2015

High Performance Decoder Architectures for Error Correction Codes

Jun Lin
Lehigh University

HIGH PERFORMANCE DECODER ARCHITECTURES FOR ERROR CORRECTION CODES

by

Jun Lin

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Lehigh University

May 2015

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Prof. Zhiyuan Yan
(Dissertation Advisor)

Accepted Date

Committee Members:

Prof. Zhiyuan Yan
(Committee Chair)

Prof. Meghanad D. Wagh

Prof. Tiffany Jing Li

Dr. Zhongfeng Wang
Broadcom Inc.

Acknowledgments

First and foremost, I would like to express my special appreciation to my advisor Prof. Zhiyuan Yan for his continuous support of my Ph.D study and research. His rich experiences, insightful instructions and valuable expertise are the most important gifts I have ever received. Instead of sticking to a single topic, Prof. Yan encouraged me to explore more interesting and challenging research topics, which broadens my views and paves ways for my future career. His patience and encouragement inspired me to pursuit my dream as an academic researcher. Moreover, he taught me how to appreciate various ideas in a comprehensive manner and express complicated concepts and works in a clear way. Prof. Yan is also my life mentor and gave me a lot of valuable advices in seeking career opportunities. He recommended me for an internship in Qualcomm, which is one of the most wonderful experiences during the Ph. D study. It is really lucky for me to pursue my Ph.D degree under his guidance. I could not imagine having a better advisor and mentor for my Ph.D study.

Besides, I would like to thank Prof. Wagh, Prof. Tiffany Jing Li and Dr. Zhongfeng Wang for serving on my Ph.D committee and spending their precious time for examming my work. I am grateful for their encouragement, insightful

comments and inspiring questions.

Many sincere thanks to my labmates and friends in Lehigh University, Feng Shi, Hongmei Xie and Chenrong Xiong. We shared a lot of memorable time and exciting discussions. It is exciting to cooperate with them on developing wonderful ideas and results. In particular, I would like to thank them for helping me in finding an apartment and shopping grocery every week. My life in Lehigh University could be much harder without their help. Many thanks also go to my friends at Lehigh: Yang Liu, Jiangfan Zhang, Xuanxuan Lu et al. The wonderful time spent with you will be a precious part of my memory.

Finally, I would like to thank my parents for their endless support and unconditional love. They provided everything financially and spiritually for me to get better educations. Without their love and encouragement, I would not have made any achievements in my life.

Contents

List of Tables

List of Figures

Abstract

Due to the rapid development of the information industry, modern communication and storage systems require much higher data rates and reliability to server various demanding applications. However, these systems suffer from noises from the practical channels. Various error correction codes (ECCs), such as Reed-Solomon (RS) codes, convolutional codes, turbo codes, Low-Density Parity-Check (LDPC) codes and so on, have been adopted in lots of current standards. With the increasing data rate, the research of more advanced ECCs and the corresponding efficient decoders will never stop.

Binary LDPC codes have been adopted in lots of modern communication and storage applications due their superior error performance and efficient hardware decoder implementations. Non-binary LDPC (NB-LDPC) codes are an important extension of traditional binary LDPC codes. Compared with its binary counterpart, NB-LDPC codes show better error performance under short to moderate block lengths and higher order modulations. Moreover, NB-LDPC codes have lower error floor than binary LDPC codes. In spite of the excellent error performance, it is hard for current communication and storage systems to adopt NB-LDPC codes due to

complex decoding algorithms and decoder architectures. In terms of hardware implementation, current NB-LDPC decoders need much larger area and achieve much lower data throughput.

Besides the recently proposed NB-LDPC codes, polar codes, discovered by Arıkan, appear as a very promising candidate for future communication and storage systems. Polar codes are considered as a major breakthrough in recent coding theory society. Polar codes are proved to be capacity achieving codes over binary input symmetric memoryless channels. Besides, polar codes can be decoded by the successive cancelation (SC) algorithm with of complexity of $\mathcal{O}(N \log_2 N)$, where N is the block length. The main sticking point of polar codes to date is that their error performance under short to moderate block lengths is inferior compared with LDPC codes or turbo codes. The list decoding technique can be used to improve the error performance of SC algorithms at the cost higher computational and memory complexities. Besides, the hardware implementation of current SC based decoders suffer from long decoding latency which is unsuitable for modern high speed communications.

ECCs also find their applications in improving the reliability of network coding. Random linear network coding is an efficient technique for disseminating information in networks, but it is highly susceptible to errors. Kötter-Kschischang (KK) codes and Mahdavifar-Vardy (MV) codes are two important families of subspace codes that provide error control in noncoherent random linear network coding. List decoding has been used to decode MV codes beyond half distance. Existing hardware implementations of the rank metric decoder for KK codes suffer from limited throughput, long latency and high area complexity. The interpolation-based list decoding algorithm for MV codes still has high computational complexity, and its

feasibility for hardware implementations has not been investigated.

In this exam, we present efficient decoding algorithms and hardware decoder architectures for NB-LDPC codes, polar codes, KK and MV codes. For NB-LDPC codes, an efficient shuffled decoder architecture is presented to reduce the number of average iterations and improve the throughput. Besides, a fully parallel decoder architecture for NB-LDPC codes with short or moderate block lengths is also presented. Our fully parallel decoder architecture achieves much higher throughput and area efficiency compared with the state-of-art NB-LDPC decoders. For polar codes, a memory efficient list decoder architecture is first presented. Based on our reduced latency list decoding algorithm for polar codes, a high throughput list decoder architecture is also presented. At last, we present efficient decoder architectures for both KK and MV codes.

Chapter 1

Introduction

Error correction codes (ECCs), such as Reed-Solomon (RS) codes, convolutional codes, turbo codes, Low-Density Parity-Check (LDPC) codes and so on, are widely used in current communication and storage systems. Non-binary LDPC codes and polar codes are recently emerged ECCs for future applications. However, NB-LDPC codes suffer from high decoding algorithms and inefficient hardware decoder architecture. The successive cancelation (SC) based list (SCL) decoding algorithm for polar codes has much better error performance that the SC algorithm. However, the hardware implementations of the SCL decoding algorithm suffer from long decoding latency, which is unsuitable for high speed applications. Besides, ECCs also find applications in random linear network coding (RLNC). The Kötter-Kschischang (KK) codes and Mahdavifar-Vardy (MV) codes are two important families of subspace codes that provide error control in noncoherent random linear network coding.

In this chapter, we first explain our motivations of our research in Sec. 1.1, and then present our main contributions in this dissertation as well as the organization

of this dissertation in Sec. 1.2.

1.1 Motivations

1.1.1 Non-binary LDPC Codes

Binary low-density parity-check (LDPC) codes are more and more popular in applications because of their capacity-approaching performance. In terms of performance, binary LDPC codes start to show their weaknesses when the codeword length is small or moderate, or when a higher order modulation is used. For these cases, nonbinary LDPC (NB-LDPC) codes over high order Galois fields have shown great potential [3, 4]. For instance, in [1], a rate-1/2 NB-LDPC code of length 84 over GF(64) is shown to perform 0.375dB better than a rate-1/2 binary irregular LDPC code of equivalent length 504 bits in [5] over binary input additive white Gaussian noise (AWGN) channel. Over the QAM-AWGN channels, NB-LDPC codes with a field order greater than or equal to the size of constellation have the advantage that the encoder/decoder works directly with symbols. All mapping choices of the codeword symbols to the constellation points are equivalent and lead to the same performance. In [1], an NB-LDPC code over GF(256) performs 0.5dB better than a rate-1/2 binary LDPC codes of the equivalent length 1008 bits over QAM256-AWGN channel.

