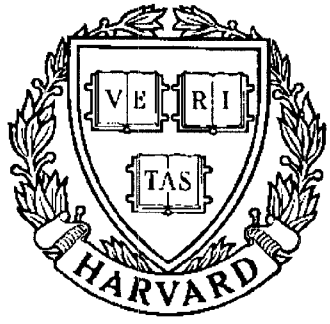
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

VLSI Architectures and Implementation of Predictive Tree-Searched Vector Quantizers for Real-Time Video Compression

by S-S. Yu, R.K. Kolagotla and J.F. JáJá

TR 92-48

VLSI Architectures and Implementation of Predictive Tree-Searched Vector Quantizers for Real-Time Video Compression [1]

Shu-Sun Yu, Ravi K. Kolagotla and Joseph F. JáJá

Department of Electrical Engineering,
Systems Research Center, and
Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742.

Abstract

We describe a pipelined systolic architecture for implementing Predictive Tree-Searched Vector Quantization (PTSVQ) for real-time image and speech coding applications. This architecture uses identical processors for both the encoding and decoding processes. The overall design is regular and the control is simple. Input data is processed at a rate of 1 pixel per clock cycle, which allows real-time processing of images at video rates. We implemented these processors using $1.2\mu m$ CMOS technology. Spice simulations indicate correct operation at 40 MHz. Prototype version of these chips fabricated using $2\mu m$ CMOS technology work at 20 MHz.

[1]This research was supported in part by the National Science Foundation Engineering Research Center Program, NSFD CD 8803012, and by the Ford Aerospace Corporation, MIPS contract 121.45.

1 Introduction

Vector Quantization (VQ) is an important data compression technique used in image coding systems for efficiently compressing images. Most real-time signal processing applications such as HDTV, transmission of images from satellites, and storage of images require the handling of enormous amount of data. Data compression has emerged as an important tool to reduce the memory storage and bandwidth requirements for these applications[1, 2, 3].

According to Shannon's rate-distortion theory, the performance of a Vector Quantizer approaches the rate-distortion bound at a given rate when the vector dimension is allowed to grow arbitrarily large. VQ provides better performance than scalar quantization and has been extensively investigated [2, 3]. However, the computational complexity of VQ increases exponentially with an increase in vector dimension. Most memoryless VQs are designed for small dimension vectors, because it is difficult to design and implement VQs for large vector dimensions. On the other hand, VQs with memory exploit interblock correlation, and hence provide better performance than memoryless VQs of the same vector dimension.

In the last decade, a considerable amount of work has been done in developing different VQ schemes with memory, such as Predictive VQ (PVQ)[4, 5, 6, 7, 8] and Finite-State VQ (FSVQ)[9, 10, 11]. Predictive VQ is a class of VQ with memory which exploits the correlation between neighboring vectors to achieve better performance for a given bit-rate and vector dimension. The basic idea of this scheme is to remove the mutual redundancy between adjacent vectors and encode only the new information. Memoryless VQ does not

take advantage of the inter-vector correlation. PVQ can perform better than a memoryless VQ of the same rate. Although nonlinear prediction can provide superior performance, we concentrate on linear prediction in this paper because of its simplicity and effectiveness [1]. Predictive TSVQ (PTSVQ) is a PVQ which uses a Tree-Searched VQ (TSVQ) as a building block. PTSVQ has an $O(\log N)$ codebook search complexity and provides an efficient way to compress the input data.

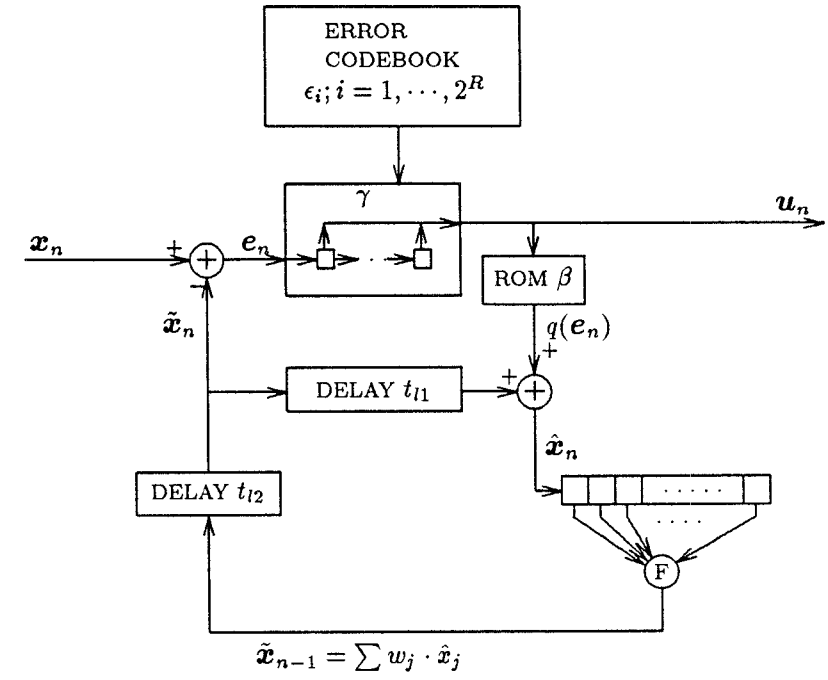
The computational requirements for real-time video processing are quite intensive. It is desirable to realize compact high-performance processors that can achieve a very high throughput. A typical example is to handle images of size 1024×1024 pixels at a rate of 30 frames/sec in real-time. Rapid advances in VLSI technology make it possible to realize various VQ schemes using a small chip area at a low cost. In this paper, we present the design and implementation of efficient VLSI architectures for Predictive VQ to achieve this goal. These architectures use a Tree-Searched VQ (TSVQ) as a building block and achieve real-time performance at a low cost. In Section 2, we give the basic definitions and background of the predictive vector quantization. In Section 3, we describe the basic architecture of a TSVQ, whereas architectures suitable for realizing PTSVQ are presented in Section 4. A bit-level systolic architecture and its implementation in VLSI is given for the PTSVQ.

2 Preliminaries

A VQ involves a scheme for mapping an L-dimensional input vector onto an output or reproduction vector selected from a precomputed set of codevectors. It consists of two parts: an encoder and a decoder. The encoder compares the input vector with every codevector in the codebook and returns the index of the codevector that is the closest approximation of the input vector under some error measurement criterion. The decoder maps this index onto its corresponding codevector from the codebook.

The straightforward codebook search process in VQ encoding requires highly intensive computation; for a codebook of N codevectors, it requires the evaluation of $\Theta(N)$ distortion measures to isolate the desired codevector. However, if the codebook is structured as a tree, the computational complexity is reduced from $\Theta(N)$ to $O(\log N)$ [2, 12, 13, 14, 15, 16, 17, 18]. This reduction in computational complexity results in a sub-optimal VQ.

A Predictive VQ is a VQ with memory which has been extensively studied in recent years [4, 5, 6, 7, 8, 19, 20]. Predictive VQ (PVQ) makes use of interblock correlation to predict the current input vector based on past outputs; it then vector quantizes the difference between the actual input and its predicted value. We concentrate here on the case where the difference is quantized using a TSVQ. The resulting system is called a Predictive TSVQ (PTSVQ). As mentioned earlier, TSVQ is a sub-optimal VQ which trades off computational complexity for performance. It is easily seen that PTSVQ has a much smaller complexity than PVQ with only a minor performance degradation [1, 8].



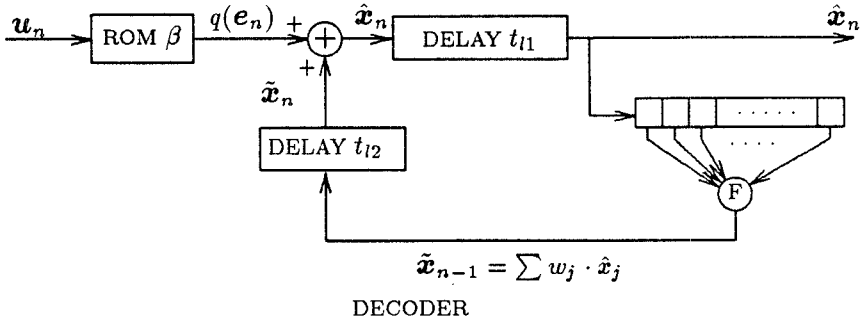


Figure 1: Scheme of PVQ.

