Parallel Decoding of Trellis Stages for Low Latency Decoding of Tail-Biting Convolutional Codes

Amaael Antonini*, Wenhui Sui,*, Zihan Qu,*, and Richard D. Wesel*
*Department of Electrical and Computer Engineering, University of California, Los Angeles, CA 90095, USA
Email: {amaael, wenhui.sui, brucequ, wesel}@ucla.edu

***Abstract*—Convolutional codes are widely used in many applications. The encoders can be implemented with a simple circuit. Decoding is often accomplished by the Viterbi algorithm or the maximum a-posteriori decoder of Bahl et al. These algorithms are sequential in nature, requiring a decoding time proportional to the message length. For low latency applications this this latency might be problematic. This paper introduces a low latency decoder for tail-biting convolutional codes TBCCs that processes multiple trellis stages in parallel. The new decoder is designed for hardware with parallel processing capabilities. The overall decoding latency is proportional to the log of the message length. The new decoding architecture is modified into a list decoder, and the list decoding performance can be enhanced by exploiting linearity to expand the search space. Certain modifications to standard TBCCs are supported by the new architecture and improve frame error rate performance.**

***Index Terms*—Channel codes, convolutional codes, maximum likelihood decoder, list decoding.**

I. Introduction

We present a low latency Trellis decoder architecture for tail-biting convolutional codes (TBCC) that works well on parallel processing hardware like graphic processing units (GPU's). The design exploits the structure of trellis decoders to maximize the number of simultaneous operations.

Convolutional codes are typically decoded via the Viterbi algorithm [1], [2] that minimizes the frame error rate (FER), or the algorithm introduced by Bahl, Cocke, Jelinek, and Raviv [3] (BCJR), that minimizes the symbol error rate. Low delay requirements have been a limiting factor for Viterbi decoders. Several works have achieved a level of parallelism to address this problem. Fettweis and Meyr [4] combined every P stages of an K-stage trellis using a smaller trellis per each of the M states, and achieved a linear speedup of P. Mohammadidoost and Hashemi [5] optimized the memory access for both the surviving path and the trace-back path on a GPU. Peng *et al.* [6] introduce a method to achieve a high throughput also using GPU. Their method is flexible enough for application on general block codes and is shown to attain a throughput of up to 1.8 Gb/s on convolutional codes. Zhihui *et al*. [7] and Hanif *et al.* [8] show that part of the computations can be done independently. In [7] a sequence is split into smaller segments and the processing of each segment is further split into an independent part and a dependent part. A matrix representation of the process is used in [8] that allows a GPU to exploit the independence of some operations. Seshadri and Sundberg [9] compare convolutional codes with LDPC coders under delay constraints and show that convolutional codes outperform LDPC in the more restrictive regime. In [10] Rachigner *et al.* implement a List Viterbi Decoder with parallel processing to increase throughput. An FPGA version of a Viterbi decoder is also provided by Ben Asher *et al.* in [11], with parallelism that can be tuned via a parameter P. The method provides a linear speedup similar to that in [4], with a decoding time proportional to $\frac{N}{P} + P$. List decoding has also been widely studied. Elias [12] developed an (n, e, L) list decoder that corrects all sets of e or fewer errors. Soong and Huang [13] proposed a fast trellis search method to obtain a list of the best L hypotheses in speech recognition. Seshadri and Sundberg [9] also developed a list Viterbi decoder capable to find the best L decoding estimates and combined it with an error-detecting code to obtain significant performance improvements. Many works have studied methods to reduce the complexity of list decoders, and some can be found in [14]–[18].

This research is supported by National Science Foundation (NSF) grant CCF-2008918. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect views of NSF.

A. Contributions

We implement very low latency Viterbi decoder for Tail biting convolutional codes. Our decoder leverages the structure of the code trellis to performs operations on the entire length of the trellis simultaneously. The trellis length is reduced by half at each step until a single stage trellis is obtained. The number of steps required is proportional to the $\log_2$ of the original trellis size, resulting in a significant decoding speed-up. The cost of the lower delay is a significant increase in the number of operations, which sets a constraint on the codes that can be implemented with a given hardware. Part of the cost in additional operations could be offset by the ease of managing simultaneous operations that allows a more efficient use of vector processors and GPUs. The main contributions of this work are as follows:

- This work demonstrates a low delay trellis decoder implementation for TBCC's codes, suitable for GPU's or other parallel processing systems.
- We propose two list decoder versions of the parallel trellis stage decoder that produces a fixed size list with no additional delay for the first version and a significant performance increase with a small delay penalty for the second version. The improved performance of the list

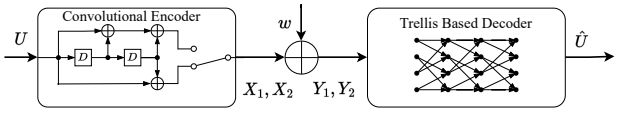

Fig. 1. System model describing a convolutional code, the channel and a trellis based decoder.

decoders could allow to achieve a target FER with a simpler code and low delay.