A significant obstacle to the application of NB-LDPC codes is that their decoding algorithms have high complexities. Hence, a lot of research effort has been spent on efficient decoding algorithms for NB-LDPC codes [1, 6]. Among them, the EMS [1] and the Min-Max [6] algorithms draw a lot of attention because of their low

computation and memory complexity. For an NB-LDPC code over GF(2^m), both the EMS and the Min-Max algorithms store only the n_m ($n_m \ll 2^m$) most reliable messages, thus reducing the memory requirement at the cost of small performance degradation. The check node processing of the EMS algorithm needs additions and comparisons, while the Min-Max algorithm needs only maximizations and comparisons in check node processing. The trellis-based check node processing (TBCP) algorithm [7] reduces the computational complexity of check node processing of the Min-Max algorithm by eliminating unnecessary check-to-variable messages.

Besides the Min-Max and EMS decoding algorithms, stochastic decoding [2, 8, 9] is another way to reduce the hardware complexity of NB-LDPC decoders while maintaining the decoding performance. Compared to conventional belief propagation decoding algorithms, the stochastic decoding algorithm has lower hardware complexity [9]. The relaxed half-stochastic decoding algorithm optimized for NB-LPDC codes with variable node degree 2, called the RD2 algorithm, was proposed in [8]. The RD2 decoding algorithm reduces the decoding complexity by reducing the number of real multiplications significantly. An improved version of the RD2 algorithm, called the NoX decoding algorithm [2], is proposed to further reduce the computational complexity.

Recently, a considerable amount of research effort has been spent on efficient decoder architectures for NB-LDPC codes [9–17]. Existing NB-LDPC decoders still suffer from low throughput and large hardware complexity. For example, a (248, 124) NB-LDPC decoder over GF(32) [15] achieves a throughput of 47.69 Mb/s at the cost of 10.33 mm^2 silicon area under 90nm technology. An (837, 726) NB-LDPC decoder [16] over GF(32) achieves a throughput of 60Mb/s at the cost of 1.29M

standard NAND gates using the 180nm technology. The FPGA implementation of a (192, 96) stochastic AMSA decoder [9] over GF(256) achieves a throughput of 64Mb/s at the frequency of 108MHz.

1.1.2 Polar Codes

Polar codes, recently introduced by Arıkan [18], are a significant breakthrough in coding theory. It is proved that polar codes can achieve the channel capacity of any discrete or continuous memoryless channel [18, 19]. Polar codes can be efficiently decoded by the low-complexity successive cancelation (SC) decoding algorithm [18] with a complexity of $O(N \log N)$, where N is the block length. To approach the channel capacity using the SC algorithm, polar codes require very large code block length (for example, $N > 2^{20}$ [20]), which is impractical in many applications. For short or moderate length, the error performance of polar codes under the SC algorithm is worse than that of Turbo or low-density parity-check (LDPC) codes [21].

Lots of efforts [21–28] have already been devoted to the improvement of error-correction performance of polar codes with short or moderate lengths. An SC list (SCL) decoding algorithm was proposed recently in [21], which performs better than the SC algorithm and performs almost the same as a maximum-likelihood (ML) decoder [21]. In [22–24], the cyclic redundancy check (CRC) is used to pick the output codeword from L candidates, where L is the list size. The CRC-aided SCL algorithm performs much better than the SCL algorithm at the expense of negligible loss in code rate.

In terms of hardware implementations of the SC algorithm, an efficient semi-parallel SC decoder was proposed in [20], where resource sharing and semi-parallel

processing were used to reduce the hardware complexity. An overlapped computation method and a pre-computation method were proposed in [29] to improve the throughput and to reduce the decoding latency of SC decoders. Compared to the semi-parallel decoder architecture in [20], the pre-computation based decoder architecture [29] can double the throughput. A simplified SC decoder for polar codes, proposed in [30], reduces the decoding latency by more than 88% for a rate 0.7 polar code with length 2^{18}.

Despite its significantly improved error performance, the hardware implementations of SC based list decoders [31–34] still suffer from long decoding latency and limited throughput due to the serial decoding schedule. In order to reduce the decoding latency of an SC based list decoder, M ($M > 1$) bits are decoded in parallel in [35–37], where the decoding latency can be reduced by M times ideally. However, for the hardware implementations of the algorithms in [35–37], the actually achieved decoding latency reduction is less than M due to extra decoding cycles on finding the L most reliable paths among $2^M L$ candidates, where L is list size. A software adaptive SSC-list-CRC decoder was proposed in [38]. For a (2048, 1723) polar+CRC-32 code, the SSC-list-CRC decoder with $L = 32$ was shown to be about 7 times faster than an SC based list decoder. However, it is unclear whether the list decoder in [38] is suitable for hardware implementation.

1.1.3 Error Control Decoders for RLNC

Random linear network coding (RLNC) is an efficient technique for disseminating information in networks (see, for example, [39–42]). Due to its random linear operations, RLNC not only achieves network capacity with high probability in a

distributed manner, but also provides robustness against varying network conditions [43]. Unfortunately, it is highly susceptible to errors due to noise, malicious or malfunctioning nodes, or insufficient min-cut [44]. As a result, error control is vital for RLNC.

Error control methods proposed for RLNC assume two transmission models. The methods for the first model (see, for example, [45]) depend on and take advantage of the underlying network topology or the particular linear networking operations performed at various nodes. The methods for the other model (see, e.g., [44, 46]) assume that both the transmitter and the receiver have no knowledge of such channel transfer characteristics. The two models are referred to as coherent and noncoherent network coding, respectively. In this paper, we focus on error control for noncoherent RLNC.

An error control code for noncoherent network coding [44], called a subspace code, is a set of subspaces. Information is encoded in the choice of a subspace spanned by a set of transmitted packets. A subspace code is called a constant-dimension code (CDC) if all subspaces are of the same dimension. CDCs lead to simplified network protocols due to the constant dimension. A class of asymptotically optimal CDCs, referred to as Kötter-Kschischang (KK) codes, has been proposed in [44]. A decoding algorithm based on interpolation for bivariate linearized polynomials is also proposed for KK codes in [44]. It was shown in [46] that KK codes correspond to lifting of Gabidulin codes, a class of optimal rank metric codes. As a result, KK codes can be decoded by the generalized decoding algorithm for the rank metric codes [46].

Motivated by KK codes, a new family of subspace codes, referred to as Mahdavifar-Vardy (MV) codes in this paper, was proposed [47–49]. List decoding, which has been used to decode beyond the error correction diameter bound [50], can be applied to the decoding of MV codes. Using algebraic list decoding, it was shown [49] that MV codes can achieve a better tradeoff between rate and decoding radius than KK codes.

Error control for RLNC comes at the expense of additional computations needed for encoding and decoding. The complexities of existing decoding algorithms [44, 49, 51] for KK and MV codes are much higher than those of encoding, and are hence critical to applications of RLNC. Most previous works focus on theoretical aspects of network coding. For example, the decoding complexities of KK and MV codes were analyzed in [44, 46] and [47–49], respectively. However, theoretical analysis does not completely reflect how the decoding algorithms affect the hardware implementation results, such as area and throughput. For KK codes, decoder architectures based on the generalized decoding algorithm for rank metric codes [46] was proposed in [43]. Unfortunately, the rank metric decoder architectures in [43] suffer from limited throughput, long decoding latency and high area complexity. Besides, to the best of our knowledge, decoder architectures for MV codes and their hardware implementations have not been investigated in the open literature.

