A 638 Mbps Low-Complexity Rate 1/2 Polar Decoder on FPGAs

Pascal Giard*, Gabi Sarkis*, Claude Thibeault‡, and Warren J. Gross*
*Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada.
Email: {pascal.giard, gabi.sarkis}@mail.mcgill.ca, warren.gross@mcgill.ca
‡Department of Electrical Engineering, École de technologie supérieure, Montréal, Québec, Canada.
Email: claude.thibeault@etsmtl.ca

***Abstract*—Polar codes are capacity-achieving error-correcting codes with an explicit construction that can be decoded with low-complexity algorithms. In this work, we show how the state-of-the-art low-complexity decoding algorithm can be improved to better accommodate low-rate codes. Dedicated hardware is added to efficiently decode new constituent codes. Also, we use polar code construction alteration to further improve the latency and throughput. A polar decoder for a (1024, 512) code is implemented on two different FPGAs. It has 25% lower latency over the previous work and a coded throughput of 436 Mbps and 638 Mbps on the Xilinx Virtex 6 and Altera Stratix IV FPGAs, respectively.**

I. Introduction

Polar codes are a family of capacity-achieving error-correcting codes with an explicit construction introduced by Arıkan [1]. They were proved to achieve the capacity of binary symmetric memoryless channels when decoded with the low-complexity successive-cancellation (SC) algorithm. However, the SC decoding algorithm is sequential in nature leading to decoder implementations with high latency and low throughput.

To increase speed, two new decoding algorithms derived from SC were introduced [2], [3]. These two algorithms work by using dedicated algorithms on parts of a polar code; they exploit the recursive construction of polar codes and the a priori knowledge of the information bits' location. The more efficient of the two is the Fast-SSC algorithm introduced in [3].

While the Fast-SSC algorithm represents a significant improvement over the previous decoding algorithms, it works better with high-rate codes. In this paper, modifications to the algorithm and implementation are proposed that make it perform better for low-rate codes. We also alter the polar code construction to further improve latency and throughput at the cost of a small error-correction performance degradation.

We start this paper by providing some background in Section II. We then discuss how a polar code construction is altered to improve the latency and throughput of a decoder in Section III. In Section IV, modifications to the original Fast-SSC algorithms are proposed in order to improve the same metrics. Sections V and VI present the implementation details along with the detailed results for two different FPGAs. Finally, Section VII concludes this paper.

II. Background

A. Polar Codes

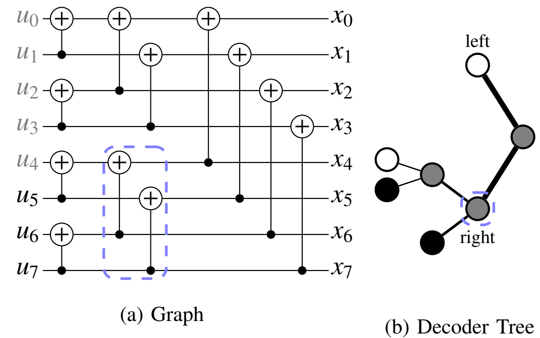
Polar codes are a type of block code with a recursively-built generator matrix, i.e. a polar code of length N is built from the concatenation of two constituent polar codes of length $N_v = N/2$.

An (N, k) polar code has a length of N bits, contains k information bits, and $N - k$ parity bits. The encoding process can be described as follows. A vector containing the k information bits is first expanded into a vector of length N by inserting frozen bits at given locations. The resulting vector is then multiplied by the generator matrix to yield the polar codeword. The frozen bits are usually set to zero and their optimal locations depend on the channel type and condition as discussed in [4].

It was shown in [5] that polar codes can be encoded and decoded systematically. This leads to an improved bit-error rate (BER) without affecting the frame-error rate (FER). In this work, systematic polar codes are used.

