IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 24, 2020, accepted March 30, 2020, date of publication April 6, 2020, date of current version April 22, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2985909

An Area-Efficient Hybrid Polar Decoder With Pipelined Architecture

YU WANG, (Student Member, IEEE), QINGLIN WANG, YANG ZHANG, SHIKAI QIU, AND ZUOCHENG XING, (Member, IEEE)

National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China

Corresponding author: Yu Wang (wangyu16@nudt.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61804183.

ABSTRACT As the first kind of capacity-achieving forward error correction (FEC) codes, polar codes have attracted much research interest recently. Compared with traditional FEC codes, polar codes show better error correction performance when successive cancellation list (SCL) decoding with cyclic redundancy check is adopted. However, its serial decoding nature and high complexity of list management lead to its low throughput. Though the adaptive SCL decoding and hybrid decoding can improve the throughput, it comes at cost of implementation area. In this paper, we propose a pipelined hybrid decoding procedure and the corresponding hardware architecture to improve the area efficiency. In our design, the idle decoding cores are employed for successive cancellation (SC) decoding when SCL decoding is not working. The SCL decoding will be activated when the SC decoding fails. Different decoding cores work according to their own operation sequences and share one common processing array to improve the utilization ratio of processing elements. Constant receiving interval is supported with the design of input buffer to store all received codewords. A software platform is established to optimize the design parameters for each module of decoder. Moreover, the corresponding architecture is implemented using 65nm technology. Experimental results show that the proposed decoder can achieve a similar error correction performance with the SCL decoding with list size 16. Compared to the state-of-the-art available hybrid decoder, our proposed pipelined hybrid decoder is 3.07× more area efficient.

INDEX TERMS Polar decoder, hybrid decoding, pipelined architecture, area-efficient.

I. INTRODUCTION

Polar codes, proposed by Arikan [1], have drawn much research attention in the past ten years for their extraordinary error correction performance and regular decoding algorithm. Polar codes with successive cancellation (SC) decoding can theoretically achieve channel capacity for symmetric binary-input, discrete, memoryless channels (B-DMC) in the asymptotic sense. However, for short to moderate practical blocklengths, the performance of SC decoding is worse than that of Turbo codes [2] or low-density parity-check (LDPC) codes [3]. To improve the performance, SC list (SCL) decoding [4] is proposed by keeping L codeword candidates in the decoding procedure. Concatenated with cyclic redundancy check (CRC) [5], the error correction performance of SCL decoding can be further improved. It has been proved by the results shown in [5], [6] that with a large list size ($L \geq 16$), CRC-aided SCL (CA-SCL) decoding can outperform LDPC codes and Turbo codes, and hence polar codes with CA-SCL decoding has been adopted for the control channel in the 5G enhanced Mobile BroadBand (eMBB) scenario [7].

Although SCL decoding can significantly improve the error correction performance of polar codes, it has a low throughput and high area occupation when implemented in hardware. Its long decoding latency comes from two aspects, one of which is its inner sequential nature, while the other is the list pruning and sorting. Continuous efforts have been made to reduce the decoding latency. One approach is to prune the SC decoding tree by decoding multi-bit subcodes at the same time so that fewer list management (LM) operations are needed. Parallel decoded sub-codes can be either general codes comprised of consecutive $M = 2^m$ bits [8]–[10] or matching a special code pattern with variable length [11]–[14]. The first kind decodes M bits simultaneously, where M is a fixed and predefined value. In this method, the LM is executed at the leaf node of the decoding tree with M bits. The second kind method runs simplified LM algorithms for variable-length sub-codes. Especially, an extra

The associate editor coordinating the review of this manuscript and approving it for publication was Qinghua Guo.

structure needs to be designed to identify the code patterns online [13], [14]. As another method to reduce latency, the number of LM operations can be reduced by not splitting at the reliable bits [15]. Besides, the sorting method itself can be simplified to further reduce the latency of LM operation. In recent researches, two kinds of metric sorting methods are proposed to extract L paths with the smallest metrics out of $2L$ paths. For the first kind, the sorting problem is transposed into the calculation of the median threshold [16] or the double thresholds of the $2L$ metrics [17], while the second kind extract L unsorted paths with the smallest metrics first, and then sort the extracted metrics in parallel with other decoding operations [18]. However, with the increase of list size, the SCL decoding architecture still suffers from low throughput.

The experimental results shown in [19] prove that most of the codewords can be correctly decoded by the SCL algorithm with a small list size, so an adaptive SCL (A-SCL) decoding was proposed. In this algorithm, a codeword is first decoded by a single SC decoding. If the decoded codeword fails to pass CRC validation, the list size will continue to double until it reaches the predefined maximum list size L_{max}. The A-SCL decoding with L_{max} has an equivalent error correction performance as that of SCL decoding with $L = L_{max}$, while the average list size $\bar{L}$ is much smaller than the original one, i.e. $\bar{L} \ll L_{max}$. However, a directly-mapped A-SCL hardware architecture needs to support multiple SCL decoders with different list sizes, which inevitably leads to an increase in area occupation and is unpractical. Simplified A-SCL decoding was proposed in [20] with only one SC decoder and one SCL decoder with L_{max}. Whereas, its $\bar{L}$ is much large than the traditional A-SCL, which leads to the degradation of throughput gain. Considering the hardware implementation, a hardware-friendly two-staged adaptive SCL decoding algorithm is proposed in [21], which composes of one SCL decoder with small list size and one SCL decoder with large list size. The average list size $\bar{L}$ is reduced due to most codewords can be correctly decoded by the small list decoder. Besides, in recent research, an asymmetric adaptive SCL with several SC decoders and one SCL decoder was proposed in [22] to improve the utilization ratio of SCL decoder. In this hybrid decoder, the difference between an SC decoder and an SCL decoder in terms of computational complexity and workload is considered.

The first hardware implementation of SCL decoding was proposed in [23] by calculation log-likelihood (LL) values. Then in [24], the basic hardware architecture of SCL decoder using log-likelihood ratios (LLRs) was proposed to reduce the computational complexity and memory usage. Based on this design, the first fabricated hybrid polar decoder was presented in [25], in which one flexible decoder and one unrolled decoder are integrated. The flexible decoder is used to support SC, SC-flip (SCF) [26] and SCL decoding, while the unrolled decoder is used to improve the throughput. However, these two inner decoders are designed for different applications and there is no load balancing between two decoders. To support large list size and code length, another hybrid polar decoder was proposed in [27] with three inner decoders: one SC decoder for long code length, one SCL decoder with a flexible list size and one ultra-reliable SCL decoder with list size $L = 32$. Later, in [22], a load-balanced hybrid polar decoder was proposed, considering the difference between the SC decoder and the SCL decoder. However, in these designs, they only made several optimization techniques for internal LLRs message storage [28], [29], but not considered the optimization for processing elements (PEs). The PEs also occupy many areas in hardware implementation, and most of time their utilization ratio is very low.

In this work, we focus on improving the area efficiency of hybrid polar decoding by adopting a pipelined architecture, which is similar to our previous work [30]. Here, the main contributions of this work are summarized as follows:

- The decoding procedure of pipelined SC decoding and hybrid decoding. Since the SCL decoding is an SC-based decoding scheme, the idle decoding cores can be employed for SC decoding when SCL decoding is not working.
- A link-level simulation platform is established to select the design parameters of each module in the decoder.
- Each module in the decoder is redesigned to adapt pipelined decoding. An input buffer buffering all received codewords is designed to support the constant receiving interval. Different decoding cores share one common processing array to improve the utilization ratio of PEs.
- Experimental results show that our decoder achieves a similar error correction performance as an SCL decoder with list size $L = 16$ and that the hardware implementation is about $3.1\times$ more area-efficient than that of the state-of-the-art hybrid architecture [21].

The rest of this paper is organized as follows. In Section II, an overview of polar codes and SC decoding are presented, together with the SCL decoding. In Section III, the algorithm of pipelined SC decoding and hybrid decoding will be introduced. The hardware architecture of the pipelined hybrid decoder will be introduced in Section IV. Besides, the design and the selection of parameters of each decoding module will be described. The simulation and implementation results of the proposed pipelined hybrid decoder will be presented in Section V. Conclusions will be drawn in Section VI.

II. PRELIMINARIES

A. CONSTRUCTION OF POLAR CODES

Polar codes characterized by $(N, K, \mathcal{A})$ can achieve channel capacity via the phenomenon of channel polarization [1]. As the channel polarization theorem states, a completely polarized subchannel becomes either a noiseless channel or a pure noisy channel when the blocklength N goes to infinity, meaning the error probability of which is close to

0.5 or 0 respectively. Without loss of generality, we assume the code length $N = 2^n$ in this work, where n is an integer. By transmitting information bits over the noiseless subchannels and transmitting frozen bits which are known by both transmitter and receiver over the noisy subchannels, polar codes can achieve the channel capacity. Hence, constructing a polar code is equivalent to find the K most reliable subchannels over which the information bits are transmitted, and an information set $\mathcal{A}$ indicating these subchannel positions. The complement of $\mathcal{A}$ is defined as the frozen set $\mathcal{A}^c$, in which the bits are called the frozen bits and are always set to 0. Many construction methods [31]–[33] have been proposed to calculate the reliability of subchannels. In this paper, we adopt the construction method proposed in [33] to reduce the computational complexity of the relative reliabilities. Besides, as for CRC-aided polar codes, an r-bit CRC code is encoded using the last r information bits, which check the other $K - r$ information bits, and the effective code rate changes to $R = \frac{K-r}{N}$.

