A Vector Quantizer with Fully Pipelined Data and Control Flow

Ding-Ming Kwai and Behrooz Parhami

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

Abstract – Previously proposed linear array structures for vector quantization tend to treat memory access as an independent issue, implying that load and store operations are performed through separate data and control pins. For most applications, it is essential to adaptively update codevectors during normal operation. We present a design where global wiring is restricted to power and clock supplies, thus making the architecture truly scalable. Since data and control signals are not broadcast but propagated through local connections, they have to be accessed by the host via processing elements at the array boundary. The contribution of this paper is to ensure these desirable properties while maintaining its pipelined and parallel operation. An important objective is to demonstrate that our proposed linear array architecture using a data-driven control scheme exhibits flexibility and expandability which make it very attractive for VLSI implementation.

I. Introduction

Vector quantization (VQ) has been proposed as a data compression technique for certain speech and image processing applications [4]. A VQ-based system parses data into non-overlapped groups or vectors. According to Shannon's source coding theory, better performance can be achieved by coding vectors rather than scalars. The fundamental computation associated with VQ is a pattern matching process in which each input vector (pattern) is compared with a set of representative codevectors (templates) to identify the closest match. These codevectors are contained in a codebook such that only the indices of the matched codevectors are stored or transmitted. The indices are used to recover the codevectors.

Studies of computational complexity in VQ-based systems focus on the encoding process, since the reverse decoding process is a simple table lookup. This is underscored by the fact that an optimal VQ encoder generally requires to search through the entire codebook. Adopting tree search, with the codebook organized in multiple levels, significantly reduces the computational complexity, at the expense of increased storage requirement and degraded quality in terms of signal-to-noise ratio. Since an L-level tree-search VQ encoder can be realized by means of a cascaded series of L full-search VQ encoders [3], we will concentrate on the design of full-search VQ encoders.

In view of low interconnection density for VLSI layout, bounded I/O requirements, and ease of global clocking in the presence of propagation delays, most designs for VQ encoders are based on linear array architectures [2], [5], [6], [7]. However, previously proposed designs tend to treat memory access as an independent issue, implying that load and store operations are performed through separate I/O pins or, alternatively, assuming the use of a permanent memory in which the codebook has been installed during fabrication. For most applications, the ability to support in-operation update of the codebook is essential. With existing designs, this capability is in conflict with the requirement that I/O has to be performed via processing elements (PEs) at the array boundary. The design method considered here allows us to include these desirable functions into the array while maintaining its pipelined and parallel operation.

The rest of this paper is organized as follows. Section II briefly describes the VQ full-search algorithm. Section III introduces a unidirectional linear array for VQ encoding. In Section IV, we propose a data-driven control scheme to solve the problems of data loading, array initialization, and index labeling. Section V examines various design choices for the PE structure and analyzes these designs with regard to cost and performance. Section VI contains our conclusions.

II. VQ Full Search Algorithm

Let $\boldsymbol{x}_k$ be an M-dimensional vector, where the subscript k indicates its position in a stream of source vectors entering a VQ encoder. The VQ encoder searches through the codebook $\{\boldsymbol{w}_i : 0 \le i \le N-1\}$ and derives an index I_k pointing to the best matched codevector for input vector $\boldsymbol{x}_k$. Associated with the pattern matching process is a distortion measure, which serves as the basis for quantifying the dissimilarity between each codevector $\boldsymbol{w}_i$ and input vector $\boldsymbol{x}_k$.

Let us denote $\boldsymbol{x}_k = (x_{k0}\ x_{k1}\ \dots\ x_{k,M-1})$ and $\boldsymbol{w}_i = (w_{i0}\ w_{i1}\ \dots\ w_{i,M-1})$. The squared error distortion measure can be formulated as follows.

$$\|\boldsymbol{x}_k - \boldsymbol{w}_i\|^2 = \sum_{j=0}^{M-1} \left(w_{ij}^2 - 2w_{ij}x_{kj} + x_{kj}^2 \right)$$

For the purpose of codebook search, the term $\Sigma_{j=0}^{M-1} x_{kj}^2$ in the above equation is common to all codevectors, and hence, will not affect the comparison result. This simplifies the distortion measure to an inner-product form of M multiplication-accumulation steps.

$$d(x_k, w_i) = \sum_{j=0}^{M-1} w_{ij}(w_{ij} - 2x_{kj})$$

In addition, the VQ encoder needs to find the minimal value among all N distortion measures. The index $I_k = i$ is selected if $d_k = d(x_k, w_i)$ yields a minimal value.