B. From graph to decoder trees

As shown in [2], [3], polar codes can be represented as trees. Using only the three node types introduced in [2]—rate-0, rate-1 and rate-R—the graph of Fig. 1a can be represented as the decoder tree of Fig. 1b. The black and gray u_i labels of Fig. 1a correspond to information and frozen bits, respectively.



(a) Graph (b) Decoder Tree

Fig. 1: Representing polar codes.

In Fig. 1b, the rate-0 nodes are white, rate-1 nodes are black and the gray ones are of rate R, where $0 < R < 1$. Rate-R

nodes can be seen as constituent codes with an associated frozen bit pattern. As will be shown in the next section, a decoder tree can be further trimmed when specialized node types are created to decode certain constituent codes of rate-R.

C. Fast-SSC Decoding

The Fast-SSC algorithm presented in [3] further trims the decoder tree by using different, more efficient algorithms to decode some constituent codes. The constituent codes directly decoded in the Fast-SSC algorithm have different maximum lengths N_v depending on the complexity of the corresponding decoding algorithm e.g. for a repetition node, N_v was set to 16 while it was set to 4 for an exhaustive search maximum-likelihood (ML) node. Beyond direct decoding of constituent codes, other methods were proposed to exploit the code structure, leading to reduced decoding latency and increased throughput.

The decoder tree node types corresponding to the decoding algorithm of [3] are summarized in Table I. Note that the 01 node was named ML in [3]. We briefly review the most important operations and leaf node types below.

1) F Operations: The F operation generates the messages to be sent to a left child and is performed using the min-sum approximation as defined in [6]:

$$\begin{aligned}\alpha_l[i] &= F(\alpha_v[i], \alpha_v[i + {}^{N_v}/_2]) \\ &= \text{sgn}(\alpha_v[i])\text{sgn}(\alpha_v[i + {}^{N_v}/_2]) \min(|\alpha_v[i]|, |\alpha_v[i + {}^{N_v}/_2]|),\end{aligned} \quad (1)$$

where α_v are soft reliability values, represented as log-likelihood ratios (LLRs), from the parent node, and N_v is the node input length.

2) G and G_0R Operations: The G operation generates the messages to be sent to a right child node. It is performed as defined in [6]:

$$\begin{aligned}\alpha_r[i] &= G(\alpha_v[i], \alpha_v[i + {}^{N_v}/_2], \beta_l[i]) \\ &= \begin{cases}\alpha_v[i + {}^{N_v}/_2] + \alpha_v[i], & \text{when } \beta_l[i] = 0; \\ \alpha_v[i + {}^{N_v}/_2] - \alpha_v[i], & \text{otherwise,}\end{cases}\end{aligned} \quad (2)$$

where β_l is the bit estimate vector generated by the left sibling in the subtree.

The G_0R operation is a special case of the G operation, where the left-hand-side sibling in the subtree is a rate-0 node, i.e. β_l is a all-zero vector.

3) Combine and Combine_0R Operations: The *Combine* operation corresponds to the concatenation of two bit-estimate vectors generated from the left and right child nodes. As an example, going up the tree, the node circled in blue in Fig. 1b combines the bit-estimate vectors from its children to provide the root node of the decoder tree with a bit estimate vector for the right-hand-side subtree. In the graph representation illustrated in Fig. 1a, the same operation is also circled in blue and illustrates how the *Combine* operation calculates the bit estimate vector β_v:

$$\beta_v[i] = \begin{cases}\beta_l[i] \oplus \beta_r[i], & \text{when } i < {}^{N_v}/_2; \\ \beta_r[i - {}^{N_v}/_2], & \text{otherwise,}\end{cases} \quad (3)$$

TABLE I: Decoder tree node types supported by the original Fast-SSC polar decoder.

