Symbol-by-Symbol APP Decoding of the Golay Code and Iterative Decoding of Concatenated Golay Codes

Li Ping, *Member, IEEE*, and Kwan L. Yeung, *Member, IEEE*

***Abstract*— An efficient coset based symbol-by-symbol soft-in/soft-out APP decoding algorithm is presented for the Golay code. Its application in the iterative decoding of concatenated Golay codes is examined.**

***Index Terms*—Concatenated codes, coset decoding, Golay code, iterative decoding, turbo codes, turbo decoding.**

I. Introduction

The symbol-by-symbol soft-in/soft-out APP (*a posteriori* probability), also known as MAP (maximum *a posteriori*), decoding algorithm [1] plays an important role in the iterative decoding of concatenated codes [2]–[4]. The exact APP decoding of the Golay code C can be accomplished by applying the BCJR algorithm to the 256-state minimum conventional trellis of C [5] or the eigenvector algorithm [6] to the 16-state minimum tail-biting trellis of C [7]. The complexities of both methods are quite high. A lower cost alternative is the approximate method of [6] applied to the 16-state tail-biting trellis [7], resulting in complexity (normalized to operations per information bit) of about twice the BCJR algorithm for a rate $1/2$, 16-state conventional convolutional code.[1]

This correspondence presents an efficient, exact APP decoding algorithm for C based on the coset decoding principle [8]–[12]. The complexity of the algorithm is comparable to the BCJR algorithm for a rate $1/2$, 16-state conventional convolutional code. We will examine the application of the proposed algorithm in the iterative decoding of the concatenated Golay codes. We will show that, for short interleaver lengths, the concatenated Golay codes can achieve performance similar to or better than the turbo codes.

II. Preliminaries

In [8], Pless introduced an elegant 4×6 matrix construction of C. In this section, we will express this method in a product form of the hexacode and the SPC (single parity check) codes. We will then consider the APP decoding problem for C and establish a starting point for developing the new algorithm in the next section. For simplicity, we will express C over $\{+1, -1\}$. It is related to C over GF(2) by the standard mapping $\{+1 \leftrightarrow 0, -1 \leftrightarrow 1\}$.

A. Pless' Construction of C

Denote by $\mathbf{0}, \mathbf{1}, \omega, \overline{\omega}$ the four elements of GF(4), referred to as characters. The hexacode is a $[6, 3, 4]$ linear code over GF(4) with the generator matrix [8], [9]

$$\begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \omega & \overline{\omega} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \overline{\omega} & \omega \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix}. \tag{1}$$

Manuscript received May 4, 1998; revised May 18, 1999.

The authors are with the Department of Electronics Engineering, City University of Hong Kong, Hong Kong (e-mail: {eeliping; eekyeung}@cityu.edu.hk).

Communicated by A. M. Barg, Associate Editor for Coding Theory.
Publisher Item Identifier S 0018-9448(99)07313-7.

[1] This is estimated based on the example in [6], assuming that for a short length tail-biting trellis, the extra wrap length is roughly equal to the trellis length; also see Section III-D.

The following are two mappings from the characters to 4×1 vectors over $\{+1, -1\}$. The parity refers to the number of -1 in a column.

$$\begin{array}{cc} \text{even interpretations:} & \begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ \hline \mathbf{0} & \mathbf{1} & \omega & \overline{\omega} \end{array} \\ \text{odd interpretations:} & \begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ \hline \mathbf{0} & \mathbf{1} & \omega & \overline{\omega} \end{array} \end{array} \tag{2}$$

Applying the even (resp., odd) interpretations to the 64 codewords in the hexacode results in 64 4×6 binary arrays, collectively referred to as H^e (resp., H^o). Let P^e (resp., P^o) be the length-6 even and odd SPC codes over $\{+1, -1\}$ containing an even (resp., odd) number of -1, each with 32 codewords. Let

$$\boldsymbol{h} = \{h[i, j]\} = [\boldsymbol{h}_1, \boldsymbol{h}_2, \boldsymbol{h}_3, \boldsymbol{h}_4, \boldsymbol{h}_5, \boldsymbol{h}_6]$$

be a 4×6 array, where $\{\boldsymbol{h}_j\}$ are 4×1 vectors. Denote by "$\oplus$" the union of two nonoverlapping sets. It is straightforward to verify the equivalence between the construction in [8] and the following definition of C:

$$C = C^e \oplus C^o \tag{3a}$$
$$C^e \triangleq \{\boldsymbol{h} * \boldsymbol{p} \colon \boldsymbol{h} \in H^e, \boldsymbol{p} \in P^e\} \tag{3b}$$
$$C^o \triangleq \{\boldsymbol{h} * \boldsymbol{p} \colon \boldsymbol{h} \in H^o, \boldsymbol{p} \in P^o\} \tag{3c}$$
$$\boldsymbol{h} * \boldsymbol{p} \triangleq [\boldsymbol{h}_1 p_1, \boldsymbol{h}_2 p_2, \boldsymbol{h}_3 p_3, \boldsymbol{h}_4 p_4, \boldsymbol{h}_5 p_5, \boldsymbol{h}_6 p_6]. \tag{3d}$$