After the selection of information subchannels, the encoding process of a polar code can be represented with a matrix multiplication like

$$x_N = u_N G_N, \quad G_N = B\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{\otimes n}, \tag{1}$$

where vector u_N, holding the information bits and the frozen bits, denotes the source codeword to be encoded, while vector x_N denotes the encoded codeword. G_N is the generator matrix, and B is a bit-reversal permutation matrix, while $\otimes$ denotes the Kronecker product. More information about this encoding process can refer to [1].

B. SUCCESSIVE CANCELLATION DECODING

In successive cancellation decoding, we denote by y the data received from the channel detection and use them as the inputs of decoder. The outputs of SC decoder are denoted by vector $\hat{u}_1^N$, where $\hat{u}_i$ is the estimation of the bit u_i by hard decision. This hard decision is made according to its corresponding log likelihood ratio (LLR) $L_i = log(\frac{Pr(y,\hat{u}_1^{i-1})|u_i=0}{Pr(y,\hat{u}_1^{i-1})|u_i=1})$ and the function h:

$$\hat{u}_i = h(L_i) = \begin{cases} 0 & if\ i \in \mathcal{A}^c \\ \dfrac{1 - \mathrm{sgn}(L_i)}{2} & if\ i \in \mathcal{A}, \end{cases} \tag{2}$$

where $\mathrm{sgn}(L_i) = \pm 1$. For the SC decoding method, the i-th LLRs at different decoding stage l can be computed iteratively by following functions:

$$L_{l,i} = \begin{cases} f(L_{l+1,i}; L_{l+1,i+2^l}) & \text{if } \dfrac{i}{2^l} \text{ is even} \\ g(u_{s_{l,i-2^l}}; L_{l+1,i-2^l}; L_{l+1,i}) & \text{otherwise} \end{cases} \tag{3}$$

And in the LLR domain, the function f and g perform the following calculations by giving inputs LLRs L_a and L_b:

$$f(L_a, L_b) = \log(\frac{e^{L_a+L_b}+1}{e^{L_a}+e^{L_b}}) \tag{4}$$

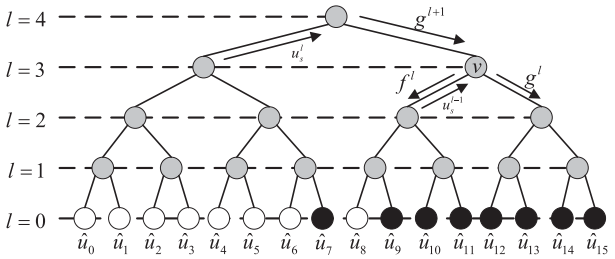


FIGURE 1. The successive cancellation decoding tree for polar code (16, 8).

$$g(L_a, L_b, u_s) = (-1)^{u_s} L_a + L_b \tag{5}$$

In the (3) and (5), the u_s denotes the partial sums of decoded bits $\hat{u}_1^{i-1}$, which are the bits that have been decoded previously. For g function at stage l, its partial sums u_s are obtained by

$$\left[u_{j+1}^l, \ldots, u_{j+2^l}^l\right] = \left[\hat{u}_{j-2^s+1}, \ldots, \hat{u}_j\right] \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{\otimes s} \tag{6}$$

Since the encoding procedure of u_s and its crucial role in the g function, the SC decoder has to decode serially. As depicted in Fig.1, the decoding process of SC decoding can be seen as a depth-first traversal of the decoding tree.

C. SUCCESSIVE CANCELLATION LIST DECODING

In order to improve the error correction performance, the SCL decoding employs L SC decoders in parallel, where L is a power of two. As shown in Fig.2, every time an information bit $\hat{u}_{I_i}$ needs to be estimated, instead of (2), the SCL decoding creates two paths corresponding to the decision $\hat{u}_{I_i} = 0$ and $\hat{u}_{I_i} = 1$ and the decoding paths are doubled. When the number of decoding paths comes to list size L, a list management (LM) operation is executed at each new bit to keep the number of survival paths to L. In order to measure the reliability of each path, a path metric (PM) is associated to each path and updated at every new estimation. This PM can be treated as a cost function, and the L paths with the lowest PM are allowed to survive. In the LLR-based formulation of SCL [24], the PM can be computed as

$$PM_{ip} = \sum_{j=0}^{i} \ln(1 + e^{-(1-2\hat{u}_{jp})L_{jp}}) \tag{7}$$

where p is the path index, $\hat{u}_{jp}$ is the estimate of bit j at path p, L_{jp} is the decision LLR. A hardware-friendly formulation of (7) can be treated as

$$\mathrm{PM}_{i_p} = \begin{cases} \mathrm{PM}_{i-1_p}, & \text{if } \hat{u}_{i_p} = \dfrac{1}{2}\left(1 - \mathrm{sgn}\left(L_{i_p}\right)\right) \\ \mathrm{PM}_{i-1_p} + \left|L_{i_p}\right|, & \text{otherwise} \end{cases} \tag{8}$$

$$= \frac{1}{2}\sum_{j=0}^{i} \mathrm{sgn}\left(L_{j_p}\right) L_{j_p} - \left(1 - 2\hat{u}_{j_p}\right) L_{j_p} \tag{9}$$

As for the frozen bit $\hat{u}_F$ in the decoding procedure, the PM of each path also needs to be updated and sorted to reduce

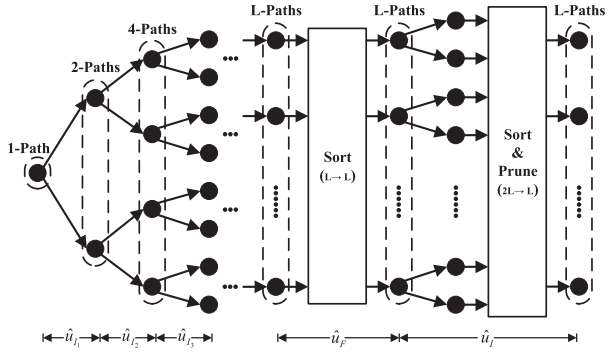


FIGURE 2. The decoding scheme of successive cancellation list decoding.

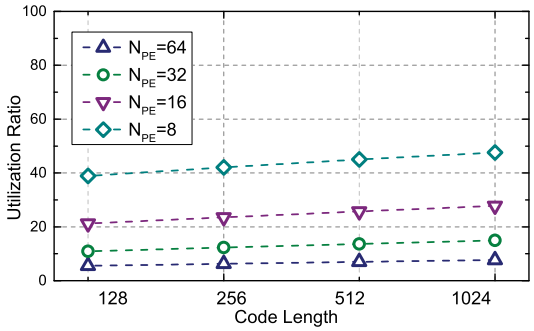


FIGURE 3. The utilization ratio of PEs for semi-parallel architecture with different code length.

the complexity of sorting at the next information bit, which is detailed in [24].

Eventually, the SCL decoder outputs a list of L codeword candidates. When a polar code is concatenated with a CRC, the CRC for each of the L candidates is checked after the decoding. The most reliable candidate out of all candidates that pass the CRC is selected as the decoded codeword. If all candidates fail the CRC, the path with the lowest PM is picked as the decoded codeword.

III. PIPELINED DECODING ALGORITHM

In current hybrid polar decoders, each inner SC decoding core in the SCL decoder has its own independent PEs. Though the semi-parallel architecture proposed in [34] can improve the utilization ratio of processing elements, the utilization ratio is still very low, especially when calculating the LLRs at the lower stages of the SC decoding tree. As shown in Fig.3, for SC decoding core with 64 PEs, the average utilization ratio is lower than 10%. The utilization ratio increases as the number of PEs decrease, while it inevitably leads to much increased decoding latency. In [35] and [36], an overlapped decoding approach is proposed to improve utilization. However, it needs the codewords received cycle by cycle, which is unpractical. In this work, we propose a pipelined decoding procedure for hybrid polar decoder, in which different decoding cores only hold its own internal LLRs and share one common processing array. In this section, we will first introduce the procedure of pipelined SC decoding, and then introduce the pipelined hybrid decoding.

Procedure 1 Pipelined SC With Multi-Core

procedure PIPELINED SC DECODER(y_1^N, $\mathcal{A}^c$)
  if $CouldWrite(p_{wr}) == 1$ **then**
    $InputBuffer(p_{wr}) = y_1^N$, $NeedDecode = 1$
  else Drop the y_1^N
  end if
  if $NeedDecode == 1 \&\& CorePrepared == 1$ **then**
    $ChannelSource(core_i) = p_{wr}$
    $ProcessFIFO(p_{wr}) = core_i$
    $CoreState(core_i) = 1$, $IssueState(core_i) = 1$
  end if
  for each $core_i$ in $ProcessFIFO$ **do**
    if $IssueState(core_i) == 1$ **then**
      Prepare LLR_{core_i}, PS_{core_i}, OP_{core_i}
      $IssueFIFO(p_{wr}) = core_i$
    end if
  end for
  for each $core_i$ in $IssueFIFO$ **do**
    Push($PreLLR_{buf}$, LLR_{core_i})
    Push(PS_{buf}, PS_{core_i})
    Push(OP_{buf}, OP_{core_i})
  end for
  LLR_{result} = Process($PreLLR_{buf}$, PS_{buf}, OP_{buf})
  Push($PostLLR_{buf}$, LLR_{result})
  for each $core_i$ in $IssueFIFO$ **do**
    if $Addr_{core_i} + 2^{stage_{core_i}} \leq N_{PE}$ **then**
      $IssueState(core_i) = 1$
    else $IssueState(core_i) = 0$
    end if
  end for
  for each $core_i$ in $ProcessFIFO$ **do**
    if $stage_{core_i} == 0$ **then**
      $\hat{u}_i = h(LLR_{core_i}, \mathcal{A}^c)$
      $psum_{core_i} = PartialSum(\hat{u}_0^i)$
    end if
    if $BitIndex_{core_i} \geq N/2$ **then**
      $CouldWrite(ChannelSource(core_i)) = 1$
    end if
    if $BitIndex_{core_i} == N$ **then**
      $Output = (\hat{u}_1^N)_{core_i}$
    end if
  end for
end procedure

A. PROCEDURE OF PIPELINED SC DECODING

The pipelined SC decoding procedure is performed based on the pipelined architecture, whose block diagram is shown in Fig.4. As a whole, it composes of six parts: Input Buffer, Core Management, Issue Management, Operation Generator, Processing Array, and List Management.

