MEMORY REDUCTION TECHNIQUES FOR SUCCESSIVE CANCELLATION DECODING OF POLAR CODES

Bertrand Le Gal, Camille Leroux and Christophe Jego

Bordeaux INP, IMS Laboratory, University of Bordeaux, Talence, France

ABSTRACT

Polar coding is a new coding scheme that asymptotically achieves the capacity of several communication channels. Polar codes can be decoded with a successive cancellation (SC) decoder. In terms of hardware implementation, architectural performance of SC decoders is limited by the memory complexity. In this paper, two complementary methods are proposed to reduce the memory footprint of current state-of-the-art SC decoders. These methods must also applicable to SC-List decoders. The impacts the decoding performance in a rather negligible manner (<0.02dB), as shown by performed simulations. The association of both methods allows a reduction of 16 $\sim$35% of the memory complexity for SC decoders depending on their quantization format.

Index Terms— Error-correcting codes, polar codes, successive-cancellation decoding, hardware implementation

1. INTRODUCTION

Polar codes (PC) constitute a family of error correction codes that asymptotically reach the capacity of various communication channels [1]. The associated decoder is the successive cancellation (SC) decoder. Several SC decoder architectures were proposed in the literature to improve the throughput and to reduce the computational complexity [2–5]. In [5], an SC decoder for a size $N = 2^{20}$ was implemented on an FPGA device. This decoder only uses 2% of the computation resources while consuming 72% of the available memory. It demonstrates the existing asymmetry between memory and computation complexities in current SC decoders. ASIC results in [6] lead to the same conclusion: 90% of the silicon area is due to the memory. In this paper, two memory optimization methods are proposed. The first method suggests to recalculate half of the Log-Likelihood Ratios (LLRs) instead of storing them into the memory. It reduces the channel memory complexity without affecting the FER decoding performance. The second method reduces the dynamic range of internal LLRs. It induces a negligible decoding performance loss (<0.02dB). The combination of the two methods reduces the memory complexity by 16 $\sim$ 35%. These methods could be applied to other Polar codes decoding algorithms that suffer from the same drawbacks e.g. list decoding [7, 8].

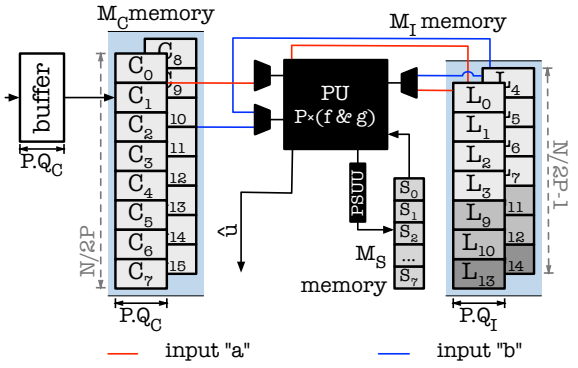

Fig. 1. Architecture of SC decoder [5] with $N = 16$ et $P = 1$

In the remainder of the paper, the architecture of a state of the art SC decoder is presented in Section 2. Then, the two proposed memory reduction techniques are successively presented in Sections 3 and 4. The improvement in terms of memory complexity is then estimated on state of the art architectures in Section 5.

2. STATE-OF-THE-ART SC DECODER ARCHITECTURE

Figure 1 depicts the simplified architecture of a state-of-the-art semi-parallel SC decoder as it was proposed in [3], and then improved in [5]. Without loss of generality, and for the sake of clarity, a small code length ($N = 16$) and a parallelism level of one ($P = 1$) were selected. For a size N and a parallelism level P, the SC decoder consists of the following elements: (i) a *channel buffer* storing the input LLRs into packets of P LLRs; (ii) a *channel memory* M_C storing data coming from the channel buffer. M_C is divided into two banks, each including $\frac{N}{2P}$ addresses. A total of P Q_C-bits LLRs are stored in each address; (iii) an *internal memory* M_I storing intermediate LLR values. M_I is divided into two banks each including $\frac{N}{2P}$ addresses. Internal LLRs are quantized with $Q_I = (Q_C + \Delta)$ bits; (iv) a *processing unit* (PU) which includes P processing elements (PE); (v) a *partial sum update unit* (PSUU), whose complexity grows only with P [5]; and (vi) a *partial sum memory* M_S of N bits which stores PSUU results.

