Length–Limited Variable–to–Variable Length Codes For High–Performance Entropy Coding

Joshua Senecal*‡ Mark Duchaineau† Kenneth I. Joy‡

*Institute for Scientific Computing Research
†Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

‡Center for Image Processing and Integrated Computing
Computer Science Department
University of California, Davis

Abstract

Arithmetic coding achieves a superior coding rate when encoding a binary source, but its lack of speed makes it an inferior choice when true high–performance encoding is needed. We present our work on a practical implementation of fast entropy coders for binary messages utilizing only bit shifts and table lookups. To limit code table size we limit our code lengths with a type of variable–to–variable (VV) length code created from source string merging. We refer to these codes as "merged codes". With merged codes it is possible to achieve a desired level of speed by adjusting the number of bits read from the source at each step. The most efficient merged codes yield a coder with a worst–case inefficiency of 0.4%, relative to the Shannon entropy. Using a hybrid Golomb–VV Bin Coder we are able to achieve a compression ratio that is competitive with other state–of–the–art coders, at a superior throughput.

1 Introduction

With the rapid progression of computer technology high–performance computing is becoming cheaper and more widely available. Computer simulations are examining more complex problems at increasingly greater levels of detail, and these computations are creating greater amounts of data that must be stored for analysis. For example, a simulation of a Richtmyer–Meshkov instability and turbulent mixing was performed at a resolution of 2048 x 2048 x 1920 on the IBM Sustained Stewardship TeraOp system located at Lawrence Livermore National Laboratory [8]. The simulation run generated over three terabytes of data which needed to be compressed and stored. We would like to add to simulation codes the ability to encode data on the fly, before it is written to disk. Such an entropy coder must have a high throughput that is not adversely affected by changes in the incoming data. This minimizes degradation of the computation's performance, and ensures that the amount of degradation is known

*L–419, PO Box 808, Livermore, CA 94551, Tel: 925–422–3764, Fax: 925–422–7819 *senecal1@llnl.gov*

†L–561, PO Box 808, Livermore, CA 94551, *duchaine@llnl.gov*

‡One Shields Ave, Davis, CA 95616, *kijoy@ucdavis.edu*

Proceedings of the Data Compression Conference (DCC'04)
1068-0314/04 $ 20.00 © 2004 IEEE

and relatively constant, no matter the data being encoded. The coder must also have good compression ratios, so that the resulting compression is worth the computational expense. Arithmetic coders and their variants achieve excellent compression ratios, but their speed varies for different bit distributions, and this makes them unsuitable for a high–performance computing application.

We present here some results stemming from our work on developing a fast entropy coder that performs only bit shifts and table lookups during the coding process. When using lookup tables the symbol and code lengths become critical, since the number of entries in the decode table is dictated by the length of the longest codeword. To place a limit on the sizes of our code tables, our coder uses merged source string codes (hereafter referred to as "merged codes"), which are a type of variable–to–variable length (VV) code. Each code has a source window of W bits, reading up to W bits per coding iteration, and outputs a codeword that is up to C bits in length. When referring to a merged code created with a specific (W, C) we call it a (W, C) merged code.

We create a set of optimal VV code tables for codes restricted to string lengths of 15 bits and code lengths of 13 bits. We use these tables along with Golomb codes in a hybrid bin coder that is able to encode a binary source with a compression ratio comparable to state–of–the–art arithmetic coders, but at a superior throughput.

Our work is related to the State–Tree Code (STC) adaptive VV coding method proposed in [11], but differs in some respects. The STC's codes are generated from a set of parse trees with a specified number of leaves. The codes used are optimum—generated through an exhaustive search—and the number of strings in the code is necessarily kept small (8– and 16–leaf parse trees are used). Our codes are merely optimal within a limited search space, but have a higher leaf count. We do not care how many leaves are in the parse tree, we only concern ourselves with the longest code length. Also, the STC does not divide up source bits according to probability, but only switches code trees after encoding a binary string from the source.

