Rate-Compatible Punctured Low-Density Parity-Check Codes With Short Block Lengths

Jeongseok Ha, Jaehong Kim, Demijan Klinc, and Steven W. McLaughlin, *Fellow, IEEE*

***Abstract*—We consider the problem of rate-compatible puncturing of low-density parity-check (LDPC) codes over additive white Gaussian noise (AWGN) channels. In previous work, it was shown that "good" puncturing distributions exist for LDPC codes but the code length was large. In this correspondence, we give a procedure for determining the puncturing distributions for LDPC codes with short block lengths (a few thousand bits) and show that careful puncturing can produce good performance. We compare the performance of the rate-compatible punctured LDPC codes with dedicated LDPC codes across a range of rates and see that the rate-compatible codes have a favorable complexity/performance tradeoff.**

***Index Terms*—Low-density parity-check (LDPC) codes, puncturing, rate-adaptive, short block lengths.**

I. Introduction

Over time-varying channels, error-correction codes are required to be flexible with respect to their code rates depending on the current channel state. Rate adaptability [1] can be realized with several pairs of encoders and decoders for the desired code rates, which is sometimes unacceptable due to complexity. It can also be realized by puncturing a low rate channel code (*mother code*), resulting in only one encoder and decoder for each mother code. It is assumed that the decoder knows the locations of punctured coded symbols. If the punctured parity bits in a lower rate code are also punctured in a higher rate code, the rate-adaptive channel code is called rate-compatible [1]–[3].

Rate-compatible punctured channel codes have another advantage with type-II hybrid automatic-repeat-request (ARQ) protocols. Here, a transmitter sends partial redundancies by puncturing a mother code based on the channel state. If the receiver fails to recover the message, the receiver progressively requests additional redundancies which were previously punctured and hence not transmitted. For a given code rate the coding gain of a punctured code is usually poorer than that of the corresponding dedicated code, i.e., optimized code for the rate. Thus, punctured bits must be chosen such that the performance gap between the dedicated and punctured codes is minimized. A selection of punctured symbols is called a *puncturing distribution* for a code rate in this paper.

Previous work [2], [4], [5] gives a puncturing method with irregular low-density parity-check (LDPC) codes, which is based on the degree distributions of irregular LDPC codes. It was shown that there exist good (capacity approaching) puncturing distributions over a broad range of code rates. Moreover, the designed puncturing distributions are rate-compatible. However, these results presume infinite block lengths. Here, we are interested in practical punctured LDPC codes that are either regular or irregular at small block lengths (a few thousand bits). Although there have been several studies [3], [6] about rate-compatible punctured LDPC codes at moderate block lengths, the locations of punctured symbols were arbitrarily chosen in these previous works. In such a way, randomly obtained puncturing distributions may have severe performance losses especially at high code rates where significant amounts of parity bits are punctured. It is also possible for some subsets of punctured symbols to be unrecoverable, *stopping sets* [7]. Recently, Tian *et al.* [8] proposed rate-compatible LDPC codes by puncturing lower-triangular parity-check matrices where the puncturing does not violate the degree distribution profiles of mother codes.

In this paper, we propose a systematic way to find the locations of punctured symbols in LDPC codes which is rate-compatible and minimizes the performance loss due to the puncturing. Our idea is based on a fact that a punctured node will be recovered with reliable messages when it has 1) more neighboring (check) nodes, and 2) each of the check nodes has more reliable neighbors (variable nodes) except for the punctured one. For example, a punctured variable node that has check nodes whose remaining neighboring variable nodes are unpunctured will have nonzero messages from the check nodes in the first iteration. We call the process for a punctured node to have messages from check nodes as *recovery* by analogy with the one over binary-erasure channels. The punctured node in the preceding example will be called a one-step-recoverable (1-SR) since the node is recovered in the first iteration. The 1-SR nodes and unpunctured nodes will help recover some of the remaining punctured nodes in the second iteration, and so on. In general, the punctured nodes recovered in the kth iteration are called k-SR nodes. It is assumed that the more iterations a punctured node needs for its recovery, the less statistically reliable the recovery message is. Thus, it is better to puncture nodes that require a smaller number of iterations, which results not only in less iterations to decode codewords but also in better performance at a given code rate.

Based on this, our idea focuses on how to find nodes to be recovered in a certain number of iterations. First, we maximize the size of a group containing 1-SR nodes $\boldsymbol{G}_1$. After that, we maximize the size of a group with 2-SR nodes $\boldsymbol{G}_2$, and so on. The groups $\boldsymbol{G}_1, \boldsymbol{G}_2, \ldots, \boldsymbol{G}_K$ will be used to determine the puncturing patterns for increasing the rates in a rate-compatible fashion. Namely, we will first puncture variable nodes from $\boldsymbol{G}_1$ then nodes from $\boldsymbol{G}_2$, and so on to achieve higher and higher rates.

The correspondence is organized as follows. In Section II we introduce notations and some definitions which are used in the other sections. In Section III, we describe the details of the proposed idea. In Section IV, we compare the performances of the proposed rate-compatible punctured LDPC codes with conventional (randomly) punctured LDPC codes and dedicated LDPC codes at various block lengths. Finally, in Section V, we summarize our results and discuss future work.

II. Basic Notations

A low-density parity-check (LDPC) code is a linear block code defined by a sparse parity-check matrix $\boldsymbol{H}$ which has m rows and n columns. In this paper it is assumed that parity-check matrices are defined over $\mathrm{GF}(2)$. That is, the element $H_{j,k}$ at the intersection of row j and column k in $\boldsymbol{H}$ is either 1 or 0. Each row represents a parity-check equation, and the columns with '1' in the row tells the indices of bits in a codeword which are involved in the parity-equation. That is, a codeword satisfies the parity-check equation defined by row j as

$$\sum_{k=1}^{n} H_{j,k} \cdot c_k = 0$$

Manuscript received February 22, 2005; revised October 8, 2005. The material in this correspondence was presented in part at the IEEE International Symposium on Information Theory, Chicago, IL, June/July 2004.

J. Ha is with the School of Engineering, Information and Communications University (ICU), 305-714, Korea (e-mail: jsha@icu.ac.kr).

J. Kim and S. W. McLaughlin are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: onil@ece.gatech.edu; swm@ece.gatech.edu).

D. Klinc was with Lehrstuhl für Nachrichtentechnik (LNT), Technical University Munich, Germany. He is now with the Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: demi@gimb.org).

Communicated by M. P. Fossorier, Associate Editor for Coding Techniques.
Digital Object Identifier 10.1109/TIT.2005.862118

0018-9448/$20.00 © 2006 IEEE

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

where $\boldsymbol{c} = \{c_1, c_2, \ldots, c_n\}$ is a codeword, c_k is in GF(2), and $1 \leq j \leq m$. A parity-check matrix can be equivalently expressed with a bipartite graph in which there are two types of nodes, *check* nodes (a set of all check nodes is denoted as $P = \{p_1, p_2, \ldots, p_m\}$) and *variable* nodes (a set of all variable nodes is denoted as $V = \{v_1, v_2, \ldots, v_n\}$). Check node p_j and variable node v_k correspond to row j and column k in a parity-check matrix, respectively. The value '1' of $H_{j,k}$ is represented with a connection (*edge*) between check node j and variable node k in the corresponding bipartite graph where no edge connects two nodes of the same type. Such a graph is often called a Tanner graph [9]. Both a Tanner graph and its corresponding parity-check matrix uniquely define an LDPC code. Thus, Tanner graphs and parity-check matrices are used interchangeably here.

For an arbitrary node v we define $\mathcal{N}_v$ as a set of all nodes that can be reached from v by traversing only one edge. The nodes in $\mathcal{N}_v$ and the size of $\mathcal{N}_v$ are called the neighbors of v and the degree of v, respectively. Since one type of nodes have the other type of nodes as their neighbors in Tanner graphs, a check node has variable nodes as its neighbors and *vice versa*.

Puncturing of the mother code can be performed either randomly or according to carefully chosen locations of punctured bits optimized for a given parity-check matrix. We will refer to the first as *random puncturing* and to the latter as *intentional puncturing* in the subsequent sections. We often use Tanner graphs to explain our ideas, where we refer to bits as variable nodes.

Suppose that some variable nodes are punctured. The decoder will set their log-likelihood ratios (LLRs) to 0 at the initialization. If a punctured variable node receives a nonzero LLR message from one of its neighboring check nodes, it is said to be recovered, regardless of whether the nonzero LLR value implies the correct bit value or not. Through iterations, all punctured variable nodes have to be recovered with the information from the unpunctured nodes.

The updating rule for variable nodes suggests that a punctured variable node v will be recovered when at least one of the incoming check node messages is nonzero. On the other hand, a check node p cannot decide its message to the punctured node v if there is at least one punctured node among $\mathcal{N}_p \setminus \{v\}$. Accordingly, we refer to a punctured variable node v as *one-step-recoverable* (1-SR) if it has at least one neighbor p, called the survived check node, such that all variable nodes from the set $\mathcal{N}_p \setminus \{v\}$ are unpunctured. The check nodes from $\mathcal{N}_v$ that do not satisfy this condition are referred to as dead check nodes. We illustrate an example of a 1-SR node in Fig. 1 where S_1 and S_2 are the survived check nodes, and D is the dead check node.

In general, a punctured variable node v is called *k-step recoverable* (k-SR) if v has at least one neighbor p, called the survived check node, such that the set $\mathcal{N}_p \setminus \{v\}$ contains at least one $(k-1)$-SR node while the others are m-SR, where $0 \leq m \leq k-1$.[1] A k-SR node, $k \geq 1$, can have multiple survived check nodes but the puncturing algorithm that we propose in this paper guarantees an existence of at least one survived check node for each k-SR node. We refer to it as the guaranteed survived check node. Notice that a k-SR node will be recovered after exactly k iterations.