Name	Color	Description
0R	White and gray	Half-left side is frozen.
R1	Gray and black	Half-right side is all information.
RSPC	Gray and yellow	Half-right side is an SPC code.
0SPC	White and yellow	Half-left side is frozen, half-right side is an SPC code.
Rep	Green	Repetition code, maximum length N_v of 16.
RepSPC	Green and yellow	Concatenation of a Repetition code on the left and an SPC code on the right, $N_v = 8$.
01	Black and white	Fixed-length pattern $N_v = 4$ where the left half is frozen and the right half is all one.
rate-R	Gray	Mixed rate node.

where β_l and β_r are bit estimates emanating from the left and right child nodes, respectively.

Similar to the G_0R operation, the $Combine_0R$ operation is a special case where the left-hand-side sibling in the subtree is a rate-0 node i.e. β_l is a all-zero vector.

4) Repetition Nodes: Repetition nodes provide the bit estimate vector for a repetition code. A repetition code contains a single information bit that is replicated over the length N_v of the code. The maximum-likelihood decoding rule for these codes is to perform threshold detection on the sum of the input LLRs [3].

5) Single-parity-check Nodes: Single-parity-check (SPC) nodes provide bit estimates for SPC codes. The least significant bit of an SPC code is the parity bit of the other, information, bits. The maximum likelihood algorithm to decode them consists of first verifying that the parity bit is consistent with the hard decisions on the information bits' value. In case where it is not, the hard decision of the least reliable information bit is flipped.

The remaining nodes are a concatenation of the operations and nodes described above, or an extension. For example, the RepSPC node is the concatenation of a Repetition node with an SPC node with a constant length $N_v = 8$. Another example is the 0SPC node, which is the concatenation of a rate-0 node and an SPC node. Thus, it replaces the execution of three operations: a G_0R operation, an SPC operation and a $Combine_0R$ operation.

With the specialized nodes and operations of Fast-SSC, the trimmed decoder tree for the $(8, 3)$ polar code illustrated in Fig. 1 is made of a single 0SPC node.

III. Altering the Code Construction

In [7], [8], formalized methods for altering the construction of polar codes with the aim of optimizing the performance of a simplified successive-cancellation (SSC) decoder were presented. In this section, we show how we use polar code construction alterations to significantly improve the latency and throughput of a hardware polar decoder utilizing the Fast-SSC decoding algorithm.

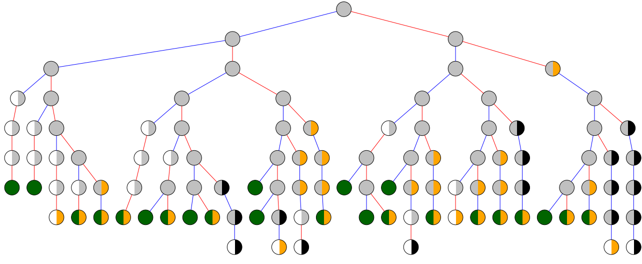


Fig. 2: Decoder tree for the polar code built using [4] and decoded with the nodes and operations of [3].

A. Original Construction

As mentioned in Section II-A, a good polar code is constructed by selecting which bits to freeze, according to the type of channel and its conditions. Many construction methods using approximations were published e.g. [1], [4], [9], [10].

Fig. 2 shows the decoder tree corresponding to the $(1024, 512)$ polar code constructed using the technique of [4] where only the node types defined in Table I are used with the same constraints of [3]. The polar code was optimized for a E_b/N_0 of 2.5 dB. The F and G blocks are constrained to a maximum of $P = 256$ inputs meaning that, for nodes with a length $N_v > P$, $\lceil N_v/P \rceil$ cycles are required. Thus the decoding latency to decode the tree of Fig. 2 using the algorithm and implementation of [3] is 220 clock cycles (CC) and the coded throughput is 4.65 bits/CC.

Altering a polar code to further trim the decoder tree can result in a significant latency reduction, without affecting the code rate. By making these modifications however, the error-correction performance is degraded. Although, as will be shown in the next section, the impact can be small, especially if the number of changes is limited.

