Algorithms for Text and Image Compression

Amar Mukherjee amar@cs.ucf.edu
Holger Kruse kruse@cs.ucf.edu
Kunal Mukherjee mukherje@cs.ucf.edu

Department of Computer Science
University of Central Florida
Orlando, FL 32816

1 Introduction

The primary objective of data compression algorithms is to reduce redundancy in data representation in order to decrease data storage requirement. Data compression also offers an attractive approach to reduce the communication cost by effectively utilizing the available bandwidth in the data links. In the nineties, we have seen an unprecedented explosion of digital data on the information superhighways of the world. This data represents a variety of objects from the multimedia spectrum such as text, images, video, sound, computer programs, graphs, charts, maps, tables and mathematical equations. To be able to compact large amounts of multimedia data, and route it through a busy network has emerged as one of the biggest technological challenges of our times. Recently, we have witnessed the simultaneous and rapid development of easy and flexible information browsing on the internet, parallel and multiprocessing, specialized hardware and software for imaging, DSP and multidemdia technologies. These developments have generated an ever increasing number of 'killer' applications which are testing the existing technologies to their limit. Some of these applications are: digital library initiative, medical imaging, digital video production, parallel rendering and visualization, text and image browsing and retrieval by content. Most of these applications typically deploy large and powerful centralized resources serving heterogeneous and widespread client workstations or web browsers. For these applications, a critical need is being felt for aggressive and speedy compression/decompression schemes to deal with the staggering amounts of data on full to bursting network channels.

A generic compression/transmission system has three component parts: an *encoder*, a *channel* and a *decoder*. There has been and continues to be a flurry of activities in the design and implementation of all three components. For this paper, we will assume a noiseless channel and concentrate on the design of algorithms for the encoding and decoding processes, alternately called the *compression* and the *decompression* processes, respectively. The function of the encoder is to produce a representation of the data in a form that takes much less storage than the original data and the function of the decoder is to recover after transmission through the channel the original data. Thus, the decoder performs a reverse function of the encoder function. If the recovery of data is exact, the compression algorithms are called *lossless*. The lossless compression algorithms are used for all kinds of text, scientific and statistical databases, medical and biological images, astro-physical and remote-sensed satellite data. If the recovery of data is approximate, the compression algorithms are called *lossy*. Lossy algorithms are useful in image and video processing and constitute a significant part of data transmission activities. Lossy algorithms also use some form of lossless algorithm at the final stage of encoding to obtain improvement of compression performance.

The objective of this paper is three-fold: first, to make a tutorial presentation of the most important lossless compression algorithm; second, to make a similar presentation of some of the basic concepts and algorithms for image compression using vector quantization, wavelet transform and optical flow; third, to present new research results in both text and image compression conducted in our research group during the last couple of years.

2 Lossless Compression Algorithms

In this section, we will present a survey of lossless compression algorithms. We will discuss fundamental concepts of modeling, entropy coding and then discuss several classical coding techniques such as Huff-

man, arithmetic and some adaptive modeling techniques leading to PPM (prediction by partial match) and and DMC(Dynamic Markov Compression) algorithms. We will then describe the dictionary model based algorithms such as th class of LZ algorithms and conclude by presenting a recently discovered compression algorithm called *bzip* based on Burrows-Wheeler transform.

3 Image Compression algorithms

The main objective of image and video compression algorithms is to compress the image or video data into a compressed representation with constraints imposed by channel bandwidth and storage overhead while maintaining the highest possible quality. Unlike text compression, image compression offers a much wider spectrum of compression from high-definition TV at 20 Mbps to very low-resolution telephone transmission at the rate of 9.6 kbps. The field of image compression is based on a solid foundation of classical methods of transform coding, vector quantization and recent advances of wavelet theory. With the advent of ubiquitous internet technologies and multimedia applications, new research is needed to invent compression algorithms that meet the challenges of network demands, bit rate, image quality and transmission delays for real-time performance.

In this section, we will review briefly some of the fundamental concepts of image compression theory. We will describe the use of block-based transform techniques such as discrete cosine transform(DCT) as well a vector quantization(VQ) and hierarchical vector quantization(HVQ). We will then present the basic theory of wavelets and combine these to develop multi resolution wavelet based hierarchical vector quantization (WHVQ) and embedded zero tree (EZW) algorithm. We will conclude this section with a discussion of predictive methods used for video compression such as differential pulse code modulation(DPCM), and block matching algorithm(BMA) and optical flow methods for motion compensation. Finally, the JPEG and MPEG standards will be presented.

4 New Algorithms for Text and Image Compression

4.1 Text Compression

We first present our results on text compression algorithms.