- We also propose the use of a block code different from a convolutional code to further improve the performance without changing the parallel trellis stage decoder.

B. Organization

The rest of the paper proceeds as follows: in Sec. II we discuss the encoder, channel and decoder model. In Sec. III we introduce our low latency parallel trellis stage decoder and two list decoder versions with similar latency, using the same architecture. In Sec. IV we show simulation results of our decoders and in Sec. V we conclude the paper.

II. System Model

The model consists of an encoder that receives a K-symbols information sequence $\mathbf{U}$ and transmits an N-symbols sequence $\mathbf{X}$; a noisy channel that adds a noise component W_i to each symbol X_i to produce a channel sequence $\mathbf{Y}$, where $Y_i = X_i + W_i$, $i = 0, 1, \ldots, N-1$; and a decoder that receives the channel sequence $\mathbf{Y}$ and produces an estimate $\hat{\mathbf{U}}$ of the information sequence, as describes in Fig. 1. We restrict our focus to tail-biting convolutional codes. The encoder for these codes are fully described by a digital circuit that implement a finite-state machine with k inputs, n outputs, ν memory elements and M states. The number of states M is given by the number of possible values in the circuit's memory. To obtain k inputs, the information sequence $\mathbf{U} = U_0, U_1, \ldots, U_{K-1}$ is split into k streams and n-output streams are produced. The code rate is given by k/n, the number of input streams divided by the number of output streams. In the rest of the paper we only consider binary codes with input and output symbols in the binary field $GF(2)$, where $U_i \in \{0, 1\}$ and $M = 2^\nu$ and with a single input stream, $k = 1$. We use binary phase shift keying modulation (BPSK) of the code output, that is $X_i \in \{-1, 1\}$ for $\mathbf{X} = X_1, X_2, \ldots X_N$, where $N = K\frac{n}{k}$. The channel used in our simulations is the additive white Gaussian noise (AWGN) channel, where $W_i \sim \mathcal{N}(0, \sigma^2)$ for some noise variance σ^2 computed from the signal to noise ratio (SNR).

A. Decoding Latency Problem

The problem we address in this paper is to design a maximum likelihood (ML) decoder for TBCC with decoding latency of order $O(\log(K))$, where K is the length of the information sequence $\mathbf{U}$. By latency we mean the time taken by the decoder to produce an estimate $\hat{\mathbf{U}}$ from the time when the full received sequence $\mathbf{Y}$ becomes available, which we denote by T. Let $f_{y|x}(\mathbf{Y} \mid \mathbf{X})$ be the likelihood function of $\mathbf{Y}$, the p.d.f. (or p.m.f.) of $\mathbf{Y}$ given an input sequence $\mathbf{X}$. The problem can be expressed by:

$$\text{find} \qquad \hat{\mathbf{U}} = \underset{\mathbf{x}}{\operatorname{argmax}} f_{y|x}(\mathbf{Y} \mid \mathbf{x}) \tag{1}$$

$$\text{subject to} \qquad T \propto O(\log(K)) \tag{2}$$

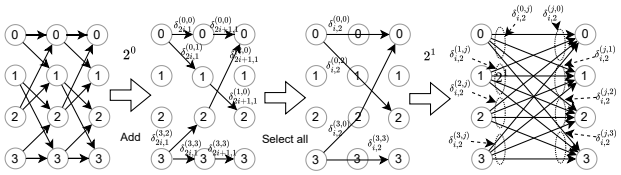

Fig. 2. Construction of a 2-step, fully connected, trellis stage i with four states, from two initial 1-step, not fully-connected, trellis stages $2i, 2i+1$.

III. The Parallel Trellis Stage Decoder (PTS)

We implement a trellis based decoder for convolutional codes with a high level of parallelism. Fettweis and Meyer [4] and Ben Asher *et al.* pre-processed M stage trellis segments first, and used the Viterbi Algorithm on a new trellis, called an M-step trellis in [4], rather than on the original K-stage trellis, which was called a 1-step trellis in [4]. We borrow the notation of [4] to define a 2^m-step survivor trellis which is an M-step trellis as defined in [4], with no parallel branches and where $M = 2^m$. Our design consists of recursively reducing a pair of 2^m-step survivor trellis stages into a single stage of a 2^{m+1}-step trellis. The latency in [4] and [11], when the next M-step stage is constructed in advance, is a function of $N/M + M$, which is linear in N for fixed M. Furthermore, the pre-processing method to obtain an M-state trellis segment in [4] takes linear time, and the method in [11], grows in complexity exponentially with M, via $O(2^M)$. In both cases the largest M that can be effectively used is limited to small values. Note that a ν-step trellis becomes fully connected when the 1-step trellis is binary, see Fig. 2, where 2^ν is the number of states. Furthermore, there will be 2^ν paths connecting every start and end state of a 1-step trellis segment 2ν stages long, see Fig. 3. Instead of the linear speedup in [4], [11], we target a logarithmic decoding time, with a complexity equivalent to that in [11] when $M = \nu$.