A block diagram of the PVQ system is shown in Fig. 1. Predictive VQ (PVQ) can be viewed as a straightforward vector extension of the traditional scalar predictive quantization or Delta Pulse Code Modulation (DPCM). In the encoder, a predicted vector is formed from the past reconstructed vectors, $\hat{\boldsymbol{x}}_{n-1}$, $\hat{\boldsymbol{x}}_{n-2}, \cdots$. An error vector, $\boldsymbol{e}_n$, is generated based on the difference between the predicted vector $\tilde{\boldsymbol{x}}_n$ and the actual input vector $\boldsymbol{x}_n$. This error vector is quantized using a memoryless VQ. Then, the index is transmitted over a channel.

In the decoder, this error vector is recovered from the received channel index by table lookup. The original vector is reconstructed by adding this error vector to its corresponding predicted vector. A PVQ system [4] can be formally defined as follows:

1. An encoder γ which is a memoryless VQ that assigns to each error vector, $\boldsymbol{e}_n = \boldsymbol{x}_n - \tilde{\boldsymbol{x}}_n$, an index symbol u_n from an index set M to identify the closest codeword in codebook $\hat{B}$.

2. A decoder β which is a mapping that assigns to each index u_n a vector in a reproduction codebook $\hat{B}$.

3. A prediction function f which predicts the input vector $\tilde{\boldsymbol{x}}_n$ based on the previous reconstructed inputs, and hence $\tilde{\boldsymbol{x}}_n = f(\hat{\boldsymbol{x}}_{n-1}, \hat{\boldsymbol{x}}_{n-2}, \cdots)$. Typically, only finite order, say p, of prediction is assumed to be used, i.e. the above expression can be simplified as $\tilde{\boldsymbol{x}}_n = f(\hat{\boldsymbol{x}}_{n-1}, \hat{\boldsymbol{x}}_{n-2}, \cdots, \hat{\boldsymbol{x}}_{n-p})$.

Given a sequence of input vectors and an initial prediction $\tilde{\boldsymbol{x}}_1$, the index sequence u_n, reproduction sequence $\hat{\boldsymbol{x}}_n$, and prediction sequence $\tilde{\boldsymbol{x}}_{n+1}$ for

$n = 1, 2, \cdots$ are defined recursively as follows:

$$\begin{aligned} u_n &= \gamma(\boldsymbol{e}_n) = \gamma(\boldsymbol{x}_n - \tilde{\boldsymbol{x}}_n) \\ \hat{\boldsymbol{x}}_n &= \tilde{\boldsymbol{x}}_n + \beta(u_n), \\ \tilde{\boldsymbol{x}}_{n+1} &= f(\hat{\boldsymbol{x}}_n, \hat{\boldsymbol{x}}_{n-1}, \cdots). \end{aligned}$$

For a *linear prediction function of finite order* p, $\tilde{\boldsymbol{x}}_n$ can be expressed as

$$\tilde{\boldsymbol{x}}_n = \sum_{i=1}^{p} A_i \hat{\boldsymbol{x}}_{n-i},$$

where A_i is an $L \times L$ predictor matrix for an L-dimensional vector.

3 Overall Architecture

In this section, we describe the mapping of PTSVQ onto a VLSI architecture for real-time image coding. This system consists of:

1. TSVQ for encoding the difference between the predicted vector and the input vector into a channel index,

2. Inverse TSVQ (ITSVQ) for decoding the channel index into its corresponding codevector, and

3. Predictor Processor (PP) which computes the residual vector, and executes the prediction process.

The overall architecture of the PTSVQ is shown in Fig. 2. This architecture consists of a Predictor Processor, a linear array of SNP processors which realize a binary TSVQ, and an Inverse TSVQ (ITSVQ). There are three types

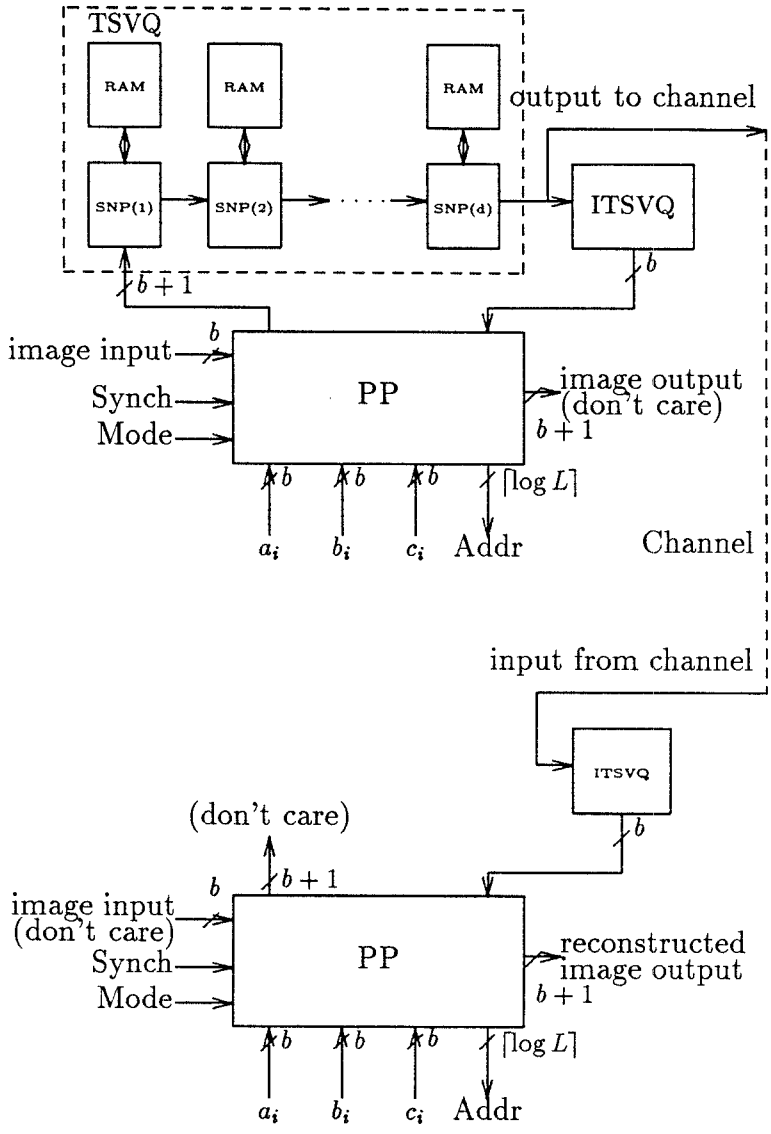


Figure 2: Architecture for PTSVQ system. PP is the Predictor Processor to perform the prediction function for the incoming vectors. PP is identical in both the encoder and the decoder.

of basic building blocks in the system, namely the Predictor Processor (PP), the Single Node Processor (SNP) used for realizing TSVQ, and the ITSVQ processor. The ITSVQ chip can be implemented as a table lookup, using either a ROM or a PLA. The Predictor Processor subtracts the predicted vector from the input vector, buffers the past vectors, and generates the predicted vector according to the specified prediction function. Due to the similarity between the encoding and decoding parts of the PTSVQ, the PP can be used either as an encoder or as a decoder without additional circuitry. The SNP performs the distortion computation corresponding to a node of a binary TSVQ. We now describe each of these blocks in detail.

3.1 Single Node Processor

In a binary TSVQ, the L-dimensional input vector $\boldsymbol{x} = (x_1, \ldots, x_L)^T$ must be compared with two codevectors at each node. Let $\boldsymbol{c}_1 = (c_{1,1}, \ldots, c_{1,L})^T$, and $\boldsymbol{c}_2 = (c_{2,1}, \ldots, c_{2,L})^T$ represent the two vectors in the codebook of a given node. The processing performed at each node is reduced to testing the condition:

$$d(\boldsymbol{x}, \boldsymbol{c}_1) \geq d(\boldsymbol{x}, \boldsymbol{c}_2), \tag{1}$$

where d is the distortion measure.

The weighted mean-squared error distortion is specified by

$$d(\boldsymbol{x}, \boldsymbol{c}_i) = (\boldsymbol{x} - \boldsymbol{c}_i)^T \boldsymbol{W} (\boldsymbol{x} - \boldsymbol{c}_i), \quad i = 1, 2,$$

where $\boldsymbol{W}$ is the weighting matrix. Equation (1) can be expressed as:

$$(\boldsymbol{x} - \boldsymbol{c}_1)^T \boldsymbol{W} (\boldsymbol{x} - \boldsymbol{c}_1) - (\boldsymbol{x} - \boldsymbol{c}_2)^T \boldsymbol{W} (\boldsymbol{x} - \boldsymbol{c}_2) \quad \geq 0 \tag{2}$$

If equation (2) is satisfied, the input vector $\boldsymbol{x}$ is closer to codeword $\boldsymbol{c}_2$. Otherwise $\boldsymbol{x}$ is closer to $\mathbf{c_1}$. We expand equation (2) to obtain:

$$\sum_{j=1}^{L} \{\alpha_j x_j\} + \beta \quad \geq 0 \tag{3}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_L) = 2(\boldsymbol{c}_2 - \mathbf{c_1})^T \boldsymbol{W}$, and $\beta = \mathbf{c_1}^T \boldsymbol{W} \mathbf{c_1} - \boldsymbol{c}_2^T \boldsymbol{W} \boldsymbol{c}_2$. For the special case of the mean-squared error distortion measure, $\boldsymbol{W} = \boldsymbol{I}$, and hence $\alpha_j = 2(c_{2,j} - c_{1,j})$, and $\beta = \sum_{j=1}^{L}(c_{1,j}^2 - c_{2,j}^2)$.

