Increasing the Speed of Polar List Decoders

Gabi Sarkis*, Pascal Giard *, Alexander Vardy†, Claude Thibeault‡, and Warren J. Gross*
*Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada.
Email: {gabi.sarkis, pascal.giard}@mail.mcgill.ca, warren.gross@mcgill.ca
†Department of Electrical Engineering, University of California San Diego, La Jolla, CA, USA.
Email: avardy@ucsd.edu
‡Department of Electrical Engineering, École de technologie supérieure, Montréal, Québec, Canada.
Email: claude.thibeault@etsmtl.ca

***Abstract*—In this work, we present a simplified successive cancellation list decoder that uses a Chase-like decoding process to achieve a six time improvement in speed compared to successive cancellation list decoding while maintaining the same error-correction performance advantage over standard successive-cancellation polar decoders. We discuss the algorithm and detail the data structures and methods used to obtain this speed-up. We also propose an adaptive decoding algorithm that significantly improves the throughput while retaining the error-correction performance. Simulation results over the additive white Gaussian noise channel are provided and show that the proposed system is up to 16 times faster than an LDPC decoder of the same frame size, code rate, and similar error-correction performance, making it more suitable for use as a software decoding solution.**

I. Introduction

Polar codes provably achieve the symmetric channel capacity as the code length N increases, when they are decoded with the low-complexity successive-cancellation (SC) decoding algorithm [1]. However, the error-correction performance of SC decoding of polar codes at moderate lengths is mediocre. List [2] and stack decoding [3] have been proposed to improve the error-correction performance without increasing code length.

To further improve the error-correction capability of polar codes, various concatenation schemes have been proposed [4]–[6]. The most successful one is a serial concatenation of a polar code (PC) with a cyclic redundancy check (CRC) code, where the latter is used as an outer code [4]. For a given length N, the resulting code is shown to match or exceed the error-correction performance of turbo [5] as well as low-density parity-check (LDPC) codes [4].

The throughput of SC decoders is low due to the serial nature of the algorithm. This issue was resolved by the simplified successive cancellation (SSC) [7] and the Fast-SSC [8] decoding algorithms. The latter of which has fast hardware [8] and software decoders [9]. Since list decoders are dependent on SC decoders as their major components, their throughput is also very low and they would benefit from improvements to the SC decoders. However, the SSC-based algorithms are not directly applicable to list, and list-CRC, decoding because they present a single estimate of codewords; whereas list decoders require multiple candidates with soft-valued reliabilities.

In this work, we modify the SSC algorithm to present multiple candidate codewords using a Chase-decoding-like process and we present SSC-based list decoders that offer higher throughput (average decoding speed) and lower latency (worst case decoding time) than their SC-based counterpart.

It was shown in [9] that, for software implementations, polar decoders were faster than LDPC decoders with equivalent error-correction performance despite the longer lengths required for polar codes. In this work we show that software list polar decoders are faster than equivalent-performance LDPC decoders at the same code lengths.

We start with a review of polar codes, list and list-CRC decoding, and SSC decoding in Section II. We present our SSC-List decoder in Section III and a higher throughput adaptive version in Section IV. Finally, we discuss the proposed decoder's throughput, latency, and error-correction performance in Section V, comparing it with SC-List and LDPC decoders.

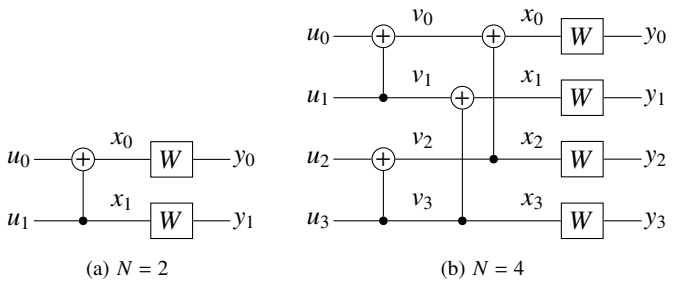
II. Background

A. Polar Codes

Polar codes approach the symmetric capacity of a channel W, as the code length $N \to \infty$, by exploiting channel polarization. Such constructions for $N \in \{2, 4\}$ are shown in Fig. 1. In Fig. 1a, the probability of correctly estimating u_0 given y_0 and y_1 is lower than that of correctly estimating x_0 given y_0, which is in turn lower than that of estimating u_1 given y_0, y_1, and u_0. Longer codes are built by recursively applying the linear polarizing construction. Fig. 1b shows the case of $N = 4$. As the code length increases, the probability of estimating each bit tends to either 0.5 (completely unreliable) or 1 (perfectly reliable). The proportion of the latter bits, called *reliable* bits, approaches the capacity of the channel W as $N \to \infty$ [1].

