Unrolled Polar Decoders, Part II: Fast List Decoders

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. This is the second in a two-part series of papers on unrolled polar decoders. Part I focuses on hardware SC polar decoders. In this paper (Part II), 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], exploit the channel polarization phenomenon to 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 the finite-length performance of successive-cancellation decoding of polar codes, list decoding was proposed by Tal and Vardy [2]. The improvement due to list decoding is significant for low signal-to-noise ratios (SNR), but decreases as the channel conditions improve. Concatenating the polar code with a cyclic redundancy check (CRC) as an outer code, and using the CRC to select from among the candidates provided by the list decoder provides significant improvements to the error-correction performance that persist even as the channel improves [3]. Polar codes with CRC under list decoding (“list-CRC”) have been shown to be able to exceed the error-correcting performance of LDPC codes of similar length [3].

However, list-CRC decoding comes with a downside. The sequential “bit-by-bit” decoding order of the SC algorithm imposes a constraint on the speed of practical implementations, which becomes more apparent 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: hardware implementations to date have been limited to under 300 Mbps, one order of magnitude lower than the fastest SC decoder hardware [4], which achieves information throughput of 1.0 Gbps at 100 MHz in FPGA. The fastest belief propagation polar decoder in literature achieves 2.34 Gbps at 300 MHz in 65nm CMOS [5]. Reported hardware list decoder implementations achieve coded throughputs of 285 Mbps at 714 MHz for $N = 1024$ and $L = 2$ [6], and 220 Mbps at 558 MHz for $N = 1024$ and $L = 2$ [7].

The key to increasing the speed of SC decoders is to break the serial constraint imposed by successive cancellation. Almadar-Yazdi and Kschischang recognized that certain decoding steps in SC decoding were redundant for certain groups of bits that could instead be estimated simultaneously, given appropriate implementations [8]. In their 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 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 [4, 9]. The Fast-SSC algorithm in [4] considers a variety of different constituent codes, such as single-parity-check and repetition codes, decoding them with parallel hardware, estimating several bits per clock cycle.

Polar codes were also shown to be well suited for fast software SC decoder implementations [10], [11], and [12], using single-instruction multiple-data (SIMD) instructions to parallelize the decoding process. 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 $\mu$s. An alternate approach using intra-frame parallelism targeting low-latency implementations was used by [12] to reach an information throughput up to 1.3 Gbps and 1 $\mu$s latency.

This is the second in a two-part series of papers on unrolled polar decoders. In Part I, we describe how unrolling the main decoding loop leads to very high speed (hundreds of Gbps)
pipelined hardware architectures for SC decoders. In a similar fashion, software unrolling exposes parallelism amongst the core decoding functions and is the basis for the implementation in this paper (Part II), where we show how to increase the speed of polar list software decoders by an order of magnitude. More precisely, we show how to apply the concepts from the Fast-SSC algorithm to list decoding and then obtain further speedups by unrolling the main decoding loop.

This work expands and improves on previous conference publications [11] and [13]. The algorithm in this paper has been reformulated in terms of log-likelihood ratios (LLRs), which yields speed improvements over the preliminary work in [11]. Furthermore, the conference paper implemented a list decoding algorithm based on SSC decoding (list-SSC), while this work considers elements from the Fast-SSC algorithm (list-Fast-SSC), yielding further performance improvements. Finally, unrolling [13] 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 Section III we review the construction of polar codes and the list-CRC and the Fast-SSC decoding algorithms. We 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 II-V. The speed and error-correction performance of the proposed decoder are studied in Section II-VII and compared to those of LDPC codes of the 802.3an [14] and 802.11n [15] standards in Section II-VII.

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 \( \hat{u} \) using the channel output \( y_0^{N-1} \) and the previously estimated bits \( \hat{u}_0^{i-1} \) according to

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

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 \( N \).