A. Parallel Trellis Stages Decoder Architecture

Our parallel decoder attains logarithmic decoding time by recursively combining every two stages of a 2^m-step survivor trellis into a 2^{m+1} step survivor trellis until a single stage remains. The speedup derives from the ability to construct all the stages of the 2^{m+1}-step trellis simultaneously. The starting trellis could be a 1-step trellis, as shown in Fig. 2, with $h = 1$, or could be an h-step trellis for some $h \leq 2\nu$. We found convenient to use from ν to $\nu+3$ stages wide initial segments . For ease of notation, we will call a $h2^m$-step survivor trellis, that combines $h2^m$ stages of a 1-step trellis, just a 2^m-step trellis, even if $h > 1$, and let L denote the number $\log_2$ of the number of stages in the initial h-step trellis, so that $K = h2^L$. The building block of our parallel decoder is a module reduces two 2^m-step survivor trellis stages into a single stage of a

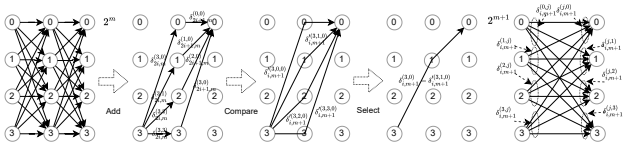

Fig. 3. Construction of a 2^{m+1}-step, fully connected, survivor trellis stage i from two 2^m-step, fully connected, survivor trellis stages $2i, 2i+1$.

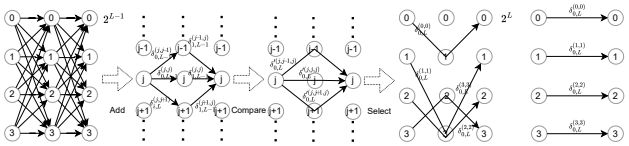

Fig. 4. Tail-bit step (last step) that constructs a single 2^L-step, not fully-connected, survivor trellis, from the last two 2^{L-1}-step, fully-connected, survivor trellis stages.

2^{m+1}-step survivor trellis. At each step $m = 2, 3, \ldots, L$ the decoder uses 2^{L-m} of these modules in parallel, which we describe next.

The trellis reduction module consists of two fundamental operations. The first is a trellis expansion, that obtains all the paths that connect every source state s of the first 2^m-step survivor trellis to every destination state d of the second 2^m-step survivor trellis through every intermediate state r, where $s, r, d = 0, 1, \ldots, 2^\nu - 1$. The second fundamental operation is to select, for every s, d pair, the single best intermediate state r that defines path connecting s and d. A stage i of a 2^m-step trellis is fully defined by the edges connecting each source state s to each destinations state d, $(s, d) \in \{0, 1, \ldots, 2^\nu - 1\}^2$, see Fig. 3. The module reduces stages $2i$ and $2i+1$ of of a 2^m-step survivor trellis into stage i of a 2^{m+1}-step survivor trellis, see left of Fig. 3. Let the negative log domain metric of the branches from source state s to destination state d that define stage i of a 2^m-step survivor trellis be $\delta_{i,m}^{s,d}$, and let

$$\delta_{i,m+1}^{\prime s,r,d} \triangleq \delta_{2i,m}^{s,r} + \delta_{2i+1,m}^{r,d} \,. \tag{3}$$

The first fundamental operation is to obtain every $\delta_{i,m+1}^{\prime s,r,d}$, for each $i = 0, 1, \ldots, 2^{L-m-1} - 1$, and every $s, d = 0, 1, \ldots, 2^\nu - 1$. If the 2^m-step survivor trellis is fully connected, then every state $r \in \{0, 1, \ldots, 2^\nu - 1\}$ will be a valid intermediate state, and the number of branches with metric $\delta_{i,m+1}^{\prime s,r,d}$ at each stage i will be $2^{3\nu}$. Otherwise, r could be from a smaller subset, like the trellis of Fig. 2, where a single intermediate state r of the 1-step trellis connects every s, d of the 2-step trellis. In the second operation, the branch metrics $\delta_{i,m+1}^{s,d}$ of the 2^{m+1}-step survivor trellis are selected from the metrics $\delta_{i,m+1}^{\prime s,r,d}$'s, via:

$$\delta_{i,m+1}^{s,d} \triangleq \min_{r \in \{0,1,\ldots,2^\nu - 1\}} \{\delta_{i,m+1}^{\prime s,r,d}\} \,. \tag{4}$$

This method allows to cap the number of edges of every 2^m-step survivor trellis stage at $2^{3\nu}$ edges, which is reached when the trellis becomes fully connected see right of Fig. 2 and both ends of Fig. 3.

