Approaching Miscorrection-Free Performance of Product Codes with Anchor Decoding

Citation for the original published paper (version of record):
Häger, C., Pfister, H. (2018). Approaching Miscorrection-Free Performance of Product Codes with Anchor Decoding. IEEE Transactions on Communications, 66(7): 2797-2808.
http://dx.doi.org/10.1109/TCOMM.2018.2816073

(article starts on next page)

Approaching Miscorrection-Free Performance of Product Codes with Anchor Decoding

Christian Häger, *Member, IEEE* and Henry D. Pfister, *Senior Member, IEEE*

***Abstract*—Product codes (PCs) protect a two-dimensional array of bits using short component codes. Assuming transmission over the binary symmetric channel, the decoding is commonly performed by iteratively applying bounded-distance decoding to the component codes. For this coding scheme, undetected errors in the component decoding—also known as miscorrections—significantly degrade the performance. In this paper, we propose a novel iterative decoding algorithm for PCs which can detect and avoid most miscorrections. The algorithm can also be used to decode many recently proposed classes of generalized PCs such as staircase, braided, and half-product codes. Depending on the component code parameters, our algorithm significantly outperforms the conventional iterative decoding method. As an example, for double-error-correcting Bose–Chaudhuri–Hocquenghem component codes, the net coding gain can be increased by up to 0.4 dB. Moreover, the error floor can be lowered by orders of magnitude, up to the point where the decoder performs virtually identical to a genie-aided decoder that avoids all miscorrections. We also discuss post-processing techniques that can be used to reduce the error floor even further.**

***Index Terms*—Braided codes, fiber-optic communication, hard-decision decoding, iterative bounded-distance decoding, optical communication systems, product codes, staircase codes.**

I. Introduction

A product code (PC) is the set of all $n \times n$ arrays where each row and column in the array is a codeword in some linear component code $\mathcal{C}$ of length n [1]. Recently, a wide variety of related code constructions have been proposed, e.g., braided codes [2], half-product codes [3], [4], continuously-interleaved codes [5], half-braided codes [4], [6], and staircase codes [7]. All of these code classes have Tanner graph representations that consist exclusively of degree-2 variable nodes, i.e., each bit is protected by two component codes. We use the term generalized product codes (GPCs) to refer to such codes.

The component codes of a GPC typically correspond to Reed–Solomon or Bose–Chaudhuri–Hocquenghem (BCH) codes, which can be efficiently decoded via algebraic bounded-distance decoding (BDD). The overall GPC is then decoded by iteratively applying BDD to the component codes. This iterative coding scheme dates back to 1968 [8] and has been shown to offer excellent performance in practice. In particular for the binary symmetric channel (BSC) at high code rates, iterative decoding of GPCs with binary BCH component codes can achieve performance close to the channel capacity [7], [9], [10]. Moreover, the decoder data flow can be orders of magnitude lower than that of comparable low-density parity-check (LDPC) codes under message-passing decoding [7]. This facilitates decoder throughputs of tens or even hundreds of Gigabits per second. Indeed, GPCs are popular choices for high-bit-rate applications with limited soft information such as regional/metro optical transport networks [3]–[7], [9], [11]–[16]. Besides data transmission, GPCs are also used in storage applications [17]–[21].

PCs and GPCs can also be decoded using iterative soft-decision decoding (SDD), see, e.g., [22], [23]. In this case, however, an efficient hardware implementation at high data rates may be an issue due to the increased complexity of the component decoding. Moreover, the decoder data flow reduction shown in [7] is based on a syndrome compression effect which only occurs for BDD and not SDD. Indeed, we are unaware of any studies that show a favorable performance–complexity trade-off for PCs or GPCs under SDD compared to other code classes, e.g., LDPC codes. Therefore, we focus on iterative BDD in this paper.

For GPCs over the BSC, undetected errors in the component decoding—also known as miscorrections—significantly degrade the performance of iterative decoding. In particular, let $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{n}$, where $\boldsymbol{c}, \boldsymbol{n} \in \{0,1\}^n$ denote a component codeword and a random error vector, respectively. For a t-error-correcting component code $\mathcal{C}$, BDD yields the correct codeword $\boldsymbol{c} \in \mathcal{C}$ if and only if $d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{c}) = w_{\mathrm{H}}(\boldsymbol{n}) \leq t$, where d_{H} and w_{H} denote Hamming distance and weight, respectively. On the other hand, if $w_{\mathrm{H}}(\boldsymbol{n}) > t$, the decoding either fails or there exists another codeword $\boldsymbol{c}' \in \mathcal{C}$ such that $d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{c}') \leq t$. In the latter case, we say that a miscorrection occurs, in the sense that BDD is technically successful but the decoded codeword $\boldsymbol{c}'$ is not the correct one. Miscorrections are highly undesirable because they introduce additional errors (on top of channel errors) into the iterative decoding process. Moreover, from a theoretical perspective, miscorrections are notoriously difficult to analyze in an iterative decoding scheme [4], [7], [14], [24]–[28]. In fact, despite the widespread use in practice and to the best of our knowledge, no rigorous analytical results exist characterizing the finite-length performance of GPCs under iterative BDD over the BSC.

For specific code proposals in practical systems, the mis-

Parts of this paper have been presented at the 2017 European Conference on Optical Communication (ECOC), Gothenburg, Sweden.

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 749798. The work was also supported in part by the National Science Foundation (NSF) under Grant No. 1609327. Any opinions, findings, recommendations, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these sponsors.

C. Häger is with the Department of Electrical Engineering, Chalmers University of Technology, SE-41296 Gothenburg, Sweden and the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA (e-mail: christian.haeger@chalmers.se). H. D. Pfister is with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA (e-mail: henry.pfister@duke.edu).

correction problem is typically addressed by appropriately modifying the component code that is used to construct the GPC, see, e.g., [4], [7], [9], [16]. In particular, for binary t-error-correcting BCH codes, it is known that miscorrections occur approximately with probability $1/t!$ [4], [29]. In order to reduce this probability, one may employ a subcode of the original code [4], [7], extend the code [16], and/or apply code shortening [7], [16]. On the other hand, such modifications invariably lead to a code rate reduction. Moreover, even with a modified component code, miscorrections can still have a significant effect on the performance.

The main contribution in this paper is a novel iterative decoding algorithm for GPCs which can detect and avoid most miscorrections. The algorithm relies on so-called anchor codewords to resolve inconsistencies across component codes. This can lead to significant performance improvements in the waterfall and error-floor regimes, in particular when t is small, i.e., $t \in \{2, 3\}$. We also discuss the application of post-processing (PP) techniques [9], [21], [28], [30]–[32], which can be combined with the proposed algorithm to reduce the error floor even further. On the other hand, the practical usefulness of the proposed algorithm diminishes in cases where miscorrections do not significantly affect the performance, e.g., for doubly-extended BCH component codes with $t \geq 4$.

Decoder modifications that target miscorrections have been proposed before in the literature. Usually these modifications are minor and they are tailored to a specific GPC. As an example, staircase codes [7] can be seen as a convolutional-like (or spatially-coupled) version of PCs. The associated code array consists of an infinite number of square blocks that are arranged to look like a staircase [7, Fig. 4]. Decoding is facilitated by using a sliding window which comprises only a finite number of blocks. In order to reduce miscorrections, one may reject certain bit flips from component codes that are associated with the newest (most unreliable) block from the channel [33, p. 59]. However, simulations suggest that the performance gains using this approach are limited. On the other hand, the proposed anchor decoding can closely approach miscorrection-free performance when applied to staircase codes [34].

In [4], a decoder modification for PCs is suggested based on the observation that the miscorrection probability is reduced by a factor of n if only $t-1$ errors are corrected for a t-error-correcting component code. For large n, this is significant and the author thus proposes to only correct $t-1$ errors in the first iteration of iterative BDD. We will see later that this indeed gives some notable performance improvements. This trick can also be easily combined with the proposed algorithm.

The work in this paper is inspired by another comment made in [4] where it is mentioned that for PCs it may be desirable to "take special actions in case of conflicts" caused by miscorrections. In particular, it is suggested to use the number of conflicts for a particular component code as an indicator for the reliability of the component code. Besides this suggestion, no further details or results are provided. Our algorithm builds upon this idea and we develop a systematic approach that exploits component code conflicts and can be applied to an arbitrary GPC.

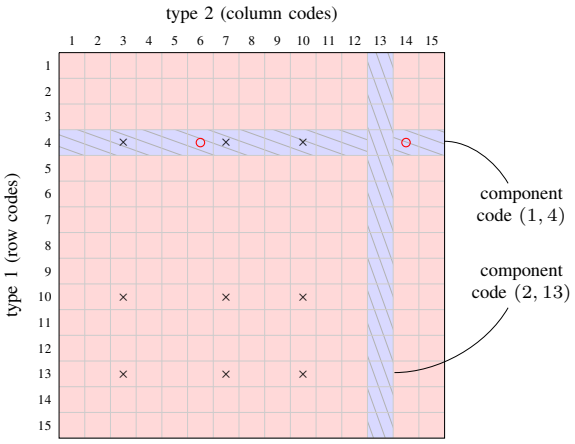

Fig. 1. PC array for a length-15 component code. Two particular component codewords are highlighted. The black crosses correspond to a minimal-size stopping set when $t = 2$. Red circles indicate a miscorrection, see Example 3.