1.2 Contributions and Organization

This dissertation has the following contributions and is organized as follows.

- In Chapter 2, the shuffled decoding algorithm and its corresponding decoder

architecture are investigated. Our main contributions of this chapter are two-fold. First, we propose a shuffled schedule (SS) of the Min-Max algorithm for NB-LDPC codes. To reduce the memory requirement and improve the throughput, we also propose a modified shuffled schedule (MSS), which employs a novel shuffle sort (SST) algorithm to reduce the complexity of check node processing significantly. Our simulation results show that both the SS and MSS converge faster and have slightly better error performance than the flooding schedule, and that the degradation of the MSS in error performance as well as convergence rate is negligible. The simulation results also show that the error performance of the MSS and layered schedule are almost the same. Second, an efficient shuffled decoder architecture for NB QC-LDPC codes is proposed based on the Min-Max algorithm using the modified shuffled schedule. The proposed architecture has a similar top structure to other partly parallel decoder architectures for binary and nonbinary LDPC codes. However, it has several key novelties: 1) its underlying modified shuffled schedule is novel; 2) on-the-fly computation and hardware re-usage have been used to reduce memory consumption and to improve the throughput; 3) a random memory address generator (RMAG) has been employed in the check node unit (CNU) to reduce the number of cycles required by CNP; 4) since the variable node unit (VNU) becomes complex for decoders storing only the n_m most reliable values, the variable-to-check messages are stored in an improved way so as to simplify the message access.

- In Chapter 3, a fully parallel decoder architecture based on the proposed decoding algorithm is also proposed. The main contributions of this paper are

as follows:

1. Based on the Min-Max algorithm, a reduced memory complexity trellis based check node processing (RTBCP) algorithm is proposed.
2. A simplified algorithm is proposed to reduce the computational complexity of variable node processing (VNP). As a result, compared with the RHS algorithm in [2], a stochastic decoder, the proposed decoding algorithm needs fewer real multiplications but more real comparisons and finite field additions.
3. For each *a priori* message, all LLRs except several most reliable ones are approximated with a linear function. Two kinds of low complexity LLR generation units are also proposed for the approximation of the check-to-variable (c-to-v) LLR and *a priori* LLR, respectively. With 5-bit quantization scheme and $n_m = 32$, the areas of the two LGUs are 10.7% and 13.3%, respectively, of that of an SRAM which stores an LLR vector under a 90nm CMOS technology. A similar approach was proposed in [52] to approximate *a priori* LLR. The main differences between our work and that in [52] are as follows:
   - Besides the approximation of channel LLR vectors, we try to approximate check-to-variable LLR vectors.
   - A simplified variable node processing (SVNP) algorithm is proposed to compensate the performance degradation caused by LLR approximation.
4. A parallel check node unit (CNU) and a low-latency variable node unit (VNU) are proposed. Based on the proposed CNU and VNU, an efficient

fully parallel decoder architecture is also proposed. A fully parallel NB-LDPC decoders based on GF(256) is implemented with 28nm CMOS technology. The decoder over GF(256) achieves a throughput of 546Mb/s and an energy efficiency of 0.178nJ/b/iter.

Since routing congestion tends to be challenging for fully parallel LDPC decoder architectures, the proposed decoder architecture is not suitable for very long LDPC codes. The proposed fully parallel decoder architecture is particularly advantageous for NB-LDPC codes over large fields, since the memory reduction will be more significant when n_m is large.

- In Chapter 4, we focus on efficient architectures and their hardware implementations of interpolation based decoders for KK and MV codes. The main contributions of this paper are:

  1. The decoder of KK codes has two stages: interpolation and factorization. The generalized interpolation algorithm in [51] is used for the first stage since it is more efficient than Gaussian elimination [51]. For factorization, we propose a reformulated right division algorithm for linearized polynomials, which is suitable for hardware implementations.

  2. The list decoder of MV codes also has two stages: interpolation and factorization. The generalized interpolation algorithm in [51] is used in the interpolation process. A linearized Roth-Ruckenstein (LRR) algorithm [53] is proposed in [47] to solve the factorization problem for MV codes. In this paper, we make a more detailed study on the LRR algorithm. For list size $L = 2$, we derive the equations used to compute all

the information symbols and uncover the relation between two possible solutions. A matrix based LRR (M-LRR) algorithm, which is suitable for hardware implementations, is also proposed for factorization.

3. A serial decoder architecture and an unfolded decoder architecture for KK codes are proposed for applications with moderate and high throughputs, respectively. Both architectures are implemented for KK codes over GF(2^8) and GF(2^{16}) to demonstrate their efficiency. To the best of our knowledge, this is the first efficient implementation of interpolation-based decoder for KK codes. Compared to the rank metric decoder architectures for KK codes [43], the proposed serial decoder architecture improves the throughput by 4.9 and 13.2 times, while its gate counts are only 56% and 76% of their respective counterparts in [43]. Moreover, for these two codes, the unfolded architecture achieves a throughput of 12.5Gb/s and 41.6Gb/s, much higher than the throughput of 214Mb/s and 134Mb/s of their respective counterparts in [43]. The throughputs per thousand NAND gates of our architectures are much higher and their latency much shorter than their counterparts in [43].

4. A serial list decoder architecture for MV codes is proposed. To the best of our knowledge, this is the first hardware implementation of MV decoders. An efficient architecture for solving equations over an extension field GF(q^{ml}) ($q > 2$ is moderate) is proposed. The proposed equation solver does not require complicated inversion operations over GF(q^{ml}). Besides, an implementation of factorization that computes all L possible transmitted packets in parallel is proposed, where L is the list size for

list decoding.

- In Chapter 5, we propose the first hardware implementation of the CA-SCL algorithm to the best of our knowledge. Based on both algorithmic and architectural improvements, our decoder architecture achieves better error performance and higher area efficiency compared with the decoder architecture in [31]. Specifically, the major contributions of this work are:

  1. Message memories account for a significant fraction of an SC or SCL decoder [20, 31]. In this chapter, an area efficient message memory architecture is proposed. Besides, a new compression method for the channel messages is used to reduce the area of the proposed decoder architecture.

  2. An efficient processing unit (PU) is proposed. For the proposed list decoder architecture, a fine grained PU profiling (FPP) algorithm is proposed to determine the minimum quantization size of each input message for each PU so that there is no message overflow. By using the quantization size generated by the FPP algorithm for each PU, the overall area of all PUs is reduced.

  3. An efficient scalable path pruning unit (PPU) is proposed to control the copying of decoding paths. Based on the proposed memory architecture and the scalable PPU, our list decoder architecture is suitable for large list sizes.

  4. A low-complexity direct selection scheme is proposed for the CA-SCL algorithm when a strong CRC is used (e.g. CRC32). The proposed direct selection scheme simplifies the selection of the final output data

word.

5. For a (1024, 512) rate-$\frac{1}{2}$ polar code, the proposed list decoder architecture is implemented for list size $L = 2$ and 4, respectively, under a 90nm CMOS technology. Compared with the decoder architecture in [31] synthesized under the same technology, our decoder achieves 1.24 to 1.83 times area efficiency (throughput normalized by area). Besides, the proposed CA-SCL decoder has better error performance compared with the SCL decoder in [31].

