Fast List Decoders for Polar Codes

Gabi Sarkis, Pascal Giard, *Student Member, IEEE*, Alexander Vardy, *Fellow, IEEE*, Claude Thibeault, *Senior Member, IEEE*, and Warren J. Gross, *Senior Member, IEEE*

***Abstract*—Polar codes asymptotically achieve the symmetric capacity of memoryless channels, yet their error-correcting performance under successive-cancellation (SC) decoding for short and moderate length codes is worse than that of other modern codes such as low-density parity-check (LDPC) codes. Of the many methods to improve the error-correction performance of polar codes, list decoding yields the best results, especially when the polar code is concatenated with a cyclic redundancy check (CRC). List decoding involves exploring several decoding paths with SC decoding, and therefore tends to be slower than SC decoding itself, by an order of magnitude in practical implementations. In this paper, we present a new algorithm based on unrolling the decoding tree of the code that improves the speed of list decoding by an order of magnitude when implemented in software. Furthermore, we show that for software-defined radio applications, our proposed algorithm is faster than the fastest software implementations of LDPC decoders in the literature while offering comparable error-correction performance at similar or shorter code lengths.**

***Index Terms*—polar codes, list decoding, software decoders, software-defined radio, LDPC.**

I. Introduction

Polar codes, proposed by Arıkan [1], achieve the symmetric capacity of memoryless channels as the code length $N \to \infty$ using the low-complexity successive-cancellation (SC) decoding algorithm. Their error-correction performance, however, is mediocre for codes of short and moderate lengths (a few thousand bits) and is worse than that of other modern codes, such as low-density parity-check (LDPC) codes. To improve their performance, polar codes are concatenated with a cyclic redundancy check (CRC) as an outer code and decoded using the list decoding algorithm ("list-CRC"). The resulting error-correction performance can exceed that of LDPC codes of similar length [2].

However, list-CRC decoding comes with a downside: the sequential "bit-by-bit" decoding order of the SC algorithm limits the speed of practical implementations, which further decreases with increasing list size L. The complexity of SC decoding is $O(N \log N)$, however a list decoder has a higher complexity of $O(LN \log N)$. The result is that practical hardware and software implementations of list decoders have low throughput that is an order of magnitude lower than the fastest SC decoder hardware [3], which achieves information throughout of 1.0 Gbps at 100 MHz in FPGA. The fastest belief propagation polar decoder is also faster: it achieves 2.34 Gbps at 300 MHz in 65nm CMOS [4]. On the other hand, reported hardware list decoder implementations achieve coded throughputs of 285 Mbps at 714 MHz for $N = 1024$ and $L = 2$ [5], and 335 Mbps at 847 MHz for $N = 1024$ and $L = 2$ [6]. For a list size $L = 16$, the fastest decoder has a coded throughput of 220 Mbps at a clock frequency of 641 MHz [7].

The key to increasing the speed of SC decoders is to break the serial constraint imposed by successive cancellation. In [8], it was recognized that certain decoding steps in SC decoding were redundant for certain groups of bits that could instead be estimated simultaneously, given appropriate implementations. In that approach, called *simplified successive cancellation* (SSC), groups of frozen bits do not need to be explicitly decoded, since their values are already known (usually zero), and groups of information bits can be estimated by thresholding, instead of serial successive cancellation. When viewing the polar code in a tree representation, it is easy to see that the code is a concatenation of smaller constituent codes. Groups of frozen bits can be viewed as comprising a "Rate-0" code and information bit groups are a "Rate-1" code. Later work further increased the speed of SC decoding by parallel decoding some of the other "Rate-R" codes in the tree [3], [9]. The Fast-SSC algorithm in [3] considers a variety of different constituent codes, such as single-parity-check (SPC) and repetition codes, decoding them with parallel hardware, estimating several bits per clock cycle. The first portion of this work describes how the Fast-SSC decoding algorithm was adapted for use in the context of list decoding.

The second part describes how this algorithm performs when implemented on a general purpose processor using single-instruction multiple-data (SIMD) instructions. Such systems were shown to have fast software SC decoders: the decoder in [10] employs inter-frame parallelism, decoding many frames in parallel, to achieve information throughput of 2.2 Gbps and latency of 26 μs. Alternatively, intra-frame parallelism targeting low-latency implementations was used by [11] to reach an information throughput up to 1.3 Gbps with 1 μs latency. In addition, encoding of polar codes is a low complexity, $O(N \log N)$, operation that is well suited for software implementation as it does not require permutation of data [12].

The low encoding complexity combined with the good error-correction performance of list-CRC decoding will significantly improve the communication ability of wireless sensors networks (WSN) using software-defined radio (SDR). The sensor nodes benefit from the ability to use shorter codes, reducing transmission time and energy as well as the ability to

Manuscript received. The associate editor coordinating the review of this paper and approving it for publication was ...

G. Sarkis, P. Giard, and W. J. Gross are with the Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada (e-mail: {gabi.sarkis, pascal.giard}@mail.mcgill.ca, warren.gross@mcgill.ca).

A. Vardy is with the Department of Electrical Engineering, University of California San Diego, La Jolla, CA. USA (e-mail: avardy@ucsd.edu).

C. Thibeault is with the Department of Electrical Engineering, École de technologie supérieure, Montréal, Québec, Canada (e-mail: claude.thibeault@etsmtl.ca).

reduce transmission power. Alternatively, instead of reducing transmission power, one can increase the distance between the nodes and base stations, reducing the number of base stations in the process. The nodes also benefit from the very low complexity of polar encoders [12], [13]. The base stations, which generally have less stringent energy requirements, can use general purpose processors, including SIMD capable embedded ARM processors, to implement the proposed list-CRC decoding algorithm and to process data on site. This reduces the cost and development time of the WSN and increases its flexibility as a result of the SDR components. This work could also be used in other SDR applications that do not have the scale to justify a custom hardware implementation but where a throughput in the tens of Mbps is desirable. Quantum key distribution is such an example where a general purpose processor or a graphics processing unit is used to perform error correction [14]. Multiple SDR systems either include a general purpose Intel processor or must be connected to a computer [15]–[17], providing target platforms where our proposed algorithm can be used.

This work expands and improves on previous conference publications [18] and [19]. The algorithm in this paper has been reformulated in terms of log-likelihood ratios (LLRs), which yields speed improvements over the preliminary work in [18]. Furthermore, the conference paper implemented a list decoding algorithm based on SSC decoding (list-SSC), while this work develops the Fast-SSC algorithm for list decoding (list-Fast-SSC) and implements it, yielding further performance improvements. In addition, a general path metric is derived from codeword likelihoods, which is then used as the basis for calculating all the specialized decoders' output metrics. Finally, unrolling [19] is applied to list decoders in this work. The results show that our improved list decoding algorithm results in a speedup of 11.9 times compared to LLR-based list-SC decoding. In addition to the decoder in [18], those of [20] and [21] also perform multi-bit decisions and decoding. The main difference between them and the proposed decoder is that the former perform multi-bit decision for any constituent code of length M bits using an exhaustive-search decoding algorithm. Whereas the proposed decoder uses specialized, low-complexity algorithms to decode any constituent code to which these algorithms apply, regardless of the constituent code length. A version of [21] limited to 2-bit constituent codes appears in [22].

It should be noted that multi-bit decoding for Rate-1, Rate-0, and repetition constituent codes was proposed in the context of likelihood-based Reed-Muller (RM) decoders [23] and likelihood-based RM list decoders [24]. This work differs from [24] in that it targets LLR-based list decoders in the context of polar codes and recognizes more special constituent multi-bit decoders. The algorithms introduced in this work focus on low implementation complexity, especially for SIMD processors and parallel hardware.