We seek an automated procedure for determining exactly which variable nodes should be punctured and in which order they should be punctured. What follows is a systematic search algorithm for determining the order of nodes to be punctured. The algorithm first attempts to maximize the number of punctured nodes that are 1-SR. When no more 1-SR nodes can be found, it proceeds with 2-SR nodes, etc., until every node in V has been chosen to be either unpunctured or punctured. We say the search algorithm assigns the variable nodes to groups $\boldsymbol{G}_0, \boldsymbol{G}_1, \ldots \boldsymbol{G}_k$, where k indicates the level of recoverability of variable nodes in that group. For instance, if a variable node is chosen to be unpunctured, it is assigned to $\boldsymbol{G}_0$, whereas if it is chosen to be punctured such that it will be recovered in the kth iteration, it is assigned to $\boldsymbol{G}_k$. At any particular time during the search algorithm, all variable nodes whose status (punctured or unpunctured) has not been determined yet are in $\boldsymbol{G}_\infty$. Hence, when the algorithm starts $\boldsymbol{G}_\infty = V$, and when it stops $\boldsymbol{G}_\infty = \emptyset$.

[1]A 0-SR node denotes an unpunctured node.

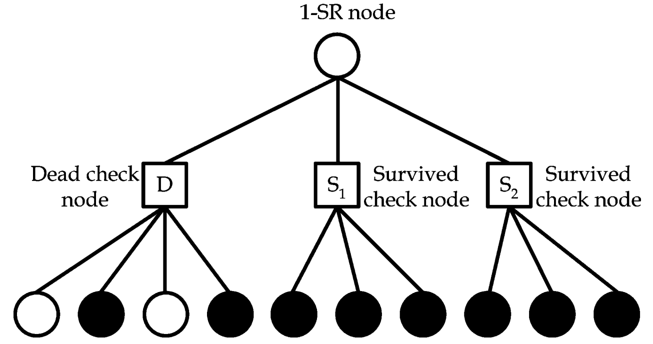

Fig. 1. One-step recoverable node; the filled and unfilled circles are unpunctured and punctured variable nodes, respectively. The squares S_1 and S_2 are the survived check nodes, and the square D is the dead check node.

To get a better understanding of the search algorithm we need a clear picture of how a node is to be recovered. Consider now a k-SR node v. We build a tree originating from v in the following way. First, we link v with its guaranteed survived check node p and subsequently link p with all variable nodes from the set $\mathcal{N}_p \setminus \{v\}$. In the next step, this process is repeated on every new punctured variable node in the tree until every branch terminates with an unpunctured variable node. The resulting tree is called the *recovery tree* of v. We show an example of a recovery tree in Fig. 2. Later, the number of unpunctured nodes in the recovery tree of v will be of big importance, so we designate it as $\mathcal{S}(v)$. In Fig. 2 $\mathcal{S}(v)$ is equal to 14. Assuming that v is recovered with the message-passing decoding algorithm on the recovery tree of v, we define the recovery-error probability of v, $P_e(v)$ as follows.

Definition 1: (Recovery error probability $P_e(v)$ of a k-SR node v) For $v \in G_k$ and $k \geq 1$, $P_e(v)$ is the probability that v is recovered with a wrong message in the k-th iteration, where the message is based only on the information from the unpunctured nodes in the recovery tree of v.

When we transmit a punctured LDPC code over a BEC with an erasure probability of ζ, the probability that a variable node v in $\boldsymbol{G}_k$ is recovered in its recovery tree is expressed in a recursive form as shown in Definition 2. Thus, the recovery-error probability of v is simply $(1-\Psi(v,\zeta))/2$. The term $\Psi(v,\zeta)$ is also useful to compute a recovery-error probability of a punctured variable node over AWGN channels. We will show the details of recovery-error probabilities in Section III.

Definition 2:

$$\Psi(v,\zeta) = \begin{cases} (1-\zeta), & \text{for } v \in \boldsymbol{G}_0 \\ \prod_{j=1}^{d_c - 1} \Psi(\gamma_j, \zeta), & \text{for } v \in \boldsymbol{G}_k \text{ and } k > 0 \end{cases}$$

where $\boldsymbol{G}_0$ and $\boldsymbol{G}_k$ are sets of unpunctured nodes and k-SR nodes, respectively, d_c is a degree of the survived check node of v, γ_j's are the neighbors of the survived check node except for v, and $\gamma_j \in \boldsymbol{G}_h$ for $0 \leq h \leq k-1$.

Definition 3: (Effective column weight of a column γ with respect to a subset of rows $\boldsymbol{R}$, $\mathrm{cw}_{\mathrm{eff}}(\gamma, \boldsymbol{R})$)

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

Fig. 2. Recovery tree of a 3-SR node; the squares are survived check nodes, the filled circles are unpunctured variable nodes, and the unfilled circles are punctured variable nodes.

For a parity-check matrix $\boldsymbol{H} = [H_{j,k} \in \{0,1\}]$ where $1 \leq j \leq m$ and $1 \leq k \leq n$, let $\boldsymbol{\Lambda}^{\gamma} = \{\rho : H_{\rho,\gamma} = 1 \text{ and } 1 \leq \rho \leq m\}$ and $\boldsymbol{R}$ be a subset of row indices, respectively. $\mathrm{cw}_{\mathrm{eff}}(c, \boldsymbol{R})$ is defined as $|\boldsymbol{\Lambda}^{c} \cap \boldsymbol{R}|$, where $|\cdot|$ is a cardinality of a set [10].

Definition 4: (Effective row weight of a row ρ with respect to a subset of columns $\boldsymbol{C}$, $\mathrm{rw}_{\mathrm{eff}}(\rho, \boldsymbol{C})$)

For a parity-check matrix $\boldsymbol{H} = [H_{j,k} \in \{0,1\}]$ where $1 \leq j \leq m$ and $1 \leq k \leq n$, let $\boldsymbol{\Gamma}^{\rho} = \{\gamma : H_{\rho,\gamma} = 1 \text{ and } 1 \leq \gamma \leq n\}$ and $\boldsymbol{C}$ be a subset of column indices, respectively. $\mathrm{rw}_{\mathrm{eff}}(r, \boldsymbol{C})$ is defined as $|\boldsymbol{\Gamma}^{r} \cap \boldsymbol{C}|$, where $|\cdot|$ is a cardinality of a set.

III. Algorithm for Determining Puncturing Patterns

In what follows we present a search algorithm consisting of two steps: Grouping and Sorting. In the Grouping step we distribute all variable nodes into groups $\boldsymbol{G}_0, \boldsymbol{G}_1, \ldots, \boldsymbol{G}_K$, while in the subsequent Sorting step we determine the order of puncturing of nodes within each group. We start by deriving some results on the recovery error probability, which is computed based on the information from the nodes in the recovery tree – the exact error probability of the recovery message seems to be too complicated. When we have to choose a new node out of several candidates for the grouping and sorting, we will choose the one with the smallest recovery-error probability.

Fact 1: The recovery error probability $P_e(v)$ of a variable node $v \in \boldsymbol{G}_{k>0}$ over a BEC with an erasure probability of ζ is $\frac{1}{2}(1-\Psi(v,\zeta))$, where $\Psi(v,\zeta) = (1-\zeta)^{\mathcal{S}(v)}$ for $v \in \boldsymbol{G}_k$ and $k > 0$ and $\Psi(v,\zeta)$ is the same as the probability that the variable node v is recovered.

Proof: We will prove this fact by induction. For $k = 1$, all variable nodes in the recovery tree of v are in $\boldsymbol{G}_0$. Thus,

$$P_e(v) = \frac{1}{2}(1-(1-\zeta)^{d_c - 1}) = \frac{1}{2}(1-(1-\zeta)^{\mathcal{S}(v)}) = \frac{1}{2}(1-\Psi(v,\zeta))$$

where d_c is a degree of a survived check node of v. For $\gamma \in \boldsymbol{G}_k$, assume that

$$P_e = \frac{1}{2}(1-\Psi(\gamma,\zeta)) = \frac{1}{2}(1-(1-\zeta)^{\mathcal{S}(\gamma)})$$

then for $v \in \boldsymbol{G}_{k+1}$,

$$\begin{aligned} P_e &= \frac{1}{2}(1 - \prod_{j=1}^{d_c-1} \Psi(\gamma_j,\zeta)) \\ &= \frac{1}{2}(1-(1-\zeta)^{\sum_{j=1}^{d_c-1} \mathcal{S}(\gamma_j)}) \\ &= \frac{1}{2}(1-(1-\zeta)^{\mathcal{S}(v)}) \\ &= \frac{1}{2}(1-\Psi(v,\zeta)) \end{aligned}$$

where d_c is a degree of the survived check node of v, γ_j's are the neighbors of the survived check node except for v, $\sum_{j=1}^{d_c-1} \mathcal{S}(\gamma_j) = \mathcal{S}(v)$, and $\gamma_j \in \boldsymbol{G}_h$ for $1 \leq h \leq k$. □

Fact 2: The recovery error probability $P_e(v)$ of a variable node $v \in \boldsymbol{G}_k$ over a Binary Symmetric Channel with an erasure probability of ζ has a relation with $\Psi(v,\zeta)$, $P_e(v) = \frac{1-\Psi(v,2P_e)}{2}$.