- In Chapter 6, a tree based reduced latency list decoding algorithm and its corresponding high throughput hardware architecture are proposed for polar codes. The main contributions are:

  – A tree based reduced latency list decoding (RLLD) algorithm over logarithm likelihood ratio (LLR) domain is proposed for polar codes. Inspired by the simplified successive cancelation (SSC) [30] decoding algorithm and the ML-SSC algorithm [54], our RLLD algorithm performs the SC based list decoding on a binary tree. Previous SCL decoding algorithms visit all the nodes in the tree and consider all possibilities of the information bits, while our RLLD algorithm visits much fewer nodes in the tree and consider fewer possibilities of the information bits. When configured properly, our RLLD algorithm significantly reduces the decoding latency and hence improves throughput, while introducing little performance degradation.

– Based on our RLLD algorithm, a high throughput list decoder architecture is proposed for polar codes. Compared with the state-of-arts SCL decoders in [32, 33, 36], our list decoder achieves lower decoding latency and higher area efficiency (throughput normalized by area).

More specifically, the major innovations of the proposed decoder architecture are:

– An index based partial sum computation (IPC) algorithm is proposed to avoid copying partial sums directly when one decoding path needs to be copied to another. Compared with the lazy copy algorithm in [55], our IPC algorithm is more hardware friendly since it copies only path indices, while the lazy copy algorithm needs more complex index computation.

– Based on our IPC algorithm, a hybrid partial sum unit (Hyb-PSU) is proposed so that our list decoder is suitable for larger block lengths. The Hyb-PSU is able to store most of the partial sums in area efficient memories such as register file (RF) or SRAM, while the partial sum units (PSUs) in [31–33] store partial sums in registers, which need much larger area when the block length N is larger. Compared with the PSU of [32], our Hyb-PSU achieves an area saving of 23% and 63% for block length $N = 2^{13}$ and 2^{15}, respectively, under the TSMC 90nm CMOS technology.

– For our RLLD algorithm, when certain types of nodes are visited, each current decoding path splits into multiple ones, among which the L most reliable paths are kept. In this paper, an efficient path pruning unit (PPU) is proposed to find the L most reliable decoding paths among

the split ones. For our high throughput list decoder architecture, the proposed PPU is the key to the implementation of our RLLD algorithm.

– For the fixed-point implementation of our RLLD algorithm, a memory efficient quantization (MEQ) scheme is used to reduce the number of stored bits. Compared with the conventional quantization scheme, our MEQ scheme reduces the number of stored bits by 17%, 25% and 27% for block length $N = 2^{10}$, 2^{13} and 2^{15}, respectively, at the cost of slight error performance degradation.

Chapter 2

An Efficient Shuffled Decoder Architecture for Nonbinary Quasi-Cyclic LDPC Codes

2.1 Introduction

Binary low-density parity-check (LDPC) codes are more and more popular in applications because of their capacity-approaching performance. In terms of performance, binary LDPC codes start to show their weaknesses when the code word length is small or moderate, or when higher order modulation is used for transmission. For these cases, nonbinary LDPC (NB-LDPC) codes over high order Galois fields have shown great potential [3, 4]. For instance, in [1], a rate-1/2 NB-LDPC code of length 84 over GF(64) is shown to perform 0.375dB better than a rate-1/2 binary irregular LDPC code of equivalent length 504 bits in [5] over binary input

additive white Gaussian noise (AWGN) channel. Over the QAM-AWGN channels, NB-LDPC codes with a field order greater than or equal to the size of constellation have the advantage that the encoder/decoder works directly with symbols. All mapping choices of the codeword symbols to the constellation points are equivalent and lead to the same performance. In [1], an NB-LDPC code over GF(256) performs 0.5dB better than a rate-1/2 binary LDPC codes of the equivalent length 1008 bits over QAM256-AWGN channel.

A significant obstacle to the application of NB-LDPC codes is that their decoding algorithms are of high complexity. Hence, a lot of research effort has been spent on efficient decoding algorithms for NB-LDPC codes [3, 6, 56–58]. Among them, the EMS [58] and the Min-Max [6] algorithms draw the most attention because their low computation and memory complexity. Both the EMS and the Min-Max algorithms can store only the n_m ($n_m \ll q$) most reliable messages, thus reducing the memory requirement at the cost of small performance degradation. The check node processing of the EMS algorithm needs addition and comparison operations, while the Min-Max algorithm needs only maximization and comparison in check node processing. The trellis-based check node processing (TBCP) algorithm [7] reduces the computational complexity of check node processing (CNP) of the Min-Max algorithm by eliminating unnecessary check-to-variable messages from CNP.

Recently, a considerable amount of research effort has already been spent on efficient decoder architectures for NB-LDPC codes [7, 10, 11, 15, 59] based on the EMS or the Min-Max algorithm. The existing NB-LDPC decoders still suffer from low throughput and large hardware complexity. For example, a (248, 124) NB-LDPC decoder over GF(32) [15] achieves a throughput of 47.69 Mb/s at the cost of

10.33 mm^2 silicon area under 90nm technology, while a (1200, 720) binary LDPC decoder [60] achieves a throughput of 5.92 Gb/s at the cost of 13.5 mm^2 silicon area under 180nm technology.

The main contributions of this chapter are two-fold. First, we propose a shuffled schedule (SS) of the Min-Max algorithm for NB-LDPC codes. To reduce the memory requirement and improve the throughput, we also propose a modified shuffled schedule (MSS), which employs a novel shuffle sort (SST) algorithm to reduce the complexity of check node processing significantly. Our simulation results show that both the SS and MSS converge faster and have slightly better error performance than the flooding schedule, and that the degradation of the MSS in error performance as well as convergence rate is negligible. The simulation results also show that the error performance of the MSS and layered schedule are almost the same. Second, an efficient shuffled decoder architecture for NB QC-LDPC codes is proposed based on the Min-Max algorithm using the modified shuffled schedule. The proposed architecture has a similar top structure to other partly parallel decoder architectures for binary and nonbinary LDPC codes. However, it has several key novelties: 1) its underlying modified shuffled schedule is novel; 2) on-the-fly computation and hardware re-usage have been used to reduce memory consumption and to improve the throughput; 3) a random memory address generator (RMAG) has been employed in the check node unit (CNU) to reduce the number of cycles required by CNP; 4) since the variable node unit (VNU) becomes complex for decoders storing only the n_m most reliable values, the variable-to-check messages are stored in an improved way so as to simplify the message access.

For NB-LDPC codes, a shuffled schedule for the EMS algorithm was proposed

in [1], and a shuffled schedule and a probabilistic shuffled schedule for the belief propagation were proposed in [61]. The work herein differs from the previous works in [1, 61] in three aspects. First, while the works in [61] and [1] focus on the belief propagation and EMS decoding of NB-LDPC codes, respectively, our work considers the Min-Max algorithm. Second, while our shuffled schedule is similar to those in [1, 61], our modified shuffled schedule with reduced-complexity check node processing is novel. Third, while the works in [1, 61] focus on decoding algorithms, this work considers not only decoding algorithms but also decoder architectures as well as their hardware implementations.