Star(*) Encoding

It is possible to replace certain characters in a word by a special place holder character and retain a few key characters so that the word is still retrievable. Consider the set of 6-letter words: *packet, parent, patent, peanut*. Denoting an arbitrary character by a special symbol '$*$', the above set of words can be unambiguously spelled as $**c***$, $**r***$, $**t***$, $*e****$. An unambiguous representation of a word by a partial sequence of letters from the original sequence of letters in the word interposed by special characters '$*$' as place holders will be called a **signature of the word**. The collection of English words in a dictionary in the form of a lexicographic listing of signatures will be called an **encoded dictionary** and an English text completely transformed using signatures from the encoded dictionary will be called an **encoded** or **preprocessed** text. We partition the encoded dictionary D into disjoint dictionaries $D_i, (1 \leq i \leq n)$, i denoting the length of words in D_i. Within each D_i we sort the words in descending order based on available frequency information. We then assign a signature to each word based on its location in the ordering. The first word receives a signature consisting of i $*$'s. The next $52i$ words receive a signature consisting of a single letter (either lower case or upper case) in a unique position, surrounded by $i-1$ $*$'s. For example, the second word of length five receives the signature $a****$. The next $52 \times 52 \times C(i,2)$ words receive a signature consisting of two letters in unique positions as a pair surrounded by $i-2$ $*$'s (where $C(i,2)$ represents the number of ways of choosing two positions from i positions). For example, one five-letter word receives a signature of $b*D**$. *It was never necessary to use more than two letters for any signature in the dictionary using this scheme*, although it should be clear how to continue the pattern for three, four, etc., letters. It is important to note that for any encoded text, the most frequently used character will likely be '$*$' and data compression schemes are able to use such redundancy. In fact, '$*$' needs no more than one bit for all the compression methods that we implemented, with the exception of Bzip. The Bzip algorithm uses a different method to achieve its compression, and our research shows that a different method of encoding words yields better results with Bzip.

Properties of Encoding Methods

Each encoding scheme used during preprocessing can be described by a set of rules defining a transformation from an input character sequence to an output character sequence. Each *character* is an element from the set S of valid input symbols. Typically S is the

ASCII character set or a superset of it. In our tests we used the ECMA Latin-1/94 character set, which is an 8-bit extension of the ASCII character set, augmented by a set of international characters.

S can be partitioned into two sets: L, containing all *letters*, and P, containing all other *characters*. *Words* in a dictionary are sequences of *letters*, i.e. sequences of elements of L. *Words* in the input text are sequences of *letters*, which are surrounded by the beginning-of-file marker, the end-of-file marker, or elements of P. This means in the input text a character sequence of elements of L, which immediately follows or precedes another element of L, is not considered a word. A preprocessing algorithm scans through the input text, finds words, tries to locate those words in the dictionary, and if it finds a match it then replaces each word with its *word encoding*. In order to allow the receiver to undo the preprocessing done by the sender, a *word encoding* must have certain properties, to distinguish and delimit it from other parts of the text, and to allow a unique reverse mapping. We defined the following set of rules, which is common to all preprocessing schemes we used:

- The set P is further partitioned into three sets: P_e, which contains a single *escape character*, P_s, which is the set of all *characters* which are used as special characters for *word encodings*, and P_r, which contains all remaining elements of P.
- Each encoding is a sequence of *characters* which contains at least one *character* from P_s, plus zero or more *characters* from L.
- No two different *words* may have the same *word encoding*.
- Any character from P_s or P_e occurring in the input text is escaped using the escape character from P_e.

These rules ensure that each word encoding is unique, and that the receiver can distinguish and delimit word encodings from other portions of the preprocessed file. Typically one of the characters in P_s is used to identify a sequence as a word encoding and to encode the particular word. The other characters in P_s are used to encode special properties of a word, such as capitalization. With the 'star encoding' scheme $P_e = \{\backslash\}$, $P_s = \{\text{\textasciitilde}\,\text{`}\,\text{\textasciicircum}\,*\}$.

The 'Bzip' Encoding Method

The 'Bzip2' compression algorithm described in detail in Section 2, has the advantage that it combines the high speed of algorithms such as gzip with the good compression ratio usually only achieved by rather slow and memory-intensive compression algorithms such as PPM or DMC. The idea behind Bzip2 is that in the second stage 'similar' text sequences in the input file are sorted together. The characters cyclically immediately preceding these sequences are located in the last character position of the corresponding rotation, and are thus passed to the third stage closely together. Often, data files have some amount of local context, i.e. similar text sequences are preceded by the same character, so character sequences can be used to predict the preceding character. Bzip2 uses this principle by sorting rotations in such a way that the characters predicted by similar character sequences are clustered together in the input to the third stage. This increases the locality in the data file, which can then be exploited by move-to-front encoding and Huffman encoding in the third stage. We tried to combine our preprocessing method with Bzip2, to improve the compression ratio of Bzip2 even further. Initial improvements were not as significant as for other compression algorithms (our gain with Bzip2 only averaged 1.6%), so we attempted to find out how our method could be improved to yield better results with Bzip2. An analysis of the interaction between our algorithm and Bzip2 revealed two reasons for the low gain:

- The run-length-encoding algorithm used in the first stage of Bzip2 tends to get used very frequently with the files generated by star-encoding, to replace long sequences of '*' characters by shorter sequences. Even though this decreases the size of the input file, it partially defeats the purpose of star-encoding, because it throws off the model of the second stage of Bzip2, so the encoder does not get the benefit of using very short codes for '*' characters.
- The second stage of Bzip2 basically relies on the observation that a character sequence in the input file can be used to help predict the character preceding it. This property typically holds for the English language (with sequences being words or syllables), but does not hold as well for star-encoded files, because the sequences of '*' characters do not provide enough context to make a reasonable prediction of the preceding character.