The final, single stage, 2^L-step survivor trellis is constructed from a 2^{L-1}-step survivor trellis with only two stages. The tail-biting constraint of a TBCC demands that the best path starts and ends at the same state. Thus, for the 2^L-step trellis from stage 0 to stage L we need not construct every $\delta_{i,m+1}^{\prime s,r,d}$ and $\delta_{i,m+1}^{s,d}$. Only the ones where $s = d$ are needed, that is, every $\delta_{i,m+1}^{\prime s,r,s}$ and $\delta_{i,m+1}^{s,s}$, as shown in Fig. 4, since the others define invalid paths. At this point there are 2^ν valid tail biting L-long edges from each source and destination pair $(s, s), s = 0, 1, \ldots, 2^\nu - 1$. At this point, it only remains to recover the estimate $\hat{\mathbf{U}}$. Note that an estimate can be constructed from any sequence of connected states $s_0, s_1, \ldots, s_{2^L}$ for each base trellis stage, the edges connecting each s_i to s_{i+1}, $i = 0, 1, \ldots, 2^L$ and corresponding h-symbol inputs.

To recover the estimate $\hat{\mathbf{U}}$ defined in (1), we need the unique sequence of states $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{2^L}$, with $\hat{s}_{2^L} = \hat{s}_0$, that define the tail-biting path with the best metric. This metric will be $\delta_L^{\hat{s}_0,\hat{s}_0}$, where $\hat{s}_0$ is the state s that minimizes, $\delta_L^{s,s}$ given by:

$$\hat{s}_0 \triangleq \underset{s \in \{0,1,\ldots,2^\nu - 1\}}{\operatorname{argmin}} \{\delta_{0,L}^{s,s}\} \,. \tag{5}$$

The rest of the state $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_{2^L - 1}$ obtained recursively, with each $\hat{s}_{(2i+1)2^{m-1}}$ recovered from $\hat{s}_{i2^m}$ and $\hat{s}_{(i+1)2^m}$, for $m = L, L-1, \ldots, 1$ and $i = 0, 1, \ldots, 2^{L-m} - 1$ via:

$$\hat{s}_{(2i+1)2^{m-1}} \triangleq \underset{r \in \{0,1,\ldots,2^\nu - 1\}}{\operatorname{argmin}} \{\delta_{i,m+1}^{\prime s,r,d} \colon s = \hat{s}_{i2^m}, d = \hat{s}_{(i+1)2^m}\} \,. \tag{6}$$

These r values are the argument selected in (4), and could just be looked-up using the "keys" $s = \hat{s}_{i2^m}$ and $d = \hat{s}_{(i+1)2^m}$, out of the $2^\nu \times 2^\nu$ saved in (4). Lastly, if the edges of the initial h-step survivor trellis were selected from parallel edges, then the arguments and inputs to those edges are also needed to reconstruct $\hat{\mathbf{U}}$. Otherwise, the input that define a transition from each state $\hat{s}_i$ to each state $\hat{s}_{i+1}$ are unique.

We claim that our parallel trellis stage decoder is a maximum likelihood (ML) decoder. An ML trellis decoder is guaranteed to produce the codeword with the best (lowest) square metric, defined in (1). To prove our claim it suffices to show that our decoder also produces this metric. Note that the tail-biting constraint demands that the ML path be a valid path that starts and end at the same state. The parallel trellis stage decoder first finds metrics $\delta_{0,L}^{s,s}$ for each valid starting state s, and then selects the best out of $s = 0, 1, \ldots, 2^\nu$. Thus, we only need to show that every $\delta_{0,L}^{s,s}$ is the best (lowest) possible metric for a path from $s_0 = s$ to $s_{2^L} = s$. The proof can be argued with a contradiction.

Proof: The Parallel Trellis State is an ML Decoder:

Suppose that there is some $\delta_{0,L}^{*s,s} < \delta_{0,L}^{s,s}$. This $\delta_{0,L}^{*s,s}$ defines a sequence of $\delta_{i,m}^{*s,d}$ for $m = 0, 1, \ldots, L-1$ and $i = 0, 1, \ldots, 2^{L-m} - 1$. Let m^* be the minimum such m:

$$m^* \triangleq \min_m \{\delta_{i,m}^{*s,d} < \delta_{i,m}^{s,d} : s, d \in \{0, 1, \ldots, 2^\nu - 1\}^2\} \tag{7}$$

$$m < m^* \implies \delta_{i,m}^{*s,d} = \delta_{i,m}^{s,d} \,. \tag{8}$$

If such $\delta_{0,L}^{*s,s} < \delta_{0,L}^{s,s}$ exists, then m^* exists, and $m^* \leq L$, since L is a candidate. Since each $\delta_{i,0}^{s,d}$ of the starting h-step trellis

is unique, then and m^* cannot be 0, thus $m^* > 0$. However, if $\delta^{*s,d}_{i,m^*} < \delta^{s,d}_{i,m^*}$, then, there is a pair $\delta^{*s,r}_{2i,m^*-1}$ and $\delta^{*r,d}_{2i+1,m^*-1}$ where either $\delta^{*s,r}_{2i,m^*-1} < \delta^{s,r}_{2i,m^*-1}$ or $\delta^{*r,d}_{2i,m^*-1} < \delta^{r,d}_{2i+1,m^*-1}$, which contradicts (8). The proof is complete. ■