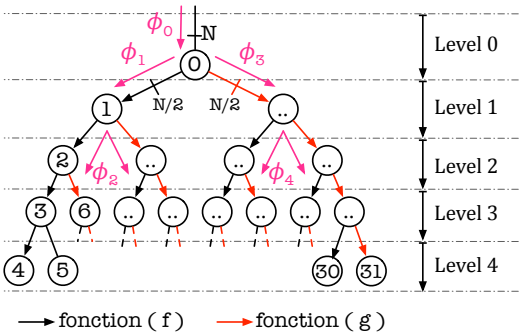

Fig. 2. SC decoding on a binary tree

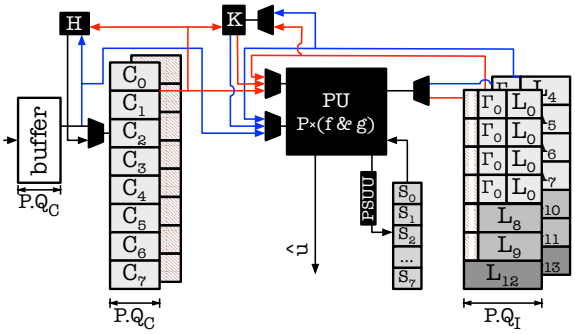

Fig. 3. SC decoder with LLR recomputation and dynamic reduction

It was shown in [3] that even with a relatively low P value ($P \approx 64$), the decoder throughput is hardly affected. In terms of complexity, the area consumed by the channel buffer, the PU and the PSUU grows linearly with P and is independent of N. As a consequence, the decoder total area is dominated by the memory complexity which grows linearly with N as shown in [5]. The memory footprint of an SC decoder can be calculated using Equation 1. $A(M_x)$ is the memory footprint of the memory M_x in bits. M_R represents the miscellaneous memories that are used for buffering, storing the frozen bits, etc. Since $A(M_R)$ is hard to estimate and often negligible, we will focus our analysis on reducing $A_0 = A'_0 - A(M_P)$. In order to have a fair estimation of the memory reduction, A'_0 will be considered in the experimental results section.

$$A'_0 = A(M_C) + A(M_I) + A(M_S) + A(M_R) \quad (1)$$

$$A_0 = Q_C \times N + (Q_C + \Delta) \times N + N \quad (2)$$

The SC decoding process consists of five phases. In **Phase** ϕ_0: N channel LLRs (C_i) are shifted in one by one. The channel buffer packs P consecutive LLRs and then send them in parallel to M_C. A total of N/P packets of $P \times Q_C$ bits are sent to M_C. In **Phase** ϕ_1: f functions are computed based on the LLRs read from M_C. The results are then stored in the internal memory M_I. In **Phase** ϕ_2: a succession of f and g functions are computed based on LLRs (L_i) read from M_I. The results are stored back into M_I. In the meantime, partial sums are updated and stored in M_S. In **Phase** ϕ_3: g function are computed based on the LLRs read from M_C and partial sums read from M_S. Results are stored in M_I. In **Phase** ϕ_4: a succession of f and g functions are computed based on LLRs stored in M_I. The results are stored in this same memory. At the end of this phase, the codeword is fully decoded. As suggested in [2] f and g functions can be calculated such that:

$$f(a,b) = sgn(a).sgn(b).\min(|a|,|b|) \quad (3)$$

$$g(a,b,s) = (-1)^s a + b \quad (4)$$

3. RECOMPUTATION TECHNIQUE FOR THE MEMORY REDUCTION

In the state-of-the-art SC decoder architecture, a large portion of the memory is consumed by M_C. In this paper, it is suggested to modify the decoding process so that M_C can be significantly reduced. This first approach does not alter the decoder functionality and consequently guarantees that the decoding performance remains unchanged. The idea is the following: when the second half of the channel LLRs is shifted in, instead of storing these incoming LLRs in M_C, the $f(a,b)$ function is directly computed. The first operand (a) is read from M_C while the second operand (b) directly comes from the channel buffer. This direct computation is possible because the $f(a,b)$ function is applied on operands with indices i and $i + N/2$: $f(C_i, C_{i+N/2}), 0 \leq i < N/2$. The results are then stored in M_I. An additional bit (Γ_i) has to be calculated and stored in M_I such that: $\Gamma_i = 0$ if $|f(a,b)| = |a|$, $\Gamma_i = 1$ otherwise. In the mean time, another function $h(a,b)$ is computed and stored in M_C such that $h(a,b) = b$ if $\Gamma_i = 0$, $h(a,b) = a$ otherwise. In other words, the bit Γ_i indicates the location of $|a|$ and $|b|$ in either M_C or M_I: $\Gamma_i = 0$ indicates that $|a|$ is stored in M_I and $|b|$ in M_C. $\Gamma_i = 1$ means the exact opposite. From the value stored in M_I and M_C one can recalculate a and b required for the g function:

$$a = k_a(C_i, L_i) = \begin{cases} C_i & \text{if } \Gamma_i = 1 \\ sign(C_i).L_i & \text{otherwise} \end{cases} \quad (5)$$

$$b = k_b(C_i, L_i) = \begin{cases} C_i & \text{if } \Gamma_i = 0 \\ sign(C_i).L_i & \text{otherwise} \end{cases} \quad (6)$$

The corresponding architecture is depicted in Figure 3. The decoding process is modified as follow: in **Phase** ϕ_0: $N/2$ channel LLRs (C_i) are shifted in one by one. The channel buffer stores P consecutive LLRs and then send them to M_C. A total of $N/2P$ packets of $P \times Q_C$ bits are sent to M_C. In **Phase** ϕ_1: the functions $f(C_i, C_{i+N/2})$, $h(C_i, C_{i+N/2})$ and the bit Γ_i are simultaneously computed for each operand couple $(C_i, C_{i+N/2})$. C_i are read from M_C while $C_{i+N/2}$ are directly shifted out by the channel buffer. f results are

stored in M_I while h results are memorized in M_C. In **Phase** ϕ_2: same as standard SC decoding. In **Phase** ϕ_3: operands a and b of g function are first recalculated, and then the g function is computed : $g(s, k_a(C_i, L_i), k_b(C_i, L_i))$. The results are stored in M_I. Finally, in **Phase** ϕ_4: same as standard SC decoding.

This architecture modification saves the memorization of $N/2$ channel LLRs. It represents an amount of $Q_C \times N/2$ bits. However, this method requires the memorization of an additional bit (Γ_i) for each of the $N/2$ computed values. The memory cost of this modified architecture is then $A_1 = N/2(Q_C + 2(Q_C + \Delta) + 3)$. The memory reduction relative the state-of-the-art architecture [5] is expressed as:

$$r_1 = 1 - \frac{A_1}{A_0} = \frac{Q_C - 1}{2 \times (2Q_C + \Delta + 1)} \tag{7}$$

One should notice that the memory reduction is independent of N and $r_1 > 0, \forall Q_C \geq 1$.

This memory reduction comes at the cost of some extra hardware resources for the computation of h, k_a and k_b. However, these functions can share some hardware with f anf g. Moreover, since $P << N$, this overhead is negligible in comparison with the memory complexity as confirmed by experimental results in section 6.

