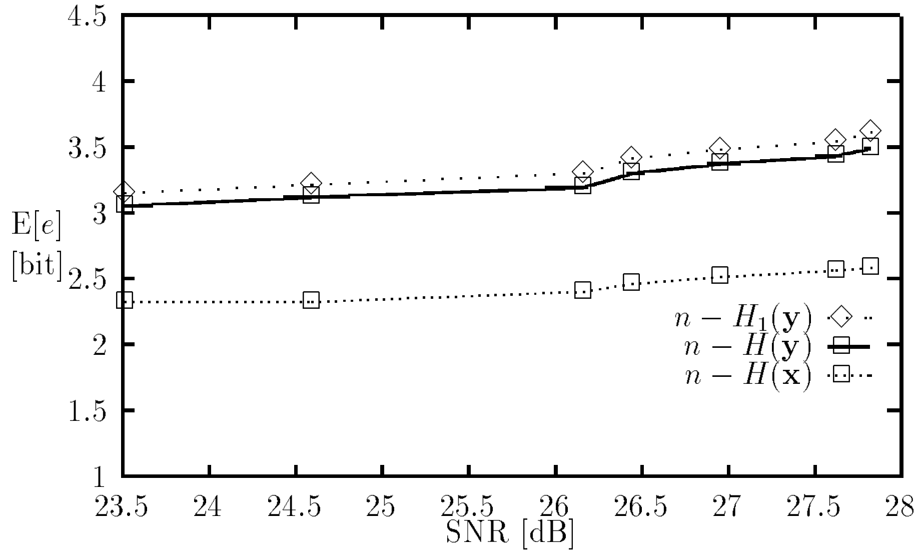


Figure 3.2: Bits reduced with Huffman coding for $K = 16$ codebook.

by a similar one. As can be seen, as a residual of vector prediction, the hyperplane codevector has lower entropy than the original vector. Figure 3.2 plots the first order entropy, which shows the effect of removing the correlation. As can be seen, this can lead to slightly better compression. This suggests that in the codevector subtraction most of the correlation among the elements was also removed. Entropy $H(Y)$ is then close to the maximum possible reduction in storage. Implementation of first order predictor would be both complex and restrict the throughput. To predict the adjacent element, the evaluation needs to be sequential. The evaluation, moreover, would require a multiplier and a subtractor for prediction. These are in addition to the Huffman decoding of each vector element.

As shown in Figure 3.3, the VPR compression ranges from 2.4 bits to 3.4 bits. In other words, on average the largest element in the vector was 1/5 to 1/10 of the dynamic range. Since the vector elements are 10-bit, the reduction in storage ranges from 24% to 34%. The shown memory reduction accounts for exponent storage:

$$e_t(\mathbf{y}) \quad = \quad e(\mathbf{y}) - \frac{4}{K}$$

where four bits overhead to store the exponent is subtracted from the VPR's bit saving. As shown in Appendix B, when including other implementation detail, the overhead increases only 3% per element (from 4.0 to 4.5 bits per codevector).

In these plots, both axis are logarithmic: The exponent e is the logarithm of the maximum of the elements log(max(x_i)). The SNR is logarithm of error's variance -log(σ) in encoding images from the training set. Figure 3.3 shows the same positive linear-like trend in reduction of the entropy as shown in Figure 3.2. We explain the following way: As

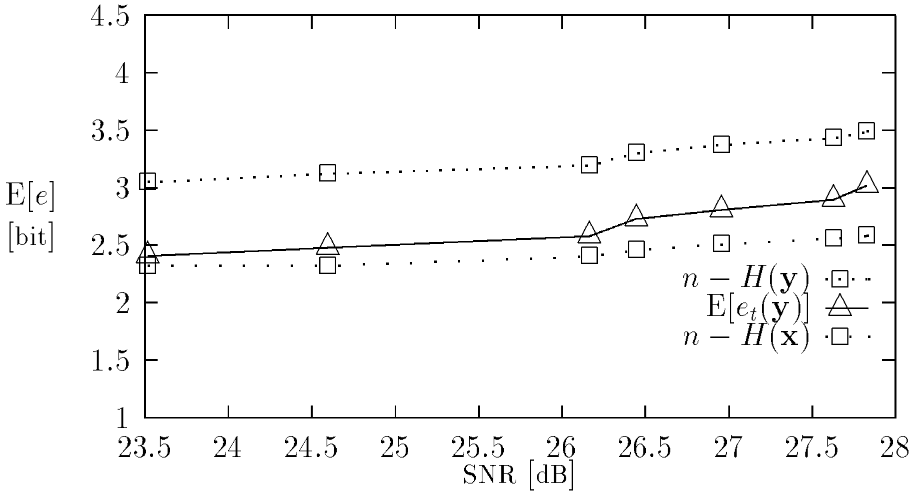


Figure 3.3: Bits reduced for $K = 16$ codebook.

number of codevectors increases, the K-dimensional space, crowded by more codevectors, is partitioned to smaller cells. The smaller distant among the centroid codevectors results in shorter hyperplane codevectors. This improves the compression index $\mathrm{E}[e(\mathbf{y})]$.

By categorizing codevectors in compression, VPR accommodates non-stationary nature of codevectors. As shown in Figure 3.4, the reduction for branches added to the bottom of the tree structure can surpass four bits per element. Its performance surpasses that of Huffman coding, which assumes, incorrectly, that the data is stationary. An entropy coder (e.g., Huffman coder) compresses all elements of a vector with a single coder. Similarly, the scalar Laplacian quantizer used by Dezhgosha [DJK92] has one quantizer for all the codevectors. On the other hand, VPR groups vectors based on exponents and codes each group differently. It exploits the non-stationarity of codevectors.

Figure 3.4 shows that not only subtracting the codevectors improves $\mathrm{E}[e]$, but also preprocessing source vectors, by reducing the vector energy improves $\mathrm{E}[e]$. We will investigate this with VPR representation of auto-regressive sequences.

Finally, VPR is robust. We measured the codebook compression for both 16 and 32 elements codebooks which were coded using MRVQ and Interpolative VQ (IVQ) [HH88]. We also pruned the codebooks to N codevectors. Table 3.1 shows the compression using Huffman and VPR coding. As can be seen, VPR has similar savings for the 32 element codevectors ($K = 32$).

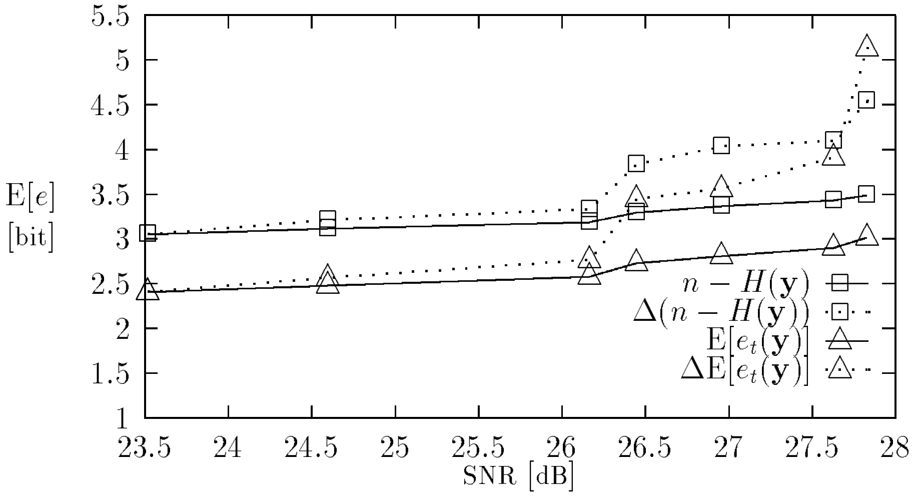


Figure 3.4: Bits reduced for added branches of codebook tree.

Table 3.1: VPR compression of several codebooks.

Code Book	VQ	K	Tree Pruned	N	$n - H(\hat{\mathbf{x}})$ [bit]	$n - H(Y)$ [bit]	E[e] [bit]	Storage Reduced
1	MRVQ	16	Yes	466	2.32	3.05	2.66	26%
2	”	16	No	1906	2.58	3.49	3.27	33%
3	”	32	Yes	292	2.32	3.35	2.65	26%
4	”	32	No	913	2.53	3.78	3.30	33%
5	IVQ	32	Yes	875	2.06	3.68	3.36	34%
6	”	32	No	922	2.11	3.72	3.40	34%

3.5 Analysis of Preprocessing

We will show that source vector preprocessing improves the VPR compression. VQ preprocessing aims at improving the signal compression by extracting features from the signal and compressing them separately. It yields a hybrid coder which achieves better signal compression without requiring bigger codebooks. Some preprocessing techniques such as mean-residual VQ reduce the energy of the source vector ($\mathbf{x}$). Another example we will investigate, gain-shape VQ, normalizes the energy of the source vectors. Both such techniques reduce L_∞ of the source vector x, max(x_i). As we will see, this aids the codebook compression ($e \propto \log(\max(x_i))$).

We will analyze the effect of preprocessing of the source vectors on the codevector compression. We will compress first-order Auto-Regressive (AR) sequence. AR sequence accounts for the correlation which exists in real-world image and sound [Gra84] and which is rooted in the continuity of real signals:

$$\begin{aligned} x_1 &= N\left(\mu = 0, \sigma_n^2 = \sigma_x^2\right) \\ x_k &= \rho x_{k-1} \; + \; N(\mu = 0, \sigma_n^2 = \sigma_x^2(1 - \rho^2)) \;\; k = 2, ..., K \end{aligned}$$

where the vector elements are output of a first order AR source which is excited by a Normal (Gaussian) random variable. This is also referred to as a Gauss-Markov source. The first element is a zero mean Normal RV with variance σ_x^2. Subsequent element are a fraction of the previous element summed with a Normal RV of variance σ_n^2:

$$\sigma_n^2 = \sigma_x^2(1 - \rho^2)$$

The fraction is dependent on the variance-normalized correlation ρ,

$$\rho = \frac{E[X_k X_{k-1}]}{\sigma_x^2}, \;\; 0 \leq \rho \leq 1$$

We note that due to correlation between vector elements, a RV with small σ_n^2 results in vectors elements with large σ_x^2.

In the following analysis, we assume that preprocessing of source vectors effects codevectors similarly. Each codevector $\hat{\mathbf{x}}$ is the centroid of a cluster of source vectors $\mathbf{x}$ determined during the training. During the training on the source vectors, codevectors track the statistical properties of source vectors, inheriting their characteristics. For example, in mean-residual VQ, where preprocessing results in zero mean source vectors, codevectors which we inspected also had zero mean. Or the inspected codevectors $\hat{\mathbf{x}}$ had correlation within the neighborhood of source vector correlation (e.g., $\rho_{\mathbf{x}} = 0.23$ while $\rho_{\hat{\mathbf{x}}} = 0.29$).

Mean-Residual VQ

Mean-residual VQ is a simple form of preprocessing of input signal. The vector mean, which is coded separately, is used to predict the vector. The residual vector which has lower variance is then the vector which is quantized.

Subtracting the vector mean, results in significant reduction in storage. Figure 3.5 shows the affect of removing the mean. It measures the average exponent $\mathrm{E}[e]$ for 10,000 auto regressive vectors with and without their mean. The variance was set at $\sigma_x^2 = 0.17$, and correlation was set at $\rho = 0.98$, while vector dimension K ranged from 1 to 64. This was a relatively high variance picture (256×256 pixel copy of *Baboon*) which was far more than the average values measured on a set of 20 images. For an average AR sequence segment, subtracting the mean, increases the exponent often by more than one bit. For convenience, every pixel was normalized ($-1 \leq x \leq 1$).

One would strictly assume that subtraction increases the dynamic range. To avoid overflow the maximum number of bits used in the representation should increase by one. On average, however, the residual has a smaller maximum. As a result, $\mathrm{E}[e]$ of codevector improves and less bits are used. A supporting argument is that $\mu_{\mathbf{x}}$ estimates the elements. For a vector dimension of $K = 32$, the variance of the residual vector was decreased to 1/6 of the original. Moreover, VPR of the 10-bit element codevectors with a high probability had also the MSB zero $p(e(\mathbf{x} - \mathbf{i}\mu_x) \geq 1) \approx 0.95$, where $\mathbf{i}$ denotes the vector $(1, 1, , ..., 1)$.

In VPR coding, the average exponent of mean-residual codebook gives a conservative estimate on performance of other VQ preprocessing schemes. Mean-residual VQ was chosen for the ease which it could be implemented with. In actual systems, preprocessing can be more elaborate. Other techniques such as interpolative VQ [HH88], which during prediction aim to reduce the maximum, would reduce the codebook storage further. Alternatively, mean-residual VQ can be viewed as a two-band sub-band coder where the high band is vector quantized. More than two bands are shown to be desirable in sub-band coding [WBBW88]. With more bands, the variance of the sub-band coded image decreases. This will also make VPR compress more. For example with the maximum possible number of sub-bands, the coder will be performing a discrete cosine transform of the whole image, where the variance of most coefficients is quite small.

Gain-Shape VQ

Gain-shape VQ is a VQ preprocessing technique which encodes the energy of a vector $\mathbf{x}$ and the normalized shape vector $\mathbf{u}$ ($\mathbf{u} = \mathbf{x}/||\mathbf{x}||$) separately. Average energy of an element of $\mathbf{u}$, $\mathrm{E}[u_i^2]$, becomes

$$\mathrm{E}[u_i^2] = \frac{||\mathbf{u}||^2}{K} = \frac{||\frac{\mathbf{x}}{||\mathbf{x}||}||^2}{K}$$

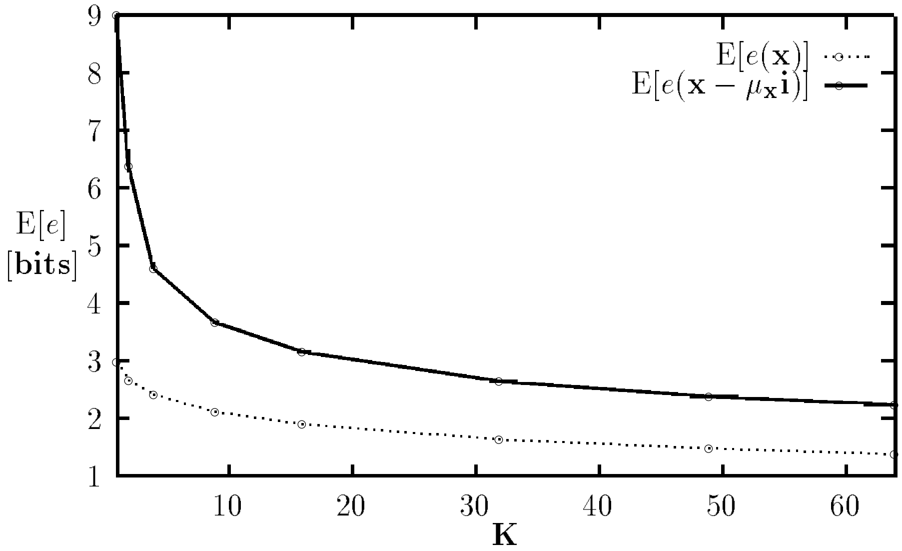


Figure 3.5: Increase in bit reduction E[e] by subtracting vector mean $\mu_{\mathbf{x}}$.

$$\mathrm{E}[u_i^2] \quad = \quad \frac{1}{K}$$

Figure 3.6 shows the performance for normalized vectors, with K varying from 1 to 64. With increasing K, the average energy of an element u_i decreases, improving the average exponent E[e]. In the Figure we have also shown the probability that the MSB is zero, $p(e(\mathbf{u}) \geq 1)$. For large vector dimension K, it is almost one (e.g., $p(e(\mathbf{u}) \geq 1) \approx 0.993 for K = 32$). If we drop the MSB of the magnitude, very few if any of the normalized codevector elements will saturate the reduced dynamic range. This result can be useful in fixed-precision implementation.

The simulations were cross checked. To observe the effect of the pseudo-random generation, two different random numbers were used. The difference in result between the two set of vectors was less than 0.1 bit. The simulation results are insensitive to the statistical fluctuations due to averaging and random number generators to less than 0.1 bits. Each simulation had a set of 10,000 pseudo-random vectors $\mathbf{x}$.

3.6 Theoretical Performance of VPR

We analyze the performance of VPR as a function of vector variance, using an analytical model. The model yields only a lower bound, even so it gives a significant insight. The model measures the performance for a stationary codevector *pdf.* It conservatively deviates from measured values due to the non-stationary nature of codevectors.

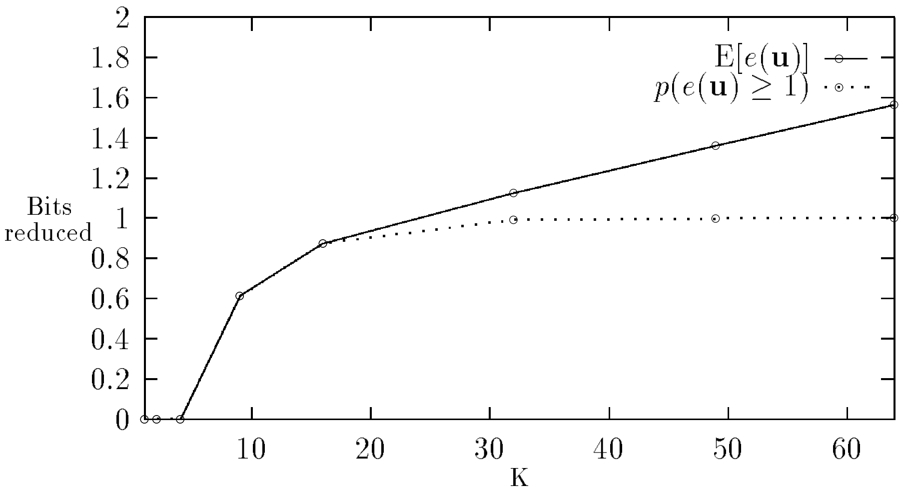


Figure 3.6: For large K the MSB almost always zero ($p(e(\mathbf{u}) \geq 1) \approx 1$).

Recalling the definition for hyperplane codevector $\mathbf{y}$,

$$\mathbf{y} \triangleq \hat{\mathbf{x}}^i - \hat{\mathbf{x}}^h$$

where i and h are two nearby centroid codevectors. A hyperplane codevector is the residual vector after one centroid codevector predicts a nearby one.

We observed that the histogram of hyperplane codevectors $\mathbf{y}$ resembles a two sided exponential *pdf*. Figure 3.7 shows a histogram of the hyperplane codevector elements. The codevectors are from the $K = 16$ codebook mentioned previously. The hyperplane codevector's elements, which are summation of many RV, might be suspected to have a Normal *pdf*. However, as generally known [JN84], the histogram of pixels after prediction instead of being bell shaped has a sharp peak at zero. There are two reasons. First, a Normal *pdf* requires that all the summed random variables to be independent. In real images this cannot be expected. Second, a Normal *pdf* requires that the random variables be stationary. Images, on the other hand, are composed of variety of textures.

As shown in Figure 3.7, two sided exponential *pdf*, also called Laplacian, match the *pdf* of codevector elements well. The Laplacian density of the RV Y is

$$f_y(y) \;=\; \frac{1}{\sqrt{2}\sigma_y} e^{\frac{-\sqrt{2}}{\sigma_y}|y|}$$

Superimposed on the histogram is a zero mean Laplacian *pdf* with $\sigma = 0.097$, where the standard deviation was that of the codevector elements $L(\mu = 0, \sigma = 0.097)$.

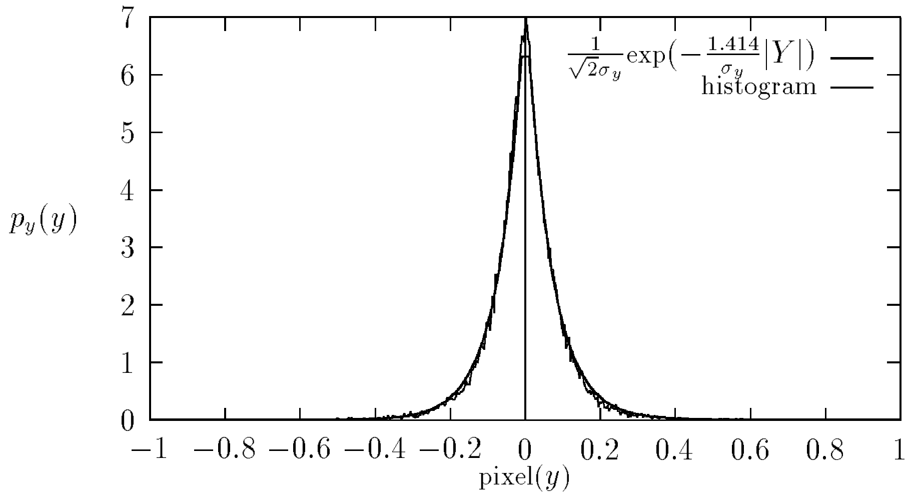


Figure 3.7: A Laplacian *pdf* superimposed on histogram of codevector elements.

Let's estimate the average reduction for a vector of independent identically distributed Laplacian random variables. Since we are looking at the magnitude of the elements, let $G_y(y)$ denote the Probability Distribution Function (PDF) of $|y|$:

$$\begin{aligned} G_y(y) &\triangleq P(|Y| < y) = \int_0^y 2f_y(u)du \\ &= 1 - e^{\frac{-\sqrt{2}}{\sigma_y}|y|} \end{aligned}$$

$G_y(y)$ can be also viewed as the PDF of an exponential *pdf* with mean $\mu = \frac{\sigma_y}{\sqrt{2}}$. Using $G_y(y)$, we can estimate the average exponent $\mathrm{E}[e(\mathbf{Y})]$:

$$\begin{aligned} \mathrm{E}[e(\mathbf{Y})] &= \sum_{i=1}^{n} iP(e(\mathbf{Y}) = i) = \sum_{i=1}^{n} P(e(\mathbf{Y}) \geq i) \\ &= \sum_{i=1}^{n} P(L_\infty(\mathbf{Y}) < 2^{-i}) \\ &= \sum_{i=1}^{n} G_y(2^{-i})^K \\ &= \sum_{i=1}^{n} \left(1 - e^{\frac{-\sqrt{2}}{\sigma_y}2^{-i}}\right)^K \end{aligned}$$

Now, let's compare this analytical result to the experimental results.

The analytical estimate, illustrated in Figure 3.8, is conservative. It significantly underestimates the actual reduction of non-stationary vectors. The analytical estimate assumes all vector elements belong to a memoryless random sequence with Laplacian *pdf* and constant

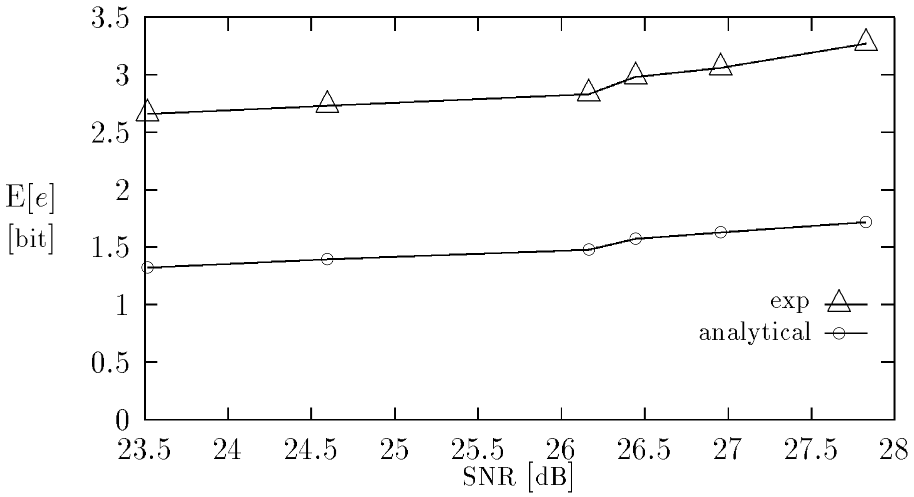


Figure 3.8: Actual VPR reduction and its conservative analytical estimate.

variance. In practice, the hyperplane codevector's variance differs depending on which level of the tree they are located at. Hyperplane codevectors at the top of the tree are the difference between two centroids representing different classes of codevectors, they have high variance. Hyperplane codevectors at the bottom of the tree are the difference between two nearby adjacent centroids: they have low variance. As we descend the codebook tree, the variance of hyperplane codevector $\mathbf{y}$ decreases. The vector element y statistically does not belong to a single stationary class; the codevector and its variance are non-stationary.

As codebook size decreases, the experimentally measured reduction improves faster than the analytical model. Likewise as more codevectors are pruned from the codebook tree, the standard deviation σ_y decreases from 0.12 to 0.09. This means that the pruned codevectors have lower than average variance, supporting our hypothesis that the variance is not stationary.

3.7 M-ary Tree Storage

VPR can be generalized to reduce the storage for M-ary trees. We will present a scheme for storing the difference of any number of codevectors M, where the computation can also be made incremental. In other cases, storing the codevector's difference would be only for removing the redundancy in codevectors. For these, we will present a scheme which will allow concurrent codevector accesses.

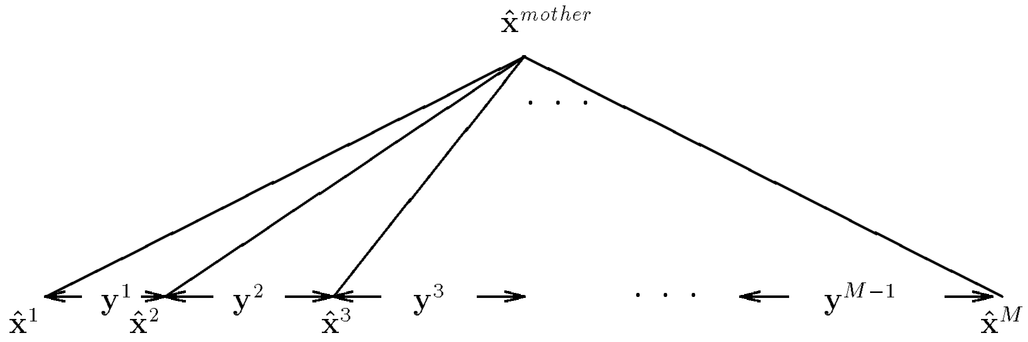


Figure 3.9: Storing codevectors as chain of differences.

Storing Chain of Differences

As shown in Figure 3.9, an M-ary tree can be stored as a chain of differences. Each sibling is predicted by the closest codevector which has not been put in the storage. The sibling is subtracted from the predictor and is stored. The differences forms the link of the chain. To reconstruct a codevector, all the previous codevectors in the chain are accessed. This is ideal when the the M codevectors in the codebook are searched sequentially in a fixed order.

Depending on the sequence of storage, the compression varies. The compression is denoted by the sum of the exponent $e(\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^h)$ of the codevector difference varies. In the following equation, we change the enumeration of M codevectors such that the number of bits used in storage is minimized:

$$\max_{\forall enumeration} e(\hat{\mathbf{x}}^1) + \sum_{i=2}^{M} e(\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^{i-1})$$

Akin to the traveling salesman problem, it can be solved with similar heuristics. To access a codevector $\hat{\mathbf{x}}^j$, we would have to access all the previous codevectors in the tour:

$$\hat{\mathbf{x}}^j = \hat{\mathbf{x}}^1 + \sum_{i=2}^{j} (\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^{i-1})$$

where we have stored $\hat{\mathbf{x}}^1, \hat{\mathbf{x}}^2 - \hat{\mathbf{x}}^1, \hat{\mathbf{x}}^3 - \hat{\mathbf{x}}^2, \cdots, \hat{\mathbf{x}}^M - \hat{\mathbf{x}}^{M-1}$.