B. The APP Decoding Problem of C

Let the transmitted codeword be a 4×6 array $\boldsymbol{u} = \{u[i, j]\} \in C$ and its noisy observation be $\boldsymbol{x} = \{x[i, j]\}$. The output of a soft-in/soft-out APP decoder is [1], [2]

$$L[i, j] = \frac{1}{2} \log \left(\frac{\Pr\{u[i, j] = +1 | \boldsymbol{x}\}}{\Pr\{u[i, j] = -1 | \boldsymbol{x}\}} \right). \tag{4}$$

Here a factor of $1/2$ is included for convenience. Let $\boldsymbol{v} = \{v[i, j]\}$ be a 4×6 array of the bit confidence values conditioned on individual received symbol, i.e.,

$$v[i, j] = \frac{1}{2} \log \left(\frac{\Pr\{u[i, j] = +1 | x[i, j]\}}{\Pr\{u[i, j] = -1 | x[i, j]\}} \right). \tag{5}$$

For example, $v[i, j] = x[i, j]/\sigma^2$ for an additive white Gaussian noise (AWGN) channel of variance σ^2. Assuming that the entries of $\boldsymbol{v}$ are independent and all the codewords have equal probability of occurrence, then (4) can be evaluated as [3]

$$\begin{aligned} L[i, j] &= \frac{1}{2} \log \frac{\displaystyle\sum_{\substack{\boldsymbol{c} \in C \\ c[i, j] = +1}} e^{\langle \boldsymbol{c}, \boldsymbol{v} \rangle}}{\displaystyle\sum_{\substack{\boldsymbol{c} \in C \\ c[i, j] = -1}} e^{\langle \boldsymbol{c}, \boldsymbol{v} \rangle}} \\ &= \frac{1}{2} \log \frac{\displaystyle\sum_{\substack{\boldsymbol{c} \in C^e \\ c[i, j] = +1}} e^{\langle \boldsymbol{c}, \boldsymbol{v} \rangle} + \sum_{\substack{\boldsymbol{c} \in C^o \\ c[i, j] = +1}} e^{\langle \boldsymbol{c}, \boldsymbol{v} \rangle}}{\displaystyle\sum_{\substack{\boldsymbol{c} \in C^e \\ c[i, j] = -1}} e^{\langle \boldsymbol{c}, \boldsymbol{v} \rangle} + \sum_{\substack{\boldsymbol{c} \in C^o \\ c[i, j] = -1}} e^{\langle \boldsymbol{c}, \boldsymbol{v} \rangle}} \end{aligned} \tag{6}$$

where

$$\langle \boldsymbol{c}, \boldsymbol{v}\rangle = \sum_{j=1}^{6} \langle \boldsymbol{c}_j, \boldsymbol{v}_j\rangle = \sum_{j=1}^{6}\sum_{i=1}^{4} c[i, j]v[i, j]. \tag{7}$$

In (7), $\boldsymbol{c}_j$ and $\boldsymbol{v}_j$ are the jth columns of $\boldsymbol{c}$ and $\boldsymbol{v}$, respectively. Substitute (3) into (7)

$$\langle \boldsymbol{c}, \boldsymbol{v}\rangle = \langle \boldsymbol{h} * \boldsymbol{p}, \boldsymbol{v}\rangle = \sum_{j=1}^{6} \langle \boldsymbol{h}_j p_j, \boldsymbol{v}_j\rangle = \sum_{j=1}^{6} p_j \langle \boldsymbol{h}_j, \boldsymbol{v}_j\rangle = \langle \boldsymbol{p}, \boldsymbol{w}\rangle \tag{8}$$

with

$$\boldsymbol{w} = [w_1, \cdots, w_6] = [\langle \boldsymbol{h}_1, \boldsymbol{v}_1\rangle, \cdots, \langle \boldsymbol{h}_6, \boldsymbol{v}_6\rangle].$$

The summations in (6) then become

$$\sum_{\substack{\boldsymbol{c}\in C^x\\ c[i,j]=\pm 1}} e^{\langle \boldsymbol{c}, \boldsymbol{v}\rangle} = \sum_{\substack{\boldsymbol{c}\in C^x\\ c[i,j]=\pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle}, \qquad x = e, o. \tag{9}$$

Equation (9) is over 4096 codewords for every $c[i, j]$, which represents the bulk of the computation involved in (6). A straightforward summation is apparently very costly. We will explore an improved solution below.

III. Efficient APP Decoding Method for C

In this section we will first introduce an h-coset partitioning of C. We will show that (9) can be evaluated partially over each h-coset using a simple rule. The partial results can be efficiently combined through a set partitioning hierarchy of C. These form the core parts of the new algorithm.