4. QUANTIZATION REDUCTION OF INTERNAL LLRS

SC decoding can be represented graphically by the traversal of a binary tree as shown in Figure 2. The phase ϕ_0 corresponds to the upper edge. Phases ϕ_1 and ϕ_3 are represented by left and right edges of the root node respectively. Phases ϕ_2 and ϕ_4 correspond to the recursive traversal of left and right sub-trees of the root node. In standard implementations, LLRs on the upper edge are quantized with Q_C bits while LLRs on the other edges are quantized with $Q_C + \Delta$ bits. A higher dynamic range of internal LLRs is necessary because the g function can actually generate LLRs with a larger dynamic range. One should however notice that the f function does not require any increase in the quantization since it consists in a minimum calculation. When the g function is applied on LLRs, the dynamic range has to be increased by 1 bit in order to avoid overflows. One can alternatively saturate the result at the cost of some decoding performance lost. In standard implementations, Δ extra bits are used for internal LLRs and a saturation is used to avoid overflows. If Δ is sufficiently large, the decoding performance loss can be found negligible [9].

In this second optimization, instead of using Δ extra bits on all tree levels, we propose to gradually increase the dynamic range across tree levels. The following notation is used to denote the proposed varying quantization scheme of the LLRs in the memory: $Q_C = x, Q_I = \{y, z, t\}$. It means that x bits are used for the channel LLRs (level 0 in the tree), y bits for the internal LLRs at level 1, z bits for level 2, and t bits for all the lower levels. In a standard architecture, the quantization scheme can be denoted as $Q_C = x, Q_I = \{x + \Delta\}$ because the same quantization is used for all internal LLRs. We propose to gradually increase the quantization on the first levels of the internal memory : $Q_C = x, Q_I = \{x + 1, x + 2, \ldots, x + \Delta\}$. This approach allows to reduce the memory footprint on the first memory stages which are the most expensive. In terms of memory gain, a reduction of q bits at level l in the tree saves $\frac{qN}{2^l}$ bits. For instance, removing 1 bit at level 1 saves $N/2$ bits. The memory footprint A_2 of this method can be calculated using equations 8 and 9. Equation 10 expresses the memory reduction in comparison with the standard architecture.

$$A_2 = N \times (Q_C + (Q_C + \Delta - \epsilon) + 1) \tag{8}$$

$$\epsilon = \sum_{l=1}^{\Delta-1} \frac{\Delta - l}{2^l} \tag{9}$$

$$r_2 = 1 - \frac{A_2}{A_0} \tag{10}$$

This method does not degrade the decoding performance in comparison with standard implementations (*i.e.* the one using $Q_C = x, Q_I = \{x + \Delta\}$) because 1 extra bit is added at each stage in order to avoid the saturation of LLRs. It is however possible to gain extra memory by further reducing the quantization with the following format: $Q_C = x, Q_I = \{x, x + 1, \ldots, x + \Delta\}$. This scheme induces a potential saturation of LLR at level 1. However, simulation results for several code lengths and code rates show that this extra reduction only slightly affect decoding performance as shown in Figure 4. The memory footprint of such an approach is:

$$A_2^* = N \times (Q_C + (Q_C + \Delta - \epsilon^*) + 1) \tag{11}$$

$$\epsilon^* = \sum_{l=1}^{\Delta} \frac{\Delta + 1 - l}{2^l} \tag{12}$$

$$r_2^* = 1 - \frac{A_2^*}{A_0} \tag{13}$$

5. MEMORY REDUCTION ESTIMATION

As shown in Figure 3, one can combine the two proposed memory reduction techniques. The memory footprint of such a decoder is $A_{(1,2)} = A_0 + (A_0 - A_1) + (A_0 - A_2)$. Given that $r_{(1,2)} = 1 - \frac{A_{(1,2)}}{A_0}$, one can show that $r_{(1,2)} = r_1 + r_2$. This is also true for A_2^*: $r_{(1,2)}^* = r_1 + r_2^*$

The memory reduction $r_{(1,2)}$ only depends on the quantization format (Q_C, Q_I) and not on the code size N. Let us suppose that both methods are applied. Assuming the format $Q_C = 6, Q_I = \{6, 7, 8\}$ is used, the memory reduction is