Instead of using the raw codebook online, we can determine these α and β coefficients off-line and store them in memory chips[2]. This algorithm is based on Binary Hyperplane Testing [21]. Directly implementing equation (1) requires $2(L^2 + L)$ multiplications, $2(L^2 - 1)$ additions and $L^2 + L$ words of memory storage, while implementing equation (3) requires only L multiplications, L additions, and $L + 1$ words of memory storage.

Several different multiplier designs are available for digital signal processing applications. They offer trade-offs between speed and complexity[22]. Our goal here is to map the architecture down to the bit level such that the entire system can handle the input image in real-time at a low cost. The pipelined parallel multiplier developed by McCanny and McWhirter[23] is best suited for our purpose. It is a bit-level array multiplier such that addition can be directly incorporated into the multiplication process. Fig. 3 shows an example of a 3×3 array multiplier. It performs the operation $a \times b + s$. The regularity and simplicity of the basic cell and the absence of

[2]Some applications use a weighting matrix $\boldsymbol{W}(\boldsymbol{x})$ that depends on the input vector $\boldsymbol{x}$. Equation (3) is still valid in this case, but a preprocessor is needed to compute the α and β coefficients in real-time.

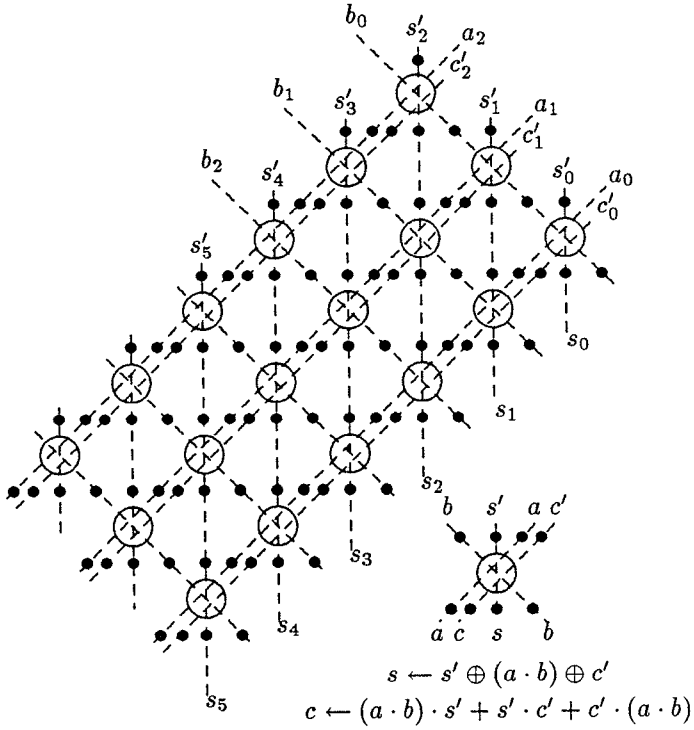


Figure 3: Example of a 3 × 3 pipelined parallel multiplier.

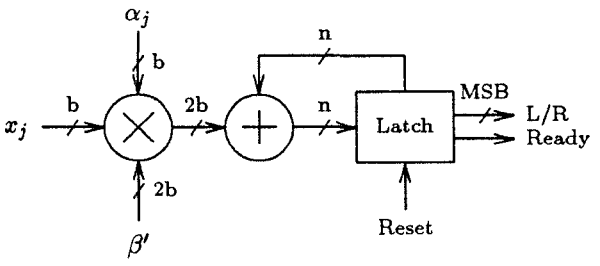


Figure 4: Block diagram of the Single Node Processor (SNP). Coefficients α_j and β' are stored in off-chip memories. The multiplier computes $p_j = \alpha_j x_j + \beta'$.

global communication make this multiplier attractive for VLSI implementation.

Fig. 4 shows a block diagram of the Single Node Processor (SNP). The SNP performs the computations stated in equation (3). The inner product of two input sequences of size L are calculated. The SNP's output is a '0' if equation (3) is satisfied and a '1' otherwise. We do not need a comparator unit in the SNP. The most significant bit (MSB) of the accumulated products directly represents the processor's output.

The multiplier takes two b-bit numbers α_j and x_j, and a $2b$-bit number β', and returns a $2b$-bit number $p_j = \alpha_j x_j + \beta'$. We define $\beta' = \beta/L$ and add it during each of the L multiplication steps. This can be done without any additional hardware and eliminates the need for a comparator unit to compare the accumulated sums with β. The bits of $p_j = p_{j,2b}, p_{j,2b-1}, \ldots, p_{j,1}$ are available in a skewed fashion, least significant bit (LSB) first. The latency of the multiplier depends on the bit position; it is b for the LSB bit $p_{j,1}$, and

$3b$ for the MSB bit $p_{j,2b}$. The accumulator must have a precision of

$$n = 2b + \lceil \log L \rceil$$

bits, to prevent overflow when L $2b$-bit numbers are added together. The output of the multiplier is sign extended by $\lceil \log L \rceil$ bits and is directly applied to the accumulator.

The accumulator consists of a linear array of cells, and operates on skewed input data [15]. The accumulator computes

$$S = \sum_{j=1}^{L} p_j,$$

and returns the sign of S. A Reset signal is generated once every L clock cycles. Reset is propagated along the array and each cell is reset in turn. This allows the next set of L numbers to be accumulated immediately after the last number of the current set is applied to the accumulator. The latency of the accumulator is $n + L$ clock cycles. Hence, the latency of the SNP is

$$L_{SNP} = b + n + L = 3b + \lceil \log L \rceil + L. \tag{4}$$

For example, if the word size $b = 8$, and the vector dimension $L = 16$, we have $L_{SNP} = 44$ clock cycles.

3.2 TSVQ architecture

The computations performed by a TSVQ can be viewed as finding a path from the root to a leaf in a binary tree. While traversing a binary tree, only one node is encountered at each level. Hence, the computations at each level

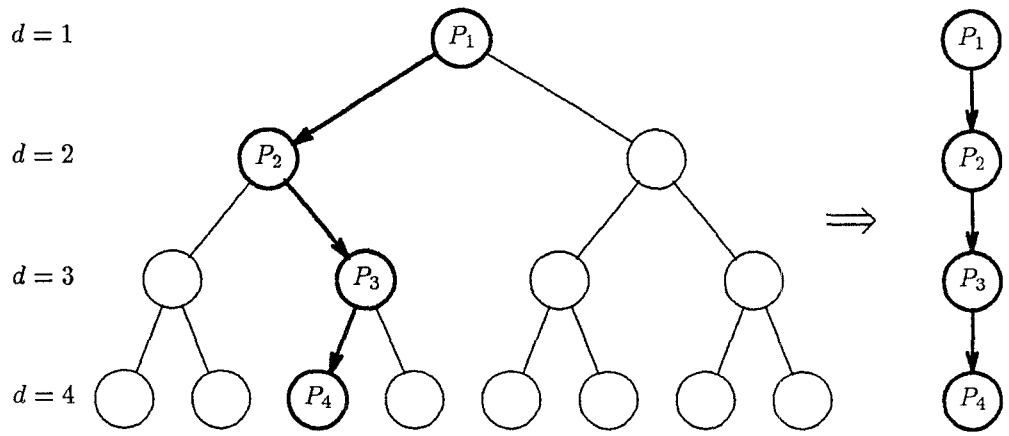


Figure 5: Traversal of a binary tree of depth $d = 4$, and its mapping onto a linear array of processors.

can be performed by a single processor. A tree of depth d can be mapped onto a linear array of d processors as shown in Fig. 5.