B. Altered Construction

As shown in [8], it is possible to improve the latency of an SC-based polar decoder by changing the location of frozen bits. We do so by freezing a bit location that was previously set to carry an information bit and by unfreezing a bit location that will now hold an information bit. We call this process "bit swapping". This technique does not alter the code rate as the total number of information and frozen bits remain the same.

The bits to swap are determined to meet two objectives. The first one is improving the latency and throughput of the decoder. Thus, these bit swaps must eliminate constituent codes for which we do not have an efficient decoding algorithm and create ones for which we do. The second one is minimizing the impact on the error-correction performance. To do so, we alter the state of the least-reliable information and the most-reliable frozen bits whose reliability values are similar.

Respecting the above, we were able to decrease decoding latency from 220 to 189 clock cycles, a 14% improvement, with 5 bit swaps. That increases the coded throughput to 5.42 bits/CC. The corresponding decoder tree is shown in Fig. 3.

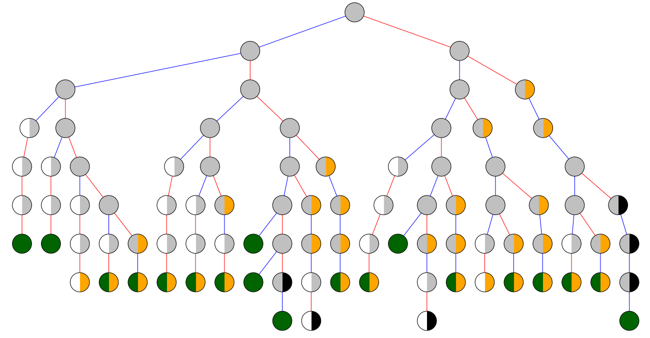


Fig. 3: Decoder tree for the altered polar code.

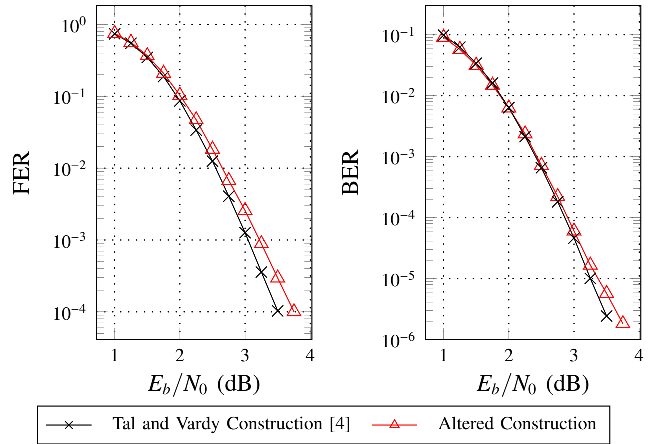


Fig. 4: Error-correction performance using binary phase shift keying over an additive white Gaussian channel of the altered code compared to that of the original code.

As expected, the error-correction performance of the altered code is degraded as illustrated in Fig. 4. However, the loss amounts to less than 0.25 dB at a FER of 10^{-4}. For wireless applications, which are usually the target for codes of such lengths and rates, this represents the FER range of interest.

IV. New Constituent Decoders

Looking at the decoder tree of Fig. 3, it can be seen that some frozen bit patterns occur often. Adding support for more constituent codes to the Fast-SSC algorithm will result in a reduced latency and increased throughput under the constraint that the corresponding computation nodes do not significantly lengthens the critical path of a hardware implementation. As a result of an investigation, the constituent codes of Table II were added. Furthermore, post-place and route timing analysis showed that the maximum length N_v of a Repetition node could be increased from 16 to 32 without affecting the critical path.

The new decoder tree shown in Fig. 5 has a decoding latency of 165 clock cycles, a 13% improvement over the decoder tree of Fig. 3 decoded with the original Fast-SSC algorithm. Thus, the coded throughput is 6.206 bits/CC for this polar code.

TABLE II: New functions performed by the proposed decoder.