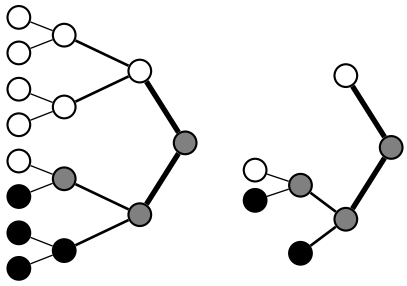
To build an (N, k) polar code, the k information bits are transmitted through the k most reliable locations. The remaining $N-k$ locations correspond to the least reliable bits and are set to 0 and called the *frozen* bits. Determining the reliability of the bit locations depends on the type and conditions of the channel W and is studied for different channels in [1] and [10]. A polar code is constructed for a given channel and channel condition, and can be represented using a generator matrix, $G_N = F_N = F_2^{\otimes \log_2 N}$, where $F_2 = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$ and $\otimes$ is the Kronecker power. In [1], a bit-reversal operator was used so

U.S. Government work not protected by U.S. copyright.



(a) $N = 2$ (b) $N = 4$

Fig. 1: Construction of polar codes of lengths 2 and 4



(a) SC (b) SSC

Fig. 2: Decoder trees corresponding to the SC and SSC decoding algorithms

that $G_N = B_N F_N$; however, it was shown in [9] that not bit-reversing the rows of F_N provides better memory layout and vectorization opportunities for software polar decoders. The frozen bits are indicated by setting their values to 0 in the source vector $\boldsymbol{u}$.

SC decoding works sequentially by estimating an information (non-frozen) bit u_i using the received channel values $\boldsymbol{y}$ and the previously estimated bits $\hat{\boldsymbol{u}}_0^{i-1}$ according to

$$\hat{u}_i = \begin{cases} 0 & \text{when } \Pr[\boldsymbol{y}, \hat{\boldsymbol{u}}_0^{i-1} | \hat{u}_i = 0] \geq \Pr[\boldsymbol{y}, \hat{\boldsymbol{u}}_0^{i-1} | \hat{u}_i = 1]; \\ 1 & \text{otherwise.} \end{cases} \tag{1}$$

B. List-CRC Decoding

Instead of selecting one value for an estimate (1), a list decoder works by assuming both 0 and 1 are estimates of the bit u_i and generates two paths that are decoded using SC decoding. Without a set limit, the number of paths doubles for every information bit, growing exponentially and thus becoming a maximum-likelihood (ML) decoder. To constrain the complexity, a maximum of L distinct paths, the most likely ones, are kept at the end of every step. Thus, a list decoder presents the L most likely codeword candidates after it has estimated all bits. The codeword among the L with the best path reliability metric, i.e. the largest likelihood value, is chosen to be the decoder output.

Noticing that when a polar list decoder failed, the correct codeword was often among the L final candidates, the authors of [2] proposed concatenating a CRC with the information bits, increasing the rate of the polar code to accommodate the additional bits and maintain the overall system rate. The CRC provides the criterion for selection from among the candidate, final codewords. The likelihood of the codewords is only consulted either when two or more candidates satisfy the CRC constraint or when none do. The resulting list-CRC decoder offers a significant improvement in error-correction performance over regular list decoding, to the extent where polar codes were shown to be able to outperform turbo codes [5] and LDPC codes [2] of similar lengths and rates.

List-SC decoding, like SC decoding, remains a sequential process. Moreover, L paths must now be decoded instead of one, increasing the latency from $O(N \log N)$ to $O(LN \log N)$ and decreasing throughput by the same factor [2].