The remainder of the paper is structured as follows. In Sec. II, we review PCs and the conventional iterative decoding scheme. We also give a brief overview of theoretical methods that have been proposed to analytically predict the performance. The proposed anchor decoding is described in Sec. III. In Sec. IV, simulation results are presented and discussed for various component code parameters. PP is discussed in Sec. V. We discuss some implementation details for the proposed algorithm in Sec. VI. Finally, the paper is concluded in Sec. VII.

II. Product Codes and Iterative Decoding

This paper focuses on PCs, with which we believe many readers are familiar. Other classes of GPCs are discussed separately in Sec. III-F.

A. Product Codes

Let $\boldsymbol{H} \in \mathbb{F}_2^{(n-k)\times n}$ be the parity-check matrix of a binary linear $(n, k, d_{\min})$ code $\mathcal{C}$, where n, k, and $d_{\min}$ are the code length, dimension, and minimum distance, respectively. A PC based on $\mathcal{C}$ is defined as

$$\mathcal{P}(\mathcal{C}) \triangleq \{\boldsymbol{X} \in \mathbb{F}_2^{n\times n} \mid \boldsymbol{H}\boldsymbol{X} = \boldsymbol{0}, \boldsymbol{H}\boldsymbol{X}^{\mathsf{T}} = \boldsymbol{0}\}. \tag{1}$$

It can be shown that $\mathcal{P}(\mathcal{C})$ is a linear $(n^2, k^2, d_{\min}^2)$ code. The codewords $\boldsymbol{X}$ can be represented as two-dimensional arrays. The two conditions in (1) enforce that the rows and columns in the array are valid codewords in $\mathcal{C}$.

We use a pair (i, j) to identify a particular component code and the corresponding codewords or received words. The first parameter $i \in \{1, 2\}$ refers to the component code type which can be either a row $(i = 1)$ or a column $(i = 2)$. The second parameter $j \in [n]$ enumerates the codes of a given type.

Example 1. The code array for a PC where the component code has length $n = 15$ is shown in Fig. 1. The row and column corresponding to component codes $(1, 4)$ and $(2, 13)$, respectively, are highlighted. △

For PCs, the coded bits can be identified by their two coordinates within the PC array. However, this way of specifying

bits does not generalize well to other classes of GPCs because the associated code array may have a different shape (or there may not exist an array representation at all). In order to keep the notation general, we therefore use the convention that a coded bit is specified by two component codes (i, j) and (k, l), i.e., four parameters (i, j, k, l) in total.

Example 2. The highlighted row and column in Fig. 1 intersect at the bit corresponding to $(1, 4, 2, 13)$ or $(2, 13, 1, 4)$. △

B. BCH Component Codes

We use binary t-error-correcting BCH codes as component codes, as well as their singly- and doubly-extended versions. Recall that a singly-extended BCH code is obtained through an additional parity bit, formed by adding (modulo 2) all coded bits $c_1, c_2, \ldots, c_{2^\nu-1}$ of the BCH code, where ν is the Galois field extension degree. On the other hand, a doubly-extended BCH code has two additional parity bits, denoted by c_{2^ν} and $c_{2^\nu+1}$, such that

$$c_1 + c_3 + \cdots + c_{2^\nu-1} + c_{2^\nu+1} = 0, \tag{2}$$

$$c_2 + c_4 + \cdots + c_{2^\nu-2} + c_{2^\nu} = 0, \tag{3}$$

i.e., the parity bits perform checks separately on odd and even bit positions. The overall component code has length $n = 2^\nu - 1 + e$, where $e \in \{0, 1, 2\}$ indicates either no ($e = 0$), single ($e = 1$), or double ($e = 2$) extension. In all three cases, the guaranteed code dimension is $k = 2^\nu - 1 - \nu t$. For $e = 0$, the guaranteed minimum distance is $d_{\min} = 2t + 1$. This is increased to $d_{\min} = 2t + 2$ for $e \in \{1, 2\}$. We use a triple (ν, t, e) to denote all BCH code parameters.

C. Bounded-Distance Decoding and Miscorrections

Consider the transmission of a component codeword $\boldsymbol{c} \in \mathcal{C}$ over the BSC with crossover probability p. The error vector introduced by the channel is denoted by $\boldsymbol{n}$, i.e., the components of $\boldsymbol{n}$ are i.i.d. Bernoulli(p) random variables. Applying BDD to the received word $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{n}$ results in

$$\mathrm{BDD}(\boldsymbol{r}) = \begin{cases} \boldsymbol{c} & \text{if } d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{c}) = w_{\mathrm{H}}(\boldsymbol{n}) \le t, \\ \boldsymbol{c}' \in \mathcal{C} & \text{if } w_{\mathrm{H}}(\boldsymbol{n}) > t \text{ and } d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{c}') \le t, \\ \texttt{FAIL} & \text{otherwise.} \end{cases} \tag{4}$$

In practice, BDD is implemented by first computing the syndrome $\boldsymbol{s}^\intercal = \boldsymbol{H}\boldsymbol{r}^\intercal = \boldsymbol{H}\boldsymbol{n}^\intercal \in \mathbb{F}_2^{n-k}$. Each of the 2^{n-k} possible syndromes is then associated with either an estimated error vector $\hat{\boldsymbol{n}}$, where $w_{\mathrm{H}}(\hat{\boldsymbol{n}}) \le t$, or a decoding failure. In the first case, the decoded output is computed as $\boldsymbol{r} + \hat{\boldsymbol{n}}$.

The second case in (4) corresponds to an undetected error or miscorrection.

Example 3. Consider the component code $(1, 4)$ in Fig. 1 and assume that the all-zero codeword $\boldsymbol{c} = \boldsymbol{0}$ is transmitted. The black crosses represent bit positions which are received in error, i.e., $n_i = 1$ for $i \in \{3, 7, 10\}$ and $n_i = 0$ elsewhere. For a component code with $t = 2$ and $e = 0$, we have $d_{\min} = 2t + 1 = 5$, i.e., there exists at least one codeword $\boldsymbol{c}' \in \mathcal{C}$ with Hamming weight 5. Assume we have $\boldsymbol{c}' \in \mathcal{C}$ with $c'_i = 1$ for $i \in \{3, 6, 7, 10, 14\}$ and $c'_i = 0$ elsewhere. Applying BDD to $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{n}$ then introduces two additional errors at bit positions 6 and 14. This is shown by the red circles in Fig. 1. △

Algorithm 1: Iterative BDD of product codes

```
for l = 1, 2, ..., ℓ do
    for i = 1, 2 do
        for j = 1, 2, ..., n do
            apply BDD to received component word (i, j)
    if valid codeword is found then
        break
```

Code extension reduces the probability of miscorrecting, at the expense of a slightly increased code length (and hence a small rate loss). To see this, let $e = 0$ and assume that $w_{\mathrm{H}}(\boldsymbol{n}) > t$. Consider now the decoding of a random syndrome $\boldsymbol{s}$ where the components in $\boldsymbol{s}$ are i.i.d. Bernoulli(0.5). In that case, the probability of miscorrecting is simply the ratio of the number of decodable syndromes and the total number of syndromes, i.e.,

$$\frac{\sum_{i=0}^{t}\binom{n}{i}}{2^{n-k}} = \frac{\sum_{i=0}^{t}\binom{n}{i}}{2^{\nu t}} \approx \frac{\frac{1}{t!}n^t}{n^t} = \frac{1}{t!}. \tag{5}$$

For $e = 1$ and $e = 2$, the total number of syndromes is increased to $2^{\nu t+e}$ and the miscorrection probability is thus reduced by a factor of 1/2 and 1/4, respectively. We note that this reasoning can be made precise, see, e.g., [4], [29].

Example 4. Consider the same scenario as in Example 3. If we use the singly-extended component code, the minimum distance is increased to $d_{\min} = 2t+2 = 6$. Assuming that there is no additional error in the last bit position, i.e., $n_{16} = 0$, the miscorrection illustrated in Example 3 can be detected because the parity-check equation involving the additional parity bit is not satisfied. △

Remark 1. As an alternative to extending the code, one may employ a subcode of the original BCH code. For example, the singly-extended BCH code behaves similarly to the even-weight subcode of the BCH code, which is obtained by multiplying its generator polynomial by $(1 + x)$. The doubly-extended BCH code behaves similarly to the BCH subcode where odd and even coded bits separately sum to zero. This subcode is obtained by multiplying the generator polynomial by $(1+x)^2$ and was used for example in the original staircase code construction [7]. Note that this subcode is not cyclic. Subcodes have a reduced code dimension k and hence lead to a similar rate loss as the code extension.

D. Iterative Bounded-Distance Decoding

We now consider the transmission of a codeword $\boldsymbol{X} \in \mathcal{P}(\mathcal{C})$ over the BSC with crossover probability p. The conventional iterative decoding procedure consists of applying BDD first to all rows and then to all columns. This is repeated ℓ times or until a valid codeword in $\mathcal{P}(\mathcal{C})$ is found. Pseudocode for the iterative BDD is given in Algorithm 1.