2 Encoding a Binary Source

2.1 Review and Terms

Given a binary memoryless source each bit b_i in the source has a probability of being either a 0 or a 1, denoted respectively by the pair (p_i, q_i). We assume without loss of generality that the more probable symbol is 0, that is, $q \leq 0.50$. For a binary memoryless source with probabilities (p, q) the entropy H of the source is defined as [9]:

$$H(p, q) = -(p \log_2 p + q \log_2 q). \tag{1}$$

From this we define the *coding inefficiency* I of a coder K at (p, q) as

$$I_K(p, q) = \frac{R_K(p, q) - H(p, q)}{H(p, q)} \times 100 \tag{2}$$

where $R_K(p,q)$ is the *coding rate* of coder K at (p,q). $R_K(p,q)$ for a coder is computed empirically as

$$R_K(p,q) = \frac{num\_bits\_out}{num\_bits\_in} \tag{3}$$

Given a binary memoryless source and a parse tree of strings for the source, a string s of some length $|s|$ has a weight $w = p^z q^y$, where z is the number of zeros in the string and y is the number of ones. If the parse tree recognizes N strings, we compute the theoretical coding rate as

$$R_K(p,q) = \frac{\sum_{i=1}^{N} |c_i| w_i}{\sum_{i=1}^{N} |s_i| w_i} \tag{4}$$

where $|c_i|$ is the length of the prefix–free codeword assigned to s_i. *Theoretical inefficiency* refers to I computed with a theoretical R, while *empirical inefficiency* refers to I computed with an empirical R. In this work we consider a code to be *efficient* if it is within 1% of the source entropy, i.e $I \leq 1$. We define a code's *point of inefficiency*, as q tends to 0, as the value of q at which the inefficiency curve goes above 1% and does not return below it.

2.2 Methods and Prior Work

A binary source can be encoded bit–by–bit, or by creating a parse tree for the source, thus creating a set of binary strings that may appear in the source, and encoding the source one string at a time. Binary arithmetic coders [16] use the former approach. They give a theoretically superior coding rate, but are slow. Golomb codes [5] also use this approach, and for skewed probabilities can quickly encode runs of bits at a time, but Golomb codes are unable to encode efficiently at certain probabilities.

For speed, encoding a source in blocks or strings is preferred, as table lookups can be used to get string–codeword associations. There are three common types of codes that lend themselves to table–lookup methods: Block–to–Variable (BV), Variable–to–Block (VB), and Variable–to–Variable (VV).

Probably the best–known example of the BV codes is the use of the Huffman algorithm [6] to generate codes for encoding byte data, such as ASCII text. This method can also be used to encode binary data in n–bit blocks, resulting in 2^n possible binary strings. The best example of the VB codes is the Tunstall algorithm (reviewed in [2]). In this method the parse tree is grown by splitting and extending the most probable leaf until there are 2^n leaves. Then n–bit codewords are assigned to each leaf.

The most common way of creating a VV code uses a variant of the Tunstall extension approach to create the parse tree, and uses the Huffman algorithm to generate the code words. To our knowledge most work in VV code algorithms (see [3, 4, 10, 11] for example) focuses on selecting a leaf for extension such that the resulting parse tree gives the best coding rate. The process of determining which leaf to extend is notoriously difficult, and currently it seems that an exhaustive search is the only possible method for finding an optimal code.

All methods based on table lookups have a potential problem: for a direct lookup table approach the number of entries in the encoding and decoding tables is a power of 2 of the length of the longest string or codeword. Even a reasonably small string or code length may result in an undesirably large lookup table.

In these situations length–limited (LL) codes are preferred. The best–known algorithm for generating length–limited codes is the Package–Merge (PM) algorithm [7, 12, 13]. PM can often reduce the longest code length dramatically, with a miniscule cost to coding efficiency. Figure 1 shows the coding inefficiency of the most efficient codes generated by the Tunstall, Huffman, and PM algorithms, where symbol and code lengths are not allowed to be over 15 bits. For the Huffman and Tunstall methods, length limiting is performed by respectively shortening the input and output block lengths.

3 Length–Limited VV Codes

When length–limited codes are required an alternative to PM and related methods is the family of VV codes. Despite the difficulty in creating and analyzing them theoretically, VV codes are ideal for length–limited binary encoding. They give a throughput superior to arithmetic coders and have a competitive coding rate.