For binary LDPC codes, a layered decoder architecture tends to be more efficient than a shuffled decoder architecture. This is because both the CNP and variable node processing (VNP) are relatively simple. However, for NB-LDPC codes, layered decoder architectures [7, 15, 59] have several drawbacks. First, The check node processing of existing decoding algorithms for NB-LDPC codes is complex and requires many cycles to finish. Processing the rows serially in the layered fashion requires more cycles than processing all rows at the same time. Second, the variable node processing for NB-LDPC codes is also complex compared to that of binary LDPC codes. A round of variable node processing for a variable node takes $2n_m$ cycles for the nonbinary layered decoder architecture in [7]. Third, for layered decoder architectures, the variable node processing may be performed several times for one variable node during an iteration, leading to lower throughput. In contrast, the shuffled decoder architecture proposed in this chapter processes all rows concurrently and needs only a round of variable node processing for all variable nodes during an iteration. This reduces the number of cycles needed for an iteration.

The shuffled schedule has also been used in decoder architectures for binary LDPC codes (see, e.g., [62]). However, each check to variable (c-2-v) or variable to check (v-2-c) message is a vector for NB-LDPC codes, whereas each c-2-v or v-2-c message is just a single log likelihood ratio (LLR) for binary codes. This fundamental difference also leads to higher complexities for decoder architectures of NB-LDPC codes than their binary counterparts. The key novelties mentioned above also apply when comparing the proposed architecture herein with that in [62].

The rest of this chapter is organized as follows. Section 2.2 briefly reviews the TBCP algorithm. Section 2.3 proposes the shuffled and modified shuffled schedule and presents our simulation results. Our decoder architecture and the hardware implementation results are presented in Section 2.4. The conclusion is drawn in Section 2.5.

2.2 Background

Consider check node m and variable node n in the Tanner graph [63] defined by H, respectively. Let $M(n)$ denote the set of neighboring check nodes connected to n, and $N(m)$ the set of variable nodes connected to m. For $a \in \mathrm{GF}(q)$, let $L_n(a)$ be the *a priori* information of variable node n concerning the symbol a and $Q_n(a)$ be the *posteriori* information of the same symbol. $R_{m,n}(a)$ and $Q_{m,n}(a)$ denote the messages passed from m to n and from n to m concerning a, respectively. Let c_n be the $(n+1)$-th coordinate of a codeword and s_n be the most likely symbol for c_n. The Min-Max decoding algorithm [6] can be formulated as follows, where $I_{\max}$ denotes the maximal number of iterations and $A(m|a_n = a) \stackrel{\text{def}}{=} \{(a_j)(j \in$

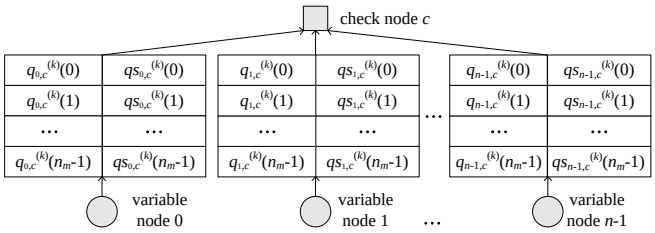

Figure 2.1: Messages sent form check node to variable node

$N(m) \setminus \{n\})| \sum_{j \in N(m)\setminus\{n\}} h_{m,j}a_j + h_{m,n}a = 0\}$:

Algorithm 1: Min-Max Algorithm [6]

Initialization:

$L_n(a) = \ln(\mathrm{P}_r(c_n = s_n|\text{channel})/\mathrm{P}_r(c_n = a|\text{channel}))$

$Q_{m,n}(a) = L_n(a)\ (0 \le m < M, 0 \le a < q), i = 0$

Iteration:

while $HC \neq 0$ **and** $i < I_{max}$ **do**

check node Processing:

$$R_{m,n}(a) = \min_{(a_j)\in A(m|a_n=a)} (\max_{j\in N(m)\setminus\{n\}} Q_{m,j}(a_j))$$

variable node processing:

$$Q'_{m,n}(a) = L_n(a) + \sum_{m'\in M(n)} R_{m',n}(a)$$

$$Q'_{m,n} = \min_{a\in GF(q)} Q_{m,n}(a)$$

$$Q_{m,n}(a) = Q'_{m,n}(a) - Q'_{m,n}$$

tentatively decoding:

$$c_n = \operatorname*{argmin}_a(Q_{m,n}(a))$$

$$i = i + 1$$

The Min-Max algorithm stores the n_m most reliable messages. Suppose there are n variable nodes connected with check node c. The truncated messages sent from n variable nodes to check node c are shown in Fig. 2.1, where $q_{v,c}(k)$ is an LLR value and $qs_{v,c}(k) \in GF(q)$, where GF(q) is a finite field with q elements. Each v-2-c message contains n_m LLRs and n_m associated field symbols.

Normally, the CNP is performed in a forward-backward way [6], which is memory

demanding. The TBCP algorithm [7] eliminates unnecessary v-2-c messages from CNP, thus reducing the memory consumption. The TBCP algorithm first sorts these $n(n_m-1)$ nonzero LLRs in non-decreasing order as $x_c(1), x_c(2), \cdots, x_c(X)$, and only the X smallest ones are kept. Their associated field symbols are $\alpha_c(1)$, $\alpha_c(2)$, $\cdots$, $\alpha_c(X)$, and they belong to variable nodes with indices $e_c(1), e_c(2), \cdots, e_c(X)$. A path construction (PC) algorithm shown in Algorithm 2 is proposed in [7] to compute the truncated c-2-v message: $r_{c,v}$ and $rs_{c,v}$, where $r_{c,v}$ is the n_m-dimension LLR vector and $rs_{c,v}$ is the associated field symbol.

Algorithm 2: PC Algorithm [7]

input : $x_c(i), \alpha_c(i), e_c(i)$ $i = 1, \cdots, X$; $z_c(v)$ $v = 0, \cdots, d_c - 1$; α_{sum}
output: $r_{c,v}(j), rs_{c,v}(j)$ for $j = 0, \cdots, n_m - 1$

Initialization:
$r_{c,v}(0) = 0, rs_{c,v}(0) = \alpha_{sum} \oplus z_c(v), i = 1, cnt = 1, P_{c,0} = [0, 0, \cdots, 0]$
while $cnt < n_m$ **do**
  if $e_c(i) \neq v$ **then**
    $j = cnt$
    for $k = 0$ *to* $j - 1$ **do**
      $\alpha = rs_{c,v}(k) \oplus z_c(e_c(i)) \oplus \alpha_c(i)$
      if $P_{c,k}(e_c(i)) \neq 1$ ***for*** $\alpha \notin rs_{c,t}$ **then**
        $r_{c,v}(cnt) = x_c(i); rs_{c,v}(cnt) = \alpha$
        $P_{c,cnt}(e_c(i)) = 1$
        $P_{c,cnt}(s) = P_{c,k}(s)$ for $s \neq e_c(i)$
        $cnt = cnt + 1$
  $i = i + 1$

As shown in Algorithm 2, $z_c(v) = \alpha_{c,v}(0)$, $v = 1, \cdots, n$ and $\alpha_{sum} = \sum_{k=1}^{n} z_c(j)$. $P_{c,k}$ is an n-dimension vector over GF(2) which stores the constructed path. $\oplus$ denotes addition over GF(q). The constructed n_m LLRs are picked from the sorted list $x_c(i)$, and stored in $r_{c,v}$. Their associated field symbols are stored in $rs_{c,v}$. X

is set to $1.5n_m$ [7] so that the decoding performance could be maintained. It takes $2n_m$ iterations to compute $r_{c,v}, rs_{c,v}$ [7].