Fig. 1 depicts the dependence graph of the full-search VQ algorithm for $N = 4$ and $M = 3$. We have added the rightmost column of dark nodes to show the comparison steps after the inner-product computation is completed at each row. Note that in the dependence graph, it is required to insert a zero at the first node of each row as a new inner-product computation begins. The dark nodes at the left are thus different from the clear nodes that receive intermediate results passed from the previous nodes. The computations associated with the rightmost column of dark nodes can be overlapped with the next computation, so the input data can be continuously fed to the PEs without interspersed delay.

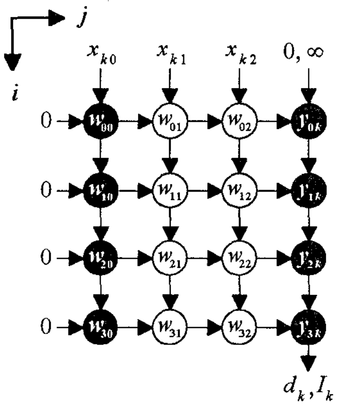

Fig. 1. Dependence graph of full-search VQ.

III. Array Architecture for Full-Search VQ

Our design is modular; identical PEs are cascaded into a linear array. It is derived by projecting the dependence graph of Fig. 1 along the j direction. The projection direction is coincident with the vector dimension in such a way that the ith row is mapped onto the ith PE. The linear array has two types of channels: c_{kj} is for control signals in which a two-bit pattern c_1c_2 passes through unchanged; x_{kj}, d_{kj}, and I_{kj} are for data signals of the input vector, the intermediate result of the minimal distortion measure, and the index of the corresponding codevector, respectively.

The operation of the linear array is similar to that of a distributed pipeline [1]. The input data stream travels from left to right, one PE per clock cycle. Each PE performs M inner-product steps with its stored codevector for each input vector received element by element. In the mean time, the input vector is passed to the right. The distortion measure is obtained by the summation of the products and compared with the minimal value (up to the previous PE) received from the left. The comparison result then directs the same or a smaller distortion measure and its associated index to the right. The next PE repeats this process.

Fig. 2 shows the detailed structure of each PE. It is essentially composed of data latches (rectangular boxes), two-way multiplexers (circles with a cross), a comparator (CMP), and a multiplier-accumulator module (MA) that is sign-extended by an appropriate number of bits to account for carry generation. For the ith PE, the codevector w_i is stored in a circular queue: w_i is recycled through the queue so that its element w_{ij} emerges as the element x_{kj} of the input vector x_k arrives for computing the product $w_{ij}(w_{ij} - 2x_{kj})$. The result is fed back to the accumulator to add on to another product.

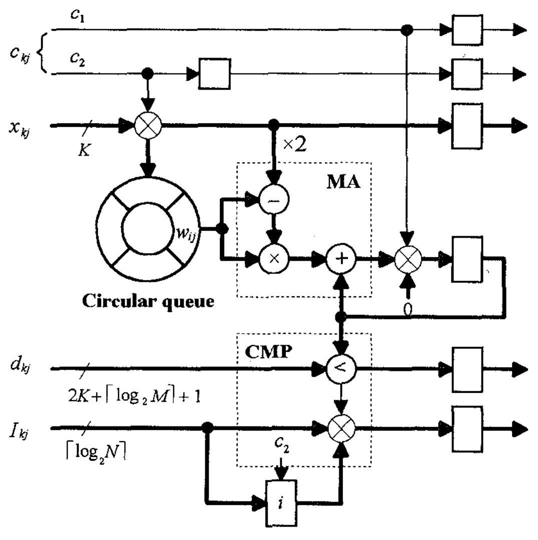

Fig. 2. PE structure.