From these observations we decided to modify our encoding scheme as follows, called 'Bzip-encoding' in this proposal. Bzip-encoding uses the same sets L, P, P_e, P_s as star-encoding. However the way word

encodings are formed is different: Each word is encoded by a sequence of characters of the same length as the word itself. Exceptions are capitalized words, which require additional encoding. The scheme used to encode capitalization is identical to the one described in [FM96, ?], using characters from P_s. The first character of each word encoding is a '*' character. No other '*' characters appear in an encoding. This allows Bzip2 to strongly predict the space character preceding a '*' character. For words of four or more characters in length, the last three characters form an encoding of the dictionary offset of the corresponding word, in the following manner: entry $D_i[0]$ is encoded as 'zaA'. For entries $D_i[j]$ with $j > 0$ the last character cycles through [A-Z], the second-to-last character cycles through [a-z], and the third-to-last character counts downward from 'z' to 'a', in this order. This allows for 17576 word encodings for each word length, which is sufficient for the dictionaries we used. If more word encodings are required, then this scheme could easily be extended to four or more characters. For words of more than four characters, the characters between the initial '*' and the final three-character-sequence in the word encoding is filled up with a suffix of the string '...nopqrstuvw'. For instance, the first word of length 10 would be encoded as '*rstuvwzaA'. words with three or fewer characters are treated as special cases, and use different encodings:

- Words of length three start with a '*', followed by two characters encoding the dictionary offset. The last characters cycles through [A-Z], and the second-to-last character cycles through [a-z,A-Z], in this order, allowing 1352 different word encodings.
- Words of length two start with a '*', followed by a character that cycles through [A-Z,a-z], allowing 52 different word encodings.
- Only one word of length one can be encoded. Its encoding is a single '*' character.

This encoding scheme provides a stronger local context within each word encoding, and on the border between a word encoding and its delimiters, because there is a correlation between the type of character ('*' character, upper-case characters, lower-case characters with a high value, lower-case character with a low value) and the relative position of that character within the word encoding. This stronger local context is supposed to increase the ability of Bzip2 to predict the preceding character for a character sequence, and thus increase the compression ratio of Bzip2 as a back-end compression algorithm.

Experimental Results

The results reported in this section are preliminary, but we believe that they demonstrate the significant potential of the approach. We used ten text files and an English dictionary as listed in Table I obtained from World Wide Web.[1] We used an electronic version of an English dictionary for our work. This dictionary contained nearly 60,000 words of up to 21 letters long. For frequencies of words in English text, we referred to [HoCo92, CDR71] and used information about the most frequent 100 words. In English, the most frequent words are less than five letters long.

Each of our experiments considered a different compression algorithm augmented with our encoding approach. The compression algorithms used were Unix compress (comp), GNU zip with minimal compression (gzip -1), GNU zip with maximal compression (gzip -9)[Table 2], and arithmetic (arith) using a character based model, DMC, PPM[Table 3] and Bzip[table 5] methods. All compression algorithms were combined with our *-encoding method, with the exception of Bzip: that algorithm was combined with our Bzip-encoding method.

The results of these experiments are summarized in the following tables. It is well known that these algorithms beat the Huffman code in compression performance, so we do not report results for Huffman code in this proposal. The compression is expressed as *BPC* (bits per character) and also as a percentage remaining with respect to the original size of the file. Note that almost all of the encoded compressions yield uniformly better results than the unaided compressions. In addition to the benchmark texts we also used one HTML document (220396 bytes) for testing: the current PNG specification document available from `http://www.w3.org/pub/WWW/TR/REC-png.html`. A summary of average performance of star-encoding is shown in Table 4. The results shown in Table 5 indicate that the new 'Bzip' encoding algorithm improves the compression ratio of the standard compression algorithms 'compress', 'gzip' and 'Bzip' by approximately 5% to 12%, and that the encoding method is suitable for text files as well as HTML documents.

In particular our results for Bzip encoding are promising, because the Bzip algorithm combines excellent compression with low CPU time requirements,

[1]We would like to acknowledge the individuals and organizations who collected these electronic versions online, including Professor Eugene F. Irey (University of Colorado at Boulder) and Project Gutenberg and individuals responsible for the Calgary corpus

and is thus very well suited for many practical applications, and is expected to be used as a standard compression algorithm on the Internet.

4.2 Image and Video Compression

In this section, we will present in the form of abstracts, new results on text and image compression algorithms. The details of the algorithms will be presented verbally at the the Workshop.

Hidden Bit Algorithm

We develop a real-time image coding system capable of adapting instantaneously to the available channel bandwidth. The range of operational bandwidths for the proposed system has a finer calibration than ordinary hierarchical vector quantization (HVQ) or wavelet based hierarchical vector quantization (WHVQ) methods suggested in the literature. These properties make it very attractive for networks with fluctuating available bandwidth, like the internet. All encoder and decoder operations are strictly constant time per pixel, proceeding through table lookups, and are intrinsically suitable for parallel and hardware implementation.

In tree-structured vector quantization (TSVQ), the codewords are arranged in a tree structure, and each input vector is successively mapped from the root node to the minimum distortion leaf node. This successively partitions and refines the input space, as the depth of the tree increases. This enables "embedded coding" to be performed, as more accuracy is added as the path from the root to the leaf is traversed, and additional code (or label) bits are added.

Numerous tree growing and pruning algorithms have been proposed in the literature. A simplistic design is to grow a balanced binary tree one level at a time, typically using the "splitting" method, which is a variation of the generalized Lloyd algorithm (GLA). The first step is identical to the GLA, i.e. finding the optimum 0 bit code - the centroid of the training population. This codeword is then split into two, and the GLA is run to produce a good 1-bit code. The algorithm now diverges slightly from the standard GLA, by splitting the 1-bit codewords by training only on the partition of the training set corresponding to the codeword from the previous step, instead of on the entire. This procedure is then iterated to find a balanced, binary codebook of the required depth, which implements a fixed-rate code.