In an M-ary tree, we can incrementally perform the inner product. This is similar to binary trees, where the search processor conveniently used difference of codevectors ($\mathbf{y} = (\hat{\mathbf{x}}^l - \hat{\mathbf{x}}^r)$) in the computation, With inner product measure, the evaluation of codevector $\hat{\mathbf{x}}^j$ can use the result of evaluation of $\hat{\mathbf{x}}^{j-1}$. This can reduce the computation:

$$\mathbf{x}^t \hat{\mathbf{x}}^j = \mathbf{x}^t \hat{\mathbf{x}}^1 + \sum_{i=2}^{j} \mathbf{x}^t (\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^{i-1})$$

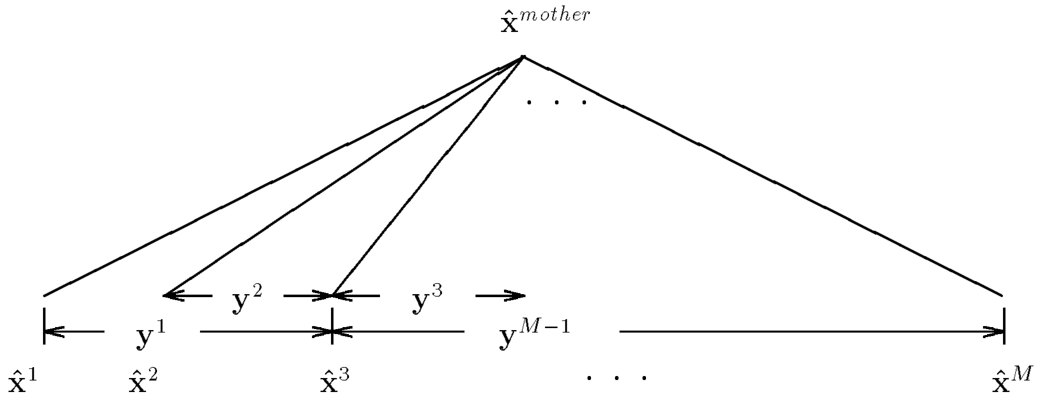


Figure 3.10: Storing differences from a sibling.

Storing Difference from One Codevector

Predicting from only one codevector permits fetching codevectors concurrently. We can test all M codevectors as the predictor, choosing the one with most reduction in saving. For example in Figure 3.10, for reconstructing codevector $\mathbf{x}^i$, we need the predictor $\mathbf{x}^3$ and the difference $\mathbf{y}^i$. Furthermore with inner product metric, we need to store only the codevector difference. This was already shown in Section 2.4. These two factors together permit reconstructing a codevector with only one accesses.

For better compression, we can optimize the prediction. We can generate the optimal codevector. Compression improves when the predicting codevector is similar to all codevectors by L_∞ measure ($\mathrm{L}_\infty(\hat{\mathbf{x}}^i - \hat{\mathbf{x}}^{predict})$). Figure 3.11 shows codevector, $x^{predict}$, where $x^{predict}$ is the centroid which uses the exponent measure for distance:

$$\mathbf{x}^{predict} = \min_{\mathbf{x}}{}^{-1} \sum_i e(\hat{\mathbf{x}}^i - \mathbf{x}^{predict})$$

The above equation yields the predictor which minimizes the distance. Note that when $L_\infty \approx L_2$, this scheme is identical to prediction from $\mathbf{x}^{mother}$. Indeed, $\mathbf{x}^{predict}$ could be iteratively derived from the parent codevector. As already discussed, the codevector $x^{predict}$ itself does not need to be stored.

Prediction from one codevector is most effective for small M. For large set of codevector M, the prediction can be poor. For example, in exhaustive search of large codebook($M = N$), one codevector cannot excel in predicting the other ones. After all, a codebook is collection of codevectors which captures the diversity of the input. Any portion of the signal which would have been common to all the codevectors would have been extracted by the VQ preprocessor before the codevectors were ever generated.

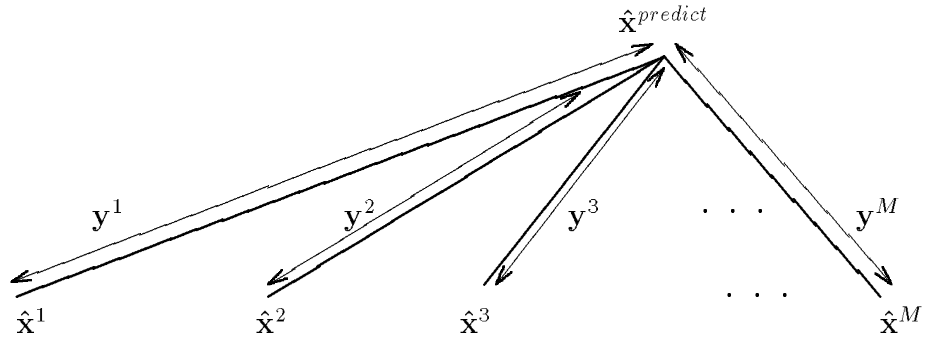


Figure 3.11: Storing difference from a single predictor.

3.8 Conclusion

Hyperplane codevectors are better suited for storage than centroid codevectors. Hyperplane codevectors which are the difference between two similar centroid codevectors have less information than centroid codevector. This was experimentally verified. Both the zeroth- and first-order entropy for 16 element ($K = 16$) codevectors were measured. The reduction with zeroth-order entropy of the centroid codevectors varied from 2.3 to 2.6 bits per element, while for the hyperplane codevectors it varied from 3.0 to 3.5. The improvement was at about 0.7 bits. On the other hand, the first-order entropy, which required a predictor could improve our reduction by about 0.1 bits. This suggests that subtracting similar centroid codevectors has removed most of the element-element correlation. Finally, with VQ preprocessing, even for centroid codevectors there was significant reduction in storage. VQ preprocessor aiming to reduce information in signal, also reduces the information in the codebook.

Variable-precision representation (VPR) is an effective noiseless encoding of hyperplane codevectors, whose decoding is amiable to arithmetic. VPR determines the vector elements' leading bits which are sign extension. It then avoids storing these bits for each vector element. Instead, for each vector, it stores the number $e(\mathbf{y})$. The decoding takes advantage of reduced precision, resulting in negative decoding cost. VPR is also effective in reducing storage. For storing a binary tree structured VQ codebook, VPR saves between 24% and 34% in storage. It compresses 75% as well as entropy coding. VPR effectiveness stems from the non-stationary nature of codevector. The savings increases as the signal to noise ratio (SNR) of VQ coder increases. With even bigger codebooks than were simulated with the current machines, we predicted the saving will be greater than 40%.

CHAPTER 4

VPR Architecture

4.1 Introduction

This chapter discusses VPR's architecture and the cost-performance of its implementation. First, we present bit-serial VPR architecture. Then, we discuss speedup using pipelined and bit-parallel evaluation. This is followed by a VPR implementation which can vector quantize real time video (MPEGII rate). Since in variable-precision representation (VPR) the noiseless compression is achieved by eliminating zero MSBs, we suspect VPR based classifier has low or even negative decoding cost. We verify this by a comparison with a conventional fixed precision implementation. We try to generalize the comparison of these two implementations by analysis of the VPR architecture vis-a-vis the conventional.

4.2 Architecture

This section looks at a bit-serial non-pipelined VPR architecture. The processor architecture is determined by the format of data storage. After we discuss organization of codevectors, we describe the details of the inner product processor.

Memory Organization

Let's consider storage of one codevector. As the precision varies by one bit, the number of bits required for representation vary by K bits. A general approach would treat the elements as separate variable-precision numbers, packed into memory words. The processor would then have to unpack them into fixed precision numbers to feed to the parallel multipliers and adders. The unpacking conversion step would require a shifter.

We present a packing scheme which avoids the unpacking step. We note the common precision of vector elements' mantissa. Let a word have K bits for K element vector. With one word for each bit of precision, the vector $\mathbf{y}$ uses $(n - e(\mathbf{y}))$ words. Each word groups bits of the same weight of all the vector elements, spreading an element across several words. Storage used, $K \times (n - e(\mathbf{y}))$ bits, is the minimum number of bits to represent a VPR vector. In Figure 4.1, the dashed line demarcates one of the vector elements. For example with $K = 16$ and $n = 10$, each vector $\mathbf{y}$ needs $(10 - e(\mathbf{y}))$ 16-bit words per vector. Next, we

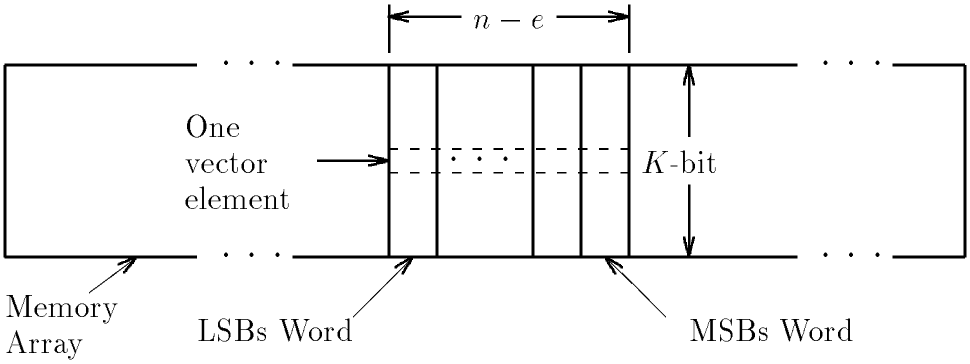


Figure 4.1: In variable-precision representation, storing elements bit-serially.

describe an inner product operator which evaluates $\mathbf{y}$ in the same order as it is stored in Figure 4.1.

Bit-Serial Inner Product

As discussed in previous chapter, the VPR computation determines

$$sign(\mathbf{x}^t\mathbf{y}' + b')$$

where,

$$y' = \frac{y}{2^e} \quad ; \quad b' = \frac{b}{2^{2e}}$$

Recall the vector $\mathbf{y}$ is the hyperplane codevector, and the bias term b is the distance of the hyperplane from the origin.

We can evaluate the codevector as it is read in VPR format. To show this, we dissect the inner product into bit-serial evaluation:

$$\begin{aligned} \mathbf{x}^t\mathbf{y}' &= \sum_k x_k y'_k \\ &= -\sum_k x_k y'_{k,0} + \sum_{j=1}^{n-1-e(\mathbf{y})} 2^{-j} \sum_k x_k y'_{k,j} \end{aligned}$$

The inner loop is implemented in hardware. In other words, in every cycle it multiplies the x_k with j-th bit of y_k and sums the result. The outer loop is sequential evaluation of $n - e(\mathbf{y})$ iterations. As shown in Figure 4.2, evaluation of the inner loop forms the bulk of the processor. For fast execution, the inner product is accumulated in carry-save form. Every cycle, partial products which have the same weight are generated and summed. After the last iteration, the sign detector adds the saved-carry to the pseudo-sum and outputs the sign.

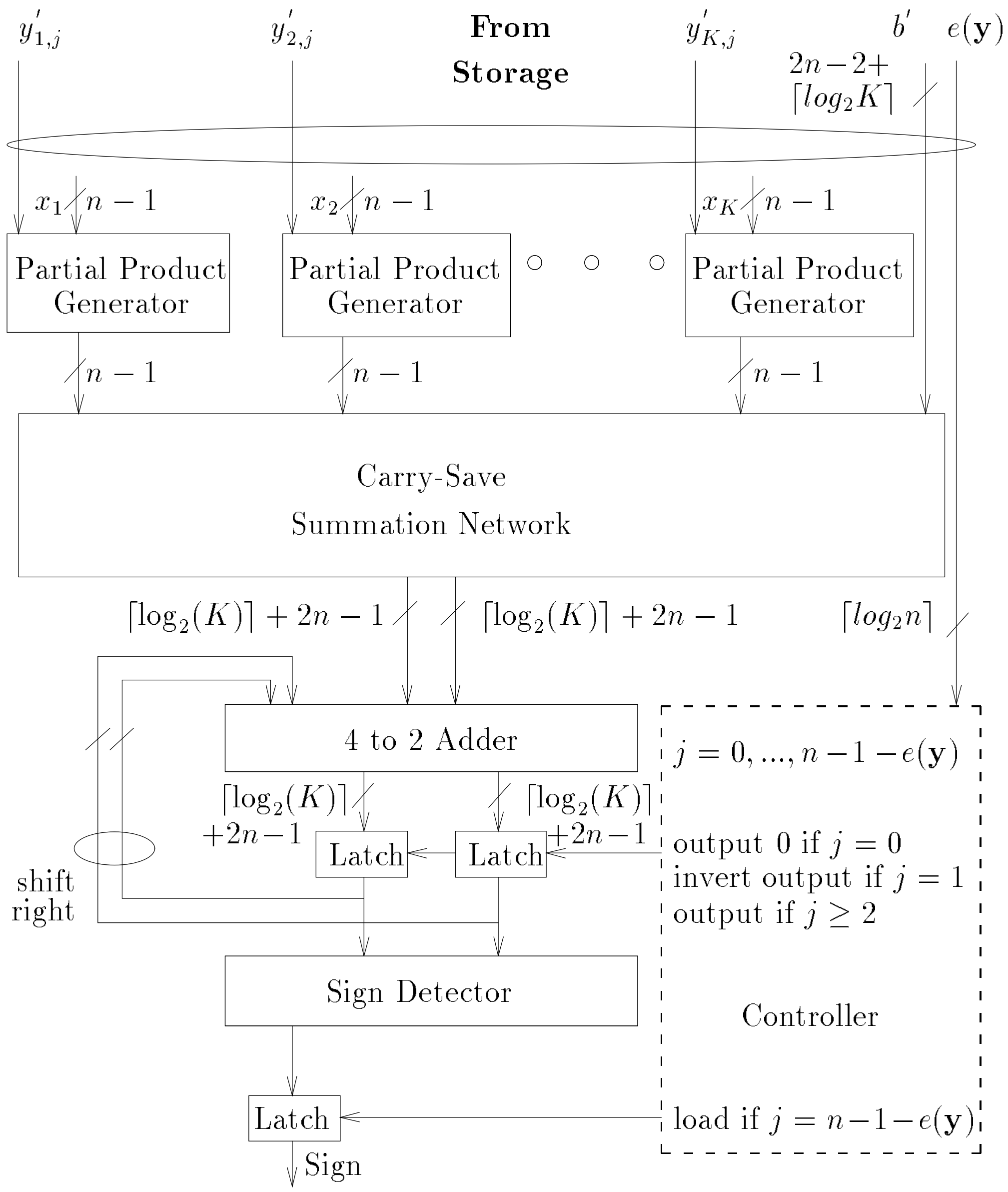


Figure 4.2: Evaluating in j-th cycle $\mathbf{x}\mathbf{y}'_j$, where $\mathbf{y}'_j$ is j-th significant bits of $\mathbf{y}'$.

As we will discuss later in Chapter 5, VPR inner product module can be incorporated into variable-precision classification (VPC) [DE90]; VPC architecture would complement VPR. While VPR would avoid the sign extension MSBs; it would skip the superfluous LSBs. The evaluation would be similar to Figure 4.2, except the bit-serial evaluation would be from MSB to LSB. It would also keep track of the error of the inner product $(\mathbf{x}^t\mathbf{y}' + b')$. When the error becomes less than the estimate of the inner product, the sign of the output cannot change with succeeding evaluation. At this point it could terminate the evaluation.

4.3 Increasing Throughput

A bit-serial architecture, although simple to implement, may not have enough throughput. For increasing the throughput, we look at both bit-parallel evaluation and pipelining.

Bit-Parallel Evaluation

For an arbitrary speedup, we can trade in chip area and pins for bit-parallel evaluation. We evaluate the inner product bit-parallel. Evaluation at higher digit radix reduces the number of iterations. Even when evaluating at higher radix, we vary the storage precision by one bit at a time (radix-2). This retains all of the VPR reduction of storage.

Let's look at the memory organization for a digit-serial processor. Figure 4.3 shows the memory organization for the m-bit digit serial processor. It is similar to the bit-serial memory organization except it widens a word from K bits to mK bits. For m-bit digit, we'll group m K-bit words into an mK-bit word. When we divide the element's $(n - e(\mathbf{y}))$ bits into m-bit chunks, we need

$$\left\lceil \frac{n - e(\mathbf{y})}{m} \right\rceil$$

words. Sometimes, though, we need to fetch one more word. For example, Figure 4.4 shows that a single m-bit access cannot fetch an m-bit element. When the previous codevector uses only part of the word, the current codevector begins at the middle of a word. This worst case scenario is shown by the codevector $\mathbf{y}^1$ which starts in the middle of a word and spreads into an additional word. At worst case, we need ω words,

$$\omega = \left\lceil \frac{n - e(\mathbf{y})}{m} \right\rceil + 1$$

Bit-parallel evaluation requires more memory-processor bandwidth. For one, it accesses m bits at a time, even when only one bit was needed. A less obvious drawback is in VPR packing which results in the MSBs of vector elements to be non-aligned with the word boundary. When vectors are accessed randomly, as is in binary TSVQ, we use only the bits

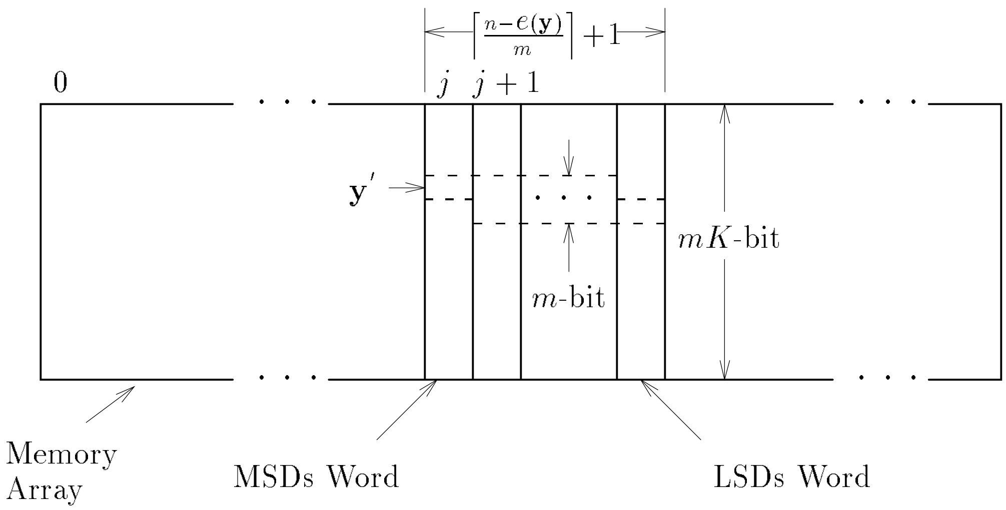


Figure 4.3: Fewer access with mK-bit words.

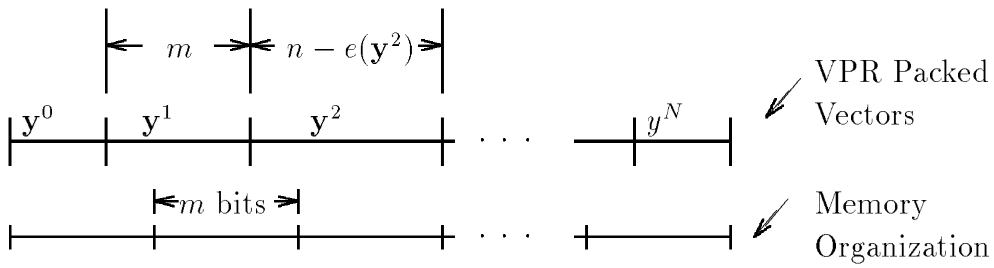


Figure 4.4: A codevector smaller than one memory word yet requiring two memory accesses.

of the target codevector, even though we fetch the whole word. Fetching bits which do not belong to the targeted codevector, incurs additional traffic on the memory-processor bus.

For an architecture with maximum parallelism at the bit-level, let $m = n$:

$$\omega \leq 2$$

It may be possible to fetch a codevector in one access ($\omega = 1$). In the worst case, however, a codevector may be stored across the word boundary. Accessing such a codevector would take two cycles.

For a bit-parallel scheme, the speed up $S_{parallel}(m)$ with respect to bit-serial becomes

$$S_{parallel}(m) = \frac{n}{\omega}$$

In words, the speedup is the ratio of the memory accesses in bit-serial and bit-parallel architectures. For an architecture with maximum bit-level parallelism, for maximum speedup, let $m = n$:

$$\begin{aligned} S_{max\ parallel} &= S_{parallel}(m = n) \\ S_{max\ parallel} &\leq \frac{n}{2} \end{aligned}$$

In other words, misalignment in storing variable length codes results in at least two cycles for evaluation. Area-time ratio of bit-parallel to bit-serial implementation is

$$\begin{aligned} \frac{A_{serial}}{A_{parallel}} \times S_{parallel}(m) &= \frac{K(A_{radix-2\ gen} + A_{adder})}{K(A_{m-bitgen} + A_{adder})} \times \\ &= \frac{K(A_{radix-2\ gen} + A_{adder})}{Km(A_{radix-2\ gen} + A_{adder})} \frac{n-e}{\omega} \\ &\approx \frac{n}{\omega m} = \frac{\frac{n-e}{m}}{\lceil \frac{n-e}{m} \rceil + 1} \end{aligned} \tag{4.1}$$

Due to truncation and alignment issue, the evaluation time does not inversely decrease with the processor size.

The inner product evaluation can be broken to summation across memory accesses, bits' weight, and vector elements:

$$\mathbf{x}^t \mathbf{y}' = \sum_{j=0}^{\omega-1} \underbrace{\sum_{i=jm}^{jm+m-1} 2^{-i} \sum_k x_k y'_{k,i}}_{\text{hardware}} \tag{4.2}$$

where the two inner summations are implemented in hardware, and the outer summation is sequential. For simplicity, we have assumed that the elements are non-negative numbers.

To reduce wiring, we reorganize the mK partial product generators of an element together into K m-bit partial product generators –K sub-multipliers of m by $(n-1)$ bits. This is reflected in changing the summation order from (4.2):

$$\mathbf{x}^t\mathbf{y}' = \sum_{j=0}^{\omega-1} \underbrace{\sum_{k} \sum_{i=jm}^{jm+m-1} 2^{-i} x_k y'_{k,i}}_{\text{hardware}}$$

Figure 4.5 shows the corresponding digit-serial processor. The digit-serial processor is also similar to bit-serial processor, except it replicates the partial product generators m times, and it masks to annul the portion of the word which belongs to another codevector. Effectively, on the fly, it trims the accessed elements to $(n - e(\mathbf{y}))$ bits.

As shown in Figure 4.6, the partial product generator incorporates m AND gates to mask out the bits not belonging to the codevectors. When accessing codevectors randomly, as is done in binary TSVQ, the masks allow us to fetch a VPR codevector with variable number of words.

Pipelining

Pipelining is an attractive means of increasing throughput. It is both low cost and simple to incorporate. It reduces the inner product evaluation time by reducing the cycle time. Instead of waiting for propagation through the summation network, we need to wait only for propagation through one pipeline stage. The pipelining speedup $S_{pipe}(l)$ [Kog81] is

$$\begin{aligned} S_{pipe}(l) &= \frac{nT_{original}}{(n+l-1)T_s}, \; T_s \approx \frac{T_{original}}{l} + T_{latch} \\ &\approx \frac{nl}{n+l-1} \end{aligned}$$

As the number of stages of the pipeline l increases, stage time T_s decreases. Even if each codevector to be chosen for classification depends on the previous one, S_{pipe} still can be large. For a codevector with precision n, there are n independent evaluations. In the limit, as the levels of pipeline l increases, the speedup approaches n:

$$\lim_{l\to\infty} S_{pipe}(l) = n$$

In words, the pipelined processor can at most fully evaluate the classification in the time the non-pipelined processor would evaluate one bit. As we will show later, there are practical impediments to increasing l. In practice, the number of pipeline stages l is few. Pipelining incurs overhead due to latency and T_{latch}.

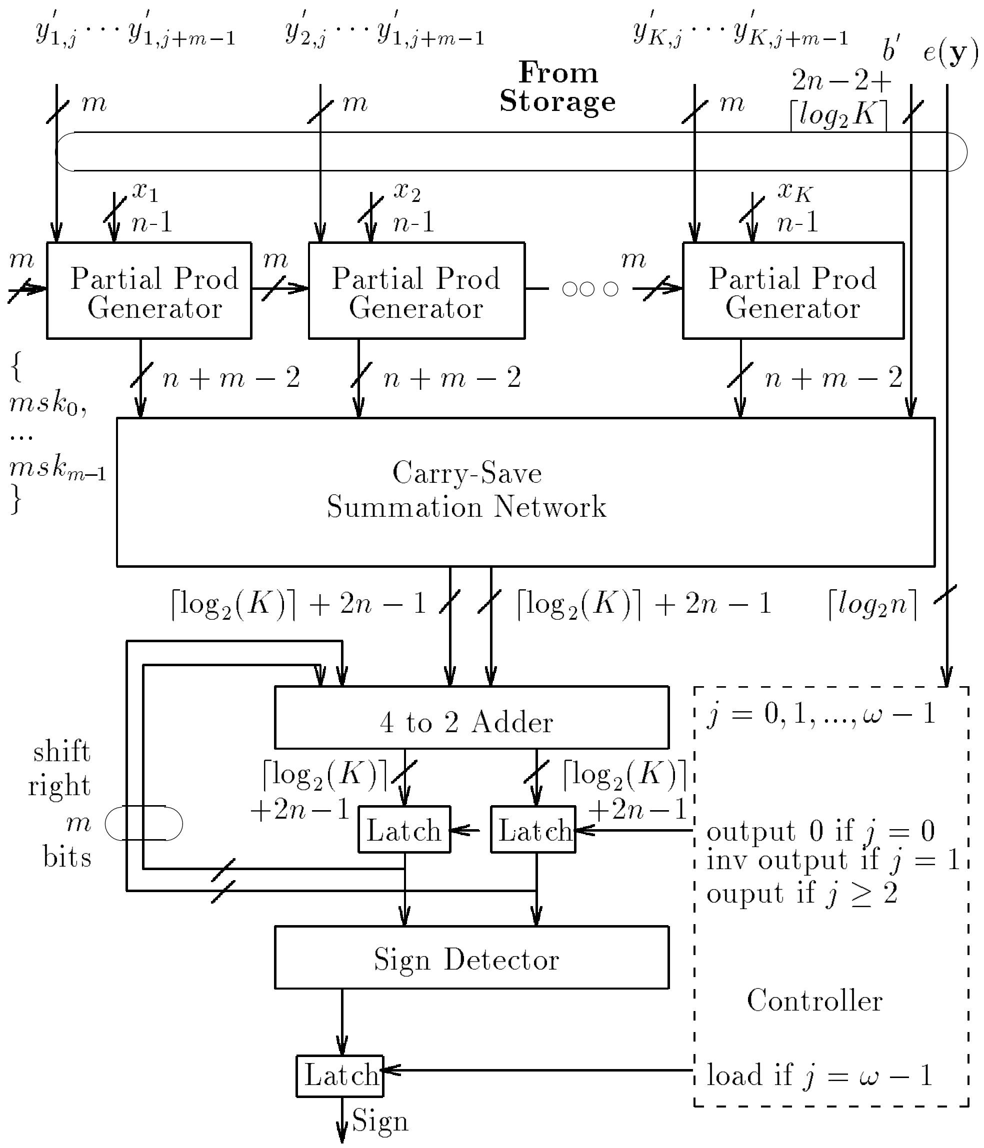


Figure 4.5: Evaluating m bits of codevector element per iteration.