To improve the decoder throughput, adaptive list decoding [11] starts with $L = 1$ and restarts with $L = 2$ if the CRC is not satisfied. The list size is subsequently doubled until the constraint is satisfied or a maximum size, $L_{\max}$, is reached, in which case the candidate with the highest reliability is selected. However, this method significantly increases latency, which becomes

$$\mathcal{L}(\text{A-SC-List}(L_{\max})) = \sum_{l=0}^{\log_2 L_{\max}-1} \mathcal{L}(\text{SC-List}(2^l));$$

where A-SC-List($L_{\max}$) is an adaptive list decoder with a maximum list size of $L_{\max}$ and SC-List(L) is a list decoder with list size L.

C. SSC Decoding

The recursive construction of a polar code makes binary trees a natural representation where each node corresponds to a constituent code of length N_v with a soft input α and an estimated codeword output β. It was observed in [7] that a subtree where all leaf-nodes correspond to frozen bits need not be traversed; its output is known a priori to be the zero-vector. Similarly, that work showed that the ML output of a subtree where all leaf-nodes are information bits, i.e. corresponding to constituent code of rate 1, can be obtained by performing threshold detection on the soft-information input vector. Any tree corresponding to a rate R code is traversed until a rate-0 or a rate-1 code is reached. As a result of these observations, the decoder tree is pruned resulting in the simplified SC (SSC) decoder tree. Decoder trees for the SC and SSC algorithms decoding the same code are shown in Fig. 2a and Fig. 2b respectively. For the SC decoder tree, white leaves correspond to frozen bits and black leaves correspond to information bits. For the SSC decoder tree, white and black leaves are called rate-0 and rate-1 nodes respectively.

The decoder tree is further pruned in [8], [12] by recognizing more types of constituent codes, resulting in lower latency and greater throughput for both hardware [8] and software decoders [9].

III. SSC-List Decoder

In this section, we present an SSC-based list decoding algorithm and discuss its implementation details. Rate-0 nodes are

ignored and their soft-input is not calculated by their parent, and rate-R nodes operate as in SC-List decoding. Therefore we focus on rate-1 nodes. We will show in Section V-D that the proposed decoder is six times as fast the SC-List decoder.

It should be noted that this decoder was implemented using log-likelihoods (LL) to represent bit reliabilities.

A. Chase-Like Decoding of Rate-1 Nodes

The function of the rate-1 node decoder is to provide a list of the L most reliable candidate codewords given its LL input α, where each LL $\alpha[i]$ consists of $\alpha_0[i]$ and $\alpha_1[i]$. For a constituent code of rate 1 and length N_v, there exists 2^{N_v} candidate codewords, rendering an exhaustive search impractical for all but the smallest of such codes. Therefore, we employ the candidate generation method of Chase decoding [13].

Maximum-likelihood decoding of a rate-1 constituent code is performed on a bit-by-bit basis [7], i.e.

$$\beta[i] = \begin{cases} 0 & \text{when } \alpha_0[i] \geq \alpha_1[i], \\ 1 & \text{otherwise.} \end{cases}$$

To provide a list of candidate codewords, the least reliable bits—determined using $r[i] = |\alpha_0[i] - \alpha_1[i]|$—of the ML decision are flipped individually. Simulation results have shown that two-bit errors must also be considered. Therefore, the list of candidates is augmented with codewords that differ from the ML decision by two of the least reliable bits.

The list of candidates is pruned to include, at most, L candidates. This is accomplished by discarding the least reliable candidates, where the reliability of a path x with an estimated output β is calculated according to

$$R_x = \sum_i \alpha_{\beta[i]}[i]. \tag{2}$$

B. Implementation of Rate-1 Decoders

The rate-1 decoder starts by initializing its set of candidates to an empty set. Then, for each source path p, it will calculate and store the ML decision and generate a set of candidate forks. Once the decoder has iterated over all source paths, it will store the up to L most reliable paths from the ML decisions and the candidate forks, discarding the rest. The top-level function corresponds to Algorithm 1. The algorithm shows how the bit reliabilities r and the path reliability R are calculated in tandem with the ML decision. The candidate forks are appended to the candidate set when there are fewer than L candidates already stored; otherwise, they replace other candidates with lower reliability.