Embedding a binary tree structure in HVQ code tables

In ordinary vector quantization, the codewords lie in an unstructured codebook, and each input vector is mapped to the nearest neighbor (in terms of Euclidean or similar distortion measure) codeword. This induces a partition of the input space into Voronoi regions. In TSVQ, the codewords are arranged in a recursive tree structure, with successive approximation mapping from the root to the leaf nodes. This induces a hierarchical partitioning of the vector space. Thus, an input vector can be accurately represented by the label of the leaf node, but can also be represented with less accuracy by a prefix of this path from the root, in a "gracefully degradation" of the encoding accuracy, as the path is made shorter. We will now illus-trate our approach, which is an extension of the Makhoul-Riskin-Gray algorithm, .

Growing

We first start the binary tree codebook growing process by finding the centroid of the training set vectors, say node A in Figure 1a. Our tree-structured hierarchical table-lookup vector quantizer (TSHVQ) codebook consists of only one entry at this point. Next, we split node A into B and C, using the standard GLA procedure outlined in [9]. At the end of the training step, our binary tree and codebook appear as shown in Figure 1b. We then follow the Makhoul-Riskin-Gray strategy of greedily splitting the node with largest lambda (lambda = -Delta D/Delta R), until the leaves number the next power of 2, when we re-arrange the codebook (Figure 1c). We carry on in this fashion until we have the required number of leaf nodes in our TSHVQ codebook.

Pruning

As shown in Figure 1d), we organize the 8 "best prunable" nodes in the bottom half of the TSHVQ table. Our definition of "best prunable" follows the generalized BFOS principle, i.e. they are the nodes which produce minimum lambda, while decreasing the bit-rate. With the codebook structure and secondary, tertiary, etc. indexing as shown in Figure 1c, the encoder is able to instantaneously prune away half of the nodes in the table, in an optimal manner, simply by using the secondary indices to index into the top half.

The difference of our TSHVQ approach with that of Makhoul/Riskin/Gray's algorithm is that we do not prune leaves one at a time, but rather prune away sets of 2i leaves at once. This is because we can only change the operational bit-rate of the encoder by integer amounts of bit-rate, and we achieve this by doubling, halving, quartering, etc. the subsets of the available codebooks to index into. The additional investment of time (in the off-line training mode) and space in maintaining the multiple indexing in the TSHVQ tables (see Figure 1c) pays off during encoding, be-

Name	*Size (bytes)*	*Description*
book1	768771	*Calgary Corpus book1*
book2	610856	*Calgary Corpus book2*
news	377109	*Calgary Corpus news*
paper1	53161	*Calgary Corpus paper1*
paper2	82199	*Calgary Corpus paper2*
twocity	760697	*A Tale of Two Cities*
dracula	863326	*Dracula*
ivanhoe	1135308	*Ivanhoe*
mobydick	987597	*Moby Dick*
franken	427990	*Frankenstein*

Table 1: Test corpus

Name	*compress*		*-compress*		*gzip-1*		*-gzip-1*		*gzip-9*		*-gzip-9*	
	%	*BPC*	*%*	*BPC*	*%*	*BPC*	*%*	*BPC*	*%*	*BPC*	*%*	*BPC*
book1	43%	3.46	38%	3.01	47%	3.80	45%	3.58	41%	3.25	37%	2.94
book2	41%	3.28	37%	2.96	41%	3.26	39%	3.15	34%	2.70	31%	2.51
news	48%	3.86	46%	3.69	44%	3.48	42%	3.40	38%	3.06	37%	2.93
paper1	47%	3.77	42%	3.37	41%	3.25	40%	3.16	35%	2.79	32%	2.58
paper2	44%	3.52	38%	3.03	43%	3.41	41%	3.27	36%	2.89	33%	2.60
twocity	41%	3.28	36%	2.86	45%	3.62	42%	3.40	38%	3.05	34%	2.74
dracula	42%	3.35	37%	2.99	46%	3.70	44%	3.5	39%	3.13	36%	2.87
ivanhoe	42%	3.34	37%	2.94	46%	3.68	43%	3.48	39%	3.11	35%	2.82
mobydick	43%	3.46	38%	3.02	47%	3.78	44%	3.55	40%	3.23	36%	2.90
franken	40%	3.19	35%	2.77	46%	3.64	43%	3.42	38%	3.04	34%	2.73

Table 2: Compression results for 'compress' and gzip algorithms

Name	*arith*		*-arith*		*DMC*		*-DMC*		*PPM*		*-PPM*	
	%	*BPC*	*%*	*BPC*	*%*	*BPC*	*%*	*BPC*	*%*	*BPC*	*%*	*BPC*
book1	57%	4.58	44%	3.49	31%	2.48	29%	2.35	31%	2.45	30%	2.38
book2	61%	4.85	48%	3.80	27%	2.19	27%	2.14	27%	2.14	27%	2.16
news	66%	5.24	60%	4.77	35%	2.77	34%	2.71	33%	2.62	32%	2.58
paper1	63%	5.07	52%	4.15	34%	2.73	32%	2.53	30%	2.36	30%	2.38
paper2	58%	4.67	44%	3.53	32%	2.59	29%	2.35	29%	2.34	29%	2.30
twocity	56%	4.51	42%	3.35	29%	2.32	28%	2.21	29%	2.31	28%	2.24
dracula	55%	4.42	42%	3.39	31%	2.44	30%	2.38	30%	2.38	29%	2.35
ivanhoe	57%	4.55	44%	3.53	30%	2.42	29%	2.29	30%	2.36	28%	2.27
mobydick	56%	4.50	42%	3.36	32%	2.57	30%	2.41	31%	2.48	30%	2.36
franken	56%	4.46	40%	3.20	30%	2.38	28%	2.22	29%	2.29	28%	2.23