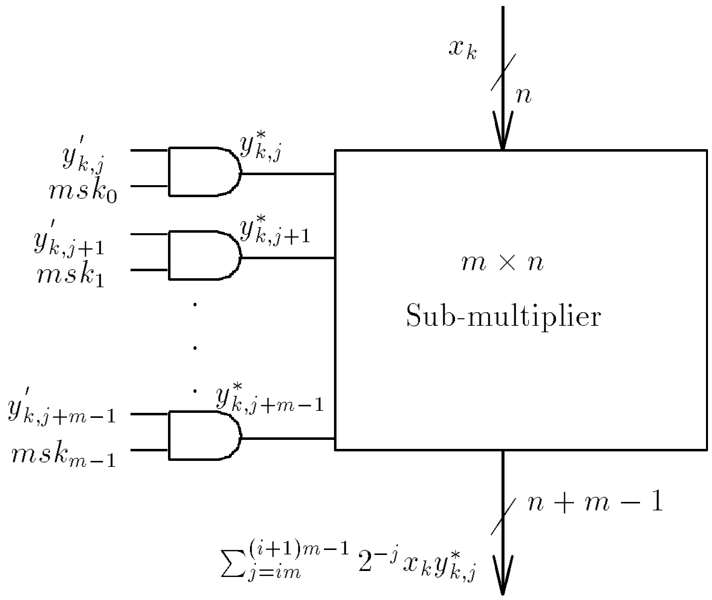


Figure 4.6: $n \times m$ partial product generator uses an m-bit mask.

Pipelining area-time ratio is

$$\frac{A_{serial}}{A_{pipe}} \times S_{pipe}(l) = \frac{1}{1 + lA_{buffer}/A_{serial}} \times \frac{nl}{n+l-1} \tag{4.3}$$

Pipelining costs little – it only needs some intermediary latches. All the while, pipelining permits far more processing for a given amount of silicon area.

Now we compare the area-time ratios of pipelining and bit-parallel evaluation by dividing (4.3) and (4.1):

$$\frac{A_{parallel}}{A_{pipe}} \times \frac{S_{pipe}(m)}{S_{parallel}(m)} = \frac{Km(A_{radix-2\ gen} + A_{adder})}{KA_{radix-2\ gen} + A_{adder} + lA_{buffer}} \times 1$$

Since $(K(A_{radix-2\ gen} + A_{adder}) > A_{buffer})$, for small m bit-parallel speedup costs more area than pipelined speedup.

Pipelining and 4-ary Search Trees

When using tree structured codebooks, a binary tree is favored over M-ary ($M > 2$) tree. A tree with 2^D leaves, when structured into a binary tree requires D inner products, while when structured into an M-ary tree requires $Mlog_{M-1}(2^D) = DMlog_{M-1}(2)$ inner products. Even though an M-ary tree requires more evaluation ($M > 2$), it might find less distorting codevector.

On a pipelined architecture, a 4-ary tree may execute in comparable or even less time than a binary tree. A 4-ary tree has one more inner product than binary tree but has one less pipeline break. In two binary branches, two inner products are evaluated randomly:

$$2\,T(\text{Binary Branch}) \;=\; 2\left(\left\lceil \frac{n}{2} \right\rceil + l - 1\right) + 1 \quad cycles$$

where we have assumed half the time a codevector would be misaligned. In 4-ary tree, three inner products are evaluated sequentially. On a pipelined processor, only the first one incurs $l-1$ cycles of pipeline latency:

$$T(\text{4-ary Branch}) \;=\; \left\lceil \frac{n+n+n}{2} \right\rceil + (l-1) + 0.5 \quad cycles$$

For what pipeline latency l, and precision n, does one 4-ary branch evaluate in comparable or less time than two binary branches?

$$\begin{aligned} T(\text{4-ary Branch}) &\le 2\,T(\text{Binary Branch}) \\ \left\lceil \frac{n+n+n}{2} \right\rceil + (l-1) + 0.5 &\le 2\left(\left\lceil \frac{n}{2} \right\rceil + l - 1\right) + 1 \\ \left\lceil \frac{n}{2} \right\rceil &\le l - 1.5 \end{aligned}$$

For the implemented radix-4 architecture which was pipelined to six levels,

$$n = 10 \quad ; \quad l = 6$$

searching a 4-ary tree would require slightly less time than searching a binary tree. It would take 2% less time than binary trees (20.5 versus 21 cycles), even though there is 50% more computation (3 versus 2 inner product).

Our implementation which we discuss next was for searching binary trees. At the time only algorithms to search 4-ary trees were not readily available. It can be easily modified for a 4-ary tree. To do so, the sign detector would be replaced with a maximizer – a register which keeps track of the maximum of inner products and a comparator which compares the maximum and the output of inner product both in redundant representation.

4.4 Implementation

To fully assess VPR, we proceed to implement the VPR core of the VQ processor. The developed core evaluates the sign of the inner product, where the codevector is in VPR format. As shown in Appendix A, it is targeted for MPEGII [1] data rate. We used VIEWlogic

[1] 720×576 pixels at 30 frames per second

for capturing the design schematic and verifying it. We also measured the critical path assuming that it was fabricated in LSI Logic 1.0-μ gate array [Cor91].

Gate array is attractive for two reasons. First, gate array provides a test ground for comparison of two competing architectures. Given the schematics of the design, we can estimate the cycle time and know the gate count. Second, the gate array is a viable means for production. Tools automatically will map the schematic into a gate array. Partially fabricated dies will expedite the fabrication time and reduce the cost.

In our analysis we have also evaluated the impact of pipelining. A six stage pipeline ($l = 6$) can decrease the total execution time about 25%. It is simple to pipeline either of the designs. For accumulation both designs used CSAs. To pipeline we only need to insert the pipelining registers.

VPR Implementation

A bit-serial (radix-2) architecture would be simple to design. It would also minimize the memory bandwidth. However, even when pipelined, the processor could not encode at the video rate of MPEGII standard. Therefore, we implemented a radix-4 design. We have also developed the pipelining. The design and its pipelining are presented in Appendix B.

The VPR implementation is based on Figure 4.5 using radix-4 partial product generators. The VPR core is 16 ($= K$) radix-4 partial product generators and a summation tree. The radix-4 partial product generators, which use string recoding (Booth) of the multiplier, act like digit-serial multipliers. Their output along with the bias term is gathered by a tree of adders and accumulated in a carry-save redundant form. Through the multipliers, the addition tree is iteratively fed the digits of the VPR codevector. It presents its final accumulation to the sign detector. For fast execution time, the sign detector is based on a carry look ahead adder as discussed in Appendix B.

VPR has 4-bit of control information for each stored codevector. For 10-bit elements ($n = 10$) in VPR format, the exponent e would nominally require four bits. Inspections of the codebooks presented in the previous chapter show that less than 1% of the time $e \in \{8, 9, 10\}$. Limiting e to three bits, results in negligible decrease in VPR compression. To minimize its storage cost, we store e in three bits. We use the fourth control bit to store an offset marker. It flags when the codevector begins in the middle of a radix-4 memory word. The VPR controller is described in detail in Appendix A.

Fixed-Precision Implementation

For comparison, a conventional architecture based on multipliers and adders is designed. It evaluates the inner product as follows:

$$\begin{aligned} \mathbf{x}^t\mathbf{y} + b &= \sum_k x_k y_k + b \\ \mathbf{x}^t\mathbf{y} + b &= \sum_{i=0}^{3} \underbrace{\sum_{k=4i+1}^{4i+4} x_k y_k + b}_{\text{hardware}} \end{aligned}$$

where the inner summation is implemented with four multipliers and adders, and the outer summation is executed in four iterations. The design is described further in Appendix B.

4.5 Comparison

For comparison, we conservatively estimate the execution time which included signal routing overhead. In both designs, the critical path has been traced. It starts from the string recoding circuitry through the addition circuitry. Since for low latency, both alternatives for the next address are fetched to the controller. As soon as the comparison determines the sign, the address multiplexor outputs either right or left branch address to fetch the next codevector. With on-chip memory, address multiplexing and memory access require 21 ns. In minimally pipelined design ($l = 2$), memory access requires one pipeline stage. In maximally pipelined design ($l = 6$), it requires three clock cycles.

In both implementations, we conservatively estimate the execution time. The critical path is determined to go from the string recoding circuitry through the addition circuitry. In the measurement of the critical path, we account for a conservative effect of routing. Since for low latency, both alternatives for the next address are fetched to the controller. As soon as the comparison determines the sign, the address multiplexor outputs either right or left branch address to fetch the next codevector. With on-chip memory, this can be performed in 21 ns. In minimally pipelined design ($l = 2$), memory access requires one pipeline stage. In maximally pipelined design ($l = 6$), it requires three clock cycles.

In radix-4 VPR design, there are overhead both due to misalignment and fetching odd number of bits. When the number of bits is odd, half the time we read in an extra bit. When the number of bits is even, half the time the word may be misaligned, forced fetching an extra word. Assuming that both the probability of being even or odd, aligned or misaligned is each 0.5, then on average half the time we fetch an extra word. This increases the average number of bits fetched to four (8 bits), where we assumed a relatively conservative 3.0 bits reduction (3.3 bits reduction was the maximum measured).

VPR compresses the codevectors, and it has a negative decoding cost. That is, VPR

Table 4.1: Memory bandwidth of VPR versus fixed-precision design.

	Pipeline [level]	CK [ns]	Avg(Worst) Time[cycle]	Avg(Worst) Time[ns]	BW [pin]	$BW \times T \times 10^{-6}$ [pin$\times$sec]
VPR	2	20.7	5(7)	103.5(144.9)	42	4.35(6.09)
Conv	2	20.7	5	103.5	45	4.66
VPR	6	8.3	9(11)	74.7(91.3)	42	3.14(3.83)
Conv	6	8.3	9	74.7	45	3.36

Table 4.2: Performance and cost of VPR versus fixed-precision design.

	Pipeline [level]	Area [gate]	CK [ns]	CK [FA]	Avg(Worst) Time[cycle]	$AT \times 10^{-3}$ [gate$\times$sec]	$AT^2 \times 10^{-12}$ [gate$\times$sec^2]
VPR	2	9100	20.7	7	5(7)	0.94(1.32)	97(191)
Conv	2	9993	20.7	8	5	1.03	107
VPR	6	11164	8.3	5	9(11)	0.83(1.02)	62(93)
Conv	6	12841	8.3	4	9	0.96	72

reduces the total time of VQ. Table 4.2 shows the number of gates, the clock cycle (CK) and the evaluation time for several designs. For VPR implementation, we also show the worst case time in parenthesis. We have calculated AT^2 ($Area \times (Time)^2$), which in many instances has been used as a complexity measure. For VQ at the required throughput, we can halve the time by doubling the area, so AT is a better complexity measure. For variable-time inner product, which can be used in searching pruned trees, VPR is on par with conventional. For a fixed time inner product, VPR's compression does incur 21% more AT cost.

Table 4.1 shows the number of bits accessed from memory and $BW \times T$. We omitted pins required in both architecture to fetch the address. In conventional architecture, in every cycle we fetch four 10-bit elements and five bits of bias. In VPR architecture, in every cycle we fetch 16 radix-4 digits, four bits of bias term, an offset bit, three bits of exponent, and in addition the address space has two bits more resolution address. On average, VPR again performed better than conventional. For the worst execution time, however, VPR has a problem similar to other variable length codes. Due to packing of codevectors, an added word sometimes has to be accessed. Even so, the number of pins used may be three pins less (45 versus 42 pins). At most in binary classification where we fetch both exponent for low latency address generation, VPR will have the same number of pins as conventional. Also in the single chip system discussed next, we note that the memory-processor bandwidth occurs inside the chip. The bandwidth comparison becomes unimportant.

VPR, through storage reduction, can reduce system complexity. Given that the VQ system meets the throughput of the application, the key criteria is the system complexity. In comparing VPR and conventional, let's consider a single chip system. We can store relatively large codebooks on the same chip which houses the processor. For storage, we choose Read Only Memory (ROM) in lieu of Random Access Memory (RAM). A ROM cell requires far less area than RAM. In the 1.0-μ technology, 7 bits of ROM can fit in the same area as a single gate. It is quite feasible to put a 4096 entry codebook with 16 element codevectors ($N = 4096, K = 16$) on ROM by the the processor. The conventional system requires 64 mm^2, while VPR permits putting the processor along with the codebook in just 45 mm^2, a 30% saving. These numbers were calculated for 1.0-μ technology. We believe that in other technologies, VPR similarly reduce the storage. We like to remind, that in a VQ system, we can use the 30% area to store bigger better performing codebooks.

4.6 Analytical Comparison

To better illustrate the on-line product summation, we compare it with two alternatives: first with several parallel multipliers and adders, and second with K conventional serial-parallel multipliers [RPT89]. The qualitative comparison is supported numerically, by estimates of the number of full adders each structure requires. To simplify the comparison, all alternatives use linear array of adders.

The inner product to be evaluated is

$$\begin{aligned}\mathbf{y}^t\mathbf{x} &= \sum_k y_k x_k \\ &= y_1x_1 + y_2x_2 + \dots + y_K x_K\end{aligned}$$

where each of the K elements for simplicity is n unsigned bits. Decomposing the multiplication to addition of partial terms,

$$\begin{aligned}\mathbf{y}^t\mathbf{x} = \; & (-y_{1,0}x_1 + 2^{-1}y_{1,1}x_1 + \dots + 2^{1-n}y_{1,(n-1)}x_1) \\ & +(-y_{2,0}x_2 + 2^{-1}y_{2,1}x_2 + \dots + 2^{1-n}y_{2,(n-1)}x_2) \\ & +\dots \\ & +(-y_{K,0}x_K + 2^{-1}y_{K,1}x_K + \dots + 2^{1-n}y_{K,(n-1)}x_K)\end{aligned} \tag{4.4}$$

A parallel multiplier concurrently evaluates the contents of the parenthesis and an adder accumulates the result. The multiplier uses n^2 and an accumulator uses $2n + \lceil \log_2 K \rceil$ full adders. Each of the K elements uses a multiplier and an adder for one cycle. The total number of full adders $\times$ cycles, $FA_{parallel}$, is

$$FA_{parallel}(K,n) = K(n^2 + 2n + \lceil \log_2 K \rceil)$$

Now, we rearrange the above equation to sum the terms with the same weight concurrently, and iteratively shift and accumulate the sum:

$$\begin{aligned}\mathbf{y}^t\mathbf{x} = \; & -(y_{1,0}x_1 + y_{2,0}x_2 + \dots + y_{K,0}x_K) \\ & +2(y_{1,1}x_1 + y_{2,1}x_2 + \dots + y_{K,1}x_K) \\ & +\dots \\ & -2^{1-n}(y_{,1(n-1)}x_1 + y_{2,(n-1)}x_2 + \dots + y_{K,(n-1)}x_K)\end{aligned} \tag{4.5}$$

This reordering is similar to the way Peled and Liu [PL74] distributed the multiplications in a digital filter when implementing with read only memory (ROM). In the ROM, all different ways elements of $\mathbf{x}$ can be summed together were precomputed and stored. The bit-vector of the same significant bits $\mathbf{y}_i$ would then address the ROM. The output of the ROM would be equivalent to one of the terms in the parenthesis. Instead of embedding the processing bit-serially using a ROM, VPR embeds the processing in a summation network.

For n iterations, the VPR structure uses K partial term adder and an accumulator. In VPR, the partial term adder operates on longer words than conventional. A partial term adder has $n + \lceil \log_2 K \rceil$ Full Adders (FA). The accumulator has $2n + \lceil \log_2 K \rceil FA$. This includes one bit of sign extension for on-line sign detection. The total number of full adders $\times$ cycles, FA_{vpr} is

$$FA_{vpr}(K,n) = nK(n + \lceil \log_2 K \rceil) + n(2n + \lceil \log_2 K \rceil)$$

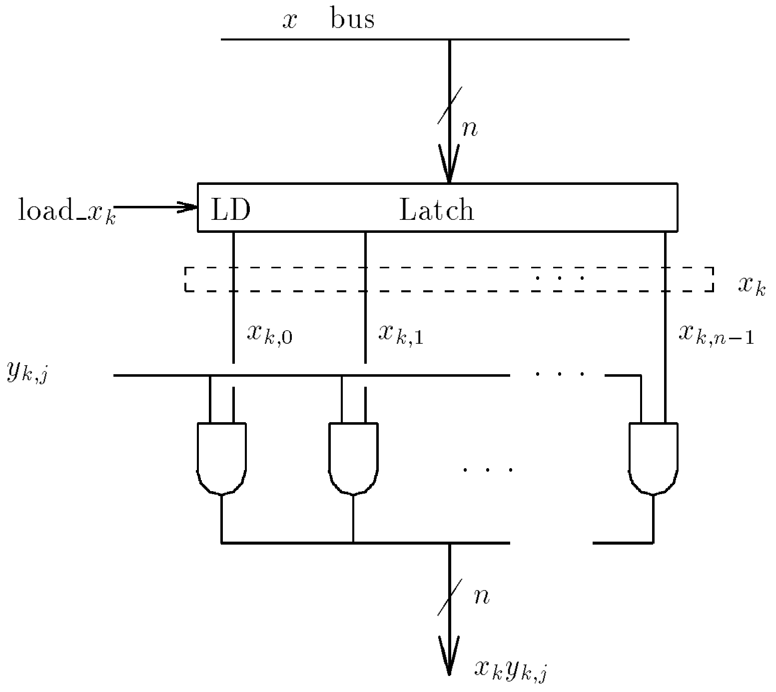


Figure 4.7: Bit-serial partial term generator as a $1 \times n$ multiplier.

Finally, we compare VPR with bit-serial multipliers. Figure 4.7 shows the partial term generator used in the bit-serial arithmetic of Figure 4.2. In conjunction with an accumulator, it forms a serial-parallel multiplier [NR86], [RPT89]. The conventional sequential multiplier is an accumulator and a partial term generator, which transfers the result to an auxiliary latch. For n cycles, K multiplier iterate over $(2n + \lceil \log_2 K \rceil)$ full adders. With shimming the multiplications of adjacent elements in the vector by n cycles, one multiplier can be initialized with the result of the adjacent multiplier. Then, just K sequential multipliers perform the inner product.

$$FA_{serial}(K, n) \quad = \quad nK(2n + \lceil \log_2 K \rceil)$$

A parallel adder and a latch replace K bit-serial adders and K parallel auxiliary latches which would sum the multipliers' output. Moreover, when not bit-level pipelined, there is no equivalent of accumulation latch. In comparison to the VPR inner product, this design approximately needs additional nK full adders and latches. This design is similar to the bit-serial implementation by Ramamoorthy and *et. al.* [RPT89].

Routing Comparison

In full custom implementation, the VPR inner product structure may incur more routing overhead than the multiply accumulate structure. When the multiplier is the linear-multiplier, the multiplicand can be transferred between the partial product generators by abutment. On the other hand, in VPR structure all the elements x_k are fed to partial term generator. This increases the routing overhead with respect to the linear-multiplier,

$$
\begin{aligned}
\Delta_{wiring} &\propto K \times \text{VPR PPGEN}_{wiring} - \lceil K/n \rceil \times \text{Multiplier}_{wiring} \\
&\propto nK - \lceil K/n \rceil n \\
&\propto nK
\end{aligned}
$$

In words, the increase in routing depends linearly on both K and n. On the positive note, storing the source vector $\mathbf{x}$ on-chip requires only as many registers as there are partial product generators (i.e., nK bits). Since the number of latches are few, the high bandwidth between the inner product hardware and the source vector can be encapsulated in the chip.

VPR can have a lower latency summation network than conventional. In our analytical comparison, we assumed a linear summation network. Now let's look at a tree summation network used to minimize the summation delay. In VPR, summation is done by one network. With multipliers and accumulators the summation is broken into two networks. First, the multiplier sums the partial terms. Then the adders accumulate the output of the multipliers. This partitioning results in non-optimality. It increases the overall depth of the adder tree by two full adders when compared to VPR's summation network.

VPR for a Programmable-Precision Filter

Since inner product is used also as a digital filter, VPR can be also used for digital filtering, although the application is different. In a programmable filter not only the coefficients may vary, but also their precision may vary also. VPR can be used for increasing the range of throughput the filter module can accommodate. The vector coefficient $\mathbf{y}$ would be stored as VPR. VPR can have similar application to what the architecture by Peled and Liu [PL74] implemented using ROM. The VPR network can be used to implement programmable precision filter. As the required precision of coefficients decreases, the number of iterations also decreases. For example:

Coefficient Precision		Bit-Serial Iterations	String Recoded Iterations
5-bit	$\Rightarrow$	5	3
10-bit	$\Rightarrow$	10	5

That is, if the application requires less precision, one could demand a higher throughput. VQ and filters differ only in accessing the operands $\mathbf{y}$: VQ fetches a codevector $\mathbf{y}$ from the memory, while FIR filter fetches it from a shift register acting as a tapped delay line.

4.7 Conclusion

Variable-precision representation is noiseless encoding which is amiable to arithmetic. Unlike other compression schemes not only it does not require costly decoding, it even reduces the computational complexity. VPR determines the vector elements' leading bits which are sign extension. VPR avoids storing $e(\mathbf{y})$ leading bits of each vector element. Instead, for each vector, it stores the number $e(\mathbf{y})$. In the radix-4 implementation, we also store an offset bit, which marks if the number uses half of a radix-4 memory word. Over all, a VPR codevector has four bits of overhead.

A VPR inner product is implemented. It processes the codevector element's digit-serially, and the source vector's elements in bit-parallel. Incorporating string recoding, it achieves a high throughput and demonstrates the issues in using bit-serial variable-precision representation and radix-4 evaluation. It has lower complexity than a sequential multiplier based design. When used as a variable-time module, it also has lower complexity than conventional multiply-accumulate approach.

We also noticed the effect of pipelining on inner product module. In a pipeline module, evaluating 4-ary branch would take slightly less time (i.e., 2%) than evaluating two binary branches. In two binary branches, since the two inner products are evaluated randomly, the pipe is flushed twice. In a 4-ary branch, since the three inner products are evaluated sequentially, the pipe is flushed only once.

CHAPTER 5

Variable-Precision Classification (VPC)

5.1 Introduction

Classification is the cornerstone of digital signal processing. Originally, it was used in the communication channels for distinguishing signals from noise. Now it also compresses sounds and images [Gra84], [MRG85] being used as a vector quantizer. It also appears in recognition system for speech [Kav86] or characters [NJ85], and in image segmentation [GBJM79]. Moreover, cluster of classifiers are being evaluated for connectionist computations [Wid87]. Many researchers develop and enhance systems using the classifier as one of the given building blocks.

It was conventionally believed that as the precision of the operands increases, so would the complexity of the classifier. For example, a conventional inner product based classification which uses multipliers and adders would have $O(n^2)$ complexity with respect to precision n since multipliers have $O(n^2)$ complexity with respect to precision. This chapter describes variable-precision classification (VPC) algorithm and shows that the average computation complexity is independent of operand precision. In practice, VPC can reduce the average number of iterations by skipping evaluation of the unnecessary LSBs. This conveniently complements VPR's elimination of superfluous (zero) MSBs.

First in Section 5.2, we show how Binary VPC (BVPC) algorithm evaluates the discriminant bit-serially to the necessary precision. Ensuing this, we derive an upper bound on the operand precision necessary for determining the output's sign. Next in Section 5.3, we derive an upper bound on the average necessary-precision in BVPC algorithm. Finally in Section 5.4, we generalize Binary VPC to M-ary VPC, and let an auxiliary theorem generalize the upper bound for BVPC to M-ary VPC.

5.2 Binary VPC

From Chapter 2, we determine the key operation in VQ as binary classification

$$c = \begin{cases} 2 & \text{if } \tilde{\mathrm{g}}(\mathbf{x}, \hat{\mathbf{x}}^1, \hat{\mathbf{x}}^2) \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

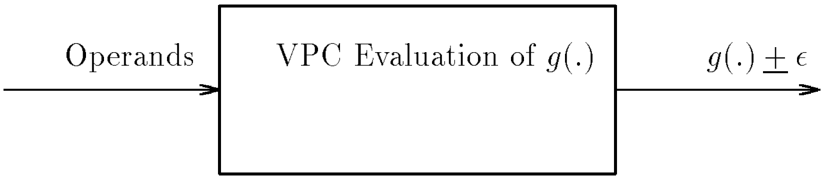


Figure 5.1: Variable-precision classifier (VPC) computes $g(.)$ and its error.

Let's define the discriminant $g^{i,h}$ which is output in binary fractional representation,

$$g^{i,h}(\mathbf{x}) \equiv \frac{\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)}{2^{\lceil \log_2(\max(|\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)|)) \rceil}} \tag{5.1}$$

That is, $g^{i,h}$ is the discriminant $\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)$ divided by the smallest power of two which normalizes the discriminant. ($|g^{i,h}| \leq 1$). Arithmetically, $\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)$ is evaluated as $g^{i,h}(\mathbf{x})$. Then, function $\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^i, \hat{\mathbf{x}}^h)$ is replaced by its binary representation $g^{i,h}(\mathbf{x})$,

$$c = \begin{cases} 2 & \text{if } g^{1,2}(\mathbf{x}) \geq 0 \\ 1 & \text{otherwise} \end{cases} \tag{5.2}$$

Figure 5.1 shows the key aspect of VPC: it calculates incrementally $g(.)$ *and* its accuracy. The evaluation stops when we determine the $\text{sign}(g^{1,2}(\mathbf{x}))$. This is possible when the error ϵ is smaller than $g(.)$. The average classification time may become significantly less than a full-precision classification.

Before analyzing VPC further, let's clarify the notion of precision. A signed fraction has j bits *precision* iff the absolute error between the real fraction and its representation is less than 2^{-j+1}. In other words, we have included the sign bit when counting the number's precision.

Definition 1 (δ) *The necessary precision, δ, is the lowest precision in evaluation such that the ignored operand bits do not affect the sign of the discriminant.*

The least possible necessary-precision is denoted by $\underline{\delta}$,

$$\underline{\delta} \triangleq \min(\delta) \tag{5.3}$$

In tailoring an architecture, this question arises: "What is the necessary precision to determine the output?" As we'll show, the answer depends on several factors such as: the function $g(.)$, the input vector $\mathbf{x}$, the discriminant's value, and the error interval ϵ.

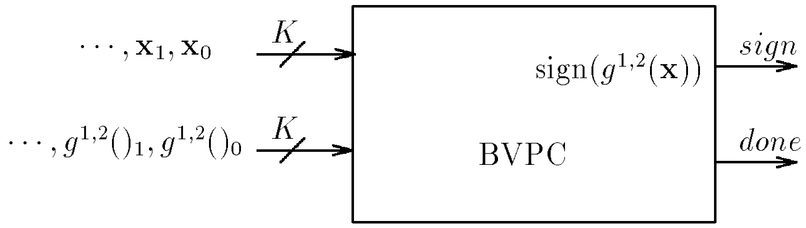


Figure 5.2: Binary variable-precision classifier (BVPC).

Figure 5.3 shows evaluation of (5.2) with BVPC. As illustrated in Figure 5.2, $\mathbf{x}$, and $g^{1,2}(.)$ arrive bit-serially, with the most significant bit first. First, BVPC evaluates $\tilde{g}(\mathbf{x}, \hat{\mathbf{x}}^1, \hat{\mathbf{x}}^2)$ using the MSB of the operands. Then, BVPC increments the precision. After δ cycles, it can terminate, delivering *sign*, and setting *done* flag. To find the necessary precision, BVPC increments the evaluation precision by one bit at a time. Later when implementing BVPC, we consider incrementing two or more bits per recurrence.

Figure 5.3 shows the content of the box shown in Figure 5.2. BVPC performs three steps in every iteration:

1. Increment the estimate's precision:

   $$w[j] \quad = \quad 2w[j-1] + G_j^{1,2}(\mathbf{x})$$

   where $w[j]$ is the estimate of the discriminant with j-bit precision vector elements:

   $$w[j] \equiv 2^j g^{1,2}(\mathbf{x}) \text{ with } j\text{-bit operands}$$

   and $G_j^{1,2}(.)$ is the function of j which increments the estimate's precision from $(j-1)$ to j bits. By doubling the estimate every iteration, we can often make $G_j^{1,2}(.)$ independent of j. For initial $\underline{\delta}$ iterations, since the output is zero, we skip steps (2) and (3).

