Fast Polar Decoders: Algorithm and Implementation

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 provably achieve the symmetric capacity of a memoryless channel while having an explicit construction. The adoption of polar codes however, has been hampered by the low throughput of their decoding algorithm. This work aims to increase the throughput of polar decoding hardware by an order of magnitude relative to successive-cancellation decoders and is more than 8 times faster than the current fastest polar decoder. We present an algorithm, architecture, and FPGA implementation of a flexible, gigabit-per-second polar decoder.

Index Terms—polar codes, successive-cancellation decoding, storage systems.

I. INTRODUCTION

POLAR codes [1] are the first error-correcting codes with an explicit construction to provably achieve the symmetric capacity of memoryless channels. They have two properties that are of interest to data storage systems: a very low error-floor due to their large stopping distance [2], and low-complexity implementations [3]. However, polar codes have two drawbacks: their performance at short to moderate lengths is inferior to that of other codes, such as low-density parity-check (LDPC) codes; and their low-complexity decoding algorithm, successive-cancellation (SC), is serial in nature, leading to low decoding throughput [3].

Multiple methods exist to improve the error-correction performance of polar codes. Using list, and list-CRC decoding [4] improves performance significantly. Alternatively, one can increase the length of the polar code. Using a code length corresponding to the block length of current hard drives [5], as we show in this work, results in a polar decoder with lower complexity than an LDPC decoder with similar error-correction performance and the same rate. Specifically, a (32768, 27568) polar code has slightly worse error-correction performance than the (2048, 1723) LDPC code of the 10GBASE-T (802.3an) standard in the low signal-to-noise ratio (SNR) region but better performance at frame error rates (FER) lower than 7×10^{-7}, with the high SNR region being more important for storage systems. In addition, polar codes can be made to perform better than the LDPC code starting at FER of 2×10^{-4} as shown in Section II.

Among the many throughput-improving methods proposed in literature, simplified successive-cancellation (SSC) [6] and simplified successive-cancellation with maximum-likelihood nodes (ML-SSC) [7] offer the largest improvement over SC decoding. This throughput increase is achieved by exploiting the recursive nature of polar codes—where every polar code of length N is formed from two constituent polar codes of length $N/2$—and decoding the constituent codes directly, without recursion, when possible. SSC decodes constituent codes of rates 0 and 1 directly and ML-SSC additionally enables the direct decoding of smaller constituent codes.

In this work, we focus on improving the throughput of polar decoders. By building on the ideas used for SSC and ML-SSC decoding, namely decoding constituent codes without recursion, and recognizing further classes of constituent codes that can be directly decoded, we present a polar decoder that, for a (32768, 29492) code, is 40 times faster than the best SC decoder [3] when implemented on the same field-programmable gate-array (FPGA). For a (16384, 14746) code, our decoder is more than 8 times faster than the state of the art polar decoder in litterature [8], again when implemented on the same FPGA. Additionally, the proposed decoder is flexible and can decode any polar code of a given length.

We start this paper by reviewing polar codes, their construction, and the successive-cancellation decoding algorithm in Section II. The SSC and ML-SSC decoding algorithms are reviewed in Section III. We present our improved decoding algorithm in Section IV, including new constituent code decoders. The decoder architecture is discussed in detail in sections V, VI, and VII. Implementation results, showing that the proposed decoder has lower complexity on an FPGA than the 10GBASE-T LDPC decoder with the same rate and comparable error-correction performance, are presented in Section VIII.

We focus on two codes: a (32768, 29492) that has a rate of 0.9 making it suitable for storage systems; and a (32768, 27568) that is comparable to the popular 10GBASE-T LDPC code in error-correction performance and has the same rate, which enables the implementation complexity comparison in Section VIII.

II. POLAR CODES

A. Construction of Polar Codes

By exploiting channel polarization, polar codes approach the symmetric capacity of a channel as the code length, N, increases. The polarizing construction when $N = 2$ is shown in Fig. 1(a), where the probability of correctly estimating bit u_0 decreases; while that of bit u_1 increases compared to when the bits are transmitted without any transformation over the
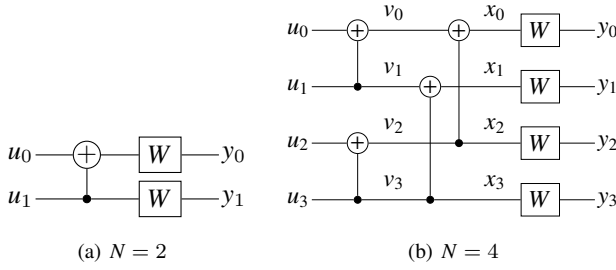
Fig. 1. Construction of polar codes of lengths 2 and 4

channel W. Channels can be combined recursively to create longer codes, Fig. 1(b) shows the case of $N = 4$. As $N \to \infty$, the probability of successfully estimating each bit approaches 1 (perfectly reliable) or 0.5 (completely unreliable), and the proportion of reliable bits approaches the symmetric capacity of W [1].

To create an (N, k) polar code, N copies of the channel W are transformed using the polarizing transform and the k most reliable bits, called the information bits, are used to send information bits; while the $N - k$ least reliable bits, called the frozen bits, are set to 0. Determining the locations of the information and frozen bits depends on the type and conditions of W and is investigated in detail in [9]. Therefore, a polar code is constructed for a given channel and channel condition. A polar code of length N can be represented using a generator matrix, $G_N = F_N = F_2^{\otimes \log_2 N}$, where $F_2 = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$ and $\otimes$ is the Kronecker power. The frozen bits are indicated by setting their values to 0 in the source vector $\boldsymbol{u}$.

Polar codes can be encoded systematically to improve bit error-rate (BER) [10]. Furthermore, systematic polar codes are a natural fit for the SSC and ML-SSC algorithms [7].

In [1], bit-reversed indexing is used, which changes the generator matrix by multiplying it with a bit-reversal operator B, so that $G = BF$. In this work, we use natural indexing to review and introduce algorithms for reasons of clarity. However, it was shown in [3] that bit-reversed indexing significantly reduced data-routing complexity in a hardware implementation; therefore, we used it to implement our decoder architecture. In Section III-D, we review how to combine systematic encoding and bit-reversal without using any interleavers.

B. Successive-Cancellation Decoding

Polar codes achieve the channel capacity asymptotically in code length when decoded using the successive-cancellation (SC) decoding algorithm, which sequentially estimates the bits $\hat{u}_i$, where $0 \le i < N$, using the channel output $\boldsymbol{y}$ and the previously estimated bits, $\hat{u}_0$ to $\hat{u}_{i-1}$, denoted $\hat{\boldsymbol{u}}_0^{i-1}$, according to:

$$\hat{u}_i = \begin{cases} 0, & \text{if } \lambda_{u_i} \ge 0; \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