This work starts by reviewing the construction of polar codes and the list-CRC and the Fast-SSC decoding algorithms in Section II. We then describe how to generate a software polar decoder amenable to vectorization in Section II-D. Section III introduces the proposed list decoding algorithm and a software implementation is described in Section IV. The speed and error-correction performance of the proposed decoder are studied in Section VI and compared to those of LDPC codes of the 802.3an [25] and 802.11n [26] standards in Section VII. In the second comparison, we show that polar codes can match or exceed the speed and error-correction performance of software LDPC decoders while using shorter codes.

II. Background

A. Polar Codes

A polar code of length N is constructed recursively from two polar codes of length $N/2$. Successive-cancellation (SC) decoding provides a bit estimate $\hat{u}_i$ using the channel output y_0^{N-1} and the previously estimated bits $\hat{u}_0^{i-1}$ according to

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

As $N \to \infty$, the probability of correctly estimating a bit approaches 1 or 0.5. This is the channel polarization phenomenon that is exploited by polar codes, which use reliable bit locations to store information bits and set the unreliable, called frozen, bits to zero. As a result, when the SC decoder is estimating a bit u_i, it is zero if the bit is frozen, or is calculated according to (1).

Fig. 1a shows the graph of an (8, 4) polar code where frozen bits are labeled in gray and information bits in black. The SC decoder can also be viewed as a tree that is traversed depth first. Such a tree is illustrated in Fig. 1b, where each sub-tree corresponds to a constituent code. The white nodes correspond to frozen bits, and the black ones to information bits. The gray nodes represent the concatenation operations combining two constituent codes.

Two types of messages are passed along the edges of the tree in the decoder: soft reliability values—LLRs in this work,—α, and hard bit estimates, β. When a node corresponding to a constituent code of length N_v receives the reliability values from its parent, represented using LLRs, the output to its left child is calculated according to the F function:

$$\begin{aligned} \alpha_l[i] &= F(\alpha_v[i], \alpha_v[i + N_v/2]) \\ &= 2\operatorname{atanh}(\tanh(\alpha_v[i]/2)\tanh(\alpha_v[i + N_v/2]/2)) \\ &\approx \operatorname{sgn}(\alpha_v[i])\operatorname{sgn}(\alpha_v[i + N_v/2])\min(|\alpha_v[i]|, |\alpha_v[i + N_v/2]|); \end{aligned} \tag{2}$$

where the approximation is the min-sum approximation.

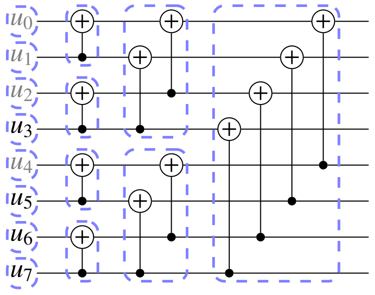
Once the output of the left child β_l is available the message to right one is calculated using the G function

$$\begin{aligned} \alpha_r[i] &= G(\alpha_v[i], \alpha_v[i + N_v/2], \beta_l[i]) \\ &= \alpha_v[i + N_v/2] - (2\beta_l[i] - 1)\alpha_v[i]. \end{aligned} \tag{3}$$

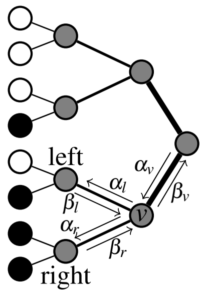
Finally, when β_r is known, the node's output is computed as

$$\beta_v[i] = \begin{cases} \beta_l[i] \oplus \beta_r[i] & \text{when } i < N_v/2; \\ \beta_r[i - N_v/2] & \text{otherwise;} \end{cases} \tag{4}$$

where $\oplus$ is an XOR operation and we refer to the operation as the *Combine* operation.



(a) Graph



(b) Decoder tree

Fig. 1: The graph of an (8, 4) polar code and its corresponding tree representation.

The output β_v of a frozen node is always zero, and is calculated using threshold detection for an information node:

$$\beta_v = h(\alpha_v) = \begin{cases} 0 & \text{when } \alpha_v \geq 0; \\ 1 & \text{otherwise.} \end{cases} \tag{5}$$

B. List-CRC Decoding

When estimating an information bit, a list decoder continues decoding along two paths, the first assumes that '0' was the correct bit estimate, and the second '1'. Therefore at every information bit, the decoder doubles the number of possible outcomes up to a predetermined limit L. When the number of paths exceeds L, the list is pruned by retaining only the L most reliable paths. When decoding is over, the estimated codeword with the largest reliability metric is selected as the decoder output. It was observed in [2] that using a CRC as the primary criterion for selecting the final decoder output, increased the error-correction performance significantly. In addition, the CRC enables the use of a adaptive decoder where the list size starts at two and is gradually increased until the CRC is satisfied or a maximum list size is reached [27].

Initially, polar list decoders used likelihood [2] and log-likelihood values [28] to represent reliabilities. Later, log-likelihood ratios (LLRs) were used in [6] to reduce the amount of memory used by a factor of two and to reduce the processing complexity. In addition to the messages and operations presented in Section II-A, the algorithm of [6] stores a reliability metric PM_l^i for each path l that is updated for every estimated bit i according to:

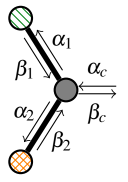
$$\text{PM}_l^i = \begin{cases} \text{PM}_l^{i-1} & \text{if } \hat{u}_i = h(\alpha_v), \\ \text{PM}_l^{i-1} - |\alpha_v| & \text{otherwise.} \end{cases} \tag{6}$$

It is important to note that the path metric is updated when encountering both information and frozen bits.

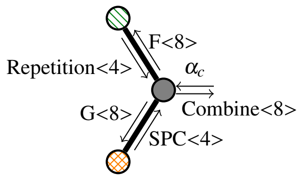
C. Fast-SSC Decoding

The SC decoder traverses the code tree until reaching leaf nodes corresponding to codes of length one before estimating a bit. This was found to be superfluous as the output of sub-trees corresponding to constituent codes of rate 0 or rate 1 of any length can be estimated without traversing their sub-trees [8]. The output of a rate-0 node is known a priori to be an all-zero vector of length N_v; while that of rate-1 can be found by applying threshold detection element-wise on α_v so that

$$\beta_v[i] = h(\alpha_v[i]) = \begin{cases} 0 & \text{when } \alpha_v[i] \geq 0; \\ 1 & \text{otherwise.} \end{cases} \tag{7}$$



(a) Messages



(b) Operations

Fig. 2: Fast-SSC decoder graph for an $(8, 4)$ polar code.

The Fast-SSC algorithm utilizes low-complexity maximum-likelihood (ML) decoding algorithms to decode constituent repetition and single-parity check (SPC) codes instead of traversing their corresponding sub-trees [3], [9].

The ML-decision for a repetition code is

$$\beta_v[i] = \begin{cases} 0 & \text{when } \sum_j \alpha_v[j] \geq 0; \\ 1 & \text{otherwise.} \end{cases} \tag{8}$$

The SPC decoder performs threshold detection (7) on its output to calculate the intermediate value HD. The parity of HD is calculated using modulo-2 addition and the least reliable bit is found according to

$$j = \arg\min_j |\alpha_v[j]|.$$

The final output of the SPC decoder is

$$\beta_v[i] = \begin{cases} \text{HD}[i] \oplus \text{parity} & \text{when } i = j; \\ \text{HD}[i] & \text{otherwise.} \end{cases} \tag{9}$$

Fig. 2 shows a Fast-SSC decoder tree for the (8, 4) code, indicating the messages passed in the decoder and the operations used to calculate them.

The Fast-SSC decoder and its software implementation [11] utilize additional specialized constituent decoders that are not used in this work as they did not improve decoding speed. In addition, the operations mentioned in this section and implemented in [11] present a single output and therefore cannot be applied directly to list decoding. In this work, we will show how they are adapted to present multiple candidates and used in a list decoder.