Table 3: Compression results for arithmetic, DMC, and PPM algorithms

Name	*original BPC*	*BPC with *-enc*	*gain*
compress	3.57	3.20	10.4%
gzip -1	3.45	3.31	4.1%
gzip -9	2.97	2.73	8.1%
arith	4.84	3.96	18.2%
DMC	2.73	2.51	8.1%
PPM	2.94	2.74	7.8%

Table 4: Average performance for star-encoding

Name	*Bzip2 -9*		*e-Bzip2 -9*	
	%	*BPC*	*%*	*BPC*
book1	30%	2.42	29%	2.32
book2	26%	2.06	25%	2.00
news	31%	2.52	31%	2.45
paper1	31%	2.49	29%	2.35
paper2	30%	2.44	28%	2.27
twocity	28%	2.26	27%	2.17
dracula	29%	2.36	29%	2.30
ivanhoe	29%	2.39	28%	2.23
mobydick	31%	2.46	29%	2.33
franken	29%	2.29	27%	2.16

Table 5: Compression results for Bzip algorithm

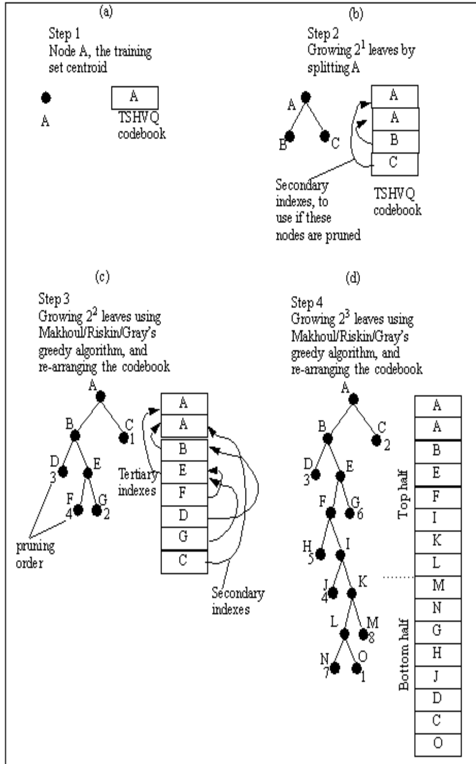

Figure 1: Growing and Organizing Binary Tree Structured Codebook

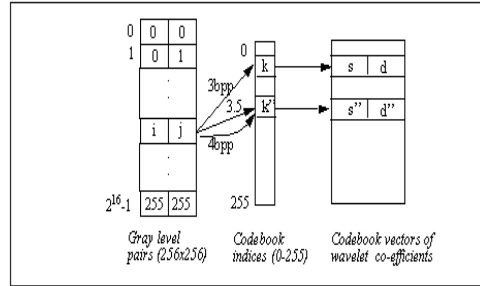

Figure 2: Codebook with Multiple Indexing for Tunable Bit Rates

cause we are able to instantaneously and optimally (due to BFOS principle) switch the operational bit-rate of the encoder, on the fly, in response to fluctuations in available bandwidth of the channel. So the encoder is very suitable for operating in a feed-back loop with a channel monitoring module.

Encoder and Decoder operations

We will illustrate these operations with reference to an example, and Figure 2. Say we are encod-ing at 4 bpp, i.e. sending the 8 bit codes for length 2-vectors, after 1 level of wavelet based HVQ (WHVQ). If instead we mapped all possible incoming pairs (28 x 28) to a subset of 27 of the 28 codebook vectors, then we would be using $7/2 = 3.5$ bpp. We would also not be using (bottom) half of the available codebook. If we used a 26 subset, we would use 3 bpp, and so on. This is pos-sible, because we create multiple (i.e. secondary and tertiary in this case) indexing in the 28 x 28 input table, for different bit rates. This is shown in Figure 2.

We are assured that the increased distortion at lower operational bit-rates will be optimal, due to the binary tree-structured HVQ codebook design explained in section 2. This makes the act of indexing into the top half of the available codebook equivalent to optimally pruning away all the leaves in the search tree, and yet doing this implicitly, through table lookups, and in constant time.

If we carry on the lookup-and-transform to the next stage, then we get 2 bpp if we use all 28 code-book vectors from the second level lookup table, 1.75 bpp if we use a 27 sub-set, 1.5 for 26, etc. If we need a bit rate of 2 or less, we should use the full codebook set of the next level, as the ultimate coding rate will only depend on the rate of the final stage, and there is no point in coarse coding at the preceding levels. This gives us a finer calibration of operational bit-rates than either the last-stage table TSHVQ, or straight HVQ. The decoder stays the same as in ordinary HVQ - it

directly looks up the codebook vectors, and uses these to reconstruct the encoded images.