In order to analyze the bit error rate (BER) of PCs under iterative BDD, the prevailing approach in the literature is to assume that no miscorrections occur in the BDD of the

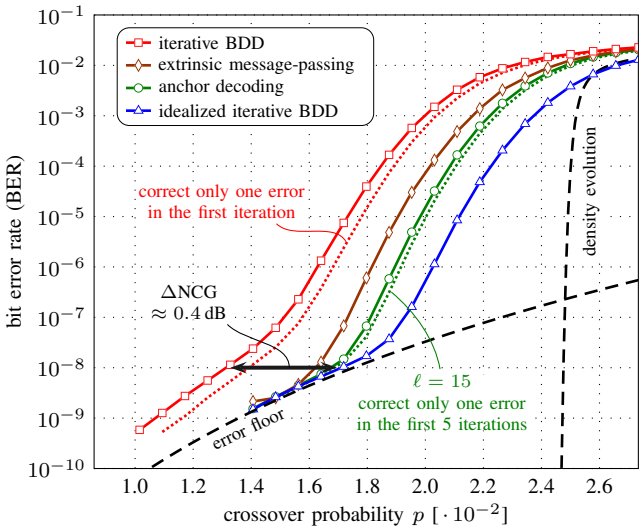

Fig. 2. Simulation results using different decoding schemes with $\ell = 10$ iterations. The component code is a BCH code with parameters $(7, 2, 1)$, i.e., an extended double-error-correcting BCH code with length $n = 2^7 = 128$.

component codes, see, e.g., [4], [14], [24]–[26]. To that end, we define

$$\mathrm{BDD}'(\boldsymbol{r}) = \begin{cases} \boldsymbol{c} & \text{if } d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{c}) = w_{\mathrm{H}}(\boldsymbol{n}) \leq t, \\ \texttt{FAIL} & \text{otherwise,} \end{cases} \tag{6}$$

which can be seen as an idealized version of BDD where a genie prevents miscorrections. Conceptually, this is similar to assuming transmission over the binary erasure channel (BEC) instead of the BSC [24].

Using (6) instead of (4), a decoding failure for a PC is related to the existence of a so-called core in an Erdős–Rényi random graph [4], [25]. This connection can be used to rigorously analyze the asymptotic performance as $n \to \infty$ using density evolution (DE) [4], [14], [25]. Moreover, the error floor can be estimated by enumerating stopping sets, also known as stall patterns. A stopping set is a subset of bit positions such that every component code with at least one bit in the set must contain at least $t + 1$ bits in the set. For PCs, a minimal-size stopping set involves $t+1$ rows and $t+1$ columns and has size $s_{\min} = (t + 1)^2$. For example, the black crosses shown in Fig. 1 form such a stopping set when $t = 2$. If we consider only stopping sets of minimal size, the BER can be approximated as

$$\mathrm{BER} \approx \frac{s_{\min}}{n^2} M p^{s_{\min}}, \tag{7}$$

for sufficiently small p, where $M = \binom{n}{t+1}^2$ is the total number of possible minimal-size stopping sets, also referred to as the stopping set's multiplicity. Unfortunately, if miscorrections are taken into account, DE and the error floor analysis are nonrigorous and become inaccurate.

Example 5. Consider a BCH code $\mathcal{C}$ with parameters $(7, 2, 1)$. The resulting PC $\mathcal{P}(\mathcal{C})$ has length $n^2 = 128^2 = 16384$ and code rate $R = k^2/n^2 \approx 0.78$. For $\ell = 10$ decoding iterations, the outcome of DE (see Appendix A for details) and the error floor analysis via (7) are shown in Fig. 2 by the dashed black lines. The analysis can be verified by performing idealized iterative BDD using (6). The results are shown by the blue line (triangles). However, the actual BER with true BDD (4) deviates significantly from the idealized decoding, as shown by the red line (squares). The BER can be moderately improved by treating the component codes as single-error-correcting in the first iteration, as suggested in [4]. This is shown by the red dotted line. △

There exist several approaches to quantify the performance loss due to miscorrections. In terms of the error floor, the authors in [7] derive an expression similar to (7) for staircase codes. To account for miscorrections, this expression is modified by introducing a heuristic parameter, whose value has to be estimated using Monte–Carlo simulations. In terms of asymptotic performance, the authors in [27] recently proposed a novel approach to analyze a GPC ensemble that is structurally related to staircase codes. The presented method is shown to give more accurate asymptotic predictions compared to the case where miscorrections are ignored.

Rather than analyzing the effect of miscorrections, the approach taken in this paper is to try to avoid them by modifying the decoding. In the next section, we give a detailed description of the proposed decoding algorithm. Its BER performance for the code parameters considered in Example 5 is shown in Fig. 2 by the green line (circles). The results are discussed in more detail in Sec. IV below.

Remark 2. Iterative BDD can be interpreted as a message-passing algorithm with binary "hard-decision" messages. The corresponding message-passing rule is *intrinsic*, in the sense that the outgoing message along some edge depends on the incoming message along the same edge. In [10], the authors propose an *extrinsic* message-passing algorithm based on BDD. The BER for this algorithm when applied to the PC in Example 5 is shown in Fig. 2 by the brown line (diamonds). Similar to the proposed algorithm, extrinsic message-passing provides significant performance improvements over iterative BDD. However, it is known that the decoder data-flow and storage requirements can be dramatically increased for message-passing decoding compared to iterative BDD [7]. One reason for this is that iterative BDD can leverage a syndrome compression effect by operating entirely in the syndrome domain. We show in Sec. VI that this effect also applies to the proposed algorithm. For extrinsic message-passing, it is an open question if an efficient syndrome domain implementation is possible. Due to this, we do not consider the extrinsic message-passing further in this paper.

III. Anchor Decoding

In the previous section, we have seen that there exists a significant performance gap between iterative BDD and idealized iterative BDD where a genie prevents miscorrections. Our goal is to close this gap. In order to do so, the key observation we exploit is that miscorrections lead to inconsistencies (or conflicts) across component codes. In particular, two component codes that protect the same bit may disagree on its value. In this section, we show how these inconsistencies can be used to (a) reliably prevent miscorrections and (b) identify

miscorrected codewords in order to revert the corresponding decoding decisions.

A. Preliminaries

The proposed decoding algorithm relies on so-called anchor codewords which are simply codewords that have been decoded successfully. No further additional corrections from other component codes are then allowed if this would overturn the decision of an anchor and lead to a conflict. However, some anchors may be miscorrected. Therefore, the decoding decisions of anchors are reversed or backtracked if too many other component codes are in conflict with a particular anchor. In order to make this more precise, we start by introducing some additional concepts and notation in this subsection.

First, consider the BDD of a single received (component) word. We explicitly regard this component decoding as a two-step process. In the first step, the actual decoding is performed and the outcome is either an estimated error vector $\hat{\boldsymbol{n}}$ or a decoding failure. In the second step, error-correction is performed by flipping the bits corresponding to the error locations. These two steps are separated in order to perform consistency checks (described below). These checks are used to determine if the error-correction step should be applied.

It is more convenient to specify the estimated error vector $\hat{\boldsymbol{n}}$ in terms of a set of error locations. For component code (i, j), this set is denoted by $\mathcal{E}_{i,j}$, where $|\mathcal{E}_{i,j}| \leq t$. The set comprises those component codes that are affected by the bit flips implied by $\hat{\boldsymbol{n}}$.

Example 6. Consider again the scenario described in Example 3, where the received word $(1, 4)$ shown in Fig. 1 is miscorrected with an estimated error vector $\hat{\boldsymbol{n}}$ such that $\hat{n}_i = 1$ for $i \in \{6, 14\}$ and $\hat{n}_i = 0$ elsewhere. The two affected component codes correspond to columns specified by $(2, 6)$ and $(2, 14)$ and, hence, the corresponding set of error locations is given by $\mathcal{E}_{1,4} = \{(2, 6), (2, 14)\}$. △

Remark 3. It may seem more natural to define $\mathcal{E}_{i,j}$ in terms of the bit positions of the BCH code, e.g., $\mathcal{E}_{1,4} = \{6, 14\}$ in the previous example. However, defining $\mathcal{E}_{i,j}$ in terms of the affected component codes leads to a more succinct description of the proposed algorithm. Moreover, this definition also generalizes more easily to other classes of GPCs, see Sec. III-F below.

Furthermore, we denote by $\mathcal{L}_{i,j}$ the set of component codes that are in conflict with code (i, j) due to miscorrections. Lastly, each component code has an associated status to signify its current state. The status values range from 0 to 3 with the following meaning:

- 0: anchor
- 1: eligible for BDD
- 2: BDD failed in last iteration
- 3: frozen

The precise use of the status and the transition rules between different status values are described in the following. We remark that the reliability of a component code is implicitly encoded both in the status and in the number of conflicts $|\mathcal{L}_{i,j}|$. Roughly speaking, frozen component codes are deemed highly unreliable, whereas the reliability of an anchor depends on the number of conflicts.

Algorithm 2: Main routine of anchor decoding

```
1  if (i,j).status = 1 then
2      R ← (i,j).decode        /* R indicates success (1) or failure (0) */
3      if R = 1 then
4          for each (k,l) ∈ E_{i,j} do          /* consistency checks */
5              if (k,l).status = 0 then          /* conflict with anchor */
6                  if |L_{k,l}| ≥ δ then
7                      add (k,l) to B            /* mark for backtracking */
8                  else
9                      (i,j).status ← 3          /* freeze */
10                     add (k,l) to L_{i,j}      /* save the conflict */
11                     add (i,j) to L_{k,l}      /* save the conflict */
12         if (i,j).status = 1 then              /* if not frozen */
13             for each (k,l) ∈ E_{i,j} do
14                 error-correction for (i,j,k,l)   /* see Alg. 4 */
15             (i.j).status ← 0                  /* code becomes an anchor */
16             for each (k,l) ∈ B do
17                 backtrack anchor (k,l)        /* see Alg. 3 */
18     else
19         (i,j).status ← 2                      /* indicate decoding failure */
```

B. Main Algorithm Routine

The algorithm is initialized by setting the status of all component codes to 1. We then iterate ℓ times over the rows and columns in the same fashion as in Algorithm 1, but replacing line 4 with lines 1–19 in Algorithm 2. Algorithm 2 represents the main routine of the proposed anchor decoding. It can be divided into 4 steps which are described in the following.[1]

Step 1 (Lines 1–3): If the component code is eligible for BDD, i.e., its status is 1, we proceed to decode the corresponding received word. If the decoding is successful, we proceed to the next step, otherwise, the status is set to 2 and we skip to the next component code.

