A HASHING-BASED SCHEME FOR ORGANIZING VECTOR QUANTIZATION CODEBOOK

Chang Y. Choo, Erik Kristenson
Dept. of Electrical Engr.
San Jose State Univ.
San Jose, CA 95192-0084
cychoo@sparta.sjsu.edu

Nasser M. Nasrabadi
Dept. of Electr. & Comptr. Engr.
State Univ. of New York
Buffalo, NY 14260

Xiaonong Ran
System Technology Group
National Semiconductor Corp.
Santa Clara, CA 95052

ABSTRACT

One of the problems in vector quantization (VQ) is its relatively long encoding time especially when an exhaustive search is made for the codevector. This paper presents a hashing-based technique to organize the codebook so that the search time can be significantly reduced. Hashing gives the speed advantages of a direct search, while maintaining a codebook of reasonable size. Experiments show that hashing-based VQ sustained image quality as the encoding time was reduced, while full search VQ suffered greatly. For example, for 2×2 *vectors and with 1024 codebook entries, encoding time was reduced by a factor of 10 without significant loss of image quality.*

1. INTRODUCTION

One of the problems in vector quantization (VQ) is its relatively long encoding time especially when an exhaustive search is made to find the best codevector for each image vector. A number of techniques have been proposed to reduce the encoding time which include tree-structured VQ, finite-state VQ, and cache VQ [2,3,4,5,6].

Hashing is a database search technique, where the records are ordered according to a hashing function [1]. The hashing function is a formula, whose input is a portion of the record, called a key, and whose output is an index to the database. Hashing gives the speed advantages of a direct search, while maintaining a database of reasonable size.

An example of a hashing based database could be a set of customer records for a cash machine. The key is a 19 digit number of the ATM card. To find the customer's record, an exhaustive search can be made through the database until the key on the card matches that of the record, resulting in an unreasonably long search time. An alternative would be to put the 19-digit number through a hashing function, which would create an index with a smaller range. The hashing function could, for example, be something as simple as a mask which preserves the 5 least significant digits of the key. Since the range is smaller than the original key, more than one key may hash to the same index. When this happens, a collision is said to have occurred. Because of collisions, the index points to a "bucket" where more than one record may reside. Accessing a customer's record now localizes the search to the contents of a bucket.

Hashing is used extensively in database applications to speed up search operations, but has yet to be formally introduced to VQ encoding, which is often criticized for its long encoding time. In this paper, we present a hashing-based technique to organize the codebook so that the search time can be significantly reduced.

This paper is organized as follows. In the next section, the hashing based codebook organization is described in detail. In the following section, encoding procedure of the hashing based VQ is described. Some experimental results are presented in the following section, along with discussions. Concluding remarks are made in the last section.

2. HASHING BASED CODEBOOK ORGANIZATION

Hashing function maps input image vectors to the integers equivalent to the codebook indices. When the hashing technique is applied to VQ encoding, a portion of the vector is used as the input to the hashing function, and the output is a codebook index. Due to the likelihood of collisions, this index will point to a bucket in the codebook, which consists of a group of codevectors. One example of hashing function for VQ is to take the most significant bits of some or all pixels

0-7803-2431-5/95 $4.00 © 1995 IEEE

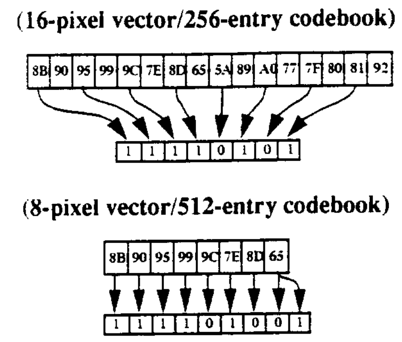
(16-pixel vector/256-entry codebook)
8B 90 95 99 9C 7E 8D 65 5A 89 A0 77 7F 80 81 92
1 1 1 1 0 1 0 1
(8-pixel vector/512-entry codebook)
8B 90 95 99 9C 7E 8D 65
1 1 1 1 0 1 0 0 1

Figure 1: Illustration of MSB (Most Significant Bit) hashing function.

in the input vector to create a codebook index as shown in Figure 1. We will call it the MSB hashing function.