2.3 Shuffled and Modified Shuffled Schedule

2.3.1 Shuffled Schedule

Suppose H is divided into G block columns: $H = [H_0\ H_1\ ...\ H_{G-1}]$, where H_i is an $M \times g$ sub-matrix and $g = N/G$. Let $M(n)$ denote the set of neighboring check nodes connected to n, and $N(m)$ denote the set of variable nodes connected to m. Let $I_v(i), S_v(i)$ for $i = 0, \cdots, n_m - 1$ denote an n_m-ary *a priori* message, where I_v is the LLR vector and S_v is the corresponding field symbol vector, respectively. Let $\mathrm{CM}_{c,l}^{(k)}(i) = (x_{c,l}^{(k)}(i), \alpha_{c,l}^{(k)}(i), e_{c,l}^{(k)}(i))$ for $i = 1, \cdots, X$, which are the inputs of the PC algorithm [7] when computing updated c-2-v messages within block column l in iteration k. Let $(q_{v,c}^{(k)}, qs_{v,c}^{(k)})$ and $(r_{c,v}^{(k)}, rs_{c,v}^{(k)})$ denote a v-2-c and c-2-v message in iteration k, respectively. Suppose the row weight for each H_i is exactly 1, which can be easily satisfied by QC-LDPC codes. The proposed shuffled schedule is shown in Algorithm 3.

During the initialization step, for each check node c, the init_sort (IS) algorithm sorts out X LLR values in a non-decreasing order from incoming $d_c(n_m - 1)$ v-2-c message elements, where d_c is the corresponding check node degree. The IS algorithm is shown in Algorithm 4, where $n = d_c$; t, ts and ti are all X-dimension vectors.

The block_vnp(l) function in Algorithm 3 computes the corresponding $q_{v,c}^{(k+1)}$,

Algorithm 3: Shuffled Schedule

Initialization:

$q_{c,v}^{(0)} = I_v, qs_{c,v}^{(0)} = S_v$ for $c \in M(v)$

for $c = 0$ **to** $M - 1$ **do**

  $\mathrm{CM}_{c,0}^{(0)} = \mathrm{IS}(\{(q_{v,c}^{(0)}, qs_{v,c}^{(0)}) | v \in N(c)\})$

Iteration:

for $k = 0$ **to** $I_{max} - 1$ **do**

  for $l = 0$ **to** $G - 1$ **do**

    for $c = 0$ **to** $M - 1$ **do**

      for $\{v \in N(c)$ **and** $lG \leq v < (l+1)G\}$ **do**

        $(r_{c,v}^{(k+1)}, rs_{c,v}^{(k+1)}) = \mathrm{PC}(\mathrm{CM}_{c,v}^{(k)})$

    $(q_{v,c}^{(k+1)}, qs_{v,c}^{(k+1)}) = \mathrm{block\_vnp}(l)$

    for $c = 0$ **to** $M - 1$ **do**

      for $v \in N(c)$ **do**

        if $v < (l+1)G$ **then**

          $tq_{v,c} = q_{v,c}^{(k+1)}; tqs_{v,c} = qs_{v,c}^{(k+1)}$

        else $tq_{v,c} = q_{v,c}^{(k)}; tqs_{v,c} = qs_{v,c}^{(k)}$

      $\mathrm{CM}_{c,l+1}^{(k)} = \mathrm{IS}(\{(tq_{v,c}^{(k+1)}, tqs_{v,c}^{(k+1)}) | j \in N(c)\})$

  $\mathrm{CM}_{c,0}^{(k+1)} = \mathrm{CM}_{c,G}^{(k)}$

Algorithm 4: IS Algorithm

input : $(q_{v,c}^{(k)}, qs_{v,c}^{(k)}) | v \in N(c) = \{v_0, v_2, \cdots, v_{d_c-1}\}$
output: $\mathrm{CM}_{c,l}^{(k)}(i) = (x_{c,l}^{(k)}(i), \alpha_{c,l}^{(k)}(i), e_{c,l}^{(k)}(i))$ for $i = 1, \cdots, X$

$t(i) = q_{v_1,c}^{(k)}(i), ts(i) = qs_{v_1,c}^{(k)}(i), ti(i) = 0$ for $i = 1, 2, \cdots, n_m - 1$
for $j = 1$ **to** $d_c - 1$ **do**
  $a = b = 1$
  for $i = 1$ **to** X **do**
    if $t(a) \leq q_{v_j,c}^{(k)}(b)$ **then**
      $\mathrm{T}_{c,l}^{(k)}(i) = (t(a), ts(a), ti(a))$; $a = a + 1$
    else $\mathrm{T}_{c,l}^{(k)}(i) = (q_{v_j,c}^{(k)}(b), q_{v_j,c}^{(k)}(b), j)$; $b = b + 1$
  $(t(i), ts(i), ti(i)) = \mathrm{CM}_{c,l}^{(k)}(i)$ for all i
$\mathrm{CM}_{c,l}^{(k)} = \mathrm{T}_{c,l}^{(k)}$

$qs_{v,c}^{(k+1)}$ messages within block l. Take variable node v as an example. The q-ary message $Q_{v,c}$ are firstly computed as $Q_{v,c}(s) = L_v(s) + \sum_{c' \in M(j) \backslash c} R_{c',v}(s)$ for $s = 0, 1, \cdots, q-1$. Here, $L_v(s) = I_v(i)$ if $S_v(i) = s$, otherwise $L_v(s) = I_v(n_m - 1)$ which is maximum of I_v. Besides, $R_{c',v}(s) = r_{c',v}^{(k+1)}(i_{c'})$ if $rs_{c',v}^{(k+1)}(i'_c) = s$, otherwise $R_{c',v}(s) = r_{c',v}^{(k+1)}(n_m - 1)$. Finally, $q_{v,c}^{(k+1)}$ and $qs_{v,c}^{(k+1)}$ are computed by sorting $Q_{v,c}$.

2.3.2 Modified Shuffled Schedule

For the shuffled schedule in Algorithm 3, the computation of $\mathrm{CM}_{c,l+1}^{(k)}$ employ the IS algorithm, which needs $(d_c - 1)X$ comparisons. Thus, $d_c(d_c - 1)X$ comparisons are needed for computing all $\mathrm{CM}_{c,l}^{(k)}$ during an iteration. Besides, all v-2-c messages need to be stored. This results in low throughput as well as a significant memory requirement. Instead, we propose a modified shuffled schedule (MSS) which uses a shuffled sort (SST) algorithm to compute $\mathrm{CM}_{c,l+1}^{(k)}$ as shown in Algorithm 5.

Algorithm 5: SST Algorithm

input : $(q_{v,c}^{(k+1)}, qs_{v,c}^{(k+1)}, l), \mathrm{CM}_{c,l}^{(k)}$

output: $\mathrm{CM}_{c,l+1}^{(k)}$

$a = 1, b = 1, c = 1$

for $x = 0$ **to** $n_m + X - 2$ **do**

  if $e_{c,l}^{(k)} == l$ **then** $b = b + 1$; continue

  if $q_{v,c}^{(k+1)}(a) \leq x_{c,l}^{(k)}(b)$ **then**

    $\mathrm{CM}_{c,l+1}^{(k)}(c) = (q_{c,v}^{(k+1)}(a), qs_{c,v}^{(k+1)}(a), l)$

    $c = c + 1$, $a = a + 1$

  else $\mathrm{CM}_{c,l+1}^{(k)}(c) = \mathrm{CM}_{c,l}^{(k)}(b)$; $c = c + 1$, $b = b + 1$

  if $c == X + 1$ **then** break

The proposed SST algorithm needs at most $n_m + X - 1$ comparisons to compute $\mathrm{CM}_{c,l+1}^{(k)}$ based on $\mathrm{CM}_{c,l}^{(k)}$ and $(q_{v,c}^{(k+1)}, qs_{v,c}^{(k+1)})$. It only takes at most $(d_c - 1)X + (d_c - 1)(n_m + X - 1) = (d_c - 1)(n_m + 2X - 1)$ comparisons to compute all $\mathrm{CM}_{c,l}^{(k)}$ during an iteration. Besides, only $(q_{v,c}^{(k+1)}, qs_{v,c}^{(k+1)})$ need to be stored. As a result, compared to the shuffled schedule in Algorithm 3, the MSS using the SST algorithm needs fewer comparisons and less memory.