Where λ_{u_i} is the log-likelihood ratio (LLR) defined as $\Pr[\boldsymbol{y}, \hat{\boldsymbol{u}}_0^{i-1} | \hat{u}_i = 0] / \Pr[\boldsymbol{y}, \hat{\boldsymbol{u}}_0^{i-1} | \hat{u}_i = 1]$ and can be calculated recursively using the min-sum (MS) approximation according to [3]

$$\lambda_{u_0} = f(\lambda_{v_0}, \lambda_{v_1}) = \text{sign}(\lambda_{v_0})\text{sign}(\lambda_{v_1})\min(|\lambda_{v_0}|, |\lambda_{v_1}|); \quad (2)$$
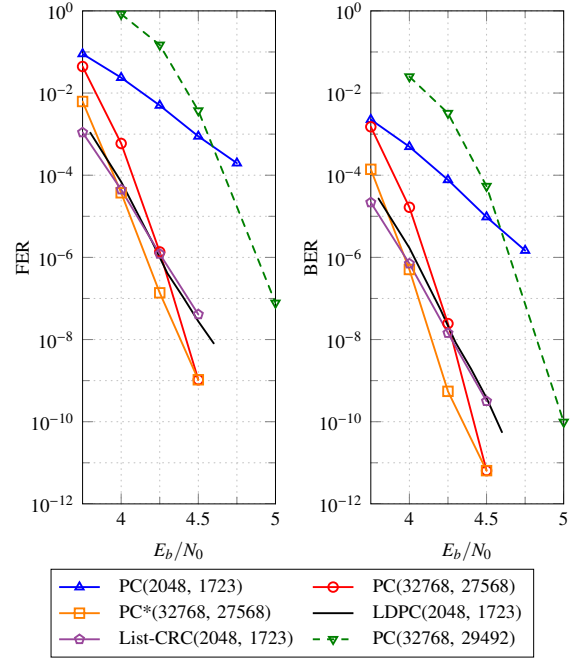

Fig. 2. Error-correction performance of polar codes compared with that of an LDPC code with the same rate. In addition to the performance of a rate 0.9 polar code.

and

$$\lambda_{u_1} = g(\lambda_{v_0}, \lambda_{v_1}, \hat{u}_0) = \begin{cases} \lambda_{v_0} + \lambda_{v_1} & \text{when } \hat{u}_0 = 0, \\ -\lambda_{v_0} + \lambda_{v_1} & \text{when } \hat{u}_0 = 1. \end{cases} \quad (3)$$

C. Performance of SC Decoding

Fig. 2 shows the error-correction performance of the (2048, 1723) 10GBASE-T LDPC code compared to that of polar codes of the same rate. These results were obtained for the binary-input additive white Gaussian noise (AWGN) channel with random codewords and binary phase-shift keying (BPSK) modulation. The first observation to be made is that the performance of the (2048, 1723) polar code is significantly worse than that of the LDPC code. The polar code of length 32768, labeled PC(32768, 27568), was constructed to be optimal for $E_b/N_0 = 4.5$ dB and performs worse than the LDPC code until the $E_b/N_0 = 4.25$ dB. Past that point, it outperforms the LDPC code with a growing gap. The last polar error-rate curve, labeled PC*(32768, 27568), combines the results of two (32768, 27568) polar codes. One is constructed for 4.25 dB and used up to that point, and the other is constructed for 4.5 dB. Due to the regular structure of polar codes, it is simple to build a decoder that can decode any polar code of a given length. Therefore, it is simpler to change polar codes in a system than it is to change LDPC codes.

From these results, it can be concluded that a (32768, 27568) polar code constructed for 4.5 dB or a higher E_b/N_0 is required to outperform the (2048, 1723) LDPC one in the low error-rate region, and a combination of different polar codes can be used to outperform the LDPC code even in high error-rate regions. Even though the polar code has a longer length, its decoder still has a lower implementation complexity than the LDPC decoder as will be shown in Section VIII.

Decoding the (2048, 1723) code using the list-CRC algo-
rithm [4], with a list size of 32 and a 32-bit CRC, reduces
the gap with the LDPC code to the point where the two codes
have similar performance as shown in Fig. 2. However, in spite
of this improvement, we do not discuss list-CRC decoding
in this work as it cannot directly accommodate the proposed
throughput-improving techniques, which are designed to pro-
vide a single estimate instead of a list of potential candidates.
Further research is required to adapt some of these techniques
to list decoding.

The throughput of SC decoding is limited by its serial na-
ture: the fastest implementation currently is an ASIC decoder
for a (1024, 512) polar code with an information throughput
of 48.75 Mbps when running at 150 MHz [11]; while the
fastest decoder for a code of length 32768 is FPGA-based
and has a throughput of 26 Mbps for the (32768, 27568) code
[3]. This low throughput renders SC decoders impractical for
most systems; however, it can be improved significantly by
using the SSC or the ML-SSC decoding algorithms.

III. SSC AND ML-SSC DECODING

A. Tree Structure of an SC Decoder

A polar code of length N is the concatenation of two polar
codes of length $N/2$. Since this construction is recursive,
as mentioned in Section II-A, a binary tree is a natural
representation for a polar code where each node corresponds
to a constituent code. The tree representation is presented in
detail in [6] and [7]. Fig. 3(a) shows the tree representation
for an (8, 3) polar code where the white and black leaves
correspond to frozen and information bits, respectively.

A node v, corresponding to a constituent code of length N_v,
receives a real-valued message vector, $\boldsymbol{\alpha}_v$, containing the soft-
valued input to the constituent polar decoder, from its parent
node. It calculates the soft-valued input to its left child, $\boldsymbol{\alpha}_l$
using (2). Once the constituent codeword estimate, $\boldsymbol{\beta}_l$, from
the left child is ready, it is used to calculate the input to the
right, $\boldsymbol{\alpha}_r$, according to (3). Finally, $\boldsymbol{\beta}_v$ is calculated from $\boldsymbol{\beta}_l$
and $\boldsymbol{\beta}_r$ as

$$\boldsymbol{\beta}_v[i] = \begin{cases} \boldsymbol{\beta}_l[i] \oplus \boldsymbol{\beta}_r[i], & \text{when } i < N_v/2; \\ \boldsymbol{\beta}_r[i - N_v/2], & \text{otherwise.} \end{cases} \quad (4)$$

For leaf-nodes, $\boldsymbol{\beta}_v$ is 0 if the node is frozen. Otherwise, it is
calculated using threshold detection, defined for an LLR-based
decoder as:

$$\boldsymbol{\beta}_v = \begin{cases} 0, & \text{when } \boldsymbol{\alpha}_v \geq 0; \\ 1, & \text{otherwise.} \end{cases} \quad (5)$$

The input to the root node is the LLR values calculated from
the channel output, and its output is the estimated systematic
codeword.

B. SSC and ML-SSC Decoder Trees

In [6], it was observed that a tree with only frozen leaf-
nodes rooted in a node $\mathcal{N}^0$, does not need to be traversed
as its output will always be a zero-vector. Similarly, it was
shown that the output of a tree with only information leaf-
nodes rooted in $\mathcal{N}^1$ can be obtained directly by performing
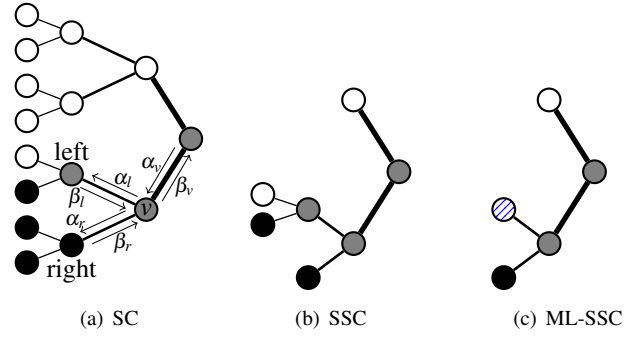threshold detection (5) on the soft-information vector $\boldsymbol{\alpha}_v$,

Fig. 3. Decoder trees corresponding to the SC, SSC, and ML-SSC decoding
algorithms

without any additional calculations. Therefore, the decoder
tree can be pruned reducing the number of node visitations and
latency. The remaining nodes, denoted $\mathcal{N}^R$ as they correspond
to codes of rate $0 < R < 1$, perform their calculations as in
the SC decoder. The pruned tree for an SSC decoder is shown
in Fig. 3(b) and requires nine time steps compared to the 14
steps required to traverse the SC tree in Fig. 3(a).