B. Decoding latency

The latency of our algorithm depends on the code parameters ν, L and h, and the hardware available. We proceed to describe the lowest possible decoding delay. Suppose that $2^L \times 2^{3\nu}$ computing nodes are available and broadcasting data across nodes does not cause delay. At each step $m+1$ each of the 2^{L-m-1} stages could be processed simultaneously. For each stage i, the decoder computes $2^{2\nu}$ values of $\delta^{s,d}_{i,m+1}$, $(s,d) \in \{0,1,\ldots,2^\nu-1\}^2$, each selected from 2^ν values of $\delta'^{s,r,d}_{i,m+1}$ that could be computed simultaneously. Finding the minimum $\delta^{s,r,d}_{i,m+1}$, see equation (4), could take as low as ν time units using binary search, which would require significantly more hardware. In a more realistic scenario, constructing each the 2^{m+1}-step survivor trellis would take a time proportional to ν. We also need to factor the time required to initialize the h-step trellis with 2^L stages, which entails computing the $\log$ domain likelihood metrics $\delta^{s,d}_{i,0}$. There are a total of $2^L \times 2^{2\nu}$ such values, that for the AWGN channel, are sum of the square differences between the h modulated output bits of each edge and the h received symbols $Y_{ih}, Y_{ih+1}, \ldots, Y_{(i+1)h-1}$. With the same number of processing nodes this could take a time proportional to h time units. Reconstructing the estimate $\hat{\mathbf{U}}$ could take as little as L time units and is a much simpler process. If $h = \nu$, a very convenient choice, the final decoding time T could be a linear function of νL, in time units, that is, of order $O(\nu L)$. Since $K = \nu \times 2^L$, we could also express the order of T by $O(\nu(\log_2(K) - \log_2(\nu))$. For fixed ν the latency grows with the $\log$ of the block length, while the required hardware would only increase linearly.

C. Complexity

The lower decoding latency of the parallel trellis stage decoder comes at the price of increase computational complexity. The operations are equivalent to running the Viterbi algorithm once for every possible starting state. However, it also allows 2^ν parallel branches to form before comparing and selecting a survivor, instead of the usual ν pairs for every ν stages of a 1-step trellis. To make an estimate of the number of operations, suppose $h = \nu$ and $K = \nu 2^L$. When $m = 1$ we need to compute all $\delta^{s,d}_{i,2}$. First we use $2^{L-1} \times 2^{3\nu}$ additions to obtain each $\delta'^{s,r,d}_{i,1}$, see (3), then we need about as many comparisons to select all $\delta^{s,d}_{i,m+1}$, each out of 2^ν metrics $\delta'^{s,r,d}_{i,1}$, see (4). The number of stages reduces to half at every step $m = 2, 3, \ldots, L-1$, and the 2^L-step trellis constructed at step L requires fewer operations, but this does not significant impact the computational complexity. Thus, the computational complexity is of order $2^L \times 2^{3\nu}$, or $O(\frac{K}{\nu} \times 2^{3\nu})$, since $K = \nu \times 2^l$. In contrast, a forward pass of the standard Viterbi algorithm executes two additions and one comparison per state at each stage, and achieves a computational complexity of order $O(\nu 2^L \times 2^\nu)$ or $O(K \times 2^\nu)$. The Viterbi algorithm, however, does not produce the ML estimate in a single pass. To guarantee ML performance it would need to run 2^ν times, once per starting state. In practice, sub-optimal methods like the wrap-around Viterbi algorithm (WAVA) [19] are used. WAVA requires running the Viterbi algorithm more than once, and approaching ML performance usually takes about four iterations. The complexity of WAVA is also of order $O(\nu 2^l \times 2^\nu)$, provided the number of iterations is constant for any ν, but this might not hold for larger ν values. The complexity of our parallel trellis stage decoder is higher by a factor of $\frac{2^{2\nu}}{\nu}$ which increases with ν. The point where the complexities are similar is at about $\nu = 2$. However, for the increased complexity, we get a very low delay, parallel processing, and ML performance.

D. Modification of the TBCCs

The parallel trellis stage decoder could support some modifications to the TBCC code to improve performance further. Suppose that we select a starting trellis segment size h greater than ν. This results in a starting h-step trellis with $2^{h-\nu}$ parallel edges, each h-binary stages long. We could divert from a convolutional code, and select the entries of the edges to increase the code distance spectrum. The edge lengths could be $h = \nu+1, \nu+2, \ldots, 2\nu$. Note that at $h = 2\nu$ there will be 2^ν parallel edges for each state pair in an initial h-step trellis, after which the complexity order starts increasing above that of the trellis reduction steps. Note that the decoder architecture depends on the length of the edges, the number 2^L of initial stages, and the number of states 2^ν, but not on the entries of the edges. Thus, a larger value of h allows us the freedom to implement any block code with the same h-step trellis shape, and and possibly improve the frame error rate performance at no additional complexity or latency cost.