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 \( \alpha \), and hard bit estimates \( \beta \). When a node corresponding to a constituent code of length

\[ N \]

receives the reliability values from its parent, represented using LLRs, the output to its left child is calculated according to the \( F \) function:

\[
\alpha_l[i] = F(\alpha_r[i], \alpha_r[i + N_r/2]) = \text{sgn}(\alpha_r[i]) \text{sgn}(\alpha_r[i + N_r/2]) \min(|\alpha_r[i]|, |\alpha_r[i + N_r/2]|).
\]

Once the output of the left child \( \beta_l \) is available the message to right one is calculated using the \( G \) function

\[
\alpha_r[i] = G(\alpha_l[i], \alpha_l[i + N_r/2], \beta_l[i]) = \alpha_l[i + N_r/2] - (2\beta_l[i] - 1)\alpha_l[i].
\]

Finally, when \( \beta_r \) is known, the node’s output is computed as

\[
\beta_r[i] = \begin{cases} 
\beta_l[i] \oplus \beta_r[i] & \text{when } i < N_r/2; \\
\beta_l[i - N_r/2] & \text{otherwise};
\end{cases}
\]

where \( \oplus \) is an XOR operation and we refer to the operation as the Combine operation.

The output \( \beta_r \) of a frozen node is always zero, and is calculated using threshold detection for an information node:

\[
\beta_r = h(\alpha_r) = \begin{cases} 
0 & \text{when } \alpha_r \geq 0; \\
1 & \text{otherwise}.
\end{cases}
\]

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 [3] that using a CRC as metric 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 [16].
Initially, polar list decoders used likelihood [3] and log-likelihood values [17] to represent reliabilities. Later, log-likelihood ratios (LLRs) were used in [7] 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 [7] stores a reliability metric \( PM_i \) for each path \( i \) that is updated for every estimated bit \( i \) according to:

\[
PM_i = \begin{cases} 
PM_{i-1} & \text{if } \hat{u}_i = h(\alpha_i), \\
PM_{i-1} - |\alpha_i| & \text{otherwise.}
\end{cases}
\]

(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 \( N_r \); while that of rate-1 can be found by applying threshold detection element-wise on \( \alpha \), so that

\[
\beta_{i}[i] = h(\alpha_{i}[i]) = \begin{cases} 
0 & \text{when } \alpha_{i}[i] \geq 0; \\
1 & \text{otherwise.}
\end{cases}
\]

(7)

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 [4], [9].

The ML-decision for a repetition code is

\[
\beta_{i}[i] = \begin{cases} 
0 & \text{when } \sum_j \alpha_{i}[j] \geq 0; \\
1 & \text{otherwise.}
\end{cases}
\]

(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

\[
j = \arg \min_j |\alpha_{i}[j]|.
\]

The final output of the SPC decoder is

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

(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 [12] utilize additional specialized constituent decoders that are not used in this work. In addition, the operations mentioned in this section and implemented in [12] 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 [11] 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 [13], 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 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 \( N_r \) 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 \( F \), \( G \), and \( \text{Combine} \) 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 [11], where path reliability metrics in a priority queue so that the worst candidate can be quickly found and replaced with a new path when appropriate.

Rate-0 nodes do not generate new paths; however, like their length-1 counterparts in SC-list decoding [7], they alter path reliability values. In [7], the path metric was updated according to

\[
PM_i^t = \begin{cases} 
PM_i^{t-1} & \text{if } h(\alpha_i) = 0, \\
PM_i^{t-1} - |\alpha_i| & \text{otherwise}.
\end{cases}
\]

This can be extended to rate-0 nodes of length \( > 1 \) so that

\[
PM_i^t = PM_i^{t-1} - \sum_{i=0}^{N_t-1} h(\alpha_i[i])|\alpha_i[i]|; 
\]

where \( t \) is the leaf-node index.

In this section, we describe how each of the remaining nodes 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.