A. Partition of C using h-Cosets

Fixing any $\boldsymbol{h}$ in H^e, we can obtain a unique subset of 32 codewords $\{\boldsymbol{h} * \boldsymbol{p}: \boldsymbol{p} \in P^e\}$ according to (3b). We will call them collectively as an h-coset generated by $\boldsymbol{h}$. For all the elements in H^e, we obtain 64 h-cosets. Similarly, another 64 h-cosets in the form of $\{\boldsymbol{h} * \boldsymbol{p}: \boldsymbol{p} \in P^o\}$, each containing 32 codewords, can be generated by $\boldsymbol{h} \in H^o$. It can be shown that these 128 h-cosets form a coset partition of C.[2] Notice that $\boldsymbol{h} * \boldsymbol{p} = \boldsymbol{h}$ only if $\boldsymbol{p} = [1, 1, 1, 1, 1, 1]$. Thus $\{\boldsymbol{c} = \boldsymbol{h} * \boldsymbol{p}: \boldsymbol{p} \in P^o\}$ does not include its generator $\boldsymbol{h}$ since $[1, 1, 1, 1, 1, 1] \notin P^o$. This distinguishes an h-coset generator from a coset leader.

Based on the above definition, (9) can be rewritten as (recall that $c[i, j] = h[i, j]p_j$, see (3))

$$\sum_{\substack{\boldsymbol{c}\in C^x\\ c[i,j]=\pm 1}} e^{\langle \boldsymbol{c}, \boldsymbol{v}\rangle} = \sum_{\boldsymbol{h}\in H^x} \sum_{\substack{\boldsymbol{p}\in P^x\\ p_j = c[i,j]/h[i,j]}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle}, \qquad x = e, o. \tag{10}$$

The inner summation above is over the 32 codewords in an h-coset and the outer one is over the 64 h-cosets for each parity. The following is a two-stage technique for evaluating (10).

B. Efficient Method for the Inner Summation in (10)

Recall that $\boldsymbol{p} \in P^e$ (resp., $\boldsymbol{p} \in P^o$) must contain an even (resp., odd) number of -1. It leads to a simple rule for evaluating the inner summation in (10)

$$\boldsymbol{h}\text{ even:}\qquad \sum_{\substack{\boldsymbol{p}\in P^e\\ p_j=\pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \frac{e^{\pm w_j}}{2}\left(\frac{A^+}{a_j^+} \pm \frac{A^-}{a_j^-}\right),\qquad j = 1, 2, \cdots, 6 \tag{11a}$$

$$\boldsymbol{h}\text{ odd:}\qquad \sum_{\substack{\boldsymbol{p}\in P^o\\ p_j=\pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \frac{e^{\pm w_j}}{2}\left(\frac{A^+}{a_j^+} \mp \frac{A^-}{a_j^-}\right),\qquad j = 1, 2, \cdots, 6 \tag{11b}$$

$$\text{with}\quad a_j^{\pm} = e^{+w_j} \pm e^{-w_j}\quad\text{and}\quad A^{\pm} = \prod_{j=1}^{6} a_j^{\pm}. \tag{11c}$$

Equations (11a) and (11b) are equivalent to the APP decodings for the SPC codes P^e and P^o, respectively. They can be verified by expanding $A^{\pm}$ and canceling out redundant terms.

C. Efficient Method for the Outer Summation in (10)

After completing (11) for 128 h-cosets, the results can be substituted into the outer summation in (10). A straightforward method is to sum over all the h-cosets for every output bit. In this way, some of the partial summations involve common h-cosets, which are duplicated for different output bits. The following technique can be used to avoid such unnecessary operations.

Denote by $J(\boldsymbol{h}_j)$ all the h-cosets whose generators have the jth column fixed by $\boldsymbol{h}_j$. Similarly, denote by $J(\boldsymbol{h}_j, \boldsymbol{h}_{j'})$ and $J(\boldsymbol{h}_j, \boldsymbol{h}_{j'}, \boldsymbol{h}_{j''})$ the collection of h-cosets with two and three columns fixed, respectively (with j, j', and j'' different). Then C^e and C^o can be partitioned progressively as shown below and (10) can be evaluated accordingly.

- For j fixed, C^e can be partitioned into four nonoverlapping subsets $\{J(\boldsymbol{h}_j): \boldsymbol{h}_j = \boldsymbol{0}, \boldsymbol{1}, \omega, \overline{\omega}\}$ using even interpretations. This is also true for C^o using odd interpretations. Thus (10) can be rewritten as

$$\sum_{\boldsymbol{h}\in H^x} \sum_{\substack{\boldsymbol{p}\in P^x\\ p_j = c[i,j]/h[i,j]}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \sum_{4\text{ choices of }\boldsymbol{h}_j} \left\{ \sum_{J(\boldsymbol{h}_j)} \sum_{\substack{\boldsymbol{p}\in P^x\\ p_j = c[i,j]/h[i,j]}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} \right\}, \quad x = e, o. \tag{12}$$

Consider the braced part in (12) with $c[i, j]$ and $\boldsymbol{h}_j$ fixed. Then p_j, $e^{\pm w_j}$ and $a_j^{\pm}$ (see (11)) are also fixed. Substituting (11) into the braced part in (12) we have

$$\boldsymbol{h}_j\text{ even:}\ \sum_{J(\boldsymbol{h}_j)} \sum_{\substack{p\in P^e\\ p_j=\pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \frac{e^{\pm j w_j}}{2}\left(\frac{1}{a_j^+}\sum_{J(\boldsymbol{h}_j)} A^+ \pm \frac{1}{a_j^-}\sum_{J(\boldsymbol{h}_j)} A^-\right) \tag{13a}$$

$$\boldsymbol{h}_j\text{ odd:}\ \sum_{J(\boldsymbol{h}_j)} \sum_{\substack{p\in P^e\\ p_j=\pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \frac{e^{\pm j w_j}}{2}\left(\frac{1}{a_j^+}\sum_{J(\boldsymbol{h}_j)} A^+ \mp \frac{1}{a_j^-}\sum_{J(\boldsymbol{h}_j)} A^-\right). \tag{13b}$$