We implemented the modification procedure to construct a decoder for a rate $1/2$, $\nu = 3$ TBCC code with input length 40, with $h = \nu + 2 = 5$, so that the initial trellis was a 5-step trellis. Then, we modified the 10-symbol edges to obtain a better performing block code. The performance of the original TBCC and the modified block code are shown in Fig. 5.

E. Parallel Decoding of Trellis Stages for List Decoding

We now introduce two list decoder versions of our parallel Viterbi algorithm. For both approaches, the encoder first adds m redundancy bits to the K-long information sequence, using a cyclic redundancy check (CRC) code, and then encodes the $K+m$ sequence using a tail-biting convolutional code. The m redundancy bits allow the list decoder to check if an estimate sequence $\hat{\mathbf{U}}$ is a valid codeword of the CRC code. When the check fails, the estimate is rejected, and the list decoder searches for a new estimate. The additional m redundancy bits reduce the code rate by a a factor of $(K-m)/K$. The goal is that the error detection capabilities offset the rate loss for a better overall performance, measure by the frame error rate FER Vs. the ratio of energy per information bit and noise energy $E_b N_0$.

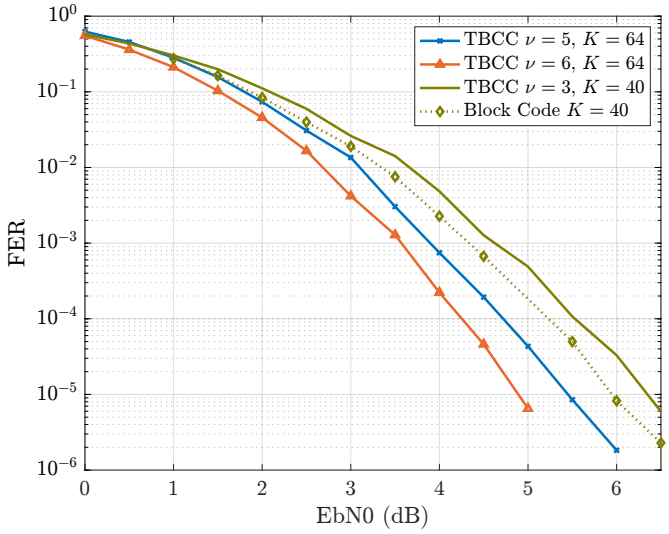

Fig. 5. FER vs E_sN_0(dB) performance our parallel decoder designed for three different TBCC codes with $3, 5$ and 6 memory elements ν and two block codes based on $\nu = 3$ and $\nu = 6$ codes. The top green solid line is an 8 state $\nu = 3$ TBCC code with polynomials $\{13, 17\}$ and $K = 40$. The green dotted line with diamonds is a block code that modify the $\nu = 3$ TBCC code and is decoded with the same decoder. The blue line with x is and orange line with triangles are TBCC code with $K = 64$ and $\nu = 5, 6$, the polynomials are $\{43, 75\}$ for $\nu = 5$ and $\{133, 171\}$ for $\nu = 6$.

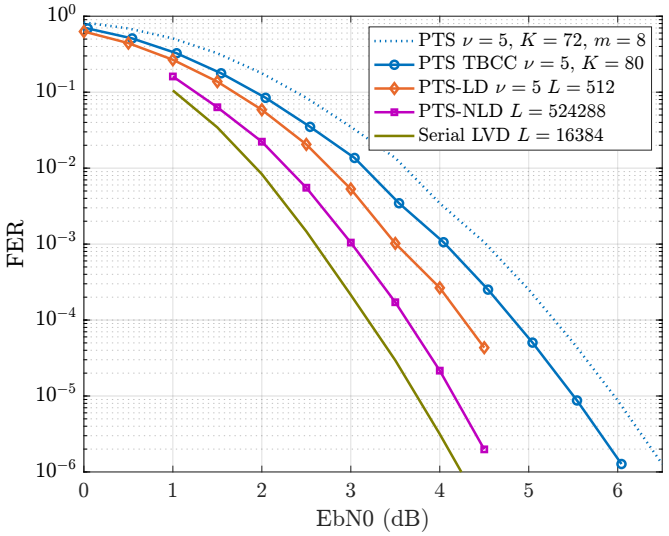

Fig. 6. FER vs E_sN_0 (dB) performance of the two list decoders we implemented and three reference curves. The top blue dotted line is the performance of the code with 72 bit information bits and an 8 bit CRC that is decoded without list decoding. The CRC can only be used for error detection. The blue solid line with circles is the performance of of the code in the previous line, but without the rare loss from the CRC bits; the orange solid line with diamonds is for our parallel list decoder that produces a list of of 2^l estimates per state; the solid magenta line with squares is the performance of our second list decoder that searches a neighborhood around the estimates generated by the previous list. For reference, we include the performance of a serial list Viterbi decoder with maximum list size $L = 16384$, solid green line. For all curves the code is a CRC aided TBCC code with $\nu = 5$ polynomials $\{53, 75\}$ octal, $K = 72$, and 8-bit CRC $0x1AB$.