Fig. 6 shows the architecture of a TSVQ using d SNP processors. The coefficients necessary for each processor's computations are stored in memories and will in general depend on the distortion measure used. Processor SNP(i) adds the results of its computations to a partial index datapath and generates a Go signal to initiate processing by processor SNP(i+1). This Go signal is used to reset the accumulator. The final processor SNP(d-1), returns the complete index u. The memory bandwidth is $3b$ for each processor. (Memory bandwidth can be reduced from $3b$ to b by preloading β' into on-chip registers.) The size of the memory is different for different processors. The first processor needs a memory of $L + 1$ words to store β' and the L components of α_j. Processor SNP(i+1) needs twice as much memory as pro-

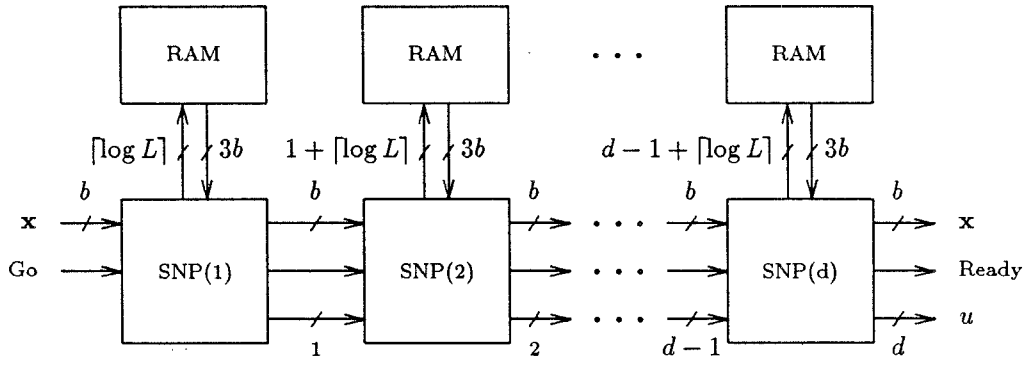


Figure 6: Systolic architecture for computing TSVQ. Each processor adds its partial index to the index data-path, and generates a control signal to initiate processing by its neighbor down the tree. No global control signals are needed.

cessor SNP(i). The last processor needs a memory of $2^{d-1}(L+1)$ words. The throughput of this scheme is one L-dimensional vector per L clock cycles. The total latency of the TSVQ is

$$L_{TSVQ} = dL_{SNP} = d(b + n + L).$$

For a tree of depth d=8, and a vector dimension of $L = 16$ (which corresponds to a bit-rate of 0.5 bit/pixel), we have $L_{TSVQ} = 352$ clock cycles. This architecture has been implemented using $2\mu m$ N-well technology; it is described in detail in[15].

Recently, alternate TSVQ architectures have been reported in the literature [18, 24]. The scheme in [18] uses a VQ slave processor and a controller to implement arbitrary depth pruned trees. This architecture has a smaller throughput than our TSVQ architecture for the same clock rates. For the example considered above, this architecture would have a throughput of only 1 pixel every 4.5 clock cycles. In addition, a complicated controller is required

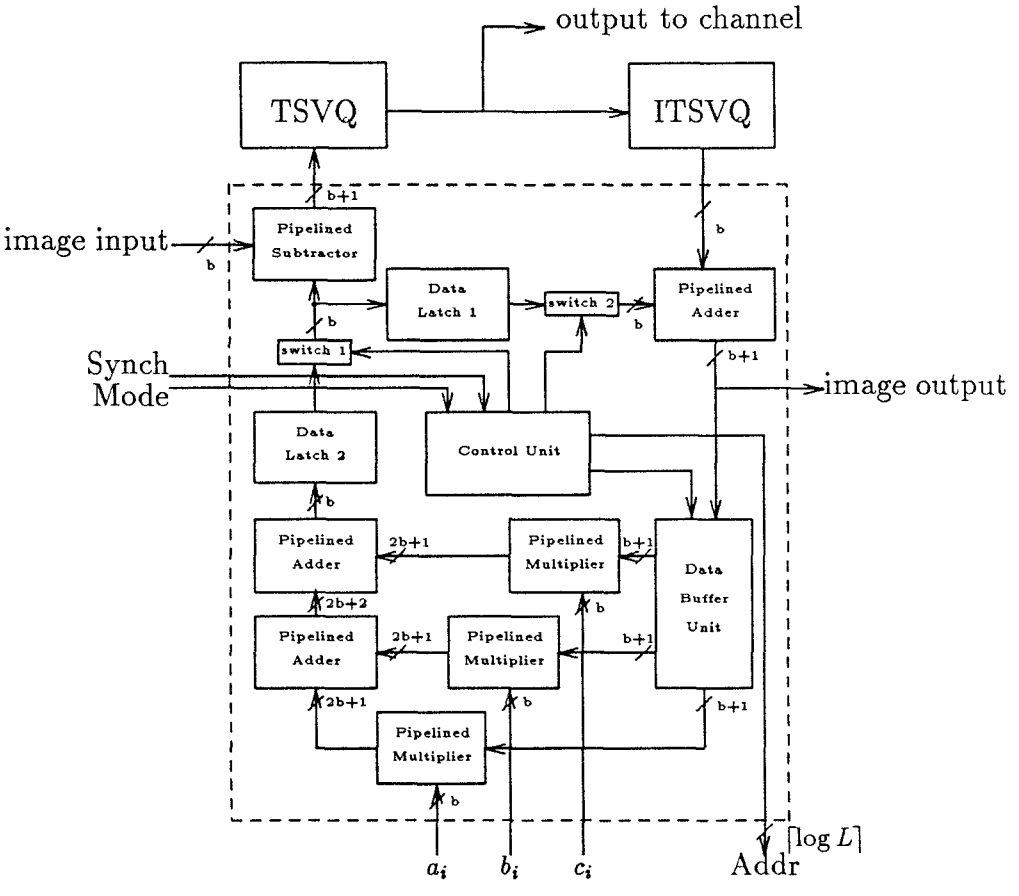


Figure 7: Block diagram of the Predictor Processor.

by this scheme, while no global control signals are needed by our TSVQ architecture. A multiplier free architecture was proposed in [24]. However, this scheme uses a clustering technique that results in an approximate codebook. Its performance compared to the LBG technique is not known.

3.3 PTSVQ architecture

In this section, we describe how the PTSVQ can be built using a TSVQ as a building block. A detailed block diagram of the Predictor Processor is shown in Fig. 7. A pipelined subtractor is used to subtract the values of the predicted vector from the corresponding input vector. The bit-level adder/subtractor, shown in Fig. 8, operates on skewed input data in bit-serial fashion. The result, i.e. the difference or residue vector, is then deskewed and sent to the TSVQ to generate the channel index. The predicted vector is delayed by the latency of the TSVQ and the ITSVQ modules. This delayed vector is added to the output of the ITSVQ module to generate the reconstructed vector. This reconstructed vector is then fed into a data buffer unit. The data buffer unit correctly taps the pixel values from these vectors for the linear prediction module as explained next.

Image Input Format

For image compression, we consider an input image of size $N \times M$ pixels such that each pixel is represented as a b-bit number. The input image frame is partitioned into small subblocks each of size $k \times k$. Each input frame contains $\frac{N}{k} \times \frac{M}{k}$ subblocks and each subblock can be treated as a vector, $\boldsymbol{x}$, of dimension $L = k^2$. Typically, the size of the subblock is 4×4 or 8×8 pixels. A sequence of vectors is formed by raster scanning along the consecutive rows of subblocks. Within each subblock, the pixels are scanned from left to right and top to bottom as shown in Fig. 9. This sequence of input subblocks can be treated as a vector-valued random process $\{\boldsymbol{x}_n\}_{n=1}^{NM/k^2}$.

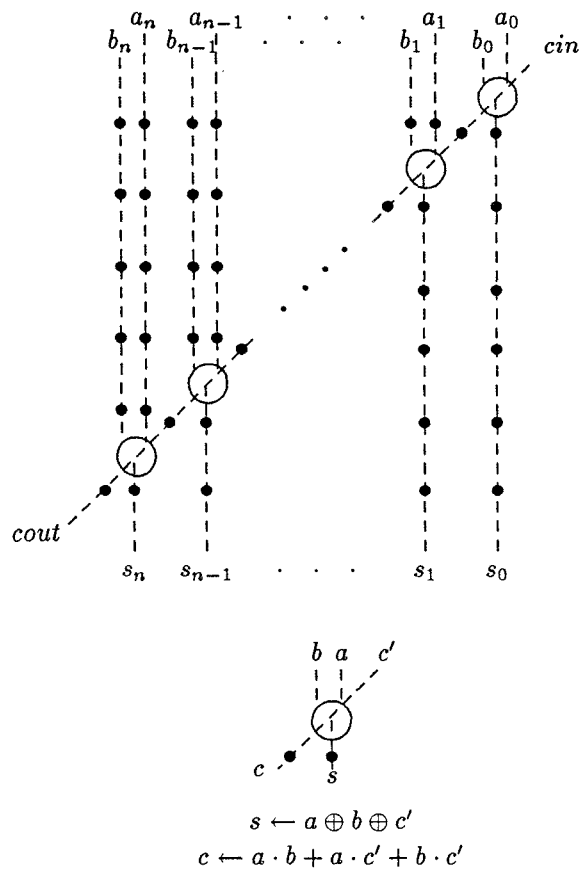


Figure 8: $n \times n$ pipelined bit-serial adder with skew and deskew latches.

