

Gear-Shift Decoding

Masoud Ardakani, *Member, IEEE*, and Frank R. Kschischang, *Fellow, IEEE*

Abstract—This paper considers a class of iterative message-passing decoders for low-density parity-check codes in which the decoder can choose its decoding rule from a set of decoding algorithms at each iteration. Each available decoding algorithm may have a different per-iteration computation time and performance. With an appropriate choice of algorithm at each iteration, overall decoding latency can be reduced significantly, compared with standard decoding methods. Such a decoder is called a gear-shift decoder because it changes its decoding rule (shifts gears) in order to guarantee both convergence and maximum decoding speed (minimum decoding latency). Using extrinsic information transfer charts, the problem of finding the optimum (minimum decoding latency) gear-shift decoder is formulated as a computationally tractable dynamic program. The optimum gear-shift decoder is proved to have a decoding threshold equal to or better than the best decoding threshold among those of the available algorithms. In addition to speeding up software decoder implementations, gear-shift decoding can be applied to optimize a pipelined hardware decoder, minimizing hardware cost for a given decoder throughput.

Index Terms—Extrinsic information transfer (EXIT) charts, iterative decoders, low-density parity-check (LDPC) codes.

I. INTRODUCTION

CODES defined on graphs together with iterative message-passing algorithms are capable of approaching the capacity of many channel types, e.g., [1]–[7]. There are many different message-passing algorithms with different performance and complexity. High-complexity decoding algorithms, such as the sum-product algorithm [8], can correct more errors created by the channel noise, so they are very attractive when complexity and computation time are not an issue. Low-complexity decoding algorithms, however, are more attractive when fast decoders are required for delay-sensitive applications or high-throughput systems.

Low-complexity decoding rules, however, have two main drawbacks. First, they have a worse threshold of decoding compared with high-complexity algorithms, so, to ensure convergence, a lower code rate should be used. Second, due to their relatively poor performance, more iterations of such algorithms are required to achieve a given target bit-error rate. Hence, it is not obvious if the overall computations are any less.

Paper approved by C. Schlegel, the Editor for Coding Theory and Techniques of the IEEE Communications Society. Manuscript received March 14, 2004; revised August 2, 2005.

M. Ardakani was with the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada. He is now with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2V4, Canada (e-mail: ardakani@ece.ualberta.ca).

F. R. Kschischang is with the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: frank@comm.utoronto.ca).

Digital Object Identifier 10.1109/TCOMM.2006.877971

Consider, now, the following scenario: a long-length (4,8) regular low-density parity-check (LDPC) code¹ is used with binary antipodal $\{+1, -1\}$ signaling for data transmission over an additive white Gaussian noise channel whose signal-to-noise ratio (SNR) (defined as $1/\sigma^2$ where σ^2 is the per-sample noise variance) is 2.5 dB. Four different decoding algorithms: sum-product, min-sum, Algorithm E (a ternary message-passing algorithm) [1] and Gallager's Algorithm B [1], [9] are implemented in software. Based on these implementations, the computation time of these algorithms are 5.2, 2.8, 0.9, and 0.39 μs per bit per iteration ($\mu\text{s}/\text{b}/\text{iter}$), respectively.

A routine analysis shows that nine iterations of sum-product decoding are required to achieve a target message-error rate of less than 10^{-7} . That is to say, the decoding time is $46.8 \mu\text{s}/\text{b}$. Using min-sum decoding, density-evolution (DE) analysis shows that 20 iterations are required, hence, the decoding time would be $56.0 \mu\text{s}/\text{b}$. It can be easily verified that decoding will fail if Algorithm E or Gallager's Algorithm B are used. Notice that although min-sum is a lower complexity algorithm, its overall decoding time is, in fact, greater than that of the sum-product algorithm.

Now assume that the decoder is allowed to vary its decoding strategy at each iteration. Using DE, one can verify that by performing four iterations of the sum-product algorithm, followed by five iterations of Algorithm E, followed by four iterations of Algorithm B, the decoder will achieve the same target error rate with a computation time of just $26.9 \mu\text{s}/\text{b}$. This is less than 60% of the time required by a sum-product decoder, and is more than twice as fast as a min-sum decoder. It also makes use of Algorithms E and B, which are very fast decoding algorithms, but which could not be used on their own (in the initial iterations) with this level of channel noise.

We call the strategy of switching from one algorithm to another “gear-shift decoding,” as the decoder changes gears (its algorithm) in order to speed up decoding. In this paper, given a set of decoding algorithms and their computation time, we show that the optimum combination of algorithms (in the sense of minimizing the decoding time) can be found via dynamic programming. We also show that, with some modifications, gear-shift decoding can be used to design pipeline decoders with minimum hardware cost for a given throughput.

The remainder of this paper is organized as follows. In Section II, we formulate the gear-shift decoding optimization problem, and then we show that using extrinsic information transfer (EXIT) chart analysis, the problem can be solved through dynamic programming. The minimum hardware design problem is defined and solved in Section III. Section IV presents some examples, and Section V concludes the paper.

¹i.e., a code in whose Tanner graph all variable nodes have degree 4 and all check nodes have degree 8.