2. Update the error interval. This accounts for increase in estimate's accuracy: The error interval is first set to $INITIALBOUNDS$. Subsequently, the recurrence updates the error interval. At the j-th iteration, $(\underline{\epsilon}[j], \overline{\epsilon}[j])$ bounds the error between the discriminant and its estimate:

   $$\underline{\epsilon}[j] < 2^j g^{1,2}(\mathbf{x}) - w[j] < \overline{\epsilon}[j]$$

   Equivalently,

   $$w[j] \quad \in \quad (2^j g^{1,2}(\mathbf{x}) + \underline{\epsilon}[j], 2^j g^{1,2}(\mathbf{x}) + \overline{\epsilon}[j])$$

3. Terminate if the estimate has the necessary precision. If $w[j] \in (\underline{\epsilon}[j], \overline{\epsilon}[j])$, the discriminant needs to be more precise. We go back to step 1 for another iteration. Otherwise, once $w[j]$ exits the ambiguity interval $(\underline{\epsilon}[j], \overline{\epsilon}[j])$, we can terminate. The precision necessary to make the discrimination unambiguous is then j $(\delta = j)$.

```
begin BVPC
```

$(\underline{\epsilon}[\underline{\delta}], \overline{\epsilon}[\underline{\delta}]) = INITIALBOUNDS;$

$w[-1] = 0;$

//Bit-serial recurrence

for $j = 0, 1, ..., n-1$

  $w[j] = 2w[j-1] + G_j^{1,2}(\mathbf{x});$

  if $j > \underline{\delta}$ then $(\underline{\epsilon}[j], \overline{\epsilon}[j]) = UPDATE(\underline{\epsilon}[j-1], \overline{\epsilon}[j-1], j);$

  if $j \geq \underline{\delta}$ then begin

$$sign = \begin{cases} +1 & \text{if } w[j] \geq \overline{\epsilon}[j] \\ -1 & \text{if } w[j] < \underline{\epsilon}[j] \\ U & \text{otherwise} \qquad\qquad //\text{U = Undetermined} \end{cases}$$

    if $(sign = -1$ or $sign = +1)$ then begin

$$c = \begin{cases} 2 & \text{if } sign = +1 \\ 1 & \text{otherwise} \end{cases}$$

      terminate BVPC;

    end_if

  end_if

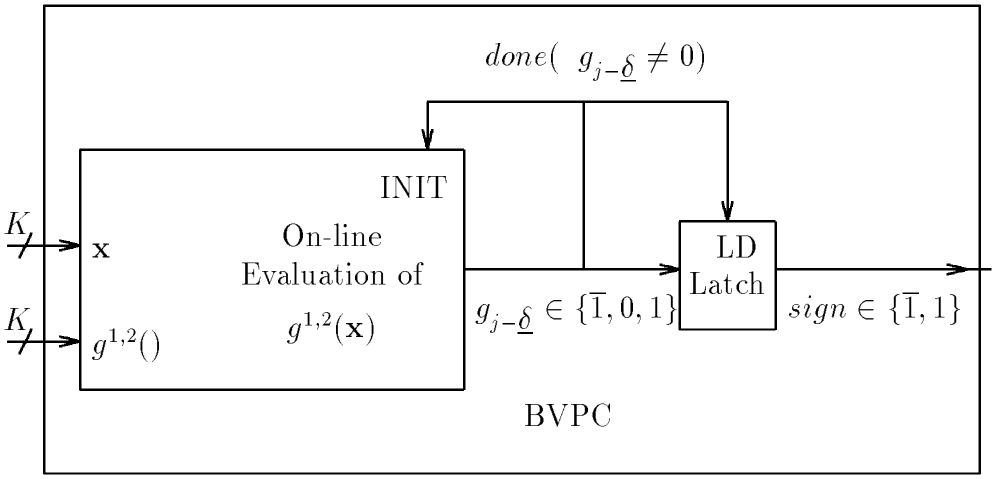
end_for

$$c = \begin{cases} 2 & \text{if } w[n-1] \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

end BVPC

Figure 5.3: Binary variable-precision classification algorithm.

Even with full precision n-bit evaluation of the operands, the discriminant in (5.2) may still be zero. We can decrease the number of ambiguous decisions by storing the vector elements at higher precision. Later in the chapter, a lemma will show that linear increase in precision results in exponential decrease in likelihood of ambiguities when discriminating. With few zero discriminants, we arbitrarily have mapped these source vectors into $c = 2$ as was shown in (5.2).



$$\overline{1}11\cdots 1 < 0$$
$$1\overline{1}\overline{1}\cdots\overline{1} > 0, \text{ where } \overline{1} \equiv -1$$

Figure 5.4: BVPC as an on-line evaluation.

5.3 Necessary Precision

We can analyze variable-precision arithmetic in the framework of on-line arithmetic [EL88]. BVPC is a special case of on-line arithmetic, where operands arrive bit-serially, one bit at a time with the most significant bit first. For the first ($\underline{\delta}$-1) recurrences nothing is output, so the output *sign* cannot be determined. Afterwards, at the iteration j, the (j-$\underline{\delta}$)-th MSB of $g_j^{1,2}(\mathbf{x})$ is represented with redundant binary Signed Bit (SB), where the bit $g_j \in$ {-1, 0, 1}. SB inherits the following property from weighted binary representation:

$$|g_j|2^{-j} > \sum_{i=j+1}^{n} |g_i|2^{-i} \tag{5.4}$$

In other words, the weight of a bit is larger than the combined weight of all less significant bits. Therefore, the most significant non-zero bit determines the sign: $g_j = +1$ implies $sign = +1$, and $g_j = -1$ implies $sign = -1$. As illustrated in Figure 5.4, BVPC terminates the on-line evaluation at the output of the first non-zero bit.

Duprat and *et. al.* [DHM89] constructed bounds for $\underline{\delta}$, and tabulated the results for on-line evaluation of common functions. For a single variable function with continuous derivative, they showed the following bound,

$$\underline{\delta}(f(x)) \leq \left\lceil \log_2 \max_{x\in[0,1]} \left| \frac{df(x)}{dx} \right| \right\rceil + 1 \tag{5.5}$$

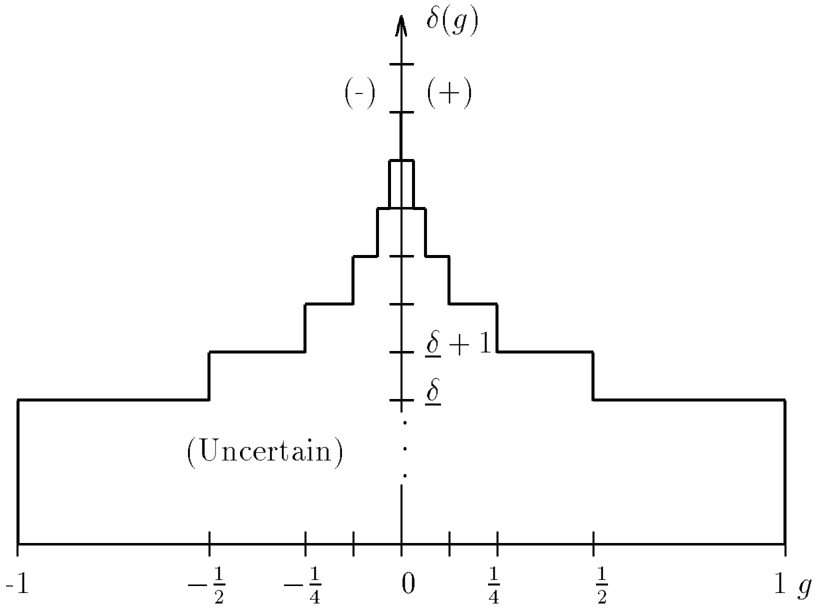


Figure 5.5: The operand precision (in bits) necessary to determine sign of the normalized discriminant g.

Given $\underline{\delta}$ of an on-line function, we can construct the error interval for BVPC. At the j-th iteration, the error interval is $(-2^{-j+\underline{\delta}}, 2^{-j+\underline{\delta}})$, $j \geq \underline{\delta}$. After evaluating $\underline{\delta}$ bits of the operands, every recurrence halves the error bounds with respect to the estimate. This is shown in Figure 5.5. The error interval can sidestep the update stage of the BVPC algorithm. Since the estimate is conveniently doubled in every recurrence, the error interval can be kept constant at (-1/2, 1/2). This resembles algorithms such as division and square root which scale remainders.

In Figure 5.6, we show three examples of variable-precision classifications. The estimates used in the three classifications are denoted by "+", "x", and "*". The collection of error intervals are normalized with respect to the initial interval and shown as $\delta(g)$. BVPC terminates as soon as the estimate comes up from the uncertain region under the curve $\delta(g)$. Even a simple function $\delta(g)$ will bound the average precision necessary for classification.

Theorem 1 *A function of the discriminant, $\delta(g)$ is an upper bound on the necessary precision δ:*

$$\delta \leq \delta(g) = \underline{\delta} + \lfloor -\log_2 \mid g \mid \rfloor \;\; \forall \; g$$

Proof There is an integer m such that,

$$2^{-m} \leq \mid g \mid \leq 2^{-m+1}$$

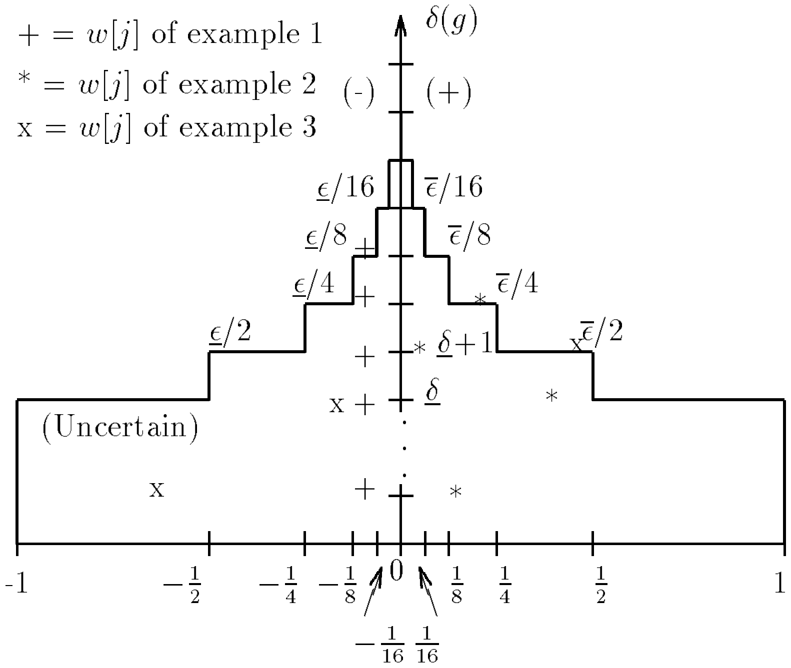


Figure 5.6: Instances of BVPC where $\delta(g)$ is aggregate of error intervals.

In words, we find an integer power of two which bounds g. Inverting the above equation:

$$m = \lfloor -\log_2(\mid g \mid) \rfloor$$

In SB representation, m has a specific meaning. The significant bits i, $i > m$ have a weighted sum less than 2^{-m}. For g to be greater than 2^{-m}, at least one of the most significant m bits of g has to be nonzero. As stated by (5.4), the most significant nonzero bit unambiguously specifies the sign. We then need at most m MSBs of the discriminant. We may need less bits, due to the multiple representations possible for g.

Given m MSBs of g, by the on-line property of Definition 5.3:

$$\begin{aligned} \delta &\leq \underline{\delta} + m \\ &\leq \underline{\delta} + \lfloor -\log_2 \mid g \mid \rfloor \end{aligned}$$

In other words, by evaluating $\delta + m$ bits of the operands, we evaluate m bits of discriminant. This upper bound is shown in Figure 5.5. □

Figure 5.7 puts the relationship among various variables into perspective. The lower bound is $\underline{\delta}$. The upper bound is $\delta(g)$, which itself it capped by n, the number of bits per codevector element. In other words,

$$1 \leq \underline{\delta} \leq \delta \leq \delta(g) \leq n$$

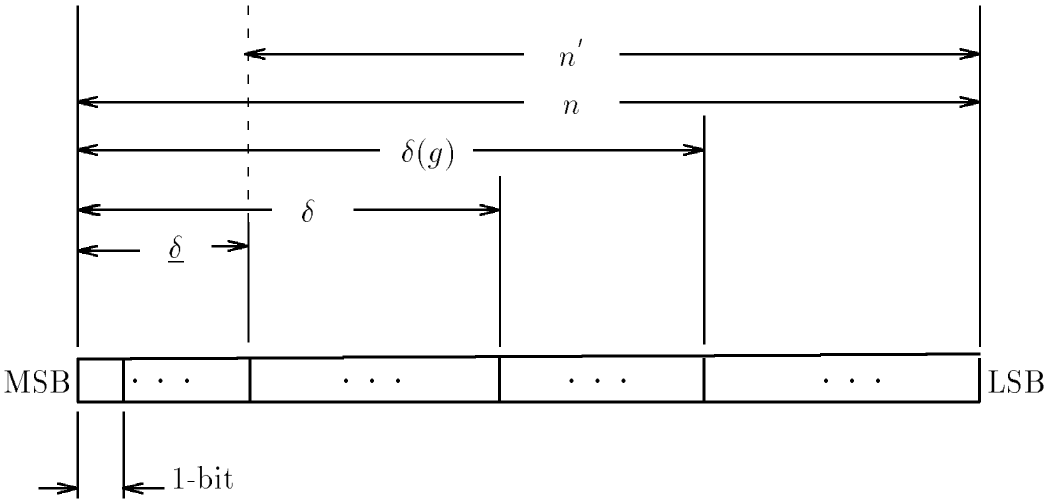


Figure 5.7: The relation among various notations.

As a property of on-line arithmetic, the maximum discriminant's precision n' is $\underline{\delta}$ less than the operand precision:

$$n' \triangleq n - \underline{\delta}$$

where the operands use nonredundant (e.g., 2's complement) representation and the discriminant uses SB representation.

5.4 Average Necessary-Precision

In binary VPC, average necessary-precision is usually independent of operand precision. The average necessary-precision or average precision for short depends on the statistical character of the discriminant $g(\mathbf{x})$. Let's define the probability density function (*pdf*) of $p(g)$ as

$$p(g) \triangleq \lim_{m \to \infty} p_m(g) 2^{-m} \tag{5.6}$$

In words, $p(g)$ represents the probability mass function $p_m(g)$ as m, the number of bits used in representation of g, becomes large. We then use E[.], the expectation operator to denote the average necessary-precision for $p(g)$. Average of $\delta(g)$, $E[\delta(g)]$ upper bounds the average necessary-precision.

First, let's show the average precision when the normalized discriminant is equally likely for any value. If g has uniform probability of occurrence, (i.e., $p(g) = 0.5$) then on the average less than ($\underline{\delta}$+ 1) bits of the operands are sufficient regardless of the operand precision. As seen in Figure 5.5, in half of the undetermined interval, a one bit increase in the estimate's resolution determines the sign of g. Equivalently, the probability of evaluating operand bits decreases exponentially with their significance. As the operand precision increases, the sum of this geometric series approaches $\underline{\delta}+1$ $(= \underline{\delta} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots)$.

To analyze the average precision, we have to answer: "What is the density of $p(g)$ near zero?" In particular, as g approaches zero, $p(g)$ weights more on the average precision. The following states the key condition necessary for a bound on the average precision:

Definition 2 (non-singularity) *The pdf $p(g)$ is non-singular iff $p(g)$ is finite in a finite neighborhood of zero:*

$$p(g)\ \textit{non-singular} \iff \textit{finite}\ Q,\ \ Q \equiv \max_{g\in(-2^{-n},2^{-n})} p(g)$$

where n is a nonnegative integer. In other words, Q is the least upper bound of $p(g)$, for values of g small enough to need more than n bits to be distinguished from zero.

Discriminants, which we have encountered, have non-singular *pdf*. These include metrics with sum, multiply, and absolute difference operations. The vector $\{\hat{\mathbf{x}}\}$ and the vectors $\{\mathbf{x}\}$ when generated by Gauss-Markov sources have finite *pdf*. Term by term multiplication, or the absolute difference between two such vectors, would not create impulses in their *pdf*. Since the resulting *pdf* is the convolution of the probability density functions of these elements, summing the elements of the resulting vector also would not create impulses in *pdf* of g. Consequently, the inner product between these vectors would not result in impulses in the function $p(g)$. Using similar arguments, the $p(g)$ of the L_1 distance (sum of absolute difference) is non-singular.

Our theoretical result can be better assessed in the context of theory of Information Based Complexity (IBC) [TW91]. IBC evaluates the effect of precision of numbers on the complexity of computations. In IBC, the worst case complexity is the longest computation time of all the instances of the computations. For classification with metrics which use multiplication and addition, the complexity grows as square of precision. The average case complexity, on the other hand is independent of the operand precision and the discriminant function. This runs counter to typical observations in IBC analysis, where the computation becomes intractable as the precision increases.

Theorem 2 *If $p(g)$ is non-singular in a finite neighborhood of zero, then average case complexity of classification with respect to precision n is $O(1)$.*

Proof Let's calculate the average precision necessary for classification, $\mathrm{E}[\,\delta(g)]$. As Figure 5.8 demarcates, we evaluate the summation $\mathrm{E}[\,\delta(g)]$ in two parts: experimentally we can determine the average precision necessary for a finite n. This will be denoted as $\mathrm{E}[\delta_n(g)]$ $(\mathrm{E}[\delta_n(g)] \leq n)$. The discriminant precision n' $(= n+\underline{\delta})$ will not be sufficient to classify when

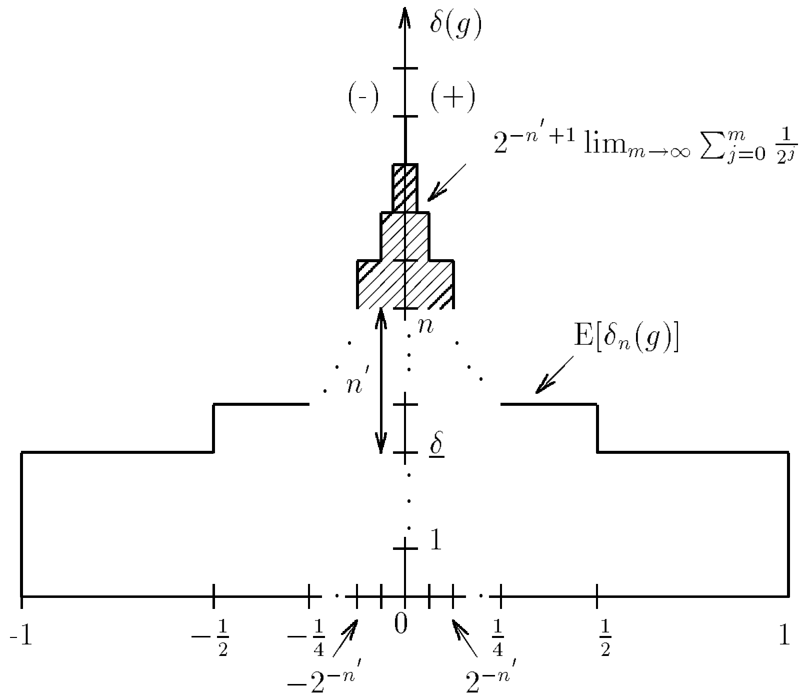


Figure 5.8: $\mathrm{E}[\delta(g)]$ evaluated as $\mathrm{E}[\delta_n(g)]$ plus an infinite geometric series.

the discriminant is within the interval $[-\frac{1}{2^{n'}}, \frac{1}{2^{n'}}]$. Let this part be bounded by an infinite geometric summation. Using $[-o, o]$ as a short hand for $[-\frac{1}{2^{n'}}, \frac{1}{2^{n'}}]$,

$$\begin{aligned}
\mathrm{E}[\,\delta(g)] &= \lim_{n\to\infty}\sum_{g=-1}^{1}\delta(g)p_n(g) \\
&= \sum_{g=-1}^{-o}\delta_n(g)p_n(g) + \sum_{g=o}^{1}\delta_n(g)p_n(g) + \lim_{m\to\infty}\sum_{g=-o}^{o}\delta_m(g)p_m(g) \\
&= \mathrm{E}[\delta_n(g)] + \lim_{m\to\infty}\sum_{g=-o}^{o}p_m(g)(\delta(g)-n) \\
&= \mathrm{E}[\delta_n(g)] + \lim_{m\to\infty}\sum_{g=-o}^{o}p(g)2^{-m}(\delta(g)-n) \\
&= \mathrm{E}[\delta_n(g)] + \int_{g=-o}^{o}p(g)(\delta(g)-n)dg
\end{aligned}$$

If $p(g),\ g\in[-\frac{1}{2^{n'}}, \frac{1}{2^{n'}}]$ is non-singular then the upper bound on $\mathrm{E}[\,\delta(g)]$ is

$$\mathrm{E}[\,\delta(g)] \;\le\; \mathrm{E}[\delta_n(g)] + Q\int_{g=-o}^{o}(\delta(g)-n)dg$$

$$\leq \quad \mathrm{E}[\delta_n(g)] + 2Q \int_{g=0}^{o} (\delta(g) - n) dg$$

As shown in Figure 5.8, the integration of $\delta(g)$ with respect to the domain is a simple summation along the range. Changing the variable of summation to j:

$$\begin{aligned}
\mathrm{E}[\, \delta(g)] &\leq \mathrm{E}[\delta_n(g)] + 2Qo(\overbrace{\lim_{m\to\infty} \sum_{j=0}^{m} \frac{1}{2^j}}^{=2}) \\
\mathrm{E}[\, \delta(g)] &\leq \mathrm{E}[\delta_n(g)] + \frac{1}{2^{\underline{\delta}+n}} 4Q
\end{aligned}$$

□

Lemma 1 *If $p(g)$ is continuous then a simulation with sufficiently large operand precision, n, can be extrapolated to the limit of $n \to \infty$.*

Let $p(g)$ be continuous iff it can be approximated by its mean μ arbitrarily well in a finite neighborhood $g \in [-\frac{1}{2^{n'}}, \frac{1}{2^{n'}}]$. Basing the $p(g)$ on Monte-Carlo simulations with metric evaluations, mean μ of the neighborhood becomes

$$\mu \quad = \quad \frac{p(\ n\text{-bit eval.})}{2^{-n'+1}}$$

where $p(\ n$-bit eval.), is the probability of n-bit evaluation of the operands. And $2^{-n'+1}$ is the width of the neighborhood.

If $p(g)$ is continuous, then for a sufficiently large n, we can replace Q in the theorem's upper bound for μ. This will result in an approximate value for the average precision given infinite-precision operands:

$$\begin{aligned}
\mathrm{E}[\, \delta(g)] &\leq \mathrm{E}[\delta_n(g)] + 2^{-n'+2} \times Q \\
&\approx \mathrm{E}[\delta_n(g)] + 2^{-n'+2} \times \mu \\
&\approx \mathrm{E}[\delta_n(g)] + 2^{-n'+2} \times \frac{p(\ n\text{-bit eval.})}{2^{-n'+1}} \\
&\approx \mathrm{E}[\delta_n(g)] + 2p(\ n\text{-bit eval.})
\end{aligned}$$

□

To find a sufficiently large n' ($= n + \underline{\delta}$), operand precision n is incremented in the simulations until the probability of using n-bit precision halves at each increment. This indicates that for the neighborhood $[-\frac{1}{2^{n'}}, \frac{1}{2^{n'}}]$, g is sufficiently small for the discriminant's distribution $p(g)$ to be approximated well by its mean, μ.

With finite evaluation precision, ambiguous decisions cause higher error rate in classification, and produce more distortion in vector quantization. However, as the next two lemma encapsulate, these adverse affects decrease exponentially with respect to operand precision n.

Lemma 2 *In classification with m-bit operands and sufficiently large n ($m > n$), probability of an arbitrary decision follows this relation*

$$p(g^{1,2}(\mathbf{x}) = 0) \ \propto \ 2^{m-n}$$

With an experimentally measured n, $p(g)$ can be approximated by μ for $g \in [-\frac{1}{2^{n'}}, \frac{1}{2^{n'}}]$. The additional ($m$-$n$) precision will decrease the occurrence of ambiguous decisions by 2^{m-n}.

□

Lemma 3 *In vector quantization with m-bit operands and a sufficiently large n ($m > n$), the distortion becomes*

$$\mathrm{E}[(\mathbf{x} - \hat{\mathbf{x}}^c|_n)^2] - \mathrm{E}[(\mathbf{x} - \hat{\mathbf{x}}^c|_m)^2] \ \propto \ 2^{m-n}$$

where $(\mathbf{x} - \hat{\mathbf{x}}^c|_n)$ is the quantization distortion with n-bit operands.

After increasing the precision beyond n-bit, the *pdf* becomes uniform. Each additional bit of precision then can halve the distortion. □

5.5 M-ary VPC

We now extend binary VPC to M-ary VPC. From definition of VQ in Chapter 2, M-ary classification is

$$c = \max_{i=1,\ldots,M}{}^{-1} s(\mathbf{x}, \hat{\mathbf{x}}^i) \tag{5.7}$$

We can find the best codevector c by $(M-1)$ binary classifications. Each binary classification would determine that one codevector is a better replacement than another one. This is akin to removing the worst of the two from the list of codevectors contending for the maximum. With $(M-1)$ binary classifications, $(M-1)$ codevectors would be eliminated from the list. There will be only one codevector remaining. This will be the one most similar to the source vector.

Figure 5.9 shows M-ary VPC (MVPC). MVPC uses variable-precision evaluation similar to BVPC, except the estimate is generalized:

$$w[j] \equiv 2^j(s(\mathbf{x}, \hat{\mathbf{x}})) \text{ with } j\text{-bit evaluation}$$

Also, $G_j^{1,2}(\mathbf{x})$ is replaced by $G_j(\mathbf{x}, \hat{\mathbf{x}}^i)$:

$$w[j] \equiv 2w[j-1] + G_j(\mathbf{x}, \hat{\mathbf{x}})$$

In the outer loop, c is the index of the codevector with the highest similarity measure at any iteration of i. The i-th iteration, $i \in \{2, ..., M\}$, chooses between the vector $\hat{\mathbf{x}}^i$ and the vector $\hat{\mathbf{x}}^c$. Finally after $(M-1)$ iterations, the index for best codevector is c.

To reduce computation, $s(\mathbf{x}, \hat{\mathbf{x}}^c)$ which is the highest similarity measured at any iteration of i, is stored as $max[m]$, its precision is stored as m, and its error bounds are stored as $\underline{\varepsilon}$ and $\overline{\varepsilon}$:

$$\underline{\varepsilon}[m] < 2^m s(\mathbf{x}, \hat{\mathbf{x}}^c) - max[m] < \overline{\varepsilon}[m]$$

In the inner loop, which follows the BVPC Algorithm, the notation is identical: The variables are indexed by the iteration variable j. In the outer loop, for clarity, only the transition of the variables from one iteration to the other are marked: The variable after the execution of the assignment has " ' " superscript. Since M-ary VPC is based on binary VPC, it also asymptotically requires a constant average precision independent of operand precision.

Theorem 3 *If $p(s(\mathbf{x}, \hat{\mathbf{x}}))\forall(\mathbf{x}, \hat{\mathbf{x}})$ is non-singular, then the average case complexity of M-ary classification is $O(1)$ with respect to precision n.*

Proof Using formulation (5.2) for classification between $\hat{\mathbf{x}}^i$ and $\hat{\mathbf{x}}^h$:

$$c = \begin{cases} h & \text{if } g^{i,h}(\mathbf{x}) \geq 0 \\ i & \text{otherwise} \end{cases}$$

where $g^{i,h}(\mathbf{x}) = s(\mathbf{x}, \hat{\mathbf{x}}^h) - s(\mathbf{x}, \hat{\mathbf{x}}^i)$. From Theorem 2, the average precision for binary classification between $\hat{\mathbf{x}}^i$ and $\hat{\mathbf{x}}^h$ is bounded, if $p(g^{i,h})$ is non-singular. $p(g^{i,h})$ is non-singular if singularities of $p(s(\mathbf{x}, \hat{\mathbf{x}}^i))$ and $p(s(\mathbf{x}, \hat{\mathbf{x}}^h))$ $(i \neq h)$ are finite distance apart.