D. Unrolling Software Decoders

The software list decoder in [18] is run-time configurable, i.e. the same executable is capable of decoding any polar code without recompilation. While flexible, this limits the achievable decoding speed. In [19], it was shown that generating a decoder for a specific polar code yielded significant speed improvement by replacing branches with straight-line code and increasing the utilization of SIMD instructions. This process is managed by a developed CAD tool that divides the

Listing 1 Loop-based (8, 4) Fast-SSC Decoder

```
for (unsigned int i = 0; i < operation_count; ++i) {
  operation_processor = fetch_operation_processor(i);
  operation_processor.execute($\alpha_v$, &$\alpha_r$, &$\alpha_l$, $\beta_l$, $\beta_r$, &$\beta_v$);
}
```

Listing 2 Unrolled (8, 4) Fast-SSC Decoder

```
$\alpha_1$ = $F$<8>($\alpha_c$);
$\beta_1$ = Repetition<4>($\alpha_1$);
$\alpha_2$ = $G$<8>($\alpha_c$, $\beta_1$);
$\beta_2$ = SPC<4>($\alpha_2$);
$\beta_c$ = $Combine$<8>($\beta_1$, $\beta_2$);
```

process into two parts: decoder tree optimization, and C++ code generation.

For the list decoder in this paper we applied this optimization tool using a subset of the nodes available to the complete Fast-SSC algorithm: Rate-0 (Frozen), Rate-1 (information), repetition, and SPC nodes. The decoder tree optimizer traverses the decoder tree starting from its root. If a sub-tree rooted at the current node has a higher decoding latency than an applicable Fast-SSC node, it is replaced with the latter. If there are not any Fast-SSC nodes that can replace the current tree, the optimizer moves to the current node's children and repeats the process.

Once the tree is optimized, the corresponding C++ code is generated. All functions are passed the current N_v value as a template parameter, enabling vectorization and loop unrolling.

Listings 1 and 2 show a loop-based decoder and an unrolled one for the (8, 4) code in Fig. 2, respectively. In the loop-based decoder, both iterating over the decoding operations and selecting an appropriate decoding function (called an operation processor) to execute involve branches. In addition, the operation processor does not know the size of the data it is operating on at compile-time; and as such, it must have another loop inside. The unrolled decoder can eliminate these branches since both the decoder flow and data sizes are known at compile-time.

III. Proposed List-Decoding Algorithm

When performing operations corresponding to a rate-R node, a list decoder with a maximum list size L performs the operations F (2), G (3), and *Combine* (4) on each of the paths independently. It is only at the leaf nodes that interaction between the paths occurs: the decoder generates new paths and retains the most reliable L ones.

A significant difference between the baseline SC-list decoder and the proposed algorithm is that each path in the former generates two candidates, whereas in the latter, the leaf nodes with sizes larger than one can generate multiple candidates for each path.

All path-generating nodes store the candidate path reliability metrics in a priority queue so that the worst candidate can be quickly found and replaced with a new path when appropriate. This is an improvement over [18], where path reliability metrics were kept sorted at all times by using a red-black (RB) tree. The most common operation in candidate selection is locating the path with the minimum reliability, which is an $O(\log L)$ operation in RB-trees, the order of the remaining candidates is irrelevant. A heap-backed priority queue provides $O(1)$ minimum-value look up and $O(\log L)$ insertion and removal, and is therefore more efficient than an RB tree for the intended application.

Algorithm 3 Candidate selection process

```
for $s \in sourcePaths$ do
    Generate candidates.
    Store reliability of all candidates except the ML one.
    Store ML decision.
end for
for $p \in candidates$ do
    if fewer than $L$ candidates are stored then
        Store $p$.
    else if $\mathrm{PM}_p^t$ < min. stored candidate reliability then
        Replace min. reliability candidate with $p$.
    end if
end for
```

In this section, we describe how each node generates its output paths and calculates the corresponding reliability metrics. The process of retaining the L most reliable paths is described in Algorithm 3. Performing the candidate selection in two passes and storing the ML decisions first are necessary to prevent candidates generated by the first few paths from overwriting the input for later ones.

A. Candidate Generation and Reliability

The aim of the proposed algorithm is to directly generate candidates without traversing sub-trees whenever possible. To achieve this goal, we use the candidate-enumeration method of Chase decoding [29] to provide a list of candidate paths at the output of a rate-1 decoder.

The log-likelihood of a candidate codeword β_j is

$$\begin{aligned} l(\beta_j) &= \sum_i (1 - 2\beta_j[i])\alpha_v[i] \\ &= \sum_i (1 - 2\beta_j[i])\operatorname{sgn}(\alpha_v[i])\,|\alpha_v[i]| \\ &= \sum_i (1 - 2\left|\beta_j[i] - h(\alpha_v[i])\right|)\,|\alpha_v[i]| \end{aligned} \tag{10}$$

The factor

$$\begin{aligned} (1 - 2\beta_j[i])\operatorname{sgn}(\alpha_v[i]) &= 1 - 2\left|\beta_j[i] - h(\alpha_v[i])\right| \\ &= \begin{cases} +1 & \text{when } \beta_j[i] = h(\alpha_v[i]), \\ -1 & \text{otherwise.} \end{cases} \end{aligned}$$

The ML candidate codeword is

$$\beta_{\mathrm{ML}} = \arg\max_{\beta_i \in C} l(\beta_i),$$

where C is the set of all codewords. The other candidates are generated by flipping bits relative to the ML decision, both individually and simultaneously, subject to the constraint that the candidate is a valid codeword.

To ensure that the codeword log-likelihood values remain ≤ 0, we offset $l(\beta_j)$ by $\sum_i |\alpha_v[i]|$. In addition, we scale the metric by a factor of 0.5. The resulting codeword metric becomes

$$\begin{aligned} l'(\beta_j) &= \frac{l(\beta_j) - \sum_i |\alpha_v[i]|}{2} \\ &= \frac{\sum_i (1 - 2\left|\beta_j[i] - h(\alpha_v[i])\right|)|\alpha_v[i]| - \sum_i |\alpha_v[i]|}{2} \\ &= \frac{\sum_i (1 - 2\left|\beta_j[i] - h(\alpha_v[i])\right| - 1)|\alpha_v[i]|}{2} \\ &= -\sum_i \left|\beta_j[i] - h(\alpha_v[i])\right| |\alpha_v[i]| . \end{aligned} \tag{11}$$

This metric states that a codeword is penalized for any difference between it and the vector calculated from α_v using (7).

When starting from a source path s with reliability PM_s^{t-1}, the reliability of the path corresponding to the codeword β_j is

$$\text{PM}_j^t = \text{PM}_s^{t-1} - \sum_i \left|\beta_j[i] - h(\alpha_v[i])\right| |\alpha_v[i]| . \tag{12}$$

All specialized decoders generate their candidates based on this metric by restricting potential codewords.

B. Rate-0 Decoders

Rate-0 nodes do not generate new paths; however, like their length-1 counterparts in SC-list decoding [6], they alter path reliability values. In [6], the path metric was updated according to

$$\text{PM}_l^i = \begin{cases} \text{PM}_l^{i-1} & \text{if } h(\alpha_v) = 0, \\ \text{PM}_l^{i-1} - |\alpha_v| & \text{otherwise.} \end{cases}$$

The all-zero codeword, $\beta_j[i] = 0, \forall i$, is the only valid codeword. Therefore, based on (12), the output path metric is

$$\text{PM}_j^t = \text{PM}_s^{t-1} - \sum_i h(\alpha_v[i])|\alpha_v[i]|. \tag{13}$$

An alternative formulation for (13) is

$$\text{PM}_l^t = \text{PM}_l^{t-1} - \sum_{i=0}^{N_V-1} |\max(\alpha_v[i], 0)|. \tag{14}$$

C. Rate-1 Decoders

A decoder for a length N_v rate-1 constituent code can provide up to 2^{N_v} candidate codewords. This approach is impractical as it scales exponentially in N_v. The Chase-II decoding algorithm considers only a limited set of the least-reliable bits to generate its candidates [29]. We use the same method to limit the complexity of rate-1 decoders when enumerating the candidates selected for consideration in (12).