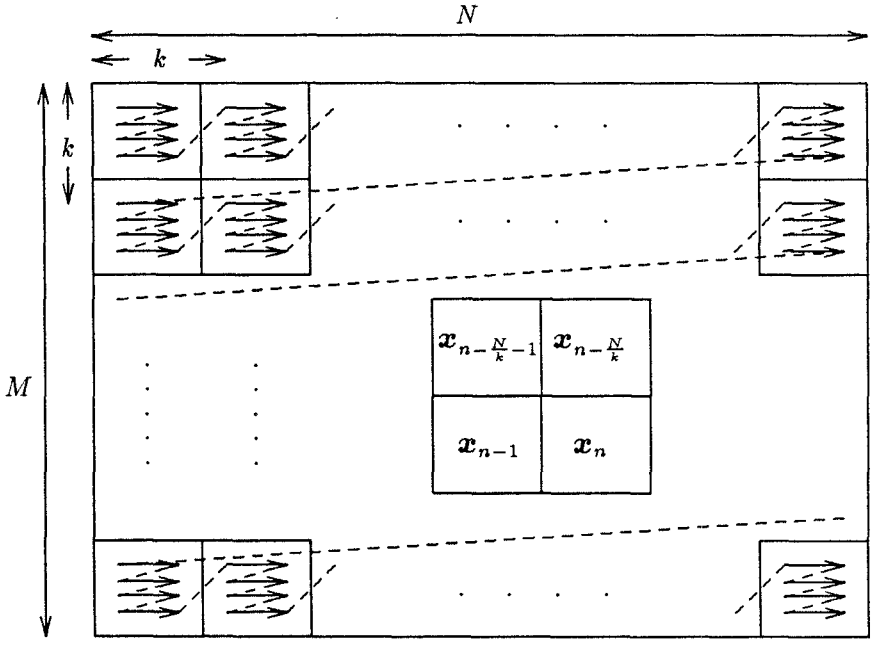


Figure 9: Block scan of an input image frame of size $N \times M$.

Linear Predictor Module

For each input vector, $\boldsymbol{x}_n = (x_1, x_2, \cdots, x_L)^T$, the nearest causal neighbors are $\boldsymbol{x}_{n-\frac{N}{k}-1}$, $\boldsymbol{x}_{n-\frac{N}{k}}$ and $\boldsymbol{x}_{n-1}$. Though different forms of linear prediction are possible, we choose the following form[8]. Each pixel $x_{i,j}$, $i, j = 1, \cdots, k$, within the vector is predicted using a linear function of past pixels as follows:

$$\tilde{x}_{i,j} = a_{i,j} \cdot y_{k,j} + b_{i,j} \cdot y_{k-1,j} + c_{i,j} \cdot y'_{k-1,k-1} + d_{i,j} \cdot y''_{i,k-1} + e_{i,j} \cdot y''_{i,k-1},$$

where $y_{k-1,j}$ and $y_{k,j}$ are the nearest pixels in the same column from the north subblock $\boldsymbol{x}_{n-\frac{N}{k}}$, $y''_{i,k-1}$ and $y''_{i,k}$ are the nearest pixels in the west subblock $\boldsymbol{x}_{n-1}$ and $y'_{k,k}$ is the lower right corner pixel of the north-west subblock $\boldsymbol{x}_{n-\frac{N}{k}-1}$ as shown in Fig. 10. Since $\tilde{x}_{i,j}$ is formed from a linear combination of pixels $y_{k,j}$, $y_{k-1,j}$, the quantization of subblock $\boldsymbol{x}_n$ cannot be started until subblocks $\boldsymbol{x}_{n-\frac{N}{k}-1}$, $\boldsymbol{x}_{n-\frac{N}{k}}$ and $\boldsymbol{x}_{n-1}$ are completely quantized. If the correlation between subblocks $\boldsymbol{x}_n$ and $\boldsymbol{x}_{n-1}$ is ignored, the PTSVQ system can be pipelined efficiently. The decrease in performance due to ignoring the west subblock is small over most rates [8]. Hence, we use a simpler 3-order prediction function which does not depend on pixels $y''_{i,k-1}$ and $y''_{i,k}$ in our PTSVQ architecture. Here, $\tilde{x}_{i,j}$ is defined as:

$$\tilde{x}_{i,j} = a_{i,j} \cdot y_{k,j} + b_{i,j} \cdot y_{k-1,j} + c_{i,j} \cdot y'_{k,k}. \tag{5}$$

The architecture of the 3-order predictor is shown in Fig. 11. The three values of the pixels and their corresponding coefficients enter these multipliers simultaneously. Input data is skewed and all internal operations are performed in a bit-skewed word-parallel fashion. The precision of the numbers, $y_{k,j}$, $y_{k-1,j}$, $y'_{k,k}$, is b-bits. The coefficients $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ have a precision of

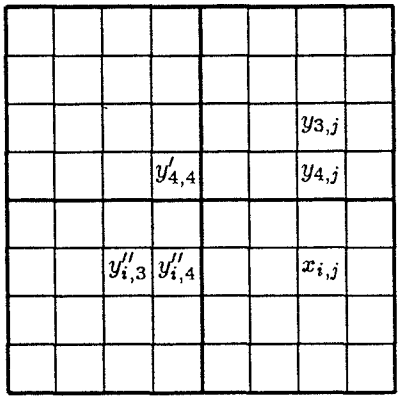


Figure 10: Consider the subblock of size 4×4. Pixel $x_{i,j}$ is predicted as a linear function of nearby pixels in adjacent vectors by $\tilde{x}_{i,j} = a_{i,j} \cdot y_{4,j} + b_{i,j} \cdot y_{3,j} + c_{i,j} \cdot y'_{4,4}$.

$(b+1)$ bits. All the multipliers and accumulators used in this architecture are pipelined at the bit-level. It takes b clock cycles for the multipliers to generate the first LSB of the product. Since multiplier outputs are already in skewed format, they can be directly fed into bit-level pipelined adders without additional skewing registers. The products $a_{i,j} \cdot y_{k,j}$ and $b_{i,j} \cdot y_{k-1,j}$ are added by the first adder, then the partial sum and $c_{i,j} \cdot y'_{k,k}$ are added by the second adder. To maintain full precision in all internal computations, the second adder takes a $(2b+1)$-bit product and a $(2b+2)$-bit partial sum from the first adder to form a $(2b+3)$-bit sum. At the last stage, skewing registers are used to deskew the predicted value. The total latency time for the linear predictor module is $3b+3$ clock cycles.

The linear predictor needs a memory of $3L$ words to store the coefficients $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ in equation (5).

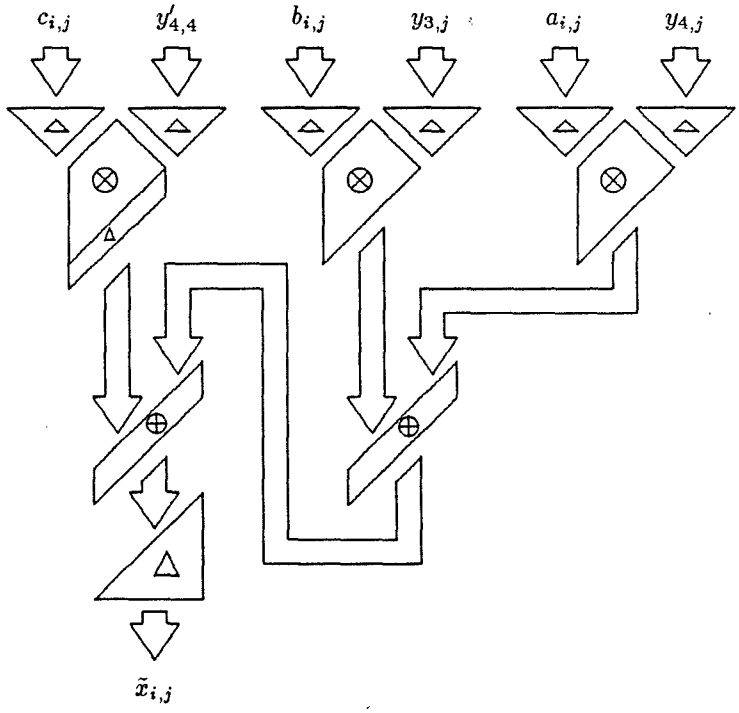


Figure 11: Systolic Architecture for third-order linear predictor. It performs the summation of three inner products.