### Algorithm 3 Candidate selection process

```plaintext
for \( s \in \text{sourcePaths} \) do
  Generate candidates.
  Store reliability of all candidates except the ML one.
  Store ML decision.
end for

for \( p \in \text{candidates} \) do
  if fewer than \( L \) candidates are stored then
    Store \( p \).
  else if \( PM_i^f < \min. \text{stored candidate reliability} \) then
    Replace min. reliability candidate with \( p \).
  end if
end for
```

### A. Rate-1 Decoders

A decoder for a length \( N_t \) rate-1 constituent code can provide up to \( 2^{N_t} \) candidate codewords. This approach is impractical as it scales exponentially in \( N_t \). Furthermore, our simulations showed that not all candidates need to be tested. Instead, we use the low-complexity candidate-enumeration method of Chase decoding [18].

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

\[
PM_0' = PM_0^{t-1}, \\
PM_1' = PM_1^{t-1} - |\alpha_s[\min_1]|, \\
PM_2' = PM_2^{t-1} - |\alpha_s[\min_2]|, \\
PM_3' = PM_3^{t-1} - |\alpha_s[\min_1]| - |\alpha_s[\min_2]|;
\]

where \( PM_0' \) corresponds to the ML decision, \( PM_1' \) to the ML decision with the least-reliable bit flipped, \( PM_2' \) to the ML decision with the second least-reliable bit flipped, and \( PM_3' \) to the ML decision with the two least-reliable bits flipped.

### B. SPC Decoders

The ML decision of an SPC decoder is calculated according to (7). The associated path reliability is

\[
PM_0^s = \begin{cases} 
PM_0^{t-1} & \text{when parity } = 0, \\
PM_0^{t-1} - |\alpha_s[\min_1]| & \text{otherwise}.
\end{cases}
\]

When generating the candidate paths, simulation results, presented in Section [11] 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 ignoring 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 remaining candidates are

\[
PM_1^s = PM_1^{t-1} - q|\alpha_s[\min_1]| - |\alpha_s[\min_2]|, \\
PM_2^s = PM_2^{t-1} - q|\alpha_s[\min_1]| - |\alpha_s[\min_1]|, \\
PM_3^s = PM_3^{t-1} - q|\alpha_s[\min_2]| - |\alpha_s[\min_1]|, \\
PM_4^s = PM_4^{t-1} - |\alpha_s[\min_2]| - |\alpha_s[\min_3]|, \\
PM_5^s = PM_5^{t-1} - |\alpha_s[\min_2]| - |\alpha_s[\min_4]|,
\]
PM'_0 = PM'_{-1} - \lfloor \alpha[i][\text{min}_3] \rfloor - \lfloor \alpha[i][\text{min}_4] \rfloor,
PM'_1 = PM'_{-1} - \lfloor q(\alpha[i][\text{min}_1]) \rfloor
- \lfloor \alpha[i][\text{min}_2] \rfloor - \lfloor \alpha[i][\text{min}_3] \rfloor - \lfloor \alpha[i][\text{min}_4] \rfloor.

These 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.

As will be shown via simulation results in Section VII for SPC codes of length > 4 and L = 2, only the candidates corresponding to the ML decision, PM'_1, PM'_2, and PM'_3 need to be evaluated.

C. Repetition Decoders

A repetition decoder has two possible outputs: the all-zero and the all-one codewords whose reliabilities are

PM'_0[α] = PM'_{-1}[α] - \sum_i \lfloor \min(\alpha_v[i], 0) \rfloor,
PM'_1[α] = PM'_{-1}[α] - \sum_i \lfloor \max(\alpha_v[i], 0) \rfloor;

where PM'_0[α] and PM'_1[α] 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.

The ML decision is found according to \arg \max_{i}(PM'_i[α]), which is the same as performing \text{6}.

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 (\text{float}) to represent the binary-valued \beta, in addition to the real-valued \alpha, since it improves vectorization of the \text{g} operation as explained in Section IV-C.