Embedded Optical Flow Based Motion Compensation in Finite State Hierarchical Vector Quantization

We propose a video coding and delivery scheme which is geared towards low bit-rate and real-time performance requirements. We use a finite state wavelet-based hierarchical lookup vector quantization (FSWHVQ) scheme, which embeds the optical flow calculations in table-lookups. This video coding scheme is both fast (table-lookups) and accurate (dense motion field), and avoids the blocking artifacts and poor prediction which plagues block coding schemes at low bit rates. For restricted image compression/ transmission scenarios like teleconferencing, for which a good training set may be available, the FSWHVQ scheme may be viewed as storing as an internal representation in its lookup tables, a valid and complete model of the problem domain. To get this kind of compression, we need to squeeze out the last drops of spatial and temporal correlation, by using a combination of motion compensation (temporal) and block coding or vector quantization (spatial) techniques. MPEG (BMA/DCT) seemed to be a good solution, by estimating the motion of each image block by a single vector. But at very low bit rates, this coarse approximation suffers from very visible and unpleasant artifacts. Therefore, we are faced with the following design criteria for any practical low bit-rate compression scheme: a) The motion estimation should be accurate. This automatically means that the residual errors will be small, and can be compactly coded; b) Rate/ distortion (R/D) should be optimal, and ideally this should be adaptable (i.e. dynamically tunable) according the bandwidth availability; and c) Both encoding and decoding should be fast, possible in real time, and/or realizable in VLSI. In this work we are motivated to satisfy the above design criteria as closely as possible. To this end, we propose the following solu-tions: a) Use optical flow instead of block matching, to obtain a dense and accurate motion field estimation; b) Use vector quantization, which uses an overall rate-distortion optimization scheme in the codebook design (i.e. training) phase; and c) use FSHVQ (finite-state hierarchical table-lookup vector quantization), which "embeds" the optical flow computa-tions, by storing state information. This gives a fast encoder and decoder which proceed through table-lookups rather than complex computations. The operational bit-rate can then be dynamically tuned to the available bandwidth, by using successive stages of table-lookups. Optical flow has been used in other video coding work, but not in a fast coding scheme like table-lookups which is practical for a real-time encoder, and which can be implemented in hardware. The idea of embedding higher-level, e.g. classification tasks in the HVQ frame-work was introduced by Chaddha and Gray. Finite state table-lookup hierarchical vector quantization (FSHVQ) was used by Chaddha, Mehrotra and Gray for still image coding, using side intensities as the "state". We extend the FSHVQ idea for image sequences, by using code-words used to encode the previous frame, and previously esti-mated motion fields as the "state". The new technique introduced here is the embedding of optical flow in FSHVQ for video coding.

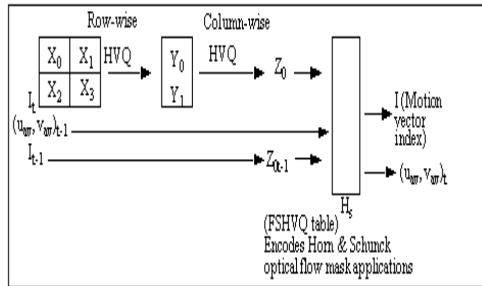

Figure 3: Derivation of the index of Optical Flow

Embedding Optical Flow in Finite State HVQ

We will illustrate the method with a specific example (Figure 3). In normal HVQ, blocks of pixels are vector quantized (using table-lookups) by quantizing, or "merging" two vectors (at the first step these are a pair of pixel values), alternatively, along rows and columns. So a 2x2 block of pixels (Figure 3) is quantized into one table index in two steps.

First, two row-wise lookups quantize the row-wise pixel values to give two table indices, $Y0$ and $Y1$, followed by a column-wise second-level table lookup, which quantizes the column-wise indices into one index, $Z0$, which represents a 2x2 code-book vector. Horn and Schunck's optical flow method proceeds by apply-ing spatial and temporal derivative masks, e.g. of size 2x2, at corresponding spatial locations of the current and previous images. So if the quantized code-book indices of the previous frame (at the 2x2 block quantization level) are stored, along with the motion field averages u_{av} and v_{av} as the state, then the optical flow vector at time t and at location x, y can be conveniently expressed as a function of the state at time t, s_t, and the previous and current images, i_{t-1} and i_t by the equation $(u, v)_t = f(s_t, i_{t-1}, i_t)$ This simply means that feeding the (2x2 level) code-book

indices $Z0_t$ and $Z0_{t-1}$ into H_s, the FSHVQ table for state s, we can look up the index of the motion vector, (u, v). The FSHVQ code books and tables are, of course, populated in the training phase, by using the Horn and Schunck formulation. This embeds the optical flow computation in table-lookups, and gives an extremely fast encoder. The residual errors for each block can be sent using a lower bit-rate HVQ scheme (since the residual errors are usually small and randomly distributed), or by a lattice vector quantization (LVQ) scheme, which requires no additional training. To complete the encoding, the state is updated by storing a matrix of the local motion field vectors (u_{av}, v_{av}), and the output or reconstructed image is stored as the "previous image", it for the next frame at $t + 1$, as shown by the equation $i_t = \hat{i}_t$. This allows the encoder to track the decoder, and prevent error accumulation - it represents the "closed loop" of the generic FSHVQ system, as the previous image, it, and the local motion field averages, (u_{av}, v_{av}), together comprise the state of this encoder.