Data Buffer and Control Units

From equation (5), we see that the inputs to the predictor module, $y_{k,j}$, $y_{k-1,j}$, $y'_{k,k}$, are used repeatedly; both $y_{k,j}$ and $y_{k-1,j}$ are used to compute the different prediction values in the same column and $y'_{k,k}$ is used for every pixel in the subblock. A simple circuit with cyclic shift registers, shown in Fig. 12, is used to handle this task. The last two rows of the image subblock in the input sequence are latched into buffers. Similarly, the last pixel of the previous subblock is latched into a single-word buffer. Control signals are used to ensure that these pixels are applied to the predictor module in the correct sequence.

The control unit consists of a $\lceil 2 \log k \rceil$-bit counter and simple combinational circuitry. The counter keeps track of each input vector to indicate the current pixel position within each incoming vector. Two input control signals are necessary; MODE indicates if the processor is being used in the encoder or the decoder, and Synch indicates if the current input is a boundary subblock.

Three control signals, ctrl1, ctrl2 and ctrl3, are internally generated to switch the multiplexers to update the content of the cyclic buffer. The counter enables us to fetch the appropriate coefficients from external memory into the linear predictor module.

The boundary subblocks in the top row and the left column of the input image frame are treated differently. In the Predictor Processor, there are two switches. Once a boundary subblock is indicated by the Synch input, the predicted value entering the adder/subtractor is set to zero. The correspond-

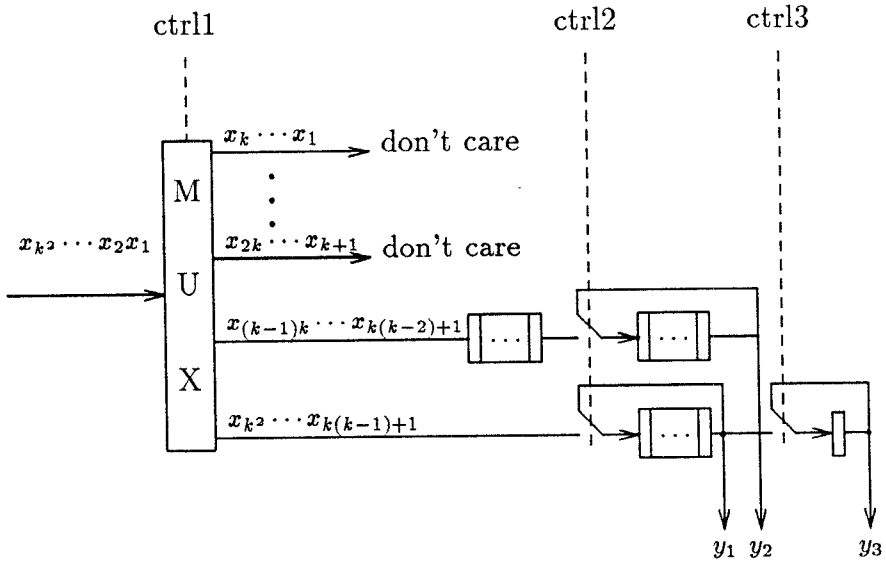


Figure 12: Circuit diagram of data buffer unit. It consists of cyclic shift register and multiplexers.

ing vector will pass through the adder/subtractor without modification. For better performance, these boundary subblocks can be coded using a different codebook from the one used for the residual vectors. However, this results in an increase in the size of the memory required. In our architecture, the boundary conditions of the image and the initialization of the system can be easily handled.

Timing and delay elements

Due to the presence of the feedback loop in the system, we have to insert delay elements to synchronize all intermediate computations. For simplicity, we introduce a notation and then discuss the details about the delay elements.

Let L_s, L_a, L_γ and L_β be the latency times through a subtractor, an adder, a VQ encoder, and a VQ decoder respectively. Let L_{LP} be the latency time of the linear predictor module and the data buffer unit, and let L_T to be the minimum time needed to compute the corresponding predicted value for the subblock in the next row after a block of pixels are fed in. The term L_T includes the processing time along the data path including the subtractor, TSVQ, ITSVQ, the adder, data buffer unit and the linear predictor module, i.e.

$$L_T = L_s + L_\gamma + L_\beta + L_a + L_{LP}.$$

Let L_S be the input separation time between two adjacent subblocks in the same column. The space-time diagram shown in Fig. 13 depicts the physical meaning of the above terms and the relative timing between them.

We notice, from Fig. 13, that as long as the latency time L_T is no larger

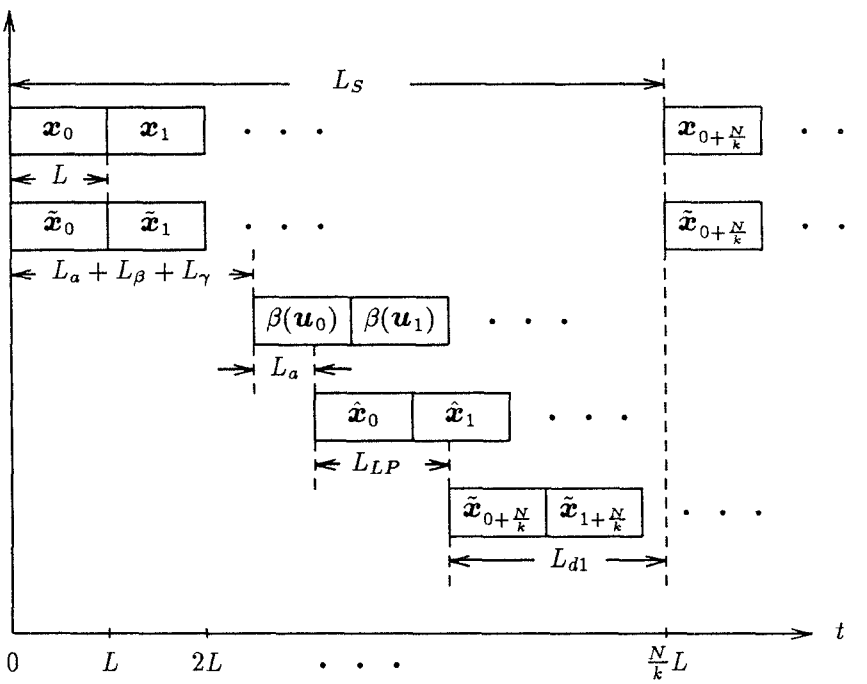


Figure 13: Space-time diagram of the PTSVQ system. The relative latency is shown at some interesting points.

than the separation time L_S, this architecture will achieve real-time performance. In order to synchronize the predicted vector and its corresponding input vector, the delay elements in the feedback loop must satisfy the following timing constraints:

$$L_s + L_\gamma + L_\beta + L_a + L_{LP} + L_{d2} = L_S,$$

and

$$L_a + L_\gamma + L_\beta = L_{d1},$$

where L_{d1} and L_{d2} are the delay times associated with the delay elements in the path.

Consider, for example, the case of an image of size 512×512 pixels where each subblock is of size 4×4 pixels. Then L_S is $512 \times 4 = 2048$ cycles. In a TSVQ of vector dimension 16 and depth 8, L_β and L_γ are 1 and 384 cycles respectively; L_{LP} is 42 cycles and includes three pipelined multiplications, two additions and a few data skew elements; L_a is equal to 9 cycles. Hence, L_{d1} and L_{d2} are 394 and 1601 respectively.

4 VLSI Implementation and Testing

In this section, we describe the VLSI implementation of the chips necessary to build a PTSVQ system using $1.2\mu m$ CMOS technology. We designed these chips using Magic, Irsim, Spice and GDT tools. Spice simulations indicate that these chips can run at frequencies up to 40 MHz. We fabricated prototype version of these chips using $2\mu m$ CMOS N-Well process, and tested these prototype chips at 20 MHz.

4.1 SNP chips

We designed a Single Node Processor using $1.2\mu m$ CMOS N-Well technology. The processor contains 25,000 transistors on a $4.8mm \times 5.5mm$ die and has 84 pins. We performed logic and timing simulations at 40 MHz on this chip. A prototype version of this chip fabricated using $2\mu m$ CMOS process works at a frequency of 20 MHz [15].

4.2 Predictor Processor

We partitioned the predictor processor into different submodules for ease of implementation.

1. *Front end processor.* It consists of a pipelined subtractor with skewing and deskewing elements. This module contains 1,650 transistors on a $1.3mm \times 1.3mm$ die using $1.2\mu m$ technology. Figure 14 shows a plot of this chip. A prototype version of this chip using $2\mu m$ technology worked at 20 MHz.