Proof: A recovery error probability of a variable node $v \in \boldsymbol{G}_1$ will be $\frac{1-(1-2\zeta)^{d_c-1}}{2} = \frac{1-\Psi(v,2\zeta)}{2}$. Assume that for a variable node $\gamma \in \boldsymbol{G}_k$

$$P_e(\gamma) = \frac{1-(1-2\zeta)^{\mathcal{S}(\gamma)}}{2} = \frac{1-\Psi(\gamma,2\zeta)}{2}.$$

Thus, for a variable node $v \in \boldsymbol{G}_{k+1}$

$$\begin{aligned} P_e(v) &= \frac{1 - \prod_{j=1}^{d_c-1}(1-2\zeta)^{\mathcal{S}(\gamma_j)}}{2} \\ &= \frac{1-(1-2\zeta)^{\sum_{j=1}^{d_c-1} \mathcal{S}(\gamma_j)}}{2} \\ &= \frac{1-\Psi(v,2\zeta)}{2} \end{aligned}$$

where d_c is the degree of a survived check node of v, γ_j's are the neighbors of the survived check node except for v, $\sum_{j=1}^{d_c-1} \mathcal{S}(\gamma_j) = \mathcal{S}(v)$, and $\gamma_j \in \boldsymbol{G}_h$ for $1 \leq h \leq k$. □

Fact 3: The recovery error probability of a variable node $v \in \boldsymbol{G}_k$ over an Additive White Gaussian Channel with Gaussian Approximation [11] is $P_e(v) = Q(\sqrt{m_u(v)/2})$, where $Q(\cdot)$ is the Q-function, $m_u(v) = \phi^{-1}(1-\phi(1-\Psi(v,\phi(m_{u_0}))))$ for $m_{u_0} = 2/\sigma^2$ and noise variance σ^2, and $\phi(x) \triangleq 1 - 1/\sqrt{4\pi x}\int_{\mathbb{R}} \tanh\frac{u}{2} e^{-(u-x)^2/4x} du$ for $x > 0$ and $\phi(x) \triangleq 1$ for $x = 0$ (see the details of $\phi(x)$ in [11]).

Proof: An updated mean of a variable node $v \in \boldsymbol{G}_1$ will be

$$\begin{aligned} \phi^{-1}(1-[1-\phi(m_{u_0})]^{d_c-1})) &= \phi^{-1}(1-[1-\phi(m_{u_0})]^{\mathcal{S}(v)})) \\ &= \phi^{-1}(1-\Psi(v,\phi(m_{u_0}))). \end{aligned}$$

Assume that for a variable node $\gamma \in \boldsymbol{G}_k$,

$$m_u(\gamma) = \phi^{-1}(1-\Psi(\gamma,\phi(m_{u_0}))).$$

Then, for a variable node $v \in \boldsymbol{G}_{k+1}$

$$\begin{aligned} m_u(v) &= \phi^{-1}\left(1 - \prod_{j=1}^{d_c-1}[1-\phi(m_{u_0})]^{\mathcal{S}(\gamma_j)}\right) \\ &= \phi^{-1}\left(1-[1-\phi(m_{u_0})]^{\sum_{j=1}^{d_c-1} \mathcal{S}(\gamma_j)}\right) \\ &= \phi^{-1}\left(1-[1-\phi(m_{u_0})]^{\mathcal{S}(v)}\right) \\ &= \phi^{-1}(1-\Psi(v,\phi(m_{u_0}))) \end{aligned}$$

where m_{u_0} is the mean of a log likelihood ratio message from the channel to unpunctured variable nodes, d_c is a degree of a survived

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

check node of v, γ_j's are the neighbors of the survived check node except for v, $\sum_{j=1}^{d_c-1} \mathcal{S}(\gamma_j) = \mathcal{S}(v)$, and $\gamma_j \in \boldsymbol{G}_h$ for $1 \le h \le k$. □

Facts 2 and 3 tell that a k-SR node v with larger $\mathcal{S}(v)$ has a larger recovery error probability. Later, in the algorithm, we will look for a variable node with a smaller $\mathcal{S}(v)$ as a new k-SR node.

Proposed Algorithm: Grouping

Step 0.0 **[Initialization]** For a given $m \times n$ parity-check matrix $\boldsymbol{H}$, $k = 1$, $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ are empty sets, $\boldsymbol{R}_\infty = \{1, 2, \ldots, m\}$, $\boldsymbol{\Gamma}^\rho = \{\gamma : H_{\rho,\gamma} = 1, \text{ and } 1 \le \gamma \le n\}$, $\boldsymbol{\Lambda}^\gamma = \{\rho : H_{\rho,\gamma} = 1 \text{ and } 1 \le \rho \le m\}$, $\boldsymbol{G}_0$ and $\boldsymbol{G}_1$ are empty sets, $\boldsymbol{G}_\infty = \{1, 2, \ldots, n\}$, $\mathcal{S}(j) = 0$ for all $1 \le j \le n$.

Step 1.0 **[Group Column Indices]** Form a set $\mathcal{G}_\infty^\rho$ such that $\mathcal{G}_\infty^\rho = \boldsymbol{\Gamma}^\rho \cap \boldsymbol{G}_\infty$ for each $\rho \in \boldsymbol{R}_\infty$.

Step 2.0 **[Find Candidate Rows]** Make a subset of $\boldsymbol{R}_\infty$ (call it $\boldsymbol{\Omega}$) such that $\forall\ \omega \in \boldsymbol{\Omega}$, $|\mathcal{G}_\infty^\omega| = \mathrm{rw}_{\mathrm{eff}}^{\min} \le |\mathcal{G}_\infty^\rho| = \mathrm{rw}_{\mathrm{eff}}(\rho, \boldsymbol{G}_\infty)$ for any $\rho \in \boldsymbol{R}_\infty$.

Step 3.0 **[Group Row Indices]** Make a set $\mathcal{C}_\infty^\gamma$ such that $\mathcal{C}_\infty^\gamma = \boldsymbol{\Lambda}^\gamma \cap \boldsymbol{R}_\infty$, for all $\gamma \in \mathcal{G}_\infty^\omega$, and $\omega \in \boldsymbol{\Omega}$.

Step 3.1 **[Find Best Rows]** Find a subset of $\boldsymbol{\Omega}$ (call it $\boldsymbol{\Omega}^*$) such that $\forall\ \omega^* \in \boldsymbol{\Omega}^*$, $\exists\ c \in \mathcal{G}_\infty^{\omega^*}$ such that $|\mathcal{C}_\infty^c| = \mathrm{cw}_{\mathrm{eff}}^{\min} \le |\mathcal{C}_\infty^\gamma| = \mathrm{cw}_{\mathrm{eff}}(\gamma, \boldsymbol{R}_\infty)$ for any $\omega \in \boldsymbol{\Omega}$ and $\gamma \in \mathcal{G}_\infty^\omega$.

Step 3.2 **[Make a Set of Ordered Pairs]** Pick a column index $c^* \in \mathcal{G}_\infty^{\omega^*}$ with $\mathrm{cw}_{\mathrm{eff}}^{\min}$ randomly, when there are more than one column index with $\mathrm{cw}_{\mathrm{eff}}^{\min}$. Then, we will have an ordered pair $\mathcal{O} = \{(\omega_1^*, c_1^*), (\omega_2^*, c_2^*), \ldots, (\omega_p^*, c_p^*)\}$, where ω_j^*'s and c_j^*'s are row and column indices with $\mathrm{rw}_{\mathrm{eff}}^{\min}$ and $\mathrm{cw}_{\mathrm{eff}}^{\min}$, respectively, and $1 \le j \le |\mathcal{O}| = p$.

Step 3.3 **[Find The Best Pair]** Pick a pair (ω^*, c^*) from $\mathcal{O}$ such that $\mathcal{W}(\omega^*) \le \mathcal{W}(\omega_j^*)$, where $1 \le j \le p$, $\mathcal{W}(\omega_j^*) = \sum_{\gamma \in \boldsymbol{\Gamma}^{\omega_j^*}} \mathcal{S}(\gamma)$. If there are more that one pair satisfying the inequality, pick one randomly.

Step 4.0 **[Update]** $\boldsymbol{G}_k = \boldsymbol{G}_k \cup \{c^*\}$, $\boldsymbol{G}_0 = \boldsymbol{G}_0 \cup \left(\mathcal{G}_\infty^{\omega^*} \setminus \{c^*\}\right)$, $\boldsymbol{G}_\infty = \boldsymbol{G}_\infty \setminus \mathcal{G}_\infty^{\omega^*}$, $\boldsymbol{R}_k = \boldsymbol{R}_k \cup \{\omega^*\}$, $\boldsymbol{R}_0 = \boldsymbol{R}_0 \cup \left(\mathcal{C}_\infty^{\omega^*} \setminus \{\omega^*\}\right)$, and $\boldsymbol{R}_\infty = \boldsymbol{R}_\infty \setminus \mathcal{C}_\infty^{c^*}$, $\mathcal{S}(\gamma) = 1$ for $\gamma \in \mathcal{G}_\infty^{\omega^*} \setminus c^*$, $\mathcal{S}(c^*) = \sum_{\gamma \in \{\boldsymbol{\Gamma}^{\omega^*} \setminus c^*\}} \mathcal{S}(\gamma)$.

Step 5.0 **[Check Stop Condition]** If $\boldsymbol{G}_\infty$ is an empty set, then **STOP**.

Step 5.1 **[Decision]** If $\boldsymbol{R}_\infty$ is not empty set, go to Step 1.0.

Step 5.2 **[No More Undetermined Rows]** $\boldsymbol{R}_\infty = \{\rho : \rho \in \boldsymbol{R}_0 \text{ and } \mathrm{rw}_{\mathrm{eff}}(\rho, \boldsymbol{G}_\infty) > 0\}$.

Step 6.0 $k = k + 1$, and go to Step 1.0.