2.3.3 Simulation Results

Fig. 2.2 shows the frame error rate (FER) performance of the FFT-BP algorithm and the Min-Max algorithm with flooding schedule, shuffled and modified shuffled schedule as well as the layered schedule for three NB-LDPC codes on GF(32) [64] over the AWGN channel with BPSK modulation. For our simulations, $I_{max} = 30$, $n_m = 16$, and $X = 1.5n_m = 24$ for the SS and MSS. The flooding and layered

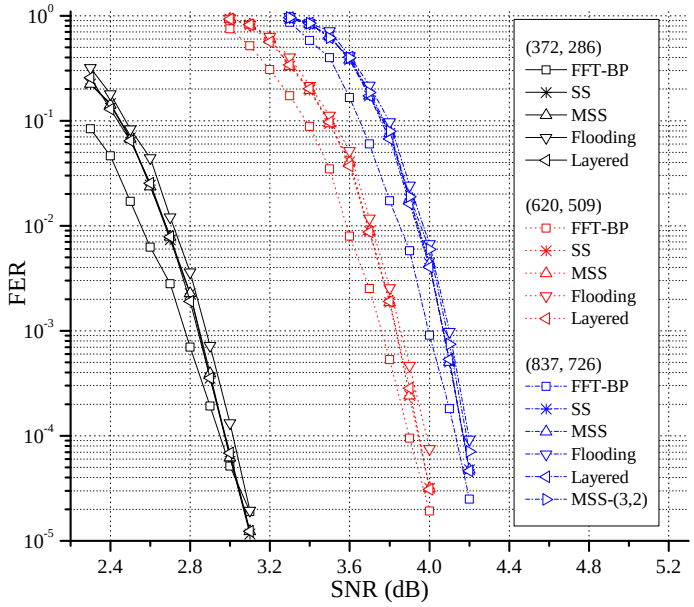

Figure 2.2: FERs of selected codes

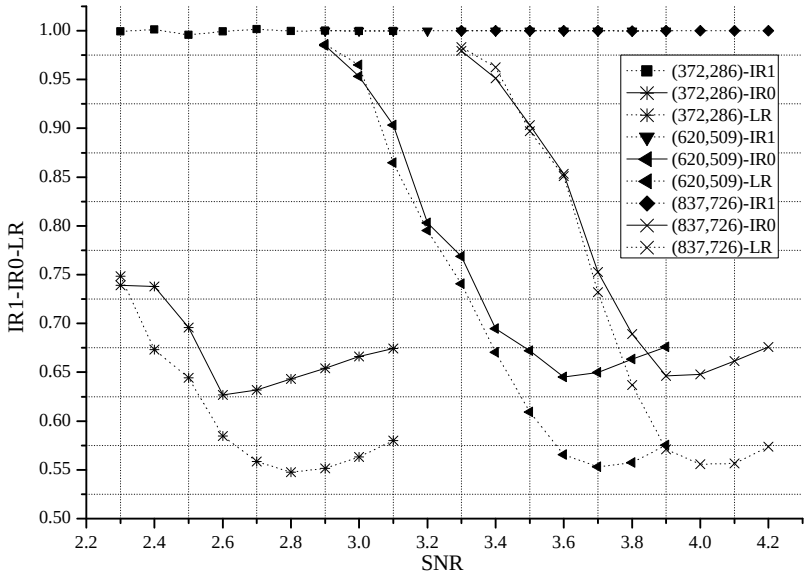

Figure 2.3: Comparison of convergence rates

schedule also use the TBCP algorithm in CNP. For all three codes, the error performance with the MSS and SS is slightly better than that with the flooding schedule. The layered schedule, the SS, and the MSS have nearly the same error performance, which implies that the MSS results in little or no error performance degradation.

We also compare the convergence speed of the MSS, SS and layered schedule with the flooding schedule in Fig. 2.3. Here IR0 = N_{mss}/N_f, IR1 = $N_{\text{ss}}/N_{\text{mss}}$ and LR = N_l/N_f, where N_{ss}, N_{mss}, N_f and N_l are the average numbers of iterations of the SS, the MSS, the flooding and layered schedule, respectively. Several observations can be made about Fig. 2.3. First, throughout the SNR range, IR0 < 1 and IR1 ≈ 1. Thus, the MSS results in no degradation in convergence speed compared with the SS, and both the MSS and SS converge faster than the flooding schedule. Second, when FER is around 10^{-4}, the average number of iteration of the MSS is only $60\% - 70\%$ of that of the flooding schedule. Third, IR0 and LR start to grow in high SNR region, since even the flooding schedule converge very fast at high SNR. Thus the advantage of the MSS and SS in convergence speed decreases when the SNR is high. The same phenomenon was observed in [61].

As shown in Fig. 2.3, the layered schedule requires fewer iterations for the simulated codes than the MSS and SS, especially when the SNR is high. However, in a hardware implementation, the MSS results in fewer clock cycles per iteration than the layered schedule for two reasons. First, all rows can be processed at the same time for the MSS, while only a block of rows can be processed concurrently for the layered schedule. Second, the proposed MSS with the shuffled sort algorithm simplifies the CNP after init_sort is finished. In order to update the c-2-v messages in one block column, only the updated v-2-c messages in the previous block column and

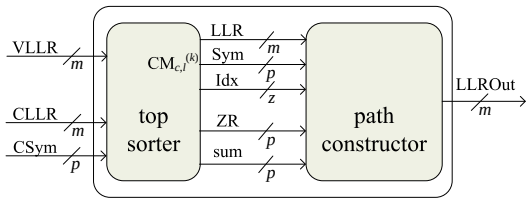

Figure 2.4: CNU Architecture

old CMs are needed. As a result, the number of real value comparisons is reduced.

2.4 Shuffled Decoder Architecture

In this section, a shuffled decoder architecture with reduced memory consumption and higher throughput is proposed for nonbinary QC-LDPC codes whose parity check matrices consist of sub-matrices that are either the zero or shifted identity matrices with nonzero entries replaced by elements of GF(q), where $q = 2^p$.

2.4.1 Check Node Unit Architecture

The architecture of the proposed CNU which includes a top sorter and a path constructor is shown in Fig. 2.4, where m denotes the quantization bits of an LLR message and $z = \lceil \log_2 d_c \rceil$. The top sorter provides corresponding inputs for the path constructor which implements Algorithm 2. Take check node c as an example, the top sorter in Fig. 2.4 computes $\mathrm{CM}_{c,0}^{(0)}$ using the IS algorithm at the initialization step. It also computes $\mathrm{CM}_{c,l+1}^{(k)}$ for $l = 0, 1, \cdots, G-1$ using the SST algorithm during the iteration process. Once $\mathrm{CM}_{c,l}^{(k)}$ is available, the path constructor computes updated c-2-v messages within block column l using Algorithm 2.