In pipelined SC decoding, the channel LLRs of each received codeword are first stored in the Input Buffer

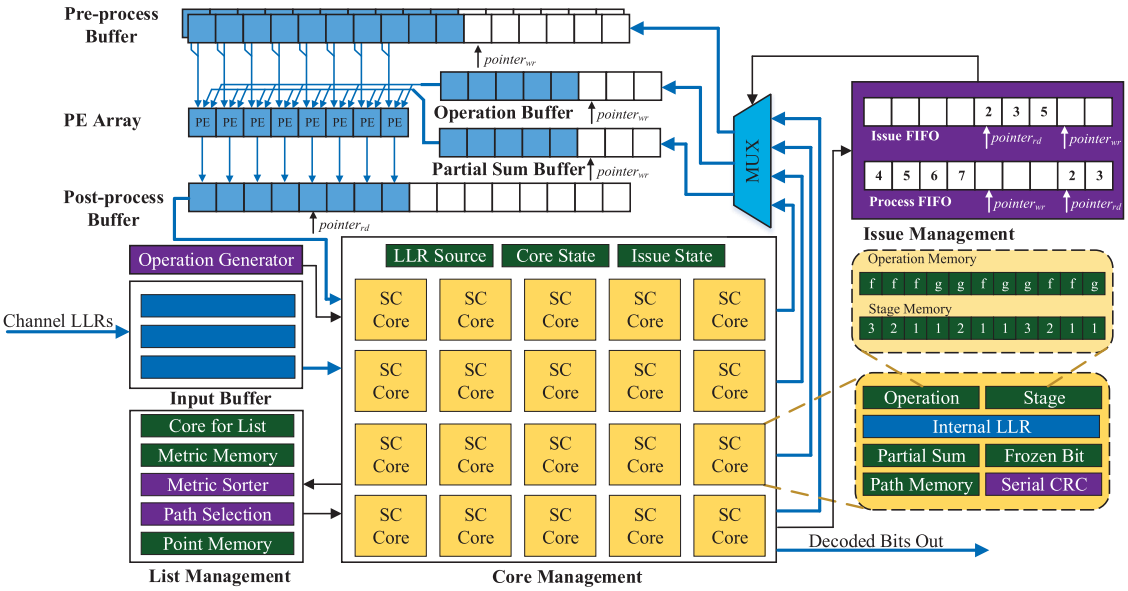


FIGURE 4. The pipelined architecture of hybrid polar decoder.

$InputBuffer(p_{wr}) = y_1^N$ when there is free space $CouldWrite(p_{wr}) == 1$. Then if there is an idle decoding core $CorePrepared == 1$ in Core Management, the address of input codeword will be dispatched to the core $ChannelSource(core_i) = p_{wr}$. When a decoding core enters the working state $CoreState(core_i) = 1$, it is pushed into the Process FIFO $ProcessFIFO(p_{wr}) = core_i$. The decoding core starts to decode the input codeword according to its operating sequences and read the channel LLRs from the corresponding Input Buffer. The decoding operation and the data scale to be processed for each cycle are stored in its own Operation Memory and Stage Memory. These operations are generated by the Operation Generator based on the code length and the index of first information bit. According to the Stage Memory, the decoding core reads the LLRs from the Input Buffer or its own Internal LLRs memory. Each core in the working state prepares its own unprocessed LLRs LLR_{core_i} and partial sum vector PS_{core_i} for corresponding decoding operation.

If the current operation of one decoding core can be issued $IssueState(core_i) == 1$, the core will be pushed into the Issue FIFO $IssueFIFO(p_{wr}) = core_i$. According to the Issue FIFO, the LLRs to be processed of different decoding cores are fetched to Pre-process Buffer Push($PreLLR_{buf}$, LLR_{core_i}). Correspondingly, the partial sum vector of each core are fetched to the Partial Sum Buffer Push(PS_{buf}, PS_{core_i}), when the operation is G function. As for the F function, the same scale of 0 are sent to the Partial Sum Buffer. The Processing Array with N_{PE} processing elements read the LLRs from Pre-process Buffer and Partial Sum Buffer, and execute the corresponding operation for each processing element. After the calculation, the processed results are sent to the Post-process Buffer Push($PostLLR_{buf}$, LLR_{result}). Each core in the Issue FIFO reads the results to extract its own results to Internal LLRs memory. When an operation is not completely executed in current cycle, the corresponding decoding core will not issue a new operation in the next cycle $IssueState(core_i) = 0$ until all LLRs of one operation have been processed. When the decoding comes to the stage of hard decision, the decision is made based on the frozen bit using (2). Then, the estimated bit will be stored in the Path Memory and used to calculate the partial sum.

After execution of each operation, the operation pointer plus one, and after one bit estimation, the bit pointer of frozen bit memory plus one. When the decoding core works in SC decoding mode, the corresponding Input Buffer no longer needs to store the LLRs when half of the codeword have been decoded $BitIndex_{core_i} \geq N/2$. Then the newly received codeword can be written to the buffer. The decoding of one codeword ends when all operations are executed $BitIndex_{core_i} == N$ and the corresponding decoded codeword is output from the path memory. The detail of pipelined SC decoding is shown in the Procedure.1.

B. PROCEDURE OF PIPELINED HYBRID DECODING

Based on the procedure of pipelined SC decoding, the List Management module is introduced to enable the SCL decoding. In pipelined hybrid decoding, the serial CRC unit is added to each decoding core to check the decoded bits. If the codeword decoded by SC decoding cannot pass the CRC $CRC(\hat{u}_0^N) ==$ fail, the SCL decoding is activated. L SC decoding cores, which are working in SCL mode, are united to perform SCL decoding, while the rest decoding cores in Core Management are still working in SC mode. The procedure of pipelined hybrid decoding is modified based on

Procedure 2 Pipelined Hybrid Decoding With Multi-Core

procedure PIPELINED HYBRID DECODER(y_1^N, $\mathcal{A}^c$)
  if CRC($\hat{u}_0^N$) == fail **then**
    $SCL\_decoding = 1$
  end if
  if ($CoreState(core_i) == 0$) **then**
    Push($CoreforList$, $core_i$)
    $ProcessFIFO(p_{wr}) = core_i$
    $CoreState(core_i) = 1$, $IssueState(core_i) = 1$
  else Hang up SCL decoding
  end if
  for each $core_i$ in $ProcessFIFO$ **do**
    if $IssueState(core_i) == 1$ **then**
      Prepare LLR_{core_i}, PS_{core_i}, OP_{core_i}
      $IssueFIFO(p_{wr}) = core_i$
    end if
  end for
  for each $core_i$ in $IssueFIFO$ **do**
    Push($PreLLR_{buf}$, LLR_{core_i})
    Push(PS_{buf}, PS_{core_i})
    Push(OP_{buf}, OP_{core_i})
  end for
  LLR_{result} = Process($PreLLR_{buf}$, PS_{buf}, OP_{buf})
  Push($PostLLR_{buf}$, LLR_{result})
  for each $core_i$ in $IssueFIFO$ **do**
    if $Addr_{core_i} + 2^{stage_{core_i}} \leq N_{PE}$ **then**
      $IssueState(core_i) = 1$
    else $IssueState(core_i) = 0$
    end if
  end for
  for each $core_i$ in $CoreforList$ **do**
    if $stage_{core_i} == 0$ **then**
      if $j \in \mathcal{A}^c$ **then**
        $\hat{u}_j = 0$, Update(PM_{core_i})
      else
        Do List Management
      end if
      $SerialCRC(\hat{u}_j)$
    end if
    if $BitIndex_{core_i} == N$ **then**
      $core_{out} = \underset{core_i \in PassCRC}{\arg\min} (PM_{core_i})$
      $Output = (\hat{u}_1^N)_{core_{out}}$
    end if
  end for
end procedure

the procedure of pipelined SC decoding, and the details of the changes are shown in Procedure.2.

When the SCL decoding starts, there is only one decoding core working for SCL decoding. When meeting an information bit that needs to be estimated, double cores are needed to continue decoding. If there are not enough idle cores for path duplicating, the SCL decoding will be hung up. When a decoding core is available, it is preferred as the decoding core of SCL decoding over SC decoding. This doubling procedure will continue until the number of decoding paths reaches L. Then the decoding procedure will be executed as the traditional SCL decoding. All cores working for SCL decoding will be added to the Core-For-List set for easier path management Push($CoreforList$, $core_i$).

The PM for each candidate decoding path is stored in the Metric Memory. At the decision stage, when the estimation is a frozen bit, the estimated bit is set to 0 and the PM is updated according to 9. As for the estimation of an information bit, the decoding procedure will enter the list management stage. The List Management module sorts and selects L paths with the lowest metric as the surviving paths. The details of list management stage will be shown in Section IV-E. Besides, to avoid the duplication of internal LLRs, the Pointer Memory is instantiated to hold the data source of each decoding path, as described in [4]. On the basis of SC decoding, the decoded bits $\hat{u}_j$ are also sent to the Serial CRC module to do CRC check. When the decoding is finished, the one with the smallest PM of all CRC verified results will be output as the final decoding codeword.