Furthermore, for a fixed $j' (\neq j)$, $J(\boldsymbol{h}_j)$ can also be partitioned into four nonoverlapping subsets $\{J(\boldsymbol{h}_j, \boldsymbol{h}_{j'}): \boldsymbol{h}_{j'} = \boldsymbol{0}, \boldsymbol{1}, \omega, \overline{\omega}\}$, and so

$$\sum_{J(\boldsymbol{h}_j)} A^{\pm} = \sum_{4\text{ choices of }\boldsymbol{h}_{j'}} \sum_{J(\boldsymbol{h}_j, \boldsymbol{h}_{j'})} A^{\pm}. \tag{13c}$$

[2] Let $\hat{\boldsymbol{h}}$ be the all-one 4×6 array. Then $\hat{C} = \{\hat{\boldsymbol{h}} * \boldsymbol{p}: \boldsymbol{p} \in P^e\}$ is a subcode in C. Denoted by $\boldsymbol{a} \circ \boldsymbol{b}$ the bit-by-bit multiplication between two arrays $\boldsymbol{a}$ and $\boldsymbol{b}$ over $\{+1, -1\}$, which is equivalent to $\boldsymbol{a} + \boldsymbol{b}$ over GF [2]. Fixing any $\boldsymbol{c} = \boldsymbol{h} * \boldsymbol{p}'$ (with $\boldsymbol{h}$ and $\boldsymbol{p}'$ having the same parity), then $\{\boldsymbol{c} \circ \hat{\boldsymbol{c}}: \hat{\boldsymbol{c}} \in \hat{C}\}$ is a coset of $\hat{C}$ led by $\boldsymbol{c}$. This coset is homomorphic to the h-coset generated by $\boldsymbol{h}$ since for any $\boldsymbol{p} \in P^e$

$$\boldsymbol{c} \circ \hat{\boldsymbol{c}} = (\boldsymbol{h} * \boldsymbol{p}') \circ (\hat{\boldsymbol{h}} * \boldsymbol{p}) = (\boldsymbol{h} \circ \hat{\boldsymbol{h}}) * (\boldsymbol{p}' \circ \boldsymbol{p}) = \boldsymbol{h} * (\boldsymbol{p}' \circ \boldsymbol{p})$$

and $\boldsymbol{p}' \circ \boldsymbol{p}$ has the same parity as $\boldsymbol{p}'$ (and so $\boldsymbol{h}$).

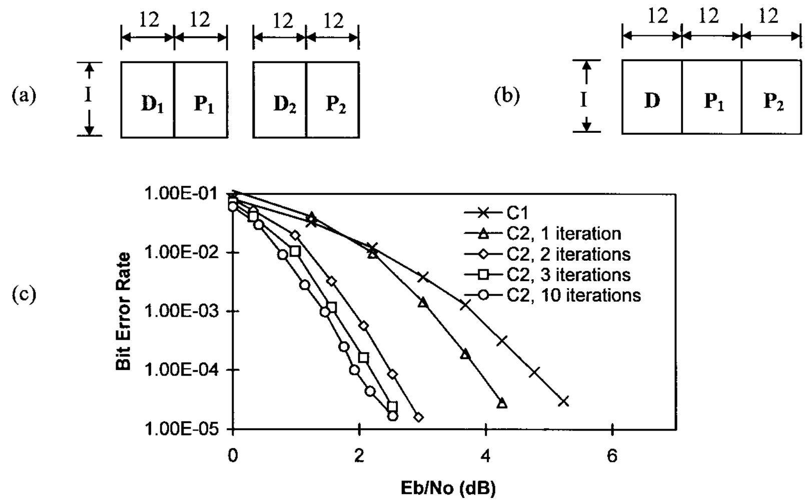

Fig. 1. Concatenation Golay codes. (a) The encoding process. (b) The overall codeword. (c) Simulation results. C1 is for the standard $[24, 12, 8]$ Golay code and C2 for the rate $1/3$ concatenated Golay codes with $I = 12$ $(L = 144)$.