Details of the Grouping Algorithm:

In Step 0.0, $\boldsymbol{\Gamma}^\rho$ ($\boldsymbol{\Lambda}^\gamma$) is a set of nonzero column (row) indices in the row ρ (column γ), and $\boldsymbol{G}_0$, $\boldsymbol{G}_1$, and $\boldsymbol{G}_\infty$ are sets of unpunctured, 1-SR and undetermined column indices, respectively. When there are no more undetermined columns ($\boldsymbol{G}_\infty$ is an empty set in Step 5.0), the algorithm will stop. $\boldsymbol{R}_k$ and $\boldsymbol{R}_\infty$ are sets of row indices which are (*guaranteed*) survived check nodes of k-SR nodes and undetermined check nodes, respectively. Suppose a row contains a k-SR node. That k-SR node is also in $d_v - 1$ other rows, where d_v is a degree of the k-SR node. The indices of the other $d_v - 1$ rows of all k-SR nodes will be assigned to $\boldsymbol{R}_0$, and the rows in $\boldsymbol{R}_0$ are excluded from the candidate rows for a new survived check node of a k-SR node. $\boldsymbol{\Gamma}^\rho$ ($\boldsymbol{\Lambda}^\gamma$) is a set containing the indices of all nonzero columns (rows) in row (column) ρ (γ).

In Step 1.0, we form a set $\mathcal{G}_\infty^\rho$ that contains all the column indices both in $\boldsymbol{\Gamma}^\rho$ and $\boldsymbol{G}_\infty$. Thus, the cardinality of $\mathcal{G}_\infty^\rho$ is $\mathrm{rw}_{\mathrm{eff}}(\rho, \boldsymbol{G}_\infty)$ by Definition 3. In Step 2.0, we look for rows in $\boldsymbol{R}_\infty$ with a minimum of $\mathrm{rw}_{\mathrm{eff}}(\rho, \boldsymbol{G}_\infty)$ which is simply denoted as $\mathrm{rw}_{\mathrm{eff}}^{\min}$. Since the size of $|\boldsymbol{G}_\infty|$ decreases by $|\mathcal{G}_\infty^\rho|$ in Step 4.0, choosing a row with $\mathrm{rw}_{\mathrm{eff}}^{\min}$ will leave us more candidate rows one of which can serve as a survived check node for the next k-SR node. In general, there may be more than one row with $\mathrm{rw}_{\mathrm{eff}}^{\min}$, and their indices are contained in the set $\boldsymbol{\Omega}$. In Step 3.0, we form a set $\mathcal{C}_\infty^\gamma$ with row indices that belong to both $\boldsymbol{\Lambda}^\gamma$ and $\boldsymbol{R}_\infty$. Similar to $\mathcal{G}_\infty^\omega$, the cardinality of $\mathcal{C}_\infty^\gamma$ is $\mathrm{cw}_{\mathrm{eff}}(\gamma, \boldsymbol{R}_\infty)$ by Definition 3. We look for rows in which there is at least one column with a minimum of $\mathrm{cw}_{\mathrm{eff}}(\gamma, \boldsymbol{R}_\infty)$ which is simply denoted as $\mathrm{cw}_{\mathrm{eff}}^{\min}$. Again, we will have more k-SR nodes with $\mathrm{cw}_{\mathrm{eff}}^{\min}$ since in Step 4.0, $|\boldsymbol{R}_\infty|$ decreases by $|\mathcal{C}_\infty^{\omega^*}|$. In Step 3.3, we make a set $\mathcal{O}$ of ordered pairs each of which has a row and a column of the row with $\mathrm{rw}_{\mathrm{eff}}^{\min}$ and $\mathrm{cw}_{\mathrm{eff}}^{\min}$, respectively. The set of ordered pairs may not be unique since a row may have several columns with $\mathrm{cw}_{\mathrm{eff}}^{\min}$. In this case, we randomly choose a column from them. In terms of maximizing $\boldsymbol{G}_k$, each ordered pair gives us statistically the same result. Among the ordered pairs, we will choose the one with the highest (smallest recovery error) probability to be recovered in the kth iteration. The facts 2 and 3 tell that the k-SR node with a smaller $\mathcal{S}(c)$ will have a smaller recovery error probability. Thus, in Step 3.3 we pick up a pair with smallest $\mathcal{S}(c)$, which is equivalently evaluated with a measure $\mathcal{W}$ for computational efficiency. In Step 4.0, we update the sets with the pair chosen in Step 3.3. In Step 5.1, the cardinality of $\boldsymbol{R}_\infty$ is checked. If it is not zero, Step 1.0 will be visited again. Otherwise, $\boldsymbol{R}_\infty$ is updated in Step 5.2 where $\boldsymbol{R}_\infty$ takes rows ρ's with nonzero $\mathrm{rw}_{\mathrm{eff}}(\rho, \boldsymbol{G}_\infty)$ in $\boldsymbol{R}_0$. In Step 6.0 we increase k by 1 and start looking for $(k+1)$-SR symbols.

The rows in $\boldsymbol{R}_k$ are the survived check nodes of the corresponding k-SR nodes. Thus, the algorithm guarantees at least one survived check node for each k-SR node. We define a set of $\boldsymbol{G}_k$'s from the algorithm as

$$\boldsymbol{\Theta} = \{\boldsymbol{G}_0, \boldsymbol{G}_1, \ldots, \boldsymbol{G}_K\}$$

where it is assumed that the algorithm runs until $\boldsymbol{G}_\infty = \emptyset$, and K designates the highest number of iterations that will be required to recover all punctured nodes.

By puncturing symbols in $\boldsymbol{G}_k$, the maximum achievable code rate $r_{\max}$ can be expressed as

$$r_{\max} = \frac{r_0}{1 - \sum_{j=1}^{K} |\boldsymbol{G}_j|/n}$$

where r_0 is the mother code rate, and n is the block length of the mother code. For a sequence of designed rates, $r_0 \le r_1 \le \cdots \le r_M \le r_{\max}$, we can compute the required number of punctured nodes np_j as

$$np_j = \left\lfloor \frac{n(r_j - r_0)}{r_j} \right\rfloor$$

for $0 \le j \le M$. We will choose np_j nodes from $\boldsymbol{G}_k$'s such that the performance loss due to the puncturing is minimized. To achieve this we choose the lower indexed group first. Next, we will take a node with a lower column index first until we have np_j nodes. That is, in the process of choosing np_1 nodes, first we choose $\boldsymbol{G}_1$ and take a node with the smallest column index from $\boldsymbol{G}_1$, in the next choose a node with the next smallest column index and so on until we either take all the nodes in $\boldsymbol{G}_1$ or have np_1 nodes from $\boldsymbol{G}_1$. If $|\boldsymbol{G}_1|$ is less than np_1, we will do the same process with $\boldsymbol{G}_2$ until we have np_1 nodes. For np_2, we will do the same process and the nodes for np_2 automatically include the ones for np_1. This idea gives us a set of rate-compatible punctured LDPC codes in which the punctured nodes for a lower rate are included in the ones for all the higher rates. Although this idea gives us reasonable performance, we elaborate more on the way how we choose punctured nodes from $\boldsymbol{G}_k$'s, which must be rate-compatible and minimize performance loss at each rate.

Now, we discuss the Sorting step where we determine the order of puncturing within each group $\boldsymbol{G}_1, \boldsymbol{G}_2, \ldots, \boldsymbol{G}_K$.

Proposed Algorithm: Sorting

[Step 0.0] **[Initialization]** For a given $m \times n$ parity-check matrix, $j = 1$, $k = 1$, $\boldsymbol{\Lambda}^c$ is a set of nonzero row indices in the column c, $\boldsymbol{P}_0$ is an empty sets, and $\boldsymbol{R} = \{1, 2, \ldots, m\}$.

[Step 0.1] If $j > M$, **STOP**.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

[Step 0.2] $\boldsymbol{P}_j = \boldsymbol{P}_{j-1}$.
[Step 0.3] $\delta np_j = np_j - |\boldsymbol{P}_j|$.
[Step 0.4] If δnp_j is zero, $j = j + 1$ and go to 0.1.
[Step 1.0] Make a set of column indices $\{c_1, c_2, \ldots, c_p\}$, for $1 \leq p \leq |\boldsymbol{G}_k|$ from $\boldsymbol{G}_k$ such that $\forall\, c_j \in \{c_1, c_2, \ldots, c_p\}$, $\mathrm{cw}_{\mathrm{eff}}(c_j, \boldsymbol{R}) = \mathrm{cw}_{\mathrm{eff}}^{\max} \geq \mathrm{cw}_{\mathrm{eff}}(c, \boldsymbol{R})$, for any $c \in \boldsymbol{G}_k$.
[Step 1.1] If $p > 1$, we take nodes c_j^* such that $\forall c \in \{c_1, c_2, \ldots, c_p\}$, $\deg(c_j^*) \leq \deg(c)$. If there are multiple such nodes, we pick one from them arbitrary and call it c^*.
[Step 2.0] $\boldsymbol{P}_j = \boldsymbol{P}_j \cup \{c^*\}, \boldsymbol{G}_k = \boldsymbol{G}_k \setminus \{c^*\}$, and $\boldsymbol{R} = \boldsymbol{R} \setminus \boldsymbol{\Lambda}^{c^*}$, $\delta np_j = \delta np_j - 1$.
[Step 3.0] If $\boldsymbol{G}_k$ is an empty set, $k = k + 1$, and $\boldsymbol{R} = \{1, 2, \ldots, m\}$.
[Step 4.0] Go to Step 0.4.

In the proposed sorting algorithm, $\boldsymbol{P}_j$ will contain column indices of the variable nodes which are punctured to achieve rate r_j. Obviously, for r_0, $\boldsymbol{P}_0$ is an empty set at the initialization. It is assumed that we will design rates from r_0 to r_M which is less than or equal to $r_{\max}$. Thus, in Step 0.1, if j is larger than M, the algorithm will stop. Subsequently, in Step 0.2, $\boldsymbol{P}_j$ inherits the column indices from $\boldsymbol{P}_{j-1}$, which makes the punctured LDPC codes rate compatible since all the punctured nodes for r_{j-1} will be punctured again for r_j. In Step 0.3, δnp_j accounts for how many additional nodes are needed to make $\boldsymbol{P}_j$ besides the ones in $\boldsymbol{P}_{j-1}$. The loop between Step 0.4 and Step 4.0 continues until δnp_j becomes zero. In Step 1.0 we look for nodes with the largest number of survived check nodes which is equivalent to nodes with $\mathrm{cw}_{\mathrm{eff}}^{\max}$.