IV. HARDWARE ARCHITECTURE OF PIPELINED HYBRID DECODER

A. OVERALL ARCHITECTURE

An overview of the pipelined hybrid polar decoder is presented in Fig.4. This hybrid decoder supports SC decoding and SCL decoding simultaneously. Its error correction performance is guaranteed by SCL decoding, while the throughput is much improved by SC decoding. As shown in Fig.4, $N_{core} = 20$ decoding cores are instantiated, 16 of which can be used for SCL decoding. The received channel LLRs y_1^N are first stored in the Input Buffer. Then the codeword is dispatched to one idle decoding core to perform SC decoding. When the SC decoding finishes, the CRC is checked. If the decoded codeword cannot pass CRC, the SCL decoding is activated with list size $L = 16$.

In order to select the design parameters for each module of our decoder, a link-level simulation platform in MATLAB is established. In the simulations, a random bit stream of length K is generated with interval $T_{interval}$. Then the CRC bits are inserted and the frozen bits are set to zero according to the frozen pattern. After that, the pre-coded bits are fed into the polar encoder. Then, the encoded polar codes are modulated by binary phase-shift keying (BPSK) modulator and transmitted through the additive white Gaussian noise (AWGN) channel. After the transmission, the received codewords are sent to the pipelined hybrid decoder. In the simulation chain, the random codewords are generated with constant interval continuously. With different design parameters, the throughput and utilization ratio are analyzed to get a high throughput and area-efficient architecture.

B. INPUT BUFFER

In our decoder, we use an Input Buffer with depth D_{input} to store the received channel LLRs y_1^N for all decoding cores.

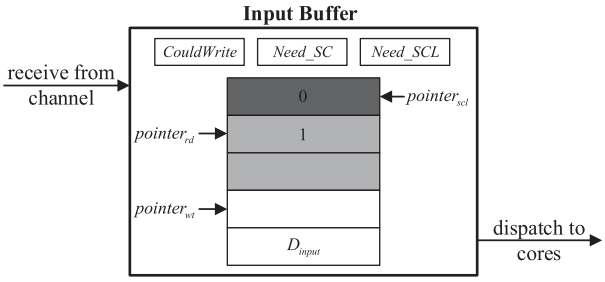


FIGURE 5. The schematic diagram of input buffer.

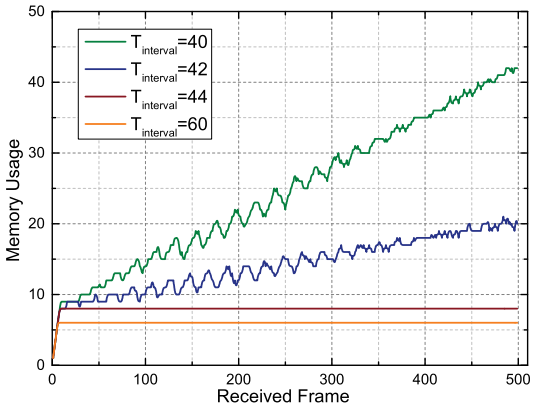


FIGURE 6. The memory usage of input buffer vs received codewords in SC mode.

The register *CouldWrite* is used to indicate whether the corresponding storage space is writable currently. If there is free space to store the received codeword, the LLRs y_1^N will be stored at the location indicated by the pointer $pointer_{wr}$ and the corresponding bit in register *CouldWrite* will be set to 0. Besides, the corresponding bit in register *Need_SC* will be set to 1 to indicate that a new codeword needs to be decoded. Then, the bit will be reset to 0 when there is an idle decoding core. Its storing address indicated by $pointer_{rd}$ will be dispatched to the core. In the decoding progress, the corresponding storage space will be released when half of the codeword have been decoded, and the bit $CouldWrite[core_i]$ will be reset to 1. This memory release strategy is performed when the decoding core is working in SC mode. In the SCL decoding mode, the corresponding storage space can only be released if the decoded codeword passes the CRC, otherwise it will continue to be stored and the bit in *Need_SCL* will be set to 1. In the Input Buffer, the pointer $pointer_{scl}$ indicates the codeword that currently needs to be decoded by SCL decoding. The maximum number of codewords that can be stored in the Input Buffer for SCL decoding is D_{list}. If there are more than D_{list} codewords need to be decoded by SCL decoding, the new failed codeword will be discarded. When half of the codeword have been decoded by SCL, the corresponding storage in Input Buffer will be released. The schematic diagram of Input Buffer is shown in Fig.5.

TABLE 1. The minimum sending interval $T_{interval}$ that can balance the codeword producer and consumer for different number of cores.

N_{core}	20	10	5	4
Minimum Interval	44	87	170	212
Maximum Memory	8	4	2	2
Average Latency	45.6	87.5	172.2	213.3

Since all received channel LLRs are first stored in Input Buffer, our design supports constant receiving interval like that does in [21]. In the design of Input Buffer, the memory depth D_{input} is a critical parameter, which is affected by the interval $T_{interval}$, the number of decoding cores N_{core}, and the maximum processing capability D_{list} of SCL decoding. Besides, the release strategy of different decoding modes also affects the reuse of memory space. In order to minimize the memory depth D_{input}, we need to know its maximum usage. The design of D_{input} is a classic Producer-Consumer problem. When data processing gets a dynamic balance with the data generating, the usage of memory will reach a steady state. For simplicity, we first simulate with all decoding cores working in SC mode. The bit stream generator generates the codeword with different interval $T_{interval}$. In the simulations, we instantiate 20 decoding cores and 256 PEs. The encoder generates 500 codewords of polar codes (512, 256) with interval $T_{interval}$ in one test. As shown in Fig.6, when the processing capability of cores is less than the supply of codewords, the memory usage increases with the number of received codewords. When the sending interval is $T_{interval} = 44$, a dynamic balance reaches between the codeword producer and consumer. As the sending interval increases further, the maximum usage of memory decreases.

The minimum sending interval $T_{interval}$ that can balance the codeword producer and consumer for the different number of cores are shown in Table 1 with corresponding maximum memory size. It could be observed that as the number of cores decreases, the minimum sending interval increases and the memory required to achieve dynamic balance decreases. Compared with traditional design, in which each decoding core has its individual channel LLRs memory, the design of Input Buffer module can reduce about 50∼60% memory space.

Then we make simulations in hybrid mode with different interval $T_{interval}$. In order to guarantee the excellent decoding performance, an SCL decoding with list size $L = 16$ is adopted in hybrid mode. As proposed in [22], an asymmetric hybrid decoder can balance the workload between SC decoder and SCL decoder. Hence, we instantiate more cores than the list size in our design so that the rest decoding cores can be used for SC decoding while SCL decoding. Considering the difference of workloads at the different channel conditions, we assume that the codewords are transmitted with the same condition $E_b/N_0 = 1.5dB$ in the simulations. We will introduce more details about the asymmetric deployment in Section IV-D. With the fixed PEs, the introduction of SCL decoding will inevitably lead to an increase in decoding

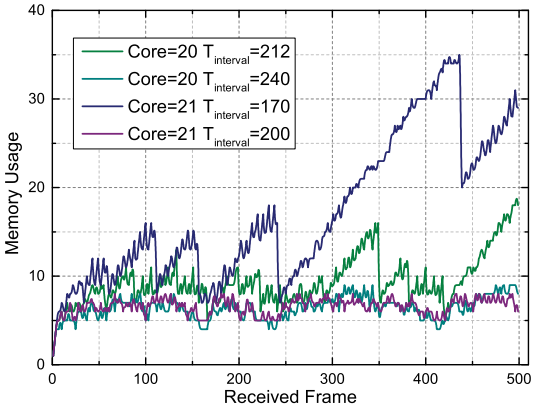


FIGURE 7. The memory usage of Input Buffer vs received codewords in hybrid mode.

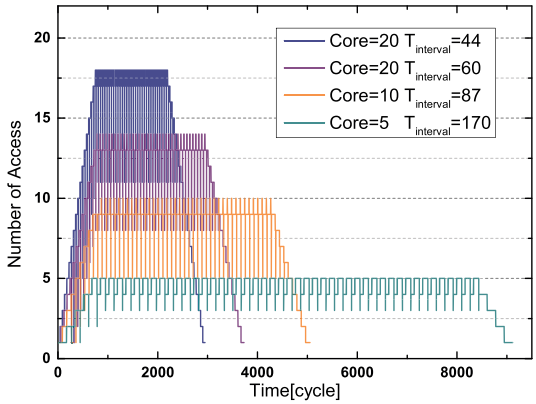


FIGURE 8. The number of accesses in the same cycle in decoding procedure with different number of cores.

latency and memory usage of Input Buffer. Besides, due to memory release strategy of SCL decoding, the balanced minimum memory usage grows further. As shown in Fig.7, the hybrid decoder with $N_{core} = 20$ decoding cores can reach decoding balance when the interval is $T_{interval} = 240$ which is much greater than that in SC mode. The maximum usage of memory increases from 8 to 10.

The impact of different deployments on Input Buffer is also shown in Fig.7. For the deployment with $N_{core} = 20$ decoding cores, one SCL($L = 16$) decoder with four SC decoders is instantiated. In compared deployment, one more decoding core is instantiated. It can be observed that with the increase of decoding core, the balanced interval decreases to $T_{interval} = 200$, while the maximum usage of Input Buffer decreases from 10 to 9.

C. OPERATION GENERATOR

In the pipelined decoder, the Operation Generator is responsible for generating the calculation operations of each decoding stage, which control the PEs to perform different calculations. Furthermore, the decoding stages, which indicate the scale of the internal LLRs to be calculated, are also generated by Operation Generator. The operations and stages are generated based on the code length and the index of the first information bit. In order to reduce the decoding latency, the decoding starts from the first information bit as [25] in our design. The state machine inside the Operation Generator generates the first several operations based on the binary representation of first information bit until its decision stage. After the decision of the first information bit, the state machine works in a depth-first path search mode like traditional SC decoding.