Algorithm 2, shows how candidates are appended to the set. Empirically, it was observed that not all bits need to be considered when enumerating potential single-bit errors, limiting the search to the c least reliable bits was sufficient, as in Chase decoding [13]. Therefore, this method performs a partial sort to find those bits. The candidates are generated by flipping those bits individually, and their reliabilities are calculated according to

$$\begin{aligned} R_i &= R_p - r[i] = R_p - |\alpha_0^p[i] - \alpha_1^p[i]| \\ &= R_p - \max(\alpha_0^p[i], \alpha_1^p[i]) + \min(\alpha_0^p[i], \alpha_1^p[i]). \end{aligned}$$

Algorithm 1 decodeRate1Code

```
candidates = {}
for p ∈ sourcePaths do
    R_p = 0
    for i = 0 to N_v − 1 do
        β^p[i] = arg max_x(α_x^p[i])
        r[i] = |α_0^p[i] − α_1^p[i]|
        R_p = R_p + max(α_0^p[i], α_1^p[i])
    end for
    storePath(p, R_p)
    if candidates.count < L then
        appendCandidates(candidates)
    else
        replaceCandidates(candidates)
    end if
end for
mergeBestCandidates(candidates)
```

Algorithm 2 appendCandidates

```
//Appends forks of path p to candidates with constraint c
partialSort(r, c)
for i = 0 to c − 1 do //Single-bit errors
    R_i = R_p − r[i]
    bitsToFlip = {bitIndex(i)}
    candidates.insert(p, R_i, bitsToFlip)
end for
for i = 0 to c − 2 do //Two-bit errors
    for j = i + 1 to c − 1 do
        R_ij = R_p − r[i] − r[j]
        bitsToFlip = {bitIndex(i), bitIndex(j)}
        candidates.insert(p, R_ij, bitsToFlip)
    end for
end for
```

Since a candidate might be later discarded if it is not among the L most reliable paths, it is important for speed reasons to minimize the amount of information stored about each candidate. Therefore only the information needed to construct a new path is stored in the candidate set: the source path p, the path reliability R_i, and the location of bits in which it differs from the source path *bitsToFlip*. Candidates with two-bit errors are generated in a similar manner by iterating over all unique pairs of bits among the c least reliable ones. To remove conditionals from the inner loops in this algorithm, the set of candidates is allowed to contain more than L candidates. Selecting the correct number of candidates to store as new paths, is performed at a later point by the rate-1 decoder.

When the set of candidates already contains L or more candidates, the decoder will only replace an existing candidate with a new one when the latter is more reliable. Algorithm 3 describes this process. It iterates over candidates with single-bit and two-bit errors and adds them to the set of candidates if their reliability is greater than the minimum stored in the set. Every time a new candidate is added to the set, the least reli-

Algorithm 3 replaceCandidates

```
//Replaces the least reliable candidates with more reliable forks of path p.
partialSort(r, c)
for i = 0 to c − 1 do //Single-bit errors
    R_i = R_p − r[i]
    if R_i > min(candidates.reliability) then
        bitsToFlip = {bitIndex(i)}
        candidates.insert(p, R_i, bitsToFlip)
        candidates.remove(candidates.leastReliable)
    end if
end for
for i = 0 to c − 2 do //Two-bit errors
    for j = i + 1 to c − 1 do
        R_ij = R_p − r[i] − r[j]
        if R_ij > min(candidates.reliability) then
            bitsToFlip = {bitIndex(i), bitIndex(j)}
            candidates.insert(p, R_ij, bitsToFlip)
            candidates.remove(candidates.leastReliable)
        end if
    end for
end for
```

able one is removed. This prevents the set of candidates from storing a large number of candidates that will be discarded later. Similar to Algorithm 2, it was observed via simulations that using a constraint c to limit the candidate search space did not noticeably affect error-correction performance while doubling the decoding speed.

The mergeBestCandidates() method retains the most reliable L paths by copying and modifying the ML decision of their source path.