Due to the rhythmic operation of VQ encoding, the circular queue can be implemented as a random access memory with a counter linked to the global clock determining the address. The address counter is increased up to the length of the circular queue and then reset to zero. Using the circular queue allows the loading of new codevectors to be pipelined with normal operation and permits the vector dimension to vary. In order to flexibly determine the vector dimension M, we assume that the circular queue has a variable length. In practice, the value of M is selected in the range from 16 to 36 [2]. A 6-bit counter is sufficient for such applications.

IV. Data-Driven Control Scheme

Note that if the dark node at left end of each row in Fig. 1 also resets the address counter of the circular queue to zero, the clear nodes can increment it up to $M - 1$. Hence, we can let the input data stream carry this information to distinguish the various functions that each PE should perform.

Since data and control signals are not broadcast but propagated through the linear array, a data-driven control scheme is proposed to allow the loading and storing of data items by sending a two-bit pattern c_1c_2. This pattern is coincident with each element x_{kj}, used to specify PE's functions as the input vector traverses the linear array. Our design has the advantage that it is readily expandable by including additional PEs for a larger codebook size. To achieve this, the index labels on PEs must also be dynamically changeable.

The I/O pins for passing data element x_{kj} and index I_{kj} are reused for loading of new codevectors and labeling on PEs, respectively. If we insert $d_{kj} = -\infty$, the comparison result will direct the incoming index I_{kj} to go through without change. Thus, the latches for passing I_{kj} in the linear array form a shift register of N stages. This is extremely useful when we let the input $x_{kj} = w_{ij}$ and $I_{kj} = i$, where i is the location where they are destined to stay, so as to store w_{ij} and label i on the ith PE. The above process can be explained by the dependence graph shown in Fig. 3. The added lightly shaded nodes transfer the data flow to the projection direction, implying that the data stay at the PE.

In Fig. 3, triples $(x_{kj}, d_{kj}, I_{kj}) = (w_{ij}, -\infty, i)$, for $0 \le i \le N-1$ and $0 \le j \le M-1$, show the data sequentially inserted in ascending order of indices during the initialization phase. In general, the index labels on PEs are not necessarily consecutive on the physical array. Thus, if spare PEs and bypass connections are available, the replacement of faulty PEs can be done without modifying the procedure. The linear array can be easily extended to incorporate fault tolerance.

The two-bit pattern c_1c_2 shown beside each node in Fig. 3 marks three distinct types of nodes. Each type is related to a different function performed by the PE. One may observe that the bit stream of c_1 flows in the vertical direction and the bit stream of c_2 flows in the diagonal direction. The latter corresponds to taking two clock cycles to pass through a PE, and hence, two delays are placed on the path of c_2. It can be verified that the two bit-streams correctly instruct the linear array to load/store the codevectors first and then to compute/compare the distortion measures.

In the initialization phase, c_2 takes on the value 1 followed by $N-1$ 0s, with the pattern repeated M times for M dimensions, while c_1 remains 1. During normal operation, c_1 takes on the value 1 followed by $M-1$ 0s, while c_2 remains 0. The c_1 bit stream also indicates the time to retrieve the minimal distortion measure and the index of the corresponding codevector, with a 0-to-1 transition of c_1 at the output. The resulting functional table, relating the PE's operations to the control tags c_1 and c_2 is shown in Fig. 4.

		$\overline{\text{Compute}}$/Reset	
	$c_2 \backslash c_1$	0	1
$\overline{\text{Store}}$/Load	0	○	●
	1		◍

Fig. 4. Functional table.

V. Serial/Parallel Processing Tradeoffs

The proposed array architecture for VQ encoding can be implemented in many different ways. We analyze the cost and performance of such arrays with four implementation alternatives. These alternatives are based on concurrency or lack thereof in handling words (word-parallel *vs.* word-serial) or bits within words (bit-parallel *vs.* bit-serial).