For codewords with the same frozen pattern, they have the same decoding schedule. Their decoding procedure can be controlled by the same set of operations and stages. However, multiple accesses to the same memory space in the same cycle are impractical for hardware implementation. The number of accesses to the operation memory at different cycles is shown in Fig.8. In these simulations, 50 codewords are transmitted with the constant interval $T_{interval}$. For decoders with 5 or

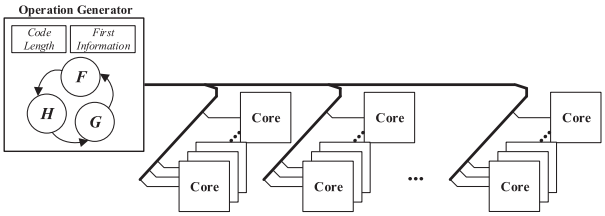


FIGURE 9. The diagram of operation generator.

10 cores, the maximum number of concurrent access is equivalent to the number of cores. As the number of decoding cores increasing, the number of concurrent accesses becomes less than the number of cores. This is due to the limited processing capability of decoding cores. In addition, the increase in sending interval also leads to a decrease in the number of concurrent accesses. Due to the concurrent access, we need to instantiate the operation memory and stage memory in each decoding core to realize immediate access to the operations and stages, which is shown in Fig.9.

D. CORE MANAGEMENT

For SC-based decoding algorithms, they have a similar decoding procedure. Therefore, when the SCL decoding is not working, the idle decoding cores in our hybrid decoder can be used for SC decoding to improve the throughput. In the Core Management module, N_{core} decoding cores are used to store the internal LLRs for one SC decoding codeword or one SCL decoding path. The register *Core_State* is used to indicate whether the decoding core is in working state. When a received codeword is dispatched to one idle decoding core, the register *LLR_Source* will record the source of channel LLRs.

Inside each core, the corresponding LLRs and partial sum are prepared according to current operation at each stage of decoding. The register *Issue_State* is used to determine whether a new operation should be issued. When a new operation is issued, the Issue Management reads LLRs from the corresponding core. At the decision stage, the hard deci-

TABLE 2. The average decoding latency of SC decoding and SCL decoding of polar code (512, 256) for different deployments.

N_{core}		17	18	19	20	21	22
Sending Interval		279	244	240	212	170	154
No Priority	Avg-SC	836	367	263	112	71	57
	Avg-SCL	857	860	863	861	860	862
	β	1.03	2.34	3.28	7.69	12.11	15.12
With Priority	Avg-SC	861	395	272	127	82	70
	Avg-SCL	847	847	847	847	847	849
	β	0.98	2.14	3.12	6.67	10.33	12.13

sion will be made by (2) inside each core. Similar to the Operation Memory, at each decision stage, there are multiple concurrent accesses to the Frozen Bit Memory, so the memory needs to be instantiated for each core. Then, the decision result is sent to the Path Memory and the Partial Sum Network (PSN) to calculate the partial sum. In SCL mode, the decision LLRs will be sent to the List Management to make list pruning and sorting. Moreover, the decision results of each path are sent to the Serial CRC module to make CRC check.

For the design of Core Management, the workload balancing of asymmetric deployment must be considered. The difference of decoding latency and error-correction performance between SC decoding and SCL decoding should be taken into account. However, different from the method proposed in [22], the decoding latency varies with different deployments, as the SC decoding and SCL decoding share the common PEs. That is to say, the speed gain β of SC decoding relative to SCL decoding is not a fixed value. The average decoding latency and speed gain of different deployments are shown in Table 2. In these simulations, the decoding latency is measured when SC decoding and SCL decoding are performed simultaneously. The codewords of polar code (512,256) transmitted with constant interval $T_{interval}$ are decoded by 256 PEs. It can be observed that the SCL decoding has a consistent decoding latency, while the SC decoding latency decreases with the increase of decoding cores. In order to reduce the decoding latency of SCL decoding, the cores for SCL decoding has the priority in the operation issue stage, that is, the LLRs of the corresponding core are read first. With this optimization, the speed gain decreases with the same deployment, which are also shown in Table 2.

As for the error correction performance, the performance of SC decoding at interest E_b/N_0 regime is shown in Fig.10. For polar codes with different code lengths at the worst channel condition $E_b/N_0 = 1.5dB$, they have similar error-correction performance, BLER = 0.25∼0.33. Considering that the performance will improve at the better channel conditions, the workload balancing can be achieved with the workload ratio 4 : 1 under the worst condition. Hence, considering the speed gain of decoding latency, the deployment $\eta = 2 : 1$ with 18 decoding cores is selected, where η represents the ratio of the number of SC decoders to the number of SCL decoders.

TABLE 3. The error probability of SC decoding during one SCL decoding and the drop probability due to memory overflow.

$Error$	0	1	2	3	4
$P(f_e)$	20.4%	39.8%	29.1%	9.5%	1.2%
$P(drop_u)$	79.6%	39.8%	10.6%	1.2%	0%

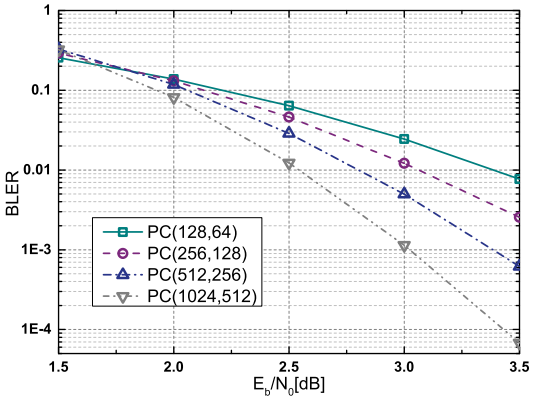


FIGURE 10. The BLER performance of SC decoding with different polar codes.

Based on the asymmetric deployment, we need to set rational upper bound of SCL processing capability D_{list}, in case that many SC decoders generate failed codewords at the same time. However, a large upper bound will result in an increase in the Input Buffer size. So we need to make a tradeoff between memory area and performance. The probability that the SC decoders have failed f_e codewords during one SCL decoding is calculated by

$$P(f_e) = \binom{f_t}{f_e} \times BLER^{f_e} \times (1 - BLER)^{f_t - f_e}, \qquad (10)$$

where $f_t = Core_{SC} * \beta$ is the total number of codewords decoded by the SC decoding cores during one SCL decoding. When there are more than D_{list} failed SC decoding codewords, the failed codewords will be dropped by the SCL decoding and output directly. The drop probability of SCL decoding with predefined D_{list} is calculated by

$$P(drop_u) = 1 - \sum_{e=0}^{u} P(f_e). \qquad (11)$$

The error probabilities of different number of failed codewords during one SCL decoding are shown in Table 3, with their drop probabilities. In the table, the BLER is obtained by transmitting polar codes (128,64) at $E_b/N_0 = 1.5dB$. According to the table, by setting the upper bound $D_{list} = 3$, 98.8% of failed codewords can be buffered for subsequent SCL decoding.

E. LIST MANAGEMENT

Different from the sorters proposed in recent researches [18], [37], [38], which are designed to reduce the sorting stage or area occupation, in our design, more attention should be paid to the release order of different decoding paths to reduce

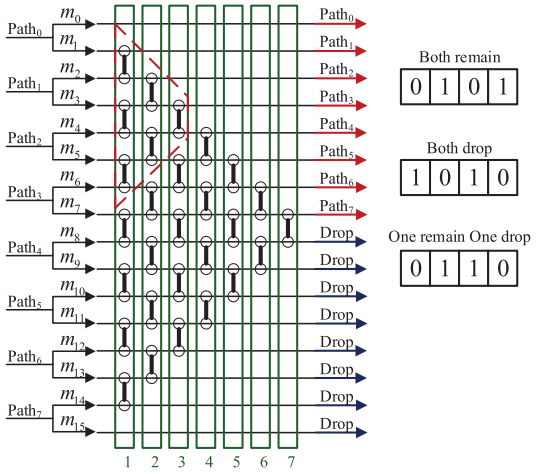


FIGURE 11. The simplified bubble sorter at the estimation of information bit with list size $L = 8$. Vertical lines represent comparisons, boxes represent stages.

the size of the Pre-process Buffer. In SCL mode, when the decoding process of one core reaches the decision stage, the issue of operations will be suspended. Then the sorter sorts the decoding paths according to their PMs and outputs the survival paths. The survival paths that have been ranked can continue to issue new operations. If multiple decoding paths are released simultaneously after passing through the sorter module, the following operation with the same data scale may lead to the overflow of Pre-process Buffer. Considering this, we adopt the simplified bubble sorter proposed in [37] as the path sorter in our design.

As shown in Fig.11, the paths to be sorted are input from the left side of the sorter, while the sorted paths are released at different sorting stages from the right side. Each vertical line represents a compare-and-select (CAS) unit that has two endpoints of the line as inputs. For each node in the structure, there are two registers to store the core index and PM, respectively. Besides, there is one state register for each node to indicate whether the current PM is ready for comparison. When the two endpoints of one vertical line are both ready, the comparison is executed, and the state register of the next stage is updated.

At the sorting stage of information bit, different paths enter the sorter module according to their previous ranking. The path with small PM enters the sorter from the upper horizontal line. The sub-paths of each path are input to two adjacent horizontal lines, where the upper one has a smaller PM than the lower one. At the other side of the sorter, the survival paths are released from the upper 8 lines, while other paths are dropped from lower 8 ones. A 4-bits register is added to each entering path to indicate whether its two sub-paths are survival. When both sub-paths are survival, the operation issuing of the lower path will be suspended until one sorted path, whose both sub-paths of that path are dropped, appears. Then the decoding state of path with large PM is copied to the dropped path for subsequent decoding.