Step 2 (Lines 4–11): For each found error location $(k, l) \in \mathcal{E}_{i,j}$, a consistency check is performed. That is, one checks if the implied component code (k, l) corresponds to an anchor. If so, $|\mathcal{L}_{k,l}|$ is the number of conflicts that this anchor is already involved in. This number is then compared against a threshold δ. If $|\mathcal{L}_{k,l}| \geq \delta$, the anchor (k, l) is deemed unreliable and it is marked for backtracking by adding it to the backtracking set $\mathcal{B}$. On the other hand, if $|\mathcal{L}_{k,l}| < \delta$, the component code (i, j) is frozen by changing its status to 3. Moreover, the conflict between the (now frozen) code and the anchor is stored by modifying the respective sets $\mathcal{L}_{i,j}$ and $\mathcal{L}_{k,l}$. Frozen component codes are always skipped (in the loop of Algorithm 1) for the rest of the decoding unless either the conflicting anchor is backtracked or any bits in the frozen received word change.

Step 3 (Lines 12–15): If the component code (i, j) still has status 1, the bit flips implied by $\mathcal{E}_{i,j}$ are consistent with all reliable anchors, i.e., anchors that are involved in δ or fewer other conflicts. If that is the case, the algorithm proceeds by applying the error-correction step for component

[1] Similar to the conventional iterative BDD, it is not guaranteed that the algorithm always terminates at a valid codeword for $\ell \to \infty$.

code (i,j), i.e., the bits (i,j,k,l) corresponding to all error locations $(k,l) \in \mathcal{E}_{i,j}$ are flipped. The error-correction step is implemented in Algorithm 4 and described in detail in Section III-E. Afterwards, the code (i,j) becomes an anchor by changing its status to 0.

Step 4 (Lines 16–17): The last step consists of backtracking all anchors in the set $\mathcal{B}$ (if there are any). Roughly speaking, backtracking involves the reversal of all previously applied bit flips of the corresponding anchor. Moreover, the backtracked code loses its anchor status. The backtracking routine is implemented in Algorithm 3 and described in more detail in Sec. III-D below.

C. Examples

We now illustrate the above steps with the help of two examples. For both examples, a component code with error-correcting capability $t = 2$ is assumed. Moreover, the conflict threshold is set to $\delta = 1$.

Example 7. Consider the scenario depicted in Fig. 3(a). Assume that we are at $(i,j) = (1,4)$, corresponding to a row with status 1 and four attached errors shown by the black crosses. The received word is assumed to be miscorrected with $\mathcal{E}_{1,4} = \{(2,5),(2,13)\}$ shown by the red circles. The column code $(2,5)$ is assumed to have status 2 (i.e., BDD failed in the previous iteration with three attached errors) and therefore the first consistency check is passed. However, assuming that the column $(2,13)$ is an anchor without any other conflicts, i.e., $\mathcal{L}_{2,13} = \emptyset$, the code $(1,4)$ is frozen during step 2. Hence, no bit flips are applied to the received word and the miscorrection is prevented. The conflict is stored by updating the two conflict sets as $\mathcal{L}_{1,4} = \{(2,3)\}$ and $\mathcal{L}_{2,3} = \{(1,4)\}$, respectively. △

Example 8. Consider the scenario depicted in Fig. 3(b), where we assume that row $(1,4)$ is a miscorrected anchor without conflicts (i.e., $\mathcal{L}_{1,4} = \emptyset$) and error locations $\mathcal{E}_{1,4} = \{(2,5),(2,13)\}$. Assume that we are at $(i,j) = (2,5)$. The column $(2,5)$ has status 1 and two attached error. Thus, BDD is successful with $\mathcal{E}_{2,5} = \{(1,4),(1,10)\}$. During step 2, the column $(2,5)$ is, however, frozen because there is a conflict with anchor $(1,4)$. After freezing the column, we have $\mathcal{L}_{1,4} = \{(2,5)\}$ and $\mathcal{L}_{2,5} = \{(1,4)\}$. We skip to the next component code $(2,6)$, which has status 1. Again, BDD is successful with $\mathcal{E}_{2,6} = \{(1,4)\}$. The implied bit flip is inconsistent with the anchor $(1,4)$. However, since this anchor is already in conflict with $(2,5)$ (and, hence, $|\mathcal{L}_{1,4}| = 1 = \delta$), the anchor is marked for backtracking and the error-correction step for bit $(2,6,1,4)$ will be applied. △

D. Backtracking

In Example 8, we have encountered a scenario that leads to the backtracking of a miscorrected anchor. The actual backtracking routine is implemented in Algorithm 3. First, all conflicts caused by the anchor are removed by modifying the respective conflict sets. Note that all component codes $(k,l) \in \mathcal{L}_{i,j}$ for anchor (i,j) necessarily have status 3, i.e., they are frozen. After removing conflicts, such component codes may be conflict-free, in which case their status is changed to 1. After this, all previously applied bit flips are reversed. In order to perform this operation, it is necessary to store the set $\mathcal{E}_{i,j}$ for each anchor. Finally, the anchor status is lost. In principle, the new status can be chosen to be either 1 or 3. However, backtracked anchors are likely to have miscorrected. We therefore prefer to freeze the component code by setting its status to 3 after the backtracking.

Algorithm 3: Backtracking anchor codeword (i,j)

```
1 for each (k,l) ∈ L_{i,j} do                          /* remove conflicts */
2     remove (k,l) from L_{i,j}
3     remove (i,j) from L_{k,l}
4     if L_{k,l} is empty then                         /* no more conflicts */
5         (k,l).status ← 1                             /* unfreeze */
6 for each (k,l) ∈ E_{i,j} do
7     error-correction step for (i,j,k,l)              /* see Alg. 4 */
8 (i,j).status ← 3                                     /* freeze */
```

Algorithm 4: Error-correction step for bit (i,j,k,l)

```
1 if not ((i,j).status = 0 and (k,l).status = 0) then
2     flip the bit (i,j,k,l)
3     if (k,l).status = 2 then
4         (k,l).status ← 1
5     else if (k,l).status = 3 then
6         (k,l).status ← 1
7         for each (k',l') ∈ L_{k,l} do                /* remove conflicts */
8             remove (k,l) from L_{k',l'}
9             remove (k',l') from L_{k,l}
```

Remark 4. Since we do not know if an anchor is miscorrected or not, it is also possible that we mistakenly backtrack "good" anchors. Fortunately, this is unlikely to happen for long component codes because the additional errors due to miscorrections are approximately randomly distributed within the component word [4]. More precisely, consider the case where two miscorrected component codes both introduce t additional errors. Assuming that these errors are randomly distributed among the n bit positions, the probability that any of the error locations overlap is $1 - \prod_{i=0}^{t-1}(1 - \frac{t}{n-i}) \approx t^2/n$, where the approximation is valid for large n.

E. Error-correction Step

The error-correction step is implemented in Algorithm 4. The input is a parameter tuple (i,j,k,l) where (i,j) is the component code that initiated the bit flip and (k,l) is the corresponding component code affected by it. Note that Algorithm 4 can be reached from both the main routine (Algorithm 2, lines 13–14) and as part of the backtracking process (Algorithm 3, lines 6–7). If the algorithm is reached via backtracking, it is possible that the affected code (k,l) is now an anchor. In this case, we use the convention to trust the anchor's decision about the bit (i,j,k,l) and not apply any changes.[2] In all other cases, apart from actually flipping the bit (i,j,k,l) (line 2), error-correction triggers a status change (lines 3–9). If the bit flip affects a frozen component code, the

[2]A different, but potentially more complex approach would be to generalize the backtracking procedure and allow for multiple codewords to be backtracked in this case.

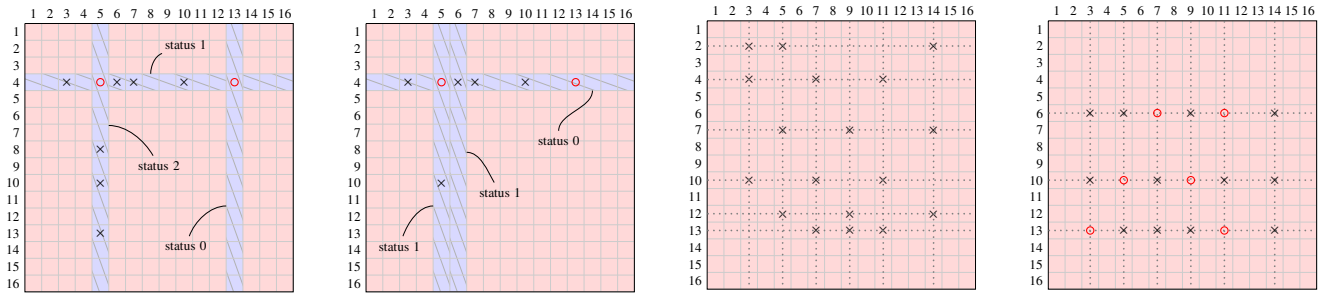

Fig. 3. (a) Error/status configuration illustrating how anchor codewords prevent miscorrections (see Example 7) , (b) Error/status configuration illustrating how backtracking is triggered for miscorrected anchors (see Example 8), (c) A minimal-size stopping set after algebraic-erasure post-processing (PP) assuming no miscorrections for $t = 2$ and $e \in \{1, 2\}$, (d) Illustration of a dominant error event for anchor decoding for $t = 2$ and $e \in \{1, 2\}$ in the error-floor regime.

code is unfrozen and we remove the conflicts that this code is involved in.