Name	Color	Description
Rep1	Green and black	Repetition code on the left, rate-1 code on the right, maximum length N_v of 8.
0RepSPC	White and lilac	Rate-0 code on the left, RepSPC code on the right, $N_v = 16$.
001	$\frac{3}{4}$ white and $\frac{1}{4}$ black	Rate-0 code on the left, 01 code on the right, $N_v = 8$.

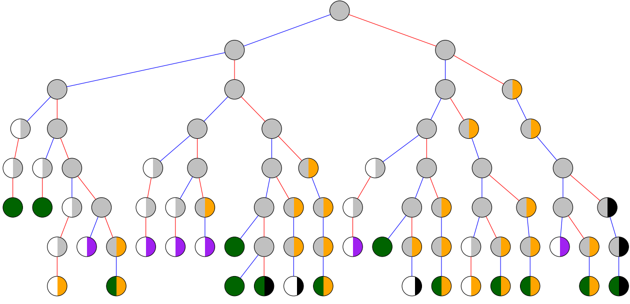


Fig. 5: Decoder tree for the altered polar code with the added nodes.

To summarize, Table III lists the frozen bit patterns that can be decoded by leaf nodes. It can be seen that the smallest possible leaf node has length $N_v = 4$ while our proposed decoder tree shown in Fig. 5 has a minimum length $N_v = 8$. In other words, Fig. 5 is representative of the patterns listed in Table III but not comprehensive.

V. Implementation

A. Quantization

Let Q_i be the total number of bits used to represent LLRs internally, Q_c be the total number of bits to represent channel LLRs, and Q_f be the number of bits among Q_i or Q_c used to represent the fractional part of any LLR. It was found through simulations that using $Q_i.Q_c.Q_f = 6.5.1$ quantization led to an error-correction performance very close to that of the floating-point number representation as can be seen in Fig. 6.

B. Rep1 Node

The Rep1 node provides a bit estimate vector for the Rep1 code of fixed length $N_v = 8$. The bit estimate vector β_0^7 is calculated using operations described in the previous sections. However, instead of performing the required operations sequentially, the dedicated hardware preemptively calculates intermediate soft values.

Fig. 7 shows the architecture of the Rep1 node. It can be seen that there are two G blocks. One preemptively calculates soft values assuming that the Rep block will output $\beta = 0$ and the other for $\beta = 1$. The Rep block provides a single bit estimate corresponding to the information bit the repetition code of length $N_v = 4$ it is decoding. The outputs of the G blocks go through a Sign block to generate hard decisions. The correct hard decision vector is then selected using the output

TABLE III: Frozen bit patterns decoded by leaf nodes.

Name	Pattern
Rep	0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001
Rep1	0001 1111
0SPC	0000 0111
RepSPC	0001 0111
0RepSPC	0000 0000 0001 0111
01	0011
001	0000 0011

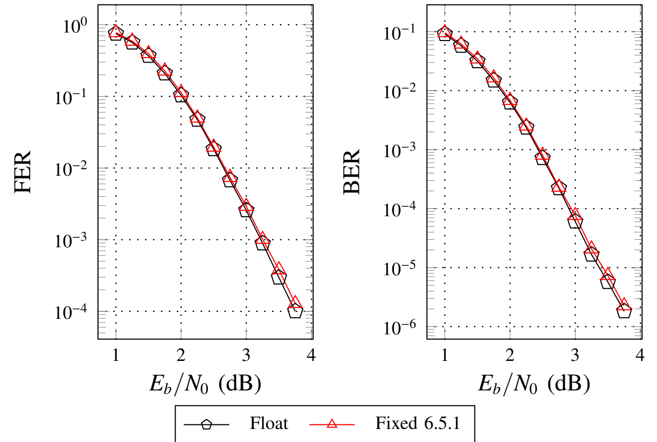


Fig. 6: Impact of quantization on error-correction performance.

of the Rep block. Finally, the bit estimate vector β_0^7 is built. The highest part, β_4^7, is always comprised of the multiplexer output. The lowest part, β_0^3, is either a copy of same output or its inverse. The inverse is selected when the output of the Rep block is 1.