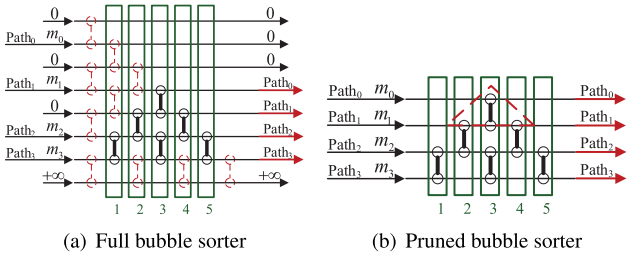


FIGURE 12. The full bubble sorter at the estimation of frozen bit with list size $L = 4$.

As for the sorting of frozen bit, the sorter is a little bit different from the simplified one, since all the paths need to be ranked. By using the method proposed in [24], we can use a triangular structure to realize full sort of all decoding paths, as shown in Fig.12a. In this structure, the CAS units in red dotted lines can be removed due to the relations of two endpoints are already known. Then we get the structure shown in Fig.12b. By using this structure, all decoding paths at the decision stage of the frozen bit can be ranked and released at different sorting stages. In this sorter, the path that arrives first enters from the upper horizontal line.

When the SCL decoder is in the path expansion state, the working cores for SCL decoding are less than the list size L. At this state, the path sorting can be finished by using only a portion of the entire sorter, which are circled by the red dotted trapezoid in Fig.11 and the red dotted triangle in Fig.12b.

Since the sorted paths are released at different stages, the decoding process of different paths are not synchronous. The path with a small PM can issue more operations before other paths are released, which leads to the path go into the sorting stage of the next bit. Hence, in our design, we instantiate two sets of sorter modules. One set is used for the current decoding bit, while another one is used for the next bit. When all paths are released by current sorter, another sorter becomes the current one and the current sorter is reset for the sorting of next bit, as shown in Fig.13. Especially when the latter bit in the consecutive two bits is an information bit, the operation issuing of the released path will be suspended in the next sorter until the previous sorter releases all decoding paths.

F. ISSUE FIFO AND PRE-PROCESS BUFFER

Since different decoding cores share the common processing array in our design, a Pre-process buffer is needed to collect the LLRs. At each cycle, the working core determines whether issue a new operation, according to the state of register *Issue_State*. When the corresponding bit is $Issue\_State[Core_i] = 1$, the core index will be pushed into the Issue FIFO, and the unprocessed LLRs will be written into the Pre-process Buffer. For the Pre-process Buffer, the write address $Addr_i$ of each core is calculated based on the *Issue_State* and the data scale of the previous cores. In addition, when there are unprocessed LLRs of previous cycle remained in the Pre-process Buffer, an offset address $Addr_{bias}$

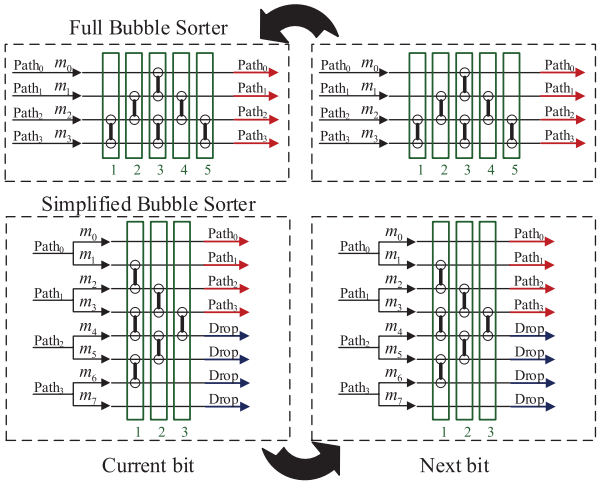


FIGURE 13. The sorters of adjacent two bits for SCL decoding with list size $L = 4$.

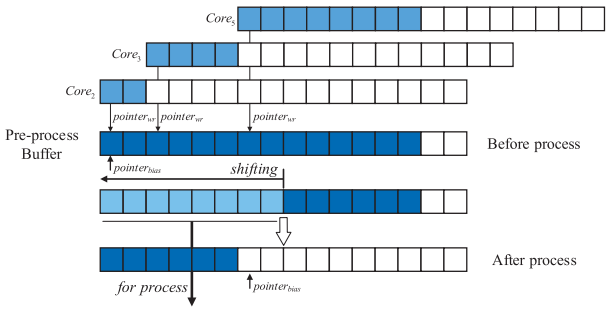


FIGURE 14. The writing scheme of Pre-process Buffer.

is added to obtain the final write address. Based on this, the write address of each issuing core can be calculated by

$$Addr_i = \sum_{c=0}^{i-1} (Issue\_State[c] \&\& (1 \ll stage_c)) + p_{bias}, \quad (12)$$

where $1 \ll stage_c$ represents the scale of LLRs to be processed. As shown in Fig.14, the unprocessed LLRs are written into the Pre-process Buffer based on their respective write address $Addr_i$. The front N_{PE} LLRs in the buffer are fetched out for calculation. After the calculation, the LLRs that cannot be processed in the current cycle will be shifted to the front-end of the buffer and processed in the next cycle. At the same time, the offset pointer $Addr_{bias}$ will move to the end of unprocessed LLRs.

To avoid buffer overflow, the length of Pre-process Buffer L_{pre} must be large enough to cache all unprocessed LLRs in one cycle. In the design of Pre-process Buffer, we make simulations to obtain the maximum usage of buffer in the hybrid decoding. 100 codewords of polar codes (512,256) are transmitted with different sending interval $T_{interval}$ at $E_b/N_0 = 2.5dB$, and there are $N_{core} = 18$ decoding cores in the decoder. In the simulations, the length of Pre-process Buffer L_{pre} is large enough to cache all LLRs to be processed at one cycle and the maximum usage of the buffer is recorded.

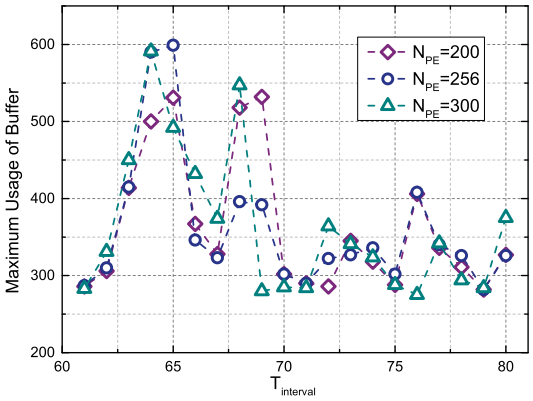


FIGURE 15. The maximum usage of pre-process buffer with different number of PEs.

As shown in Fig.15, with the increasing of sending interval, the maximum usage of buffer decreases. However, at some sending interval points, there are still surges of maximum usage.

In addition to the sending interval, the number of PEs N_{PE} also affects the design of Pre-process Buffer. The number N_{PE} represents the processing capability of the decoder. By instantiating different numbers of PEs, the maximum usage of the buffer varies. However, as depicted in Fig.15, the maximum usages of different N_{PE} have the similar overall trend. Hence, in our design, N_{PE} is set equally to the half of code length $N/2$ to meet the maximum processing requirement of one decoding core in one cycle. Besides, it can be observed from Fig.15 that for polar codes (512,256) by adopting the length $L_{pre} = 2.5 * (N/2) = 640$, the buffer can meet the maximum usage with different sending interval.

G. POST-PROCESS BUFFER

The calculated results of PEs are cached in the Post-process Buffer. Then the LLRs will be broadcast to every core, as shown in Fig.16. Each working core reads the respective results according to the same address $Addr_i$ calculated in Issue FIFO and checks the integrity of calculated LLRs. For each issuing core, the integrity is checked by whether $Addr_i + (1 \ll stage_c)$ is smaller than N_{PE}. When only a portion of the prepared LLRs are calculated in the current cycle, the decoding progress of the corresponding core will be hung up and wait for the rest results. In the next cycle, the rest of the calculated LLRs of this core will be read first and combined with the previous LLRs. In the decoding, only the cores that have all prepared LLRs be processed can be removed from Issue FIFO. Besides, when the reading address $Addr_i$ is large than N_{PE}, it can be inferred that the LLRs to be processed have not been calculated in current cycle. Therefore, the corresponding core should not issue new operation in the next cycle either. Moreover, in each decoding core, the calculated LLRs will be shifted to the front-end of the buffer before they are finally stored into the Internal LLR memory for facilitating the reading in the subsequent decoding.

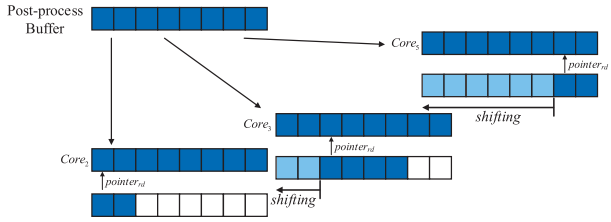


FIGURE 16. The reading scheme of post-process buffer.