We know describe the first a parallel trellis stage list decoder that exhibits the same latency as the original decoder. Note that the parallel trellis stage decoder is already a list decoder, in the sense that it produces 2^ν estimate per at each of the 2^ν valid starting (and ending) state pairs, for a total of $2^{2\nu}$ estimates, see right side of Fig. 4. However, even the second best path is not guaranteed to be among them. Our first list decoder produces even more estimates and guarantee that at least the second most likely path is included in the list. Remember that the number trellis stages reduces by half for every $m+1 = 1, 2, \ldots, L$. When selecting the survivor $\delta^{s,d}_{i,m+1}$ out of all $\delta'^{s,r,d}_{i,m+1}$, (4), we could also keep the second best path. The set of second best paths defines a second 2^{m+1}-step trellis with 2^{L-m-1} stages, for a combined total of 2^{L-m} stages. Then, at step $m + 2$ we could construct four 2^{m+2}-step trellises, each 2^{L-m-2} stages for a total of 2^{L-m} stages. At each step the number of trellises doubles and the size of each is reduce by half. Thus the number of operations at each step stays constant, at the level of the first step $m = 1$. The list decoder produces a total $2^L \times 2^{2\nu}$ list estimates, all of which are produced at the same time, and with the same latency as the basic version of the parallel trellis stage decoder. However, only the best and second best paths are guaranteed to among them, when $h \leq \nu$. The third and subsequent best paths are not guaranteed, but can only be excluded if the first three most likely paths start and end at the same state, and even then, the third one might still be included.

The second list decoder version extends the first approach by searching the neighboring tail-biting codewords of the estimates produced by the first list decoder. For this decoder, we first construct a list of tail-biting (TB) codewords with monotonically increasing Hamming weights. This can be done using the sieve method described in [20]. After getting all the estimates produced by the first list decoder, each of the estimates is then added to the neighboring TB codewords through XOR operations, which increases the number of list estimates by a factor equal to the number of TB codewords in the explored neighborhood. Since the XOR operation is fast, the overall decoding latency will be very similar to that of the first parallel trellis stage list decoder. The additional codewords explored using linearity provide an improvement in the decoding performance, as shown in Fig. 6.

IV. Simulation Results

We implemented several versions of the parallel trellis stage decoder as neural network models using PyTorch to leverage the parallel processing capabilities. To build the trellis, we generated the input and modulated output of all the edges in the trellis and saved them in separate data structures. To compute the log domain square differences metrics for each edge, we first subtract the modulated edge outputs from the receive signal, then square the difference and sum the components of each edge. From there the process consists of constructing 2^{m+1}-step survivor trellis segments out of every pair of 2^m-step survivor trellises, as depicted in Fig. 3. We also implemented both list decoder versions for a $\nu = 5$ rate $1/2$ TBCC with $h = \nu$ and $L = 4$. For the linearity step of the

second list decoder we used 1024 TB neighboring codewords. The total number of estimates produced by this list decoder is given by $2^L \times 2^\nu \times 1024 = 2^{19}$, or about $500K$.

Simulation results for the decoders we implemented are provide. The performance of the original parallel trellis stage decoder is shown in Fig. 5 and for the parallel trellis stage list decoders in Fig. 6. The graphs show frame error rate (FER) vs. ratio of bit energy and noise energy in dB, $E_b N_0$. In Fig. 5 we show the performance of three rate $1/2$ TBCCs, one with memory $\nu = 3$, $h = 5$, and message size $K = 40$ and two with $\nu = 5, 6$, $h = 8$ and message size $K = 64$. These decoders used 8 stages of an h-step trellises, with initial edges length $h = K/8$. For the 8 state TBCC with $\nu = 3$ we also designed a modified code to obtain the performance of the dotted green line with diamonds. This performance is better than that of the original TBCC code, and the decoder is the same.

Performance plots for the low latency list decoders we proposed are shown in Fig. 6. We used a $\nu = 5$ TBCC with polynomials $\{53, 75\}$ concatenated with an 8-bit CRC. The trellises have 16 stages and edges length ν. The TBCC input is a 72-bit information sequence and the 8-bit CRC. The solid orange curve with diamonds is the performance of the low latency list decoder. The solid magenta line with squares is the performance of the same list decoder but with a neighborhood search over 1024 neighboring TB codewords. For reference we include the performance of a serial list Viterbi decoder with maximum list size of $16K$ shown with solid green line. We also show the performance of the same code without list decoding, with and without adjusting for the CRC rate loss, blue lines. The list decoder results show that each version provides a significant performance improvement. The first version improves the performance by about 0.5 dB over the basic parallel trellis stage decoder at a target FER rate of 10^{-4}. The second version, with the linearity step, improves the error rate by an additional 0.8 dB, and approaches the ML decoding performance of a serial LVD with a gap of 0.4 dB. Increasing the linearity list size could further improve the decoding performance and we leave that for future work. These results show that our decoder design is suitable for practical implementation on parallel processing systems.