The circuit size per PE of each design is estimated based on a gate-level analysis of the components. The contribution of the codebook memory is not considered, since its size is MNK bits for the four designs. Ripple-carry adder and array multiplier are assumed for the multiplier-accumulator. For bit-serial designs, the multiplication is done by a sequence of shift-and-add operations, assuming that the least significant bit is received first.

With the initial word length K, the product $w_{ij}(w_{ij} - 2x_{kj})$ requires $2K + 1$ bits and the sum of M such products needs

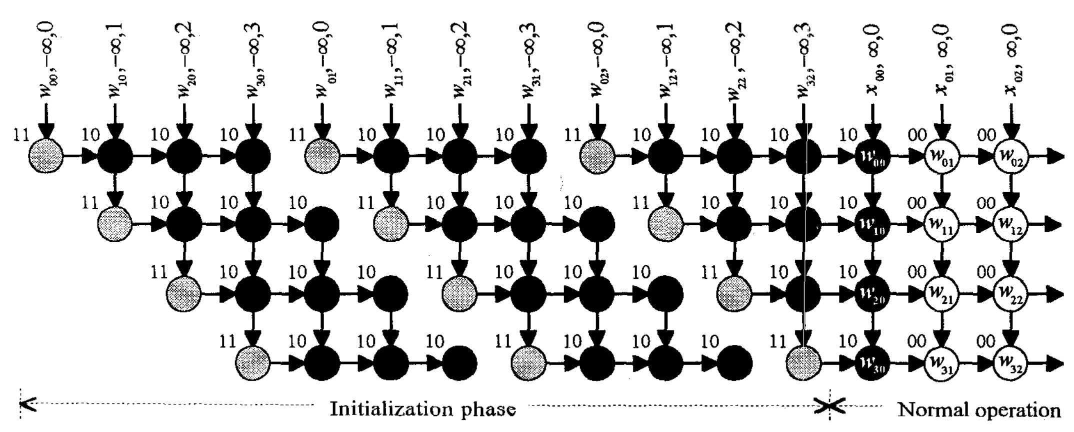

Fig. 3. Augmented dependence graph showing codevector load/store operations.

another $\lceil \log_2 M \rceil$ bits. The total number of bits for computing and passing the minimal distortion value is thus $2K + \lceil \log_2 M \rceil + 1$. Here, we assume that this additional precision is required. For bit-serial designs, the input data words must include sign extension, which reduces the throughput rate by a factor of $1/(2K + \lceil \log_2 M \rceil + 1)$.

The word-parallel bit-parallel design [2] can be seen as a direct implementation of Fig. 1. A large number of I/O pins are needed in order to support the required data bandwidth. For example, with $N = 256$, $M = 16$, and $K = 8$, the number of I/O pins is $2(MK + 2K + \log_2 M + \log_2 N + 1) = 314$. Fewer I/O pins are needed in the word-parallel bit-serial design [7]. However, in order to align incoming and outgoing data items, a large number of data latches must be added.

Unlike the above designs in which the codebook is distributed throughout the array, the word-serial bit-serial design [5] uses external centralized memory to store the codebook. The I/O bandwidth is constant, provided that the precision of the distortion value is less than M bits. Its control scheme incorporates four-bit patterns to load/store the codebook and initialize the array, but is more complicated than our design and does not allow the vector dimension to change.

We assume that the resolution $r = (\log_2 N)/M$ is kept at 0.5 bit per sample and that the input data is $K = 8$ bits wide. The four implementation alternatives are different with regard to cost-effectiveness due to the area and time trade-offs. Fig. 5 shows area-time measure of the four designs with M varying from 16 to 36. The area-time measure is obtained by dividing the total circuit size by the throughput rate.