For a fixed $j''(\neq j, j')$, $J(\boldsymbol{h}_j, \boldsymbol{h}_{j'})$ can again be partitioned into four nonoverlapping subsets $\{J(\boldsymbol{h}_j, \boldsymbol{h}_{j'}, \boldsymbol{h}_{j''})\colon \boldsymbol{h}_{j''} = \boldsymbol{0}, \boldsymbol{1}, \omega, \overline{\omega}\}$. Since the hexacode code is MDS (maximum-distance separable, see [13]) with minimum distance $d = 4$, any three columns, together with the parity, specify a unique $\boldsymbol{h}$ and so $J(\boldsymbol{h}_j, \boldsymbol{h}_{j'}, \boldsymbol{h}_{j''})$ represents a unique h-coset. As different h-cosets cannot overlap, this partition of $J(\boldsymbol{h}_j, \boldsymbol{h}_{j'})$ is invariant with respect to the choices of j''. Thus the inner summation in (13c) can be uniquely expressed as

$$\sum_{J(\boldsymbol{h}_j, \boldsymbol{h}_{j'})} A^{\pm} = \sum_{4h-\text{cosets}} A^{\pm}. \tag{14}$$

The key to efficiency improvement is avoiding the duplication of the summations in (13) and (14) for different output bits. Some details in evaluating (12)–(14) are given below.

i) We limit the (j, j') pair in (14) to $(j, j+1)$, $j = 1, 3, 5$, referred to as blocks [9]. We first evaluate (14) for 16 (character pairs) $\times$ 3 (blocks) $\times$ 2 (parities) = 96 possibilities of $(\boldsymbol{h}_j, \boldsymbol{h}_{j+1})$ pair. Each result of (14) is then used twice in (13c), where we set $j' = j+1$ for $j = 1, 3, 5$ and $j' = j-1$ for $j = 2, 4, 6$.

ii) We evaluate (13) for 6 (columns) $\times$ 4 (characters) $\times$ 2 (parities) = 48 possibilities of $\boldsymbol{h}_j$. Each result of (13) can be repeatedly used in (12) for up to three information bits in a column. The information positions for C defined in (3) can be chosen as $(i, j) = (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), (4, 1), (2, 2), (4, 2), (2, 3)$, and $(4, 3)$.

The above discussions are summarized below, with costs listed in brackets.

D. APP Decoding Algorithm for C

Preparation:

i) Generate 96 possibilities of $\{e^{\pm w_j/2}\colon j = 1, 2, \cdots, 6\}$, see the Appendix (24 *exponentials* and 66 *multiplications*).

ii) Generate 96 possibilities of $\{a_j^{\pm}\colon\ j = 1, 2, \cdots, 6\}$ (96 *additions*).

iii) Generate 256 possibilities of $\{A^{\pm}\}$ for 128 $\boldsymbol{h}$-cosets. This can be done by first generating all the partial products $a_j^{+} a_{j+1}^{+}$ and $a_j^{-} a_{j+1}^{-}$ for $j = 1, 3, 5$ and then multiplying three partial products together for every $\boldsymbol{h}$ $(4 \times 4 \times 3 \times 2 \times 2 + 128 \times 2 \times 2 = 704$ *multiplications*).

Step 1. Evaluate (14) for 96 possibilities of $(\boldsymbol{h}_j, \boldsymbol{h}_{j+1})$ pair, with $j = 1, 3, 5$ $(96 \times 2 \times 3 = 576$ *additions*).

Step 2. Evaluate (13), with $j = 1, 2, \cdots, 6$ and $j' = j+1$ if $j = 1, 3, 5$ and $j' = j-1$ otherwise, for 48 possibilities of $\boldsymbol{h}_j$ $(2 \times 3 \times 48 + 2 \times 48 = 384$ *additions* and $4 \times 48 = 192$ *multiplications, divide-by-*2 *ignored*).

Step 3. For 12 information bits, evaluate (12) and then complete (6) $(2 \times 2 \times 3 \times 12 + 2 \times 12 = 168$ *additions,* 12 *divisions* and 12 *logarithms*).

The total cost of the above algorithm is 974 multiplications, 1224 additions, 24 exponentials, and 12 logarithms (divisions counted as multiplications). The normalized cost is about 81 multiplications, 102 additions, 2 exponentials, and 1 logarithms per information bit.

For the purpose of comparison, the BCJR algorithm applied to a rate $1/2$, N-state conventional trellis (non-tail-biting) requires about $6N$ multiplications, $4N$ additions, 2 exponentials, and 1 logarithm per information bit. The complexity of the proposed algorithm is thus approximately comparable to the BCJR algorithm applied to a rate $1/2$, 16-state conventional trellis (about 96 multiplications and 64 additions per information bit).

It is interesting to note that the minimum trellis of C has 16-state [5], [7], [12]. However, this can only be achieved by the tail-biting trellis, to which the standard BCJR algorithm is not directly applicable. To the best of our knowledge, the most efficient APP algorithm for the tail-biting trellis is the approximate algorithm of [6, Algorithm A3]. For the same state number, the complexity of [6, Algorithm A3] is higher than the standard BCJR algorithm by a factor of $(W + L)/L$, where W is the wrapping length and L the trellis length. For short L, we assume $W \approx L$ (as adopted in the example in [6]). Applying the above to the 16-state tail-biting trellis of C developed in [7], the resultant cost is about twice of the proposed algorithm. Other alternatives, such as the eigenvector method in [6] (which is an exact method) applied to the 16-state tail-biting trellis of C or the BCJR algorithm applied to the minimal 256-state conventional trellis of C, appear much more costly.