A. Memory Layout for \alpha Values

Memory is organized into stages: the input to all constituent codes of length \text{N}_t is stored in stage \text{S}_{\log_2(\text{N}_t)}. Due to the sequential nature of the decoding process, only \text{N}_t values need to be stored for a stage since new 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 \text{S}_3, overwriting the input to the repetition node of the same size.

When using SIMD instructions, memory must be aligned according to 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 \text{N}_t is allocated \max(\text{N}_t, V) elements, where \text{V} is the number of \alpha values in a SIMD vector, and the total memory allocated for storing \alpha values is

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

elements; where the values in stage \text{S}_{\log_2(\text{N})} are the channel reliability information that are shared among all paths and \text{L} is the list size.

During the candidate forking process at a stage \text{S}_t, a path \text{p} is created from a source path \text{s}. The new path \text{p} shares all the information with \text{s} for stages \in [\text{S}_{\log_2(\text{N})}, \text{S}_t). This is exploited in order to minimize the number of memory copy operations by updating memory pointers when a new path is created \text{4}. For stages \in [\text{S}_0, \text{S}_t], path \text{p} gets its own memory since the values stored in these stages will differ from those calculated by other descendents of \text{s}.

B. Memory Layout for \beta Values

Memory for \beta values is also arranged into stages. However, since calculating \beta, (\text{4}) requires both \beta_1 and \beta_2, 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 \beta values is

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

As stage \text{S}_{\log_2(\text{N})} stores the output candidate codewords of the decoder, which will not be combined with other values, only \text{L}, instead of \text{2L}, memory blocks are required.

Stored \beta information is also shared by means of memory pointers. Candidates generated at a stage \text{S}_t share all information for stages \in [\text{S}_0, \text{S}_t).

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 \text{f} operation \text{2}. Extracting the sign and calculating the absolute values in \text{2} become simple bit-wise AND operations with the appropriate mask.

The \text{g} operation can be written as

\text{g}(\alpha_v[i], \alpha_v[i + N_t/2], \beta_v[i])
=\alpha_v[i + N_t/2] + \beta_v[i] \oplus \alpha_v[i].

If we use \beta \in \{+0.0, -0.0\} instead of \{0.1\}, the \text{g} operation \text{3} can be implemented as

\alpha_v[i + N_t/2] + \beta_v[i] \oplus \alpha_v[i]. \quad (11)

Replacing the multiplication (\ast) with an XOR (\oplus) operation on \text{11} is possible due to the sign-magnitude representation of IEEE-754.

Listing \text{4} shows the corresponding AVX implementations, originally presented in \text{12}, \text{13}, of the \text{f} and \text{g} functions using the SIMD intrinsic functions provided by GCC. For clarity of exposition, m256 is used instead of \text{m}256 and
Listing 4 Vectorized $f$ and $g$ functions

```
void G(alpha, alpha.cut, beta, beta.in) {
    for (unsigned int i = 0; i < Nv/2; i += 8) {
        m256 alpha = load_ps(alphain + i);
        m256 alpha = load_ps(alphain + i + Nv/2);
        m256 beta = load_ps(betain + i);
        m256 alpha = add_ps(alpha, alpha.cut);
        store_ps(alphaout + i, alpha);
    }
}
```

Listing 5 Path reliability update in Rate-0 decoders.

```
m256 ZERO = set1_ps(0.0);
m256 PMv[0] = ZERO;
for (unsigned int i = 0; i < Nv/2; i += 8) {
    PMv = add_ps(PMv, min_ps(load_ps(alphain + i), ZERO));
} PM = Sum PMv[i];
```

the _mm256_ prefix is removed from the intrinsic function names.

Rate-0 decoders set their output to the all-zero vector using store instructions. The path reliability (PM) calculation [10] is implemented as in Listing 5.