V. Conclusion

By processing trellis stages in parallel, this paper demonstrates a maximum likelihood decoder for tail-biting convolutional codes with a latency proportional to the log of the number of message bits. This low latency requires sufficient hardware to perform a large number of operations in parallel, proportional to the square of the number of states. We also demonstrated two list-decoder enhancements that further improve the performance of the decoder by about one to two dB. Finally, we presented a modification of the code design that takes advantage of the decoder structure to further improve the performance.

Our parallel architecture can easily be modified to implement the BCJR algorithm [3], as building blocks on concatenated codes like turbo codes [21]. This application could be better suited for our parallel decoder architecture, since the constituent codes generally use smaller ν. This application is currently a work in progress.

References

[1] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inf. Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[2] G. Forney, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[3] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Trans. on Information Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[4] G. Fettweis and H. Meyr, "Parallel viterbi algorithm implementation: breaking the acs-bottleneck," *IEEE Trans. on Com.*, vol. 37, no. 8, pp. 785–790, 1989.

[5] A. Mohammadidoost and M. Hashemi, "High-throughput and memory-efficient parallel viterbi decoder for convolutional codes on gpu [arxiv]," *arXiv (USA)*, pp. 8 pp. –, 2020/11/18.

[6] H. Peng, R. Liu, Y. Hou, and L. Zhao, "A gb/s parallel block-based viterbi decoder for convolutional codes on gpu," in *2016 8th Intl. Conf. on Wireless Comms. & Signal Processing (WCSP)*, 2016, pp. 1–6.

[7] Z. Du, Z. Yin, and D. A. Bader, "A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda," in *2010 IEEE Int. Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[8] M. Hanif and K.-H. Zimmermann, "Accelerating viterbi algorithm on graphics processing units," *Computing (Germany)*, vol. 99, no. 11, pp. 1105 – 23, 2017/11/. [Online]. Available: http://dx.doi.org/10.1007/s00607-017-0557-6

[9] N. Seshadri and C.-E. Sundberg, "List Viterbi decoding algorithms with applications," *IEEE Tran. Comm. Theory*, vol. 42, pp. 313–323, 1994.

[10] C. Rachinger, J. B. Huber, and R. R. Muller, "Comparison of convolutional and block codes for low structural delay." *IEEE Tran. on Comms.*, vol. 63, no. 12, pp. 4629 – 4638, 2015.

[11] Y. Ben Asher, V. Tartakovsky, K. Portman, O. Zilberman, and A. Hadar, "An fpga scalable parallel viterbi decoder," in *2018 IEEE 12th Intl. Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018, pp. 8–15.

[12] P. Elias, "Error-correcting codes for list decoding," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 5–12, 1991.

[13] F. Soong and E.-F. Huang, "A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition," in *[Proceedings] ICASSP 91: 1991 Intl. Conf. Acoust., Speech, Signal Process.*, 1991, pp. 705–708 vol.1.

[14] M. Roder and R. Hamzaoui, "Fast tree-trellis list Viterbi decoding," *IEEE Trans. Comm.*, vol. 54, no. 3, pp. 453–461, 2006.

[15] H. Yang, E. Liang, and R. D. Wesel, "Joint design of convolutional code and CRC under serial list Viterbi decoding," 2018.

[16] H. Yang, S. V. S. Ranganathan, and R. D. Wesel, "Serial list Viterbi decoding with CRC: Managing errors, erasures, and complexity," in *2018 IEEE Global Comm. (GLOBECOM)*, 2018, pp. 1–6.

[17] W. Sui, H. Yang, B. Towell, A. Asmani, and R. D. Wesel, "High-rate conv. codes with CRC-aided list decoding for short blocklengths," 2022.

[18] L. Wang, D. Song, F. Areces, and R. D. Wesel, "Achieving short-blocklength RCU bound via CRC list decoding of TCM with probabilistic shaping," in *ICC 2022 - IEEE Intl. Conf. on Commun.*, 2022, pp. 2906–2911.

[19] R. Shao, S. Lin, and M. Fossorier, "Two decoding algorithms for tailbiting codes," *IEEE Trans. on Com.*, vol. 51, no. 10, pp. 1658–1665, 2003.

[20] R. Wesel, A. Antonini, L. Wang, W. Sui, B. Towell, and H. Grissett, "Elf codes: Concatenated codes with an expurgating linear function as the outer code," in *2023 12th Int. Symposium on Topics in Coding (ISTC)*, 2023, pp. 1–5.

[21] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Proceedings of ICC '93 - IEEE Int. Conf. on Com.*, vol. 2, 1993, pp. 1064–1070 vol.2.