In order to use hashing during encoding, the codevectors that have been generated after training must first be reordered in the codebook according to the hashing function. In addition, provisions must be made for vectors that collide into the same bucket, both when organizing the codebook and when searching for an entry during the encoding process. The vectors in this bucket may be rearranged into a linked list, where only the head vector will reside at the hashed index. Other entries will be stored at unused indices, and the vectors corresponding to each individual bucket will be grouped together via pointers. To maintain a fixed codebook size, the number of vectors in a bucket multiplied by the number of buckets must be, by definition, a constant.

In order to remap a standard codebook into one that is configured for the hashing algorithm, additional information is needed besides the codebook vectors. This additional information is attached to every vector by means of a structure with the format shown in Figure 2. Note that only the codevectors are needed during the decoding sequence. This helps reduce the amount of side information that is sent along with the compressed image. During encoding, only the white and light grey areas of the structure shown in the figure are needed (as indicated by the legend), while the entire structure is needed when the codebook is being reorganized from an original codebook, to a hashing based codebook.

When a codebook generation is complete, the codebook is then reorganized into an array of structure elements. This organization is needed, so that during the encoding process, an input image vector will be able to locate the corresponding codevector by using only the hashing formula. Described below are the elements of the structure:

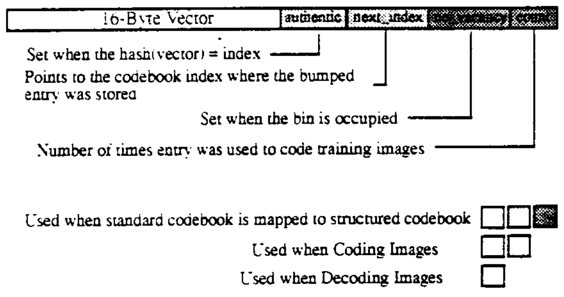
16-Byte Vector
authentic
next_index
no_vacancy
count
Set when the hash(vector) = index
Points to the codebook index where the bumped entry was stored
Set when the bin is occupied
Number of times entry was used to code training images
Used when standard codebook is mapped to structured codebook
Used when Coding Images
Used when Decoding Images

Figure 2: Elements of a hashed codebook structure.

count Since collisions may occur when more than one vector maps to a particular bucket, a linked list will be created. It is desirable to have the most popular codevector at the head of this list. The most popular codevector is the one most often used by the set of training images, and has the maximum count value.

no_vacancy Once a codevector is mapped to a particular bin in a particular bucket, the no_vacancy flag is set. This prevents another vector from overwriting the current one.

next_index A linked list is created when more than one codevector map to the same index. When this happens, the bumped vector must search high and low for a vacant bin. When one is found, the bumped vector will be stored there, and the previous vector's next_index field will be updated to point to the bumped vector.

authentic During the codebook remapping process, when a vector is mapped to a vacant bin, it is stored there, and the authentic field is set to 1. If another vector maps to the same bin, then a neighboring bin is found, and the vector is stored there. This vector is now known as the tail of a linked list, while the first vector is the head of the linked list. The two vectors are said to be in the same bucket.

Figure 3 shows the flow required to map a codevector from a standard codebook into a structured codebook. For each codebook entry, the hashing function is applied which results in an index to a particular bin. The codevector is mapped to this bin if it is empty, as indicated by a clear, no_vacancy bit. A collision has occurred if the no_vacancy bit is set, creating the need to find a vacant, neighboring bin to store the bumped vector in. At this point, it is not known whether the vector that resides in the bin is the head of a linked list, or it is an element of some other linked list. If