D. Rate-1 Nodes

Since $\beta \in \{+0.0, -0.0\}$ and $\alpha$ 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 [19]. 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, $v_0$ 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 $PM_0$ is calculated by accumulating the min($\alpha[i], 0.0$) using SIMD instructions. Similarly, to calculate $PM_1$, max($\alpha[i], 0.0$) 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 [16], 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

$$L(A_{max}) = L(L) + L(F);$$

where $L(L)$ and $L(F)$ 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) 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 bits/s is

$$T = \frac{k}{(1 - FER_F) L(F) + FER_L L(L)}.$$  

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 [20]. 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 [21].
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.

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. The figure shows that the performance is better at lower Eb/N0 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 [14] after 10 iterations of offset min-sum decoding for comparison.

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 [7]. 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. 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.

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  | 564 | 2294|
| Unrolled Dec-SPC-4       | 304  | 975 | 433 |
| Unrolled Dec-SPC-4+      | 263  | 802 | N/A |

Table II 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 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 I. 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 (13), 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 [14]. 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.

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

| $L$ | 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    |

B. Comparison with the 802.11n LDPC Codes

The fastest software LDPC decoders in literature are those of [21], which implement decoders for the 802.11n standard [15] using the same Intel Core i7-2600 as this work.

The standard defines three code lengths: 1944, 1296, 648; and four code rates: 1/2, 2/3, 3/4, 5/6. The work in [21] 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 [21]. 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 one with rate 2/3, the polar-CRC codes have better FER than their LDPC counterparts down to at least FER $= 2 \times 10^{-3}$.

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 [21]. 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 [21] meets the minimum throughput requirements set in [15] 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. It should be noted that neither decoder employs early termination: the LDPC decoder in [21] always uses 5 iteration, and the list-CRC decoder does not utilize adaptive decoding.

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. We also showed that polar list decoders may be suitable for...
Fig. 5: Bit-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 [21] when estimating 524,280 information bits.

| Decoder | N     | Rate | Latency (ms) | info. T/P (Mbps) |
|---------|-------|------|--------------|-----------------|
|         | [21]  | 1944 | 1/2          | 17.4            | 30.1            |
| this work | 1024 | 1/2  | 13.8         | 38.0            |
| this work | 1024 | 2/3  | 12.7         | 41.0            |
| this work | 1024 | 3/4  | 10.0         | 52.4            |
| this work | 1024 | 3/4  | 11.2         | 46.6            |
| this work | 1024 | 5/6  | 8.78         | 59.6            |
| this work | 1024 | 5/6  | 9.3          | 56.4            |

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,” in IEEE Int. Symp. on Inf. Theory (ISIT), Jul 2011, pp. 1–5.
[3] ——, “List decoding of polar codes,” IEEE Trans. Inf. Theory, vol. PP, no. 99, pp. 1–1, 2015.
[4] 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.
[5] Y. S. Park, Y. Tao, S. Sun, and Z. Zhang, “A 4.68Gbps belief propagation polar decoder with bit-splitting register file,” in Symp. on VLSI Circuits Dig. of Tech. Papers, Jun 2014, pp. 1–2.
[6] J. Lin and Z. Yan, “Efficient list decoder architecture for polar codes,” in IEEE Int. Symp. Circuits and Systems (ISCAS), Jun 2014, pp. 1022–1025.
[7] A. Balatsoukas-Stimming, M. Bastani Parizi, and A. Burg, “LLR-based successive cancellation list decoding of polar codes,” in IEEE Int. Conf. on Acoust., Speech, and Signal Process. (ICASSP), May 2014, pp. 3903–3907.
[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] 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.
[12] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, “Low-latency software polar decoders,” IEEE Trans. Signal Process., Apr 2015, submitted. [Online]. Available: [http://arxiv.org/abs/1504.00353]
[13] 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.
[14] “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.
[15] “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.
[16] 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.
[17] 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.
[18] 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.
[19] 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 Architect., ACM, 2007, pp. 348–357.
[20] 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.
[21] 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.