Since we consider this from the viewpoint of needing length–limited codes, we generate VV codes through *contraction*, via a greedy process we term "source string merging". Conceptually, given W and C, we generate a complete, balanced parse tree with 2^W leaves, where the path length to each leaf is W. Code lengths are generated for the strings, and the string with the longest code length is merged, along with its sibling, into its parent, which then becomes a leaf. The process repeats until all code lengths are $\leq C$. In essence the merging process transforms a BV code into a VV code. Given (p, q) and (W, C) the merging algorithm is shown in figure 3. The reasoning behind source string merging is akin to that of PM: the least likely items can be manipulated, because they will affect the overall coding rate the least. Unlike PM, the merging process alters or eliminates source strings, and this can increase the efficiency of the resulting code.

3.1 Theoretical Inefficiency of Merged VV Codes

We examined merged VV codes created with an input bit window size $7 \leq W \leq 15$, and an output code length limit $7 \leq C \leq 13$. For each value of q, at a granularity of 0.01 we created a coding table with the parameters (W, C). For each table we measured its inefficiency at the table's q and at ± 0.001, ± 0.002, etc., until the inefficiency in both directions went above 5%. For each value of q at a granularity of 0.001 we recorded which table was most efficient.

Figure 2 shows the inefficiency curve that results if for each value of q the most optimal code is selected from among those created in our study. The reduction in inefficiency is dramatic. Excluding the value where the curve rises to the point of inefficiency a coder using these codes would have a worst-case inefficiency of 0.4%.

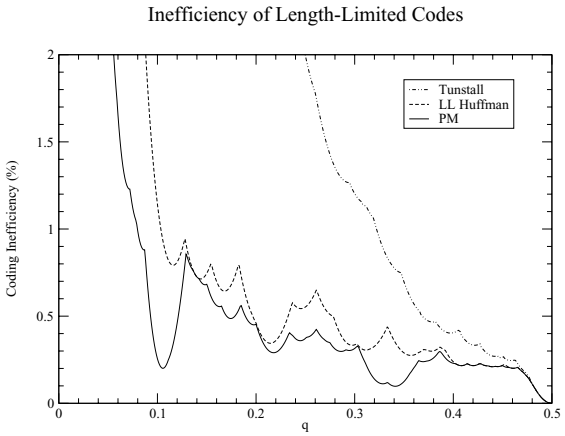

Figure 1: Coding inefficiency of length–limited codes.

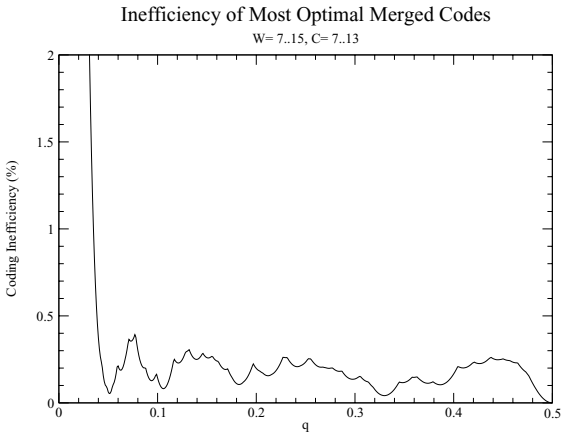

Figure 2: Coding inefficiency of most optimal codes.

Generate all 2^W source strings
Compute each strings' weight
Compute a code length c_i for each string
while ($\exists\ c > C$) {
    select string with longest code length c
    merge with sibling into parent
    compute weight of new, shorter string
    reassign code lengths to all strings
}
Generate prefix–free codes

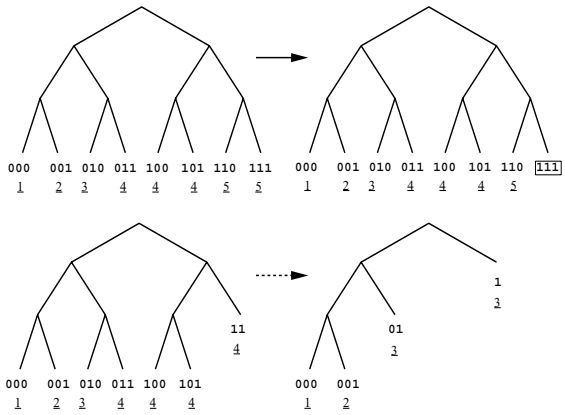

Figure 3: Example of merging leaves on the source parse tree. Underlined numbers are code lengths.