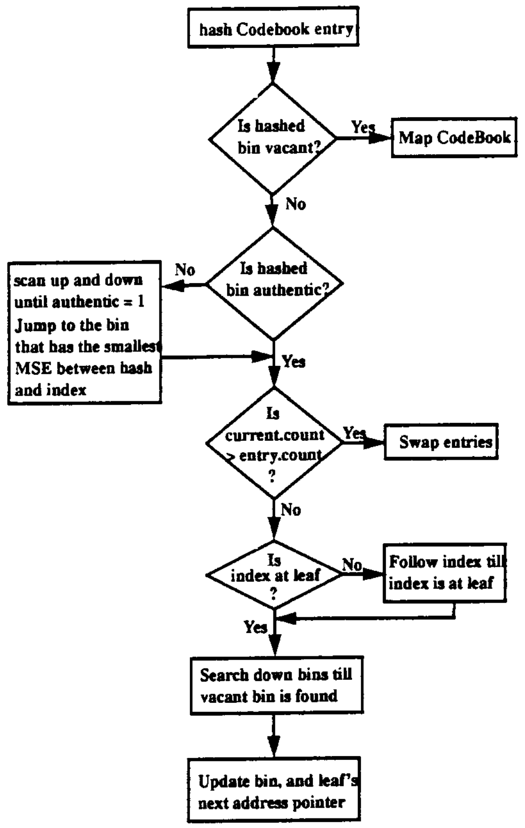
hash Codebook entry
Is hashed bin vacant?
Yes
Map CodeBook
No
Is hashed bin authentic?
No
scan up and down until authentic = 1 Jump to the bin that has the smallest MSE between hash and index
Yes
Is current.count > entry.count ?
Yes
Swap entries
No
Is index at leaf ?
No
Follow index till index is at leaf
Yes
Search down bins till vacant bin is found
Update bin, and leaf's next address pointer

Figure 3: Remapping standard codebook to hashing based codebook.

it is the head, as indicated by the authentic bit being set, then the current vector should attach itself to this linked list. All vectors that are in a linked list are considered to be in the same bucket. If the authentic bit is not set, then the vector residing at the index is an element of a different linked list, (i.e., belongs to a different bucket). Since it is conceivable that the head vector of the linked list in question could have radically different characteristics than the current vector, it is not advisable to add it to the bucket. Instead, a search is made to find the nearest head of a linked list, and to add the current vector to this bucket instead.

When adding an entry to a bucket, it is desirable to place the most popular vector at the head of the list. This is because during the encoding process, the search will begin with the head vector, and continue with other entries in the bucket only if the first vector is not similar enough to the image vector. The flowchart in Figure 3 shows that the vectors in a linked list will be ordered from most popular to least popular.

3. HASHING BASED VQ ENCODING

Figure 4 shows the flowchart of the encoding process. VQ encoding usually starts the search in the codebook at index 0. With a hashing algorithm, an image vector is hashed to produce an index where the search will begin. The search is complete when the mean square error between the codevector and the image vector is less than or equal to a specified threshold.

Before the search begins, the authentic bit is checked to confirm that the entry is the head of a linked list. If it is, then all entries in the bucket are searched until the MSE is less than or equal to the threshold. Otherwise, it is understood that we have hashed to the middle, or the end of a linked list. In order to find a codevector that is similar to the image vector, a search must be made both high and low for the head of a linked list. This is where the best codevector will reside.

If there are no vectors in the bucket that satisfy the threshold requirements, then the search is continued in the neighboring buckets. As part of the algorithm, the minimum and the maximum buckets are actually neighbors, so the best codevector is not far from the starting search point. If the threshold is set too high during the encoding process, then an exhaustive search will be made of the codebook, and the index whose vector had the lowest MSE will be coded. The results will be identical to the full-search VQ encoding process.

4. EXPERIMENTAL RESULTS

A C program was written to simulate the hashing-based VQ scheme. Various codebooks with different size and vector dimension were trained using the LBG algorithm [4]. Two representative results are presented in the following.

As shown in Figure 5, for 2×2 vectors and with 1024 codebook entries, encoding time was reduced by a factor of 10 without significant loss of image quality (less than 1 dB). For 4×4 vectors and with 256 codebook entries, the savings in encoding time dropped to a factor of 3 with less than 1 dB degradation in image quality (see Figure 6). Note that on both graphs the full-search VQ is represented as the endpoint on the top right of the curves.

5. CONCLUSION

Hashing-based VQ is a logical step in the right direction to reduce encoding times. Experiments show that hashing-based VQ sustained image quality as the encoding time was reduced, while full-search VQ suffered