II. GEAR-SHIFT DECODING

Gear-shift decoding, simply put, means allowing the decoder to change its decoding algorithm during the process of decoding. Gear-shift decoding for LDPC codes, interestingly, dates back to Gallager and his original work on LDPC codes [9]. For binary message-passing decoders, Gallager noticed that by allowing a decision threshold to be changed during the process of decoding, the decoder's performance and convergence threshold improves significantly. This gear-shifting algorithm of Gallager is referred to as Algorithm B in [1]. A version of gear-shift decoding termed "two-stage hybrid decoding" also appears in [10, Sec. IV-D], where the authors combine several sum-product decoding iterations with hard-decision majority-logic decoding of finite-geometry LDPC codes. The authors indicate that decoding latency can be improved significantly using this method. Although not, strictly speaking, a gear-shift decoder, another related decoding strategy involving several different binary message-passing algorithms is the "hybrid decoding" approach of [11], in which the variables of an LDPC decoder are partitioned into several subsets, with the nodes in different subsets implementing a different update rule. This results in an overall behavior which can be better than the convergence behavior of every single algorithm.

Although gear-shift decoding can improve the decoding threshold for certain binary message-passing decoders, such improvements are *not* expected when the sum-product algorithm is available, since, as is well known, sum-product decoding minimizes error probability when the factor graph of the code is cycle-free. The point of introducing gear-shift decoding in this case is *not* to improve the decoding threshold, but rather to reduce the decoding complexity. As we will show in this paper, complexity reductions can be quite significant.

In order to formulate the gear-shift decoding problem, we assume that an EXIT chart analysis is accurate enough for the purpose of comparing the performance of different algorithms. Different measures of decoding complexity can be used. In this paper, we first assume a software decoder and consider decoding latency, i.e., the time needed to decode a received word, as the complexity measure to be minimized. Later, we will also consider a pipeline-structured hardware decoder implementation, and use circuit area as the complexity measure.

A. EXIT Chart Analysis of Iterative Decoders

Message-passing iterative decoders can be analyzed using a technique called DE [2]. DE tracks the evolution of the probability density function (pdf) of extrinsic messages iteration by iteration. This analysis becomes intractable when the constituent codes are complex, e.g., in turbo codes. Even in the case of LDPC codes with the simplest constituent code (simple parity checks), this algorithm is quite computationally intensive. Another approach for analyzing iterative decoders, including codes with complicated constituent codes, is to use EXIT charts [12]–[15].

In EXIT chart analysis, the idea is to track the evolution of a single parameter of the message pdf iteration by iteration [12]–[15]. Clearly (unless the message density truly is characterized by a single parameter, as in an erasure channel), EXIT chart analysis is not as accurate as DE; however, for many

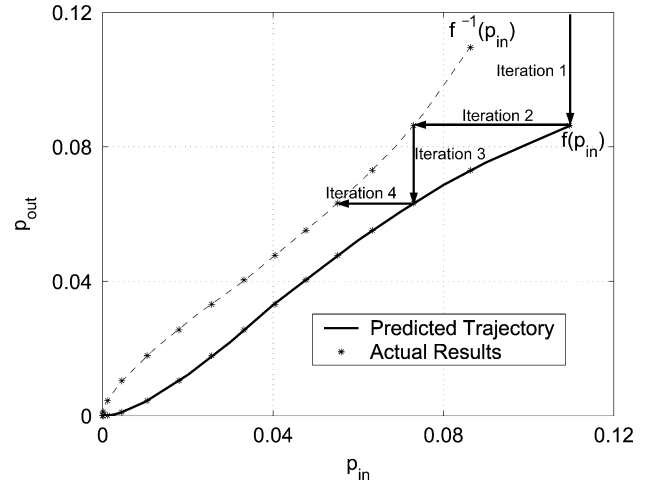


Fig. 1. EXIT chart for a (3,6) regular LDPC code on a Gaussian channel at an SNR of 1.8 dB, comparing the predicted trajectory with the actual results for a code of length 200 000.

applications, EXIT charts are very accurate indeed (see, e.g., [12] and [16]). Methods of obtaining EXIT charts for turbo codes and LDPC codes are discussed in detail in [12]–[16].

In the remainder of this paper, to compare the performance of different decoding algorithms, we use EXIT charts that track the extrinsic message-error rate, i.e., the (expected) fraction of messages that convey an incorrect belief about their corresponding variable. Such EXIT charts are described in detail in [16]. For the purposes of this paper, it is enough to think of an EXIT chart as a function

$$p_{\text{out}} = f(p_{\text{in}}, p_0) \quad (1)$$

where p_0 is the intrinsic message-error rate (i.e., the expected fraction of incorrect received symbols), p_{in} is the extrinsic message-error rate at the input of the iteration, and p_{out} is the extrinsic message-error rate at the output of the iteration. If the decoder is to make progress (reducing the fraction of messages conveying an incorrect belief) from iteration to iteration, one clearly needs

$$f(p_{\text{in}}, p_0) < p_{\text{in}} \quad \text{for all } p_{\text{in}} \in (0, p_0]. \quad (2)$$

EXIT charts are usually plotted, for a fixed p_0 , with both f and its inverse f^{-1} shown on the same graph. This makes it easy to visualize the decoder's progress, as the output p_{out} from one iteration transfers to the input p_{in} of the next, as illustrated in Fig. 1. Each arrow in Fig. 1 represents one iteration of decoding. EXIT charts therefore allow one to study how many iterations are required to achieve a target message-error rate.

If the "decoding tunnel" of an EXIT chart is closed, i.e., if for some p_{in} (2) is not satisfied, convergence does not happen. In such cases, we say that the EXIT chart is closed. The so-called *convergence threshold* p_0^* is defined as the worst intrinsic message-error rate for which the tunnel is open, i.e.,

$$p_0^* := \arg \sup_{p_0} \{ f(p_{\text{in}}, p_0) < p_{\text{in}} \quad \text{for all } 0 < p_{\text{in}} \leq p_0 \}.$$

Fig. 1 shows the EXIT chart for a (3,6) regular LDPC code of length 2×10^5 on a Gaussian channel under sum-product decoding, and compares it with the actual results. This comparison shows that such analyses can be highly accurate, at least for long codes.