4 A Hybrid Bin Entropy Coder

VV codes are extremely difficult to adapt and this means that their only practical use is for encoding a memoryless source. The only way to use VV codes to encode a source where the bit probabilities may change is to have a collection of code tables on hand and swap tables on the fly as needed or, more practically, use a bin coder, each bin designed for a certain probability range. As the probability of each bit is determined the bit is placed into the appropriate bin.

Bin coders are not usually thought of as being a good solution because the binning operations hurt performance. This is true only because most applications of bin coding seek to interleave the bin outputs into a single coded bitstream. The interleaving process requires that the encoder maintain bin priorities, and add extra bits ("flush bits") to bins to force output when needed. Usually bin outputs are interleaved on a codeword–by–codeword basis. This type of fine–grained bin interleaving is appropriate for circumstances such as live streaming over a single channel, but may not be

necessary for other applications, such as batch processing large amounts of data. A much coarser level of interleaving (or none at all, in the extreme case) can be used with a corresponding decrease in the overhead, and increase in the execution speed and compression ratio.

Figure 2 shows that the VV codes' point of inefficiency is about $q = 0.04$. To encode bits that fall into $0 < q \leq 0.04$ an alternative coding method is needed, and we choose Golomb coding. In this range the expected run lengths are such that optimizations can be made to quickly encode runs of bits in a single block and not bit–by–bit. We therefore propose a hybrid bin scheme: all bins below $q = 0.04$ use Golomb coding, and all others use VV codes.

5 Results

We created two hybrid bin coders, one with with 22 bins and the other with 25. In each case the coder used 18 VV code bins, and the remaining were Golomb bins. VV codes used were selected from those created in our study. These coders perform no interleaving. We compared our bin coders to three other coders: the Augmented ELS–coder [14][1], the Z–coder [1][2], and an arithmetic coder written by us. The ELS–coder does not appear to have been designed with speed as a concern [15], but we include it because its coding rate is state–of–the–art. In this work we are interested in measuring and comparing the efficiencies of the coders themselves, not the efficiency of the probability estimation techniques developed for these coders—we supply the bits and the probabilities to be used to encode them. We therefore decoupled the the actual coders from their state tables and from each state table created a lookup table so that, given a bit and a probability, the coder could obtain the information it needed.

In all of our tests we assume without loss of generality that the least probable bit is a 1, and all our results are given with respect to q, the probability of encountering a 1. We tested each coder on nonstationary sources, where q varies bit–to–bit. To test for a nonstationary source we created a block of 256 probabilities, given according to the following equation:

$$P(i) = \frac{i^x}{2 \times 256^x}, 0 < i \leq 256 \tag{5}$$

where $1 < x \leq 10.1$ is a floating–point parameter that determines the probability distribution, and i is an integer. The resulting probabilities were converted to the form used by each coder. After a block was generated it was permuted, and a 30 megabyte bitstream was generated by repeatedly iterating through the block, generating one bit per entry based on the probability in that entry.

Timings were taken on a 1–GHz Intel Pentium III computer running Red Hat Linux. All execution times given are an average over 5 runs. Execution time for the bin coder is a total of time to both bin and encode all the source bits. Results for

[1] ftp://www.pegasustools.com/Osaugels.zip

[2] http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/djvu/djvulibre-3.5/libdjvu

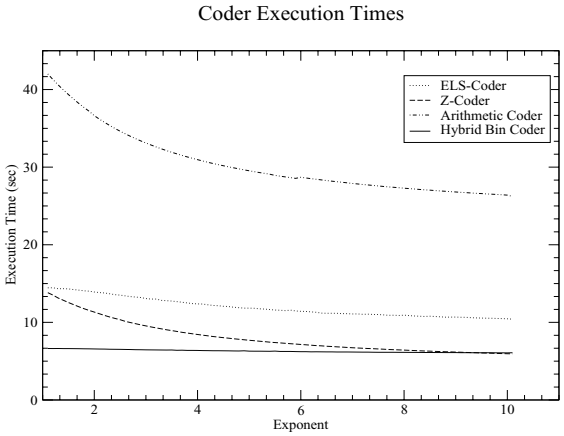

Figure 4: Execution time of the coders.

