Hardware architectures for successive cancellation decoding of polar codes

Camille Leroux, Ido Tal, Alexander Vardy, Warren J. Gross

Institutions: McGill University, University of California, San Diego

Published on: 22 May 2011 - International Conference on Acoustics, Speech, and Signal Processing

Topics: Sequential decoding, List decoding, Block code, Polar code and Decoding methods

Related papers:

- Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels
- A Semi-Parallel Successive-Cancellation Decoder for Polar Codes
- List Decoding of Polar Codes
- A Simplified Successive-Cancellation Decoder for Polar Codes
- Fast Polar Decoders: Algorithm and Implementation

Share this paper: 🇺🇸 🇨🇳 🇦🇺 🇫🇷 🇨🇦 🇦᮶️ 🇳🇦

View more about this paper here: https://typeset.io/papers/hardware-architectures-for-successive-cancellation-decoding-22o7i8psa9
Hardware architectures for successive cancellation decoding of polar codes
Camille Leroux, Ido Tal, Alexander Vardy, Warren J. Gross

To cite this version:

HAL Id: hal-00670695
https://hal.archives-ouvertes.fr/hal-00670695
Submitted on 15 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
HARDWARE ARCHITECTURES FOR SUCCESSIVE CANCELLATION DECODING OF POLAR CODES

Camille Leroux\(^1\), Ido Tal\(^2\), Alexander Vardy\(^2\), Warren J. Gross\(^1\)

\(^1\)McGill University
Montréal, Québec, Canada

\(^2\)University of California San Diego
La Jolla, California, USA

ABSTRACT

The recently-discovered polar codes are widely seen as a major breakthrough in coding theory. These codes achieve the capacity of many important channels under successive cancellation decoding. Motivated by the rapid progress in the theory of polar codes, we propose a family of architectures for efficient hardware implementation of successive cancellation decoders. We show that such decoders can be implemented with \(O(n)\) processing elements and \(O(n)\) memory elements, while providing constant throughput. We also propose a technique for overlapping the decoding of several consecutive codewords, thereby achieving a significant speed-up factor. We furthermore show that successive cancellation decoding can be implemented in the logarithmic domain, thereby eliminating the multiplication and division operations and greatly reducing the complexity of each processing element.

Index Terms— Polar codes, successive cancellation decoding, hardware implementation, VLSI.

1. INTRODUCTION

Polar codes [1] are a family of error correcting codes with an explicit construction and efficient encoding and decoding algorithms. Moreover, they achieve capacity (asymptotically in the code length \(n\)) if the underlying channel is symmetric, memoryless, and has binary-input. To date, no other family of codes possesses these attributes, hence polar codes are seen as a major breakthrough in coding theory. Not surprisingly, polar codes have garnered much interest recently in the coding theory community. From a practical point of view, the capacity of the channel can be approached at the expense of a large code length \((n = 2^m)\) bits. In some information theoretic applications, polar codes are the only known solution which is both explicit and efficient: for example achieving the secrecy capacity of the wiretap channel in the general case [2].

Polar codes have recently been shown to have an efficient construction [3]. Recent results have started to address the issue of long code length. It was shown in [4] that applying belief propagation decoding on polar codes helps in reducing the required code length at the expense of extra complexity due to the iterative nature of belief propagation. Driven by the recent rapid progress in the theory of polar codes our motivation is to find efficient hardware architectures for SC decoding that will allow high-throughput and low-area implementations. Despite the numerous studies on polar codes construction and performance, the issue of hardware implementation of SC decoders remains an open problem. Initial results and a general framework for the implementation of belief propagation decoders for polar codes are given in [5]. However, due to its lower complexity compared to belief propagation, we are motivated to study the hardware implementation of successive cancellation decoding. Arıkan [1] showed that the SC decoding algorithm can be implemented in complexity \(O(n \log_2 n)\), where \(n\) is the code length.