B. Definitions

Consider now a set of N different decoding algorithms numbered from 1 to N , the i th of which has an EXIT chart given as $f_i(p_{\text{in}}, p_0)$. The computation time for one iteration of algorithm i is denoted t_i , and is assumed to be independent of p_{in} and p_0 .

A gear-shift decoder can be described by a *gear-shift sequence*, i.e., a sequence of integers $S = (a_1, a_2, \dots, a_{l(S)})$, where $l(S)$ denotes the number of iterations, and where $a_i \in \{1, 2, \dots, N\}$ for all $i \in \{1, \dots, l(S)\}$. The gear-shift sequence $S = (a_1, a_2, \dots, a_{l(S)})$ indicates that Algorithm a_1 is used in the first iteration, Algorithm a_2 is used in the second iteration, and so on. We will sometimes use a more compact notation for gear-shifting schedules: $(a_1^{m_1}, a_2^{m_2}, a_3^{m_3}, \dots)$ is the gear-shifting schedule in which m_1 iterations of Algorithm a_1 are followed by m_2 iterations of Algorithm a_2 , m_3 iterations of Algorithm a_3 , etc. The gear-shift decoder with gear-shift sequence S will be referred to as gear-shift decoder S .

The total computation time T_S for gear-shift decoder $S = (a_1, \dots, a_{l(S)})$ is given by

$$T_S = \sum_{i=1}^{l(S)} t_{a_i}.$$

Since the initial extrinsic message-error rate is equal to p_0 , up to the accuracy of the EXIT chart analysis, the output message-error rate P_S of gear-shift decoder S after $l(S)$ iterations is

$$P_S(p_0) = f_{a_{l(S)}}(f_{a_{l(S)-1}}(\dots f_{a_2}(f_{a_1}(p_0, p_0), p_0), \dots), p_0). \tag{3}$$

To avoid pathological cases, we make the reasonable assumption that EXIT charts are increasing functions of p_{in} for a fixed p_0 , since, for most algorithms, having a greater p_{in} , i.e., having more extrinsic message errors at the input of an iteration, will result in more message errors at the output of the iteration, i.e., a greater p_{out} . We also make the reasonable assumption that EXIT charts are nondecreasing functions of p_0 , i.e., for all i , we have that $p'_0 < p_0$ implies $f_i(p_{\text{in}}, p'_0) \leq f_i(p_{\text{in}}, p_0)$ for all $p_{\text{in}} \leq p'_0$. In this case, from (3), it is obvious that

$$p'_0 < p_0 \text{ implies } P_S(p'_0) \leq P_S(p_0) \tag{4}$$

for any gear-shifting sequence S , i.e., providing a better initial channel to a gear-shift decoder is never harmful to its output message-error rate.

For a given target message-error rate p_t , we say that gear-shift decoder S is *admissible* if $P_S(p_0) \leq p_t$. From (4), we see that if S is admissible for an intrinsic message-error rate p_0 , then it is admissible for all intrinsic message-error rates $p'_0 < p_0$. The optimum gear-shift decoder has admissible gear-shift sequence

S^* , whose computation time T_{S^*} is less than that of all other admissible gear-shift sequences; i.e., S^* is admissible, and if S is also admissible, then $T_S \geq T_{S^*}$.

We will begin by assuming that the output messages of each algorithm are compatible with the input of each algorithm so that, in principle, each decoding algorithm in $\{1, \dots, N\}$ may be followed by any other decoding algorithm in $\{1, \dots, N\}$. Later in the paper, we also consider restrictions in the choice of algorithms; for example, allowing coarsely quantized message-passing algorithms to succeed finely quantized algorithms, but not vice versa.

C. Equally Complex Algorithms

The simplest case arises when all the available decoding algorithms have equal computation time, i.e., t_i is a constant. In [17], it is rigorously proved that when all the decoding rules have the same complexity, the optimum gear-shift decoder chooses the algorithm that gives the best performance at each iteration, i.e., the one with smallest $f_i(p_{\text{in}}, p_0)$ (where p_{in} is the extrinsic message-error rate from the previous iteration). Gallager’s Algorithm B for LDPC codes is a good example for this case.

D. Dynamic Programming for Optimizing Gear-Shift Decoders

When the computation times of different algorithms are not equal, it is not clear which combination of algorithms will result in the fastest decoding. Choosing the algorithm with the best performance may cost a lot of computation, while for the same amount of computation, another algorithm might produce better results. In this section, we formulate the problem of finding the sequence of algorithms with the minimum computational complexity as a dynamic program. We first define the problem.

We use the general setup of Section II-B. At a fixed p_0 , for each EXIT function $f_i(p_{\text{in}}, p_0)$, we define a quantized version $\hat{f}_i(p_{\text{in}})$ whose domain and range is set at $\mathcal{P} = \{p_0, p_1, \dots, p_n\}$. Here, p_0 is the intrinsic message-error rate, $p_0 > p_1 > \dots > p_n$, and p_n is equal to p_t , the target message-error rate. For $p_{\text{in}} = p_m, 0 \leq m \leq n$ we define

$$\hat{f}_i(p_{\text{in}}) \triangleq \begin{cases} p_0, & \text{if } f_i(p_{\text{in}}, p_0) \geq p_0 \\ p_k & \text{otherwise} \end{cases}$$

where $k, k \in \{0, \dots, n\}$ is the largest integer for which $f_i(p_{\text{in}}, p_0) \leq p_k$. This way, \hat{f}_i is a pessimistic approximation of f_i ; hence, an admissible gear-shift decoder based on \hat{f} is guaranteed to achieve p_t .