F. Generalized Product Codes

In principle, anchor decoding can be applied to an arbitrary GPC. Indeed, Algorithms 2–4 are independent of the underlying GPC. The global code structure manifests itself only through the set of error locations $\mathcal{E}_{i,j}$. This set is defined in terms of the affected component codes, which implicitly uses the code structure.

Compared to PCs, the main difference for other GPCs is that Algorithm 1 has to be replaced by a version that is appropriate for the specific GPC. For anchor decoding, Algorithm 1 simply specifies the order in which the received words of the component codes are traversed during the iterative decoding.

Example 9. We have considered anchor decoding of staircase codes in the conference version of this paper [34]. In that case, Algorithm 1 is replaced by a window decoding schedule, see [34, Alg. 1].[3] △

Note that staircase codes have more than two types of component codes. In particular, the component code types indicate the position of the component codes in the staircase code sliding window. For GPCs that do not admit a description in terms of a finite number of types (e.g., tightly-braided block codes), one may simply use the convention that each component code forms its own type.

Anchor-based decoding can also be applied to GPCs that are based on component codes with different lengths and/or error-correcting capabilities. For example, PCs can be defined such that different component codes are used to protect the rows and columns of the code array. More generally, the component codes may even vary across rows and columns, leading to irregular PCs [35], [36].

While Algorithms 2–4 are agnostic to changes in the overall code structure, it may be beneficial to adopt different conflict thresholds δ for the component codes in a generalized product code or when using a mixture of component codes with different error-correcting capabilities. For example, for staircase codes or other convolutional-like GPCs, different conflict thresholds may be adopted depending on the component code position in the sliding window to reflect the different component code reliabilities. A concrete example where different conflict thresholds are useful is discussed in Section IV-C.

[3] [34] contains two typos. First, the for-loop in [34, Alg. 1] over i should start from $W - 1$ (not W). Second, the window size used for the simulations is $W = 9$ (not $W = 8$).

IV. Simulation Results

In this section, we present and discuss simulation results assuming different BCH component codes. For the conflict threshold, we tested different values $\delta \in \{0, 1, 2, 3\}$ for a wide variety of component code parameters ($\nu \in \{7, 8\}, t \in \{2, 3, 4\}, e \in \{0, 1, 2\}$) and BSC crossover probabilities. In all cases, $\delta = 1$ was found to give the best performance. Hence, $\delta = 1$ is assumed in the following. The only exception are the PCs discussed in Section IV-C which are based on different component codes for the rows and columns.

Remark 5. For the staircase codes considered in [34], a conflict threshold of $\delta = 1$ for all component codewords (regardless of their position in the sliding window) also gave the best performance.

A. BCH Codes with $t = 2$

Double-error-correcting BCH codes are of particular interest because they can be decoded very efficiently in hardware [16], [37]. On the other hand, the resulting PCs suffer from relatively high error floors and their performance is significantly affected by miscorrections. This was shown in Example 5 in Sec. II-D. Therefore, performance improvements, in particular in the error floor regime, are highly relevant for practical applications with stringent reliability constraints.

Recall that Example 5 uses a BCH component code with parameters $(7, 2, 1)$ and $\ell = 10$ decoding iterations. For these parameters, the BER of anchor decoding is shown in Fig. 2 by the green line (circles). The algorithm closely approaches the performance of the idealized iterative BDD in the waterfall regime. Moreover, virtually miscorrection-free performance is achieved in the error-floor regime. The remaining gap between the idealized decoder and the anchor decoder is due to instances where too many miscorrections occur. We also note that the code parameters for this example were chosen such that the error floor is high enough to be within the reach of software simulations. In certain applications, e.g., optical transport networks, lower BERs ($< 10^{-15}$) may be required.

In this case, other code parameters have to be used to reduce the error floor below the application requirements.

In order to quantify the performance gain with respect to iterative BDD, we use the net coding gain (NCG). To that end, assume that a coding scheme with code rate R achieves a BER of p_{out} on a BSC with crossover probability p. The NCG (in dB) is then defined as

$$\text{NCG} \triangleq 10 \log_{10} \left(R \frac{(Q^{-1}(p_{\text{out}}))^2}{(Q^{-1}(p))^2} \right). \tag{8}$$

The NCG assumes antipodal binary modulation over an additive Gaussian noise channel and measures the difference in required E_b/N_0 between uncoded transmission and coded transmission using the coding scheme under consideration. As an example, it can be seen from Fig. 2 that the iterative BDD and the anchor decoding achieve a BER of 10^{-8} at approximately $p = 1.31 \cdot 10^{-2}$ and $p = 1.69 \cdot 10^{-2}$, respectively. The code rate in both cases is $R = 0.78$. Hence, the respective NCGs are given by $6.96\,\text{dB}$ and $7.37\,\text{dB}$. The proposed algorithm thus achieves a NCG improvement of approximately $\Delta\text{NCG} = 0.4\,\text{dB}$, as indicated by the arrow in Fig. 2.

As suggested in [4], correcting only one error in the first iteration of iterative BDD gives some moderate performance improvements. This trick can also be used in combination with the anchor decoding. In that case, a decoding failure (in line 2 of Algorithm 2) also occurs when BDD is successful but $|\mathcal{E}_{i,j}| = 2$. Since the status of such component codes is set to 2, it is important to reset the status to 1 after the first iteration in order to continue the decoding with the full error-correction capability. For the PC in Example 5 with $\ell = 10$ iterations, we found that correcting only one error in the first iteration does not lead to noticeable performance improvements for the anchor decoding. Some small improvements can be obtained, however, by increasing the total number of iterations to $\ell = 15$ and correcting only one error in the first 5 iterations. This is shown by the green dotted line in Fig. 2. It is important to stress that without the gradual increase of t, the BER for $\ell = 10$ and $\ell = 15$ is virtually the same. Thus, the improvement shown in Fig. 2 is indeed due to the artificial restriction of the error-correcting capability.

Next, we provide a direct comparison with the results presented in [16]. In particular, the authors propose a hardware architecture for a PC that is based on a BCH component code with parameters $(8, 2, 1)$. The BCH code is further shortened by 61 bits, leading to an effective length of $n = 195$ and dimension $k = 178$. The shortening gives a desired code rate of $R = k^2/n^2 = 178^2/195^2 \approx 0.833$. The number of decoding iterations is set to $\ell = 4$. For these parameters, BER results are shown in Fig. 4 for iterative BDD, anchor decoding, and idealized iterative BDD (labeled "w/o PP"). As before, the outcome of DE and the error floor prediction via (7) are shown by the dashed black lines as a reference. Compared to the results shown in Fig. 2, the anchor decoding approaches the performance of the idealized iterative BDD even closer and virtually miscorrection-free performance is achieved for BERs below 10^{-7}. This can be attributed to the quite extensive code shortening, which reduces the probability of miscorrecting compared to an unshortened component code.

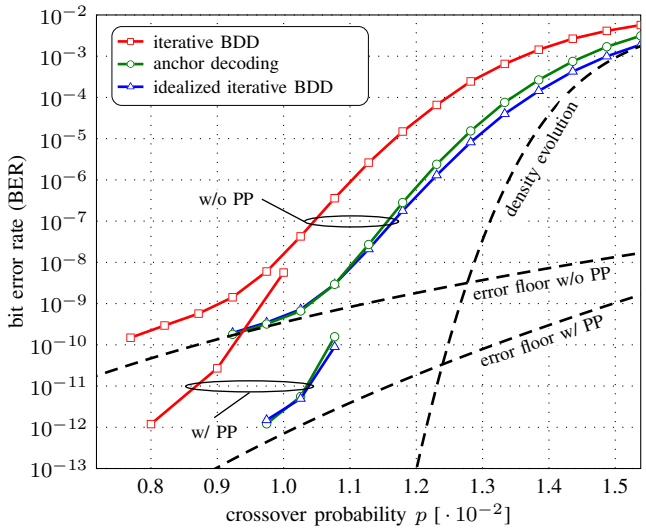

Fig. 4. Simulation results assuming a shortened $(195, 178)$ BCH component code with parameters $(8, 2, 1)$. The resulting product code is considered in [16]. Simulation data for iterative BDD including bit-flip-and-iterate post-processing (PP) was provided by the authors of [16].