In Algorithms 2 and 3, it is observed that the most common operations performed on the set of candidates, denoted *candidates*, are insertion, deletion, and finding the minimum. Red-Black trees are well suited for implementing such a data structure since all these operations are performed in $O(\log_2 N_v)$ time in the worst case [14]. In addition, mergeBestCandidates() requires that the most reliable candidates be indicated and red-black trees store their contents sorted by key.

IV. Adaptive SSC-List-CRC Decoder

List decoders have a high latency and a low throughput that are constant regardless of the channel condition. Based on the observation that at high E_b/N_0 values the average list size L required to successfully correct a frame is low, an adaptive SC-List-CRC decoder was proposed in [11].

In Section III, we introduced an SSC-List decoding algorithm that has a lower latency and greater throughput than the SC-List decoding algorithm. Despite the improvement, the throughput of that decoder is still significantly lower than a Fast-SSC decoder [8]. We thus propose using an adaptive SSC-List-CRC decoding algorithm similar to that of [11]:

1) Decode a frame using the Fast-SSC algorithm.
2) Verify the validity of the estimated codeword by calculating its CRC.
3) Stop the decoding process if the CRC is satisfied, otherwise move to the next step.
4) Relaunch the decoding process using the SSC-List algorithm and generate a list of L candidate codewords sorted by their path reliability metric.
5) Pick the most reliable candidate among the list generated above that satisfies the CRC.
6) If none of the L candidates satisfy the CRC, pick the codeword with the best path reliability metric.

The difference between this proposed algorithm and that of [11] is that in order to reduce latency, the list size is not increased gradually. Instead, it is changed from $L = 1$, i.e. using the Fast-SSC decoder, to $L = L_{\max}$. Therefore, the latency (worst case) is

$$\begin{aligned}&\mathcal{L}(\text{A-SSC-List}(L_{\max}))\\ &= \mathcal{L}(\text{SSC-List}(L_{\max})) + \mathcal{L}(\text{Fast-SSC})\\ &\approx \mathcal{L}(\text{SSC-List}(L_{\max})).\end{aligned}$$

Since the latency of the single SSC-List decoder using $L = L_{\max}$ is much greater than that of the Fast-SSC decoder.

Let $\mathcal{L}(\text{L}) = \mathcal{L}(\text{SSC-List}(L_{\max}))$ and $\mathcal{L}(\text{F}) = \mathcal{L}(\text{Fast-SSC})$, and denote as FER_F the frame-error rate (FER) at the output of the Fast-SSC decoder. The expression for the information throughput (on average) of the proposed adaptive SSC-List decoder when decoding a code with dimension k is

$$\mathcal{T} = \frac{k}{(1-\text{FER}_\text{F})\mathcal{L}(F) + \text{FER}_\text{F}\mathcal{L}(L)};$$

where it can be observed that for sufficiently low FER_F value, the throughput will determined mostly by the speed of the Fast-SSC decoder.

V. Simulation Results

A. Methodology

All error-correction performance results were obtained for the binary-input additive white Gaussian noise (AWGN) channel with random codewords and binary phase-shift keying (BPSK) modulation. Polar codes were constructed using the technique described in [10] and systematic encoding was used [15]. The throughput and latency values were measured on an Intel Core-i7 2600 running at 3.4 GHz using the methodology described in [9]. Finally, as mentioned in Section II-B, in list-CRC decoders, the rate of the polar code is adjusted to maintain the same overall system rate. For example, when comparing a list-CRC decoder with the (2048, 1723) LDPC decoder and a 32-bit CRC is utilized, the polar code used is PC(2048, 1755) and the overall system rate remains $1723/2048$. All the simulation results presented in this section had the constraint $c = 2$.