As illustrated in Fig. 2, we form a trellis, termed the *gear-shifting trellis*, whose vertices are labeled by the elements of \mathcal{P} , and whose edges are determined by $\hat{f}_1, \dots, \hat{f}_N$. This trellis has one vertex at depth 0, corresponding to the intrinsic message-error rate p_0 . From each vertex at depth d in the trellis, we construct N edges directed to states at depth $d+1$. In particular, for each $i \in \{1, 2, \dots, N\}$, a trellis state at depth d , associated with quantized extrinsic message-error probability $p_m \in \mathcal{P}$, is connected by an edge to the trellis state at depth $d+1$, associated with $\hat{f}_i(p_m)$. Associated with this edge is the weight or “cost” t_i . Also included in such a trellis is a “null algorithm” of zero cost used to tie together the terminal states labeled p_n .

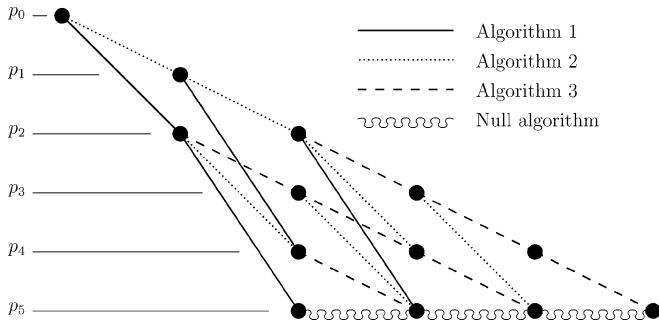


Fig. 2. Simple gear-shifting trellis with \mathcal{P} of size six and three algorithms. Notice that some vertices have fewer than three outgoing edges; this happens when some algorithms have a closed EXIT chart at this message-error rate, or when two algorithms result in a parallel edge (in which case, only the lower complexity algorithm is retained).

Clearly, every gear-shifting sequence S corresponds to a path in the gear-shifting trellis starting at the depth-zero vertex, and the total computation time T_S is obtained as the sum of the weights of the edges along that path. Furthermore, a path that ends at the vertex associated with p_n corresponds to an admissible gear-shifting sequence. An optimum gear-shifting sequence is, therefore, associated with a path of minimum total cost starting at vertex p_0 and ending at vertex p_n in the gear-shifting trellis. Such a path can easily be found by dynamic programming (e.g., via the Viterbi algorithm).

Notice that the number of edges in the gear-shifting trellis is given as $\mathcal{O}(Nn)$, so increasing n (in order to reduce the quantization error) results in a corresponding linear increase in the number of trellis edges. Since it is computationally quite feasible to handle gear-shifting trellises having even a few thousand edges, the set \mathcal{P} of quantized probability values can easily have several thousand elements. In practice, we use $n = 4000$ logarithmically spaced probability values in the range of $p_t - p_0$.

Since we are interested only in the minimum-weight path from p_0 to p_n , we can remove all branches that connect a vertex p_m at depth d to a vertex p_k at time $d + 1$ when $p_k \geq p_m$, as such branches correspond to gear shifts that are harmful to decoder progress. This further simplifies the dynamic program.

We may also expand the gear-shifting trellis to account for constraints that may be imposed on the gear-shifting sequence. For example, we may wish to impose the constraint that an iteration of highly quantized message passing such as Gallager-B or Algorithm E may follow an iteration of the sum-product or min-sum algorithms, but *not* vice-versa. In this case, there are two algorithm classes (Class 0 and Class 1, say), and so we can expand the number of vertices in the gear-shifting trellis by a factor of two. A vertex labeled p_i in the unconstrained gear-shifting trellis is split into two vertices labeled $(p_i, 0)$ and $(p_i, 1)$, respectively. Here, the second component of the vertex label is a Boolean variable indicating whether (or not) an algorithm in Class 1 has been used. The initial vertex is labeled $(p_0, 0)$. From every vertex at depth d labeled $(p_i, 0)$, we allow edges corresponding to each available algorithm, leading to a vertex at depth $d + 1$ labeled $(p_j, 0)$ for some j if the algorithm is in Class 0, and labeled $(p_j, 1)$ for some j if the algorithm is in Class 1. On the other hand, from every vertex at depth

d labeled $(p_i, 1)$, we allow edges corresponding only to algorithms in Class 1, leading to a vertex at depth $d + 1$ labeled $(p_j, 1)$. Thus, once an algorithm in Class 1 has been used in a gear-shifting sequence, the algorithms in Class 0 become unavailable. Once more, an optimum constrained gear-shifting sequence can be found by dynamic programming using the expanded gear-shifting trellis.

E. Convergence Thresholds

For equally complex algorithms, it is clear that the convergence threshold of the gear-shift decoder is equal to or better than the best of the available decoders. Note that the decoder chooses the most open EXIT chart at each iteration. The resulting EXIT chart is then the lower hull of all the EXIT charts, and hence, has an equal or even better threshold.

In the case of algorithms with different complexities, the optimum gear-shift decoder chooses the algorithms based on their complexity and performance. Hence, it might use an algorithm with a worse performance due to its lower complexity. Therefore, it is not obvious how this might affect the threshold of convergence. The following theorem shows that even in this case, the threshold of convergence can only be improved by gear-shift decoding.

Theorem 1: The convergence threshold of the gear-shift decoder is better than or equal to the best threshold among those of the available decoding algorithms.