Calculations are carried out in one clock cycle. The output of the F, G and Rep blocks are not stored in memory. Only the final result, the bit-estimate vector β_0^7, is stored in memory.

C. High-Level Architecture

The decoder resembles a processor. The high-level architecture of the decoder is presented in Fig. 8. When the decoder is started, the controller signals the channel loader to start storing channel LLRs, 32 LLRs (160 bits) per clock cycle, into the channel RAM. The controller then starts to execute functions on the processing unit. The processing unit reads LLRs from the Channel or α-RAM and writes LLRs to the α-RAM. It reads or writes hard decisions to the β-RAM. The last *Combine* operation writes the estimated codeword into the Codeword RAM, a memory accessible from outside the decoder.

The decoder is complete with all input and output buffers to accommodate loading a new frame and reading an estimated codeword while a frame is being decoded. The required memory could be made smaller if the nominal throughput required is lower. The loading or outputting of a full frame takes less clock cycles than the actual decoding, we have

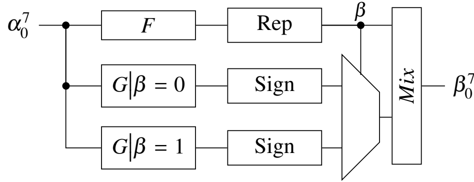


Fig. 7: Architecture of the Rep1 Node.

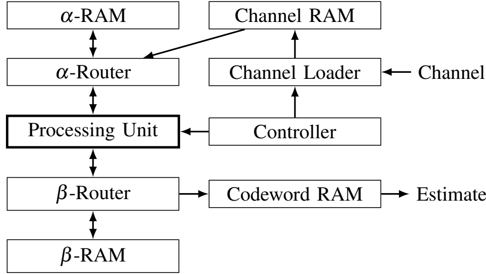


Fig. 8: High-level architecture of the decoder.

a pipelined operation; under normal operation, the decoder should not be slowed down by the I/O operations.

D. Processing Unit or Processor

The core of the decoder is the processing unit or processor illustrated in Fig. 9 and based on the Fast-SSC implementation of [3]. Thus, the processor features all the modules required to implement the nodes and operations described in sections II-C and IV. Notably, the 01 and RepSPC blocks connected to the G block implement the 001 and 0RepSPC nodes, respectively, where the all-zero vector input is selected at the mux m_0. The critical path of the decoder corresponds to the 0RepSPC node i.e. goes through G, RepSPC, the mux m_3, *Combine* and the mux m_2. It is slightly longer than that of [3].

VI. Results

A. Methodology

Random codewords were generated for transmission using binary phase shift keying (BPSK) over an additive white Gaussian noise (AWGN) channel using a software model. The functionality of the design was verified at the RTL level using a simulator. We then ran post-place and route with timing verification of the hardware implementation for two different FPGAs using the same simulator. Finally, the design was realized with an FPGA and a testbench was run using FPGA-in-the-loop with MATLAB and a Xilinx ML605 board.

B. Error-Correction Performance

To validate that the error-correction performance of the hardware implementation matches expectations, a large number of frames[1] are first decoded through post-place and route with timing simulation. The netlists with timing annotations were generated by Xilinx XST. Later, the same frames are decoded on an FPGA using the FPGA-in-the-loop setup. All results were in agreement.

[1]For all E_b/N_0 values, a minimum of 100 frames in errors were simulated.