B. Choosing a Suitable CRC Length

As discussed in Section II-B, a CRC serves as better criterion for selecting the correct codeword from the final L

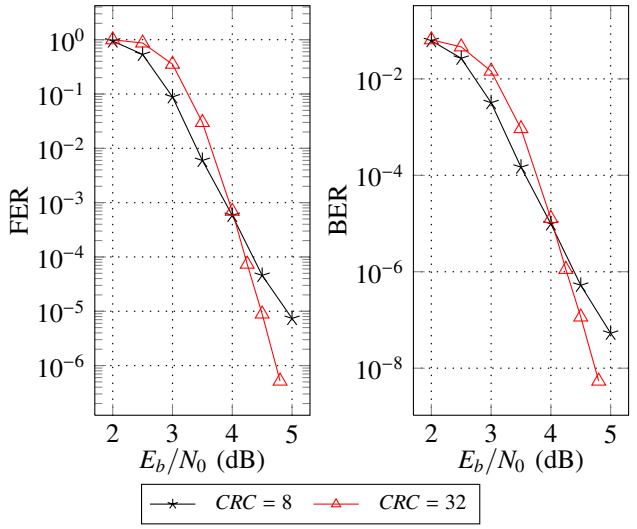


Fig. 3: The effect of CRC length on the error-correction performance of $(1024, 860)$ list-CRC decoders with $L = 128$.

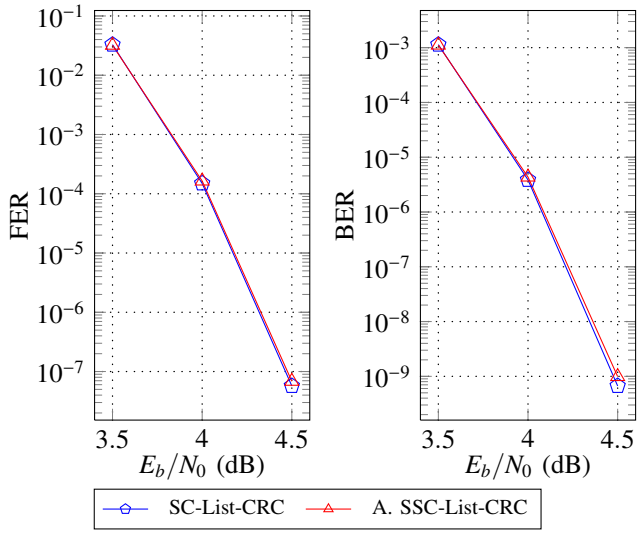


Fig. 4: Error-correction performance of $(2048, 1723)$ SC- and SSC-List-CRC decoders with $L = 32$.

candidates even when its likelihood is not the largest. The length of the chosen CRC has an impact on the error-rate that varies with E_b/N_0. Fig. 3 shows the error-correction performance of a $(1024, 860)$ system consisting of polar code concatenated with a CRC of length 8 or 32 and decoded with a list-CRC decoder with list size $L = 128$. It shows that a polar code concatenated with the shorter CRC will perform better at lower E_b/N_0 values but will eventually achieve higher error-rates than the polar code concatenated with the longer CRC.

Therefore, the length of the CRC can be chosen to improve error-correction performance in the targeted E_b/N_0 or BER/FER range.

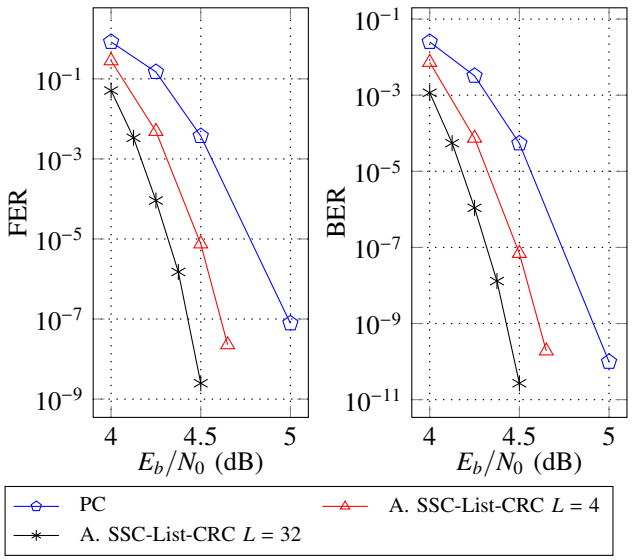


Fig. 5: Error-correction performance of (32768, 29492) polar code, denoted PC, with that of (32768, 29492) list-CRC decoders with different list sizes.