Coding Results

We show some experimental results of coding the "Claire" news caster sequence at 100 KB/s (sustained) after the initial frame was sent (intracoded) in Figure 4.

The optical flow technique gave an average PSNR of 36.0dB, whereas BMA gave 34.8dB. Also our algorithm performed much better at the edges, whereas BMA gave severe blocking artifacts.

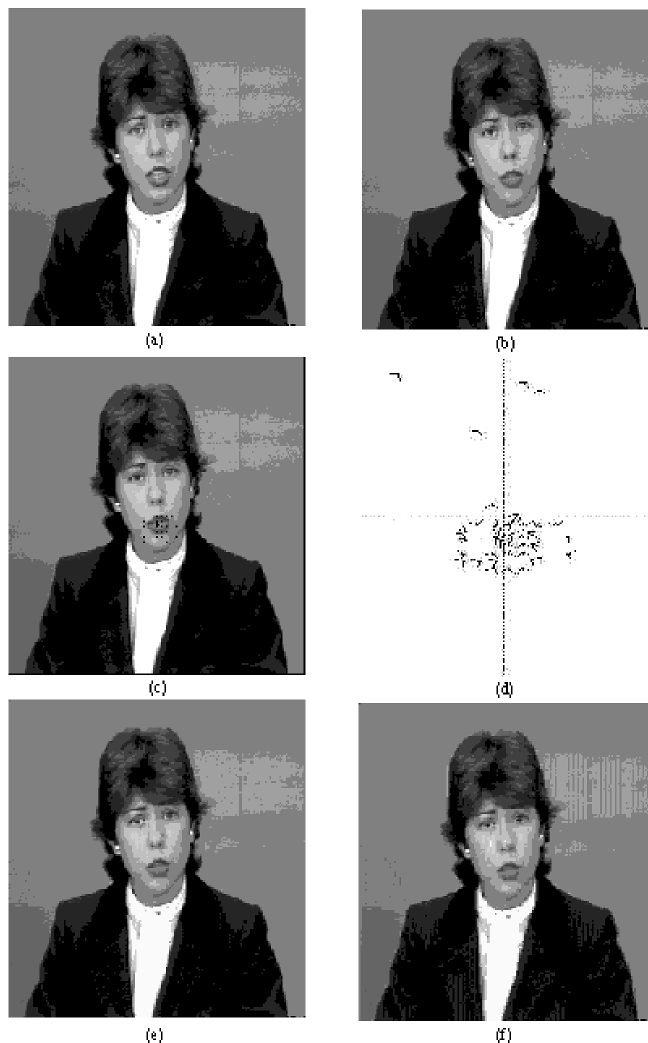

Figure 4: Experintal Results

References

[BCW90] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice-Hall, 1990.

[BM89] T. C. Bell and A. Moffat, "A Note on the DMC data compression scheme", *Computer Journal*, Vol.32, No.1,1989,pp.16-20.

[BuWh94] M. Burrows, D.J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm", SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA.94301, May 10,1994.

[CDR71] J. B. Carroll, P. Davies, B. Richman. *The American Heritage Word Frequency Book*,1971.

[CW84] J. G. Cleary and I.H. Witten, "A Comparison of Enumerative and Adaptive Codes",

IEEE Transactions on Information Theory, Vol.IT-30,1984,pp.306-315.

[CH84] G. V. Cormack and R. N. Horspool, "Data Compression Using Dynamic Markov Modeling", *Computer Journal*,Vol.30,No.6, 1987,pp.541-550.

[CoKi78] T. M. Cover and R. C. King,"A Convergent Gambling Estimate of the Entropy of English",*IEEE Trans. on Information Theory*,Vol.IT-24, pp.257-264, 1978.

[El75] P. Elias, "Universal Codeword Sets and Representation ons of the Integers", *IEEE Trans. Info. Theory* 21, 2(Mar.1975), pp.194-203.

[Fa49] R.M. Fano, "Transmission of Information", *MI T Press*, Cambridge, Mass., 1949.

[FiGr89] E.R. Fiala and D.H. Greene. "Data Compression with Finite Windows", **Comm. ACM**, 32(4), pp. 490-505, April, 1989.

[FM96] R. Franceschini and Amar Mukherjee, "Data Compression Using Encrypted Text", *Proc. Forum on Research and Technology Advances in Digital Library* , Washington D.C., May 13-15, 1996.

[Ga78] R.G. Gallager, "Variations on a theme by Huffman", *IEEE Trans. Information Theory*, IT-24(6), pp.668-674, Nov, 1978.

[HL90] D.S. Hirschberg and D.A. Lelewer. "Efficient Decoding of Prefix Codes", *Communications of the ACM*, Vol.23,No.4,pp.449-458, April,1990.

[HoCo92] R.N. Horspool and G.V. Cormack. "Constructing Word-based Text Compression Algorithms", *Proc. Data Compression Conference*, 1992, (Eds. J.A. Storer and M. Cohn), IEEE Computer Society Press, 1992, pp. 62-71.

[Hu52] D.A.Huffman. "A Method for the Construction of Minimum Redundancy Codes", *Proc.IRE*, 40(9),pp.1098-1101,1952.

[KrMu96] Holger Kruse, Amar Mukherjee. "Data Compression Using Text Encryption", *Proc. Data Compression Conference*, 1997, IEEE Computer Society Press, 1997, p. 447.