The maximum-likelihood decoding rule for a rate-1 code is (7). Additional candidates are generated by flipping the least reliable bits both independently and simultaneously. Empirically, we found that considering only the two least-reliable bits, whose indexes are denoted $\min_1$ and $\min_2$, is sufficient to match the performance of SC list decoding. Therefore, for each source path s, the proposed rate-1 decoder generates four candidates with the following reliability values

$$\begin{aligned} \text{PM}_0^t &= \text{PM}_s^{t-1}, \\ \text{PM}_1^t &= \text{PM}_s^{t-1} - |\alpha_v[\min_1]|, \\ \text{PM}_2^t &= \text{PM}_s^{t-1} - |\alpha_v[\min_2]|, \\ \text{PM}_3^t &= \text{PM}_s^{t-1} - |\alpha_v[\min_1]| - |\alpha_v[\min_2]|; \end{aligned}$$

where PM_0^t corresponds to the ML decision, PM_1^t to the ML decision with the least-reliable bit flipped, PM_2^t to the ML decision with the second least-reliable bit flipped, and PM_3^t to the ML decision with the two least-reliable bits flipped.

D. SPC Decoders

Codewords of an SPC code must satisfy the even parity constraint, i.e. $\sum_i \beta_j[i] = 0$ where the summation is performed using binary arithmetic. As such, 2^{N_v-1} candidate codewords are available, leading to impractical implementations with exponential complexity. Similar to the rate-1 decoders, we use the Chase-II candidate generation to limit the number of candidates. Simulation results, presented in Section VI, showed that flipping combinations of the four least-reliable bits caused only a minor degradation in error-correction performance for $L < 16$ and SPC code length > 4. The error-correction performance change was negligible for smaller L values. Increasing the number of least-reliable bits under consideration decreased the decoder speed to the point where not utilizing specialized decoders for SPC codes of length > 4 yielded a faster decoder.

We define q as an indicator function so that $q = 1$ when the parity check is satisfied and 0 otherwise. Using this notation, the reliabilities of the candidates, in an expanded form of (12), are

$$\begin{aligned} \text{PM}_0^t &= \text{PM}_s^{t-1} - (1-q)|\alpha_v[\min_1]| \\ \text{PM}_1^t &= \text{PM}_0^t - q|\alpha_v[\min_1]| - |\alpha_v[\min_2]|, \\ \text{PM}_2^t &= \text{PM}_0^t - q|\alpha_v[\min_1]| - |\alpha_v[\min_3]|, \\ \text{PM}_3^t &= \text{PM}_0^t - q|\alpha_v[\min_1]| - |\alpha_v[\min_4]|, \\ \text{PM}_4^t &= \text{PM}_0^t - |\alpha_v[\min_2]| - |\alpha_v[\min_3]|, \\ \text{PM}_5^t &= \text{PM}_0^t - |\alpha_v[\min_2]| - |\alpha_v[\min_4]|, \\ \text{PM}_6^t &= \text{PM}_0^t - |\alpha_v[\min_3]| - |\alpha_v[\min_4]|, \\ \text{PM}_7^t &= \text{PM}_0^t - q|\alpha_v[\min_1]| \\ &\quad - |\alpha_v[\min_2]| - |\alpha_v[\min_3]| - |\alpha_v[\min_4]|; \end{aligned}$$

where PM_0^t is reliability of the ML decision calculated according to (9). The remaining reliability values correspond to flipping an even number of bits compared to the ML decision so that the single-parity check constraint remains satisfied. Applying this rule when the input already satisfies the SPC constraints generates candidates where no bits are flipped, two bits are flipped, and four bits are flipped. Otherwise, one and three bits are flipped.

When the list size $L = 2$, at most two candidates from any given source path are retained. Therefore, only the two most reliable candidates, corresponding to PM_0^t and PM_1^t, need to be evaluated for each each source path, regardless of the length of the SPC code. This is supported by the simulation results shown in Section VI.

E. Repetition Decoders

A repetition decoder has two possible outputs: the all-zero and the all-one codewords and, according to (12), their reliabilities are

$$\begin{aligned}\text{PM}_0^t &= \text{PM}_s^{t-1} - \sum_i h(\alpha_v[i])|\alpha_v[i]|,\\ \text{PM}_1^t &= \text{PM}_s^{t-1} - \sum_i |1 - h(\alpha_v[i])|\,|\alpha_v[i]|.\\ &= \text{PM}_s^{t-1} - \sum_i h(-\alpha_v[i])|\alpha_v[i]|.\end{aligned}$$

where PM_0^t and PM_1^t are the path reliability values corresponding to the all-zero and all-one codewords, respectively. The all-zero reliability is penalized for every input corresponding to a '1' estimate, i.e. negative LLR; and the all-one for every input corresponding to a '0' estimate. These two equations can be rewritten as

$$\begin{aligned}\text{PM}_0^t &= \text{PM}_s^{t-1} - \sum_i |\min(\alpha_v[i], 0)|,\\ \text{PM}_1^t &= \text{PM}_s^{t-1} - \sum_i |\max(\alpha_v[i], 0)|;\end{aligned}$$

The ML decision is found according to $\arg\max_i(\text{PM}_i^t)$, which is the same as performing (8).

IV. Implementation

In this section we describe the methods used to implement our proposed algorithm on an x86 CPU supporting SIMD instructions. We created two versions: one for CPUs that support the AVX instructions, and the other using SSE for CPUs that do not. For brevity, we only discuss the AVX implementation when both implementations are similar. In cases where they differ significantly, both implementations are presented.

We use 32-bit floating-point (`float`) to represent the binary-valued β, in addition to the real-valued α, since it improves vectorization of the g operation as explained in Section IV-C.

A. Memory Layout for α Values

Memory is organized into stages: the input to all constituent codes of length N_v is stored in stage $S_{\log_2 N_v}$. Due to the sequential nature of the decoding process, only N_v values need to be stored for a stage since old values are discarded when new ones are available. For example, the input to SPC node of size 4 in Fig. 2, will be stored in S_2, overwriting the input to the repetition node of the same size.

When using SIMD instructions, memory must be aligned according the SIMD vector size: 16-byte and 32-byte boundaries for SSE and AVX, respectively. In addition, each stage is padded to ensure that its size is at least that of the SIMD vector. Therefore, a stage of size N_v is allocated $\max(N_v, V)$ elements, where V is the number of α values in a SIMD vector, and the total memory allocated for storing α values is

$$N + L \sum_{i=0}^{\log_2 N - 1} \max(2^i, V)$$

LLR (`float`) elements; where the values in stage $S_{\log_2 N}$ are the channel reliability information that are shared among all paths and L is the list size.

During the candidate forking process at a stage S_i, a path p is created from a source path s. The new path p shares all the information with s for stages $\in [S_{\log N}, S_i)$. This is exploited in order to minimize the number of memory copy operations by updating memory pointers when a new path is created [2]. For stages $\in [S_0, S_i]$, path p gets its own memory since the values stored in these stages will differ from those calculated by other descendants of s.

B. Memory Layout for β Values

Memory for β values is also arranged into stages. However, since calculating β_v (4) requires both β_l and β_r, values from left and right children are stored separately and do not overwrite each other. Once alignment and padding are accounted for, the total memory required to store β values is

$$L * (N + 2 \sum_{i=0}^{\log_2 N - 1} \max(2^i, V)).$$

As stage $S_{\log N}$ stores the output candidate codewords of the decoder, which will not be combined with other values, only L, instead of $2L$, memory blocks are required.