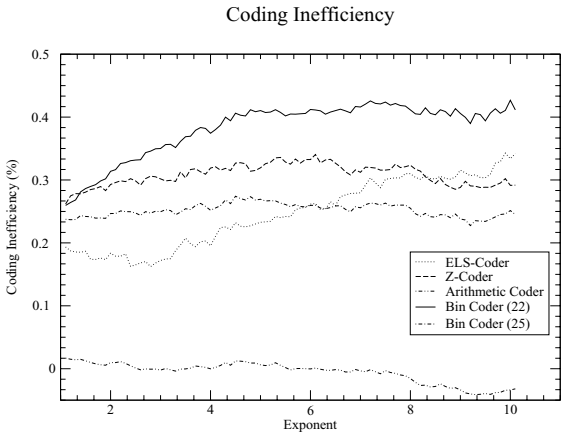

Figure 5: Coding Inefficiency of the five coders.

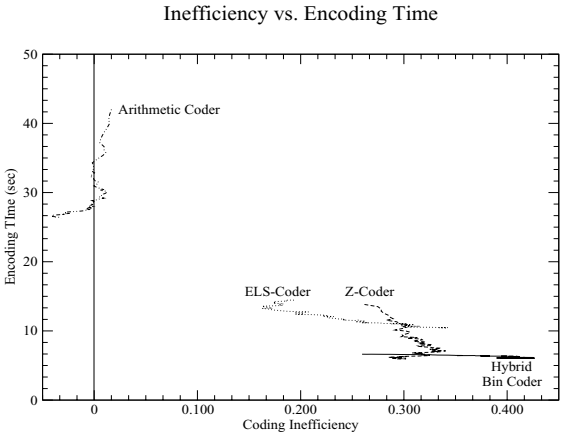

Figure 6: Coding Inefficiency vs. Time of the 22–bin hybrid bin coder.

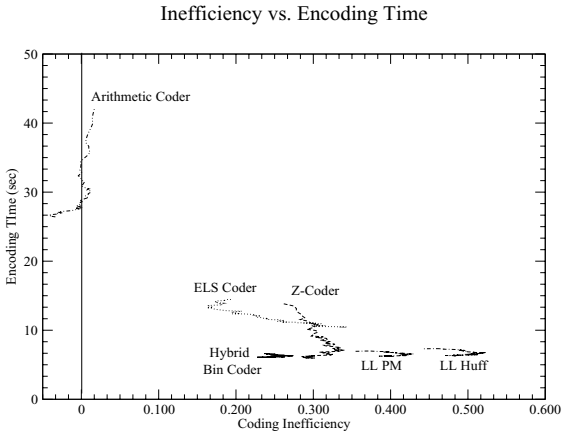

Figure 7: Coding Inefficiency vs. Time of the 25–bin hybrid bin coder.

the execution times of the coders are given in figure 4. Time results for the 25–bin coder were similar to the 22–bin one, and are not shown. The "Exponent" axis values indicate the parameter x, used to determine the bit distribution.

Figure 5 gives the coding inefficiency of each coder, as computed by equation 2. For more skewed probability distributions there is some experimental error, which is particularly apparent with the arithmetic coder in the form of its negative inefficiency. This is due to using an insufficient number of bits when taking the measurements at these skewed probabilities.

A more informative picture of the results is figures 6 and 7, comparing the ELS, Z, and Arithmetic coders respectively to the 22– and 25–bin hybrid coders. These figures give a graph of each coder's execution time versus its inefficiency. Smaller values are better, so an ideal coder would be one that is close to the origin, having a small inefficiency and fast execution time. In figure 7 we also show results for hybrid coders using 15–bit maximum length–limited BV codes, instead of VV codes. These BV codes are generated by the PM and Huffman algorithms, and are selected from

the codes used to generate figure 1. The PM bin coder has 26 bins total, and the Huffman bin coder has 28, of which, respectively, 9 and 10 are Golomb code bins.