We compute the additional number of punctured nodes to the ones in the previous rate r_{j-1}. If we need additional nodes, the algorithm looks for nodes with a maximum effective column weight $\mathrm{cw}_{\mathrm{eff}}^{\max}$, which means the node with the maximum number of survived check nodes. We exclude rows with k-SR nodes from $\boldsymbol{R}$ in Step 2.0. Thus, $\mathrm{cw}_{\mathrm{eff}}^{\max}$ counts only survived check nodes of a variable node.

Each variable node in $\{c_1, c_2, \ldots, c_p\}$ has $\mathrm{cw}_{\mathrm{eff}}^{\max}$ survived check nodes and $\deg(c_j) - \mathrm{cw}_{\mathrm{eff}}^{\max}$ dead check nodes since check nodes of a variable node are either survived or dead. In Step 1.1, we choose nodes with the smallest column degree, $\deg(c^*)$, which is equivalent to choosing variable nodes with the smallest number of dead check nodes. Some of the dead nodes are survived check nodes of already punctured nodes and become dead due to puncturing the new k-SR node c^*. If we choose a node with more dead check nodes in Step 1.1, it will make more survived check nodes of previously punctured nodes dead. Thus, we look for nodes with the largest number of survived check nodes to maximize the number and reliability of incoming messages and the smallest number of dead check nodes to minimize the disturbance caused to the other k-SR nodes. It looks obvious that as we choose nodes from $\boldsymbol{G}_k$, the node chosen later may have smaller $\mathrm{cw}_{\mathrm{eff}}^{\max}$, which is intended to minimize performance loss at lower rates.

IV. Simulations

A. Simulations With Regular LDPC Codes

The proposed algorithms in Section III are based on the claim that a puncturing distribution that will be recovered in the smallest number of iterations guarantees better performance. In this section, we verify the claim with computer simulations, where we show that puncturing according to the proposed algorithms yields better performance than conventional random puncturing in terms of bit-error rate (BER)[2] and word-error rate (WER). The BER and WER performances are measured after observing at least 50 erroneous code words at each E_b/N_0 value to guarantee statistical confidence. The punctured LDPC codes, in this case regular, are also compared with dedicated LDPC codes which are designed at the rates of the punctured LDPC codes.

[2]To evaluate BER, we observe only message bits, which are all unpunctured.

TABLE I
Block Lengths of Punctured LDPC Codes; The Lengths in Parentheses Are the Numbers of Punctured Symbols at the Rates

block lengths	code rates			
	0.5	0.6	0.7	0.8
1024	1024 (0)	853 (171)	731 (293)	640 (384)
4096	4096 (0)	3413 (683)	2926 (1170)	2560 (1536)

First, we implement half rate mother LDPC codes with a regular structure ($\lambda(x) = x^2$ and $\rho(x) = x^5$) at block lengths of 1024 and 4096. We deliberately avoid 4-cycle loops in parity-check matrices of the mother codes to achieve better minimum distance property [12]. Subsequently, we puncture the mother codes to obtain punctured LDPC codes at rates $0.5, 0.6, 0.7,$ and 0.8. The block lengths of the punctured LDPC codes and the numbers of punctured parity bits for the rates are listed in Table I. It should be noticed that the punctured LDPC codes have shorter block lengths at higher rates due to a higher number of punctured parity bits. Although punctured LDPC codes have shorter block lengths, a received LDPC code is decoded on the Tanner graph of its mother code (1024 or 4096 in our simulations). For fair comparisons, dedicated LDPC codes are designed at the block lengths of the corresponding punctured LDPC codes for the given rates. Thus, the dedicated codes are decoded on shorter Tanner graphs as compared to the punctured LDPC codes.

In the case of block length 1024, puncturing distributions are designed with the intentional and random puncturing for the code rates in Table I. The distributions with the random puncturing are implemented three times with different random seeds to see the performance variations due to the different maximum levels of recoverability.

We analyze the group distributions of the selections of punctured parity bits obtained by the proposed grouping algorithm (denoted as *Intentional* in Table II) and the three different random selections of punctured bits (denoted as Random 1, 2, and 3 in Table II). Since the punctured parity bits with the grouping algorithm are in $\boldsymbol{G}_0 \sim \boldsymbol{G}_3$, the distribution with the intentional puncturing requires three iterations to recover all the punctured parity bits. That is, the maximum level of recoverability is 3. However, the distributions obtained by the random puncturing require at least 9 iterations, which results in higher recovery-error probabilities of the symbols in the groups with higher indices.

In Fig. 3, we compare BER performances of the randomly punctured LDPC codes with the three different random seeds at block length 1024, where the ones with the smallest and largest maximum levels of recoverability (denoted as Random 1 and 3 in Table II, respectively) show the best and the worst BER performances at rate 0.8, respectively. Thus, the simulation results justify our design rule which looks for selections of punctured parity bits with a smaller level of recoverability for the highest rate, 0.8. However the performances at the intermediate rates (0.6 and 0.7 in our simulations) in Fig. 3 do not seem to depend on the levels of recoverability since the punctured LDPC code with the puncturing distribution Random 3 has better performance than the one with Random 2. The distributions in Table II describe levels of recoverability of the punctured LDPC codes at the highest rate. Thus, better group distribution of a punctured LDPC code does not guarantee better performance at intermediate rates if we do not carefully choose the order of puncturing. For the intentional puncturing, we apply the sorting algorithm to determine the order of puncturing but in the case of random puncturing, we puncture parities with smaller node indices first. The performances of the randomly punctured LDPC codes at the

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

TABLE II
GROUP DISTRIBUTIONS OF THE INTENTIONAL PUNCTURING AND THE THREE DIFFERENT TRIALS OF THE RANDOM PUNCTURING OF A REGULAR LDPC CODE WITH $\lambda(x) = x^2$ AND $\rho(x) = x^5$ AT A BLOCK LENGTH OF 1024; THE LARGEST CODE RATE IS 0.8

	$\mathbf{G}_0$	$\mathbf{G}_1$	$\mathbf{G}_2$	$\mathbf{G}_3$	$\mathbf{G}_4$	$\mathbf{G}_5$	$\mathbf{G}_6$	$\mathbf{G}_7$	$\mathbf{G}_8$	$\mathbf{G}_9$	$\mathbf{G}_{10}$	$\mathbf{G}_{11}$	$\mathbf{G}_{12}$	$\mathbf{G}_{13}$	$\mathbf{G}_{14}$
Intentional	640	294	78	12	0	0	0	0	0	0	0	0	0	0	0
Random 1	640	108	60	44	45	37	42	33	11	4	0	0	0	0	0
Random 2	640	100	50	38	36	26	28	27	30	28	19	2	0	0	0
Random 3	640	89	46	32	26	18	19	17	20	23	25	25	26	15	3

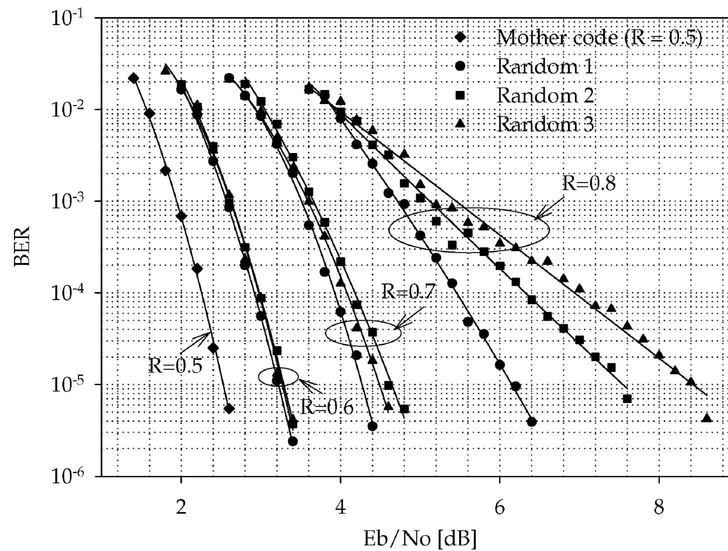

Fig. 3. Randomly punctured LDPC codes with three different random seeds; the circles, squares and triangles correspond to BERs of Random 1, 2, and 3 in Table II , respectively, and the BERs of the half rate mother code at block length 1024 are represented with the diamonds.

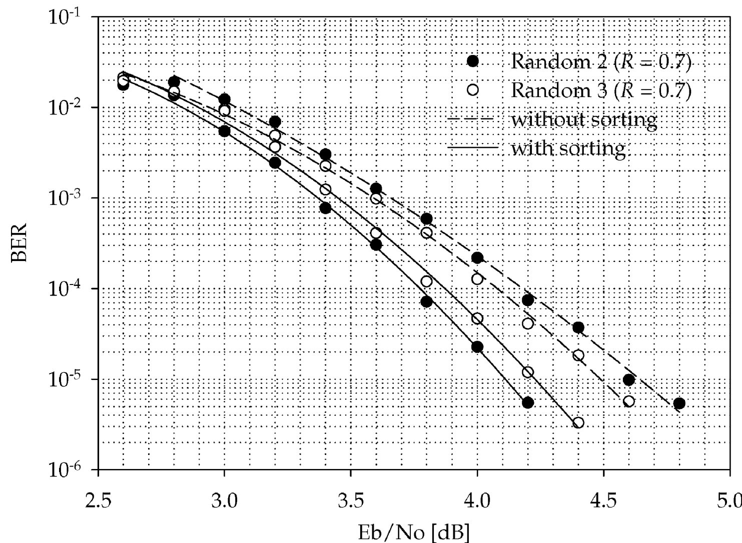

Fig. 4. Randomly punctured LDPC codes at rate 0.7 with (solid lines)/without (dashed lines) the sorting algorithm; the filled and unfilled circles are performances of the punctured LDPC codes with the puncturing distributions Random 2 and 3, respectively and the mother code has a block length of 1024.

intermediate rates can also be improved by applying the sorting algorithm. That is, we can analyze the randomly obtained puncturing distributions and give priority to the nodes with a smaller level of recoverability. In Fig. 4, we compare the performances of the punctured LDPC codes with Random 2 and Random 3 at rate 0.7 with/without the sorting algorithm. The sorting algorithm improves the E_b/N_0 performances of the punctured LDPC codes at a BER of 10^{-5} by 0.5 dB for Random 2 and 0.3 dB for Random 3. After applying the sorting algorithm, the punctured LDPC code with Random 2 has better performance than that of Random 3, which means the punctured LDPC with the smaller level of recoverability at the highest rate has better performance.