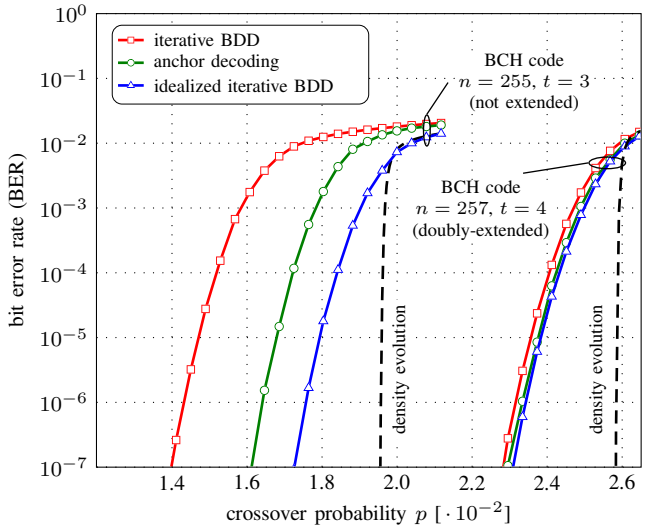

Fig. 5. Simulation results assuming two different BCH component codes with parameters $(8, 3, 0)$ and $(8, 4, 2)$.

B. BCH Codes with $t > 2$

For BCH component codes with error-correcting capability larger than 2, the error floor is generally out of reach for our software simulations. Hence, we focus on the performance improvements that can be obtained in the waterfall regime.

In Fig. 5, we show the achieved BER for two different PCs. The first PC is based on a BCH component code with parameters $(8, 3, 0)$. For these parameters, miscorrections significantly degrade the performance. Consequently, there is a large performance gap between iterative BDD and idealized iterative BDD. The anchor decoding partially closes this gap and achieves a NCG improvement of around $0.23\,\text{dB}$ over

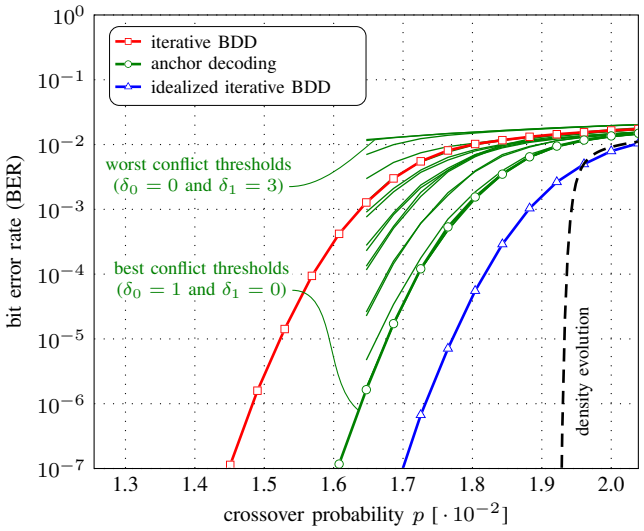

Fig. 6. Simulation results assuming that the row and column BCH component codes have different parameters $(8, 4, 0)$ and $(8, 2, 0)$, respectively. Solid lines without markers correspond to different combinations of conflict thresholds $\delta_0, \delta_1 \in \{0, 1, 2, 3\}$.

iterative BDD at BER $= 10^{-7}$. The second PC is based on a BCH component code with parameters $(8, 4, 2)$. The miscorrection probability is reduced approximately by a factor of 16 compared to the first PC. Hence, there is only a small gap between iterative BDD and idealized iterative BDD. The anchor decoding manages to close this gap almost completely. The NCG improvement at BER $= 10^{-7}$ in this case is, however, limited to around 0.01 dB.

C. Different Row and Column Component Codes

Lastly, we consider PCs where the rows and columns are protected by component codes with different error-correcting capabilities. In particular, we assume that the row and column codes are BCH codes with parameters $(8, 4, 0)$ and $(8, 2, 0)$, respectively. The miscorrection probabilities in this case are roughly $1/4! \approx 0.0471$ for the rows and $1/2$ for the columns. Since the decisions of the row codes are highly unreliable compared to those of the column codes, one might expect that adopting different conflict thresholds may be beneficial. To confirm this intuition, we denote by δ_0 and δ_1 the threshold for rows and columns, respectively. An exhaustive search is performed over all possible combinations where $\delta_0, \delta_1 \in \{0, 1, 2, 3\}$. The results are shown in Fig. 6, where $\ell = 10$ decoding iterations are assumed. It can be seen that the best performance is indeed achieved when different conflict thresholds are adopted. In particular, we have $\delta_0 = 1$ and $\delta_1 = 0$, i.e., columns should be immediately backtracked in case of conflicts.

V. Post-Processing

If the anchor decoding terminates unsuccessfully, one may use some form of PP in order to continue the decoding. In this section, we discuss two PP techniques, which we refer to as *bit-flip-and-iterate* PP and *algebraic-erasure* PP. Both techniques have been studied before in the literature as a means to lower the error-floor for various GPCs assuming the conventional iterative BDD [9], [21], [28], [30]–[32].

A. Methods

Let $\mathcal{F}_i$ denote the set of failed component codes of type i after an unsuccessful decoding attempt. That is, $\mathcal{F}_1$ and $\mathcal{F}_2$ are, respectively, rows and columns that still have nonzero syndrome after ℓ decoding iterations. The intersection of these rows and columns defines a set of bit positions according to

$$\mathcal{I} = \{(i, j, k, l) \,|\, (i, j) \in \mathcal{F}_1 \text{ and } (k, l) \in \mathcal{F}_2\}. \tag{9}$$

For *bit-flip-and-iterate* PP, the bits in the intersection (9) are first flipped, after which the iterative decoding is resumed for one or more iterations. Bit-flip-and-iterate PP has been applied to PCs [30], [31], half-product codes [32], and staircase codes [28]. For *algebraic-erasure* PP, the bits in the intersection (9) are instead treated as erasures. An algebraic erasure decoder is then used to recover the bits. Assuming that there are no miscorrected codewords, algebraic-erasure PP provably succeeds as long as either $|\mathcal{F}_1| < d_{\text{min}}$ or $|\mathcal{F}_2| < d_{\text{min}}$ holds. This type of PP has been applied to braided codes in [9] and to half-product codes in [21].

B. Post-Processing for Anchor Decoding

In principle, the above PP techniques can be applied after the anchor decoding without any changes. However, it is possible to improve the effectiveness of algebraic-erasure PP by exploiting additional information that is available for the anchor decoding. In particular, recall that it is necessary to keep track of the error locations of anchors in case they are backtracked. If the anchor decoding fails, these error locations can be used for the purpose of PP as follows. Assume that we have determined the sets $\mathcal{F}_1$ and $\mathcal{F}_2$. Then, one can check for anchors that satisfy the condition

$$\mathcal{E}_{i,j} \subset \mathcal{F} \quad \text{and} \quad |\mathcal{E}_{i,j}| = t, \tag{10}$$

where $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$, i.e., anchors that have corrected t errors, where the error locations overlap entirely with the set of failed component codes. For the algebraic-erasure PP, we found that it is beneficial to include such anchors into the respective sets $\mathcal{F}_1$ and $\mathcal{F}_2$ (even though they have zero syndrome). This is because the associated component codewords are likely to be miscorrected.

Note that additional component codes should only be included into the sets $\mathcal{F}_1$ and $\mathcal{F}_2$ as long as $|\mathcal{F}_1| < d_{\text{min}}$ or $|\mathcal{F}_2| < d_{\text{min}}$ remain satisfied. Therefore, if there are more component codes satisfying (10) than allowed by these constraints, we perform the inclusion on a random basis.

Remark 6. For bit-flip-and-iterate PP, we found that the same strategy may, in fact, degrade the performance, in particular if the number of decoding iterations is relatively low. For small ℓ, the decoding often terminates unsuccessfully with only a few remaining errors left. These errors are then easily corrected using the conventional bit-flip-and-iterate PP. In that case, it is counterproductive to include additional component codes into the sets $\mathcal{F}_1$ and $\mathcal{F}_2$.

C. Example for BCH Codes with $t = 2$

Consider again the PC based on the $(195, 178)$ shortened BCH code with parameters $(8, 2, 1)$ studied in [16] (see Fig. 4). The authors in [16] also consider the application of bit-flip-and-iterate PP in order to reduce the error floor. The simulation data was provided to us by the authors and the results are reproduced[4] for convenience in Fig. 4 by the red line (squares) labeled "w/ PP". For the anchor decoding we propose to use instead the algebraic-erasure PP as described in the previous subsection. In order to estimate its performance, we first consider algebraic-erasure PP for the idealized iterative BDD without miscorrections. In this case, the dominant stopping set is of size 18 and involves 6 rows and 6 columns. An example of this stopping set is shown by the black crosses in Fig. 3(c), where the involved rows and columns are indicated by the dotted lines. It is pointed out in [28] that the multiplicity of such a stopping set can be obtained by using existing counting formulas for the number of binary matrices with given row and column weight [38]. In particular, there exist $297,200$ binary matrices of size 6×6 with uniform row and column weight 3 [38, Table 1]. This gives a multiplicity of $M = 297,200\binom{n}{6}^2$ for this stopping set. The error floor can then be estimated using (7) with $s_{\min} = 18$. This is shown in Fig. 4 by the dashed black line labeled "error floor w/ PP" and can be verified using the idealized iterative BDD including PP. The performance of anchor decoding with algebraic-erasure PP is also shown in Fig. 4 and virtually overlaps with the performance of idealized iterative BDD for BERs below 10^{-11}. Overall, the improvements translate into an additional NCG of around $0.2\,\text{dB}$ at a BER of 10^{-12} over iterative BDD with bit-flip-and-iterate PP.