Stored β information is also shared by means of memory pointers. Candidates generated at a stage S_i share all information for stages $\in [S_0, S_i)$.

C. Rate-R and Rate-0 Nodes

Exploiting the sign-magnitude floating-point representation defined in IEEE-754, allows for efficient vectorized implementation of the f operation (2). Extracting the sign and calculating the absolute values in (2) become simple bit-wise `AND` operations with the appropriate mask.

The g operation can be written as

$$\begin{aligned}&g(\alpha_v[i], \alpha_v[i + N_v/2], \beta_l[i])\\ =&\alpha_v[i + N_v/2] + \beta_l[i] * \alpha_v[i].\end{aligned}$$

If we use $\beta \in \{+0.0, -0.0\}$ instead of $\{0, 1\}$, the g operation (3) can be implemented as

$$\alpha_v[i + N_v/2] + \beta_l[i] \oplus \alpha_v[i]. \qquad (15)$$

Replacing the multiplication ($*$) with an XOR ($\oplus$) operation in (15) is possible due to the sign-magnitude representation of IEEE-754.

Listing 4 shows the corresponding AVX implementations, originally presented in [11], [19], of the f and g functions using the SIMD intrinsic functions provided by GCC. For clarity of exposition, m256 is used instead of __m256 and the _mm256_ prefix is removed from the intrinsic function names.

Listing 4 Vectorized f and g functions

```
template<unsigned int $N_v$>
void $G$($\alpha^*$ $\alpha_{in}$, $\alpha^*$ $\alpha_{out}$, $\beta^*$ $\beta_{in}$) {
  for (unsigned int i = 0; i < $N_v/2$; i += 8) {
    m256 $\alpha_l$ = load_ps($\alpha_{in}$ + i);
    m256 $\alpha_r$ = load_ps($\alpha_{in}$ + i + $N_v/2$);
    m256 $\beta_l$ = load_ps($\beta_{in}$ + i);
    m256 $\alpha'_l$ = xor_ps($\beta_l$, $\alpha_l$);
    m256 $\alpha_o$ = add_ps($\alpha_r$, $\alpha'_l$);
    store_ps($\alpha_{out}$ + i, $\alpha_o$);
  }
}
template<unsigned int $N_v$>
void $F$($\alpha^*$ $\alpha_{in}$, $\alpha^*$ $\alpha_{out}$) {
  for (unsigned int i = 0; i < $N_v/2$; i += 8) {
    m256 $\alpha_l$ = load_ps($\alpha_{in}$ + i);
    m256 $\alpha_r$ = load_ps($\alpha_{in}$ + i + $N_v/2$);
    m256 sign = and_ps(xor_ps($\alpha_l$, $\alpha_r$), SIGN_MASK);
    m256 $|\alpha_l|$ = andnot_ps($\alpha_l$, SIGN_MASK);
    m256 $|\alpha_r|$ = andnot_ps($\alpha_r$, SIGN_MASK);
    m256 $\alpha_o$ = or_ps(sign, min_ps($|\alpha_l|$, $|\alpha_r|$));
    store_ps($\alpha_{out}$ + i, $\alpha_o$);
  }
}
```

Listing 5 Path reliability update in Rate-0 decoders.

```
m256 ZERO = set1_ps(0.0);
m256 PMv = ZERO;
for (unsigned int i = 0; i < $N_v/2$; i += 8) {
  PMv = add_ps(PMv, min_ps(load_ps($\alpha_{in}$ + i), ZERO));
}
PM = $\sum_i$ PMv[$i$];
```

Rate-0 decoders set their output to the all-zero vector using store instructions. The path reliability (PM) calculation (14) is implemented as in Listing 5.

D. Rate-1 Nodes

Since $\beta \in \{+0.0, -0.0\}$ and α values are represented using sign-magnitude notation, the threshold detection in (7) is performed using a bit mask (SIGN_MASK).

Sorting networks can be implemented using SIMD instructions to efficiently sort data on a CPU [30]. For rate-1 nodes of length 4, a partial sorting network (PSN), implemented using SSE instructions, is used to find the two least reliable bits. For longer constituent codes, the reliability values are reduced to two SIMD vectors: the first, v_0 containing the least reliable bit and the second, v_1, containing the least reliable bits not included in v_0. When these two vectors are partially sorted using the PSN, $\min_2$ will be either the second least-reliable bit in v_0 or the least-reliable bit in v_1.

E. Repetition Nodes

The reliability of the all-zero output PM_0^t is calculated by accumulating the $\min(\alpha_v[i], 0.0)$ using SIMD instructions. Similarly, to calculate PM_1^t, $\max(\alpha_v[i], 0.0)$ are accumulated.

F. SPC Nodes

For SPC decoders of length 4, all possible bit-flip combinations are tested; therefore, no sorting is performed on the bit reliability values. For longer codes, a sorting network is used to find the four least-reliable bits. When $L = 2$, only the two least reliable bits need to be located. In that case, a partial sorting network is used as described in Section IV-D.

Since the SPC code of length 2 is equivalent to the repetition code of the same length, we only implement the latter.

V. Adaptive Decoder

The concatenation with a CRC provides a method to perform early termination analogous to a syndrome check in belief propagation decoders. In [27], this was used to gradually increase the list size. In this work, we first decode using a Fast-SSC polar decoder, and if the CRC is not satisfied, switch to the list decoder with the target $L_{\max}$ value. The latency of this adaptive approach is

$$\mathcal{L}(A_{\max}) = \mathcal{L}(L) + \mathcal{L}(F); \tag{16}$$

where $\mathcal{L}(L)$ and $\mathcal{L}(F)$ are the latencies of the list and Fast-SSC decoders, respectively.

The improvement in throughput stems from the Fast-SSC having lower latency than the list decoder. Once the frame-error rate (FER_F) at the output of the Fast-SSC decreases below a certain point, the overhead of using that decoder is compensated for by not using the list decoder. The resulting information throughput in bit/s is

$$\mathcal{T} = \frac{k}{(1 - \text{FER}_\text{F})\mathcal{L}(F) + \text{FER}_\text{F}\mathcal{L}(L)}. \tag{17}$$

Determining whether to use adaptive decoder depends on the expected channel conditions and the latency of the list decoder as dictated by $L_{\max}$. This is demonstrated in the comparison with the LDPC codes in Section VII.

VI. Performance

A. Methodology

All simulations were run on a single core of an Intel i7-2600 CPU with a base clock frequency of 3.4 GHz and a maximum turbo frequency of 3.8 GHz. Software-defined radio (SDR) applications typically use only one core for decoding, as the other cores are reserved for other signal processing functions [31]. The decoder was inserted into a digital communication link with binary phase-shift keying (BPSK) and an additive white Gaussian noise (AWGN) channel with random codewords.

Throughput and latency numbers include the time required to copy data to and from the decoder and are measured using the high precision clock from the Boost Chrono library. We report the decoder speed with turbo frequency boost enabled, similar to [32].

We use the term polar-CRC to denote the result of concatenating a polar code with a CRC. This concatenated code is decoded using a list-CRC decoder. The dimension of the polar code is increased to accommodate the CRC while maintaining the overall code rate; e.g. a (1024, 512) polar-CRC code with an 8-bit CRC uses a (1024, 520) polar code.