One more thing to be noticed in Fig. 3 is the BER performance variations with the different seeds. The required E_b/N_0 to achieve a BER of 10^{-5} at rate 0.8 has a difference of 2.2 dB between the best (Random 1) and worst (Random 3) cases. As mentioned in Section II, random puncturing may delete a significant amount of parity bits in a stopping set,

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

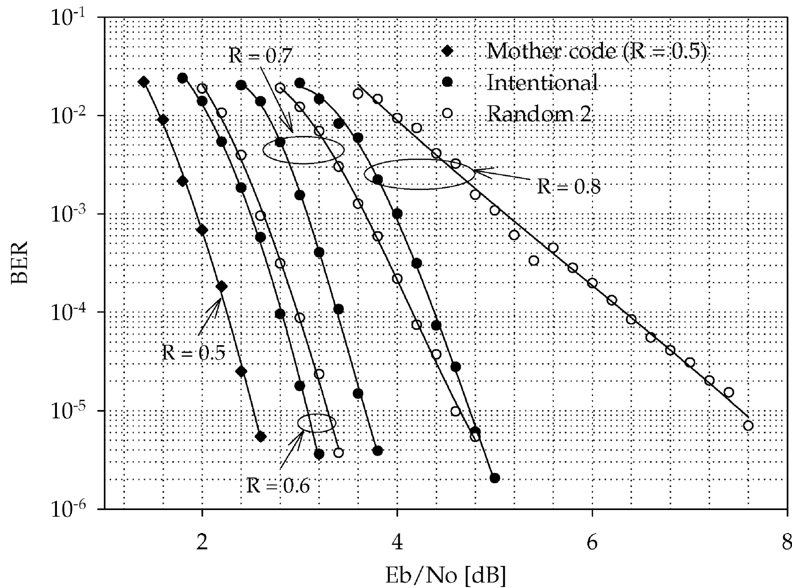

Fig. 5. Comparison between the intentional (filled) and random (unfilled) puncturing of an LDPC code at block length 1024; code rates are $0.5, 0.6, 0.7,$ and 0.8 from the left to the right, the puncturing distributions are from Intentional and Random 2 in Table II and the BERs of the half rate mother code are represented with the diamonds.

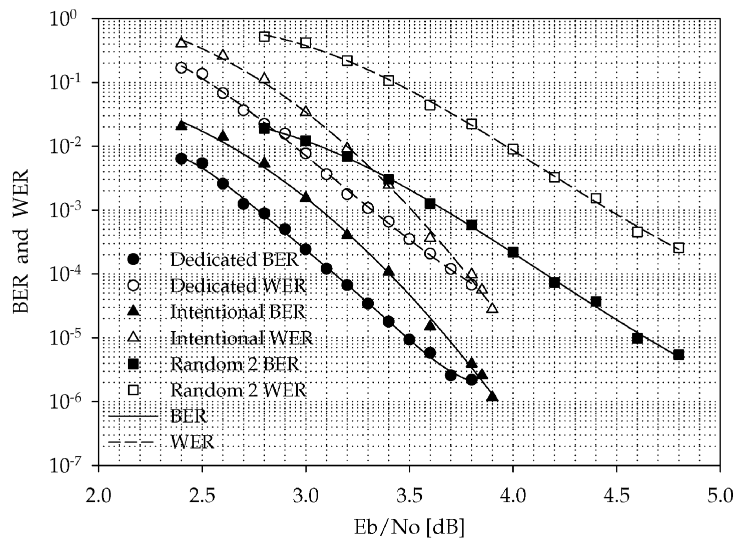

Fig. 6. BERs (filled) and WERs (unfilled) of a dedicated LDPC code, an intentional punctured LDPC code and a randomly punctured LDPC code with rate 0.7 from the left to the right, respectively; the block length of the mother LDPC code is 1024.

which results in a severe performance loss especially at higher rates. It is hard to know whether random puncturing results in catastrophic selections of punctured parity bits without time-consuming computer simulations. However, by analyzing the level of recoverability under the framework of the proposed grouping algorithm, we can predict the performance of randomly punctured LDPC codes and rule out the catastrophic selections. Thus, the proposed idea is also useful for designing randomly punctured LDPC codes.

The intentionally punctured LDPC codes are compared with the randomly punctured ones in Fig. 5. In the comparison, the intentional puncturing outperforms the random puncturing for the rates, where the performance improvement with the proposed algorithm becomes more distinctive at higher rates. At rate 0.8 and a BER of 10^{-5}, the intentionally punctured LDPC code has 3 dB better E_b/N_0 performance than that of the randomly punctured one.

To compare performances of dedicated and the punctured LDPC codes, we design a regular LDPC code ($\lambda(x) = x^2$ and $\rho(x) = x^9$) for rate 0.7 at block length 731. The block length of the dedicated and the punctured LDPC codes should be equal at each rate to ensure a fair comparison. As mentioned, the punctured LDPC codes are decoded on the Tanner graph of their mother code whose block length is 1024 in this case. However, the block length of the dedicated LDPC code is only 71% of the mother code and is decoded on the shorter Tanner graph. The differences in block lengths become more significant at higher code rates. Since dedicated LDPC codes can have smaller minimum distances due to shorter block lengths, it is possible that dedicated LDPC codes show poorer performances at high E_b/N_0 regions. In Fig. 6, we compare the BER and WER performances of the dedicated, randomly punctured and intentionally punctured LDPC codes with code rate 0.7 and block length 731. In the comparisons, the BER/WER performances are in the order of the dedicated, intentionally punctured, and the randomly punctured LDPC codes from the best to the worst. However, at high E_b/N_0 regions, there are crossover points in BER and WER curves of the dedicated and intentionally punctured LDPC codes due to the smaller block length of the dedicated LDPC code. Thus, for very low BERs/WER's, intentionally punctured LDPC

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

TABLE III
GROUP DISTRIBUTIONS OF THE INTENTIONAL PUNCTURING AND THE THREE DIFFERENT TRIALS OF THE RANDOM PUNCTURING OF A REGULAR LDPC CODE WITH $\lambda(x) = x^2$ AND $\rho(x) = x^5$ AT A BLOCK LENGTH OF 4096; THE LARGEST CODE RATE IS 0.8

	$\mathbf{G}_0$	$\mathbf{G}_1$	$\mathbf{G}_2$	$\mathbf{G}_3$	$\mathbf{G}_4$	$\mathbf{G}_5$	$\mathbf{G}_6$	$\mathbf{G}_7$	$\mathbf{G}_8$	$\mathbf{G}_9$	$\mathbf{G}_{10}$	$\mathbf{G}_{11}$	$\mathbf{G}_{12}$	$\mathbf{G}_{13}$
Intentional	2560	1155	323	58	0	0	0	0	0	0	0	0	0	0
Random 1	2560	436	236	188	168	140	133	118	82	29	6	0	0	0
Random 2	2560	398	227	178	128	111	102	99	103	84	69	35	2	0
Random 3	2560	363	207	146	124	110	102	104	115	113	85	52	13	2

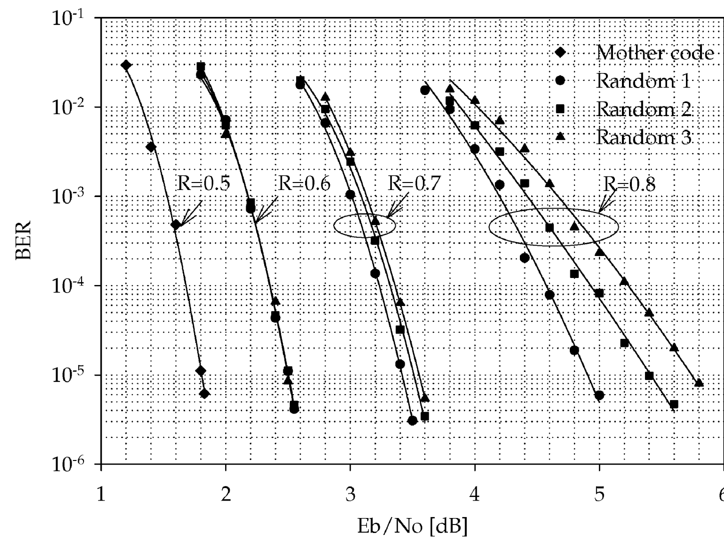

Fig. 7. Randomly punctured LDPC codes with three different random seeds; the circles, squares and triangles correspond to BERs of Random 1, 2, and 3 in Table III , respectively, and the BERs of the half rate mother code at block length 4096 are represented with the diamonds.