TABLE I: Latency and information throughput comparison for list-based decoders using a $(2048, 1723)$ polar+CRC code.

Decoder	L	Latency (ms)	$\mathcal{T}$ (kbps)
SC-List-CRC	32	23	74
SSC-List-CRC		3.3	522
SC-List-CRC	128	97	17
SSC-List-CRC		16	107

C. Error-Correction Performance

It is known that concatenating CRC improves the error-correction performance of polar list decoders. In this section, we first show that the error-correction performance of the proposed SSC-List-CRC decoder is the same as that of the SC-List-CRC decoder in Fig. 4. We then demonstrate that the benefits for longer codes are still significant.

As shown in Fig. 5, for a $(32768, 29492)$ polar code, the use of the proposed algorithm results in a coding gain greater than 0.3 dB and 0.5 dB at a FER of 10^{-5} over the Fast-SSC algorithm for $L = 4$ and $L = 32$, respectively. It can be seen that the curves are diverging as E_b/N_0 is increasing, and thus the coding gain is growing as well.

D. Comparison with the SC-List-CRC Decoder

List decoders have latency and throughput that are constant across E_b/N_0 values. Table I shows these values for the SC-List-CRC and SSC-List-CRC decoders for two different list sizes when decoding a (2048, 1723) polar+CRC-32 code. At $L = 32$, the SSC-based decoder is approximately 7 times as fast the SC-based one. At $L = 128$, it is 6 times as fast.

E. Comparison with LDPC Codes

To the best of our knowledge, the fastest CPU-based LDPC decoder in literature is [16]. Its information throughput for

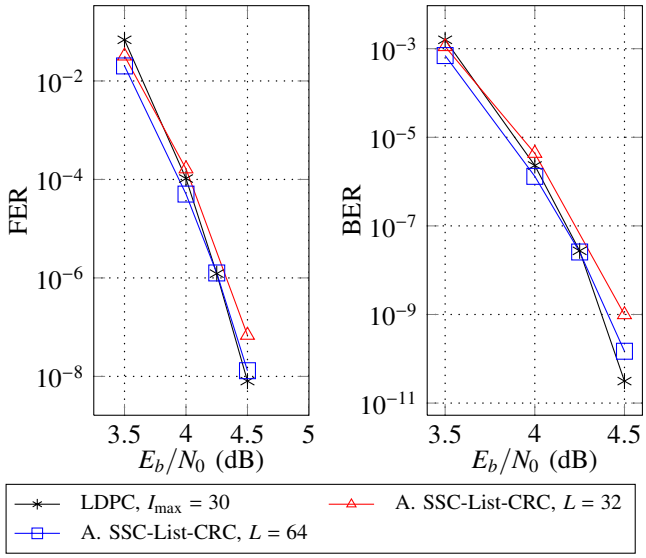


Fig. 6: Error-correction performance of (2048, 1723) LDPC and polar list-CRC decoders.

TABLE II: Information throughput in Mbps of the proposed decoder compared to an LDPC decoder at different E_b/N_0 values.

Decoder	$\mathcal{T}$ (Mbps)		
	3.5 dB	4.0 dB	4.5 dB
LDPC	1.04	1.81	2.25
A. SSC-List-CRC ($L = 64$)	0.42	2.36	36.6
A. SSC-List-CRC ($L = 32$)	0.91	4.90	54.0

a (1024, 512) LDPC running on two CPU cores was 345 kbps with a fixed number of iterations (10). The information throughput of a scaled-min-sum decoder we have developed was 555 kbps when running with the same number of iterations but on a single CPU core of similar speed. Therefore, we use our LDPC decoder for throughput comparison in this work, enabling early termination to further improve its throughput.

A polar list-CRC decoder with a 32-bit CRC and $L = 32$ is within 0.1 dB of the error-correction performance of the 10GBASE-T (802.3an) LDPC code with identical length and dimension (2048, 1723), as shown in Fig. 6. When L is increased to 64, the polar list-CRC and the LDPC decoders have similar performance. In these simulations the LDPC decoder was using the scaled-min-sum algorithm with a maximum of 30 iterations ($I_{\max} = 30$) and a scaling factor of 0.5.