ML-SSC further prunes the decoder tree by using
exhaustive-search maximum-likelihood (ML) decoding to de-
code any constituent code, $\mathcal{C}$, while meeting resource con-
straints [7]. The (8, 3) polar decoder utilizing these $\mathcal{N}^{ML}$
nodes, and whose tree is shown in Fig. 3(c), where $\mathcal{N}^{ML}$ is
indicated with a striped pattern and is constrained to $N_v = 2$,
requires 7 time steps to estimate a codeword.

C. Performance

In [7], it was shown that under resource constraints the
information throughput of SSC and ML-SSC decoding in-
creases faster than linearly as the code rate increases, and
approximately logarithmically as the code length increases.
For example, it was estimated that for a rate 0.9 polar code
of length 32768, which is constructed for $E_b/N_0 = 3.47$ dB,
the information throughput of a decoder running at 100 MHz
using SC decoding is ~ 45 Mbit/s and increases by 20 times
to 910 Mbit/s when using ML-SSC decoding. The throughput
of SSC and ML-SSC is affected by the code construction
parameters as they affect the location of frozen bits, which in
turn affects the tree structure of the decoder and the number of
nodes that can be directly decoded. For example, constructing
the rate 0.9, length 32768 polar code for an E_b/N_0 of 5.0 dB
instead of 3.47 dB, reduces the information throughput of the
decoder to 520 Mbit/s assuming the same clock frequency of
100 MHz. While this is a significant reduction, the decoder
remains 11 times faster than an SC decoder.

It was noted in [7] that the error-correction performance of
polar codes is not tangibly altered by the use of the SSC or
ML-SSC decoding algorithms.

D. Systematic Encoding and Bit-Reversal

In [10], it was stated that systematic encoding and bit-
reversed indexing can be combined. In this section, we review
how the information bits can be presented at the output of
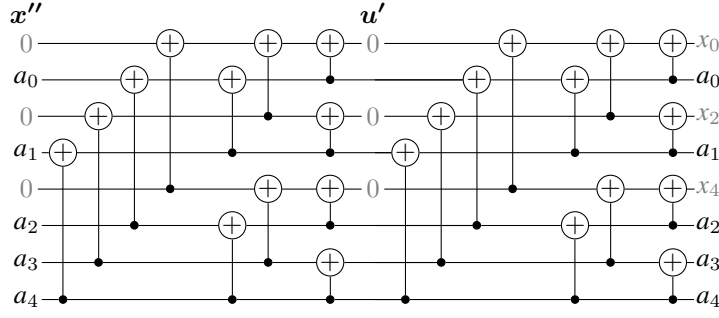the decoder in the order in which they were presented by the

Fig. 4. Systematic encoding with bit-reversal.

source, without the use of interleavers. This is of importance to the SSC decoding algorithm as it presents its output in parallel and would otherwise require an N bit parallel interleaver of significant complexity. The problem is compounded in a resource-constrained, semi-parallel SSC decoder that stores its output one word at a time in memory: since two consecutive information bits might not be in the same memory word, memory words will be visited multiple times, significantly increasing decoding latency.

To illustrate the encoding method, Fig. 4 shows the encoding process for an (8, 5) polar code with bit-reversal. (x_0'', x_2'', x_4'') are frozen and set to 0 according to the bit-reversed indices of the least reliable bits; and $(x_1'', x_3'', x_5'', x_6'', x_7'')$ are set to the information bits $(a_0, a_1, a_2, a_3, a_4)$. $\boldsymbol{x}''$ is encoded using G to obtain the vector $\boldsymbol{u}'$, in which the bits (u_0', u_2', u_4') are then set to zero. The resulting $\boldsymbol{u}'$ is encoded again yielding the systematic codeword $\boldsymbol{x}$, which is transmitted over the channel sequentially, i.e. x_0 then x_1 and so on. An encoder that does not use bit-reversal will function in the same manner, except that the frozen bit indices will be (0, 1, 2). An SSC decoder with $P = 2$ will output $(\hat{x}_0, \hat{x}_1, \hat{x}_2, \hat{x}_3)$ then $(\hat{x}_4, \hat{x}_5, \hat{x}_6, \hat{x}_7)$, i.e. the output of the decoder is $(\hat{x}_0, \hat{a}_0, \hat{x}_2, \hat{a}_1)$ then $(\hat{x}_4, \hat{a}_2, \hat{a}_3, \hat{a}_4)$ where the source data estimate appears in the correct order.

IV. PROPOSED ALGORITHM

In this section we explore more constituent codes that can be decoded directly and present the associated specialized decoding algorithms. We present three new corresponding node types: a single-parity-check-code node, a repetition-code node, and a special node whose left child corresponds to a repetition code and its right to a single-parity-check code. We also present node mergers that reduce decoder latency and summarize all the functions the new decoder must be able to perform. Finally, we study the effect of quantization on the error-correction performance of the proposed algorithm.

It should be noted that all the transformations and mergers presented in this work preserve the polar code, i.e. they do not alter the locations of frozen and information bits. While some throughput improvement is possible via some code modifications, the resulting polar code diverges from the optimal one constructed according to [9].

To keep the results in this section practical, we use P as a resource constraint parameter, similar to [3]. However, since new node types are introduced, the notion of a processing

element (PE) might not apply in certain cases. Therefore, we redefine P so that $2P$ is the maximum number of memory elements that can be accessed simultaneously. Since each PE has two inputs, P PEs require $2P$ input values and the two definitions for P are compatible. In addition, P is as a power of two as in [3].

A. Single-Parity-Check Nodes $\mathcal{N}^{SPC}$

In any polar code of rate $(N-1)/N$, the frozen bit is always u_0 rendering the code a single-parity check (SPC) code, which can be observed in Fig. 1(b). While the dimension of an SPC code is $N - 1$, for which exhaustive-search ML decoding is impractical; optimal ML decoding can still be performed with very low complexity [12]: the hard-decision estimate and the parity of the input are calculated; then the estimate of the least reliable bit is flipped if the parity constraint is not satisfied. The hard-decision estimate of the soft-input values is calculated using

$$HD[i] = \begin{cases} 0, & \text{when } \boldsymbol{\alpha}_v \geq 0; \\ 1, & \text{otherwise.} \end{cases}$$

The parity of the input is calculated as

$$\text{parity} = \bigoplus_{i=0}^{N_v-1} HD[i]. \tag{6}$$

The index of the least reliable input is found using

$$j = \arg\min_i |\boldsymbol{\alpha}_v[i]|.$$

Finally, the output of the node is

$$\boldsymbol{\beta}_v[i] = \begin{cases} HD[i] \oplus \text{parity}, & \text{when } i = j; \\ HD[i], & \text{otherwise.} \end{cases} \tag{7}$$

The resulting node can decode an SPC code of length $N_v > 2P$ in $N_v/(2P) + c$ steps, where $c \geq 1$ since at least one step is required to correct the least reliable estimate and others might be used for pipelining; whereas an SSC decoder requires $2\sum_{i=1}^{\log_2 N_v} \lceil 2^i/(2P) \rceil$ steps. For example, for an SPC constituent code of length 4096, $P = 256$, and $c = 4$, the specialized SPC decoder requires 12 steps, whereas the SSC decoder requires 46 steps. For constituent codes of length $\leq 2P$ the decoder can provide an output immediately, or after a constant number of time steps if pipelining is used.