Without the modification described in Sec. V-B, the performance of algebraic-erasure PP under anchor decoding would be slightly decreased. Indeed, we found that a dominant error event of the anchor decoding for $t = 2$ and $e \in \{1, 2\}$ is such that 3 row (or column) decoders miscorrect towards the same estimated error pattern $\hat{\boldsymbol{n}}$ of weight 6. This scenario is illustrated in Fig. 3(d). We did not encounter similar error events for the conventional iterative BDD. This indicates that the anchor decoding introduces a slight bias towards an estimated error pattern, once it is "anchored". For the error event shown in Fig. 3(d), 6 columns are not decodable, whereas all rows are decoded successfully with a zero syndrome. This implies that the set $\mathcal{F}_2$ is empty and therefore the bit intersection (9) is empty as well. Hence, conventional PP (both bit-flip-and-iterate and algebraic-erasure) would fail. On the other hand, with high probability condition (10) holds for all 3 miscorrected row codewords.

VI. Implementation Details

In this section, we discuss some implementation details of anchor decoding. In particular, we focus on the syndrome compression effect discussed in [7] and some high-level implementation differences between anchor decoding and iterative BDD in the following.

[4]The same results are shown in [16, Fig. 3].

Remark 7. In principle, it may also be desirable to quantify the increase in complexity of anchor decoding compared to iterative BDD, e.g., in terms of the additional number of operations or computations. However, this is a nontrivial task because it is not obvious how to properly define these quantities. To illustrate the difficulty, consider for example the implementation of Algorithm 1 (for simplicity without the exit criterion) as a baseline. A straightforward approach requires $2\ell n$ algebraic decodes, which one may define as an operation. On the other hand, using a status flag to indicate a syndrome change with respect to the previous decoding attempt can reduce this number to vn, where v ranges from 3–4 in practice. Implementing the status flag would also require additional operations and memory. The status flag idea is mentioned in [7] in a footnote, but it is not used for the architecture in [31] (possibly due to the low number of iterations performed). Therefore, there arises ambiguity even when attempting to quantify the baseline complexity of Algorithm 1. Thus, in order to obtain an accurate estimate for the complexity, it would be necessary to design a full hardware architecture in both cases, which is beyond the scope of this paper.

A. Syndrome Domain Implementation

One of the main advantages of iterative BDD compared to message-passing decoding is the significantly reduced decoder data flow [7]. We argue that similar considerations also apply in the case of anchor decoding. To that end, we start by reviewing the product decoder architecture in [7], which consists of three main parts or units (cf. [7, Fig. 3]): a data storage unit for the product code array, a syndrome storage unit, and a BCH component decoder unit. The main contributors to the decoder data flow (in bits/s) between these units are as follows:

- Initially, the syndromes for all received component words have to be computed and stored in the syndrome storage based on the received data bits.
- During iterative decoding, syndromes are loaded from the syndrome storage and used by the BCH component decoder unit for the component decoding.
- After a successful component decoding, the syndromes in the syndrome storage are updated based on the found error locations. Moreover, the corresponding bits in the data storage unit are flipped.

A key aspect of this architecture is that information between component codes is exchanged entirely through their syndromes. This is very efficient at high code rates: in this case, syndromes can be seen as a compressed representation of the received component word. Moreover, each successful component decoding affects at most t syndromes.

In principle, anchor decoding can also operate entirely in the syndrome domain, thereby leveraging the same syndrome compression effect as iterative BDD. In particular, the initial syndrome computation phase can be performed in the same fashion as for iterative BDD. The syndrome loading occurs before executing line 2 of Algorithm 2 and syndrome updates are triggered due to line 2 of Algorithm 4.

B. Implementation Differences Compared to Iterative BDD

While the syndrome domain implementation can be kept intact, in the following we comment on some of the implementation differences between iterative BDD and anchor decoding.

1) Component code status: Anchor-based decoding uses a status value for each component code. Status changes occur after BDD (lines 15 and 19 in Algorithm 2), after backtracking (line 8 in Algorithm 3), and after applying bit flips (lines 3–6 in Algorithm 4). This leads to an apparent complexity increase compared to a straightforward implementation of iterative BDD. On the other hand, even for iterative BDD, it is common to introduce some form of status information for each component code. For example, a status flag is often used to indicate if the syndrome for a particular component code changed since the last decoding attempt [7]. This is done in order to avoid decoding the same syndrome multiple times. The product decoder architecture in [16] also features a status flag to indicate the failure of a particular component code in the last decoding iteration. Therefore, the slightly more involved status handling for the anchor decoding should not lead a drastic complexity increase compared to practical implementations of iterative BDD.

2) Error locations: Additional storage is needed to keep track of the error locations $\mathcal{E}_{i,j}$ for each anchor codeword (i, j) in case of backtracking (lines 6–7 in Algorithm 3). Since each individual error location can be specified using $\lceil \log_2(n) \rceil$+1 bits, the total extra storage for all error locations required is $2nt(\lceil \log_2(n) \rceil$+1) bits.

3) Conflict sets: Additional storage is also needed to store the conflicts between component codewords. For a t-error-correcting component code, there can be at most t conflicts for each frozen component code. Moreover, for a conflict threshold of $\delta = 1$, it is sufficient to keep track of a single conflict per anchor codeword. Taking the larger of these two values, the conflict set size therefore has to be t. The extra storage required is thus the same as for the error locations, i.e., $2nt(\lceil \log_2(n) \rceil + 1)$ bits. One possibility to reduce the required storage is to only keep track of a single conflict for each frozen component code and ignore other conflicts. This would also lead to a very simple implementation of the loops in Algorithm 3 (line 1) and Algorithm 4 (line 7). On the other hand, this may also lead to a small performance loss.

VII. Conclusion

We have shown that the performance of product codes can be improved by applying a novel iterative decoding algorithm. The proposed algorithm uses anchor codewords to reliably detect and prevent miscorrections. It was shown that the performance improvements (and therefore the practical usefulness) depend on the component code parameters and the BSC crossover probability. In general, anchor decoding can provide significant improvements in cases where miscorrections severely affect the performance. For example, for BCH component codes with $t = 2$, NCG improvements of up to $0.4\,\mathrm{dB}$ can be obtained. Moreover, the error floor can be reduced up to the point where the performance is close to that of a genie-aided decoder that avoids all miscorrections. On the other hand, in cases where miscorrections rarely occur (e.g., for doubly-extended BCH codes with $t \geq 4$), limited performance gains are observed and the small NCG improvements likely do not warrant any complexity increase compared to the conventional iterative decoding.

References

[1] P. Elias, "Error-free coding," *IRE Trans. Inf. Theory*, vol. 4, no. 4, pp. 29–37, Apr. 1954.

[2] A. J. Feltström, D. Truhachev, M. Lentmaier, and K. S. Zigangirov, "Braided block codes," *IEEE Trans. Inf. Theory*, vol. 55, no. 6, pp. 2640–2658, Jul. 2009.

[3] J. Justesen, K. J. Larsen, and L. A. Pedersen, "Error correcting coding for OTN," *IEEE Commun. Mag.*, vol. 59, no. 9, pp. 70–75, Sep. 2010.

[4] J. Justesen, "Performance of product codes and related structures with iterated decoding," *IEEE Trans. Commun.*, vol. 59, no. 2, pp. 407–415, Feb. 2011.

[5] M. Scholten, T. Coe, and J. Dillard, "Continuously-interleaved BCH (CI-BCH) FEC delivers best in class NECG for 40G and 100G metro applications," in *Proc. Optical Fiber Communication Conf. (OFC)*, San Diego, CA, 2010.

[6] H. D. Pfister, S. K. Emmadi, and K. Narayanan, "Symmetric product codes," in *Proc. Information Theory and Applications Workshop (ITA)*, San Diego, CA, 2015.

[7] B. P. Smith, A. Farhood, A. Hunt, F. R. Kschischang, and J. Lodge, "Staircase codes: FEC for 100 Gb/s OTN," *J. Lightw. Technol.*, vol. 30, no. 1, pp. 110–117, Jan. 2012.

[8] N. Abramson, "Cascade decoding of cyclic product codes," *IEEE Trans. Commun. Tech.*, vol. 16, no. 3, pp. 398–402, Jun. 1968.

[9] Y.-Y. Jian, H. D. Pfister, K. R. Narayanan, R. Rao, and R. Mazahreh, "Iterative hard-decision decoding of braided BCH codes for high-speed optical communication," in *Proc. IEEE Glob. Communication Conf. (GLOBECOM)*, Atlanta, GA, 2014.

[10] Y.-Y. Jian, H. D. Pfister, and K. R. Narayanan, "Approaching capacity at high-rates with iterative hard-decision decoding," *IEEE Trans. Inf. Theory*, vol. 63, no. 9, pp. 5752–5773, Sep. 2017.

[11] A. Farhoodfar, F. R. Kschischang, A. Hunt, B. P. Smith, and J. Lodge, "Staircase forward error correction coding," US Patent 8,751,910 B2, 2011.

[12] L. M. Zhang and F. R. Kschischang, "Staircase codes with 6% to 33% overhead," *J. Lightw. Technol.*, vol. 32, no. 10, pp. 1999–2002, May 2014.

[13] C. Häger, A. Graell i Amat, H. D. Pfister, A. Alvarado, F. Brännström, and E. Agrell, "On parameter optimization for staircase codes," in *Proc. Optical Fiber Communication Conf. (OFC)*, Los Angeles, CA, 2015.

[14] C. Häger, H. D. Pfister, A. Graell i Amat, and F. Brännström, "Density evolution for deterministic generalized product codes on the binary erasure channel at high rates," *IEEE Trans. Inf. Theory*, vol. 63, no. 7, pp. 4357–4378, Jul. 2017.