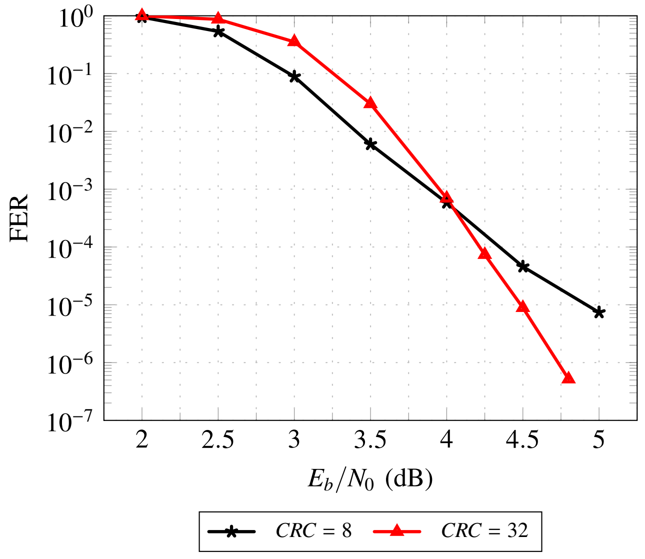


Fig. 3: The effect of CRC length on the error-correction performance of $(1024, 860)$ list-CRC decoders with $L = 128$.

B. Choosing a Suitable CRC Length

Using a CRC as the final output selection criterion significantly improves the error-correction performance of the decoder. The length of the chosen CRC also affects the error-correction performance depending on the channel conditions. Fig. 3 demonstrates this phenomenon for an $(1024, 860)$ polar-CRC code using 8- and 32- bit CRCs and $L = 128$. Such a large list size was chosen to ensure that any observed differences are solely due to the change in the CRC length and could not be counteracted by increasing the list size further. The figure shows that the performance is better at lower E_b/N_0 values when the shorter CRC is used. The trend is reversed for better channel conditions where the 32-bit CRC provides an improvement > 0.5 dB compared to the 8-bit one.

Therefore, the length of the CRC can be selected to improve performance for the target channel conditions.

C. Error-Correction Performance

The error-correction performance of the proposed decoder matches that of the SC-List decoder when no SPC constituent decoders of lengths greater than four are used. The longer SPC constituent decoders, denoted SPC-8+, only consider the four least-reliable bits in their inputs. This approximation only affects the performance when $L > 2$. Fig. 4 illustrates this effect by comparing the FER of different list sizes with and without SPC-8+ constituent decoders, labeled Dec-SPC-4 and Dec-SPC-4+, respectively. Since for $L = 2$, the SPC constituent decoders do not affect the error-correction performance, only one graph is shown for that size. As L increases, the FER degradation due to SPC-8+ decoders increases. The gap is < 0.1 dB for $L = 8$, but grows to ≈ 0.25 dB when L is increased to 32. These results were obtained with a CRC of length 32 bits. The figure also shows the FER of the (2048, 1723) LDPC code [25] after 10 iterations of offset min-sum decoding for comparison.

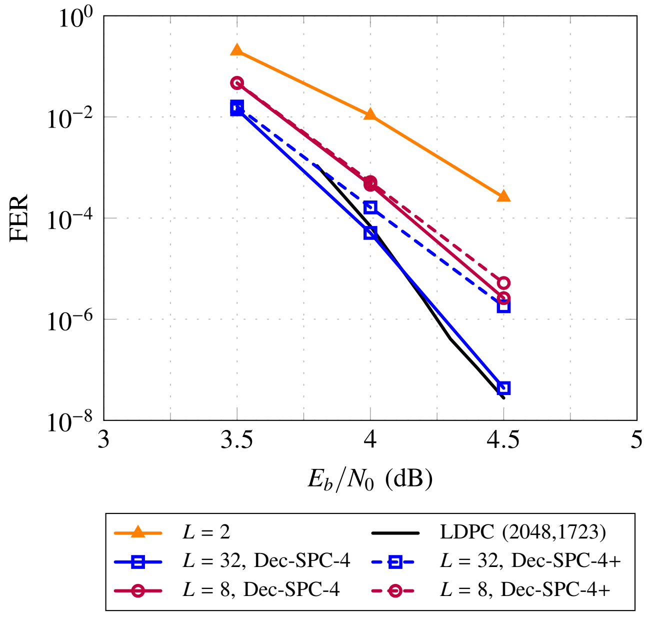


Fig. 4: FER of the polar-CRC (2048, 1723) code using the proposed decoder with different list sizes, with and without SPC decoders.

While using SPC-8+ constituent decoders degrade the error-correction performance for larger L values, they decrease decoding latency as will be shown in the following section. Therefore, the decision regarding whether to employ them or not depends on the target FER and list size.

D. Latency and Throughput

To determine the latency improvement due to the new algorithm and implementation, we compare in Table I two unrolled decoders with an LLR-based SC-list decoder implemented according to the method described in [6]. The first unrolled decoder does not implement any specialized constituent decoders and is labeled “unrolled SC-list”. While the other, labeled “unrolled Dec-SPC-4,” implements all the constituent decoders described in this work, limiting the length of the SPC ones to four. We observe that unrolling the SC-list decoder decreases decoding latency by more than 50%. Furthermore, using the rate-0, rate-1, repetition, and SPC-4 constituent decoders decreases the latency to between 63% ($L = 2$) and 18.9% ($L = 32$) that of the unrolled SC-list decoder. The speed improvement gained by using the proposed decoding algorithm and implementation compared to SC-list decoding varies between 18.4 and 11.9 times at list sizes of 2 and 32, respectively.[1] The impact of unrolling the decoder is more evident for smaller list sizes; whereas the new constituent decoders play a more significant role for larger lists.

Table I also shows the latency for the proposed decoder when no restriction on the length of the constituent SPC decoders is present, denoted “Unrolled Dec-SPC-4+”. We note that enabling these longer constituent decoder decreases

[1]The gains over an LL-based SC-list decoder are even more significant: such a decoder has a latency of 20.5 ms for $L = 32$, leading the proposed decoder to have 47 times the speed.

TABLE I: Latency (in μs) of decoding the (2048, 1723) polar-CRC code using the proposed method with different list sizes, with and without SPC decoders compared to that of SC-list decoder. Speedups compared to SC-List are shown in brackets

Decoder	L		
	2	8	32
SC-List	558	1450	5145
Unrolled SC-list	193 (2.9×)	564 (2.6×)	2294 (2.2×)
Unrolled Dec-SPC-4	30.4 (18.4×)	97.5 (14.9×)	433 (11.9×)
Unrolled Dec-SPC-4+	26.3 (21.2×)	80.2 (18.1×)	N/A

TABLE II: Information throughput of the proposed adaptive decoder with $L_{\max} = 32$.

L	info. T/P (Mbps)		
	3.5 dB	4.0 dB	4.5 dB
8	32.8	92.1	196
32	8.6	33.0	196

latency by 14% and 18% for $L = 2$ and 8, respectively. Due to the significant loss in error-correction performance, we do not recommend using the SPC-8+ constituent decoders for $L > 8$ and therefore do not list the latency of such a decoder configuration.

The throughput of the proposed decoder decreases almost linearly with L. For $L = 32$ with a latency of 433 μs, the information throughput is 4.0 Mbps. As mentioned in Section V, throughput can be improved using adaptive decoding where a Fast-SSC decoder is used before the list decoder. The throughput results for this approach are shown for $L = 8$ and $L = 32$ in Table II. As E_b/N_0 increases, the Fast-SSC succeeds more often and the impact of the list decoder on throughput is decreased, according to (17), until it is becomes negligible as can be observed at 4.5 dB where the throughput for both $L = 8$ and 32 is the same.

VII. Comparison with LDPC Codes

A. *Comparison with the (2048, 1723) LDPC Code*

We implemented a scaled min-sum decoder for the (2048, 1723) LDPC code of [25]. To the best of our knowledge, this is the fastest software implementation of decoder for this code. We used early termination and maximum iteration count of 10. To match the error-correction performance at the same code length, an adaptive polar list-CRC decoder with a list size of 32 and a 32-bit CRC was used as shown in Fig. 4.

Table III presents the results of the speed comparison between the two decoders. It can be observed that the proposed polar decoder has lower latency and higher throughput throughout the entire E_b/N_0 range of interest. The throughput advantages widens from seven to 78 times as the channel conditions improve from 3.5 dB to 4.5 dB. The LDPC decoder has three times the latency of the polar list decoder.