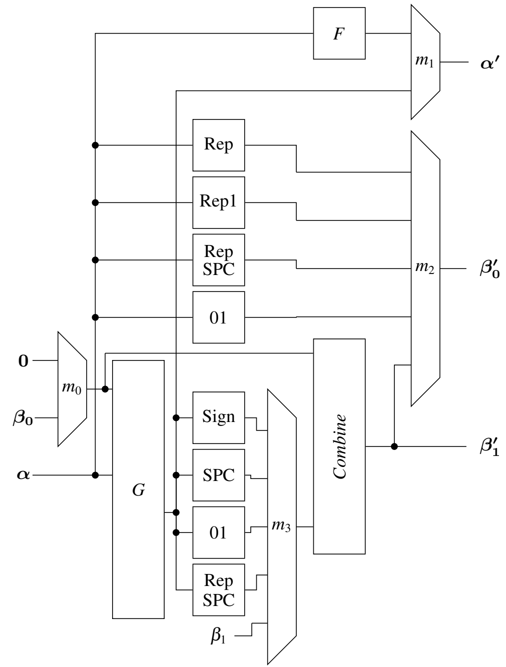


Fig. 9: Architecture of the processor.

TABLE IV: Post-fitting and coded throughput results for the proposed (1024, 512) code for two FPGAs.

FPGA	LUTs	Reg.	RAM (kbits)	f (MHz)	T/P (Mbps)
Xilinx Virtex 6	22,115	7,941	2,106	70	436
Altera Stratix IV	24,821	5,823	36	103	638

C. Area and Frequency

In this section, post-fitting results are compared for two different FPGAs: the Xilinx Virtex 6 XC6VLX240T-2 featured on the Xilinx ML605 board and the Altera Stratix IV EP4SGX530KH40C2 present on the terASIC DE4 board. Although the RTL implementation remains essentially the same, there are 2 minor differences. For the Xilinx implementation, memory (RAM) bypasses were manually implemented and synchronous reset signals were used.

Table IV shows the results. It can be observed that the number of look-up tables (LUTs) and registers required are very similar for both FPGAs. The RAM usage differs greatly. We note that Altera reports the effective RAM usage e.g. while 181 9-kbit RAM blocks are used, only 35,840 bits (2.2%) need to be stored. Xilinx reports the amount of 36-kbit and 18-kbit RAM blocks allocated but, contrary to Altera, does not specify the amount effectively used. The most noteworthy difference is the maximum clock frequency f, a gap greater than 30%.

Timing reports show that the critical path is the same on both FPGAs, and corresponds to the 0RepSPC node. Xilinx reports that its longest path amounts to 27 logic stages whereas the same path has only 19 stages on Altera. While that alone significantly contributes to the lower achievable clock frequency on the Xilinx FPGA, examining the critical path

TABLE V: Comparing implementations of decoders for a (1024, 512) polar code on an Altera Stratix IV FPGA.

Implementation	LUTs	Regs.	RAM (kbits)	f (MHz)	Latency (μs)	T/P (Mbps)
[11]	1,940	748	7.1	239	9.14	112
[3]*	23,020	1,024	42.8	103	2.14	475
altered code					1.83	553
proposed decoder	24,821	5,823	35.8	103	1.60	638

TABLE VI: Comparing implementations of decoders for a (1024, 512) polar code on a Xilinx Virtex 6 FPGA.

FPGA	LUTs	Regs.	RAM (kbits)	f (MHz)	Latency (μs)	T/P (Mbps)
[12]	193,456	6,151	N/A	0.6	3.41	600
	190,127	22,928	N/A	N/A	N/A	1,240
this work	22,115	7,941	2,106	70.2	2.35	436

also reveals some routing congestion on that FPGA.

D. Comparison with Other Works

In this section, the proposed decoder is compared with that of the state-of-the-art decoders and with that of the original Fast-SSC implementation [3]. The latter was resynthesized so that the decoder only has to accommodate polar codes of length $N = 1024$ and is marked with an asterisk (*) in Table V. We also show the effect of using a polar code altered as described in Section III with the Fast-SSC decoder. The result is listed as 'altered code' and has the same resource usage and clock frequency as [3]* since that decoder can decode any polar code of length $N = 1024$ by changing the code description in memory.