In this paper, starting from the general framework proposed by Arıkan [1], we show that SC decoding can actually be implemented with hardware complexity \(O(n)\). We also propose to increase the throughput by decoding several consecutive vectors at the same time. Finally, in order to reduce the complexity further, we address the implementation of the computational nodes by working in the logarithmic domain, thereby eliminating the multiplication and division operations. We show that the resulting transcendental functions can be approximated by the minimum function with negligible performance degradation.

2. POLAR CODES

Polar codes are linear block error-correcting codes. Assume from here onward that the underlying channel has binary input, is symmetric, and is memoryless. Fix \(n = 2^m\) as the code length. Denote by \(u = (u_0, u_1, \ldots, u_{n-1})\) the input bits, and let \(c = (c_0, c_1, \ldots, c_{n-1})\) be the corresponding codeword. The encoding operation has an FFT structure, depicted in Figure 1, for \(m = 3\). Note that the ordering of the \(u_i\) in Figure 1 is according to the bit-reversal order: if we reverse the order of the bits in the binary representation of \(i\), then we get the standard lexicographic ordering.

Recall that \(u\) is encoded to \(c\). Next, \(c\) is sent over the underlying channel (the channel is used \(n\) times). Denote by \(y = (y_0, y_1, \ldots, y_{n-1})\) the corresponding channel output. We now wish to decode \(y\). This is done in terms of a successive cancellation decoder. That is, given \(y\), we first try to deduce the value of \(u_0\), then that of \(u_1\), and so forth up until \(u_{n-1}\). We do this as follows. Assume that we are currently at stage \(i\), and so, we have already guessed the values of \(u_0, u_1, \ldots, u_{i-1}\); denote these guesses as \(\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{i-1}\). Next, for \(b \in \{0, 1\}\), denote by \(Pr(y|\hat{u}_0^{i-1}, u_i = b)\) the probability that \(y\) was transmitted, given that \(u_0^{i-1} = \hat{u}_0^{i-1}\), that \(u_i = b\), and that \(u_{i+1}, u_{i+2}, \ldots, u_{n-1}\) are independent random variables with Bernoulli distribution of parameter \(\epsilon/2\). If \(i\) is not in the frozen set (explained later), then we take the guessed value \(\hat{u}_i\) to be

\[
\hat{u}_i = \begin{cases} 
0, & \text{if } Pr(y|\hat{u}_0^{i-1}, u_i = 0) > Pr(y|\hat{u}_0^{i-1}, u_i = 1) > 1, \\
1, & \text{otherwise}, 
\end{cases}
\]

(1)

Consider the case in which we are at stage \(i\), and \(u_0^{i-1} = \hat{u}_0^{i-1}\) — that is, we have guessed correctly up until now. Then, as shown in [1], for almost all \(0 \leq i < n\) we have that the probability of guessing \(u_i\) correctly is either extremely close to 1 (very good), or extremely

\(^1\)Note that \(n\) input bits are encoded to a length \(n\) codeword. However, as we will see later on, not all of the \(n\) input bits carry information.
tends to infinity. In order to keep the assumption shows the graph of the SC decoder for Fast Fourier Transform (FFT). In the remainder of the paper, we the factor graph of the code which has a structure resembling the Arıkan showed that SC decoding can be efficiently implemented by are not frozen (the effective code rate) tends to the capacity of the probability of guessing 1 close to 0 since they involve multiplications and divisions. In Section 4, we representation. In the alternative right-to-left scheduling, any node up- nature of this scheduling is especially suitable for software implementa- to be defined. Arıkan proposed two schedulings for this decoding is known. In order to respect the data dependence, a scheduling has to be black boxes until we return to them in Section 4. The values and are likelihood ratios while are the hardware complexity of a node processor and a register respectively. It can be shown that such a decoder with the right-to-left scheduling would take 2n – 2 clock cycles to decode n bits. The throughput in bits per second would then be where and are the hardware complexity of a node processor and a register respectively. It can be shown that such a decoder with the right-to-left scheduling would take 2n – 2 clock cycles to decode n bits. The throughput in bits per second would then be

\[ T = \frac{n}{2l_{np}} \approx \frac{1}{2l_{np}} \] (4)

where is the propagation time in seconds through a node processor. It follows that every node processor is actually used once every 2n – 2 clock cycles. This motivates us to find a schedule to merge some of the nodes into a single processing element.