[KrMu98] H. Kruse and Amar Mukherjee, "Preprocessing Text to Improve Compression Ratios", accepted for publication in the Proceedings of Data Compression Conference, Snowbird, Utah, March,1998 (extended abstract)).

[LZ77] J. Ziv and A.Lempel. "A Universal Algorithm for Sequential Data Compression. *IEEE Trans on Information Theory*,IT-23,pp.337-243,1977.Also, by the same authors "Compression of Individual Sequences via Variable Rate Coding", IT-24,pp.530-536,1978.

[RiLa79] J. Rissanen and G.G. Langdon, "Arithmetic Coding" *IBM Journal of Research and Development*, Vol.23, pp.149-162, 1979.

[SK64] E.S. Schwartz and B. Kallick. "Generating a Canonical Prefix Encoding", *Communications of the ACM*, Vol.7, No.3, pp.166-169,March,1964.

[St88] J.A. Storer. "Data Compression: Methods and Theory", Computer Science Press,1988.

[St92] J. A. Storer et al. *Proceedings of the Data Compression Conferences*, Snow Bird, Utah, 1992-96.

[We84] T. Welch, "A Technique for High-Performance Data Compression", *IEEE Computer*, Vol. 17, No. 6, 1984.

[Wi91] R. N. Williams, "Adaptive Data Compression", Kluwer Academic, 1991.

[WNC87] I. H. Witten, R. Neal and J. G. Cleary,"Arithmetic Coding for Data Compression", *Communications of the ACM*, Vol.30, No.6,1987,pp.520-540

[WMB94] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York,1994.

[BB82] Ballard,D.H. and Brown,C.M., Computer Vision, Prentice Hall, 1982.

[CV96] Chaddha, N. and Vishwanath, M., "A Low Power Video Encoder with Power, Memory and Bandwidth Scalability", 9th International Conference on VLSI Design, 358-363, 1996.

[BFOS84] L.Breiman, J.H.Friedman, R.A.Olshen and C.J.Stone, "Classification and Regression Trees" (The Wadsworth Statistics/Probability Series), Belmont, CA:Wadsworth, 1984.

[CMC95] N.Chaddha, V.Mohan and P.Chou, "Hierarchical vector quantization of perceptually weighted block transforms", Proc. Data Compression Conference, IEEE, Piscataway, NJ, pp.3-12, 1995.

[CCG96] N.Chaddha, P.A.Chou and R.M.Gray, "Constrained and Recursive Hierarchical Table-Lookup Vector Quantization", Proc. Data Compression Conference, IEEE, Piscataway, NJ, pp.220-229, 1996.

[CMG85] P.C.Chang, J.May and R.M.Gray, "Hierarchical Vector Quantization with Table-Lookup Encoders", Proc. Int. Conf. on Communications, Chicago, IL, pp. 1452-1455, 1985.

[CMG96] Chaddha, N., Mehrotra, S., and Gray, R.M, "Finite State Hier-archical Table-Lookup Vector Quantization for Images", Pro-ceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, part 4, pp.2024-2027, 1996.

[CPG96] Chaddha, N., Perlmutter, K., and Gray, R.M., "Joint Image Classification and Compression using Hierarchical Table-Lookup Vector Quantization", Data Compression Conference, pp. 23-32, 1996.

[DC97] Diab, Z., and Cohen, P., "Motion Compensated Video Com-pression using Adaptive Transforms", Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, Part 4, pp. 2881-2884, Munich, Germany, 1997.

[GG92] A.Gersho and R.M.Gray, Vector Quantization and Signal Compression, Kluwer Academic Publishers, 1992.

[LS97] Lin, S. and Shi, Y.Q., "An Optical Flow Based Motion Com-pensation Algorithm For Very Low Bit-Rate Video Coding", Pro-ceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, V4, pp. 2869-2872, 1997.

[LBG80] Y.Linde, A.Buzo and R.M.Gray, "An algorithm for vector quantizer design", IEEE Transac-tions on Communications, COM-28:84-95, Jan. 1980.

[MRG85] J.Makhoul, S.Roucos and H.Gish, "Vector quantization in speech coding", Proc. IEEE, vol.73, pp.1551-1558, 1985.

[MKW97] Moulin, P., Krishnamurthy, R., and Woods, J.W., "Multiscale Modeling and Estimation of Motion Fields for Video Coding", IEEE Transactions on Image Processing, Vol. 6, No. 12, pp. 1606-1620 December 1997.

[PS97] Podilchuk,C.I. and Safranek,R.J. "Image and video compres-sion: A review", International Journal of High Speed Electronics and Systems, Vol. 8, No.1, pp. 119-177, 1997.

[RG91] E.A.Riskin and R.M.Gray, "A Greedy Tree Growing Algorithm for the Design of Variable Rate Vector Quantizers", IEEE Transactions on Signal Processing, Vol.39, No.11, 1991.

[RS97] Regunathan, S.L. and Rose, K., "Motion Vector Quantization in a Rate-Distortion Framework", Proceedings of IEEE International Conference on Image Processing, part 2, pp.21-24, 1997.

[VC94] Vishwanath, M. and Chou, P., "An Efficient Algorithm for Hierarchical Compression of Video", Proceedings of the International Conference on Image Processing - 1994, Austin, Texas, vol. 111, pp. 275-279, November 1994.