IV. Applications

Consider the application of the proposed algorithm to the concatenated Golay codes in Fig. 1. Let $\boldsymbol{D}$ be an $I \times 12$ binary information array. Let $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ be two $I \times 12$ arrays obtained from $\boldsymbol{D}$ using appropriate interleavings. Apply the systematic $[24, 12, 8]$ Golay code row-wise to $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$, producing two $I \times 12$ parity-check

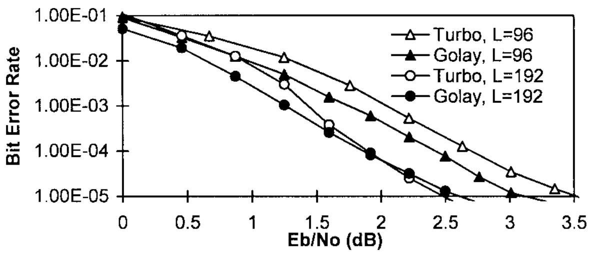

Fig. 2. Performance comparisons of the concatenated Golay codes and turbo codes. Rate $= 1/3$. Iteration number $= 10$.

arrays, $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$. The overall codeword is formed by $\boldsymbol{DP}_1\boldsymbol{P}_2$. The coding rate is $1/3$ and the interleaving length $L = I \times 12$.

The decoding procedure follows the turbo-type iterative decoding technique [2], [3], incorporating the algorithm proposed above. The simulated performances for $L = 144$ are shown in Fig. 1(c). The performance of the standard $[24, 12, 8]$ Golay code with maximum-likelihood (ML) decoder is included as reference. The interleavers used are $D_1[i, j] = D[i, j]$ and $D_2[i, j] = D[j, i]$. It is seen that most coding gain can be achieved with three iterations.

In some communication systems, interleaver lengths are restricted by delay constraints [4]. The performance of the concatenated Golay codes and turbo codes are compared in Fig. 2 for short interleaver lengths of $L = 96$ and $L = 192$. The interleavers for the Golay based codes are

$$D_1[i, j] = D[i, j] \qquad D_2[i, j] = D[(i + j) \bmod I, j + 5], \quad I = 8, 16 \text{ for } L = 96, 192, \text{ respectively.} \tag{15}$$

The turbo codes are generated by polynomials 37 (denominator) and 21 (numerator) [2], with block interleavers. The turbo code encoders are all terminated [14] and thus their actual rates are slightly less than $1/3$. It is seen that for $L = 96$, the performance of the concatenated Golay codes is better than that of the turbo code. For $L = 192$, the performances of the two methods are very close. Only marginal improvement has been observed for the Golay code-based schemes for $L > 200$.

Based on the discussion at the end of Section III-D, the decoding costs of the concatenated Golay codes and 16-state turbo codes in Fig. 2 are roughly comparable in terms of operation numbers per information bit. For the comparison of storage usage, the BCJR algorithm for the rate $1/2$, 16-state convolutional code [1], [2] needs to store $16L$ real values during the forward recursion (e.g., for $L = 192$, $16L = 3072$). On the other hand, the algorithm in Section III-D needs to store only 448 real values of $\{e^{\pm w_j/2}, a^{\pm}, A^{\pm}\}$ plus some extra buffering space (at most 96 real values).

V. Conclusion

The trellis-based BCJR APP decoding algorithm [1] is most suitable for convolutional codes [2] or block codes with simple trellis structures [3]. When applied to block codes, the trellis-based approach may not be the best choice. This is due to the fact that the minimum trellises of block codes can be tail-biting [5, 7, 15], which are more difficult to decode than conventional trellises. Alternative methods have been explored [16], [17] for some block codes. This correspondence has shown that the coset-based technique provides a cost-effective approach to the APP decoding of the Golay code.

Appendix

All the 48 possible values of $\{w_j = \langle \boldsymbol{h}_j, \boldsymbol{v}_j\rangle\}$ can be computed using the Gray code technique of [9] with 60 additions and then $e^{\pm w_j}$ can be calculated with 96 exponentials. The following is an alternative approach that is used for the complexity analysis in Section III-D. Define $t(x) = x$ if $x \geq 0$ and $t(x) = 0$ if $x < 0$ and let $T_j = \sum_{i=1}^{4} t(v[i, j])$. Then (see (8))

$$E_j^{\pm} \triangleq e^{\pm w_j - T_j} = e^{\pm\langle \boldsymbol{h}_j, \boldsymbol{v}_j\rangle - T_j} = \exp\left(\sum_{i=1}^{4}(\pm h[i, j]v[i, j] - t(v[i, j]))\right). \tag{16}$$