B. *Comparison with the 802.11n LDPC Codes*

The fastest software LDPC decoders in literature are those of [32], which implement decoders for the 802.11n standard [26] using the same Intel Core i7-2600 as this work.

TABLE III: Information throughput and latency of the proposed adaptive decoder with $L_{\max} = 32$ compared to the (2048, 1723) LDPC decoder.

Decoder	Latency (ms)	info. T/P (Mbps)		
		3.5 dB	4.0 dB	4.5 dB
LDPC	1.6	1.1	2.0	2.5
This work	0.44	8.6	33.0	196

The standard defines three code lengths: 1944, 1296, 648; and four code rates: 1/2, 2/3, 3/4, 5/6. The work in [32] implements decoders for codes of length 1944 and all four rates using a layered offset-min-sum decoding algorithm with five iterations.

Fig. 5 shows the FER of these codes using a 10-iteration, flooding-schedule offset min-sum decoder that yields slightly better results than the five iteration layered decoder [32]. The figure also shows the FER of polar-CRC codes (with 8-bit CRC) of the same rate, but shorter: $N = 1024$ instead of 1944. As can be seen in the figure, when these codes were decoded using a list CRC decoder with $L = 2$, their FER remained within 0.1 dB of the LDPC codes. Specifically, for all codes but the one with rate 2/3, the polar-CRC codes have better FER than their LDPC counterparts down to at least FER = 2×10^{-3}. For a wireless communication system with retransmission such as 802.11, this constitutes the FER range of interest. These results show that the FER of $N = 1024$ is sufficient and that it is unnecessary to use longer codes to improve it further.

The latency and throughput of the LDPC decoders are calculated for when 524,280 information bits are transferred using multiple LDPC codewords in [32]. Table IV compares the speed of LDPC and polar-CRC decoders when decoding that many bits on an Intel Core i7-2600 with turbo frequency boost enabled. The latency comprises the total time required to decode all bits in addition to copying them from and to the decoder memory. The results show that the proposed list-CRC decoders are faster than the LDPC ones. The decoder in [32] meets the minimum throughput requirements set in [26] for codes of rate 1/2 and for two out of three cases when the rate is 3/4 (MCS indexes 2 and 3). Our proposed decoder meets the minimum throughput requirements at all code rates. This shows that in this case, a software polar list decoder obtains higher speeds and similar FER to the LDPC decoder, but with a code about half as long. Since the decoder operates on individual frames (intra-frame parallelism using SIMD), the latency per frame is significantly lower and is less than 15 μs for the tested codes as shown in the table. It should be noted that neither decoder employs early termination: the LDPC decoder in [32] always uses 5 iteration, and the list-CRC decoder does not utilize adaptive decoding. The number of LDPC and polar code frames required to transmit the 524,280 information bits at each code rate are also shown in Table IV.

VIII. Conclusion

In this work, we described an algorithm to significantly reduce the latency of polar list decoding, by an order of magnitude compared to the prior art when implemented in software.

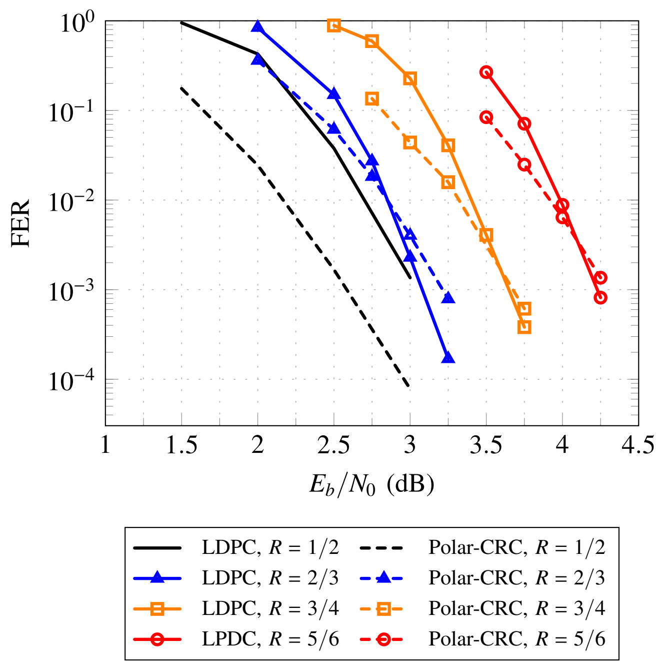


Fig. 5: Frame-error rate of the proposed decoders of length 1024 compared with those of the 802.11n standard of length 1944.

TABLE IV: Information throughput and latency of the proposed list decoder compared with the LDPC decoders of [32] when estimating 524,280 information bits.

Decoder	N	# of N-bit frames	Rate	Latency (ms)		info. T/P (Mbps)
				total	per frame	
[32]	1944	540	1/2	17.4	N/A	30.1
proposed	1024	1024	1/2	13.8	0.014	38.0
[32]	1944	405	2/3	12.7	N/A	41.0
proposed	1024	768	2/3	10.0	0.013	52.4
[32]	1944	360	3/4	11.2	N/A	46.6
proposed	1024	683	3/4	8.78	0.013	59.6
[32]	1944	324	5/6	9.3	N/A	56.4
proposed	1024	615	5/6	6.2	0.010	84.5

We also showed that polar list decoders may be suitable for software-defined radio applications as they can achieve high throughput, especially when using adaptive decoding. Furthermore, when compared with state-of-the art LDPC software decoders from wireless standards, we demonstrated that polar codes could achieve at least the same throughput and similar FER, while using significantly shorter codes. Future work will focus on implementing unrolled list decoders as application-specific integrated circuits (ASIC), which we expect to have throughput approaching 1 Gbps.

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," *IEEE Trans. Inf. Theory*, vol. 61, no. 5, pp. 2213–2226, May 2015.

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

[4] Y. S. Park, Y. Tao, S. Sun, and Z. Zhang, "A 4.68Gb/s belief propagation polar decoder with bit-splitting register file," in *Symp. on VLSI Circuits Dig. of Tech. Papers*, Jun 2014, pp. 1–2.

[5] J. Lin and Z. Yan, "An efficient list decoder architecture for polar codes," *IEEE Trans. VLSI Syst.*, vol. 23, no. 11, pp. 2508–2518, Nov 2015.

[6] A. Balatsoukas-Stimming, M. Parizi, and A. Burg, "LLR-based successive cancellation list decoding of polar codes," *IEEE Trans. Signal Process.*, vol. 63, no. 19, pp. 5165–5179, Oct 2015.

[7] Y. Fan, J. Chen, C. Xia, C.-y. Tsui, J. Jin, H. Shen, and B. Li, "Low-latency list decoding of polar codes with double thresholding," *arXiv preprint arXiv:1504.03437*, 2015.

[8] 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.

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

[10] B. Le Gal, C. Leroux, and C. Jego, "Multi-Gb/s software decoding of polar codes," *IEEE Trans. Signal Process.*, vol. 63, no. 2, pp. 349–359, Jan 2015.

[11] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, "Low-latency software polar decoders," *CoRR*, Apr 2015. [Online]. Available: http://arxiv.org/abs/1504.00353

[12] G. Sarkis, I. Tal, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Flexible and low-complexity encoding and decoding of systematic polar codes," *IEEE Trans. Commun.*, Jul 2015, submitted. [Online]. Available: http://arxiv.org/abs/1507.03614

[13] H. Yoo and I.-C. Park, "Partially parallel encoder architecture for long polar codes," *IEEE Trans. Circuits Syst. II*, vol. 62, no. 3, pp. 306–310, March 2015.

[14] P. Jouguet and S. Kunz-Jacques, "High performance error correction for quantum key distribution using polar codes," *Quantum Inf. & Computation*, vol. 14, no. 3-4, pp. 329–338, 2014.