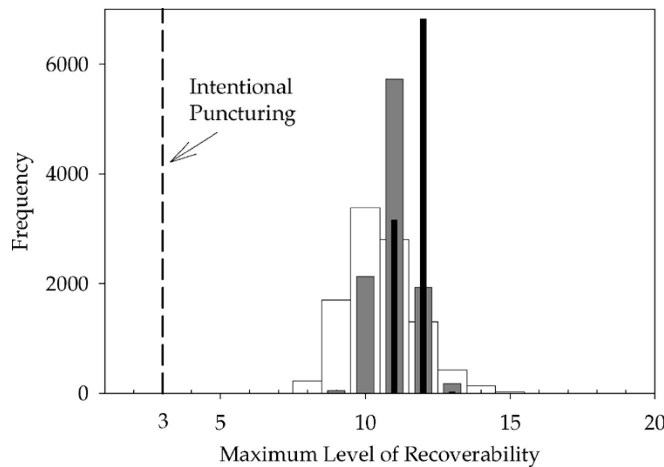

Fig. 8. Histograms of the maximum levels of recoverability; results from 10 000 trials with the regular LDPC codes ($\lambda(x) = x^2$ and $\rho(x) = x^5$) at the block lengths 1024 (unfilled), 4096 (shaded), and 65536 (filled).

codes provide not only structural advantage of the rate compatibility and a lower complexity of encoders and decoders but also better performances than those of dedicated LDPC codes.

The performance variation of random puncturing becomes smaller as the block length increases. To see the performance variation with increasing block lengths, we do the same simulations at a block length of 4096. The group distributions of the intentional and random puncturing are listed in Table III, where the maximum levels of recoverability are 3 and at least 11 for intentional and random puncturing, respectively. We evaluate the BER performances of the three randomly punctured LDPC codes in Fig. 7, where the performance variations among the different random puncturing distributions are noticeably smaller. At rate 0.8, the required E_b/N_0 for a BER of 10^{-5} has a variation of less than 0.9 dB as compared to 2.2 dB in the case of block length 1024. The smaller performance variation can also be predicted by the group distribution in Table III. Thus, locations of punctured parity bits are more important at shorter block lengths.

To see how the maximum level of recoverability changes with increasing block lengths, we design 10 000 different random puncturing distributions for the regular ($\lambda(x) = x^2$ and $\rho(x) = x^5$) LDPC codes at the block lengths of 1024, 4096, and 65536. The levels of recoverability are observed by analyzing the group distributions of

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

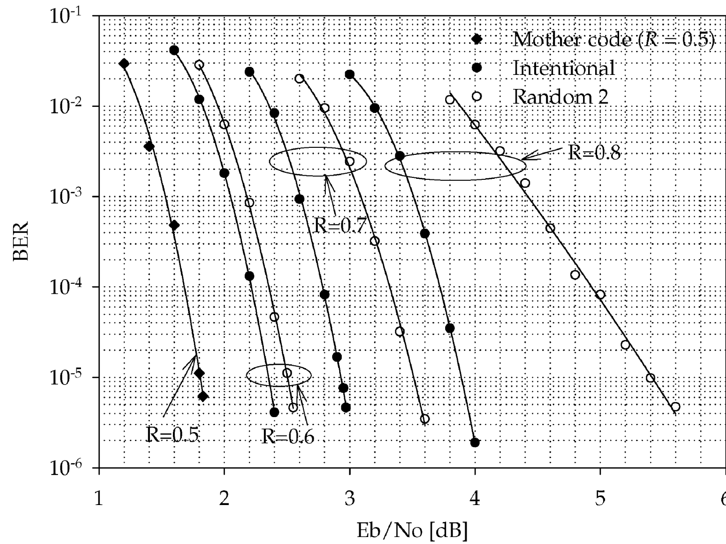

Fig. 9. Comparison between the intentional (filled) and random (unfilled) puncturing of a regular LDPC code ($\lambda(x) = x^2$ and $\rho(x) = x^5$) at block length 4096; code rates are $0.5, 0.6, 0.7,$ and 0.8 from the left to the right, and the puncturing distributions are from Intentional and Random 2 in Table III .

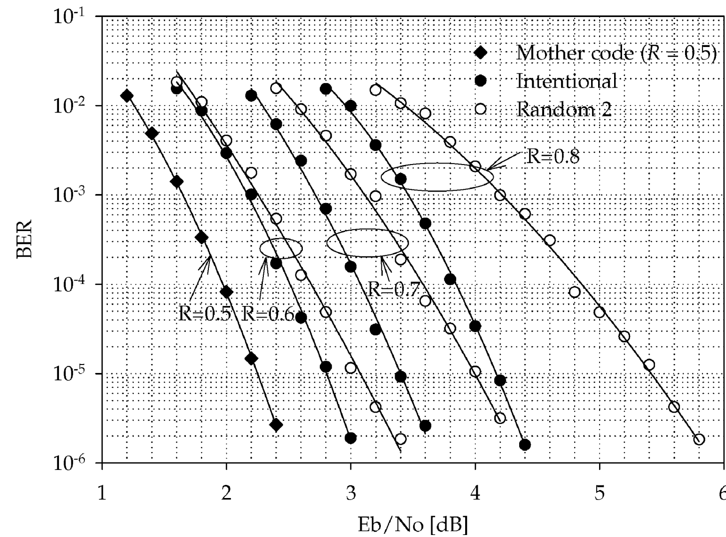

Fig. 10. Comparison between the intentional puncturing (filled dots) and random puncturing(unfilled dots); the half rate irregular mother code (leftmost) has a block length of 1024, and the punctured LDPC codes have rates of $0.6, 0.7,$ and 0.8 from the left to the right.

the puncturing distributions. Histograms of the maximum levels of recoverability with the three different block lengths (1024, 4096, and 65536) are compared in Fig. 8, where most of the time, the maximum level of recoverability is bigger than 10. However, the variations of the maximum level of recoverability become smaller at the longer block lengths. In the extreme case when the block length is 65536, the variation of the maximum level of recoverability is significantly smaller, where 11 and 12 account for over 99% of the occurrences. Thus, we confirm that careful selections of punctured bits are more important at smaller block lengths from a different perspective.

The randomly punctured LDPC codes (Random 2 in Table III) are compared with the intentionally punctured ones (Intentional in Table III) in Fig. 9, where at rate 0.8 and a BER of 10^{-5}, the intentionally punctured LDPC code requires 1.6 dB less E_b/N_0 than that of the randomly punctured one.

B. Simulations With Irregular LDPC Codes

The proposed algorithms are also applicable to irregular LDPC codes. To demonstrate the performance of irregular punctured LDPC codes, we design an irregular LDPC mother code with the rate 0.5 whose degree distribution pair is

$$\lambda(x) = 0.28286x + 0.39943x^2 + 0.31771x^7 \text{ and}$$
$$\rho(x) = 0.6x^5 + 0.4x^6.$$

A parity-check matrices for the irregular LDPC code at block lengths 1024 and 4096 are designed with Progressive Edge Growth (PEG) algorithm [12] to get a better girth distribution. The mother code is punctured randomly and intentionally based on the proposed algorithms. The group distributions are listed in Table IV, where Random (Intentional) 1024 and 4096 indicate the group distributions of random (intentional) puncturing at block lengths 1024 and 4096, respectively. For fair comparisons, we simulate random puncturing with three different random seeds, then pick the one which has middle performance among them as we did in the regular case. The performances of the randomly and intentionally punctured LDPC codes at block lengths 1024 and 4096 are evaluated and compared in Figs. 10 and 11, respectively. Again, the intentionally punctured LDPC codes outperform the randomly punctured LDPC codes at all the rates. At rate 0.8 and a BER of 10^{-5}, the intentionally punctured LDPC codes at block lengths 1024

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

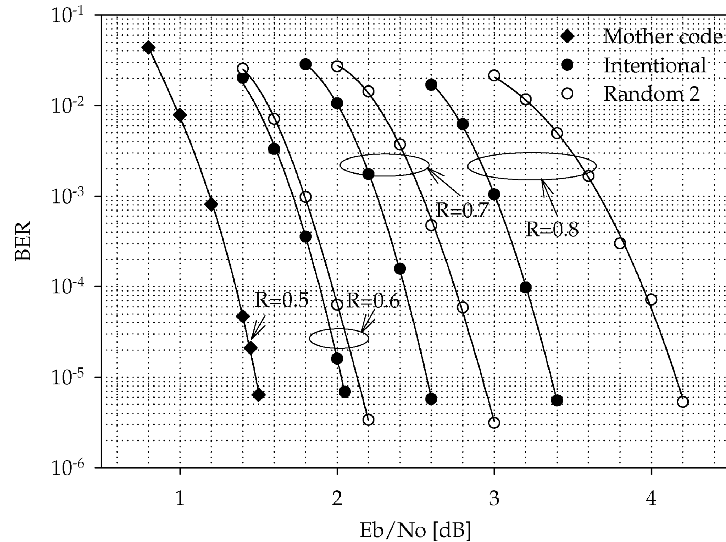

Fig. 11. Comparison between the intentional puncturing (filled dots) and random puncturing(unfilled dots); the half rate irregular mother code (leftmost) has a block length of 4096, and the punctured LDPC codes have rates of 0.6, 0.7, and 0.8 from the left to the right.

TABLE IV
GROUP DISTRIBUTIONS OF THE INTENTIONAL PUNCTURING AND THE THREE DIFFERENT TRIALS OF THE RANDOM PUNCTURING OF AN IRREGULAR LDPC CODE WITH $\lambda(x) = 0.28286x + 0.39943x^2 + 0.31771x^7$ AND $\rho(x) = 0.6x^5 + 0.4x^6$ AT A BLOCK LENGTH OF 1024

	$\mathbf{G}_0$	$\mathbf{G}_1$	$\mathbf{G}_2$	$\mathbf{G}_3$	$\mathbf{G}_4$	$\mathbf{G}_5$	$\mathbf{G}_6$	$\mathbf{G}_7$	$\mathbf{G}_8$	$\mathbf{G}_9$	$\mathbf{G}_{10}$	$\mathbf{G}_{11}$
Intentional 1024	640	333	46	5	0	0	0	0	0	0	0	0
Random 1024	640	77	63	41	41	48	38	26	23	15	10	2
Intentional 4096	2560	1311	206	19	0	0	0	0	0	0	0	0
Random 4096	2560	361	261	214	182	163	132	112	63	37	8	3

and 4096 have 1.25- and 0.8-dB E_b/N_0 improvements over those of the randomly punctured ones, respectively.