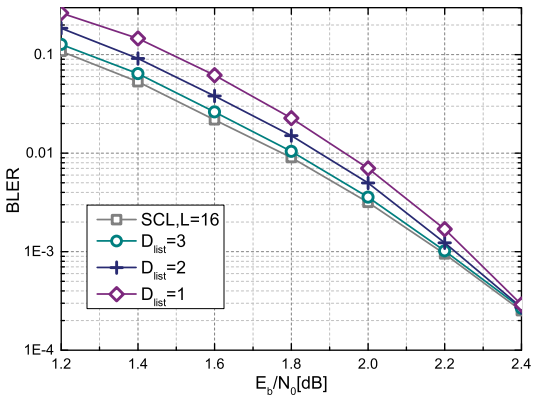
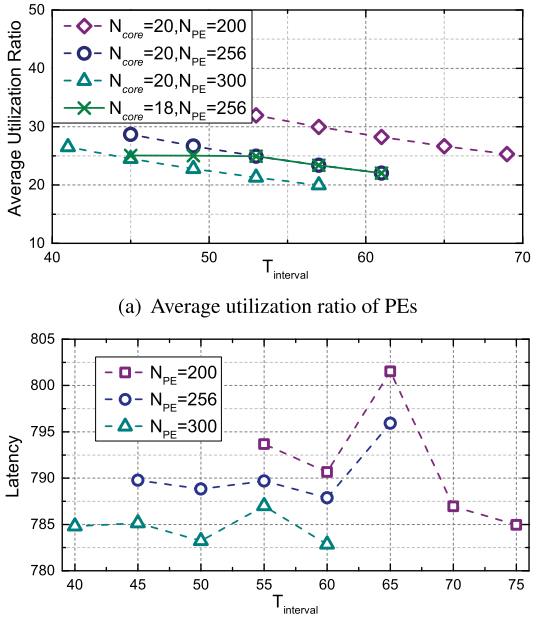


FIGURE 17. BLER performance of our decoder with different SCL decoding capability D_{list}.



(a) Average utilization ratio of PEs

(b) Average decoding latency of one codeword

FIGURE 18. The utilization ratio and decoding latency of hybrid polar decoder with different numbers of PEs and sending interval.

V. EXPERIMENTAL RESULTS

A. ERROR CORRECTION PERFORMANCE OF THE PROPOSED DECODING PROCEDURE

To evaluate the error correction performance of the proposed hybrid polar decoder, we make simulations with polar codes (512,256). In these simulations, we instantiate $N_{core} = 18$ decoding cores in the decoder, 16 of which can be used for SCL decoding. When the decoder gets a dynamic balance with the codewords generating, the performance of our decoder mainly depends on the upper bound of the SCL processing capability D_{list}. The larger D_{list}, the more error codewords of SC decoding will be cached for later SCL decoding, which will lead to the improvement of performance at the cost of throughput degradation. As shown in Fig.17, with the increase of D_{list}, the error correction performance is improved at a low SNR regime. At the high SNR regime, our proposed decoder has the same performance as SCL decoder with $L = 16$, even with a small D_{list}. Therefore, for some applications that only need to work at a high SNR regime, we can use $D_{list} = 1$ to reduce the size of Input Buffer and improve the throughput. In addition, due to the decrease of BLER of SC decoding at a high SNR regime, the ratio η can be increased, i.e. $N_{core} = 20$, to further improve the throughput.

B. DECODING LATENCY AND UTILIZATION RATIO OF PEs

In the current design of hybrid polar decoder [21], [22], most of the time the utilization ratio of PEs is very low. Hence, in our design, different decoding cores share one common processing array to improve the utilization ratio. This design will inevitably lead to the increase of decoding latency for one single codeword. However, by adopting the pipelined architecture, the overall throughput and area efficiency of the hybrid polar decoder can be much improved.

The utilization ratio of PEs depends on the number of codewords decoded simultaneously, the sending interval $T_{interval}$ and the number of PEs N_{PE}. With the same sending interval, the more decoding cores, the more codewords are processed by the PEs simultaneously. The increase of processed codewords leads to the increase of utilization ratio. As shown in Fig.18a, the decoder with $N_{core} = 20$ has higher utilization ratio than that with $N_{core} = 18$ at $T_{interval} = 45, 49$. However, when the receiving interval increases further, the utilization ratio of these two deployments becomes equivalent, since the numbers of working cores become the same. Besides, this is the reason why the utilization ratio decreases as the receiving interval increases for a given deployment. It can be observed from Fig.18a that the utilization ratio is also affected by the number of PEs N_{PE}. With a certain receiving interval, the smaller the number N_{PE}, the higher the utilization ratio. However, with the same number of working cores, the reduction in the number of PEs means that large Pre-process Buffer is needed to cache the unprocessed LLRs, and the decoding latency will increase accordingly.

The comparison of decoding latency with different number of PEs is shown in Fig.18b. With the same workload, the decoder with fewer cores has higher decoding latency. In our design, the multibit decision decoding [8] is not adopted, which makes the decoding latency of single codeword longer. However, this does not affect the throughput of our pipelined decoder, which is determined by the receiving interval of codewords. Though decoders with different numbers of PEs

TABLE 4. Implementation results for proposed hybrid polar decoder against equivalent state-of-the-art hybrid decoder or flexible decoder with polar codes(1024, 512).

Implementation	This work ($N_{core}=18$)	This work ($N_{core}=20$)	Hybrid decoder [21]	Flexible decoder [27]	Flexible decoder [25]
Algorithm	Hybrid(L=16)	Hybrid(L=16)	SCL(L=2,32)	SCL(L=8)	SCL(L=4)
Technology	65nm	65nm	90nm	16nm	28nm
Clock freq. (MHz)	476	454	465	1000	308
Throughput (Mbps)	1383$^{\diamond}$	3837†	2346	3241	130.9
Total area (mm^2)	4.06	4.41	22.00	2.27	0.44
Area efficiency(Mbps/mm^2)	341	870	106	1428	295
Normalized for 65nm*					
Clock freq. (MHz)	476	454	644	246	132
Throughput (Mbps)	1383	3837	3248	797	56
Total area (mm^2)	4.06	4.41	11.47	37.46	2.37
Area efficiency(Mbps/mm^2)	341	870	283	21	23

* Area scaled as λ^2 and frequency as $1/\lambda$, where λ is the technology feature size.
$^{\diamond}$ The throughput is measured at $E_b/N_0 = 1.5dB$.
† The throughput is measured at $E_b/N_0 = 2.5dB$.

have the same throughput when the workload is low, they have a different upper bound of throughput. When the workload increases, the throughput of decoders with more PEs will increases further, while the overflow of Input Buffer will incur in the decoders with fewer PEs.

C. IMPLEMENTATION RESULTS OF THE PROPOSED ARCHITECTURE

To compare with the implementations of other hybrid decoders, the proposed pipelined architecture with different deployments is implemented for polar codes with $(N, K, r) = (1024, 512, 24)$. They are synthesized using a 65nm technology node. For a fair comparison, the same quantization schemes in [21], i.e. $Q_i = 6, Q_c = 5, Q_{PM} = 8$, are used in our implementation. The reported throughputs are measured by coded bits and the reported total area composes of both cell and net area.

The synthesis results of our hybrid polar decoders with two different deployments, i.e. $N_{core} = 18$ and $N_{core} = 20$, are shown in Table 4. For decoder with 18 cores, it is designed for working at a low SNR regime with $D_{input} = 8$ and $D_{list} = 3$ to achieve high error correction performance. As for the decoder with 20 cores, it is implemented with $D_{input} = 12$ and $D_{list} = 1$ to improve the throughput for the applications that only need to work at a high SNR regime. They both have 512 PEs and adopt the design proposed in [39] to improve the hardware efficiency. Compared with the decoder $N_{core} = 18$, the critical path delay of the decoder with $N_{core} = 20$ is longer, since its issue management needs more operations to prepare the unprocessed LLRs. As for the area occupation, the increased area comes mainly from the decoding cores and the Input Buffer. Since more cores can work simultaneously, the decoder with $N_{core} = 20$ has higher throughput, which results in a higher area efficiency when the area of other modules does not increase much.

In current hardware implementations, there are only two implementations of hybrid decoder [21], [22]. However, the design in [22] implements on FPGA platform, so we only compare the ASIC implementation results of the design proposed in [21]. Different from the designs in [21], [22], our hybrid decoder uses the idle decoding cores for SC decoding when there is no codeword need SCL decoding, which leads to a higher throughput with deployment $N_{core} = 20$. In the design [21], there are one SCL decoder with list $L = 32$ and one with list $L = 2$, which is equally 34 decoding cores. For fair comparison, we only compare the area efficiency. As shown in Table 4, the proposed decoder with $N_{core} = 18$ cores is 1.2× more area efficiency, while the decoder with $N_{core} = 20$ is up to 3.1× more area efficiency when compared to the design in [21]. In addition, the flexible decoder proposed in [25], [27] can also perform both SC and SCL decoding in different application scenarios. Therefore, they can also be regarded as multi-core hybrid decoders. Compared with these two flexible decoders, our decoder with $N_{core} = 20$ is up to 41.4× and 37.8× more area efficiency.

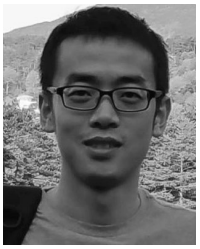
VI. CONCLUSION

In this work, a pipelined hybrid decoding procedure with corresponding hardware architecture is proposed. In our decoder, the SC decoding and the SCL decoding use the common decoding cores to improve the utilization ratio. Constant receiving interval is supported by adopting the design of Input Buffer. A link-level simulation platform is established to optimize the parameters for each module in the decoder. Each module in the decoder is redesigned to adapt pipelined decoding and to improve hardware efficiency. Experimental results show that our proposed pipelined hybrid decoder is at least 3.1× more area-efficient than the existing hybrid decoder and flexible decoder.