Table 1. Estimation of proposed techniques on previously proposed SC decoders

Parameters		Q_I	Related works			A_1		Q_I	A_2	$A_{1,2}$	Q_I	A_2^*		$A_{1,2}^*$	
n	Q_C		Ref	LUTs	FFs	A'_0 (bits)	Extra LUTs	r'_1		r'_2	$r'_{1,2}$		Extra LUTs	$r_2^{*\prime}$	Extra LUTs	$r_{1,2}^{*\prime}$
16	{5}	{5}	[9]	2866	1254	820992	+384	16%	{5, 5}	0%	0%	{5, 5}	+512	16%	+512	16%
17	{5}	{5}	[3]	221471	131764	1445632	+512	18%	{5, 5}	0%	0%	{5, 5}	+512	18%	+512	18%
17	{5}	{5}	[9]	2714	1263	1640192	+384	16%	{5, 5}	0%	0%	{5, 5}	+512	16%	+512	16%
20	{4}	{6}	[5]	5956	1366	13109248	+384	12%	{5, 6}	4%	10%	{4, 5, 6}	+384	16%	+384	22%
15	{4}	{7}	[5]	3927	1427	444672	+384	11%	{5, 6, 7}	9%	15%	{4, 5, 6, 7}	+384	20%	+384	26%
15	{4}	{8}	[5]	4141	1569	477696	+384	10%	{5, 6, 7, 8}	14%	21%	{4, 5, 6, 7, 8}	+384	25%	+384	31%
15	{4}	{9}	[5]	4673	1689	510720	+384	9%	{5, 6, 7, 8, 9}	19%	26%	{4, 5, 6, 7, 8, 9}	+384	29%	+384	35%
15	{5}	{7}	[5]	3731	1496	477440	+512	13%	{6, 7}	3%	9%	{5, 6, 7}	+512	17%	+512	22%
15	{5}	{8}	[9]	4161	1629	510464	+512	13%	{6, 7, 8}	8%	13%	{5, 6, 7, 8}	+512	20%	+512	26%
21	{5}	{9}	[9]	3571	2020	39853440	+512	10%	{6, 7, 8, 9}	11%	16%	{5, 6, 7, 8, 9}	+512	21%	+512	26%

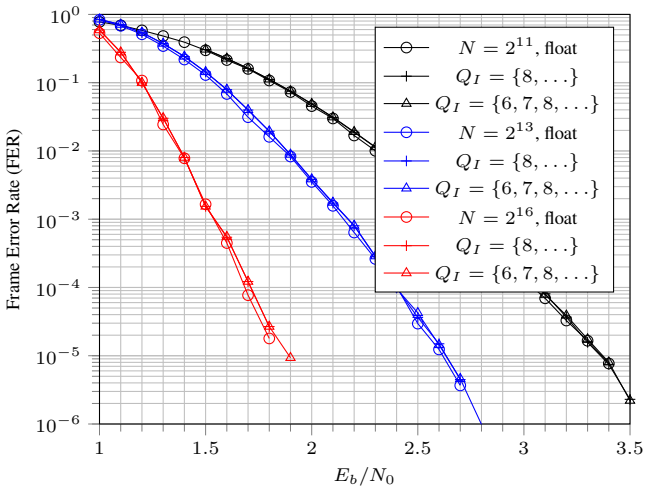

Fig. 4. FER curves for various N with $Q_C = 6$ bits.