V. CONCLUSION

We propose the grouping and sorting algorithms to design rate-compatible punctured LDPC codes at small block lengths. The algorithms are based on the claim that a punctured LDPC code with a smaller level of recoverability has better performance. We mathematically explain why the proposed algorithms provide us with better punctured LDPC codes by introducing the concepts of recovery tree and recovery error probability.

The proposed algorithms are verified by comparing performance of punctured LDPC codes based on the algorithm (called intentionally punctured LDPC codes) with randomly punctured LDPC codes. The intentionally punctured LDPC codes show better BER performances at relatively small block lengths (1024 and 4096), where the performance improvement is more distinctive at smaller block lengths. In the caseof the regular code with block length 1024, the intentionally punctured LDPC has 3 dB better E_b/N_0 performance than that of the randomly punctured one for a BER of 10^{-5} at code rate 0.8. For the longer block length 4096, the intentionally punctured LDPC code outperforms the randomly punctured LDPC code by 1.6 dB at rate 0.8 and a BER of 10^{-5}. That is, the improvement becomes smaller but is still significant.

Another important observation is the performance variation of randomly punctured LDPC codes. Especially, at small block lengths, the variation becomes unacceptable. In our simulations, we observed 2.2 dB performance difference between the best and the worst randomly punctured LDPC codes. Random puncturing makes it possible to puncture a significant amount of parities in a stopping set, which results in poor performance. In the conventional design rule of randomly punctured LDPC codes, the performance variation can be evaluated with time-consuming computer simulations. However, by analyzing group distributions of a random puncturing distributions, we predict their BER performances in a much faster way. The performance variations become smaller at larger block lengths, which is verified by evaluating histograms of the maximum levels of recoverability at three different block lengths, 1024, 4096, and 65536. In the case that we have to use random puncturing, the analysis under the framework of the grouping algorithm gives us a good random puncturing distribution.

We also show that the sorting algorithm can be applied for random puncturing. A random puncturing distribution tells the locations of parities to be punctured but the distribution does not say in which order the parities should be punctured. Although performance at the highest code rate is determined by the maximum level of recoverability, the performances of punctured LDPC codes at intermediate code rates from that of the mother code to the highest code rate depend on the order in which the parities are punctured.

Finally, we apply the proposed algorithm to irregular LDPC codes at block lengths 1024 and 4096. The performance improvements of the intentionally punctured LDPC codes are 1.25 dB for block length 1024 and 0.8 dB for block length 4096 over randomly punctured LDPC codes at code rate 0.8 and a BER of 10^{-5}.

REFERENCES

[1] J. Hagenauer, "Rate-compatible punctured convolutional codes (RCPC codes) and their applications," *IEEE Trans. Commun.*, vol. 36, no. 4, pp. 389–400, Apr. 1988.

[2] J. Ha, J. Kim, and S. W. McLaughlin, "Rate-compatible puncturing of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. IT-50, no. 11, pp. 2824–2836, Nov. 2004.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.

[3] J. Li and K. Narayanan, "Rate-compatible low density parity check codes for capacity-approaching ARQ scheme in packet data communications," in *Proc. Int. Conf. Communications, Internet, and Information Technology (CIIT)*, U.S. Virgin Islands, Nov. 2002, pp. 201–206.

[4] J. Ha and S. W. McLaughlin, "Optimal puncturing distributions for rate-compatible low-density parity-check codes," in *Proc. Int. Symp. Information Theory*, Yokohama, Japan, Jun./Jul. 2003, p. 233.

[5] ——, "Optimal puncturing of irregular low-density parity-check codes," in *Proc. IEEE Int. Conf. Communications*, Anchorage, AK, May 2003, pp. 3110–3114.

[6] M. R. Yazdani and A. H. Banihashemi, "On construction of rate-compatible low-density parity-check codes," *IEEE Commun. Lett.*, vol. 8, no. 3, pp. 159–161, Mar. 2004.

[7] C. Di, D. Proietti, İ. E. Telatar, T. J. Richardson, and R. L. Urbanke, "Finite-length analysis of low-density parity-check codes on the binary erasure channel," *IEEE Trans. Inf. Theory*, vol. 48, no. 6, pp. 1570–1579, Jun. 2002.

[8] T. Tian, C. Jones, and J. D. Villasenor, "Rate-compatible low-density parity-check codes," in *Proc. Int. Symp. Information Theory*, Chicago, IL, Jun./Jul. 2004, p. 152.

[9] M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. IT-27, no. 5, pp. 533–547, Sep. 1981.

[10] D. Klinc, "Rate-compatible punctured LDPC codes: Design and applicability for the ultra wide-band standard," Master's thesis, Technische Univ. München, München, Germany, Sep. 2004.

[11] S.-Y. Chung, T. J. Richardson, and R. L. Urbanke, "Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 657–670, Feb. 2001.

[12] X. Hu, E. Eleftheriou, and D. M. Arnold, "Progressive edge-growth Tanner graphs," in *Proc. IEEE GLOBECOM*, San Antonio, TX, Nov. 2001, pp. 995–1001.

Monomial Bent Functions

Nils Gregor Leander

***Abstract*—In this correspondence, we focus on bent functions of the form $\mathbb{F}_{2^n} \rightarrow \mathbb{F}_2$ where $x \rightarrow \mathrm{Tr}(\alpha x^d)$. The main contribution of this correspondence is, that we prove that for $n = 4r$, r odd, the exponent $d = (2^r+1)^2$ allows the construction of bent functions. This open question has been posed by Canteaut based on computer experiments. As a consequence for each of the well understood families of bent functions, we now know an exponent d that yields to bent functions of the given type.**

***Index Terms*—Bent functions, Boolean functions, monomial Boolean functions, power functions, trace expansion.**

I. Introduction

A complete classification of bent functions is elusive and looks hopeless today. As a first step toward a characterization of all bent functions, we focus on traces of power functions, so called *monomial* Boolean functions. This approach is well known in related areas like almost perfect nonlinear (APN) functions or m-sequences, but has not yet been comprehensively studied for bent functions. This approach turns out to be very fruitful for several reasons. The only known nonnormal bent functions are monomial bent functions (see [2], [3], [5]), demonstrating that the study of monomial functions leads to new classes of bent functions. Furthermore one result of our considerations is, that for each of the well studied families of bent function, there is a monomial bent function belonging to these classes. Moreover, carefully studying the proofs for the monomial bent functions all these families can quite easily be rediscovered. In this sense most of the variety of (at least known) bent functions can already be discovered by the investigation of monomial functions.

We first recall all the known cases of monomial bent functions. In the case of the Dillon–Dobbertin monomial bent function we sketch an algorithmic approach to study the dual of these bent functions.

As one of our main results in this correspondence we present a new class of monomial bent functions, not corresponding to one of the known monomial bent functions. This class was found with the help of computer experiments by Canteaut, who first conjectured that the concrete examples found belong to the new class (see [1]). This exponent actually leads to functions belonging to the Maiorana–McFarland class of bent functions.

A. Preliminaries

Throughout the correspondence let $n = 2k$ be an even integer.

Given a Boolean function $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$, the function

$$a \in \mathbb{F}_2^n \mapsto f^{\mathcal{W}}(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)+\langle a,x \rangle}$$

is called the *Walsh transform* of f. Moreover, the values $f^{\mathcal{W}}(a)$, $a \in \mathbb{F}_2^n$ are called the Walsh coefficients of f.

A measure of the linearity of a Boolean function f with respect to the Walsh transform is defined by

$$\mathrm{Lin}(f) = \max_{a \in \mathbb{F}_2^n} \left| f^{\mathcal{W}}(a) \right|.$$

For n even, f is called *bent* if $\mathrm{Lin}(f) = 2^{n/2}$, which is the minimal value that can occur and we then have $f^{\mathcal{W}}(a) = \pm 2^{n/2}$ for all $a \in \mathbb{F}_2^n$, since

$$\sum_{a \in \mathbb{F}_2^n} f^{\mathcal{W}}(a)^2 = 2^{2n} \quad \text{(Parseval's equation)}.$$

Bent functions always occur in pairs. In fact, given a bent function $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$, we define the *dual* f^* of f by

$$(-1)^{f^*(a)}\, 2^{n/2} = f^{\mathcal{W}}(a).$$

In other words, we consider the signs of the Walsh-coefficients of f. Due to the involution law the Fourier transform is self-inverse. Thus the dual of a bent function is again a bent function, and we have the rule $f^{**} = f$.

We recall some well know families of bent functions. The class of *Maiorana–McFarland* consists of bent functions of the form

$$f : \mathbb{F}_2^k \times \mathbb{F}_2^k \rightarrow \mathbb{F}_2$$
$$f(x,y) = \langle x, \pi(y) \rangle + h(y)$$

where $\pi : \mathbb{F}_2^k \rightarrow \mathbb{F}_2^k$ is a permutation and $h : \mathbb{F}_2^k \rightarrow \mathbb{F}_2$ is an arbitrary function. A special case of Maiorana–McFarland bent functions are the quadratic bent functions.

Manuscript received June 15, 2005; revised November 5, 2005. The material in this correspondence was presented in part at the Workshop on Coding and Cryptography (WCC 2005), Bergen, Norway, March 2005.

The author is with the Ruhr-University Bochum, D-44780 Bochum, Germany (e-mail: leander@cits.rub.de).

Communicated by T. Johansson, Associate Editor for Complexity and Cryptography.

Digital Object Identifier 10.1109/TIT.2005.862121

0018-9448/$20.00 © 2006 IEEE

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on December 1, 2008 at 17:23 from IEEE Xplore. Restrictions apply.