Large SPC constituent codes are prevalent in high-rate polar codes and a significant reduction in latency can be achieved

TABLE I

NUMBER OF ALL NODES AND OF SPC NODES OF DIFFERENT SIZES IN THREE POLAR CODES OF LENGTH 32768 AND RATES 0.9, 0.8413, AND 0.5.

Code	All	SPC, $N_v \in$			
		(0, 8]	(8, 64]	(64, 256]	(256, 32768]
(32768, 29492)	2065	383	91	17	13
(32768, 27568)	3421	759	190	43	10
(32768, 16384)	9593	2240	274	19	1

TABLE II

NUMBER OF ALL NODES AND OF REPETITION NODES OF DIFFERENT SIZES IN THREE POLAR CODES OF LENGTH 32768 AND RATES 0.9, 0.8413, AND 0.5.

Code	All	Repetition, $N_v \in$		
		(0, 8]	(8, 16]	(16, 32768]
(32768, 29492)	3111	474	30	0
(32768, 27568)	5501	949	53	0
(32768, 16384)	10381	2290	244	0

if they are decoded quickly. Table I lists the number of SPC nodes, binned by size, in three different polar codes: (32768, 29492), (32768, 27568), and a lower-rate (32768, 16384), all constructed for an AWGN channel with a noise variance $\sigma^2 = 0.1936$. Comparing the results for the three codes, we observed that the total number of nodes decreases as the rate increases. The distribution of SPC nodes by length is also affected by code rate: the proportion of large SPC nodes decreases as the rate decreases.

A generalized version of the single-parity-check nodes, called caterpillar nodes, was presented in [13] and was shown to improve throughput of SSC by 11–14% when decoding polar codes transmitted over the binary erasure channel (BEC) without resource constraints.

B. Repetition Nodes $\mathcal{N}^{REP}$

Another type of constituent codes that can be decoded more efficiently than using tree traversal is repetition codes, in which only the last bit is not frozen. The decoding algorithm starts by summing all input values. Threshold detection is performed via sign detection, and the result is replicated and used as the constituent decoder's final output:

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

The decoding method (8) requires $N_v/(2P)$ steps to calculate the sum and $N_v/(2P)$ steps to set the output, in addition to any extra steps required for pipelining. Two other methods employing prediction can be used to decrease latency. The first sets all output bits to 0 while accumulating the inputs, and writes the output again only if the sign of the sum is negative. The average latency of this method is 75% that of (8). The second method sets half the output words to all 0 and the other half to all 1, and corrects the appropriate words when the sum is known. The resulting latency is 75% that of (8). However, since the high-rate codes of interest do not have any large repetition constituent codes, we chose to use (8) directly.

Unlike SPC constituent codes, repetition codes are more prevalent in lower-rate polar codes as shown in Table II. Moreover, for high-rate codes, SPC nodes have a more pronounced impact on latency reduction. This can be observed in tables I and II, which show that the total number of nodes in the decoder tree is significantly smaller when only SPC nodes are introduced than when only repetition nodes are introduced, indicating a smaller tree and lower latency. Yet, the impact of repetition nodes on latency is measurable; therefore, we use them in the decoder.

C. Repetition-SPC Nodes $\mathcal{N}^{REP\text{-}SPC}$

When enumerating constituent codes with $N_v \leq 8$ and $0 < k_v < 8$ for the (32768, 27568) and (32768, 29492) codes, three codes dominated the listing: the SPC code, the repetition code, and a special code whose left constituent code is a repetition code and its right an SPC one, denoted $\mathcal{N}^{REP\text{-}SPC}$. The other constituent codes accounted for 6% and 12% in the two polar codes, respectively. Since $\mathcal{N}^{REP\text{-}SPC}$ codes account for 28% and 25% of the total $\mathcal{N}^R$ nodes of length 8 in the two aforementioned codes, efficiently decoding them would have a significant impact on latency. This can be achieved by using two SPC decoders of length 4, SPC_0 and SPC_1, whose inputs are calculated assuming the output of the repetition code is 0 and 1, respectively. Simultaneously, the repetition code is decoded and its output is used to generate the $\mathcal{N}^{REP\text{-}SPC}$ output using either the output of SPC_0 or SPC_1 as appropriate.

While this code can be decoded using an exhaustive-search ML decoder, the proposed decoder has a significantly lower complexity.

D. Node Mergers

The $\mathcal{N}^{REP\text{-}SPC}$ node merges an $\mathcal{N}^{REP}$ and an $\mathcal{N}^{SPC}$ node to reduce latency. Similarly, it was mentioned in [7] that $\mathcal{N}^R$ nodes need not calculate the input to a child node if it is an $\mathcal{N}^0$ node. Instead, the input to the right child is directly calculated.

Another opportunity for a node merger arises when a node's right child directly provides β_r without tree traversal: the calculation of α_r, β_r, and β_v can all be performed in one step, halving the latency. This is also applicable for nodes where $N_v > 2P$: P values of α_r are calculated and used to calculate P values of β_r, which are then used to calculate $2P$ values of β_v until all values have been calculated.

This can be expanded further when the left node is $\mathcal{N}^0$. Since β_l is known a priori to be a zero vector, α_r can be immediately calculated once α_v is available and β_r is combined with the zero vector to obtain β_v.

In all the codes that were studied, $\mathcal{N}^R$, $\mathcal{N}^1$, and $\mathcal{N}^{SPC}$ were the only nodes to be observed as right children; and $\mathcal{N}^1$ and $\mathcal{N}^{SPC}$ are the only two that can be merged with their parent.

E. Required Decoder Functions

As a result of the many types of nodes and the different mergers, the decoder must perform many functions. Table III lists these 12 functions. For notation, 0, 1, and R are used to denote children with constituent code rates of 0, 1, and R, respectively. Having a left child of rate 0 allows the calculation

TABLE III
A LISTING OF THE DIFFERENT FUNCTIONS PERFORMED BY THE
PROPOSED DECODER.

Name	Description
F	calculate $\boldsymbol{\alpha}_l$ (2).
G	calculate $\boldsymbol{\alpha}_r$ (3).
COMBINE	combine $\boldsymbol{\beta}_l$ and $\boldsymbol{\beta}_r$ (4).
COMBINE-0R	same as COMBINE, but with $\boldsymbol{\beta}_l = 0$.
G-0R	same as G, but assuming $\boldsymbol{\beta}_l = 0$.
P-R1	calculate $\boldsymbol{\beta}_v$ using (3), (5), then (4).
P-RSPC	calculate $\boldsymbol{\beta}_v$ using (3), (7), then (4).
P-01	same as P-R1, but assuming $\boldsymbol{\beta}_l = 0$.
P-0SPC	same as P-RSPC, but assuming $\boldsymbol{\beta}_l = 0$.
ML	calculate $\boldsymbol{\beta}_v$ using exhaustive-search ML decoding.
REP	calculate $\boldsymbol{\beta}_v$ using (8).
REP-SPC	calculate $\boldsymbol{\beta}_v$ as in Section IV-C.