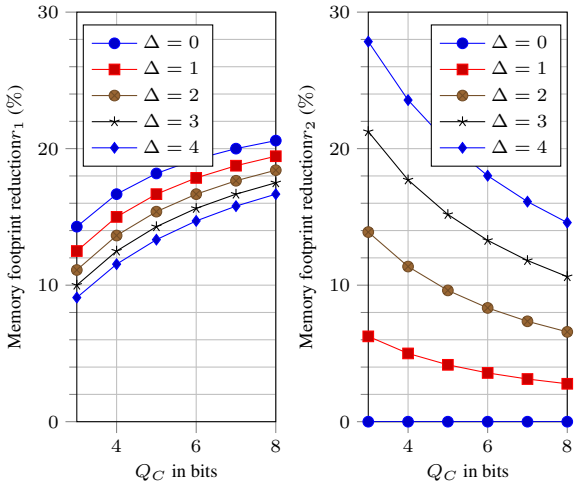

Fig. 5. Memory reductions r_1 and r_2

$r_{1,2} = 0.25$. Figure 4 shows the associated decoding FER for different N values and a code rate of $\frac{1}{2}$. No significant FER degradation can be observed (<0.02dB). The same trend was observed with the format $Q_C = 4, Q_I = \{4, 5, 6\}$. The associated memory reduction is also 25%. It is possible to further reduce the dynamic range on more tree levels, but this would degrade the decoding FER for a limited impact on the memory reduction. Indeed, lower levels in the tree require smaller memory space.

6. EXPERIMENTS

Table 1 shows estimations of the memory savings that the two techniques bring to state-of-the-art decoders. In order to have a fair estimation of the impact of the proposed techniques, the following approach was used: It is assumed that the reported numbers in [3] [5] [9] for each decoder represent the total memory footprint, that is A'_0. Since, we can compute A_0, the miscellaneous memory can then be calculated : $A(M_R) = A'_0 - A_0$. The reduced memory footprint is then computed and added to M_R in order to have a fair comparison. For instance, the reduction for the first reduction technique is $r'_1 = 1 - (A_1 + A'_0 - A_0)/A'_0$. The same approach was used to compute r'_2, $r'_{1,2}$, $r_2^{*'}$ and $r_{1,2}^{*'}$. LUT overheads were estimated thanks to logic synthesis of the h, k_a and k_b functions.

The combination of both techniques provides a memory reduction of 16% to 35% for a few hundreds of extra LUTs. The gain brought by the recomputation technique depends on the quantization format. The overhead however only depends on Q_C because recomputation are performed only on the first layer. The gain brought by the quantization reduction depends on the difference between Q_C and Q_I

7. CONCLUSION

In this paper, we proposed two memory reduction methods for the SC decoding of Polar Codes. Both methods can be combined and lead to a non-negligible memory reduction (16% to 35%). Future works include the hardware implementation of this decoder architecture. One could also apply these both memory reduction methods on the list decoding algorithm [7] and architectures [8] and the first one to software SC decoders [10, 11].

8. REFERENCES

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

[2] C. Leroux, I. Tal, A. Vardy, and W. Gross, "Hardware architectures for successive cancellation decoding of polar codes," in *Proceedings of the ICASSP Conference*, May 2011, pp. 1665–1668.

[3] C. Leroux, A. Raymond, G. Sarkis, and W. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Trans. on Signal Processing*, vol. 61, no. 2, pp. 289–299, 2012.

[4] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast polar decoders: Algorithm and implementation," *IEEE Journal on Selected Areas in Communications*, 2014.

[5] A. Raymond and W. Gross, "Scalable successive-cancellation hardware decoder for polar codes," in *Proceedings of the IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, December 2013, pp. 1282–1285.

[6] Y. Fan and C.-Y. Tsui, "An efficient partial-sum network architecture for semi-parallel polar codes decoder implementation," *IEEE Transactions on Signal Processing (TSP)*, vol. 62, no. 12, pp. 3165–3179, June 2014.

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

[8] A. Balatsoukas-Stimming, A. J. Raymod, W. J. Gross, and A. Burg, "Hardware architecture for list succesive cancellation decoding of polar codes," *IEEE Transactions on Circuits and Systems-II, Express Briefs*, vol. 61, no. 8, pp. 609–613, August 2014.

[9] A. Raymond and W. Gross, "A scalable successive-cancellation decoder for polar codes," *IEEE Transactions on Signal Processing (TSP)*, vol. 62, no. 20, pp. 5339–5347, October 2014.

[10] P. Giard, G. Sarkis, C. Thibeault, and W. Gross, "A fast software polar decoder," *arXiv:1306.6311*, Jun. 2013.

[11] B. Le Gal, C. Leroux, and C. Jego, "Multi-gb/s software decoding of polar codes," *Signal Processing, IEEE Transactions on*, vol. 63, no. 2, pp. 349–359, January 2015.