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, Jul. 2009.
[2] C. Berrou, "Near-Shannon limit error-correcting coding and decoding : Turbo codes," in *Proc. Int. Conf. Commun.*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[3] R. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 3–26, 2008.
[4] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Trans. Inf. Theory*, vol. 61, no. 5, pp. 2213–2226, Jun. 2012.
[5] K. Niu and K. Chen, "CRC-aided decoding of polar codes," *IEEE Commun. Lett.*, vol. 16, no. 10, pp. 1668–1671, Oct. 2012.
[6] K. Niu, K. Chen, and J.-R. Lin, "Beyond turbo codes: Rate-compatible punctured polar codes," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2013, pp. 3423–3427.
[7] *Final Report of 3GPP TSG RAN WG1 #87 v1.0.0*, 3rd Generation Partnership Project (3GPP), Reno, NV, USA, Nov. 2016.
[8] B. Yuan and K. K. Parhi, "Low-latency successive-cancellation list decoders for polar codes with multibit decision," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 10, pp. 2268–2280, Oct. 2015.
[9] C. Xiong, J. Lin, and Z. Yan, "Symbol-decision successive cancellation list decoder for polar codes," *IEEE Trans. Signal Process.*, vol. 64, no. 3, pp. 675–687, Feb. 2016.
[10] C. Xia, J. Chen, Y. Fan, C.-Y. Tsui, J. Jin, H. Shen, and B. Li, "A high-throughput architecture of list successive cancellation polar codes decoder with large list size," *IEEE Trans. Signal Process.*, vol. 66, no. 14, pp. 3859–3874, Jul. 2018.
[11] 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.
[12] S. A. Hashemi, C. Condo, and W. J. Gross, "Fast and flexible successive-cancellation list decoders for polar codes," *IEEE Trans. Signal Process.*, vol. 65, no. 21, pp. 5756–5769, Nov. 2017.
[13] S. A. Hashemi, C. Condo, M. Mondelli, and W. J. Gross, "Rate-flexible fast polar decoders," *IEEE Trans. Signal Process.*, vol. 67, no. 22, pp. 5689–5701, Nov. 2019.
[14] F. Ercan, T. Tonnellier, and W. J. Gross, "Energy-efficient hardware architectures for fast polar decoders," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 1, pp. 322–335, Jan. 2020.
[15] Z. Zhang, L. Zhang, X. Wang, C. Zhong, and H. V. Poor, "A split-reduced successive cancellation list decoder for polar codes," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 2, pp. 292–302, Feb. 2016.
[16] J. Lin and Z. Yan, "An efficient list decoder architecture for polar codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 11, pp. 2508–2518, Nov. 2015.
[17] Y. Fan, J. Chen, C. Xia, C.-Y. Tsui, J. Jin, H. Shen, and B. Li, "Low-latency list decoding of polar codes with double thresholding," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 1042–1046.
[18] V. Bioglio, F. Gabry, L. Godard, and I. Land, "Two-step metric sorting for parallel successive cancellation list decoding of polar codes," *IEEE Commun. Lett.*, vol. 21, no. 3, pp. 456–459, Mar. 2017.
[19] B. Li, H. Shen, and D. Tse, "An adaptive successive cancellation list decoder for polar codes with cyclic redundancy check," *IEEE Commun. Lett.*, vol. 16, no. 12, pp. 2044–2047, Dec. 2012.
[20] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast list decoders for polar codes," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 2, pp. 318–328, Feb. 2016.
[21] C. Xia, Y. Fan, and C.-Y. Tsui, "A two-staged adaptive successive cancellation list decoding for polar codes," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.
[22] J. Tong, H. Zhang, L. Huang, X. Liu, and J. Wang, "An asymmetric adaptive SCL decoder hardware for Ultra-Low-Error-Rate polar codes," in *Proc. 16th Int. Symp. Wireless Commun. Syst. (ISWCS)*, Aug. 2019, pp. 532–536.
[23] A. Balatsoukas-Stimming, A. J. Raymond, W. J. Gross, and A. Burg, "Hardware architecture for list successive cancellation decoding of polar codes," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 8, pp. 609–613, Aug. 2014.
[24] A. Balatsoukas-Stimming, M. Bastani Parizi, and A. Burg, "LLR-based successive cancellation list decoding of polar codes," *IEEE Trans. Signal Process.*, vol. 63, no. 19, pp. 5165–5179, Oct. 2015.
[25] P. Giard, A. Balatsoukas-Stimming, T. C. Muller, A. Bonetti, C. Thibeault, W. J. Gross, P. Flatresse, and A. Burg, "PolarBear: A 28-nm FD-SOI ASIC for decoding of polar codes," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 7, no. 4, pp. 616–629, Dec. 2017.
[26] O. Afisiadis, A. Balatsoukas-Stimming, and A. Burg, "A low-complexity improved successive cancellation decoder for polar codes," in *Proc. 48th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2014, pp. 2116–2120.
[27] X. Liu, Q. Zhang, P. Qiu, J. Tong, H. Zhang, C. Zhao, and J. Wang, "A 5.16 Gbps decoder ASIC for polar code in 16nm FinFET," in *Proc. 15th Int. Symp. Wireless Commun. Syst. (ISWCS)*, Aug. 2018, pp. 1–5.
[28] S. Ali Hashemi, C. Condo, F. Ercan, and W. J. Gross, "Memory-efficient polar decoders," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 7, no. 4, pp. 604–615, Dec. 2017.
[29] S. A. Hashemi, M. Mondelli, S. H. Hassani, C. Condo, R. L. Urbanke, and W. J. Gross, "Decoder partitioning: Towards practical list decoding of polar codes," *IEEE Trans. Commun.*, vol. 66, no. 9, pp. 3749–3759, Sep. 2018.
[30] Y. Wang, L. Chen, Q. Wang, Y. Zhang, and Z. Xing, "Algorithm and architecture for path metric aided bit-flipping decoding of polar codes," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2019, pp. 1–6.
[31] I. Tal and A. Vardy, "How to construct polar codes," *IEEE Trans. Inf. Theory*, vol. 59, no. 10, pp. 6562–6582, Oct. 2013.
[32] P. Trifonov, "Efficient design and decoding of polar codes," *IEEE Trans. Commun.*, vol. 60, no. 11, pp. 3221–3227, Nov. 2012.
[33] G. He, J.-C. Belfiore, I. Land, G. Yang, X. Liu, Y. Chen, R. Li, J. Wang, Y. Ge, R. Zhang, and W. Tong, "Beta-expansion: A theoretical framework for fast and recursive construction of polar codes," in *Proc. IEEE Global Commun. Conf.*, Dec. 2017, pp. 1–6.
[34] C. Leroux, A. J. Raymond, G. Sarkis, and W. J. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Trans. Signal Process.*, vol. 61, no. 2, pp. 289–299, Jan. 2013.
[35] C. Zhang and K. K. Parhi, "Low-latency sequential and overlapped architectures for successive cancellation polar decoder," *IEEE Trans. Signal Process.*, vol. 61, no. 10, pp. 2429–2441, May 2013.
[36] T. Che, J. Xu, and G. Choi, "Overlapped list successive cancellation approach for hardware efficient polar code decoder," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 2463–2466.
[37] A. Balatsoukas-Stimming, M. Bastani Parizi, and A. Burg, "On metric sorting for successive cancellation list decoding of polar codes," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 1993–1996.
[38] H. Li, "Enhanced metric sorting for successive cancellation list decoding of polar codes," *IEEE Commun. Lett.*, vol. 22, no. 4, pp. 664–667, Apr. 2018.
[39] R. Shrestha and A. Sahoo, "High-speed and hardware-efficient successive cancellation polar-decoder," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 66, no. 7, pp. 1144–1148, Jul. 2019.



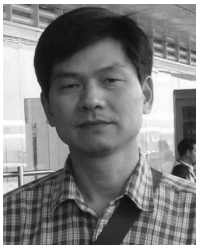
YU WANG (Student Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from Air Force Engineering University, Xi'an, China, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree in electrical engineering with the High-Performance Microprocessor Research Group, National University of Defense Technology, Changsha, China.

His current research interests include error-correction codes, hardware architecture optimization, and VLSI architecture design for digital signal processing and communication systems.

QINGLIN WANG born in 1987. He received the B.S. degree in mechanical engineering from Tsinghua University, China, in 2009, and the M.S. and Ph.D. degrees in electrical science and Technology from the National University of Defense Technology, China, in 2016. He has been a Research Assistant with the Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, since 2016. His research interests include high-performance computing, VLSI signal processing, and machine learning.

YANG ZHANG received the B.S. degree in electronic engineering and science technology from Xian Jiao Tong University, in 2011, and the M.S. degree in electrical engineering from the National University of Defense Technology, in 2013. He is currently an Assistant Professor with the High-Performance DSP Research Group, National University of Defense Technology. His research interests include the next-generation multicore architecture design, optimization of parallel programs on GPGPU, and high-efficient DSP design.

SHIKAI QIU received the B.S. degree in electrical engineering from Shanghai Jiao Tong University, Shanghai, China, in 2017. He is currently pursuing the master's degree in electronic science and technology with National University of Defense Technology, Hunan, China. His current research interests include 5G, microprocessor technology, and VLSI signal processing.



ZUOCHENG XING (Member, IEEE) received the B.S. degree from the Guilin University of Electronic and Technology, in 1987, and the M.S. and Ph.D. degrees from the National University of Defense Technology, in 1990 and 2001, respectively.

He was a Professor with the School of Computer, National University of Defense Technology, one of the academic leader of high-performance computer architecture and micro-electronics, and solid electronics and doctoral tutor. He has been engaged in teaching and research on computer science for over twenty years, and responsible or take part in over 20 important projects, including Galaxy and TH-1/1A/2 series high-performance supercomputers design and FT series high-performance general-purpose CPU design, the National Natural Science Foundation, National Defense Pre-research funds, and so on. His research interests include microprocessor architecture design, 5G wireless communications and VLSI architecture design for communication.