[15] ——, "Density evolution and error floor analysis of staircase and braided codes," in *Proc. Optical Fiber Communication Conf. (OFC)*, Anaheim, CA, 2016.

[16] C. Condo, P. Giard, F. Leduc-Primeau, G. Sarkis, and W. J. Gross, "A 9.96 dB NCG FEC scheme and 164 bits/cycle low-complexity product decoder architecture," *IEEE Trans. Circuits and Systems I: Fundamental Theory and Applications (accepted for publication)*, 2017. [Online]. Available: https://arxiv.org/pdf/1610.06050v2.pdf

[17] H. C. Chang, C. B. Shung, and C. Y. Lee, "A Reed–Solomon product-code (RS-PC) decoder chip for DVD applications," *IEEE J. Solid-State Circuits*, vol. 36, no. 2, pp. 229–238, Feb. 2001.

[18] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Proc. ACM/IEEE Int. Symp. Microarchitecture (MICRO)*, 2007.

[19] V. Tam Van and S. Mita, "A novel error correcting system based on product codes for future magnetic recording channels." *IEEE Trans. Magnetics*, vol. 56, no. 10, pp. 3320–3323, Oct. 2011.

[20] C. Yang, Y. Emre, and C. Chakrabarti, "Product code schemes for error correction in MLC NAND flash memories," *IEEE Trans. VLSI Systems*, vol. 20, no. 12, pp. 2302–2314, Dec. 2012.

[21] S. Emmadi, K. R. Narayanan, and H. D. Pfister, "Half-product codes for flash memory," in *Proc. Non-Volatile Memories Workshop*, San Diego, CA, 2015.

[22] R. M. Pyndiah, "Near-optimum decoding of product codes: block turbo codes," *IEEE Trans. Commun.*, vol. 46, no. 8, pp. 1003–1010, Aug. 1998.

[23] R. Lucas, M. Bossert, and M. Breitbach, "On iterative soft-decision decoding of linear binary block codes and product codes," *IEEE J. Sel. Areas Commun.*, vol. 16, no. 2, pp. 276–296, Feb. 1998.

[24] M. Schwartz, P. Siegel, and A. Vardy, "On the asymptotic performance of iterative decoders for product codes," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, Adelaide, SA, 2005.

[25] J. Justesen and T. Høholdt, "Analysis of iterated hard decision decoding of product codes with Reed-Solomon component codes," in *Proc. IEEE Information Theory Workshop (ITW)*, Tahoe City, CA, 2007.

[26] L. M. Zhang, D. Truhachev, and F. R. Kschischang, "Spatially-coupled split-component codes with iterative algebraic decoding," *IEEE Trans. Inf. Theory*, vol. 64, no. 1, pp. 205–224, Jan. 2017.

[27] D. Truhachev, A. Karami, L. Zhang, and F. Kschischang, "Decoding analysis accounting for mis-corrections for spatially-coupled split-component codes," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, Barcelona, Spain, 2016.

[28] L. Holzbaur, H. Bartz, and A. Wachter-Zeh, "Improved decoding and error floor analysis of staircase codes," in *Proc. Int. Workshop on Coding and Cryptography (WCC)*, Saint Petersburg, Russia, 2017.

[29] R. J. McEliece and L. Swanson, "On the decoder error probability for Reed-Solomon codes," *IEEE Trans. Inf. Theory*, vol. 32, no. 5, pp. 701–703, Sep. 1986.

[30] S. Sridharan, M. Jarchi, and T. Coe, "Product code based forward error correction system," US Patent 6,810,499 B2, 2003.

[31] C. Condo, G. Sarkis, P. Giard, W. J. Gross, and S. Member, "Stall pattern avoidance in polynomial product codes," in *Proc. IEEE Global Conf. Signal and Information Processing (GlobalSIP)*, Washington, DC, 2016.

[32] T. Mittelholzer, T. Parnell, N. Papandreou, and H. Pozidis, "Improving the error-floor performance of binary half-product codes," in *Proc. Int. Symp. Information Theory and its Applications (ISITA)*, Montenery, CA, 2016.

[33] B. P. Smith, "Error-correcting codes for fibre-optic communication systems," Ph.D. dissertation, University of Toronto, 2011.

[34] C. Häger and H. D. Pfister, "Miscorrection-free decoding of staircase codes," in *Proc. European Conf. Optical Communication (ECOC)*, Gothenburg, Sweden, 2017.

[35] S. Hirasawa, M. Kasahara, Y. Sugiyama, and T. Namekawa, "Modified product codes," *IEEE Trans. Inf. Theory*, vol. 30, no. 2, pp. 299–306, Mar. 1984.

[36] M. Alipour, O. Etesami, G. Maatouk, and A. Shokrollahi, "Irregular product codes," in *Proc. IEEE Information Theory Workshop (ITW)*, Lausanne, Switzerland, 2012.

[37] D. Gorenstein, W. W. Peterson, and N. Zierler, "Two-error correcting Bose-Chaudhuri codes are quasi-perfect," *Inf. Control*, vol. 3, no. 3, pp. 291–294, 1960.

[38] B.-Y. Wang and F. Zhang, "On the precise number of (0,1)-matrices in U(R,S)," *Discrete Mathematics*, vol. 187, pp. 211–220, 1998.

Appendix A
Density Evolution for Product Codes

In this appendix, we briefly describe how to reproduce the DE results that are shown in Figs. 2, 4, and 5.

Consider a BCH component code with parameters (ν, t, e). The goal is to predict the waterfall BER of the PC $\mathcal{P}(\mathcal{C})$ under iterative BDD as a function of the BSC crossover probability p. Recall that the component code length is $n = 2^\nu - 1 + e$ and the number of iterations is ℓ, where each iteration consists of two half-iterations with row and column codes being decoded separately. We define $c = pn$, $L = 2$, and $\boldsymbol{\eta} = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. Further, let $\mathcal{A}^{(l)} = \{1\}$ for l odd and $\mathcal{A}^{(l)} = \{2\}$ for l even, where $l \in [2\ell]$. Finally, let $\Psi_{\geq t}(\lambda) = 1 - e^{-\lambda} \sum_{i=0}^{t-1} \frac{\lambda^i}{i!}$ be the tail probability of a Poisson random variable with mean λ. With these definition, we recursively compute

$$x_i^{(l)} = \begin{cases} \Psi_{\geq t}\left(\frac{c}{L} \sum_{j=1}^{L} \eta_{i,j} x_j^{(l-1)}\right) & \text{if } i \in \mathcal{A}^{(l)} \\ x_i^{(l-1)} & \text{otherwise} \end{cases} \tag{11}$$

for $i \in [L]$ and $l = 1, 2, \ldots, 2\ell$, using $x_i^{(0)} = 1$, $i \in [L]$, as initial values. Collecting all final values in a vector as $\boldsymbol{x} = (x_1^{(2\ell)}, \ldots, x_L^{(2\ell)})$, the BER is then approximated as

$$\mathrm{BER}(p) \approx p \frac{\boldsymbol{x}\boldsymbol{\eta}\boldsymbol{x}^\intercal}{\|\boldsymbol{\eta}\|_{\mathrm{F}}^2}, \tag{12}$$

where $\|\boldsymbol{\eta}\|_{\mathrm{F}}^2$ is the number of 1s in $\boldsymbol{\eta}$.

The above procedure can be applied to compute the asymptotic performance for a wide variety of GPCs by simply adjusting the code parameters L and $\boldsymbol{\eta}$, and the decoding schedule $\mathcal{A}^{(l)}$. For example, for staircase codes, the matrix $\boldsymbol{\eta}$ is an $L \times L$ matrix with entries $\eta_{i,i+1} = \eta_{i+1,i} = 1$ for $i \in [L-1]$ and zeros elsewhere. The procedure can also be generalized to handle a mixture of component codes with different error-correcting capabilities as used for example in Section IV-C (see Fig. 6). For more details on this topic, we refer the interested reader to [14].

Christian Häger (S'11–M'16) received the Dipl.-Ing. degree (M.Sc. equivalent) in electrical engineering from Ulm University, Ulm, Germany, in 2011 and his Ph.D. degree in communication theory from Chalmers University of Technology, Gothenburg, Sweden, in 2016. Since August 2016, he is a postdoctoral researcher at the Department of Electrical and Computer Engineering at Duke University, Durham, USA, and, since April 2017, also at the Department of Electrical Engineering at Chalmers University at Technology. His research interests include modern coding theory, fiber-optic communications, and machine learning. He received the Marie Skłodowska-Curie Global Fellowship from the European Commission in 2017.

Henry D. Pfister (S'99–M'03–SM'09) received his Ph.D. in electrical engineering in 2003 from the University of California, San Diego and is currently an associate professor in the Electrical and Computer Engineering Department of Duke University. Prior to that, he was a professor at Texas A&M University (2006-2014), a post-doctoral fellow at the École Polytechnique Fédérale de Lausanne (2005-2006), and a senior engineer at Qualcomm Corporate R&D in San Diego (2003-2004).

He received the NSF Career Award in 2008 and a Texas A&M ECE Department Outstanding Professor Award in 2010. He is a coauthor of the 2007 IEEE COMSOC best paper in Signal Processing and Coding for Data Storage and a coauthor of a 2016 Symposium on the Theory of Computing (STOC) best paper. He served as an Associate Editor for the IEEE Transactions On Information Theory (2013-2016) and a Distinguished Lecturer of the IEEE Information Theory Society (2015-2016).

His current research interests include information theory, communications, probabilistic graphical models, and machine learning.