Proof: Let us denote the best convergence threshold of available algorithms with p_0^* . We need to show that when the intrinsic message-error rate is ϵ better than p_0^* , the optimum gear-shift decoder can converge successfully.

Since p_0^* is the convergence threshold of at least one algorithm, say algorithm m , at least this algorithm has an open EXIT chart at $p_0^* - \epsilon$. In other words, for all $p_{\text{in}}, p_n \leq p_{\text{in}} \leq p_0^* - \epsilon$, we have $p_{\text{out}} = f_m(p_{\text{in}}, p_0^* - \epsilon) < p_{\text{in}}$. Therefore, in the gear-shifting trellis, there is a path between p_0 and p_n , and hence, there is a path of minimum cost between them. \square

Notice that if for some p_{in} , all decoding rules except algorithm m have a closed EXIT chart, they all result in a $p_{\text{out}} > p_{\text{in}}$. Therefore, in the gear-shifting trellis, we can remove the branches associated with them at this particular p_{in} . Hence, the optimum gear-shift decoder chooses the only existing branch, i.e., algorithm m , towards convergence. This is also true when other decoding rules have a tight EXIT chart. As we see in later examples, in tight regions, the optimum gear-shift decoder uses more complex decoders.

III. MINIMUM HARDWARE DECODING

In the previous section, we studied the case of minimum decoding latency, which, in particular, is attractive for software decoders. In this section, we show that gear-shift decoding may also be attractive to optimize a pipeline-structured hardware decoder implementation, using throughput (rather than decoding latency) as the measure of optimality.

Consider the pipelined decoder structure shown in Fig. 3. In this figure, the block labeled (i, j, k) is the i th stage of the pipeline system, which performs j iterations of algorithm number k . Our goal in this section is to show how one can minimize the cost of hardware by proper choice of j and k at each

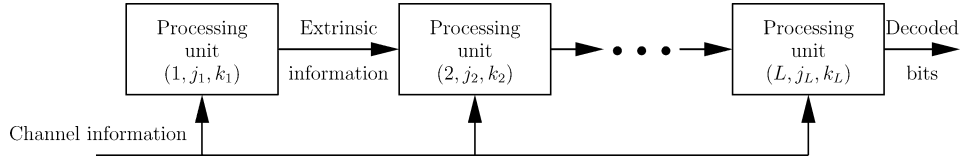


Fig. 3. Pipelined decoder architecture, where gear shifting means allowing different algorithms to be used at each pipeline stage.

block. This is another form of gear-shift decoding, specifically designed for minimum hardware. We assume the system should achieve a target message-error rate or p_t and a throughput of M b/s or greater. We also assume that the decoding latency is not an issue. If minimum latency is required, the optimum gear-shift decoder studied in the previous section is the answer, despite its hardware cost.

For a code of rate R and length N to have a throughput of M b/s, the maximum time available for decoding a codeword is N/M s. If t_j is the time for one iteration of algorithm j , a maximum of

$$n_j = \left\lfloor \frac{R \cdot N}{M \cdot t_j} \right\rfloor \quad (5)$$

iterations of algorithm j can be performed in the same hardware block. Since decoding latency is not an issue, we let all the blocks that use algorithm j iterate n_j times. Having the maximum number of iterations in a block further reduces the message-error rate and minimizes the need for extra hardware. As a result, without any extra hardware cost, all the blocks in Fig. 3 are changed to blocks of form (i, j, n_j) , where n_j is given in (5). This provides the minimum output error rate for the same hardware cost.

Now assume that the hardware cost of algorithm j is c_j . Our goal is to find a minimum-cost solution to achieve the target error rate p_t . This problem is equivalent to the problem of minimum-latency gear-shift decoding, except that the decoding time is now replaced with the hardware cost and single-iteration algorithms are replaced with multi-iteration ones. Therefore, to find the optimum solution, we need to form a similar gear-shifting trellis. A trellis edge is now associated with n_j iterations of algorithm j and its weight is c_j , and an optimal gear-shift decoder can be found via dynamic programming.

Note that this optimization is oblivious to the internal implementation of each block in the pipeline, i.e., the internal implementation of each block can be fully parallel, fully serial, or a combination of both (see, e.g., [18] and [19]). Although we are able to find a pipeline structure that achieves a given throughput with minimum hardware complexity, it is possible that some non-pipelined architecture (beyond the scope of this paper) may be able to achieve the same throughput with even smaller complexity.

IV. EXAMPLES

In this section, we solve some examples to show the effect of gear-shift decoding on decoding latency and hardware cost. In all examples, we assume that four different decoding rules are available, namely: sum-product, min-sum, Algorithm B, and Algorithm E, numbered as Algorithms 1–4, respectively.

Algorithm E is defined in [1]. Since the version of Algorithm E that we use is slightly different, we review this algorithm and mention the points of difference. In this algorithm, the message alphabet is $\{-1, 0, +1\}$. A -1 message can be thought as a vote for a 1 variable node, a $+1$ message is a vote for a 0 variable node, and a 0 message is an undecided vote. Assuming that the all-zero codeword is transmitted, the error rate of this algorithm can be defined as $\Pr(-1) + (1/2) \Pr(0)$, to be used for plotting the EXIT chart.