The average precision for M-ary classification, $\mathrm{E}[\delta(g)]$ is

$$\mathrm{E}[\delta(g)] = \frac{1}{M-1} \sum_{i=2}^{M} \left(\mathrm{E}[\delta(g^{i,h})], \ \forall h \in 1, ..., i-1 \right)$$

In words, it is the average precision in all the required binary classifications averaged over $(M-1)$ classifications. Consequently, finite $\mathrm{E}[\delta(g^{i,h})] \Rightarrow$ finite $\mathrm{E}[\delta(g)]$. (For $\mathrm{E}[\delta(g^{i,h})]$ to be finite, it is sufficient if $p(s(\mathbf{x}, \hat{\mathbf{x}}^i))$ is non-singular $\forall i$.) $\square$

begin MVPC
$m = \underline{\delta}$;
$max[\underline{\delta}] = \sum_{j=0}^{\underline{\delta}} 2^{\underline{\delta}-j} G_j(\mathbf{x}, \hat{\mathbf{x}}^1)$;
$c = 1$;
$(\underline{\varepsilon}[\underline{\delta}], \overline{\varepsilon}[\underline{\delta}]) = INITIALBOUNDS$;
for $i = 2, 3, ..., M$
    $sign = U$;
    $(\underline{\epsilon}[\underline{\delta}], \overline{\epsilon}[\underline{\delta}]) = INITIALBOUNDS$;
    $w[-1] = 0$;
    for $j = 0, 1, ..., n-1$ //Bit-serial recurrence
        $w[j] = 2w[j-1] + G_j(\mathbf{x}, \hat{\mathbf{x}}^i)$;
        if ($j > m$) then begin //Inc precision of max[.]
            $(\underline{\varepsilon}[m+1], \overline{\varepsilon}[m+1]) = UPDATE(\underline{\varepsilon}[m], \overline{\varepsilon}[m], j)$;
            $max[m+1] = 2max[m] + G_{m+1}(\mathbf{x}, \hat{\mathbf{x}}^c)$;
            $m' = m + 1$;
        end_if
        if $j > \underline{\delta}$ then $(\underline{\epsilon}[j], \overline{\epsilon}[j]) = UPDATE(\underline{\epsilon}[j-1], \overline{\epsilon}[j-1], j)$;
        if $j \geq \underline{\delta}$ then begin

$$sign = \begin{cases} +1 & \text{if } w[j] - \overline{\epsilon}[j] \geq 2^{m-j}(max[m] - \underline{\varepsilon}[j]); \\ -1 & \text{if } w[j] - \underline{\epsilon}[j] < 2^{m-j}(max[m] - \overline{\varepsilon}[j]); \\ U & \text{otherwise;} \end{cases}$$

            if ($sign = -1$ or $sign = +1$) then terminate inner loop;
        end_if
    end_for
    if $sign = +1$ or_if $w[n-1] \geq max[n-1]$ then begin //Replace maximum
        $max'[j] = w[j]$;
        $m' = j$;
        $c' = i$;
        $(\underline{\varepsilon}[j], \overline{\varepsilon}[j])' = (\underline{\epsilon}[j], \overline{\epsilon}[j])$;
    end_if
end_for
end MVPC

Figure 5.9: M-ary Variable-Precision Classification Algorithm

5.6 Conclusion

This chapter developed Variable-Precision Classification. VPC development fits in the context of information based complexity (IBC). Unlike many other algorithms, the average case complexity of VPC is independent of operand precision. Based on this, we presented a general algorithm applicable with many metrics. First, we described binary VPC algorithm. It bit-serially evaluates the discriminant to the necessary precision. Then, we derived an asymptotically constant bound on average precision. This demonstrated that as the operand precision increases, the average precision becomes independent of it. The bound holds in practice, since it only requires the discriminant *pdf* to be continuous around zero. Finally, the binary VPC algorithm was generalized to M-ary VPC, and we demonstrated that the computational complexity of M-ary classification is also independent of precision.

CHAPTER 6

Vector Quantization with VPC

6.1 Introduction

Given the encouraging theoretical results from the previous chapter, we now assess variable-precision classification (VPC) for vector quantization (VQ). As noted in Chapter 2, VQ based compression represents a block of K pixels with the index of best matching codevector. Specifically, VQ classifies an input block $\mathbf{x}$ into one of N codevectors $\mathbf{y}$, where the number of codevectors is large. For classification, it uses inner product. There have been several inner product modules implemented for VQ [DB87, MJBD92, FCS90, KYJ93, DCG88], and for pattern recognition[NJ85]. Inner product is a popular metric for classification.

The primary application of VPC is reducing the average evaluation time of VQ. Some of the best VQ search algorithms require variable number of classifications per input. VPC becomes attractive since the encompassing computation also takes variable time. For example, in PTSVQ [CLG88], searching the unbalanced tree requires variable number of classifications per source vector. This can mask the variable execution time of VPC. VPC can also be used when the classification latency can vary. For example, in off-line compression, or even in decoding in a packet switched network where the primary criteria is the average packet transmission latency.

Another application is reducing the memory-processor traffic. Currently, the processor chip uses different technology than the memory chip. These two different chips are often housed in different packages. For every inner product we need to fetch one codevector which is Kn bits. This results in extensive and costly inter-chip traffic.

In the next section, we show Binary VPC (BVPC) with inner product similarity measure. Then in Section 6.5, we measure and demonstrate the asymptotic bound on average precision for classification. We evaluate the performance for classification of auto-regressive sequences, followed by vector quantization (VQ) of real images. Finally, we present a VQ architecture using VPC, and discuss pipelining and digit-serial evaluation. In comparison to VPR and conventional approaches, this architecture reduces the total evaluation time and memory-processor bandwidth.

6.2 Binary VPC with Inner Product

In this section, we describe the variable-precision evaluation of the inner product discriminant g:

$$g^{i,h}(\mathbf{x}) = \mathbf{x}^t \mathbf{y}^{i,h} + b^{i,h}$$

After briefly covering the theoretically optimal architecture, we described a scheme more amenable to implementation.

For most benefit from variable-precision evaluation, both operands $\mathbf{x}$ and $\mathbf{y}$ can be evaluated with variable-precision. This would be theoretically optimal yielding $O(1)$ computational complexity with respect to the operand precision. The architecture would result in minimum number of full adders used for each classification. Basically, it would use serial-serial multipliers, where we incrementally increase the precision of both $\mathbf{x}$ and $\mathbf{y}$ operands. Despite its optimality, it has drawbacks. The throughput of serial-serial multipliers is notoriously low. Moreover, even though it evaluates bit-serially, it still requires registers in which to store all the bits of codevector $\mathbf{y}$ and $\mathbf{x}$ which have been so far received. To increase the throughput, we could use digit-serial processing with d digits, effectively evaluating $n \times n$ multiplication, as $m = n/d$, $d \times d$ sub-multiplications. Even so for evaluating i-th digit, we would need $(2i - 1)$ cycles. Aside from the throughput, there is an even more important drawback. If the codebook-processor bus has limited bandwidth, there is a mismatch between processing and memory-processor data transfer.

Alternatively, only $\mathbf{y}$ can be evaluated with variable-precision. This would optimize the memory-processor bus. Figure 6.1 shows the processor's inputs and its interface to memory. The source vector $\mathbf{x}$ is stored on-chip, and is evaluated at fixed-precision. For every bit of the codevector element received, it takes one cycle to generate one full-precision partial term. The architecture iterates on the elements of the vector $\mathbf{y}$ until the necessary precision is reached. This actually simplifies the controller since the processor implicitly takes on the task of buffering the source vector. On the other hand, variable-precision evaluation of only $\mathbf{y}$ changes the complexity of classification to $O(n)$.

Both approaches are variations on partitioning of a multiplier. The first design partition the multiplication into $d \times d$ sub-multiplications, where terms evaluated at a given step form a square tile of the multiplication. The second design partitions the multiplication into partial term evaluation, where each of these $1 \times d$ sub-multiplications form a thin slab of the multiplication.

Figure 6.2 presents the Binary VPC algorithm, the center piece of this chapter. The Linear-BVPC algorithm shown next is based on the Binary VPC (BVPC) algorithm presented in the previous chapter, where in each iteration one more bit of the codevector element

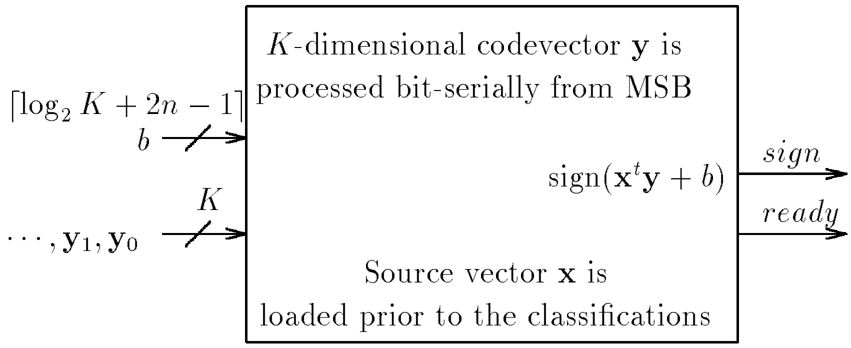


Figure 6.1: Minimized memory bandwidth for BVPC with inner product.

is evaluated. The incremental evaluation of the function $g^{i,h}(\mathbf{x})$ is

$$G_j^{i,h}(\mathbf{x}) = \begin{cases} -\sum_{k=1}^{K} y_{k,0}^{i,h} x_k + b^{i,h} & j = 0 \\ \sum_{k=1}^{K} y_{k,j}^{i,h} x_k & otherwise \end{cases} \tag{6.1}$$

where $y_{k,j}^{i,h}$ is the codevector's k-th element's j-th bit; and $b^{i,h}$ is the bias term for the (i,h) hyperplane. In other words to get the initial estimate $w[0]$, $G_j^{i,h}$ sums partial terms of the sign bits. In subsequent iterations, it sums partial terms of the next significant bits of the codevector.

6.3 Sign Selection

In the previous chapter, Figure 5.5 showed the sign selection function's outlook of the carry-saved estimate. It has three regions: positive, negative, and the interim uncertain region surrounding zero. As it inspects more bits of the estimate, the undecided region exponentially shrinks. As soon as the detector finds a non-zero digit, it outputs the sign and signals to terminate the evaluation. A sign selection function has as input the redundant estimate of the inner product, and it outputs the sign in the earliest iteration possible.

In this section, we will describe three sign selection functions. All the three error intervals are based on this inequality:

$$\begin{aligned} g(\mathbf{x}) &= |\mathbf{x}^t\mathbf{y}| \\ &= |x_1y_1 + x_2y_2 + ... + x_Ky_K| \\ &\leq |x_1||y_1| + |x_2|||y_2|| + ... + |x_K|||y_K| \end{aligned} \tag{6.2}$$

A stack of the error intervals, one for every step of the estimate refinement, forms the sign selection function. First, we'll show Interval A. It is an interval derived using on-line

begin LINEAR-BVPC
$(\underline{\epsilon}, \overline{\epsilon}) = INITIALBOUNDS;$
for $j = 0, 1, ..., n - e - 1$ //Bit-serial recurrence
    if $j = 0$ then
        $w[0] = G_0^{i,h}(\mathbf{x});$
    else
        $w[j] = 2w[j-1] + G_j^{i,h}(\mathbf{x});$

$$sign = \begin{cases} +1 & \text{if } w[j] \geq \overline{\epsilon} \\ -1 & \text{if } w[j] < \underline{\epsilon} \\ U & \text{otherwise} \qquad // \text{ U = Undetermined} \end{cases}$$

    if $sign = +1$ or $sign = -1$ then terminate loop;
end_for
if $sign = U$ then

$$sign = \begin{cases} +1 & \text{if } w[n-e-1] \geq 0 \\ -1 & \text{if } w[n-e-1] < 0 \end{cases}$$

$$c = \begin{cases} 2 & \text{if } sign = +1 \\ 1 & \text{otherwise.} \end{cases}$$

end LINEAR-BVPC

Figure 6.2: Binary VPC with inner product similarity measure.

methodology. Then, we will show intervals B and C. They are designed specifically for variable-precision classification of inner product.

Interval A

The simplest interval to evaluate, Interval A, replaces x_i and y_i in (6.2) with $(1-2^{n-1})$. It results in the maximum magnitude of the inner product of two K-dimensional vectors with n-bit elements:

$$|\mathbf{x}^t\mathbf{y}| \;\leq\; K(1-2^{-n+1})(1-2^{-n+1}) \tag{6.3}$$

The bounds on Interval A are only dependent on j, the number of MSBs of $\mathbf{y}$ evaluated:

$$(|\mathbf{x}^t\mathbf{y}|_{n\text{-bit}} - |\mathbf{x}^t\mathbf{y}|_{j\text{-bit}}) < K(1-2^{-n+1})(2^{-j}-2^{-n+1})$$

We enlarge Interval A, to simplify updating it:

$$|\mathbf{x}^t\mathbf{y}| - (|\mathbf{x}^t\mathbf{y}|\text{with } j\text{-bit operands}) < K(1-2^{-n+1})2^{-j}$$

As seen in the above equation, the interval for the iteration $(j+1)$ is half of the interval for the iteration j. This is achieved by doubling of the estimate in every iteration. As a result Interval A does not need updating:

$$INITIALBOUNDS = [-K(1-2^{-n+1}), K(1-2^{-n+1})]$$

As we will similarly see next with Interval B, any interval which is independent of the variable-precision operands, neatly avoids the update step. Since the intervals would all be shrunk with the same factor, the ratio of succeeding intervals remains a factor of two.

Interval B

Interval B starts by replacing the maximum sum of elements of y_i shown in (6.2) with the actual summation:

$$|\mathbf{x}^t\mathbf{y}| \;\leq\; \max(|y_1|, |y_2|, ..., |y_K|)\sum_{k=1}^{K}|x_k| \tag{6.4}$$

In general, this is an instance of Holder's inequality with $q \to \infty$:

$$|\mathbf{x}^t\mathbf{y}| \leq \|\mathbf{x}\|_p\|\mathbf{y}\|_q;\; 1 < p, q \leq \infty;\; 1/p + 1/q = 1$$

This interval is much smaller than interval A. Instead of maximum value of elements, it uses the summation of elements. For example, with uniformly distributed elements of $\mathbf{x}$, it is half the size of Interval A. The difference between these two interval increases after evaluating the partial terms which depend of the sign of $\mathbf{y}$. The inner product's upper and lower bounds are different than formulated in (6.4):

$$\max_{k=1,\ldots,K}(|y_k|)\sum_{k=1}^{K}\min(x_k,0) \;<\; \mathbf{x}^t\mathbf{y} \;<\; \max_{k=1,\ldots,K}(|y_k|)\sum_{k=1}^{K}\max(x_k,0)$$

In other words after the first cycle, a partial term will be positive iff the corresponding element of $\mathbf{y}$ is positive, and otherwise the partial term will be negative. Now, we can evaluate each bound separately. The elements of $\mathbf{x}$ are summed into two groups. One group is the summation of positive elements, the other of negative elements. For a uniformly distributed element probability, Interval B becomes one fourth the size of Interval A.

The upper bound is the sum of the all the positive $\mathbf{x}$ elements multiplied by the $\mathbf{y}$ element largest in magnitude. The lower bound is the sum of all the negative elements of $\mathbf{x}$ multiplied by the $\mathbf{y}$'s element largest in magnitude. Both summations are computed prior to the classifications of the source vector. To simplify evaluating the initial interval and updating it, we substitute the maximization by 1, which is slightly greater than the maximum possible magnitude of y_k (i.e., 2^{-n+1}). This simplifies the bounds' update. The upper and lower error bounds become

$$\sum_{k=1}^{K}\min(x_k,0) \;<\; \mathbf{x}^t\mathbf{y} \;<\; \sum_{k=1}^{K}\max(x_k,0)$$

where $\max(x_k,0)$ is an unary operator which substitutes 0 for all the negative elements and outputs the positive elements unaffected. Similarly, $\min(x_k,0)$ permits selection of all the elements which are less than zero. This is used as a convenient means of showing summation of negative elements.

With $j+1$ bits of $\mathbf{y}$ evaluated, the error is bounded by

$$2^{1-j}\sum_{k=1}^{K}\min(x_k,0) \;<\; \mathbf{x}^t\mathbf{y}_{n\text{-bit}} - \mathbf{x}^t\mathbf{y}_{j\text{-bit}} \;<\; 2^{1-j}\sum_{k=1}^{K}\max(x_k,0)$$

Similar to Interval A, Interval B does not require updating of the initial bounds,

$$INITIALBOUNDS = (\sum_{k=1}^{K}\min(x_k,0), \sum_{k=1}^{K}\max(x_k,0))$$

Interval C

Interval C swaps $\mathbf{x}$ and $\mathbf{y}$ in Interval B. Interval C replaces x_i in (6.2) with $\max(|x_i|)$:

$$|\mathbf{x}^t\mathbf{y}| \;\le\; \max(\,x_1,\; x_2,\; \ldots, x_K)\sum_{k=1}^{K}|y_k| \tag{6.5}$$

In words, it swaps $\mathbf{x}$ and $\mathbf{y}$ in (6.4). Shown variations of bound C, use different information about codevectors $\mathbf{y}$ to upper bound (6.5). We discuss it for comparative purposes only. Even when looking at its variations, Interval C is more costly than Interval B and does not perform as well. It can, however, give some insight.

In the simplest version, by inspecting $\mathbf{y}$, it replaces K in the Interval A by an equal or smaller integer, L. L is the maximum number of bits of the same significance which are '1' across all elements of $\mathbf{y}$. Since the maximization is among codevectors' elements, it can be precomputed and stored.

As evaluation progresses, L could potentially decrease. We can also narrow Interval C by updating L in every iteration to exclude the bits already evaluated. This, however, complicates the updating and further increases the bandwidth. We could further narrow the bound by making it a function of received vector $\mathbf{x}$. The initial bound, however, would require a multiplication.

We believe Interval B is preferable to Interval C. When $\mathbf{x}$ and $\mathbf{y}$ are statistically similar, Interval B which is the actual evaluation of the summation, bounds the inner product tighter than Interval C. In terms of hardware, Interval B requires only two accumulators and one additional comparator. Instead of these, Interval C needs more storage and bandwidth for the codebook. When a codebook is off the chip, then there is also additional off-chip traffic to bring in the Interval and its updates. This can be prohibitive.

Example of Binary VPC

The example shown in Table 6.1 and 6.2 uses BVPC to classify between $\hat{\mathbf{x}}^1$ and $\hat{\mathbf{x}}^2$ of the example previously shown in Section 2.5. The precision n of a two's complement element is eight bits ($n = 8$). The inner product of four elements requires 18 bits ($\log_2(4 \times 2^8 \times 2^7) + 1)$), which includes one bit for the addition of the bias b.

Table 6.1 illustrates the variable-precision classification with Interval B, where $\mathbf{x} = (-5,-4,4,5)^t$, and $\mathbf{y} = (-100,-10,10,100)^t$. As shown the first two iterations would suffice for the classification, even though all eight iterations are shown. Bits of the multiplier are shown in two's complement binary system. For ease of display, all other values, which were fractional, are scaled by a factor of 128 and are shown as decimal integers. The scaling does not affect any other aspect of the algorithm. The Table 6.2 shows the same classification with the intervals B and C. In this example, Intervals A, B and C require five, two and five iterations respectively. In the following section, we characterize BVPC using VQ of images.

Table 6.1: Example of BVPC with $\mathbf{x}$ = (-5,-4, 4, 5)^t, and $\mathbf{y}$ = (-100,-10,10,100)^t.

j	Multiplying Bits $(y_1, y_2, y_3, y_4)_j$	Added Partial Product	Added b Term	Estimate ($w[j]$)	Interval B	Sign
0	(1,1,0,0)	+9x128	-10100	-8948	—,—	U
1	(0,1,0,1)	+1x64	0	-8884	-576,576	-1
2	(0,1,0,1)	+1x32	0	-8852	-288,288	
3	(1,1,0,0)	-9x16	0	-8896	-144,144	
4	(1,0,1,0)	-1x8	0	-9004	-72,72	
5	(1,1,0,1)	-4x4	0	-9020	-36,36	
6	(0,1,1,0)	+0	0	-9020	-18,18	
7	(0,0,0,0)	+0	0	-9020	-9,9	

Table 6.2: Iterations of example classification with Interval A, B, and C.

j	Estimate $w[j]$	Interval A	Sign	Interval B	Sign	Interval C	Sign
0	-8948	—	U	—,—	U	—	U
1	-8884	65024	U	-576,576	-1	48768	U
2	-8852	32256	U	-288,288		24192	U
3	-8996	15872	U	-144,144		11904	U
4	-9004	7680	-1	-72,72		5760	-1
5	-9020	3584		-36,36		2688	
6	-9020	1536		-18,18		1152	
7	-9020	512		-9,9		384	

6.4 M-ary VPC

For completeness, we will show M-ary variable-precision classification with inner product metric. It is based on the general M-ary VPC presented in Figure 5.9, where the similarity measure has been set to inner product. The algorithm 6.3 has been modified to avoid intermittent increasing of the maximum's precision. Variable-precision evaluation of the maximum would decrease the average precision, but it would make pipelining difficult. In middle of an inner product, we might have to break the current pipelined evaluation to evaluate one more bit of the codevector which yielded the current maximum. Alternatively, when the inner product is replacing the current maximum we evaluate it to full precision. Then, for sign selection, the current maximum is subtracted from the estimate and the result compared against the interval.

M-ary evaluation is based on Equation (2.15) in Chapter 2. We incrementally refine the estimate $w[j]$ of $(\mathbf{x}^t\mathbf{y}^i + b_i)$. After each refinement, we compare it against the current maximum. When the sign of the difference is unequivocally negative or it is unequivocally positive but the inner product is with the last codevector $\mathbf{y}^i$, we terminate evaluation. M-ary classification's performance is comparable to binary classification. The bias term b_i reduces the number of new maximum's and hence the number of full evaluations.

6.5 Statistical Analysis

We measured the sensitivity of BVPC to variance, correlation, and vector dimension using Auto-Regressive (AR) sequences. As discussed in Chapter 3, AR sequences are made of correlated random variables which have identical PDF. Each random variable in the sequence gets a fraction of the previous element summed with an independent Normal source, with zero mean and σ_n^2 variance. AR sequences synthesize the PDF and the correlation of sample sequences of natural sounds and images:

$$\begin{aligned} x_1 &= N(\mu = 0, \sigma_n^2 = \sigma_x^2) \\ x_k &= \rho x_{k-1} + N(\mu = 0, \sigma_n^2 = \sigma_x^2(1-\rho^2)) \;\; k = 2, ..., K \end{aligned}$$

where variance of x, σ_x^2 is

$$\sigma_x^2 = \frac{\sigma_n^2}{1-\rho^2}$$

and ρ is the variance-normalized correlation between x_k and x_{k-1}. In VQ of image or speech, the vector $\mathbf{y}$linearly depends on the center of a cluster of source vectors $\mathbf{x}$. When modeling the source vectors as AR sequences, $\mathbf{y}$ becomes an AR sequence. The σ_x and ρ were chosen from among measurements on a set of 20 images. We also normalized the pixels. This made σ_x independent of the number of bits in the pixel representation.

```
begin LINEAR-MVPC
max = 0;
c = 1;
for i = 2, 3, ..., M                                   //M − 1 binary classifications
    sign = U;
    (ε̲, ε̄) = INITIALBOUNDS;
    w[0] = − Σ_{k=1}^{K} y^{(i)}_{k,(n−1)} x_k + b_i;
    //Bit-serial recurrence
    for j = 1, 2, ..., n − e − 1                       //n − 1 other partial terms
        w[j] = 2w[j + 1] + Σ_{k=1}^{K} y^{(i)}_{k,j} x_k;
        diff = w[j] − 2^{−j} maximum;
```

$$
sign = \begin{cases} +1 & \text{if } diff + \underline{\epsilon} \geq 0 \\ -1 & \text{if } diff + \overline{\epsilon} < 0 \\ U & \text{otherwise} \end{cases}
$$

```
        if sign = −1 or ( sign = +1 and i = M) then terminate inner loop;
    end_for
    if sign = U then
```

$$
sign = \begin{cases} +1 & \text{if } w[n-e-1] \geq max \\ -1 & \text{if } w[n-e-1] < max \end{cases}
$$

```
    if sign = +1 then begin
        max' = w[j];
        c' = i;                                        //Best matching codevector
    end_if
end_for
end LINEAR-MVPC
```

Figure 6.3: M-ary VPC with inner product similarity measure.

Similar to VPR simulations, these simulations were cross checked. First to verify correctness of the VPC algorithm, a conventional classifier cross checked every decision of VPC. This assured that the evaluated precision was sufficient. Then, we verified a simple simulation using the previously shown theorem. Given one element vectors $\mathbf{y} = -1$ and $\mathbf{x}$ uniformly distributed, we have observed the theoretically predicted *pdf* and average precision.

To observe the effect of the sample set on the simulation results, two different random number generators were used. We classified sets of 10,000 pseudo-random vectors $\{\mathbf{x}, \mathbf{y}\}$. Then for each curve, one of its points was re-ran with an alternative random number generator [PFTV88]. This would check not only for anomalies in the random number generator but also in the sample size. Little difference would assure us that the general behavior of the system and the conclusions we draw from the graphs were independent of our sampling. The change in *pdf* between the simulation runs using the two pseudo-random sources was small. From our observation of average precision, it was less than 0.025 bits ($< 2\%$). Since the results were close, we were assured that the 10,000 classification taken in the first run were good indicator of the discriminant *pdf*.

The elements of the sequence were evaluated at 10 bits. It was an 8-bit positive pixel which had gone through processing. In subtraction from the predicted image, it acquired a sign bit, and in subtraction from the nearest centroid codevector, its dynamic range increased by one bit. As we will also see, this can be extrapolated to the case of infinite-precision numbers.

Asymptotic Bound on Average Precision

Figure 6.4 shows the asymptotic bound on average precision as the operand precision increases. The necessary precision was evaluated for a set of 10,000 classifications. Since none of the classifications required more than 10-bit precision, and presuming we have zero probability of 10-bit necessary-precision because of the small sample set,

$$p(\delta \geq n) \leq 10^{-3} \tag{6.6}$$

At 10-bit precision, we have a very good estimate of the infinite-precision case. Using Lemma 1 from Chapter 5, we can now extrapolate to the infinite-precision case:

$$\mathrm{E}[\,\delta(g)] \approx \mathrm{E}[\delta_n(g)] + 2p(\delta \geq n) \;\leq \mathrm{E}[\delta_n(g)] + 0.002 \tag{6.7}$$

where we have substituted for $p(\delta \geq n)$ its upper bound from (6.6). The average precision is approximately 0.002 bits more than the average precision with 10-bit operands available. For all practical purpose, the simulations depict the infinite-precision case.