Since $h[i, j] \in \{+1, -1\}$, $\pm h[i, j]v[i, j] - t(v[i, j])$ results in either 0 or $-2|v[i, j]|$. Fixing j, all the 16 possible values of $E_j^{\pm}$ can be found as

$$\begin{aligned}
&1,\ e^{-2|v[1,j]|},\ e^{-2|v[2,j]|},\ e^{-2|v[3,j]|},\ e^{-2|v[4,j]|},\\
&e^{-2(|v[1,j]|+|v[2,j]|)},\ e^{-2(|v[1,j]|+|v[3,j]|)},\\
&e^{-2(|v[1,j]|+|v[4,j]|)},\ e^{-2(|v[2,j]|+|v[3,j]|)},\\
&e^{-2(|v[2,j]|+|v[4,j]|)},\ e^{-2(|v[3,j]|+|v[4,j]|)},\\
&e^{-2(|v[1,j]|+|v[2,j]|+|v[3,j]|)},\ e^{-2(|v[1,j]|+|v[2,j]|+|v[4,j]|)},\\
&e^{-2(|v[1,j]|+|v[3,j]|+|v[4,j]|)},\ e^{-2(|v[2,j]|+|v[3,j]|+|v[4,j]|)},\\
&e^{-2(|v[1,j]|+|v[2,j]|+|v[3,j]|+|v[4,j]|)}
\end{aligned} \tag{17}$$

which can be generated with 4 exponentials and 11 multiplications (multiply-by-2 ignored). For $j = 1, 2, \cdots, 6$, this amounts to 24 exponentials and 66 multiplications. Instead of calculating $\{e^{\pm w_j}\}$ from $\{E_j^{\pm}\}$, we can also simply replace $e^{\pm w_j}$ in (11) by $E_j^{\pm}$. This will not affect the final result of the algorithm in Section III-D since (11) is equivalent to (with $T = T_1 + T_2 + \cdots + T_6$)

$$\text{for } \boldsymbol{h} \text{ even: } e^{-T} \sum_{\substack{\boldsymbol{p} \in P^e \\ p_j = \pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \frac{E_j^{\pm}}{2}\left(\frac{\prod_{k=1}^{6}(E_k^{+} + E_k^{-})}{E_j^{+} + E_j^{-}} \pm \frac{\prod_{k=1}^{6}(E_k^{+} + E_k^{-})}{E_j^{+} - E_j^{-}}\right), \quad j = 1, 2, \cdots, 6 \tag{18a}$$

$$\text{for } \boldsymbol{h} \text{ odd: } e^{-T} \sum_{\substack{\boldsymbol{p} \in P^o \\ p_j = \pm 1}} e^{\langle \boldsymbol{p}, \boldsymbol{w}\rangle} = \frac{E_j^{\pm}}{2}\left(\frac{\prod_{k=1}^{6}(E_k^{+} + E_k^{-})}{E_j^{+} + E_j^{-}} \mp \frac{\prod_{k=1}^{6}(E_k^{+} + E_k^{-})}{E_j^{+} - E_j^{-}}\right), \quad j = 1, 2, \cdots, 6. \tag{18b}$$

The extra factor e^{-T} above will be canceled out during the division in (6).

Acknowledgment

The authors are very grateful for the detailed comments from the anonymous reviewers.

References

[1] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 284–287, Mar. 1974.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding turbo-codes," in *Proc. IEEE ICC-93*, May 1993, pp. 1064–1070.

[3] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 429–445, Mar. 1996.

[4] P. Jung, "Comparison of turbo-code decoders applied to short frame transmission systems," *IEEE J. Select. Areas Commun.*, vol. 14, pp. 530–537, Apr. 1996.

[5] D. J. Muder, "Minimal trellises for block codes," *IEEE Trans. Inform. Theory*, vol. 34, pp. 1049–1053, Sept. 1988.

[6] J. B. Anderson and S. M. Hladik, "tail-biting MAP decoders," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 297–302, Feb. 1998.

[7] A. R. Calderbank, G. D. Forney, Jr., and A. Vardy, "Minimal tail-biting trellises: The Golay code and more," *IEEE Trans. Inform. Theory*, vol. 45, pp. 1435–1455, July 1999.

[8] V. Pless, "Decoding the Golay codes," *IEEE Trans. Inform. Theory*, vol. IT-32, pp. 561–576, July 1986.

[9] A. Vardy and Y. Be'ery, "More efficient soft decoding of the Golay codes," *IEEE Trans. Inform. Theory*, vol. 37, pp. 667–672, May 1991.

[10] J. H. Conway and N. J. A. Sloane, "Soft decoding techniques for codes and lattices, including the Golay code and the Leech lattice," *IEEE Trans. Inform. Theory*, vol. IT-32, pp. 41–50, Jan. 1986.

[11] J. Snyders and Y. Be'ery, "Maximum likelihood soft decoding of binary block codes and decoders for the Golay codes," *IEEE Trans. Inform. Theory*, vol. 35, pp. 963–975, Sept. 1989.

[12] O. Amrani, Y. Be'ery, A. Vardy, F. W. Sun, and C. A. van Tilborg, "The Leech lattice and Golay code: Bounded distance decoding and multilevel construction," *IEEE Trans. Inform. Theory*, vol. 40, pp. 1030–1043, July 1994.

[13] F. M. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, vols. I/II. Amsterdam, The Netherlands: North-Holland, 1977.

[14] P. Robertson, "Illuminating the structure of code and decoder of parallel concatenated recursive systematic (turbo) codes," in *Proc. GLOBECOM'94*, Dec. 1994, pp. 1298–1303.

[15] N. Wiberg, H. A. Loeliger, and R. Kotter, "Codes and iterative decoding on general graphs," *Euro. Trans. Telecommun.*, vol. 6, pp. 513–526, Sept. 1995.