Following the notation of [8], the update rules of this algorithm at a check node c and a neighboring variable node v are as follows:

$$\mu_{c \rightarrow v} = \prod_{h \in n(c) \setminus \{v\}} \mu_{h \rightarrow c} \quad (6)$$

$$\mu_{v \rightarrow c} = \text{sign} \left(w \cdot \mu_{c \rightarrow v} + \sum_{y \in n(v) \setminus \{c\}} \mu_{y \rightarrow v} \right). \quad (7)$$

In (6), $n(c) \setminus \{v\}$ is the set of all the neighbors of c except v . Similarly, in (7), $n(v) \setminus \{c\}$ is the set of all the neighbors of v except c . Here $\mu_{h \rightarrow c}$ represents a message from a variable node h to the check node c , $\mu_{y \rightarrow v}$ represents a message from a check node y to the variable node v , $\mu_{c \rightarrow v}$ is the channel message to the variable node v , and w is a weight. The optimum value of w for a (3,6) code is $w = 2$ for the first iteration, and $w = 1$ for other iterations [1]. For other codes, the optimum value of w can be computed through dynamic programming. The fact that in the first iteration the extrinsic messages are not reliable makes it reasonable to have a higher weight w in early iterations, and a lower one in late iterations.

In our version of Algorithm E, the weight w is always chosen to be 1. This is mainly because, in most cases of study, Algorithm E is used in late iterations when the extrinsic messages have a relatively low error rate. Nevertheless, this slightly simplified version of Algorithm E serves our purpose of showing the impact of gear-shift decoding.

Another issue is the compatibility issue. Since the message alphabets of the above-mentioned algorithms are not the same, when a transition from one algorithm to another occurs, the output of one algorithm must be made compatible with the input of the next algorithm. Table I shows how we change the messages at the output of one algorithm for different transitions. The “not-compatible” entries refer to transitions that are not allowed. We normally avoid transitions from highly quantized (hard) algorithms E and B to unquantized (soft) algorithms, because it requires us to use a new EXIT chart for the soft algorithms, as they are now fed with hard information; however, such transitions may be considered under suitable message mappings. (One

TABLE I
MESSAGE MAPPINGS AT THE TRANSITION FROM ONE ALGORITHM TO ANOTHER

To → From ↓	Sum-Product or Min-Sum	Algorithm E {-1, 0, 1}	Algorithm B {0, 1}
Sum-Product or Min-Sum	no change	$ m > 1, m \mapsto \text{sign}(m)$ $ m \leq 1, m \mapsto 0$	$ m > 0, m \mapsto \frac{1-\text{sign}(m)}{2}$ $ m = 0, m \mapsto 0, 1$ randomly
Algorithm E	not compatible	no change	$ m = 1, m \mapsto \frac{1-m}{2}$ $ m = 0, m \mapsto 0, 1$ randomly
Algorithm B	not compatible	$m = 0, m \mapsto 1$ $m = 1, m \mapsto -1$	no change

TABLE II
EXTENDED MESSAGE MAPPINGS RESOLVING THE “NOT COMPATIBLE”
ENTRIES OF TABLE I. K IS A SCALING FACTOR

To → From ↓	Sum-Product or Min-Sum
Algorithm E	$m \mapsto K \cdot \text{sign}(m)$
Algorithm B	$m \mapsto K \cdot (2m - 1)$

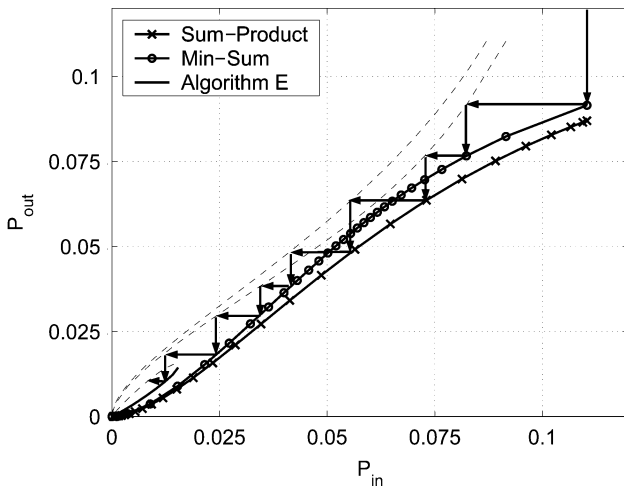


Fig. 4. EXIT charts for a (3,6) regular LDPC code under different decoding algorithms when the channel is Gaussian with SNR=1.75 dB.

possibility is suggested in Table II.) Performance of the soft decoder after such transitions is very sensitive to the mapping used for messages.

Example 1: In this example, we consider a (3,6) regular LDPC code, used to achieve a message-error rate of 10^{-6} on a Gaussian channel with SNR = 1.75 dB. The channel SNR is almost 0.65 and 0.05 dB better than the threshold of this code under sum-product and min-sum decoding, respectively. It is also more than 1.3 and 3.1 dB worse than the threshold under Algorithms E and B, respectively, i.e., these algorithms could not by themselves be used at this SNR level. We seek the minimum decoding-latency decoder.

Fig. 4 shows the EXIT charts of this code under sum-product, min-sum, and Algorithm E decoding. It also shows a decoding trajectory using different decoding algorithms. To avoid confusion, each EXIT chart is plotted only in the regions in which it

has an open decoding tunnel. The EXIT chart of Algorithm B is closed everywhere, even when the error rate of the extrinsic messages is very small, and hence, is not plotted.

Based on our implementations for a (3,6) code, one iteration of sum-product, min-sum, and Algorithm E takes $4.1 \mu\text{s/b}$, $1.7 \mu\text{s/b}$, and $0.5 \mu\text{s/b}$, respectively. These numbers were arrived at using a Pentium IV processor clocked at 1 GHz, and are intended as illustrations only. Quantizing p_{in} from $p_0 = 0.1106$ to $p_t = 10^{-6}$ in 4000 points uniformly spaced on a logarithmic scale, the optimum gear-shift decoder has decoding schedule $\mathcal{S} = (2^2, 1^5, 2^{10})$, i.e., two iterations of min-sum followed by five iterations of sum-product, followed by ten iterations of min-sum. The whole decoding takes $40.9 \mu\text{s/b}$.