3.2. Pipelined tree architecture
Looking further into the scheduling, we notice that whenever stage is activated, only 2 bits are actually updated. For example in Figure 2, when stage 0 is enabled, only one node is updated. Then the n nodes of stage 0 can be implemented using a single processing element (PE). We note that in general, for stage l, 2l processing elements (PEs) are sufficient to update the nodes. However, this resource sharing does not necessarily guarantee that the memories assigned to the merged nodes can also be merged. The memory sharing
the four variables generated in stage 2 at CC #1 are used on CC #2. The generated variable is used twice during the decoding. For example, the four data values generated by the stage activation during the decoding of one vector depends on the liveness of generated variables. Table 1 shows the architecture is shown in Figure 3 for (Table 1) the estimated bit generation during the wire routing phase of an integrated circuit design is much simpler in the tree architecture than in the FFT-like structure. Moreover, one can notice that the routing network in the decoder is possible to further reduce the number of PEs. Assuming that a PE can be reused at CC #8 to memorize function, the pipelined tree architecture while achieving the same throughput as computed in (4). It is possible to further reduce the number of PEs with only a small penalty in terms of throughput. Looking at Table 1, during the decoding of one vector, stage l is activated 2^{m-1} times. Consequently, in the line architecture of Figure 4, 5 stages are all activated at the same time only twice during the decoding of a vector, regardless of the code size. A decoder with only 5 PEs would require only 2 extra clock cycles to decode a vector. Such a semi-parallel architecture would improve the hardware efficiency at only a small decrease of throughput.

The Line SC architecture can be seen as a tree architecture in which complexity is reduced by merging some of the PEs. An alternative would be to start from the same tree architecture and use the idle stages to overlap the decoding of several codewords at once, enhancing the throughput.

3.3. Line SC Architecture

Despite the low complexity of the pipelined tree architecture, it is possible to further reduce the number of PEs. Looking at Table 1, it appears that only one stage is activated at a time. In the worst case (activation of stage m − 1), \( \frac{n}{2} \) PEs have to be activated at the same time. This means that the same throughput can be achieved with only \( \frac{n}{2} \) PEs. The resulting architecture is shown in Figure 4 for \( n = 8 \). The processing elements \( P_j \) are arranged in an line while registers keep a tree structure. Registers and PEs are connected via multiplexing resources that emulate the tree structure. For example, since \( P_{2,0} \) and \( P_{1,0} \) (in Figure 3) are merged to \( P_2 \) (in Figure 4), \( P_2 \) should write either to \( R_{2,0} \) or \( R_{1,0} \) while it should also be able to read from the channel registers or from \( R_{2,0} \) and \( R_{2,1} \). The \( \hat{u}_s \) computation block is moved out of \( P_j \) and kept close to the associated register because \( \hat{u}_s \) should also be forwarded to the PE. The overall complexity of the line SC architecture is

\[
C_T = (n-1)(C_y + C_{\hat{u}_s}) + nC_{\text{up}} + \left( \frac{n}{2} - 1 \right) 3C_{\text{max}} + nC_T
\]

where \( C_{\text{max}} \) represents the complexity of a 2-input multiplexer and \( C_{\hat{u}_s} \) is the complexity of the \( \hat{u}_s \) computation block. Despite the extra multiplexing logic required to route the data through the PE line, the savings in number of PEs makes this SC decoder less complex than the pipelined tree architecture while achieving the same throughput as computed in (4).