Table V shows that the proposed decoder achieves the best latency among all decoders. Its throughput is 5.7 times greater than the two-phase decoder of [11] and 15% to 34% greater than the Fast-SSC decoder of [3].

Our work requires an 8% increased in used LUTs in order to improve the throughput by more than 15%. The difference in registers and RAM can be mostly attributed to SRAM to register conversions, a measure taken by the fitter to shorten the critical path to meet the requested clock frequency. With leaf nodes with a minimum length N_v of 8 instead of 4, the combined memory usage shows a modest reduction of under 5% at 43.8 kbits for [3] compared to the 41.6 kbits of the proposed decoder.

Table VI compares the proposed decoder with that of [12]. Note that [12] presents results for a Xilinx Virtex 6 XC6VLX550T, an FPGA that is twice as big as the XC6VLX240T we used but is from the same family. The amount of RAM required is not reported in [12]. It can still be seen that the resource usage is an order of magnitude smaller at comparable throughput, and the latency about 30% lower at 2.35 μs versus 3.41 μs. The fastest FPGA implementation presented [12] is pipelined, making its throughput almost 3 times greater compared to that of our work. Its latency and maximum operation frequency were not reported.

VII. Conclusion

In this work, we showed how the original Fast-SSC algorithm implementation could be improved by adding dedicated decoders for three new types of constituent codes frequently appearing in low-rate codes. We also used polar code construction alterations to significantly improve the latency and throughput of a Fast-SSC decoder at the cost of a small error-correction performance loss. A proof of concept polar decoder for a $(1024, 512)$ code with competitive error-correction performance was implemented on two common FPGAs. The resulting latency of 165 clock cycles is a 25% improvement over the previous work. The coded throughput was shown to be 436 Mbps and 638 Mbps for the Xilinx Virtex 6 and Altera Stratix IV FPGAs, respectively.

Acknowledgement

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Claude Thibeault is a member of ReSMiQ. Warren J. Gross is a member of ReSMiQ and SYTACom.

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] 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, Dec. 2011.

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

[4] I. Tal and A. Vardy, "How to construct polar codes," *IEEE Trans. Inf. Theory*, vol. 59, no. 10, pp. 6562–6582, Oct 2013.

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

[6] C. Leroux, I. Tal, A. Vardy, and W. Gross, "Hardware architectures for successive cancellation decoding of polar codes," in *IEEE Int. Conf. on Acoust., Speech, and Signal Process. (ICASSP)*, May 2011, pp. 1665–1668.

[7] Z. Huang, C. Diao, and M. Chen, "Latency reduced method for modified successive cancellation decoding of polar codes," *Electron. Lett.*, vol. 48, no. 23, pp. 1505–1506, Nov 2012.

[8] A. Balatsoukas-Stimming, G. Karakonstantis, and A. Burg, "Enabling complexity-performance trade-offs for successive cancellation decoding of polar codes," in *IEEE Int. Symp. on Inf. Theory (ISIT)*, 2014, pp. 2977–2981.

[9] R. Mori and T. Tanaka, "Performance and construction of polar codes on symmetric binary-input memoryless channels," in *IEEE Int. Symp. on Inf. Theory (ISIT)*, 2009, pp. 1496–1500.

[10] P. Trifonov, "Efficient design and decoding of polar codes," *IEEE Trans. Commun.*, vol. 60, no. 11, pp. 3221–3227, 2012.

[11] A. Pamuk and E. Arıkan, "A two phase successive cancellation decoder architecture for polar codes," in *IEEE Int. Symp. on Inf. Theory (ISIT)*, Jul. 2013, pp. 1–5.

[12] O. Dizdar and E. Arıkan, "A high-throughput energy-efficient implementation of successive-cancellation decoder for polar codes using combinational logic," *CoRR*, vol. abs/1412.3829, Mar 2015. [Online]. Available: http://arxiv.org/abs/1412.3829