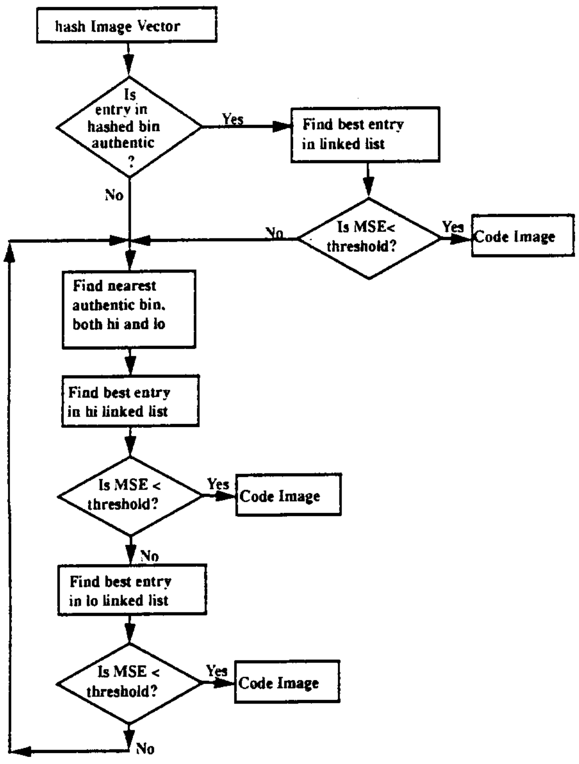
hash Image Vector
Is entry in hashed bin authentic ?
Yes
No
Find best entry in linked list
Is MSE< threshold?
Yes
No
Code Image
Find nearest authentic bin, both hi and lo
Find best entry in hi linked list
Is MSE < threshold?
Yes
Code Image
No
Find best entry in lo linked list
Is MSE < threshold?
Yes
Code Image
No

Figure 4: Hashing based VQ encoding.

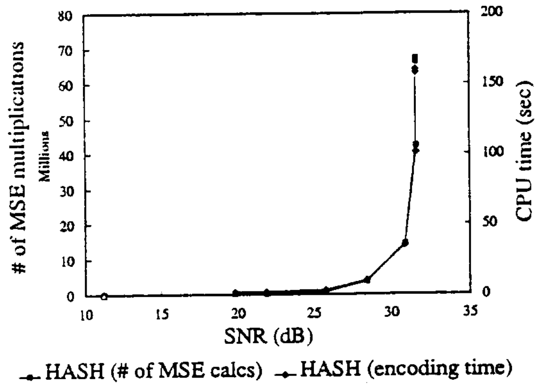

Figure 5: Performance of hashing-based VQ. (2x2 vectors)

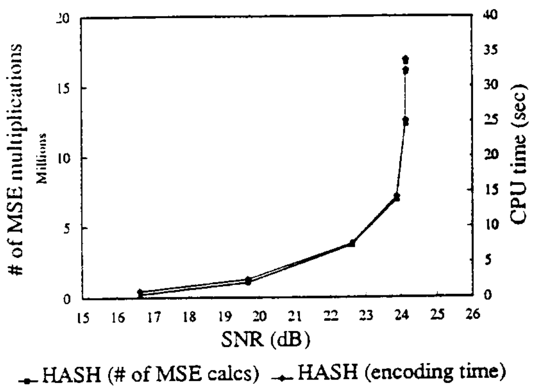

Figure 6: Performance of hashing-based VQ (4x4 vectors)

greatly. The results were consistent across the various vector lengths.

Although only one type of hashing function (i.e., MSB) was explored in this paper, future work will include use of different types of hashing function.

6. REFERENCES

[1] E. Horowitz, *Fundamentals of Data Structures in Pascal*, Computer Science Press, 1984, pp. 452–467.

[2] C. Y. Choo and N. M. Nasrabadi, "Evaluation of Design Parameters for a Cache Vector Quantization System," Proc. International Conference on Image Processing, Austin, Texas, November 1994, pp. 129–133.

[3] Y. Feng, N. M. Nasrabadi, and C. Y. Choo, "A Self-Organizing Adaptive Vector Quantization Technique," *Journal of Visual Communication and Image Representation*, Vol. 2, No. 2, pp. 129–137, June 1991.

[4] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer, Boston, Massachusetts, 1991.

[5] R. M. Mersereau, M. J. T. Smith, C. S. Kim, F. Kossentini, and K. K. Truong, "Vector Quantization for Video Data Compression," in *Motion Analysis and Image Sequence Processing*, M. I. Sezan and R. L. Lagendijk (ed.), pp. 257–283, Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

[6] N. M. Nasrabadi, C. Y. Choo, and Y. Feng, "Dynamic Finite-State Vector Quantization of Digital Images," *IEEE Transactions on Communications*, Vol. 42, No. 5, pp. 2145–2154, May 1994.