3.4. Vector-overlapping SC architecture

Let’s assume that we want to use idle cycles in the pipelined tree architecture in order to overlap the decoding of \( P \) vectors \( y \). At CC #1, \( y_1 \) is fed into stage 2 of the pipelined tree decoder. At CC #2, a second vector \( y_2 \) is shifted into stage 2 while \( y_1 \) uses stage 1. At CC #3, \( y_1 \) and \( y_2 \) are in stages 0 and 1 respectively. Then, a PE conflict occurs at CC #4 when both \( y_1 \) and \( y_2 \) need to access stage 0. This problem can be overcome by simply duplicating stage 0 so no resource conflict happens. As shown in Table 2, by duplicating

<table>
<thead>
<tr>
<th>CC</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_2 )</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S_1 )</td>
<td>f</td>
<td></td>
<td>g</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S_0 )</td>
<td>f</td>
<td>g</td>
<td>f</td>
<td>g</td>
<td>f</td>
<td>g</td>
<td>f</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( u_4 )</td>
<td></td>
<td>u_0</td>
<td>u_1</td>
<td>u_2</td>
<td>u_3</td>
<td>u_4</td>
<td>u_5</td>
<td>u_6</td>
<td>u_7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Schedule for the FFT-like and pipeline tree SC architectures (\( n = 8 \)).
stage 0 (denoted as $S_{0d}$), it is possible to overlap up to 3 vectors at the same time. It would actually be possible to insert another vector by using the remaining idle resources, but the routing of data across the tree would lose its nice regular property, making the multiplexing design more complex. Since several vectors are decoded at the same time, each PE should have access to registers associated with each vector. This means that in order to reach parallelism $P$, each stage $l$ should be duplicated $P_l + 1$ times. This vector-overlapping architecture allows us to reach a maximum parallelism value of $P = n - 1$. The complexity and the throughput of a vector-overlapping SC architecture with parallelism $P$ are

$$C_T = \left( n + \frac{P + 1}{2} \right) \left( \log \left( \frac{P + 1}{2} \right) + 1 \right) 2C_{\text{np}} + P(2n - 1)C_r,$$  

and

$$T = \frac{P}{2C_{\text{np}}}.$$  

This architecture provides a solution to enhance the parallelism of the decoder without duplicating all the resources of the decoder.

### 4. MINIMUM APPROXIMATION

SC decoding, in its original version, was proposed in the likelihood ratio domain in which the update rules $f$ and $g$ require multiplication and division. The hardware implementation of multipliers and dividers is very expensive and usually avoided in practical decoder designs. We propose to perform SC decoding in the log-domain in order to reduce the complexity of the $f$ and $g$ computation blocks. We assume that the channel information is available as the log-likelihood ratios (LLRs) $L_i$. In the LLR domain $f$ and $g$ become

$$f(L_a, L_b) = 2 \tanh^{-1} \left( \frac{\tanh \left( \frac{L_b}{2} \right) \tanh \left( \frac{L_a}{2} \right)}{2} \right),$$

$$g_u(L_a, L_b) = L_a(-1)^u_a + L_b,$$

where $L_a$ and $L_b$ are LLRs. In terms of hardware implementation, $g$ can be easily mapped to an adder/subtractor controlled by

$$f(L_a, L_b) \approx \text{sign}(L_a) \text{sign}(L_b) \min(|L_a|, |L_b|).$$  

In order to estimate the performance degradation incurred by this approximation we simulated the performance of different polar codes on an AWGN channel with BPSK modulation. There was no significant performance loss.

### 5. CONCLUSION

In this paper we showed that the architecture proposed by Arikan in [1] can be improved by taking advantage of the scheduling in SC decoding. Table 3 is a comparison of the complexity and throughput of the FFT-like SC decoder with the proposed architectures. The pipelined tree architecture and the line architecture allow us to reach the same throughput while reducing the hardware complexity. We also showed that throughput can be enhanced by decoding several vectors in parallel in a vector overlapping architecture. In this paper, we investigated fully-parallel architectures for SC decoders. For very large code lengths, it would be required to consider semi-parallel architectures in which PEs are shared within the update phase of the same stage as suggested in Section 3.3. The very regular structure of polar codes makes semi-parallel architectures straightforward to implement.

### 6. REFERENCES