6 Discussion

From figure 4 we see that the Bin Coder's execution time is generally the same for all tested bit distributions. This is a plus—assuming that the incoming bits' probabilities are reasonably modeled the execution time is consistent. The execution times of the arithmetic, Z, and ELS coders vary as the bit distribution varies.

From figures 6 and 7 we see that the the addition of three more Golomb bins has reduced the inefficiency dramatically while leaving the execution time relatively unchanged. We also note from figure 5 that the 25–bin coder's best–case inefficiency has improved from 0.27% to 0.23%, when $x = 1.1$. This indicates that proper handling of the extreme probability ranges is essential, even in cases where the percentage of bits that fall into the extreme ranges is the same as those that fall into other ranges.

The bin coders in this study do not perform output interleaving. For future work we will be implementing a bin coder that does perform output interleaving, but at a coarse granularity (interleaving outputs megabytes at a time rather than a codeword at a time). We would also like to study further the interaction between the number and type of bins, and the coder's performance.

We think that a hybrid bin coder holds promise for high–performance computing applications. The methods used in the bins (Golomb and VV coding) are fast, simple mechanisms, and the indications are that on a common architecture its performance rivals that of state–of–the–art coders.

7 Acknowledgements

We are grateful to Peter Stubley for his advice and guidance. Leon Bottou and William Douglas Withers provided valuable assistance with the Z–Coder and ELS–Coder, respectively. Andrew Turpin provided valuable assistance with the Package–Merge algorithm. We are also grateful to the reviewers for their helpful suggestions and comments.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Joshua Senecal's work was supported in part by a United States Department of Education Government Assistance in Areas of National Need (DOE-GAANN) grant #P200A980307.

References

[1] Leon Bottou, Paul G. Howard, and Yoshua Bengio. The z–coder adaptive binary coder. In *Proceedings DCC '98 Data Compression Conference*, pages 13–22, 1998.

[2] Johann A. Briffa. Investigation of the error performance of tunstall coding. B.Eng.(Hons.) Final Year Dissertation, University of Malta, Faculty of Engineering, 1997.

[3] Francesco Fabris. Variable–length–to–variable–length source coding: A greedy step–by–step algorithm. *IEEE Transactions on Information Theory*, 38(5):1609–1617, 1992.

[4] G.H. Freeman. Divergence and the construction of variable–to–variable–length lossless codes by source–word extensions. In *DCC '93: Data Compression Conference*, pages 79–88, 1993.

[5] S. W. Golomb. Run–length encodings. *IEEE Trans. Inform. Theory*, IT-12:399–401, 1966.

[6] David A. Huffman. A method for the construction of minimum–redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[7] Lawrence Larmore and Daniel Hirschberg. A fast algorithm for optimal length–limited huffman codes. *Journal of the Association for Computing Machinery*, 37(3):464–473, Jul 1990.

[8] Mirin, Cohen, Curtis, Dannevik, Dimits, Duchaineau, Eliason, Schikore, Anserson, Porter, Woodward, Shieh, and White. Very high resolution simulation of compressible turbulence on the IBM–SP system. Technical Report UCRL–JC–134237, Lawrence Livermore National Laboratory, 1999.

[9] Claude Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[10] Peter R. Stubley. *Design and Analysis of Dual–Tree Entropy Codes*. PhD thesis, University of Waterloo, 1992.

[11] Peter R. Stubley. Adaptive variable–to–variable length codes. In *DCC '94: Data Compression Conference*, pages 98–105, 1994.

[12] Andrew Turpin and Alistair Moffat. Practical length–limited coding for large alphabets. In *Eighteenth Australasian Computer Science Conference*, volume 17, pages 523–532, 1995.

[13] Andrew Turpin and Alistair Moffat. Efficient implementation of the package–merge paradigm for generating length limited codes. In *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 187–195, Jan 1996.

[14] Wm. Douglas Withers. A rapid probabilty estimator and binary arithmetic coder. *IEEE Transactions on Information Theory*, 47(4):1533–1537, May 2001.

[15] Wm. Douglas Withers. Personal correspondence. 2003.

[16] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, Jun 1987.

Proceedings of the Data Compression Conference (DCC'04)
1068-0314/04 $ 20.00 © 2004 IEEE