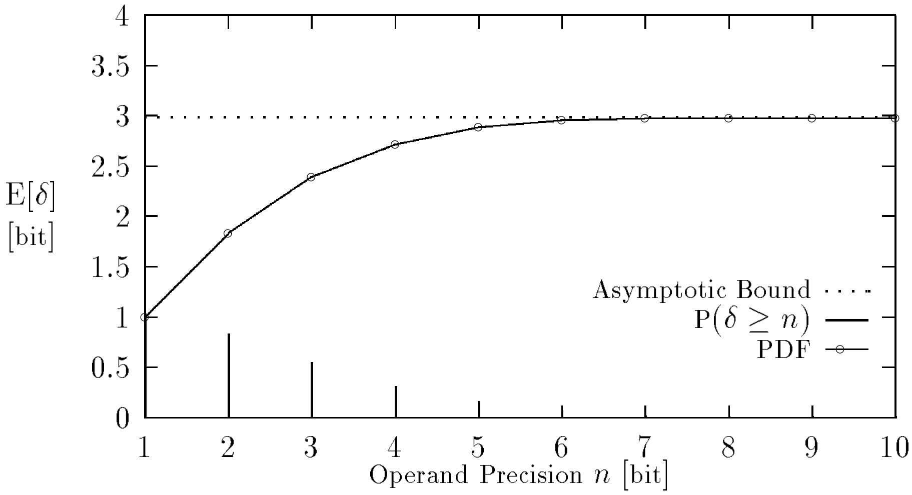


Figure 6.4: Average precision versus operand precision.

Sensitivity

Figure 6.5 shows average precision has little sensitivity to operand variance. Three curves show the sensitivity of average precision for statistically different operands. The operands are the 16 elements of a vector with ρ=0.98. As shown, normalizing the gain of the vector (i.e., GSVQ) makes the VPC performance independent of the vector element's variance. To a lesser extent, removing the mean of the vector which reduces the variance of the elements (i.e., MRVQ), also reduces its sensitivity. Even for unprocessed vectors, however, our simulations shows surprising little sensitivity. This suggests VPC is inherently independent of operand variance for other reason. We believe that this is due to VPR representation of vectors. To reduce storage, VPR normalizes a block of floating point numbers. In an arithmetically benign way, VPR is normalizing the gain. Next we set the variance $\sigma^2 = 0.029$ and plot the relationship of average precision with the correlation coefficient.

As can be seen in Figure 6.6, as correlation increases, the average precision decreases. This phenomenon can be described by the Central Limit Theorem. The theorem states that when independent random variables are added, the resulting *pdf* will approach a Normal *pdf*. When correlation is zero (ρ=0), all the elements of AR sequence are completely independent. Element by element multiplication produces independent random variables. The resulting discriminant *pdf* is similar to a Normal *pdf*. As the correlation approaches one, the elements of AR sequence become dependent. Inner product of these elements, reduces to simply scaling one of the identical elements. Since in our simulations we had used a uniform *pdf*, a low value discriminant has lower density near zero than the bell shaped Normal *pdf* has. Since small discriminants require higher necessary precision, higher correlation results in lower

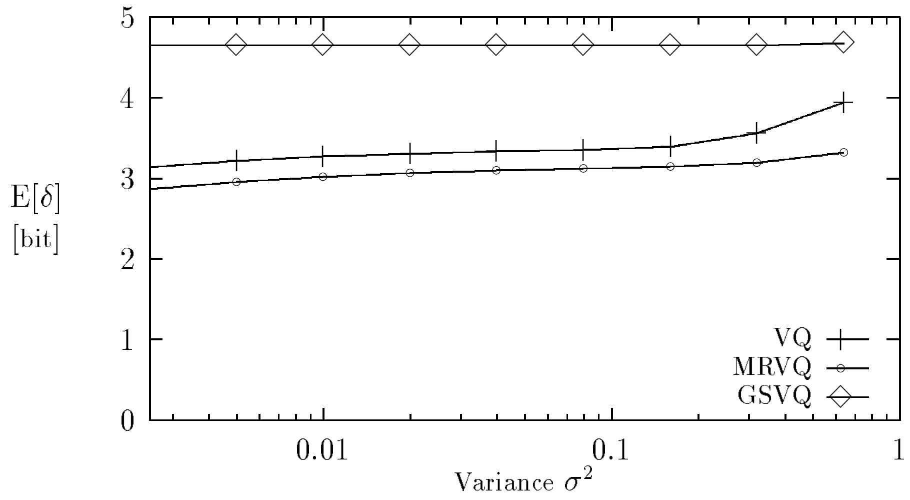


Figure 6.5: Average precision E[δ] versus vector variance σ^2.

average precision. For example, even if the element *pdf* was a double sided exponential *pdf*, for $\rho \to 0$ the discriminant *pdf* would again approach the Normal *pdf* as stated by Central Limit Theorem. In general, regardless of element *pdf* as correlation increases the average precision should become similar.

6.6 Experimental Results

To experimentally gauge the average precision, we compressed real images. In specific, we selected two images from USC data base : Baboon and Lena. Lena represented images composed primarily with low frequencies. Baboon, also known in literature as the mandrill, represented images composed of primarily high frequencies. Both were 256×256 pixels and were compressed using mean-residual VQ (MRVQ) with pruned binary trees (PBTSVQ). The number of elements in the codevector were 16 ($K = 16$), while the number of codevectors in the pruned tree ranged from 466 to 1906 codevectors ($N = 466, \cdots, 1906$).

The negative slope of the curve is as a result of the change in characteristics of the codevectors it is classifying among. As can be seen in Figure 6.7, as the compression error decreases, the average precision increases by 0.5 bits in both images. For Baboon, the increase is from 3.1 to 3.6 bits. For Lena, the increase is from 2.4 to 2.9 bits. For better compression VQ has to classify among codevectors which have less distance from each other. However, we do not believe this increases the average precision. In the last section, we showed that due to VPR, VPC is to a large extent independent of codevector variance (i.e., average distance). We believe in a more subtle reason: the type of the clusters the codevector is representing

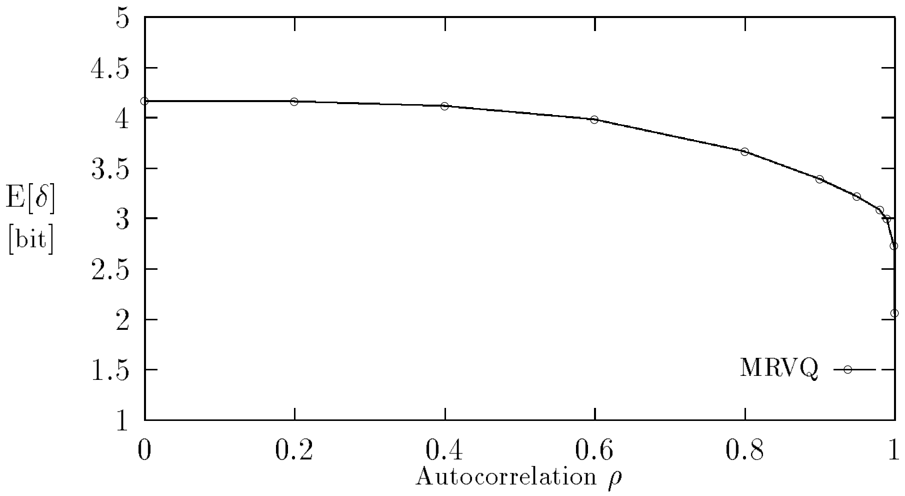


Figure 6.6: Average precision E[δ] versus vector correlation ρ.

differs. For better compression, random details are encoded. These are harder to group into clusters than the smooth details. The same factor which impedes the performance of VQ, also impedes the performance of VPC. In worst case, the VPC improvement solely would come from eliminating the leading zeroes in VPR representation. This bounds the average-precision at average storage precision (i.e., ≈ 6 bits for 10-bit operands).

The trends in the two curve and the gap between them seem to depend on the same factor. The average precision depends on the type of the cluster the codevector is representing. VQ of Lena entails classifying between well defined codevectors differentiated with few key low frequencies. On the other hand, VQ of Baboon entails classifying between codevectors which have irregular high frequency content. Such codevectors would have less well defined clusters than those used with the image of Lena. As a result, the average discriminant would be closer to zero, causing higher average precision.

6.7 Two Sign Selection Functions

In this section, we quantitatively compare Intervals A and B for sign selection. Interval A was used to theoretically show that the average precision has an asymptotic bound. It was designed based on an on-line signed-digit converter. Interval B is optimized for an early output of the sign (i.e., a non-zero digit). It narrows the bound based on the source vector elements. We will look at its increased performance in light of increased complexity. First, we will quantify the advantage of Interval B in performance. Then, we will describe the required additional hardware.

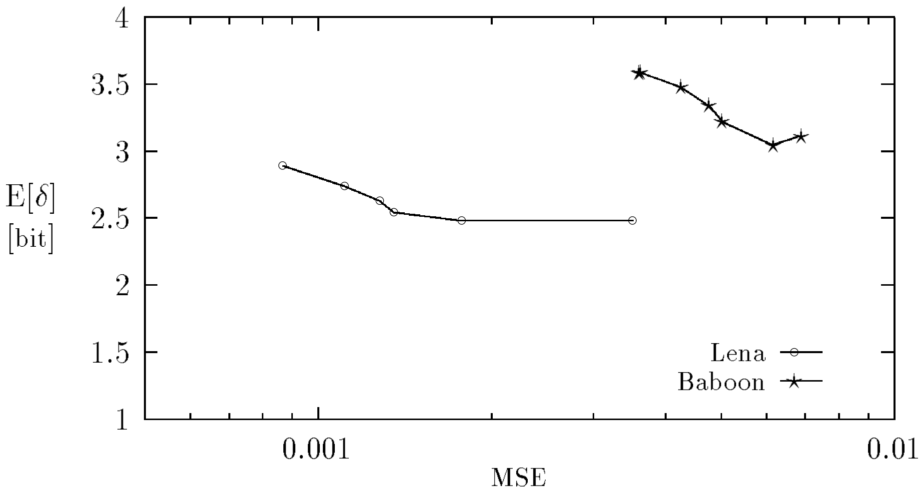


Figure 6.7: Average precision E[δ] versus VQ mean square error.

Performance Comparison

Figure 6.8 shows that Interval B is far preferable than Interval A. In the simulations, Interval B implemented specifically for VPC application improves the performance by more than a factor of two for zero mean sequences. The vector dimension was varied from 2 to 64. In both curves, due to Central Limit Theorem, as the vector rank increases the discriminant's *pdf* becomes more bell shaped around zero. Although in both cases the average precision increases, the trend is slower with Interval B. While Interval A grows with the maximum value of the discriminant. Interval B performs better. By summing the elements' magnitude, it tracks the discriminants far better.

Evaluating Interval A

The Signed-Digit (SD) converter forms the core of the composite sign detector. For E[δ] iterations, the signed-digit adder converts two MSBs from carry-saved into signed-digit form. If all the converted digits are zero, the remaining lower significant bits of the redundant number are inspected. The LSBs of PS and CS are added to with $n + \log_2(K)$ bits precision (i.e., $PS_{n-1}, \cdots, PS_{2n-1+\log_2(K)}$ and $CS_{n-1}, \cdots, CS_{2n-1+\log_2(K)}$). Since we only need the carry out in the addition, the circuitry can be simplified into Carry Look Ahead (CLA) circuitry. In gate arrays, CLAs are both fast and cost effective. Since this is on the critical path, the addition is done in parallel. On the other hand, the probability of inspecting LSBs is low, and we could also wait another cycle for its evaluation.

The SD converter adds the Pseudo Sum (PS) and Carry Saved (CS) terms. A one bit sign

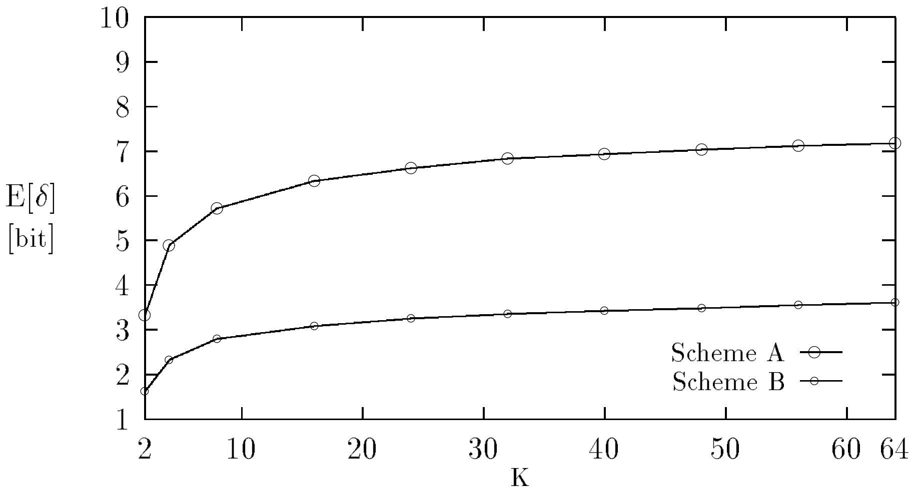


Figure 6.8: Average precision in classifying K-element auto-regressive sequences.

extension allows treating each term as a two's complement number. The two's complement number is mapped bit by bit to a SD number. If the MSB is one, it is mapped to negative one ($\bar{1}$). Otherwise each bit is copied unchanged. The one-bit sign extension would have assured that the MSBs which are being converted will not change by subsequent addition of less significant partial terms. These remaining partial terms collectively have the same maximum value that the last partial term had. Subsequent additions cause at most a carry from the last partial terms MSB, which are absorbed by the extra bit of the sign extension.

To design the SD converter, we simplify an SD adder for its two input PS and CS. Since PS is at most half the value of CS, the SD representation does not require more bits to represent. As shown in the table 6.3, the MSB outputs zero transfers ($T = 0$). In radix-4 conversion, moreover, the remaining digits of PS and CS are positive. Therefore, there are no negative transfers ($T \in \{0, 1\}$).

Evaluating Interval B

Variable size of Interval B results in more elaborate evaluation and comparison when compared to Interval A. The bounds on Interval B are not fixed. We need to precompute these for each source vector. For this, we need two accumulators. One accumulator sums the positive elements. The other sums the negative elements. These accumulators are active as $\mathbf{x}$ is being loaded on-chip for classification. Afterward, Interval B requires two high precision comparators. This lengthens the critical path. To investigate the tradeoff between the comparator precision and the average necessary-precision, we did simulations with VQ of Lena.

Table 6.3: Converting carry-saved into signed-digit form.

	T=0		T=1	
	PS_0, PS_1	PS_0, PS_1	PS_0, PS_1	PS_0, PS_1
CS_0, CS_1	00	$0\,\overline{1}$	00	$0\,\overline{1}$
00	00	$0\,\overline{1}$	01	0 0
01	01	0 0	10	0 1
$\overline{1}0$	$\overline{1}0$	$\overline{11}$	$0\overline{1}$	$\overline{1}0$
$\overline{1}1$	$\overline{1}1$	$\overline{1}0$	00	$0\,\overline{1}$

In specific, the picture was 256x256 pixel and the codebook had about 2000 16-dimensional codevectors. In each simulation, about 45000 classification were done.

As seen in Figure 6.9, we loose little in performance by looking at the 11 MSBs of the estimate. The knee of the curve is at $\hat{n} < 10$. When the comparator's precision is decreased further to 11-bit the average precision increase is still less than 0.2 bits.

As shown in Figure 6.9 as the precision of comparator approaches two bit $\hat{n} = 2$, the performance of Interval B approaches that of Interval A. The comparators for Interval B would simply evaluate digit-serially from MSB to LSB, comparing 2-bits in each cycle. This resembles the performance for Interval A. Interval A, which acts as an upper bound on performance degradation, is basically a two bit comparator. This is because the bound in A is approximated a power of two. The bound A is comparing against a number whose only one is in MSB position. In other words, Interval A is a sign selection function which is given only the two MSBs of the estimate.

Figure 6.10 shows the circuitry to calculate the error Interval B. For each received source vector $\mathbf{x}$, the bounds $(\underline{\epsilon}, \overline{\epsilon})$ are initialized as follows

$$(\underline{\epsilon}, \overline{\epsilon}) = (\sum_{i=1}^{K} \min(x_k, 0), \sum_{i=1}^{K} \max(x_k, 0))$$

There are two accumulators, for the two bounds $\overline{\epsilon}$, and $\underline{\epsilon}$. Since we exclusively add an element into the upper bound or into the lower bound, the two accumulators can share their adder.

Figure 6.11 shows comparison of 11 MSBs of redundant estimate against the two error bounds. The comparison occurs for each of the first $(n - e - 1)$ cycle. In the last cycle, we stop comparing against the error bounds. Instead, a full-precision evaluation determines the sign of the estimate. As shown, for the full-precision comparison one of the error bound

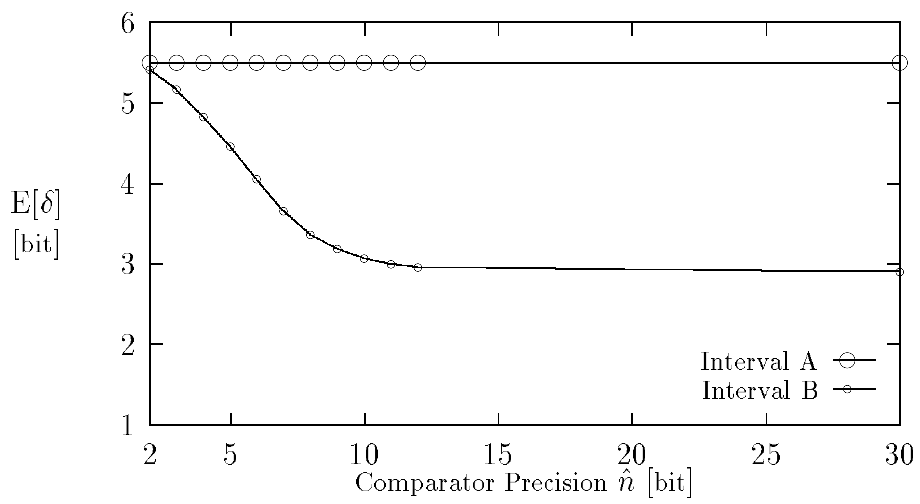


Figure 6.9: Average precision E[δ] versus comparator precision.

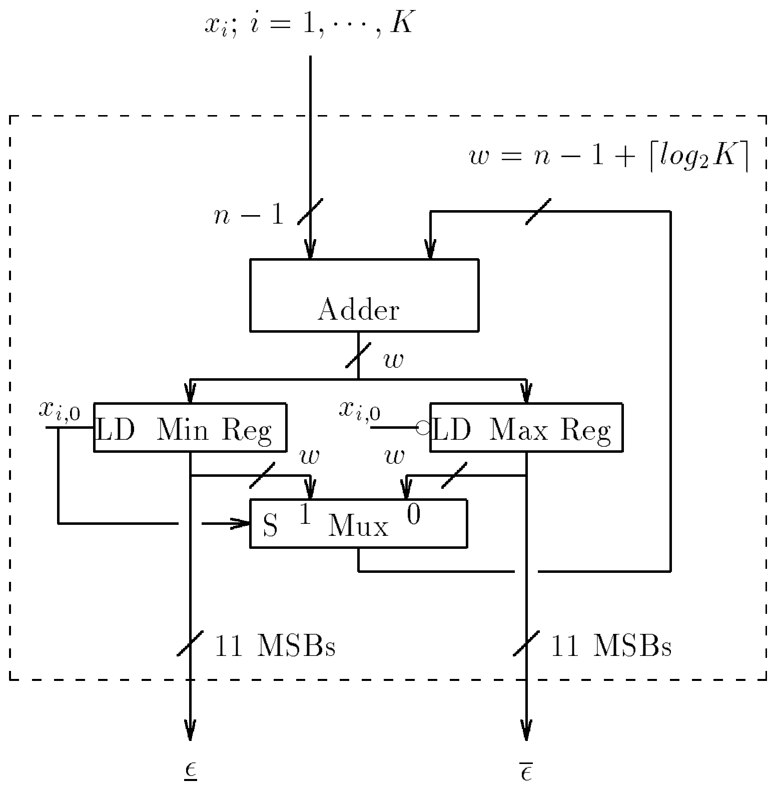


Figure 6.10: Initializing error bounds used to select sign.

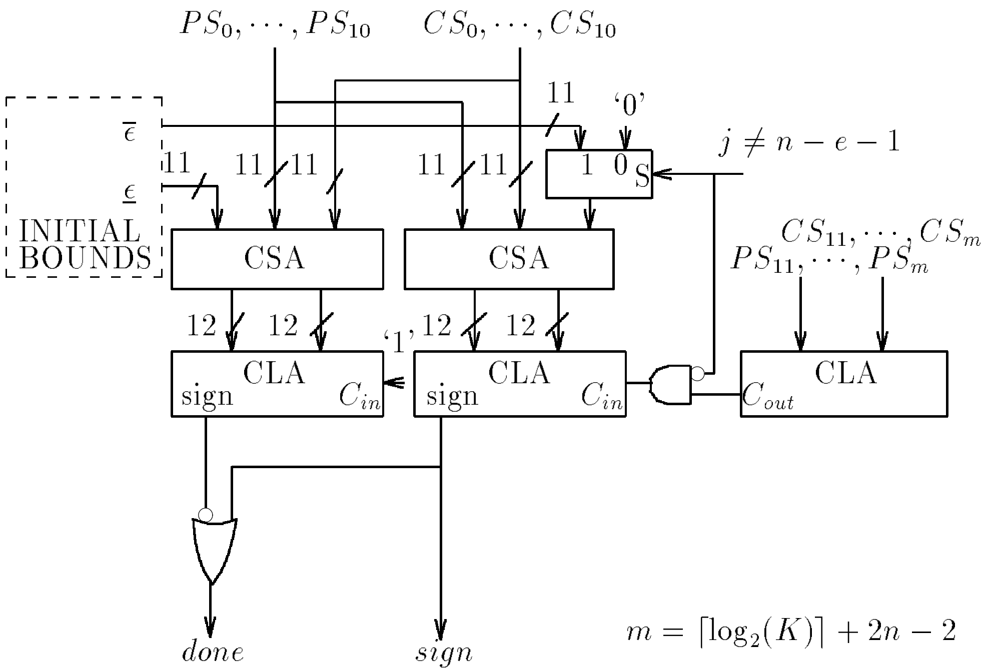


Figure 6.11: Interval B sign selection circuitry.

comparators is used in conjunction with a 12-bit carry look ahead circuitry.

Cost

Although Interval A is simple to evaluate, Interval B doe not require a lot of circuitry either. When K is either a power of two, or is replaced by the next greater power of two, Interval A requires only a simplified signed-digit adder and a $log(K) + n$ carry look ahead circuit. Evaluation of Interval B, on the other hand, needs one $log(K) + 10$ adder and two registers of the same size to compute the bounds. Although it requires more elaborate sign detection circuitry than Interval A, the increase in complexity is offset by more than a factor of two reduction in average precision necessary for classification.

6.8 VPC Architecture

We now show a simple way to design bit-serial architecture. Starting from the VPR design, it is modified for variable-precision classification (VPC). It minimizes the evaluation time and memory-processor traffic, since it can vary the precision one bit at a time. When it can meet the throughput, it is preferable for implementation.

Figure 6.12 shows the VPC architecture. While, it evaluates all the partial products of same significance concurrently, VPC can reach a decision with fewer iterations then VPR. VPC implementation differs from VPR in two aspects. First, the order of bit-serial evaluation is changed. VPC evaluates from MSB to LSB in an on-line fashion. Second, its sign detector is more sophisticated. Using two comparators, it decides on an early termination of evaluation. At the end of every iteration, the two comparators check the estimate against the lower and upper bounds on the sign selection interval. When either comparators can decide, the sign is output and the done flag raised.

Since VPC uses redundant representation in accumulating the estimate, *can VPC also benefit from redundant operand representation?* First, let's look at the codevectors and whether we may gain anything from precomputing and then storing them. Since storage access and cost are a key factor in the VQ design, any advantage which is gained from redundant representation will be very costly. Unlike codevectors, the source vectors are not stored. Their computation cost would be lowered by redundant representation. This would, however, shift the burden to VPC. In mean-residual VQ, VPC is K times more complex than the source vector preprocessing. Redundant representation only seems more relevant with source vectors. Redundant source vectors, would effectively double the number of terms to be generated and summed in the VPC inner product. This makes redundant source vectors also unattractive. Both for codevectors and source vectors, VPC does not favor redundant representation.

In an M-ary classifier, instead of just detecting the sign of the i-th estimate, the classifier compares the estimate with the maximum of the $(i-1)$-th inner products evaluated. The data path expands into two subtractors and a sign detector. In every iteration, the inner product with its two error bounds are subtracted from the maximum. If the sign is ambiguous, the evaluation continues. If the sign of the result is positive, its evaluation terminates. Otherwise if the sign of the result is negative, it evaluates the inner product at full precision and replaces the current maximum. At the last inner product, we can evaluate as a binary classification. After determining the sign we can output the index. Since in the last inner product we do not need to store the new maximum, there is no need for a full-precision evaluation.

VPC Controller

Figure 6.13 shows the timing of VPC for binary classification. The processing period for each bit $y_{i,j}$ consists of three parts: (i) fetching the j-th bits of $\mathbf{y}$ from memory, (ii) summing the partial products, and (iii) deciding whether to continue or terminate the iteration. As indicated in the figure, these parts can be overlapped. Such an architecture is attractive for low rate video and speech.

VPC controller is similar to VPR controller, except it also monitors *done* flag. VPR controller would enhance the design by Holmberg [Hol90] with a counter. The counter at

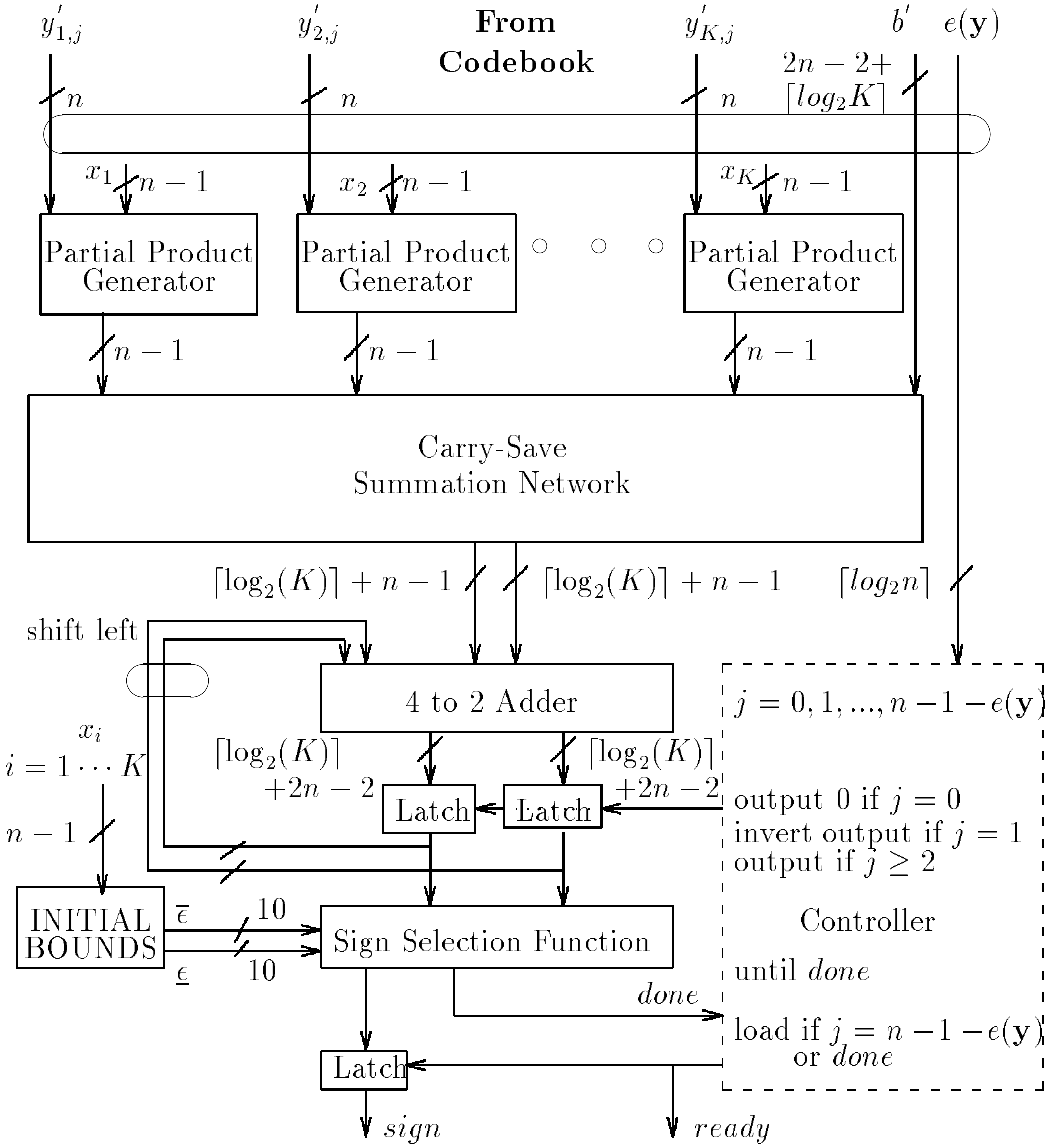


Figure 6.12: Architecture of binary linear VPC.