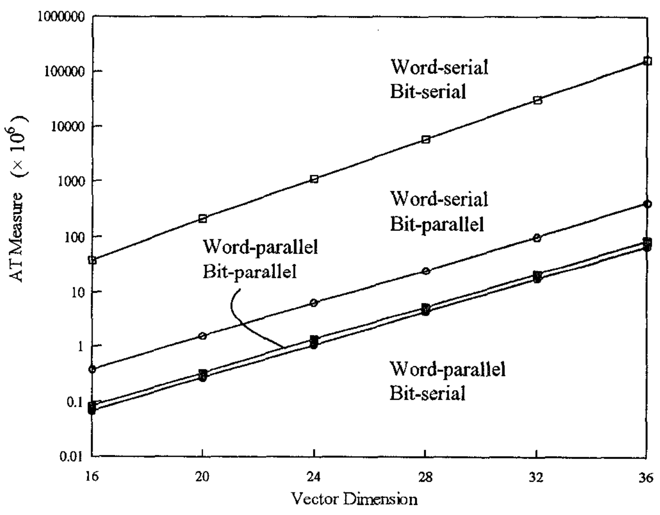

Fig. 5. The area-time measure of various designs for VQ encoding.

The estimated circuit size of the four designs is shown in Fig. 6, together with the total memory size for comparison. Since the memory size is at least one order of magnitude smaller than total circuit size, it is advantageous to distribute the codebook throughout the array, thus eliminating the need for a large external memory and complex addressing logic.

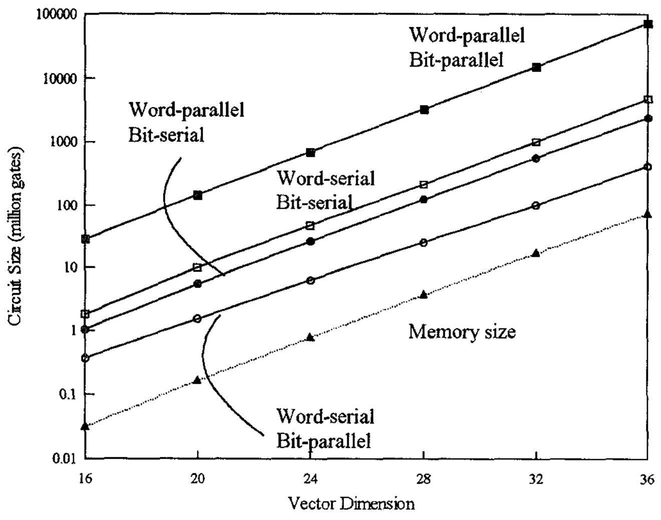

Fig. 7. The total circuit size of various designs for VQ encoding.

VI. CONCLUSIONS

In this paper, we have presented a linear array structure for vector quantization suitable for VLSI implementation. In order to make the design truly scalable, control signals are not broadcast but propagated through local and low-fanout connections. Our design encompasses a data-driven control scheme to maintain the pipelined and parallel operation while providing the capability for handling the run-time loading of codevectors, array initialization, and adaptive update of codebook and vector dimension.

The main advantages of our design are the small circuit size and I/O bandwidth requirement, operational capability in the presence of faults, and flexibility to support changing the parameters of the full-search VQ method. The data-driven control scheme can also be combined naturally with error detection and fault diagnosis to achieve more reliable computation.

REFERENCES

[1] R. G. Cooper, "The Distributed Pipeline," *IEEE Trans. Computers*, vol. C-26, pp. 1123-1132, Nov. 1977.

[2] G. A. Davidson, P. R. Cappello, and A. Gersho, "Systolic Architectures for Vector Quantization," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. ASSP-36, pp. 1652-1664, Oct. 1988.

[3] W.-C. Fang *et al.*, "VLSI Systolic Binary Tree-Searched Vector Quantizer for Image Compression," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 2, pp. 33-44, Mar. 1994.

[4] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, New York, NY: Kluwer Academic, 1991.

[5] H. Park and V. K. Prasanna, "Modular VLSI Architectures for Real-Time Full-Search-Based Vector Quantization," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 3, pp. 309-317, Aug. 1993.

[6] P.A. Ramamoorthy, B. Potu, and T. Tran, "Bit-Serial VLSI Implementation of Vector Quantizer for Real-Time Image Coding," *IEEE Trans. Circuits and Systems*, vol. 36, pp. 1281-1290, Oct. 1989.

[7] M. Yan, J. V. McCanny, and U. Hu, "VLSI Architectures for Vector Quantization," *J. VLSI Signal Processing*, vol. 10, pp. 5-23, 1995.