Fig. 5. Effect of quantization on the error-correction performance of the (32768, 27568) and (32768, 29492) codes.

TABLE IV
LATENCY OF ML-SSC DECODING OF THE (32768, 29492) CODE AND
THE EFFECT OF USING ADDITIONAL NODES TYPES ON IT.

None	SPC	REP-SPC	REP	All
5286	3360	4742	5042	2847

of $\boldsymbol{\alpha}_r$ directly from $\boldsymbol{\alpha}_v$ as explained earlier. It is important to make this distinction since the all-zero output of a rate 0 code is not stored in the decoder memory. In addition, having a right child of rate 1 allows the calculation of $\boldsymbol{\beta}_v$ directly once $\boldsymbol{\beta}_l$ is known. A P- prefix indicates that the message to the parent, $\boldsymbol{\beta}_v$, is calculated without explicitly visiting the right child node.

We note the absence of $\mathcal{N}^0$ and $\mathcal{N}^1$ node functions: the former due to directly calculating $\boldsymbol{\alpha}_r$ and the latter to directly calculating $\boldsymbol{\beta}_v$ from $\boldsymbol{\alpha}_r$.

F. Performance with Quantization

Fig. 5 shows the effect of quantization on the (32768, 27568) polar code that was constructed for $E_b/N_0 = 4.5$ dB. The quantization numbers are presented in (W, W_C, F) format, where W is total number of quantization bits for internal LLRs, W_c for channel LLRs, and F is the number of fractional bits. Since the proposed algorithm does not perform any operations that increase the number of fractional bits—only the integer ones—we use the same number of fractional bits for both internal and channel LLRs.

From the figure, it can be observed that using a $(7, 5, 1)$ quantization scheme yields performance extremely close to that of the floating-point decoder. Decreasing the range of the channel values to three bits by using the $(7, 4, 1)$ scheme significantly degrades performance. While completely removing fractional bits, $(6, 4, 0)$, yields performance that remains within 0.1 dB of the floating-point decoder throughout the entire E_b/N_0 range. This indicates that the decoder needs four bits of range for the channel LLRs. Keeping the channel LLR quantization the same, but reducing the range of the internal LLRs by one bit and using $(6, 5, 1)$ quantization does not affect the error-correction performance for $E_b/N_0 < 4.25$. After that point however, the performance starts to diverge from that of the floating-point decoder. Therefore, the range of internal LLR values increases in importance as E_b/N_0 increases. Similarly, using $(6, 4, 0)$ quantization proved sufficient for decoding the (32768, 29492) code.

From these results, we conclude that minimum number of integer quantization bits required is six for the internal LLRs and four for the channel ones and that fractional bits have a small effect on the performance of the studied polar codes. The $(6, 4, 0)$ scheme offers lower memory use for a small reduction
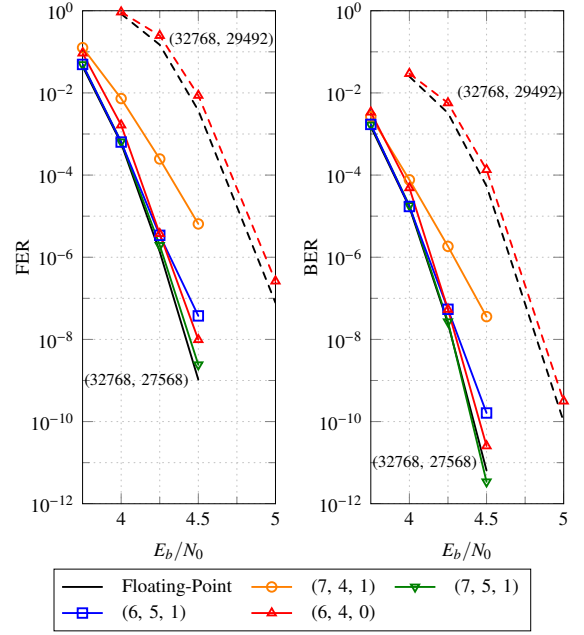
in performance and would be the recommended scheme for a practical decoder for high-rate codes. For the rest of this work, we use both the $(6, 4, 0)$ and $(7, 5, 1)$ schemes to illustrate the performance-complexity trade off between them.

G. Latency Compared to ML-SSC Decoding

The different nodes have varying effects on the latency. Table IV lists the latency, in clock cycles, of the ML-SSC decoder without utilizing any of the new node types when decoding a (32768, 29492) code. It then lists the latency of that decoder with the addition of each of the different node types individually, and finally with all of the nodes. Since this is a high rate code, $\mathcal{N}^{\text{REP}}$ nodes have a small effect on latency. An ML-SSC decoder with $\mathcal{N}^{\text{REP-SPC}}$ nodes has 89.7% the latency of the regular ML-SSC decoder, and one with $\mathcal{N}^{\text{SPC}}$ node has 63.6% the latency. Finally, the proposed decoder with all nodes has 54% the latency of the ML-SSC decoder. From these results, we conclude that $\mathcal{N}^{\text{SPC}}$ nodes have the largest effect on reducing the latency of decoding this code; however, other nodes also contribute measurably.

V. ARCHITECTURE: TOP-LEVEL

As mentioned earlier, Table III lists the 12 functions performed by the decoder. Deducing which function to perform online would require complicated controller logic. Therefore, the decoder is provided with an offline-calculated list of functions to perform. This does not reduce the decoder's flexibility
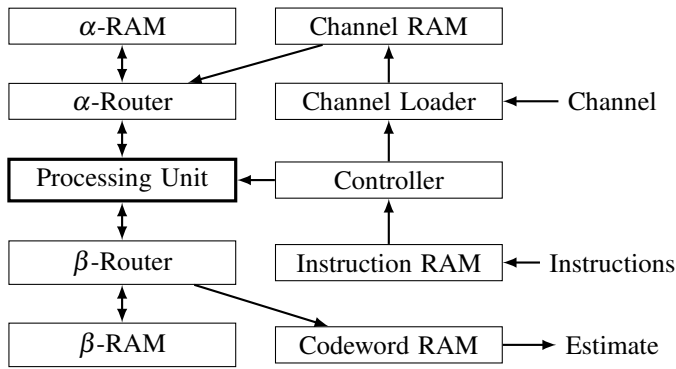
Fig. 6. Top-level architecture of the decoder.

as a new set of functions corresponding to a different code can be loaded at any time. To further simplify implementation, we present the decoder with a list of instructions, with each instruction composed of the function to be executed, and a value indicating whether the function is associated with a right or a left child in the decoder tree. An instruction requires 5 bits to store: 4 bits to encode the operation and 1 bit to indicate child association. For the $N = 32768$ codes in this work, the maximum instruction memory size was set to 3000×5 bits, which is smaller than the 32768 bits required to directly store a mask of the frozen-bit locations. This list of instructions can be viewed as a program executed by a specialized micro-processor, in this case, the decoder.

With such a view, we present the overall architecture of our decoder, shown in Fig. 6. At the beginning, the instructions (program) are loaded into the instruction RAM (instruction memory) and fetched by the controller (instruction decoder). The controller then signals the channel loader to load channel LLRs into memory, and data processing unit (ALU) to perform the correct function. The processing unit accesses data in $\alpha-$ and $\beta-$ RAMs (data memory). The estimated codeword is buffered into the codeword RAM which is accessible from outside the decoder.

By using a pre-compiled list of instructions, the controller is reduced to fetching and decoding instructions, tracking which stage is currently decoded, initiating channel LLR loading, and triggering the processing unit.