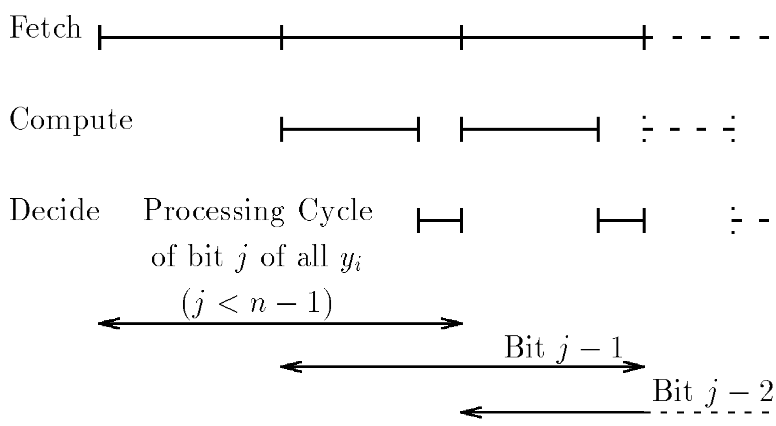


Figure 6.13: Timing of VPC.

the beginning of each evaluation is loaded with the precision of the inner product. VPC controller adds one more modification. The termination signal can be generated either by a count down to zero or when the done flag is raised. With Interval B, the controller also manages the accumulators which during loading of the source vectors evaluate the upper and lower bounds.

Implementation

Combining the VPR implementation with some design modification, we had an accurate estimate of the cost and performance of VPC. The specification and the throughput were not changed (e.g., $K = 16$ and $n = 10$, MPEGII). As shown in Figure 6.14, the VPC data-path was also similar to VPR data-path. This simplified the controller and reduced the partial product generator to just nine ($= n - 1$) AND gates. The primary difference was the sign detector that output every iteration. We also changed the order of evaluation of operand bits to be from MSB to LSB.

Similar to VPR implementation, in VPC variable-precision addition of the hyperplane bias term b would reduce the memory-processor bandwidth. With an off-chip codebook, the number of I/O pins can be an issue. Variable-precision addition of bias can reduce the 20 pins required by the specification (e.g., $K = 16$ and $n = 10$) to four pins. In a radix-2 VPC, with every bit of inner product, two bits of bias term are added. We believe there will be only a small affect on the average precision. Even though, the bias term is not added in full precision, its precision increases at twice the rate of the terms in the inner product. For the typical classification, which results in three bit average precision, $E[\delta] = 3$, the bias term will have six bits precision. In this case, the estimate itself will have around six bits precision. When the memory-processor bandwidth is at premium, the cost of multiplexing b should be negligible.

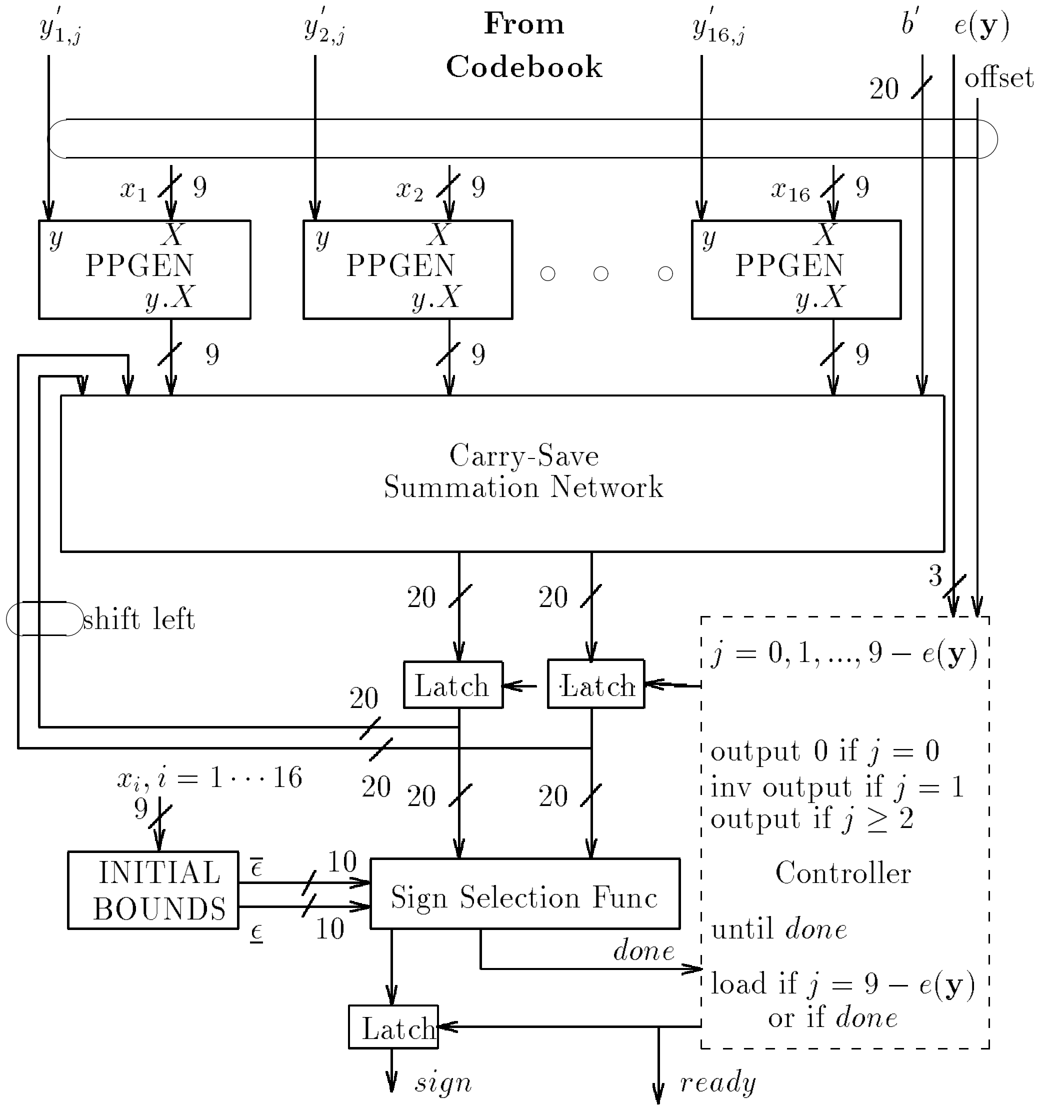


Figure 6.14: VPC implementation.

6.9 Throughput

For the shown design, the throughput would be far lower than what was shown in Chapter 4. Even so VPC would still compress 1/4 CIF video which inputs 760×10^3 pixels per second [1]. Next we will look at two ways of increasing the throughput: digit-serial evaluation and pipelining. As we will see pipelining alone permits sufficient throughput for MPEGII [2] data rates.

Increasing Throughput

We have shown a simple bit-serial architecture. By increasing the precision one bit at a time, it uses the minimum number of iterations for evaluation. Bit-serial evaluation, however, requires E[δ] cycles per classification. This limits the throughput.

An obvious way to increase the throughput is to use several inner product modules. This high level parallelism is attractive when the required throughput is very high. Although we may not be able to take advantage of variable number of iterations, it would be attractive for reduction of energy dissipation/consumption. For future work, we could look at it. In here, we will focus at increasing the evaluation radix and/or pipelining.

When the evaluation is bit-serial and non-pipelined for typical operand precision, VPC is preferable choice for implementation. On the other hand, the throughput requirements may force us to use a digit-serial and/or pipelined evaluation. *For what operand precision is VPC better than conventional approach?* To answer we use the notion of sufficient-precision $\hat{\delta}$, which is the number of bits which are evaluated. Sufficient-precision $\hat{\delta}$ adds the affect of implementation detail to the calculated necessary-precision δ. For a bit-serial non-pipelined VPC module, the sufficient precision is same as the necessary precision ($\hat{\delta} = \delta$). Next, we'll quantify the increase in precision ($\hat{\delta} \geq \delta$) for pipelining.

Pipelining which was proposed for VPR, can be used with VPC. Pipelining the VPC module is similar to pipelining the VPR module shown in Chapter 4. Pipeline latency, however, incurs higher cost. In an M-ary VPC module, we need to look at the output after every iteration. In a simple l-stage implementation, this comparison is delayed by $(l-1)$ cycles. The sufficient-precision $\hat{\delta}$ becomes

$$\hat{\delta} = \delta + l - 1$$

In other words, an l-stage pipeline increases the average precision by l bits.

The degradation caused by pipeline latency can be avoided. We can interleave evaluation of l classifications. To do so we would evaluate l classifications independently. The l stage

[1] 176×144 pixels (1/4 CIF) at 30 frames/sec = 760,320 pixel/sec

[2] 720×576 pixels at 30 frames per second

pipeline then achieves a factor of l more throughput than a non-pipelined system. This could permit high level of pipelining. We can have full bit-level pipelining of the summation network and the comparator pair. On the other hand, the interleaved evaluations does require on-chip buffering of each interleaved source vectors, and a more elaborate controller to track these concurrent classifications.

We could increase the throughput by increasing the evaluation radix. In a radix-r architecture, the time taken in number of iteration $T(r)$ becomes

$$T(r) = \frac{\delta}{\log_2 r}$$

The above multi-fold increase in throughput is applicable whenever the function can be evaluated in parallel. It would not be realizable for metrics using operations such as division. With inner product metric, where fully parallel evaluation is possible, there is no limit on the radix (e.g., $r = 2^n$).

The drawback is increase in the sufficient-precision. The increase is due to evaluating $\log_2 r$ bits at a time. Sufficient-precision $\hat{\delta}$ could be as much as

$$\hat{\delta} \leq \delta + \log_2 r - 1$$

where we assumed the worst case of evaluating one digit, while only evaluation of one bit was necessary.

Assuming uniform discriminant *pdf*, radix-4 evaluation increases the average precision by one bit. In radix-4, two bits of the multiplier are needed for each partial term. In instances when the number of bits to give the sufficient precision is odd, even number of bits is evaluated. In these instances, radix-4 multiplication results in an estimate with superfluous accuracy. Similar to VPR design, there is also overhead due to misalignment of packed codevectors. With uniform *pdf*, the sufficient-precision becomes

$$\hat{\delta} \approx \delta + 1.0$$

In a pipelined radix-4 implementation, the difference between the necessary and sufficient-precision is two-fold,

$$\hat{\delta} \approx \delta + 2(l - 1) + 1.0$$

where the 1.0 bits overhead is incurred when fetching two bits at a time.

String recoding which was used in the implementation of the VPR architecture, can also be used with VPC architecture. In the VPC, the string recoding starts from the MSB, incurring one bit delay. String recoding from MSB is similar to string recoding from LSB. When starting to recode from MSB, expecting a carry, we delay the output by one cycle. Recoding from MSB *requires* sign extension to implicitly delay by one clock cycle. To send

Table 6.4: Memory bandwidth of VPC versus VPR design.

	Pipeline [level]	CK [ns]	Avg(Worst) Time[cycle]	Avg(Worst) Time[ns]	BW [pin]	$BW \times T(\times 10^{-6})$ [pin×sec]
VPR	2	20.7	5(7)	103.5 (144.9)	42	4.35 (6.09)
VPC	3	14.8	5.6(12)	82.9 (178)	24	1.99 (4.26)
VPC	5	8.3	7.6(14)	63.1 (116.2)	24	1.51 (2.79)
VPR	6	8.3	9(11)	74.7 (91.3)	42	3.14 (3.83)

only two bits per cycle to the recoder, we extend the sign inside the recoder by one bit. In the first cycle, then we get three bits out of only two received bits. For a multiplicand with odd number of bits, the multiplication does not need additional cycle. The sufficient-precision shown in the previous equation remains valid.

For VPC to be attractive in exhaustive searched VQ, the operand precision n should be greater than the sufficient precision $\hat{\delta}$, where $\hat{\delta}$ incorporates the cost of pipeline break. This does not apply for binary searched VQ, where after the inner product and pipeline break occurs as we determine which branch to take.

6.10 Comparison

VPC reduces the computational complexity both in terms of execution time and area. Instead of radix-2 partial product generators used in VPR, VPC module uses radix-2 generators. Even though, VPC requires an elaborate sign detector with comparators and initialization of bounds, the overall gate count decreases about 15%. Table 6.4 shows the clock cycle (CK), execution time and the pin counts of the competing architectures. Even though VPC evaluates bit-serially, eliminating superfluous LSBs decreases the execution time from that of radix-4 VPR. Table 6.5 shows that both AT and AT complexity measures decrease by more than 20%. Table 6.4 shows that the VPC is also quite effective in reducing the memory bandwidth used by more than a factor of two. This is to a lesser extent due to radix-2 evaluation, which fetches only the bits of the required codevector.

When pipelining is incorporated, VPC remains preferable to both VPR and conventional designs. VPC reduces the overall execution time. Even though pipelining increases the average precision, it similarly affects VPR. VPC due to its shorter overall critical path can be pipelined with one less level than VPR. As a result the overall execution time of a radix-2

Table 6.5: Performance and cost of VPC versus VPR design.

	Pipeline [level]	Area [gate]	CK [ns]	Avg(Worst) Time[cycle]	$AT \times 10^{-3}$ [gate$\times$sec]	$AT^2 \times 10^{-12}$ [gate$\times$sec^2]
VPR	2	9100	20.7	5 (7)	0.94 (1.32)	97 (191)
VPC	3	7492	14.8	5.6 (12)	0.62 (1.33)	51 (236)
VPC	5	8230	8.3	7.6 (14)	0.52 (0.96)	33 (111)
VPR	6	11164	8.3	9 (11)	0.83 (1.02)	62 (93)

VPC is less than a radix-4 VPR. VPC, less pipelined and with radix-2 multiplication, can be implemented with fewer circuitry than VPR. Radix-4 design on the other hand has one advantage, it worst case execution time is 20% less. The worst case speed is critical when there is a fixed time for compressing a frame of image.

If we were to use off-chip memory further pipelining would be useful. For example, if the memory bus is half the speed of the processor, we could halve the processor and double its clock rate. In other words, the pipelining would be used to reduce the number of gates. For the lowest average precision, the elements would still be fetched bit-serially; one significant bit would be fully evaluated before the next one.

6.11 Conclusions

We have quantified significance of the presented arithmetic insight for classification. Most classifications can be done with low-precision. For example in image coding with VPC module, we have observed the average necessary-precision to be less than 3.6 bits. (VPC complemented VPR's elimination of superfluous MSBs.) Through simulations on auto-regressive sources, we also postulated that the average precision is largely independent of variance. Also for large vector dimensions and low correlation, the average precision is also independent of element *pdf*.

When the total execution time is important, VPC is preferable to the conventional and VPR implementations. We presented a simple modification which converts the VPR implementation into a VPC design. The decrease in execution time more than offsets the increased complexity, halving the AT^2 cost. The resulting pipelined design could vector quantize MPEGII video.

In digital signal processing, the trend has been toward higher precision. There have been several widely used 24-bit digital signal processors. Since VQ gain increases only logarithmically with the number of codevectors, increasing precision also becomes a viable avenue for VQ in future. VPC will become even more viable, when 10-bit pixels become common.

Evaluating at average precision is more valuable when the operand precision increases. VPC has a computational complexity of $O(1)$ with respect to precision. Using a corollary to VPC, the simulations can predict the performance for a higher precision. Meanwhile, the performance of VQ improves logarithmically when increasing the precision beyond 10 bits, while the conventional inner product implementation's complexity grows in square of the number of bits $O(n^2)$. This had been the intuitive reason for the limited interest in increasing the evaluation precision.

CHAPTER 7

Summary and Future Research

This dissertation explores arithmetic and number representation as a means of reducing cost of vector quantization. The investigation has two parts. Variable-precision representation (VPR) reduces the number of bits stored by eliminating leftmost bits which are sign extension in all bits of a codevector. On the least significant side, Variable-precision classification (VPC) reduces the number of iterations for classification by eliminating the rightmost bits which are superfluous. This chapter summarizes the key contributions of the dissertation and suggests potential directions for future research.

Summary

- We developed a noiseless compression technique which has negative decoding cost: variable-precision representation. We simulated the performance of VPR for real codebooks. The codevectors were trained in image compression with pruned binary-tree structured VQ (PBTSVQ). For comparison, we also measured the codevector compression using entropy coding (e.g., Huffman coding). VPR is shown to effectively reduce the codebook storage. VPR performs 75% as well in comparison to entropy coding. Although VPR is implemented for binary trees, the approach can be generalized to M-ary trees with comparable if not better performance.

- We implemented the decoder-processor module. It is a novel module, which unlike conventional evaluation incrementally increases the precision of inner product. For most compression, the precision of elements varies one bit at a time. For a high throughput, it evaluates in radix-4 (i.e., two bits at a time). Also for comparison, we implemented a conventional inner product module composed of four multipliers and adders. When VPR processor can be used in variable-time applications the decoding has negative cost. Specially when pipelined, the decoding overhead of VPR is more than offset by the average decrease in the evaluation time. The AT measure showed 22% reduction in complexity, while AT^2 measure showed 29% reduction in complexity.

- We developed the variable-precision classification algorithm. In conjunction with VPR, it reduces the number of bits evaluated. We showed that the average necessary-precision is asymptotically constant. In other words, as the operand precision increases the computational complexity of classification becomes independent of it.

- We measured the performance of VPC for image compression, where VQ was based on the inner product discriminant. The average precision for classification was measured in two ways. First, using auto-regressive vectors we showed that VPC was relatively insensitive to change in vector variance. Then, we compressed two images: Lena and Baboon, measuring the average necessary-precision. With Baboon, the average precision necessary for classification was less than 3.6 bits. With Lena, the average precision was less than 2.9 bits.

- We implemented VPC module. When the total execution time is important, VPC was shown to halve the computation based on either AT or AT^2 complexity measure. VPC is attractive when the total execution time is important such as in off-line processing or when there already is a queue buffer before and the average execution time is important. We also showed a simple generalization of VPC to M-ary VPC. Whenever we would not need to replace the maximum, VPC would avoid full-precision evaluation.

Future Research

In this dissertation, we have significant improvements in hardware for compressing images. VPC can potentially result in even greater gains when compressing speech.

- We would like to evaluate the performance of VPC in compression of speech. Speech samples are often quantized at 12-bit, and precision of 16-bit is widely used (e.g., compact disc recording). The high operand precision would make VPC performance more attractive than it was with 8-bit pixels used in image compression. It would be worthwhile to develop the corresponding software and measure the performance for the above mentioned applications.

- Derive a theoretical estimate for performance of VPC, assuming two-sided exponential *pdf* for correlated codevector and source vector elements. It can act as an upper bound on VPC's performance. It would also confirm our findings on average precision of auto-regressive sequences, that the discriminant *pdf* approaches the Normal *pdf*.

- Implement VPC in gate arrays. For estimating cost-performance we already have developed a detailed design of VPC on paper. A gate array design could show the viability of VPC.

- We would like to investigate effectiveness of VPC for reducing power consumption/ dissipation. VPC's reduction in average number of iterations reduces the total amount of energy drain. This is valuable when using VQ in a battery powered equipment. For example, VQ maybe used on-board satellites or in cellular speech and video communications.

Also as densely packed circuitry become technology feasible, VPC may address the power dissipation. For Large Constraint-Length Viterbi decoding [BDB+92], there is massively parallel implementation of classifiers. Also for high quality compression systems, there are proposals for massively parallel VQ architectures [DCG88].

APPENDIX A

VQ Application

A.1 Required Throughput

For a black and white video source, we can calculate the required throughput. Let's consider the video which is sampled at rate of f_f frames/sec, where each frame is w pixels wide and h pixels high. The resulting source vector arrival frequency is

$$f_v = \frac{f_f w h}{K}$$

where K is the number of elements in the vector.

Colored video is typically treated as an extension of monochrome video. Colored image is separated into three images: the primary intensity image, and two subsampled chromaticity images which have a quarter of the resolution of the intensity image. As a result the computation required for a colored video is about 1.5 times more than for a black and white video.

Let the processor clock frequency be f_c. For a radix-4 VPR, each K-dimensional distortion computation requires $n/2$ clock cycles. The frequency of distance evaluation becomes

$$f_d = \frac{2}{n} \times f_c$$

This throughput places an upper bound on the number of codevectors N which can be individually inspected:

$$N \leq \frac{f_d}{f_v}$$

For images, typical values are an arrival rate f_f of 10 frames/sec and block sizes of about 256×240 (in the U.S) and 360×288 elsewhere (in Europe and Japan). For a clock frequency of f_c = 20 MHz (f_c is usually limited by the Codebook access time), above inequality yields an upper bound on the codebook size N as N $\leq$ 32. Since the distortion incurred is a monotonically decreasing function of the codebook size, an N=32 VQ system would undergo undesirable distortion and performance degradation. Thus, full search schemes (which require large codebooks) are constrained by the limited processing power of the chip compared to the high sample rates. In actual system design, most often a fast search technique is employed.

Specification

At a high throughput, we intend VQ at proposed MPEGII rate (12.3×10^6 pixel/sec) video using PBTSVQ. MPEGII is the next standard for image compression, designed for compression of high quality video used in broadcasting studio. In our simulations, we use mean-residual VQ with 16-pixel vector ($K = 16$) as a simple form of preprocessed VQ. In actual system design, a more sophisticated preprocessor would be employed. In some aspect our specification is similar to a system analyzed by Modestino *et. al.* [MK92]. That system would encode at a rate ($R = 1$ bit/pixel). Instead of using a separately transmitted mean to predict the block, they predict the block by the adjacent already transmitted blocks. The image pixels are typically digitized at 8-bit precision. Subtracting the mean of the vector from its elements, adds one bit to represent the sign of the pixel. Subtracting two centroid codevector to find the hyperplane codevector results in 10 bits per stored codevector element. The specification becomes:

$$\begin{aligned}
\text{Frame Height } h &= 576 \text{ pixels} \\
\text{Frame Width } w &= 720 \text{ pixels} \\
\text{Frame Frequency} f_f &= 30 \text{ frame/sec} \\
K &= 16 \\
\text{Video Frequency} f_v &= 12.4 \times 10^6 \text{ pixel/sec} \\
n &= 10 \text{ bits} \\
\text{Compression Rate } R &\leq 1.0 \text{ bit/pixel}
\end{aligned}$$

For a replacement codevector, a pruned tree which has up to 12 levels is searched ($R \leq 0.75$ bit/pixel). Required throughput is at most

$$\begin{aligned}
f_d &= \frac{f_v}{K} \times (KR - 4) \\
&= 9.3 \times 10^6 \text{ inner product/sec}
\end{aligned}$$

where we have allocated four bits for quantizing the vector mean [Bak84], leaving $(KR - 4)$ bits for encoding the codevector index. (Equivalently, the required evaluation time T_d is 107 ns.)

APPENDIX B

Implementations

B.1 VQ Search Controller

Figure B.1 shows a controller designed to search VQ trees stored using VPR. The base address would have already been fetched. The codevectors would be stored in adjacent location. When VPR-controller raises the *done* flag, the search-controller begins fetching the next hyperplane vector. Based on the given sign, the search-controller fetches one of two codevectors. If the left branch was chosen, the base address would be used to fetch the next codevector. If the right branch was chosen, the base address is offset by the relative address of the right branch before fetching the codevector $(relative\, address = \lfloor (e(\mathbf{y}^l) + offset(\mathbf{y}^l))/2 \rfloor)$. As the search proceeds, all the signs of the hyperplane tests are shifted one bit at a time into a register. At the end of the search, the shift register has index of the chosen codevector.

A conventional VQ controller was implemented at UCLA. It was part of a video compression board which was designed by Holmberg [Hol90]. It included a controller for searching PBTSVQ with bit-serial inner product data path. Designed for FPGA but never built, the micro-controller was intended for fixed precision elements. It was similar to the VPR controller, except the relative address of the branch was a constant five.

The conventional search-controller can be easily modified for variable-precision codevectors. It needs only two changes. First, its modulo-5 counter which initiates a new computation every five cycles is removed. Instead, we initiate with the *done* signal which is output from the inner product nano-controller. Second, in calculating the address of codevectors, the relative address of the second branch is not fixed at five. It is a function of exponent $e(\mathbf{y})$.

Even though variable size of codevectors requires an address space which has two bits finer resolution, the storage overhead is still low. First, the exponent of each codevector can be stored in three bits. Also in binary classification for every codevector pair, we need to store the offset and address of the first codevector. Since with radix-4 storage of elements, there is two more bits resolution than conventional approach. All accounted, for every two codevector nine bits more are stored (i.e., 4.5 bits per codevector). Compared with M-ary classification, our overhead increases only little (i.e., 4.0 bits vs. 4.5 bits).

B.2 VPR Implementation

Data Path

As shown in Figure B.2, the VPR core is 16 ($= K$) radix-4 partial product generators and a summation tree. The radix-4 partial product generators, which use string (Booth) recoding of the multiplier, act like digit-serial multipliers. Their output along with the bias term is gathered by a tree of adders and accumulated in a carry-save redundant form. The multipliers and addition tree is iteratively fed the digits of the VPR codevector. It presents its final accumulation to the sign detector. For fast execution time, the sign detector is based on a 4-bit carry look ahead adder. For lower complexity, it is stripped to the bare circuitry needed to calculate the sign bit. Specifically, five 4-bit carry look ahead modules are concatenated.

Figure B.3 shows the string recoded partial product generator. In a radix-4 implementation, string recoding of the multiplier is attractive. String recoding simplifies the partial product generator. Instead of requiring two adders, a string recoded generator uses an adder and a 4-1 multiplexor.

String recoding can be viewed as breaking any digit which is '3' to '-1' with a '1' carried to the next significant position. This maps the digit set {0,1,2,3} into {-2,-1,0,1,2}. The digit recoding needs three adjacent bits. In every cycle, two bits are fetched. A latch which stores the overlapping bit from the previous cycle provides the third bit. This latch is reset at the beginning of the recoding.

The summation network also adds the bias term b. Bias term b is stored in variable precision to minimize storage, and is added digit-serially to minimize access. From Chapter 2 which described the inner product computation,

$$b \equiv -\frac{1}{2}\sum_{k=1}^{K}(y_k)^2 \tag{B.1}$$

With $(n - e(\mathbf{y}))$ bits in $y_k^{'}$, the number of bits in $b^{'} = b/2^{2e}$ is,

$$\begin{aligned} bits(b^{'}) &= \lceil \log_2 K \rceil + 2n - 3 - 2e(\mathbf{y}) \\ &= 21 - 2e(\mathbf{y})\ , \quad n = 10, K = 16 \end{aligned}$$

When the precision of $y_k^{'}$ decreases by one bit, the precision of $b^{'}$ decreases by two bits. Note that, LSB was eliminated when the summation was divided by two. In here, we now also truncate the second LSB,

$$bits(b^{'}) \ = \ 20 - 2e(\mathbf{y})$$

Now $b^{'}$ can be multiplexed in equal sized nibbles (i.e., 4 bits). This is convenient. With every digit of codevector $\mathbf{y}^{'}$, we store one nibble of bias $b^{'}$. The nibble memory addressing can easily be made identical to that of the codevector $\mathbf{y}^{'}$.