Using only sum-product, a routine DE analysis shows that 15 iterations are required to achieve the same message-error rate, and hence, the decoding takes a total of $61.5 \mu\text{s/b}$. This is more than 50% longer than the optimum gear-shift decoder. Using only min-sum, we would need 31 iterations, which takes $52.7 \mu\text{s/b}$, which is more than 28% longer than the optimum gear-shift decoder.

Another important benefit of the gear-shift decoder over the min-sum decoder is the decoding threshold. If the channel SNR is overestimated by only 0.05 dB, the min-sum decoder would fail to converge no matter how many iterations of decoding we were to allow. However, as Fig. 4 suggests, when the decoding tunnel of the min-sum decoder is very tight (or even closed due to overestimation of channel SNR), the gear-shift decoder switches to sum-product. Hence, the gear-shift decoder is more robust to channel estimation, while its decoding latency is also significantly less.

If we use Table II with $K = 10$ to resolve the “not-compatible” entries of Table I, the resulting optimum gear-shift decoder has decoding schedule $\mathcal{S} = (2^3, 1^4, 2^7, 4^4, 2^3)$. The decoding time is $40.5 \mu\text{s/b}$, which is only 1% better than that of the previous example. Since for variable nodes of degree 3, Algorithms E and B both have poor performance, even at low extrinsic message-error rates, they did not have an impact on the speed of the gear-shift decoder. Nevertheless, the gear-shift decoding by only sum-product and min-sum is dramatically better than using single-algorithm decoding.

Example 2: In this example, we apply the gear-shift decoding concept in a pipelined hardware decoder implementation. Since we do not have the actual hardware implementations of these codes; and a comparison of decoding time and hardware cost for different decoding rules on a single code is, to the best of

our knowledge, not available, we set up the problem with some assumptions.

A good measure for hardware cost is the area used by the circuit. In [19], which describes a five-bit implementation of an LDPC sum-product decoder, 42% of the chip area is used for memory, another 42% is used for interconnections, and the remaining 16% is used for logic. The size of memory scales linearly with the number of bits, and if the interconnections are done fully in parallel, their area also scales linearly with the number of bits. The size of the logic scales approximately as the square of the number of bits. Using these facts, for an eight-bit decoder, the area used for memory, interconnections, and logic is about 38%, 38%, and 24%, respectively. Now, if the area of an eight-bit sum-product decoder is A , the area required for implementation of Algorithm E with two-bit messages is about $0.2A$, and for Algorithm B is about $0.1A$. For min-sum decoding, only the logic is simpler than sum-product. Assuming that its logic is even half of sum-product, it requires an area no less than $0.88A$.

In terms of decoding time, since the propagation delay is proportional to the length of the wires, or approximately to the square root of the area, we make the assumption that the decoding time of each algorithm is proportional to the square root of its required area. If the decoding time for sum-product is T , then for min-sum, it is about $0.94T$, for Algorithm E is about $0.45T$, and for Algorithm B is about $0.32T$.

Now, consider an eight-bit sum-product decoder for a (4,8) regular code whose throughput is 1.6 Gb/s per iteration. Assuming that a decoder with a throughput of 0.4 Gb/s is required, a sum-product unit in the pipeline structure of Fig. 3 can perform four iterations. A min-sum decoder also can perform a maximum of four iterations. An Algorithm E unit can perform eight iterations, and an Algorithm B unit can perform 12 iterations. Given the hardware cost (required area) of each algorithm and the performance of the multi-iteration versions of these algorithms, the optimum hardware with a throughput of 0.4 Gb/s is one block of sum-product followed by one block of Algorithm E, followed by one block of Algorithm B, to achieve a target error rate of 10^{-7} . The area required for this combination is $1.3A$. This can be compared with a pipeline system that has only sum-product decoding blocks, and needs three blocks whose hardware cost is $3A$. So in this example, using gear-shift decoding, the hardware cost is reduced to less than 44%.

V. CONCLUSION

Allowing an iterative decoder to shift gears, i.e., to switch between a set of available algorithms at each iteration, is a promising method for reducing decoding latency with no sacrifice in the decoding threshold or code rate. Gallager's Algorithm B [9], [20] is a good example of a gear-shift decoder.

For long-block-length LDPC codes, the "optimum" (minimum latency) decoder can be found from the EXIT charts of the available algorithms by solving a computationally tractable dynamic program. Gear shifting can also be applied in hardware implementations as a means of optimizing a pipelined decoder. Various examples show that significant improvements in decoding latency or hardware complexity are available.

Since EXIT charts are increasing functions of p_0 , a gear-shift decoder optimized for a target error rate p_t and for a given intrinsic message-error rate p_0 is guaranteed to achieve p_t for

any other intrinsic message-error rate $p'_0 < p_0$. Gear-shift decoders may be more robust than decoders that use just a single low-complexity decoding rule, as the gear-shift decoder can switch to a more effective decoding procedure in regions where the "decoding tunnel" is tight.

Notice that irregular LDPC codes are usually highly optimized for a certain fixed decoding algorithm, with no consideration given to decoding complexity. It has been noticed in many papers, e.g., [3], [4], [7], that the EXIT chart for such highly optimized irregular codes is very tight for all p_{in} 's. In such a situation, depending on the value chosen for p_t , the optimum gear-shift decoder may degenerate to just using the given algorithm for all iterations. When decoding complexity is a consideration, it remains an interesting open problem to optimize the structure of an irregular LDPC code specifically for a gear-shift decoder.