Before discussing the details of the decoder architecture, it should be noted that this work presents a complete decoder, including all input and output buffers needed to be flexible. While it is possible to reduce the size of the buffers, this is accompanied by a reduction in flexibility and limits the range of codes which can be decoded at full throughput, especially at high code rates. This trade off is explored in more detail in sections VI-A and VIII.

VI. ARCHITECTURE: DATA LOADING AND ROUTING

When designing the decoder, we have elected to include the required input and output buffers in addition to the buffers required to store internal results. To enable data loading while decoding and achieve the maximum throughput supported by the algorithm, α values were divided between two memories:

one for channel α values and the other for internal ones as described in sections VI-A and VI-B, respectively. Similarly, β values were divided between two memories as discussed in sections VI-C and VI-D. Finally, routing of data to and from the processing unit is examined in Section VI-E.

Since high throughput is the target of this design, we choose to improve timing and reduce routing complexity at the expense of logic and memory use.

A. Channel α Values

Due to the lengths of polar codes with good error-correction performance, it is not practical to present all the channel output values to the decoder simultaneously. For the proposed design, we have settled to provide the channel output in groups of 32 LLRs; so that for a code of length 32768, 1024 clock cycles are required to load one frame in the channel RAM. Since the codes of rates 0.8413 and 0.9 require 3631 and 2847 clock cycles to decode, respectively, stalling the decoder while a new frame is loaded will reduce throughput by more than 25%. Therefore, loading a new frame while currently decoding another is required to prevent throughput loss.

The method employed in this work for loading a new frame while decoding is to use a dual-port RAM that provides enough memory to store two frames. The write port of the memory is used by the channel loader to write the new frame; while the read port is used by the α-router to read the current frame. Once decoding of the current frame is finished, the reading and writing locations in the channel RAM are swapped and loading of the new frame begins. This method was selected as it allowed full throughput decoding of both rate 0.8413 and 0.9 codes without the need for a faster second write clock while maintaining a reasonable decoder input bus width of $32 \times 5 = 160$ bits, where five quantization bits are used for the channel values, or 128 bits when using $(6, 4, 0)$ quantization. Additionally, channel data can be written to the decoder at a constant rate by utilizing handshaking signals.

The decoder operates on $2P$ channel α-values simultaneously, requiring access to a $2 * 256 * 5 = 2560$-bit read bus. In order for the channel RAM to accommodate such a requirement while keeping the input bus width within practical limits, it must provide differently sized read and write buses. One approach is to use a very wide RAM and utilize a write mask; however, such wide memories are discouraged from an implementation perspective. Instead, multiple RAM banks, each has the same width as that of the input bus, are used. Data is written to one bank at a time, but read from all simultaneously. The proposed decoder utilizes $2 * 256/32 = 16$ banks each with a depth of 128 and a width of $32 * 5 = 160$ bits.

This memory cannot be merged with the one for the internal α values without stalling the decoder to load the new frame as the latter's two ports can be used by the decoder simultaneously and will not be available for another write operation.

Another method for loading-while-decoding is to replace the channel values once they are no longer required. This occurs after 2515 and 2119 clock cycles, permitting the decoder 1116 and 728 clock cycles in which to load the new frame

for the $R = 0.8413$ and $R = 0.9$ codes, respectively. Given these timing constraints, the decoder is provided sufficient time to decode the rate 0.8413 code, but not the rate 0.9 one, at full throughput. To decode the latter, either the input bus width must be increased, which might not be possible given design constraints, or a second clock, operating faster than the decoder's, must be utilized for the loading operation. This approach sacrifices the flexibility of decoding very high-rate codes for a reduction in the channel RAM size. The impact of this compromise on implementation complexity is discussed in Section VIII.

B. Internal α Values

The f (2) and g (3) functions are the only two components of the decoder that generate α values as output: each function accepts two α values as inputs and produces one. Since up to P such functions are employed simultaneously, the decoder must be capable of providing $2P$ α values and of writing P values. To support such a requirement, the internal α value RAM, denoted α-RAM, is composed of two P-LLR wide memories. A read operation provides data from both memories; while a write operation only updates one. Smaller decoder stages, which require fewer than $2P$ α values, are still assigned a complete memory word in each memory. This is performed to reduce routing and multiplexing complexity as demonstrated in [3]. Each memory can be composed of multiple RAM banks as supported by the implementation technology.

Since read from and write to α-RAM operations can be performed simultaneously, it is possible to request a read operation from the same location that is being written. In this case, the memory must provide the most recent data. To provide this functionality for synchronous RAM, a register is used to buffer newly written data and to provide it when the read and write addresses are the same [3].

C. Internal β Values

The memory used to store internal β values needs to offer greater flexibility than α-RAM, as some functions, such as COMBINE, generate $2P$ bits of β values while others, such as ML and REP, generate P or fewer bits.

The β-RAM is organized as two dual-port memories that are $2P$ bits wide each. One memory stores the output of left children while the other that of right ones. When a read operation is requested, data from both memories is read and either the lower or the upper half from each memories is selected according to whether the read address is even or odd. Similar to the α memories, the β memories can be composed of multiple banks each.

Since β-RAM is read from and written to simultaneously, using the second port of a narrower dual-port RAM and writing to two consecutive addresses to improve memory utilization is not possible as it would interfere with the read operation and reduce throughput.

D. Estimated Codeword

The estimated codeword is generated $2P = 512$ bits at a time. These estimated bits are stored in the codeword RAM in order to enable the decoder to use a bus narrower than 512 bits to convey its estimate and to start decoding the following frame immediately after finishing the current. In addition, buffering the output allows the estimate to be read at a constant rate. The codeword RAM is a simple dual-port RAM with a $2P = 512$-bit write bus and a 256-bit read bus and is organized as $N/(2P) = 64$ words of 512 bits.

Similar to the case of α value storage, this memory must remain separate from the internal β memory in order to support decoding at full speed; otherwise, decoding must be stalled while the estimated codeword is read due to lack of available ports in RAM.

E. Routing

Since both α and β values are divided between two memories, some logic is required to determine which memory to access, which is provided by the α- and β- routers.

The α-router receives stage and word indices, determines whether to fetch data from the channel or α-RAM, and calculates the read address. Only α-RAM is accessible for write operations through the α-router . Similarly, the β-router calculates addresses and determines which memory is written to; and read operations are only performed for the β-RAM by the β-router.

VII. ARCHITECTURE: DATA PROCESSING

As mentioned in Section IV, our proposed algorithm requires many decoder functions, which translate into instructions that in turn are implemented by specialized hardware blocks.

In Fig. 7, which illustrates the architecture of the data processing unit, α, β_0, and β_1 are the data inputs; while α', β_0', and β_1' are the corresponding outputs. The first multiplexer (m_0) selects either the β_0 value loaded from memory or the all-zero vector, depending on which opcode is being executed. Another multiplexer (m_1) selects the result of f or g as the α' output of the current stage. Similarly, one multiplexer (m_2) chooses which function provides the β_0' output. Finally, the last multiplexer (m_3) selects the input to the COMBINE function.

The critical path of the design passes through g, SPC, and COMBINE; therefore, these three blocks must be made fast. As a result, the merged processing element (PE) of [3] cannot be used since it has a greater propagation delay than one implementing only g. Similarly, using two's complement arithmetic, instead of sign-and-magnitude, results in a faster implementation of the g function as it performs signed addition and subtraction.