[15] Nutaq, "Nutaq PicoSDR," accessed: Sept. 1, 2015. [Online]. Available: http://nutaq.com/en/products/picosdr

[16] Ettus Research, "USRP Networked Series," accessed: Sept. 1, 2015. [Online]. Available: http://www.ettus.com/product/category/USRP-Networked-Series

[17] ——, "USRP Bus Series," accessed: Sept. 1, 2015. [Online]. Available: http://www.ettus.com/product/category/USRP-Bus-Series

[18] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. Gross, "Increasing the speed of polar list decoders," in *IEEE Workshop on Signal Process. Systems (SiPS)*, Oct 2014, pp. 1–6.

[19] G. Sarkis, P. Giard, C. Thibeault, and W. Gross, "Autogenerating software polar decoders," in *IEEE Glob. Conf. on Signal and Inf. Process. (GlobalSIP)*, Dec 2014, pp. 6–10.

[20] C. Xiong, J. Lin, and Z. Yan, "Symbol-decision successive cancellation list decoder for polar codes," *arXiv preprint arXiv:1501.04705*, 2015.

[21] B. Yuan and K. Parhi, "Low-latency successive-cancellation list decoders for polar codes with multibit decision," *IEEE Trans. VLSI Syst.*, vol. 23, no. 10, pp. 2268–2280, Oct 2015.

[22] B. Yuan and K. K. Parhi, "Reduced-latency LLR-based SC list decoder for polar codes," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 107–110.

[23] I. Dumer, "Soft-decision decoding of Reed-Muller codes: a simplified algorithm," *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 954–963, March 2006.

[24] I. Dumer and K. Shabunov, "Soft-decision decoding of reed-muller codes: recursive lists," *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 1260–1266, March 2006.

[25] "IEEE standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," *IEEE Std 802.3an-2006 (Amendment to IEEE Std 802.3-2005)*, pp. 1–167, 2006.

[26] "IEEE standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, pp. 1–2793, Mar 2012.

[27] 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, Dec 2012.

[28] A. Balatsoukas-Stimming, A. Raymond, W. Gross, and A. Burg, "Hardware architecture for list successive cancellation decoding of polar codes," *IEEE Trans. Circuits Syst. II*, vol. 61, no. 8, pp. 609–613, Aug 2014.

[29] 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.

[30] T. Furtak, J. N. Amaral, and R. Niewiadomski, "Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms," in *Proc. ACM Symp. on Parallel Algorithms and Archit.* ACM, 2007, pp. 348–357.

[31] J. Demel, S. Koslowski, and F. Jondral, "A LTE receiver framework using GNU Radio," *J. Signal Process. Syst.*, vol. 78, no. 3, pp. 313–320, 2015.

[32] X. Han, K. Niu, and Z. He, "Implementation of IEEE 802.11n LDPC codes based on general purpose processors," in *IEEE Int. Conf. Commun. Technol. (ICCT), on*, Nov 2013, pp. 218–222.

Gabi Sarkis Gabi Sarkis received the B.Sc. degree in electrical engineering (summa cum laude) from Purdue University, West Lafayette, Indiana, United States, in 2006 and the M.Eng. degree from McGill University, Montreal, Quebec, Canada, in 2009. He is currently pursuing a Ph.D. degree at McGill University. His research interests are in the design of efficient algorithms and implementations for decoding error-correcting codes, in particular non-binary LDPC and polar codes.

Pascal Giard received the B.Eng. and M.Eng. degree in electrical engineering from École de technologie superieure (ÉTS), Montreal, QC, Canada, in 2006 and 2009. From 2009 to 2010, he worked as a research professional in the NSERC-Ultra Electronics Chair on 'Wireless Emergency and Tactical Communication' at ÉTS. He is currently working toward the Ph.D. degree at McGill University. His research interests are in the design and implementation of signal processing systems with a focus on modern error-correcting codes.

Alexander Vardy (S'88–M'91–SM'94–F'99) was born in Moscow, U.S.S.R., in 1963. He earned his B.Sc. (summa cum laude) from the Technion, Israel, in 1985, and Ph.D. from the Tel-Aviv University, Israel, in 1991. During 1985–1990 he was with the Israeli Air Force, where he worked on electronic counter measures systems and algorithms. During the years 1992–1993 he was a Visiting Scientist at the IBM Almaden Research Center, in San Jose, CA. From 1993 to 1998, he was with the University of Illinois at Urbana-Champaign, first as an Assistant Professor then as an Associate Professor. Since 1998, he has been with the University of California San Diego (UCSD), where he is the Jack Keil Wolf Endowed Chair Professor in the Department of Electrical and Computer Engineering, with joint appointments in the Department of Computer Science and the Department of Mathematics. While on sabbatical from UCSD, he has held long-term visiting appointments with CNRS, France, the EPFL, Switzerland, and the Technion, Israel.

His research interests include error-correcting codes, algebraic and iterative decoding algorithms, lattices and sphere packings, coding for digital media, cryptography and computational complexity theory, and fun math problems.

He received an IBM Invention Achievement Award in 1993, and NSF Research Initiation and CAREER awards in 1994 and 1995. In 1996, he was appointed Fellow in the Center for Advanced Study at the University of Illinois, and received the Xerox Award for faculty research. In the same year, he became a Fellow of the Packard Foundation. He received the IEEE Information Theory Society Paper Award (jointly with Ralf Koetter) for the year 2004. In 2005, he received the Fulbright Senior Scholar Fellowship, and the Best Paper Award at the IEEE Symposium on Foundations of Computer Science (FOCS). During 1995–1998, he was an Associate Editor for Coding Theory and during 1998–2001, he was the Editor-in-Chief of the IEEE TRANSACTIONS ON INFORMATION THEORY. From 2003 to 2009, he was an Editor for the SIAM Journal on Discrete Mathematics. He has been a member of the Board of Governors of the IEEE Information Theory Society during 1998–2006, and again starting in 2011.

Claude Thibeault received his Ph.D. from Ecole Polytechnique de Montreal, Canada. He is now with the Electrical Engineering department of Ecole de technologie superieure, where he serves as full professor. His research interests include design and verification methodologies targeting ASICs and FPGAs, defect and fault tolerance, as well as current-based IC test and diagnosis. He holds 11 US patents and has published more than 120 journal and conference papers, which were cited more than 550 times. He co-authored the best paper award at DVCON05, verification category. He has been member of different conference program committee, including the VLSI Test Symposium, for which he was program chair in 2010-2012, and general chair in 2014.

Warren J. Gross received the B.A.Sc. degree in electrical engineering from the University of Waterloo, Waterloo, Ontario, Canada, in 1996, and the M.A.Sc. and Ph.D. degrees from the University of Toronto, Toronto, Ontario, Canada, in 1999 and 2003, respectively. Currently, he is an Associate Professor with the Department of Electrical and Computer Engineering, McGill University, Montral, Qubec, Canada. His research interests are in the design and implementation of signal processing systems and custom computer architectures. Dr. Gross is currently Chair of the IEEE Signal Processing Society Technical Committee on Design and Implementation of Signal Processing Systems. He has served as Technical Program Co-Chair of the IEEE Workshop on Signal Processing Systems (SiPS 2012) and as Chair of the IEEE ICC 2012 Workshop on Emerging Data Storage Technologies. Dr. Gross served as Associate Editor for the IEEE Transactions on Signal Processing. He has served on the Program Committees of the IEEE Workshop on Signal Processing Systems, the IEEE Symposium on Field-Programmable Custom Computing Machines, the International Conference on Field-Programmable Logic and Applications and as the General Chair of the 6th Annual Analog Decoding Workshop. Dr. Gross is a Senior Member of the IEEE and a licensed Professional Engineer in the Province of Ontario.