STATE // D = maximum depth of tree

register	$address_{next} < D+2 >$,	// address of a codevector pair
	$e_{left} < 3 >$,	// exponent of left branch
	$e_{right} < 3 >$,	// exponent of right branch
	$offset_{next}$;	// offset of left branch
	$label < D >$;	// current node label
	$label_{leaf} < D >$,	// $label > label_{leaf}$ are leaves
counter	$address < D+2 >$,	// current codebook address
	$offset$;	
shift register	$index < D >$;	// built in multiply by 2

INPUT

boolean $reset$, $begin$, $sign$, $done$;

OUTPUT

boolean end;
integer $index < D >$;

BEGIN PRUNED BINARY TSVQ

if reset

{ reset $address$ ‖ reset $index$ ‖ reset $offset$ ‖ reset e}

if first_iteration

{ load $address_{next}$ ‖ load $offset_{next}$ ‖ load e_{left} ‖ load e_{right} ‖ load $label$ }

if $done.sign'$

{ $address = address_{next}$ ‖ $offset = offset_{next}$ ‖ $e = e_{left}$ }

if $done.sign$

{ $address = address_{next} + \lfloor (offset_{next} + e_{left})/2 \rfloor$ ‖
$offset = (address_{next})_0 \oplus offset_{next} \oplus e_{left,LSB}$ ‖ $e = e_{right}$}

if $done$

{ $index = 2index + sign$ }

if $label > label_{leaf}$

{ $end = 1$ }

END PBTSVQ

Figure B.1: Controller for searching binary tree structured VQ codebooks.

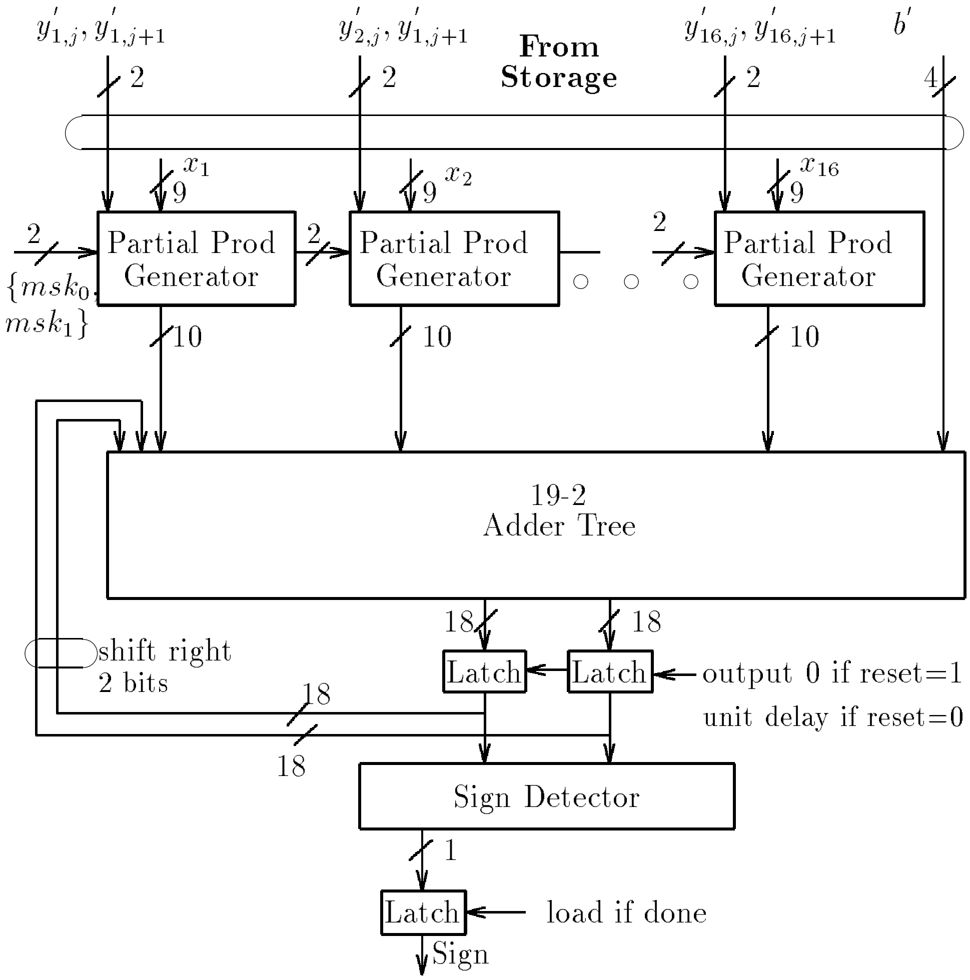


Figure B.2: Implemented VPR inner product with radix-4 string recoded partial product generators.

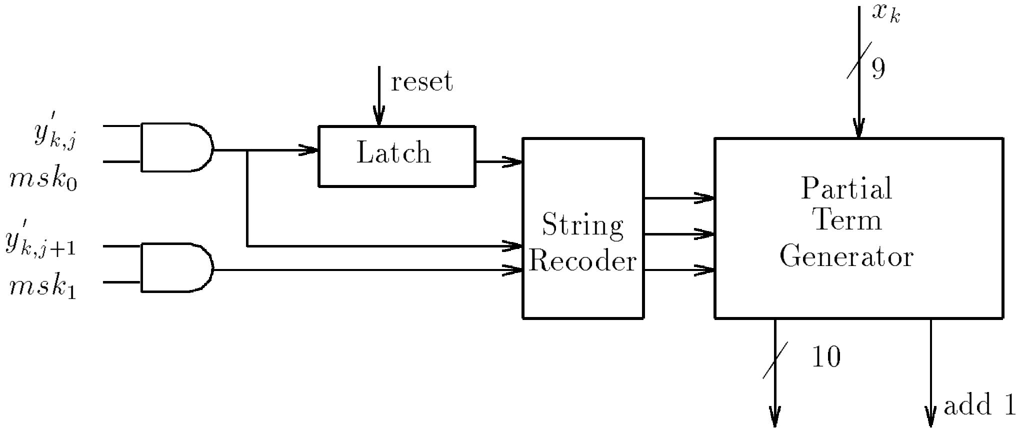


Figure B.3: 2×9 string recoded (Booth) partial product generator.

Controller

VPR has 4-bit of control information for each stored codevector. For 10-bit elements ($n = 10$) in VPR format, the exponent e would nominally require four bits. Inspections of the codebooks presented in the previous chapter show that less than 1% of the time $e \in \{8, 9, 10\}$. Limiting e to three bits, results in negligible decrease in VPR compression, while it minimizes storage. We use the fourth control bit to store an offset marker. It flags when the codevector begins in the middle of a radix-4 memory word.

As shown in Figure B.4, VPR controller has only three bits of output. It outputs the two masks needed in the radix-4 multiplication, and it flags when the execution is done. It has two parts: a variable-step down counter, and a recoder which initializes the counter. For each codevector $\mathbf{y}$ it reads $e(\mathbf{y})$ and $offset(\mathbf{y})$. For fast execution, we pipeline the recoding. In the first cycle, the only information that we need is whether codevector's first 2-bit word is aligned:

$$m_0 = offset; \quad m_1 = 1; done = 0$$

where m_0 is the mask on the LSB of the word. At the beginning of the second cycle, the initial state of the counter is loaded with w and $lone_{MSB}$. w is set to the number of non-masked words in the element ($w = \lfloor (10 - e + offset)/2 \rfloor$); $lone_{MSB}$ is set to one if the last word to be read will be half used ($lone_{MSB} = offset \oplus e_{MSB}$)). The variable-step counter is a combination of w and $lone_{MSB}$. It counts down on the number of bits of the elements to be evaluated. Finally, when the computation is done, the VPR-controller can flag the search-controller. In appendix A, we presented the VQ controller. Basically, in addition to the signals the VQ controller manages, the VPC controller deals with the terminate signals. The controller uses it to schedule a classification arbitrary cycles apart from the previous

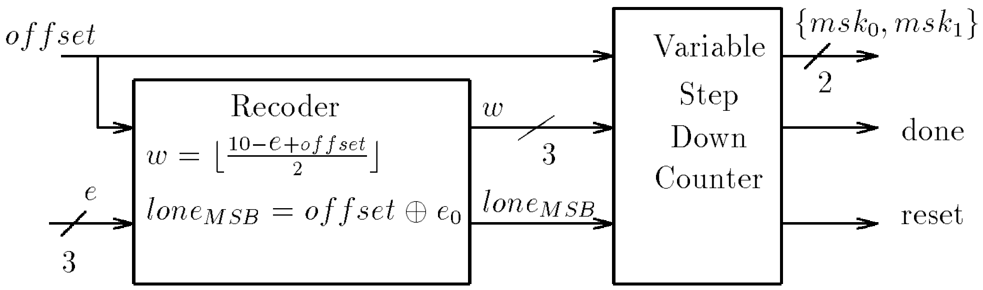


Figure B.4: VPR controller initializing a variable-step down counter for each codevector.

one.

Figure B.5 shows the input and output to VPR module. Its control section was shown in Figure B.4. The data path module is shown in detail in Figure B.6. Namely, 16 string recoded partial product generators whose output is summed and it is followed by a sign detector. The design requires 9100 gates. The critical path is calculated to be 20.7 nsec, where we have included the effect of fan-out and wiring delay had the circuitry been fabricated in 1.0-μ technology with 40% array utilization. Figure B.7 shows the implementation of masking and the string recoding.

Figure B.8 shows a sample timing diagram for VPR architecture. It contains the input and output to the implemented inner product module. The hair-line is placed between two different sections of the timing diagram. Before the hair-line the fetch registers are reset and the source vector is loaded. The hair-line marks the beginning of the three classifications. Raising of the *done* signal marks the end of each classification.

B.3 Fixed-Precision Implementation

For comparison, a conventional architecture based on multipliers and adders is designed. It evaluates the inner product as follows:

$$\begin{aligned}
\mathbf{x}^t\mathbf{y} + b &= \sum_k x_k y_k + b \\
&= \sum_{i=0}^{3} \underbrace{\sum_{k=4i+1}^{4i+4} x_k y_k + b}_{\text{hardware}}
\end{aligned}$$

where the inner summation is implemented with four multipliers and adders, and the outer summation is executed in four iteration.

As shown in Figure B.10, the conventional alternative has four string recoded multipliers whose carry-save output is summed in an adder tree along with the bias term and then

Figure B.5: VPR controller and data path.

Figure B.6: VPR data path.

Figure B.7: String recoded partial product generator.

is accumulated again in carry-save form. For $K = 16$, the evaluation completes in four cycles. To reduce memory bandwidth, the bias term b, which is 20 bits, is multiplexed. It is transmitted five bits at a time in four cycles.

For buffering the source vector, it has a circular buffer which stores four elements. As it recirculates them, it outputs the elements sequentially. To start the next vector quantization, the buffer is filled with the content of the FIFO. FIFO itself is tapped from the bus while the previous inner product is being done.

Figure B.11 is the top level schematic in the conventional architecture. As can be seen, the architecture is four string recoded multiplier which are summed and accumulated in four cycles. The design requires 9993 gates. The critical path is calculated to be 20.7 nsec, where we have included the effect of fan-out and wiring delay had the circuitry been fabricated in 1.0-μ technology with 40% array utilization. Figure B.12 shows the implementation of the multiplier. As can be seen it is comprised of four string recoded partial product generators.

Figure B.9 shows a sample timing diagram for conventional architecture. It contains the input and output to the implemented inner product module. The simulation results are divided by the hair-line. Before the hair-line, the fetch registers are reset and the source vector is loaded. The hair-line marks the beginning of the three classifications. We simulated the classification of the same source vectors shown for VPR implementation. We used equivalent clock cycles for both architecture. As can be seen for these codevectors, the total simulation time in conventional design is longer than the VPR design.

B.4 Pipelining Implementations

Pipelining of either of the designs is simple. To pipeline we need to insert the pipelining registers. Since for accumulation both designs used CSAs, there is no carry ripple to consider. With l stage pipeline, the signal along the critical path will then have l clock cycles to propagate. The other cost of pipelining is increasing the number of clock cycles for evaluation by l.

Pipelining VPR and conventional implementations are similar. First, same number of latches would be used in the pipelining. Second, the length of pipelines would be the same. At worst case, the conventional approach, which has two more levels in its summation tree, would require one more level in the pipeline.

In a pipelined architecture, two factors determine the shortest clock cycle T_s,

$$T_s = \max(T_{mem}, T_{processor})$$

where T_{mem} is the memory access time, $T_{processor}$ is the processor cycle time.

Often, when the codebook memory is off-chip the memory access time T_{mem} determines the clock cycle. In many quickly prototyped full custom circuitry, the designer opts for simplicity of a non-fully pipelined architect $T_{processor}$. Finally, pipelining due to its overhead result in a minimum stage time $T_{max\ throughput}$ which gives the maximum throughput.

Example 1: DRAM[1] memory is used.

$$\begin{aligned} T_s &= T_{mem} = T_{DRAM} \\ &\approx 60nsec \end{aligned}$$

For a given budget to be spent on memory, DRAM allows the storage of the biggest codebooks. On DRAM, when an access is local to a page of memory (≈ 256 locations), there is fast sequential access mode. Using the fast page mode, the previous DRAM can be accessed in 1/3 the time of a random access.

$$\begin{aligned} T_{mem} &= T_{DRAM\ page} \\ &\approx 30nsec \end{aligned}$$

Note that, if the memory is several times slower than the processor, we can multiplex the processor's data path and implement less circuitry.

Example 2: Processor is implemented in full custom [MJBD92]. For 2 μ LSI Logic,

$$\begin{aligned} T_s &= T_{processor} \\ &\approx 25nsec \end{aligned}$$

The available building blocks and tools limit the clock speed.

Example 3: VPR processor is down loaded into the field programmable XILINX Field Programmable Gate Array (FPGA)[Com91]. Even with a large FPGA chip[2], to fit a VQ processor requires the architecture to be modified to use the circuitry available for addition. Using the builtin carry acceleration circuitry, it reduces the number of cells, but it requires carry propagation at every level of the summation network. Moreover in FPGA technology, the full adder cell circuitry are twice as slow as the gate array technology. Combined with carry propagation, these result in a slow processor,

$$\begin{aligned} T_s &= T_{processor} = T_{carry\ propagate\ adder} \\ &\approx 35nsec;\ \text{for 12-bits} \end{aligned}$$

Example 4: We pipeline the inner product module which we have implemented. It is implemented in conventional gate array technology, where both the technology and design tools are well developed. The codebook would also be stored on-chip to reduce the access

[1] 1Mx4 AAA4M204 DRAM, NMB Technologies

[2] XILINX XC4010 with 200 configurable logic blocks

latency. The pipeline which gives the maximum throughput would have five stages ($l = 5$). We estimate the shortest clock cycle T_s as,

$$\begin{aligned} T_s &= T_{processor} = T_{3FA} \\ &\approx 10nsec \end{aligned}$$

In words, the critical path is 3 full adders. This is greater than the minimum 2 full adders used with the accumulation feedback with carry-save adders. Calculations have shown that a pipeline with 3 full adders per stage performs comparable to a pipeline with 2 full adders per stage, due to the increased latency and buffering overhead.

Figure B.8: VPR architecture timing diagram.

Figure B.9: Timing diagram for conventional architecture.

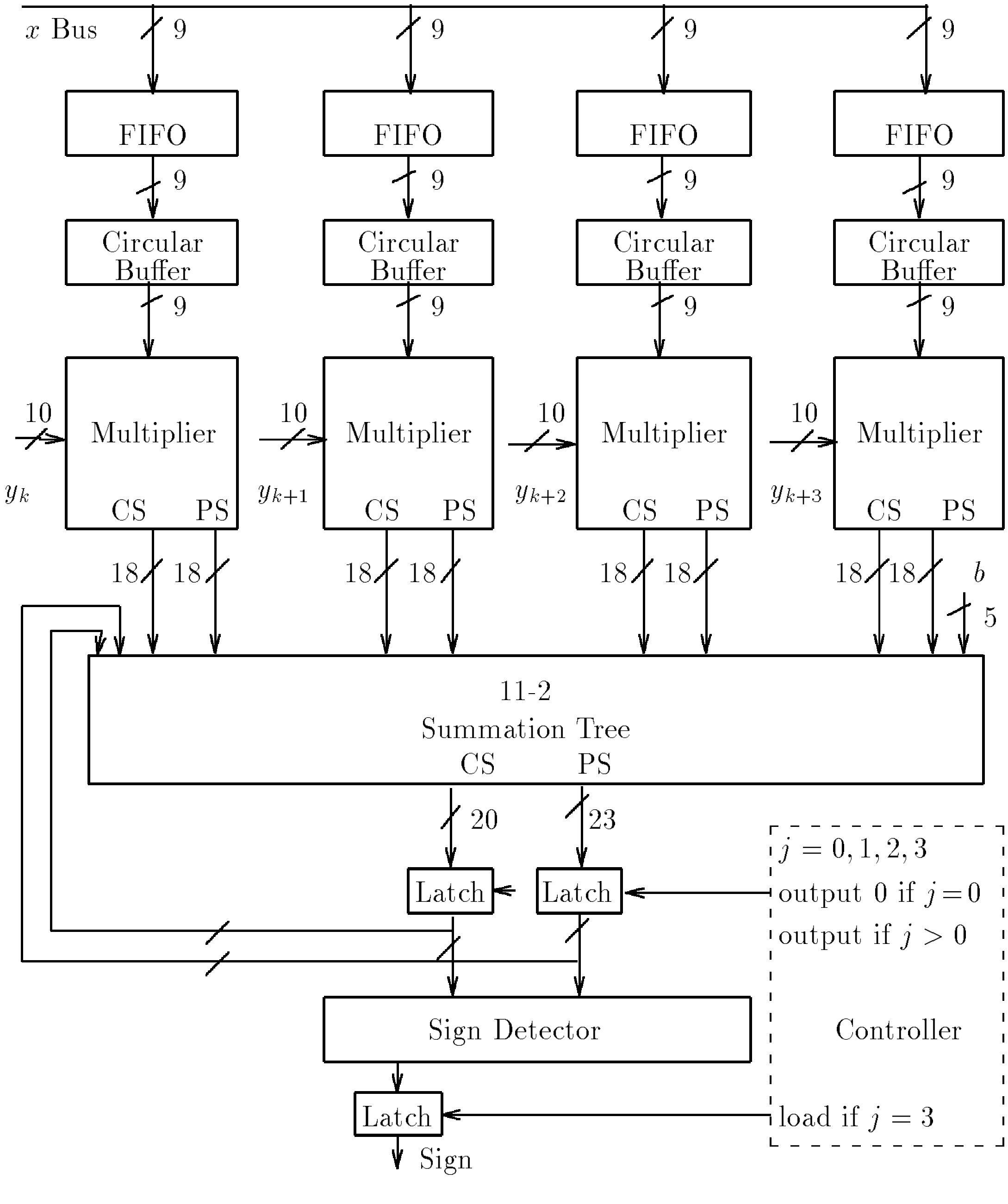


Figure B.10: Conventional inner product module implemented for comparison.

Figure B.11: Multiply-accumulate inner product architecture.

Figure B.12: String recoded multiplier.

References

[AC87] S. Adlersberg and V. Cuperman. Transform domain vector quantization. In *ICASSP*, pages 1938–1941, 1987.

[ATS87] H. Abut, B. P. M. Tao, and J. L. Smith. Vector quantizer architectures for speech and image coding. In *IEEE Conference on ASSP*, Dallas, Texas, April 1987.

[Bak84] R. L. Baker. *Vector Quantization of Digital Images.* Stanford University, Electronics Laboratory, Stanford University, Stanford, CA 94305, 1984.

[BB78] R. C. Buck and E. F. Buck. *Advanced Calculus.* McGraw-Hill, Inc., New York, 1978.

[BDB+92] M. A. Bree, D. E. Dodds, R. J. Bolton, S. Kumar, and B. L. F. Daku. A modular bit-serial architecture for large-constaint-length Viterbi decoding. *IEEE Journal of Solid-State Circuits*, pages 184–190, Februrary 1992.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Comm. ACM*, pages 509–517, September 1975.

[BG85] C. Bei and R. M. Gray. An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Trans. on Communication*, pages 1132–1133, October 1985.

[BGGM80] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel. Speech coding based vector quantization. *IEEE Trans. on ASSP*, pages 562–574, October 1980.

[CCW91] R. Chang, W. Chen, and J. Wang. Image sequence coding using adaptive tree-structured vector quantization with multipath searching. In *IEEE Int. Conf. Acous., Speech, and Signal Proc.*, pages 2281–2284, 1991.

[CLG88] P. A. Chou, T. Lookabaugh, and R. M. Gray. Optimal pruning with applications to tree-structured source coding and modeling. *IEEE Trans. Info Theory*, 1988.

[Com91] The Programmable Gate Array Company. *The XC4000 DATABOOK.* XILINX, 2100 Logic Drive, San Jose, CA 95124, 1991.

[Cor91] LSI Logic Corporation. *1.0-Micron Array-Based Products Databook.* XILINX, 1551 McCarthy Boulevard, Milpitas, CA 95035, 1991.

[DB87] R. Dianysian and R. L. Baker. A VLSI chip set for real time vector quantization of image sequences. In *IEEE International Symposium on Circuits and Systems*, Philadelphia, 1987.

[DCG88] G. A. Davidson, P. R. Cappello, and A. Gerscho. Systolic architectures for vector quantization. *IEEE Trans. on ASSP*, pages 1651–1664, October 1988.

[DE90] R. Dionysian and M. D. Ercegovac. Variable-precision linear classifier. *IEEE Workshop on VLSI Signal Processing*, Vol. IV:285–294, 1990.

[DG86] G. Davidson and A. Gerscho. Application of a VLSI vector quantization processor to real-time speech coding. *IEEE J. Selected Areas in Commun.*, pages 112–124, January 1986.

[DH73] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, Inc., New York, 1973.

[DHM89] J. Duprat, Y. Herreros, and J. Muller. Some results about on-line computation of functions. In *Proceedings of 9th Symposium on Computer Arithmetic*, pages 112–118, 1989.

[DJK92] K. Dezhgosha, M. M. Jamali, and S. C. Kwatra. A VLSI architecture for real-time image coding using a vector quantization based algorithm. *IEEE Trans. on Signal Processing*, pages 181–189, January 1992.

[EL88] M. D. Ercegovac and T. Lang. On-line arithmetic: A design methodology and applications in digital signal processing. *IEEE Workshop on VLSI Signal Processing*, Vol. III:252–263, 1988.

[FCS90] W. Fang, C. Chang, and B. J. Sheu. Systolic tree-searched vector quantizer for real-time image compression. In *VLSI Signal Processing IV*, pages 352–361, 1990.

[GBJM79] R. M. Gray, A. Buzo, A. H. Gray Jr., and Y. Matsuyama. Distortion measures for speech processing. *Proceedings of the IEEE*, pages 773–785, May 1979.

[Gra84] R. M. Gray. Vector quantization. *IEEE ASSP Magazine*, pages 4–29, April 1984.

[HH88] Hsueh-Ming Hang and Barry G. Haskell. Interpolative vector quantization of color images. *IEEE Trans. Commun.*, COM-36:465–470, April 1988.

[Hol90] E. A. Holmberg. *Design of a Vector Quantizer Board*. UCLA, Electrical Engineering Department, UCLA, Los Angeles, CA, 1990.

[HV92] P. G. Howard and J. S. Vitter. Parallel lossless image compression using huffman and arithmetic coding. In *Proceedings of Data Compression Conference*, pages 299–308, Snowbird,Utah, 1992.

[IS68] F. Itakura and S. Saito. Analysis synthesis telephone based upon the maximum likelihood method. In *Conf. Rec. 6th Int. Congress Acoust.*, Tokyo, 1968.

[JN84] N. S. Jayant and P. Noll. *Digital Coding of Waveforms*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Kav86] R. A. Kavalar. *The Design and Evaluation of a Speech Recognition System for Engineering Workstations*. University of California, Electronics Research Laboratory, University of California, Berkeley, CA 94720, 1986.

[KBSC92] S. Z. Kiang, R. L. Baker, G. J. Sullivan, and C. Y. Chiu. Recursive optimal pruning with applications to tree-structured vector quantizers. *IEEE Trans. on Image Processing*, pages 162–169, April 1992.

[Kog81] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Book Company, 1981.

[KYJ93] R. K. Kolagotla, S. S. Yu, and J. F. JaJa. VLSI implementation of a tree searched vector quantizer. *IEEE Trans. Signal Processing*, pages 901–905, February 1993.

[LBG80] Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Trans. Commun.*, COM-28:84–95, January 1980.

[Mad90] A. Madisetti. *Search Algorithms for Vector Quantization*. UCD, Electrical Engineering Department, UCD, Davis, CA, 1990.

[MJBD92] A. Madisetti, R. Jain, R. L. Baker, and R. Dianysian. Architectures and integrated circuits for real time vector quantization of images. *IEEE Int. Conf. Acous., Speech, and Signal Proc.*, 5:677–681, 1992.

[MK92] J. W. Modestino and Y. H. Kim. Adaptive entropy-coded predictive vecor quantization of images. *IEEE Trans. on Signal Processing*, pages 633–644, March 1992.

[Mok89] N. Mokhoff. HVQ advances video technology. (hierarchical vector quantization). *Electronic Engineering Times*, n561:33–35, October 1989.

[MRG85] J. Makhoul, S. Roucos, and H. Gish. Vector quantization in speech coding. *Proceedings of the IEEE*, pages 1551–1588, November 1985.

[MS74] J. L. Mannos and D. J. Sakrison. The effect of visual fidelity criterion on the encoding of images. *IEEE Trans. on Info. Theory*, pages 525–536, July 1974.

[NJ85] L. M. Ni and A. K. Jain. A VLSI systolic architecture for pattern clustering. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, pages 80–89, January 1985.

[NK88] N. M. Nasrabadi and R. A. King. Image coding using vector quantization: A review. *IEEE Trans. Communications*, pages 957–971, August 1988.

[NR86] B. E. Nelson and C. J. Read. A bit-serial VLSI vector quantizer. In *IEEE Conf. on ASSP*, pages 2211–2214, Tokyo, 1986.

[Opp70] A. V. Oppenheim. Realization of digital filters using block floating point arithmetic. *IEEE Trans. on Audio and Electroacoustics*, pages 130–136, June 1970.

[PFTV88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 32 East 57th Street, New York, NY 10022, 1988.

[PL74] A. Peled and B. Liu. A new hardware realization of digital filters. *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-22:456–462, December 1974.

[RPT89] P. A. Ramamoorthy, B. Potu, and T. Tran. Bit-serial VLSI implementation of vector quantizer for real-time image coding. *IEEE Trans. on Circuits and Systems*, pages 1281–1290, October 1989.

[SM87] M. R. Soleymani and S. D. Morgera. A high-speed search algorithm for vector quantization. In *ICASSP*, pages 1946–1948, 1987.

[TAG84] B. P. M. Tao, H. Abut, and R. M. Gray. Hardware realization of waveform vector quantizers. *IEEE J. Selected Areas in Commun.*, pages 343–352, March 1984.

[TW91] J. F. Traub and H. Wozniakowski. Information-based complexity: New questions for mathematicians. *The Mathematical Intelligencer*, 13(2):34–43, 1991.

[WBBW88] Peter H. Westerink, Dick E. Boekee, Jan Biemond, and John W. Woods. Subband coding of images using vector quantization. *IEEE Trans.*, COM-36:713–719, June 1988.

[Wha71] A. D. Whalen. *Detection of Signals in Noise.* Academic Press, Inc., Orlando, Florida 32887, 1971.

[Wid87] B. Widrow. The original adaptive neural net broom-balancer. In *IEEE International Symposium on Circuits And Systems*, pages 351–357, 1987.