In this section, we describe the architecture of the different blocks in detail as well as justify design decisions. We omit the sign block from the detailed description since it simply selects the most significant bit of its input to implement (5).

A. The f and g Blocks

As mentioned earlier, due to timing constraints, f and g are implemented separately and use the two's complement representation. The f block contains P f elements which calculate their output by directly implementing (2). To simplify

Fig. 7. Architecture of the data processing unit.

the comparison logic, we limit the most negative number to $-2^Q + 1$ instead of -2^{-Q} so that the magnitude of an LLR contains only $Q - 1$ bits. The g element also directly implements (3) with saturation to $2^Q - 1$ and $-2^Q + 1$. This reduction in range did not affect the error-correction performance in our simulations. The combined resource utilization of an f element and a g element is slightly more than that of the merged PE [3]; however the g element is approximately 50% faster.

Using two's complement arithmetic negatively affected the speed of the f element. This, however, does not impact the overall clock frequency of the decoder since the path in which f is located is short.

Since bit-reversal is used, f and g operate on adjacent values in the input α and the outputs are correctly located in the output α' for all constituent code lengths. Special multiplexing rules would need to be added to support a non-bit-reversed implementation, increasing complexity without any positive effects [3].

B. Repetition Block

The repetition block, described in Section IV-B and denoted REP in Fig. 7, also benefits from using two's complement as its main component is an adder tree that accumulates the input, the sign of whose output is repeated to yield the β value. As can be seen in Table II, the largest constituent repetition code in the polar codes of interest is of length 16. Therefore, the adder tree is arranged into four levels. Since only the sign of the sum is used, the width of the adders was allowed to grow up in the tree to avoid saturation and the associated error-correction performance degradation. This tree is implemented using combinational logic.

When decoding a constituent code whose length N_v is smaller than 16, the last $16 - N_v$ are replaced with zeros and do not affect the result.

An attempt at simplifying logic by using a majority count of the sign of the input values caused significant reduction in

error-correction performance that was not accompanied by a perceptible reduction in the resource utilization of the decoder.

C. Repetition-SPC Block

This block corresponds to the very common node with $N_v = 8$ whose left child is a repetition code and its right an SPC code. We implement this block using two SPC nodes and one repetition node. First, four f processing elements in parallel calculate the α_{REP} vector to be fed to a small repetition decoder block. At the same time, both possible vectors of LLR values—α_{SPC_0} and α_{SPC_1}, one assuming the output of the repetition code is all zeros and the other all ones—are calculated using eight g processing elements. Those vectors are fed to the two SPC nodes SPC_0 and SPC_1.

The outputs of these SPC nodes are connected to a multiplexer. The decision β_{REP} from the repetition node is used to select between the outputs of SPC_0 and SPC_1. Finally, results are combined to form the vector of decoded bits β_v out of β_{REP} and either β_{SPC_0} or β_{SPC_1}. This node is also purely combinational.

D. Single-Parity-Check Block

Due to the large range of constituent code lengths— $[4, 8192]$—that it must decode, the SPC block is the most complex in the decoder. At its core, is a compare-select (CS) tree to find the index of the least reliable input bit as described in Section IV-A. While some small constituent codes can be decoded within a clock cycle; obtaining the input of larger codes requires multiple clock cycles. Therefore, a pipelined design with the ability to select an output from different pipeline stages is required. The depth of this pipeline is selected to optimize the overall decoding throughput by balancing the length of the critical path and the latency of the pipeline.

Table I was used as the guideline for the pipeline design. As codes with $N_v \in (0, 8]$ are the most common, their output is provided within the same clock-cycle. Using this method, pipeline registers were inserted in the CS tree so that there was a one clock cycle delay for $N_v \in (8, 64]$ and two for $N_v \in (64, 256]$. Since, in the tested codes, SPC nodes only exist in a P-RSPC or a P-0SPC configuration and they receive their input from the g elements, their maximum input size is P, not $2P$. Therefore, any constituent SPC code with $N_v > P$ receives its input in multiple clock cycles. The final stage of the pipeline handles this case by comparing the results from the current input word with that of the previous one, and updating a register as required. Therefore, for such cases, the SPC output is ready in $N_v/P + 4$ clock cycles. The extra clock cycle improved operating frequency and the overall throughput. The pipeline for the parity values utilizes the same structure.

E. Maximum-Likelihood Block

When implementing a length 16 exhaustive-search ML decoder as suggested in [7], we noted that it formed the critical path and was significantly slower than the other blocks. In addition, once repetition, SPC, and repetition-SPC decoders

TABLE V
POST-FITTING RESULTS FOR A CODE OF LENGTH 32768 ON THE ALTERA
STRATIX IV EP4SGX530KH40C2.

Algorithm	P	Q	LUTs	Registers	RAM (bits)	f (MHz)
SP-SC [3]	64	5	58,480	33,451	364,288	66
This work	64	$(6, 4, 0)$	6,830	1,388	571,800	108
		$(7, 5, 1)$	8,234	858	675,864	100
	256	$(6, 4, 0)$	25,866	7,209	536,136	108
		$(7, 5, 1)$	30,051	3,692	700,892	104

TABLE VI
INFORMATION THROUGHPUT COMPARISON FOR CODES OF LENGTH 32768
ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

Algorithm	Code rate	P	Q	T/P (Mbps)
SP-SC [3]	0.84	64	5	26
	0.9	64	5	28
This work	0.84	64	$(6, 4, 0)$	425
			$(7, 5, 1)$	406
		256	$(6, 4, 0)$	791
			$(7, 5, 1)$	775
	0.9	64	$(6, 4, 0)$	547
			$(7, 5, 1)$	523
		256	$(6, 4, 0)$	1,081
			$(7, 5, 1)$	1,077

were introduced, the number of ML nodes of length greater than four became minor. Therefore, the ML node was limited to constituent codes of length four. When enumerating these codes in the targeted polar codes, we noticed that the one with a generator matrix $G = [0001; 0100]$ was the only such code to be decoded with an ML node. The other length-four constituent codes were the rate zero, rate one, repetition, and SPC codes; other patterns never appeared. Thus, instead of implementing a generic ML node that supports all possible constituent codes of length four, only the one corresponding to $G = [0001; 0100]$ is realized. This significantly reduces the implementation complexity of this node.

The ML decoder finds the most likely codeword among the $2^{k_v} = 4$ possibilities. As only one constituent code is supported, the possible codewords are known in advance. Four adder trees of depth two calculate the reliability of each potential codeword, feeding their result into a comparator tree also of depth two. The comparison result determines which of $[0000]$, $[0001]$, $[0101]$ or $[0100]$ is the most likely codeword. This block is implemented using combinational logic only.

VIII. IMPLEMENTATION RESULTS

A. Methodology

The proposed decoder has been validated against a bit-accurate software implementation, using both functional and gate-level simulations. Random test vectors were used. The bit-accurate software implementation was used to estimate the error correction performance of the decoder and to determine acceptable quantization levels.

Logic synthesis, technology mapping, and place and route were performed to target two different FPGAs. The first is the Altera Stratix IV EP4SGX530KH40C2 and the second is the Xilinx Virtex VI XC6VLX550TL-1LFF1759. They were chosen to provide a fair comparison with state of the art decoders in literature. In both cases, we used the tools provided by the vendors, Altera Quartus II 13.0 and Xilinx ISE 13.4. Moreover, we use worst case timing estimates e.g. the maximum frequency reported for the FPGA from Altera Quartus is taken from the results of the "slow 900mV 85°C" timing model.