[16] Li Ping, S. Chan, and K. L. Yeung, "Max-log-MAP filtering algorithm for decoding product F_{24} code," in *Proc. IEEE 1997 Int. Conf. Communications*, June 1997, pp. 1366–1370.

[17] Li Ping and S. Chan, "Iterative decoding of concatenated Hadamard codes," in *Proc. IEEE 1998 Int. Conf. Communications*, June 1998, pp. 136–140.

Optimal and Near-Optimal Encoders for Short and Moderate-Length Tail-Biting Trellises

Per Ståhl, *Student Member, IEEE*, John B. Anderson, *Fellow, IEEE*, and Rolf Johannesson, *Fellow, IEEE*

***Abstract*—The results of an extensive search for short and moderate-length polynomial convolutional encoders for time-invariant tail-biting representations of block codes at rates $R = 1/4, 1/3, 1/2,$ and $2/3$ are reported. The tail-biting representations found are typically as good as the best known block codes.**

***Index Terms*—Convolutional codes, error-correction coding, tail-biting encoders, trellis codes.**

I. INTRODUCTION

There exist two main principles to terminate convolutional codes into block codes. Assume for simplicity that the generator matrix for

Manuscript received February 1, 1998; revised February 19, 1999. This work was supported in part by the Swedish Research Council for Engineering Sciences under the "Visiting Professors Programme" 1996/97, and in part by the Foundation for Strategic Research–Personal Computing and Communication under Grant PCC-9706–09.

The authors are with the Department of Information Technology, Information Theory Group, Lund University, S-221 00 Lund, Sweden.

Communicated by F. R. Kschischang, Associate Editor for Coding Theory.
Publisher Item Identifier S 0018-9448(99)07009-1.

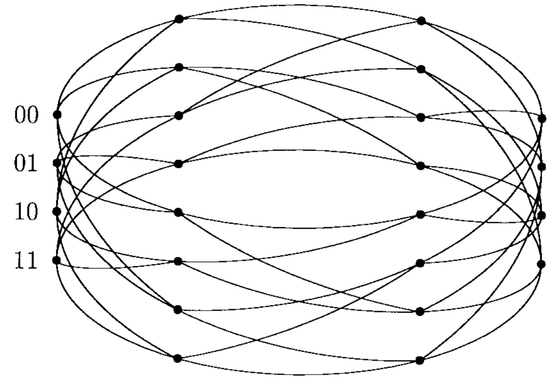

Fig. 1. Circular trellis of length $L = 6$ for a four-state convolutional encoder.

a rate $R = b/c$ convolutional code of memory m is polynomial and realized in controller canonical form.

In the first method we start in the zero state and encode the K information symbols followed by m b-tuples of zeros. Hence, we reach the zero state and the convolutional code has been *terminated* into a block code by the so-called *zero-tail* (ZT) method at the cost of a rate loss by a factor $K/(K + mb)$. If the trellis is short this rate loss might not be acceptable.

A termination method that does not suffer from any rate loss is *tail-biting*, which can be used to construct very powerful regular trellis representations of block codes. Assuming a trellis of L sections (for simplicity we assume here that $m \leq L$), the *tail-biting condition* is the restriction that the convolutional encoder state $\boldsymbol{\sigma}$ at time $t = 0$ is equal to the encoder state at time L, i.e., $\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}_L$. A tail-biting trellis of length L corresponds to a total of $K = bL$ information symbols, c symbols per branch, block length $N = Lc$, 2^b branches per trellis node; the number of codewords is

$$M = 2^K = 2^{bL} \tag{1}$$

and its rate is

$$R = K/N = b/c. \tag{2}$$

Let $\boldsymbol{u}_{[0,L)} = \boldsymbol{u}_0\boldsymbol{u}_1 \cdots \boldsymbol{u}_{L-1}$ denote the input (information) sequence, $\boldsymbol{v}_{[0,L)} = \boldsymbol{v}_0\boldsymbol{v}_1 \cdots \boldsymbol{v}_{L-1}$ the output sequence (codeword), and

$$G(D) = G_0 + G_1 D + \cdots + G_m D^m \tag{3}$$

the generator matrix. The codewords of the tail-biting representation of the block code $\mathcal{B}_{\rm tb}$ that is obtained from the convolutional code $\mathcal{C}$ encoded by the generator matrix $G(D)$ can be compactly written as

$$\boldsymbol{v}_{[0,L)} = \boldsymbol{u}_{[0,L)} \boldsymbol{G}_L^{\rm tb} \tag{4}$$

where

$$\boldsymbol{G}_L^{\rm tb} = \begin{pmatrix} G_0 & G_1 & \cdots & & G_m & & & \\ & G_0 & G_1 & \cdots & & G_m & & \\ & & \ddots & \ddots & & & \ddots & \\ & & & G_0 & G_1 & \cdots & & G_m \\ G_m & & & & G_0 & G_1 & \cdots & G_{m-1} \\ G_{m-1} & G_m & & & & \ddots & \ddots & \vdots \\ \vdots & & \ddots & & & & \ddots & G_1 \\ G_1 & G_2 & \cdots & G_m & & & & G_0 \end{pmatrix} \tag{5}$$