2. *Controller and Buffer.* It includes a pipelined adder with skewing and deskewing elements, the control circuit and the buffer unit. This module contains 4,000 transistors on a $2.8mm \times 4.1mm$ chip using $1.2\mu m$ technology. Figure 15 shows a plot of this chip. A prototype version of this chip fabricated using $2\mu m$ process worked at 20 MHz.

3. *Predictor.* It consists of the Linear Predictor Module (LPM). This module contains 36,000 transistors on a $4.7mm \times 5.5mm$ die. Figure 16

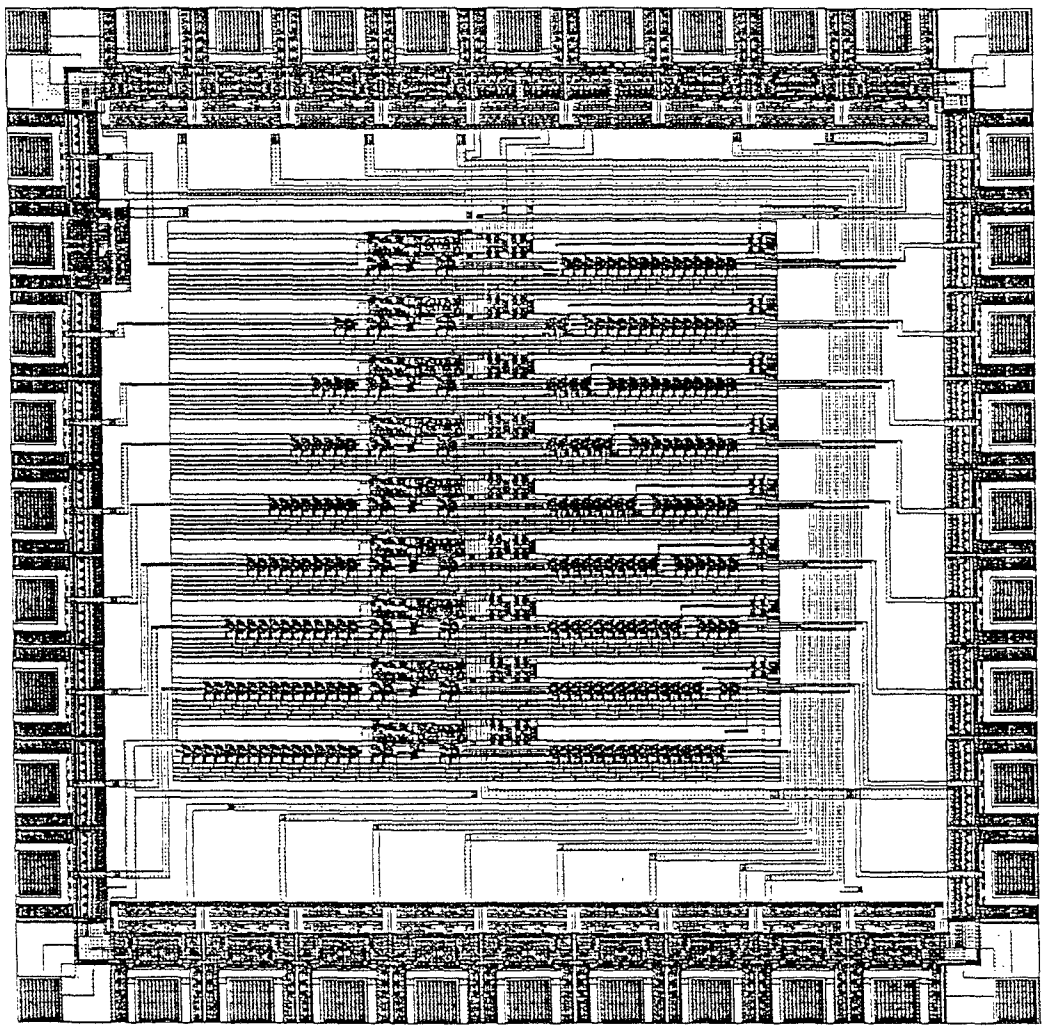


Figure 14: Plot of the front end processor of size $2.2mm \times 2.2mm$.

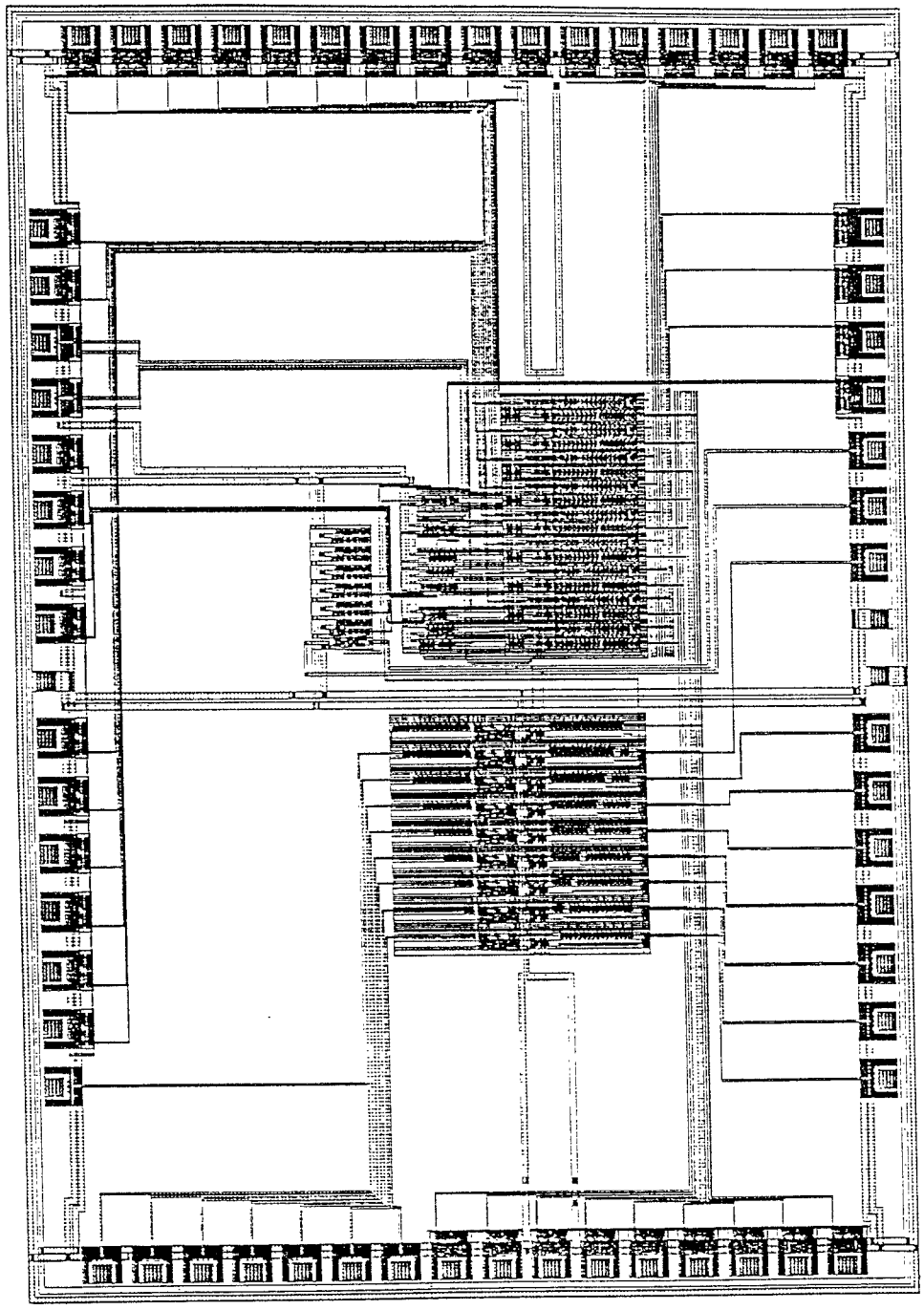


Figure 15: Plot of the controller of size $2.8mm \times 4.1mm$.

shows a plot of this chip. A prototype version of this chip fabricated using $2\mu m$ process worked at 20 MHz.