B. Comparison with the State of the Art SC- and SSC-based Polar Decoders

The fastest SC-based polar decoder in literature was implemented as an application-specific integrated-circuit (ASIC) [11] for a (1024, 512) polar code. Since we are interested

in better performing longer codes, we compare the proposed decoder with the FPGA-based, length 32768 implementation of [3]. Results for the same FPGA are shown in Tables V and VI. For a $(32768, 27568)$ code, our decoder is 15 to 29 times faster than the semi-parallel SC (SP-SC) decoder [3]. For the code with a rate of 0.9, it has 19 to 40 times the throughput of SP-SC depending on P and the quantization scheme used, and achieves an information throughput of 1 Gbps for both quantization schemes. It can be also noted that the proposed decoder uses significantly fewer LUTs and registers but requires more RAM, and can be clocked faster. If the decoder followed the buffering scheme of [3], namely, one input frame and no output buffering, its RAM usage would decrease to 507,248 bits for the $P = 256$, $(7, 5, 1)$ case and to 410,960 bits when $P = 64$ and the $(6, 4, 0)$ quantization scheme is used.

Although implementation results for $P = 256$ are not provided in [3], the throughput the SP-SC algorithm asymptotically approaches $0.5 \cdot f_{clk} \cdot R$ where f_{clk} is the clock frequency. Therefore, even when running at its maximum possible throughput, SP-SC remains 16 to 34 times slower than the proposed decoder for the (32768, 29492) code. The results for the rate 0.9 code with $P = 256$ and the $(7, 5, 1)$ quantization scheme were obtained using Synposis Synplify Premier F-2011.09-SP1-1 and Altera Quartus 11.1.

The two-phase successive-cancellation (TPSC) decoder is an SC-based decoder that optimizes the algorithm to reduce memory [8] and employs elements of SSC decoding to improve throughput. It is limited to values of N that are even-powers of two. Therefore, in Table VII we utilize a (16384, 14746) code constructed for $E_b/N_0 = 5$ dB and compare the resulting resource utilization and information throughput with the results of [8]. The quantization schemes used were $(6, 4, 0)$ for the proposed decoder and 5 bits for TPSC. Since [8] does not include the input buffers necessary to sustain the presented throughput, Table VII provides an extra entry, denoted TPSC*, that includes the added RAM required to buffer a second input frame. From the table, it can be observed that the proposed algorithm is eight times faster than TPSC even though the latter is running at more than twice the frequency. Additionally the proposed algorithm uses 1.7 times the LUTs and 1.2 times the registers of TPSC. When both decoder include buffers to store two received frames, the proposed algorithm uses 1.4 times the RAM of TPSC. Based on this comparison, it

TABLE VII
POST-FITTING AND INFORMATION THROUGHPUT RESULTS FOR A (16384, 14746) CODE ON THE ALTERA STRATIX IV EP4SGX530KH40C2.

Algorithm	P	LUTs	Reg.	RAM (bits)	f (MHz)	T/P (Mbps)
TPSC [8]	128	7,815	3,006	114,560	230	106
TPSC* [8]	128	7,815	3,006	196,480	230	106
This work	128	13,388	3,688	273,740	106	824
This work	256	25,219	6,529	285,336	106	1,091

TABLE VIII
COMPARISON WITH AN LDPC CODE OF SIMILAR ERROR CORRECTING PERFORMANCE, ON THE XILINX VIRTEX VI XC6VLX550TL.

Code	Q	LUTs	f_{max} (MHz)	T/P (Gbps)
LDPC [14]	4	99,468	30.7	1.102
This work	(6, 4, 0)	18,024	71.3	0.542
	(7, 5, 1)	21,700	71.0	0.539

can be concluded that TPSC cannot match the throughput of the proposed algorithm with the same complexity by utilizing multiple decoders decoding different frames simultaneously since the resulting TPSC system will utilize more than four times the resources of the proposed decoder. The last entry in the table presents the results achievable by the proposed decoder with $P = 256$, where the information throughput is ~ 1.1 Gbps.

C. Comparison with an LDPC code of similar error correcting performance

A fully-parallel (2048, 1723) LDPC decoder on FPGA is presented in [14]. At 30.7 MHz on a Xilinx Virtex VI XC6VLX550TL, an information throughput of 1.1 Gbps is reached. Early termination could be used to achieve 8.8 Gbps at 5 dB, however that would require support for early termination circuitry and extra buffering that were not implemented in [14].

Results for our decoder with $P = 256$ and a (32768, 27568) polar code implemented on the same FPGA as the LDPC decoder are shown in Table VIII. Our decoder requires 5 times fewer LUTs, but only achieves half of the throughput.

IX. CONCLUSION

In this work we presented a new algorithm for decoding polar codes that results in a high-throughput, flexible decoder. An FPGA implementation of the proposed algorithm was able to achieve an information throughput of 1 Gbps when decoding a (32768, 29492) polar code with a clock frequency of 108 MHz. We expect derivative works implementing this decoder as an ASIC to reach a throughput of 3 Gbps when operating at 300 MHz with a complexity lower than that required by LDPC decoders of similar error correction performance. Thus, our results indicate that polar codes are promising candidates for data storage systems.

ACKNOWLEDGEMENT

REFERENCES

[1] E. Arikan, "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] A. Eslami and H. Pishro-Nik, "On bit error rate performance of polar codes in finite regime," in *Proc. 48th Annual Allerton Conf. Communication, Control, and Computing (Allerton)*, 2010, pp. 188–194.

[3] C. Leroux, A. Raymond, G. Sarkis, and W. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Trans. Signal Process.*, vol. 61, no. 2, pp. 289–299, 2013.

[4] I. Tal and A. Vardy, "List decoding of polar codes," *CoRR*, vol. abs/1206.0050, 2012. [Online]. Available: http://arxiv.org/abs/1206.0050v1

[5] P. Chicoine, M. Hassner, M. Noblitt, G. Silvus, B. Weber, and E. Grochowski, "Hard disk drive long data sector white paper," Technical report, The International Disk Drive Equipment and Materials Association (IDEMA), Tech. Rep., 2007.

[6] 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.

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

[8] A. Pamuk and E. Arikan, "A two phase successive cancellation decoder architecture for polar codes," in *Proc. IEEE International Symposium on Information Theory ISIT 2013*, Jul. 2013, pp. 1–5.

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

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

[11] A. Mishra, A. Raymond, L. Amaru, G. Sarkis, C. Leroux, P. Meinerzhagen, A. Burg, and W. Gross, "A successive cancellation decoder asic for a 1024-bit polar code in 180nm cmos," in *Solid State Circuits Conference (A-SSCC), 2012 IEEE Asian*, 2012, pp. 205–208.

[12] J. Snyders and Y. Be'ery, "Maximum likelihood soft decoding of binary block codes and decoders for the Golay codes," *IEEE Trans. Inf. Theory*, vol. 35, no. 5, pp. 963–975, 1989.

[13] A. Alamdar-Yazdi and F. R. Kschischang, "Locally-reduced polar codes," personal communication, 2012.

[14] V. Torres, A. Perez-Pascual, T. Sansaloni, and J. Valls, "Fully-parallel LUT-based (2048,1723) LDPC code decoder for FPGA," in *Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on*, 2012, pp. 408–411.

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.

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 DVCON'05, 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.

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.

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.