We note also that the idea of gear-shift decoding is very general, and can be used for other codes with iterative decoders.

REFERENCES

- [1] T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 599–618, Feb. 2001.
- [2] T. J. Richardson, A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619–637, Feb. 2001.
- [3] A. Shokrollahi, "New sequence of linear time erasure codes approaching the channel capacity," in *Proc. Int. Symp. Appl. Algebra, Algebraic Algorithms, Error-Correcting Codes*, 1999, pp. 65–67.
- [4] A. Shokrollahi, *IMA Volumes in Mathematics and its Applications*. New York: Springer, 2000, vol. 123, pp. 153–166.
- [5] S.-Y. Chung, G. D. Forney, Jr., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, no. 2, pp. 58–60, Feb. 2001.
- [6] H. Jin, A. Khandekar, and R. McEliece, "Irregular repeat-accumulate codes," in *Proc. 2nd Int. Symp. Turbo Codes, Related Topics*, Brest, France, Sep. 2000, pp. 1–8.
- [7] S. ten Brink, "Rate one-half code for approaching the Shannon limit by 0.1 dB," *Electron. Lett.*, vol. 36, pp. 1293–1294, Jul. 2000.
- [8] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.
- [9] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.
- [10] Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-density parity-check codes based on finite geometries: A rediscovery and new results," *IEEE Trans. Inf. Theory*, vol. 47, no. 11, pp. 2711–2736, Nov. 2001.
- [11] P. Zarrinkhat and A. H. Banihashemi, "Hybrid decoding of low-density parity-check codes," in *Proc. 3rd Int. Symp. Turbo Codes*, Brest, France, Sep. 2003, pp. 503–506.
- [12] S. Ten Brink, "Iterative decoding trajectories of parallel concatenated codes," in *Proc. 3rd IEE/ITG Conf. Source, Channel Coding*, Munich, Germany, Jan. 2000, pp. 75–80.
- [13] S. Ten Brink, "Convergence behavior of iteratively decoded parallel concatenated codes," *IEEE Trans. Commun.*, vol. 49, no. 10, pp. 1727–1737, Oct. 2001.
- [14] D. Divsalar, S. Dolinar, and F. Pollara, "Low complexity turbo-like codes," in *Proc. 2nd Int. Symp. Turbo Codes*, Sep. 2000, pp. 73–80.
- [15] H. El Gamal and A. R. Hammons, "Analyzing the turbo decoder using the Gaussian approximation," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 671–686, Feb. 2001.
- [16] M. Ardakani and F. R. Kschischang, "A more accurate one-dimensional analysis and design of irregular LDPC codes," *IEEE Trans. Inf. Theory*, vol. 52, no. 12, pp. 2106–2114, Dec. 2004.
- [17] M. Ardakani and F. R. Kschischang, "Gear-shift decoding," in *Proc. 21st Biennial Symp. Commun.*, Kingston, ON, Canada, Jun. 2002, pp. 86–90.
- [18] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, Mar. 2002.
- [19] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. VLSI Syst.*, vol. 11, no. 12, pp. 976–996, Dec. 2003.

- [20] M. Ardakani and F. R. Kschischang, "Properties of optimum binary message-passing decoders," *IEEE Trans. Inf. Theory*, vol. 51, no. 10, pp. 3658–3665, Oct. 2005.
- [21] A. Ashikhmin, G. Kramer, and S. ten Brink, "Extrinsic information transfer functions: Model and erasure channel properties," *IEEE Trans. Inf. Theory*, vol. 50, no. 11, pp. 2657–2673, Nov. 2004.



Masoud Ardakani (S'00–M'05) received the B.Sc. degree from Isfahan University of Technology, Isfahan, Iran, in 1994, the M.Sc. degree from Tehran University, Tehran, Iran, in 1997, and the Ph.D. degree from the University of Toronto, Toronto, ON, Canada, in 2004, all in electrical engineering.

He was a Postdoctoral Fellow at the University of Toronto from 2004 to 2005. From 1997 to 1999, he was with the Electrical and Computer Engineering Research Center, Isfahan, Iran. He is currently an Assistant Professor of Electrical and Computer Engineering at the University of Alberta, Edmonton, AB, Canada, where he holds an Informatics Circle of Research Excellence (*iCore*) Junior Research Chair in Wireless Communications. His research interests are in the general area of digital communications, codes defined on graphs associated with iterative decoding, and MIMO systems.



Frank R. Kschischang (S'83–M'91–SM'00–F'06) received the B.A.Sc. degree (with honors) from the University of British Columbia, Vancouver, BC, Canada, in 1985 and the M.A.Sc. and Ph.D. degrees from the University of Toronto, Toronto, ON, Canada, in 1988 and 1991, respectively, all in electrical engineering.

He is a Professor of Electrical and Computer Engineering and Canada Research Chair in Communication Algorithms at the University of Toronto, where he has been a faculty member since 1991. During 1997–1998, he spent a sabbatical year as a Visiting Scientist at the Massachusetts Institute of Technology (MIT), Cambridge. His research interests are focused on the area of coding techniques, primarily on soft-decision decoding algorithms, trellis structure of codes, codes defined on graphs, and iterative decoders. He has taught graduate courses in coding theory, information theory, and data transmission.

Dr. Kschischang was the recipient of the Ontario Premier's Research Excellence Award. From October 1997 to October 2000, he served as an Associate Editor for Coding Theory for the *IEEE TRANSACTIONS ON INFORMATION THEORY*. He also served as Technical Program Co-chair for the 2004 IEEE International Symposium on Information Theory.