5 Discussion

A new Predictive TSVQ architecture is presented for real-time video coding applications. Pipelined arithmetic components are used to speed up the computation and to provide for regularity in design. This high throughput architecture is suitable for implementing a fully pipelined real-time PTSVQ system. This architecture has been implemented as a VLSI chip set using $1.2\mu m$ CMOS technology. Identical processors are used for both the encoding and decoding components. Spice simulations indicate correct operation at 40 MHz. For a typical real-time image processing system with 30 frames/sec and 1024×1024 pixels/frame, the input pixel rate is 31.5 Mpixels/sec. This architecture is capable of processing 40Mpixels/sec and can handle the above case in real-time. We fabricated prototype versions of these chips using $2\mu m$ CMOS technology. These prototype chips work at 20 MHz. Our architecture can be extended easily to handle other classes of VQ with memory such as Trellis VQ.

Acknowledgments

We had many beneficial discussions with Yunus Hussain, Nam Phamdo and Nariman Farvardin. We are grateful for their contribution to this work.

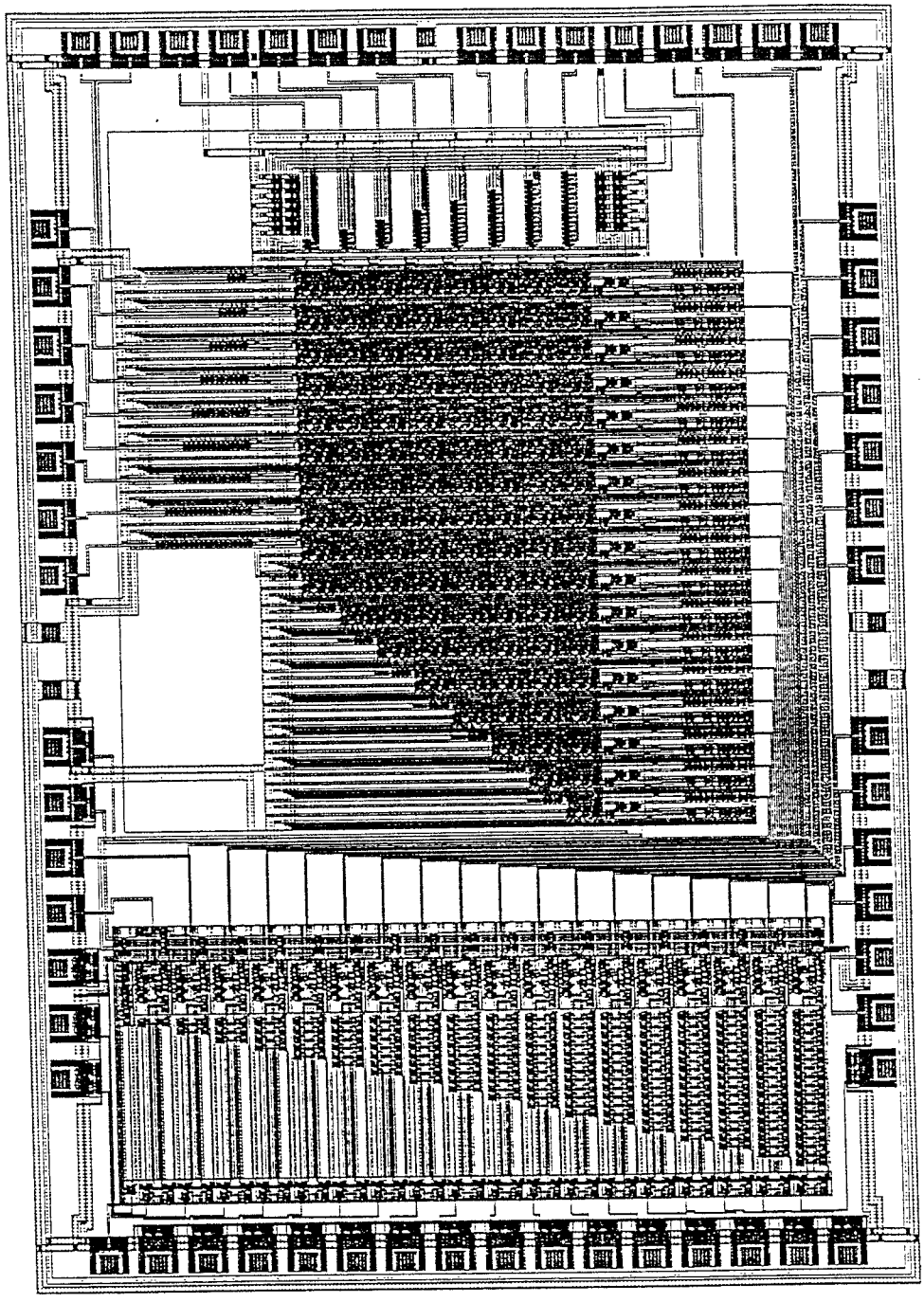


Figure 16: Plot of the predictor of size $7.9mm \times 9.2mm$

References

[1] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression.* Kluwer Academic Press, 1992.

[2] R. M. Gray, "Vector quantization," *IEEE ASSP Mag.*, vol. 1, pp. 4–29, 1984.

[3] N. Nasrabadi and R. King, "Image coding using vector quantization: A review," *IEEE Trans. Commun.*, vol. COM-36, pp. 957–971, Aug. 1988.

[4] P. Chang and R. M. Gray, "Gradient algorithms for design predictive vector quantization," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 679–690, Aug. 1986.

[5] A. Haoui and D. Messerschmitt, "Predictive vector quantizer," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, 1984.

[6] V. Cuperman and A. Gersho, "Vector predictive coding of speech at 16 kbits/s," *IEEE Trans. Commun.*, vol. COM-33, pp. 685–696, July 1985.

[7] H. M. Hang and J. W. Woods, "Predictive vector quantization of images," *IEEE Trans. Commun.*, vol. COM-33, no. 11, pp. 1208–1219, 1985.

[8] E. A. Riskin, *Variable Rate Vector Quantization of Images.* PhD thesis, Stanford Univerity, May 1990.

[9] J. Foster, R. M. Gray, and M. O. Dunham, "Finite-state vector quantization for waveform coding," *IEEE Trans. Infor. Theory*, vol. IT-31, pp. 348–359, May 1985.

[10] M. O. Dunham and R. M. Gray, "An algorithm for the design of labeled-transition finite-state vector quantizers," *IEEE Trans. Commun.*, vol. COM-33, pp. 83–89, Jan. 1985.

[11] R. K. Kolagotla, S.-S. Yu, and J. F. JáJá, "Systolic architectures for finite-state vector quantization," Tech. Rep. SRC TR 91–102, University of Maryland, Dec. 1991.

[12] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel, "Speech coding based upon vector quantization," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-28, pp. 562–574, Oct. 1980.

[13] T. Lookabaugh, "Architectures for tree structured vector quantization." Unpublished work, May 1987.

[14] W. C. Fang, C. Y. Chang, and B. J. Sheu, "Systolic tree-structured vector quantizer for real-time image compression." Private communication, Oct. 1990.

[15] R. K. Kolagotla, S.-S. Yu, and J. F. JáJá, "VLSI implementation of a tree searched vector quantizer," *IEEE Trans. Signal Processing*, Apr. 1993.

[16] T. Markas, J. Reif, W. Elliot, and E. Elliot, "Memory-shared parallel architectures for vector quantization algorithms." Private communication, Nov. 1991.

[17] M. Yan and J. McCanny, "A bit-level systolic architecture for implementing a VQ tree search," *Journal of VLSI Signal Processing*, vol. 2, pp. 149–158, Nov. 1990.

[18] A. Madisetti, R. Jain, R. L. Baker, and R. Dianysian, "Architectures and integrated circuits for real time vector quantization of images," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. V–677–680, 1992.

[19] V. Bhaskaran, "Predictive VQ schemes for grayscale image compression," in *Proc. GLOBECOM '87*, pp. 11.8.1–11.8.6, Dec. 1987.

[20] J. W. Modestino and Y. H. Kim, "Adaptive entropy-coded predictive vector quantization of images," *IEEE Trans. Signal Processing*, vol. 40, pp. 633–644, Mar. 1992.

[21] D. Y. Cheng and A. Gersho, "A fast codebook search algorithm for nearest-neighbor pattern matching," in *Proc. IEEE Int'l. Conf. on Acoustics, Speech and Signal Processing*, pp. 265–268, 1986.

[22] G. K. Ma and F. J. Taylor, "Multiplier policies for digital signal processing," *IEEE ASSP Magazine*, vol. 7, pp. 6–20, Jan. 1990.

[23] J. V. McCanny and J. G. McWhirter, "Completely iterative, pipelined multiplier array suitable for VLSI," *IEE Proc.*, vol. 129, pp. 40–46, Apr. 1982.

[24] H. Park, V. K. Prasanna, and C.-L. Wang, "An architecture for tree search based vector quantization for single chip implementation," Tech. Rep. 289, IRIS, Univ. of Southern California, 1992.