Table II shows the throughput values for the proposed adaptive SSC-List-CRC decoder with $L = 64$ compared with that of our offset-min-sum LDPC decoder with $I_{\max} = 30$ and an adaptive SC-List-CRC decoder at different E_b/N_0 values when decoding (2048, 1723) codes. We first observe that throughput of the decoders improves as E_b/N_0 increases since they employ early termination methods: syndrome checking for the LDPC decoder and CRC checking for the adaptive SSC-List one. The LDPC decoder is faster than the proposed decoder at $E_b/N_0 = 3.5$ dB. At 4.0 dB and 4.5 dB however, the adaptive SSC-List decoder becomes 1.3 and 16 times as fast as the LDPC one, respectively. The latency was 5.5 ms and 7.1 ms for the LDPC and adaptive SSC-List decoders, respectively. The table also shows the throughput of the adaptive SSC-List decoder with $L = 32$, which at 3.5 dB runs at 87% the speed of the LDPC decoder and is 2.7 and 24 times as fast at 4.0 dB and 4.5 dB, respectively. The latency of this decoder is 3.3 ms and, as mentioned in this section, its error-correction performance is within 0.1 dB of the LDPC decoder.

VI. Conclusion

In this work, we presented a new polar list decoding algorithm whose software implementation is at least 6 times as fast as the original list decoder. We also showed an adaptive decoder which significantly increased the throughput to the point where its throughput is up to 16 times that of an LDPC decoder of the same length, rate, and similar error-correction performance. We believe that such improvements in speed, combined with the error-correction performance, make the adaptive SSC-List decoder a viable option for use as a decoder in software defined radio and other applications. Future work will focus on switching the list decoder to log-likelihood ratios (LLRs) from LLs in order to further reduce latency.

References

[1] E. Arıkan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[2] I. Tal and A. Vardy, "List decoding of polar codes," in *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, July 2011, pp. 1–5.

[3] K. Niu and K. Chen, "Stack decoding of polar codes," *Electronics letters*, vol. 48, no. 12, pp. 695–697, 2012.

[4] I. Tal and A. Vardy, "List decoding of polar codes," *CoRR*, vol. abs/1206.0050, 2012.

[5] K. Niu and K. Chen, "CRC-aided decoding of polar codes," *IEEE Commun. Lett.*, vol. 16, no. 10, pp. 1668–1671, 2012.

[6] H. Mahdavifar, M. El-Khamy, J. Lee, and I. Kang, "On the construction and decoding of concatenated polar codes," in *IEEE Int. Symp. on Inf. Theory (ISIT)*, 2013.

[7] A. Alamdar-Yazdi and F. R. Kschischang, "A simplified successive-cancellation decoder for polar codes," *IEEE Commun. Lett.*, vol. 15, no. 12, pp. 1378–1380, 2011.

[8] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast polar decoders: Algorithm and implementation," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 946–957, May 2014.

[9] P. Giard, G. Sarkis, C. Thibeault, and W. J. Gross, "Fast software polar decoders," in *IEEE Int. Conf. on Acoustics, Speech, and Signal Process. (ICASSP)*, 2014, *to appear*.

[10] I. Tal and A. Vardy, "How to construct polar codes," *CoRR*, vol. abs/1105.6164, 2011. [Online]. Available: http://arxiv.org/abs/1105.6164

[11] B. Li, H. Shen, and D. Tse, "An adaptive successive cancellation list decoder for polar codes with cyclic redundancy check," *IEEE Commun. Lett.*, vol. 16, no. 12, pp. 2044–2047, 2012.

[12] G. Sarkis and W. J. Gross, "Increasing the throughput of polar decoders," *IEEE Commun. Lett.*, vol. 17, no. 4, pp. 725–728, 2013.

[13] D. Chase, "Class of algorithms for decoding block codes with channel measurement information," *IEEE Trans. Inf. Theory*, vol. 18, no. 1, pp. 170–182, 1972.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[15] E. Arıkan, "Systematic polar coding," *IEEE Commun. Lett.*, vol. 15, no. 8, pp. 860–862, 2011.

[16] G. Falcão, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 309–322, 2011.