The architecture of the proposed top sorter is shown in Fig. 2.5. It consists

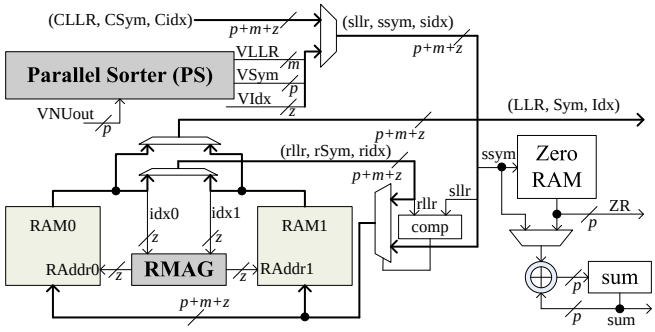

Figure 2.5: Architecture of Proposed Top Sorter

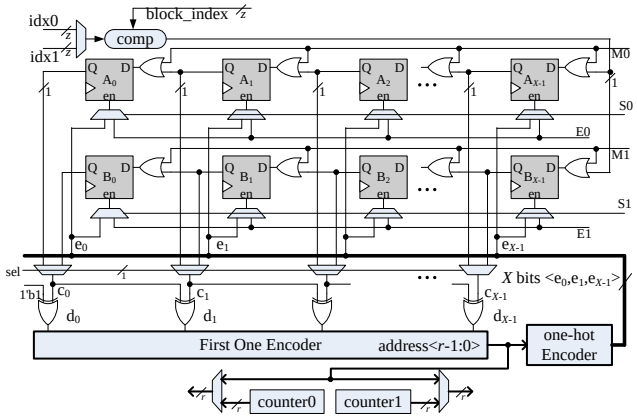

Figure 2.6: Proposed RMAG Architecture

of a parallel porter (PS), two $X\times(m+p+z)$-bits RAMs (RAM0 and RAM1) used to store $\mathrm{CM}_{c,l}^{(k)}$, and a random memory address generator (RMAG). Each word of RAM0 and RAM1 stores an LLR value, the associated field symbol and index. At the initialization step, the computation of $\mathrm{CM}_{c,0}^{(0)}$ is carried out in d_c rounds. In the first round, $q_{v_1,c}^{(0)}$ and $qs_{v_1,c}^{(0)}$ are copied into the corresponding location of RAM0. Besides, the index values of each word of RAM0 are set to 1. In the second round, the temp results $\mathrm{T}_{c,0}^{(0)}$ as shown in Algorithm 4 is stored in RAM1. In the next round, $\mathrm{T}_{c,0}^{(0)}$ will be stored in RAM0. This repeats until $\mathrm{CM}_{c,0}^{(0)}$ is computed.

The computation of $\mathrm{CM}_{c,l+1}^{(k)}$ can be implemented in the same way as the computation of $\mathrm{CM}_{c,0}^{(0)}$. However, extra cycles are spend on testing whether $e_{c,l}^{(k)} = l$ as shown in Algorithm 5. Under the worst condition, $n_m - 1$ cycles are used on the index testing. As shown in Fig. 2.6, a random memory address generator is proposed to eliminate the cycles used in index testing during the computing of $\mathrm{CM}_{c,l+1}^{(k)}$. As a result, only X cycles are needed for the computation of $\mathrm{CM}_{c,l+1}^{(k)}$. Assuming $X = 5$, suppose we need to compute $\mathrm{CM}_{c,1}^{(0)}$, and $\mathrm{CM}_{c,0}^{(0)}$ is stored in RAM0. Suppose $e_{c,0}^{(0)}(1) = e_{c,0}^{(0)}(3) = 0$. Then the RMAG generates a read address sequence (0, 2, 4).

The RMAG first stores a binary sequence $S_l = (s_1, s_2, \cdots, s_X)$, where $s_i = 1$ if $e_{c,l}^{(k)}(i) = l$, otherwise $s_i = 0$. Then, S_l is used to generate the read address sequence for the computation of $\mathrm{CM}_{c,l+1}^{(k)}$. As shown in Fig. 2.6, at the initiation step, S_0 is computed and stored in registers $A_0, A_1, \cdots, A_{X-1}$, which are used to generate read address sequence when computing of $\mathrm{CM}_{c,1}^{(0)}$. Meanwhile, S_1 are stored in $B_0, B_1, \cdots, B_{X-1}$, which are used in the computation of $\mathrm{CM}_{c,2}^{(0)}$. This repeats until the decoding of a codeword is finished. The First One Encoder outputs the smallest i such that $d_i = 1$. The one-hot Encoder (OE) in Fig. 2.6 outputs a binary sequence $(e_0, e_1, \cdots, e_{X-1})$, where $e_x = 1$. $e_j = 0$ if $j \neq x$. Here x is the decimal value of the input of OE.

The proposed path constructor shown in Fig. 2.7 is almost the same as that in [7] except: 1) the size of the CRAM is $q \times w$ instead of $n_m \times w$; 2) the maximum of $r_{c,v}^{(k)}$ is stored in MaxR; 3) part of the hardware in path construct will be used in VNP. These improvements simplify VNP when only part of c-2-v messages are stored. As shown in Fig. 2.7, $r_{c,v}^{(k)}(i)$ is stored in the memory word whose address is $rs_{c,v}^{(k)}(i) \otimes h_{i,j}^{-1}$, where $\otimes$ denotes multiplication over GF(q). According to Algorithm 2,

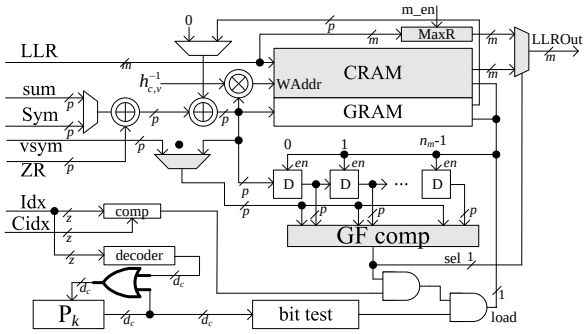

Figure 2.7: Proposed Path Constructor Architecture

only n_m LLRs are generated. So $q - n_m$ words of CRAM are undefined. During the VNP, CRAM outputs $R_{c,v}(s)$ to VNU. If one VNU needs $R_{c,v}(s)$, then the input vsym in Fig. 2.7 equals s. If $s \in rs_{c,v}^{(k)}$ which is stored in n_m p-bit registers, then CRAM(s) is sent to the output. Otherwise, CRAM(s) is not defined and MaxR is sent to the output.

2.4.2 Variable Node Unit Architecture

The proposed VNU architecture is similar to that used in binary LDPC decoder [65]. As shown in Fig. 2.8, suppose the variable node degree is 4, a $w \times q$ RAM (TempRAM in Fig. 2.8) is employed to store channel LLR values in the same way that c-2-v values are stored in the CRAM in Fig. 2.7. For variable node v, the proposed VNU computes the q elements of $Q_{v,c}$ serially. Meanwhile, these q elements are sent to PS which sorts out the n_m minimal LLRs and their corresponding field symbols. The architecture of PS is similar to the sorter proposed in [10], and hence is omitted in this chapter.

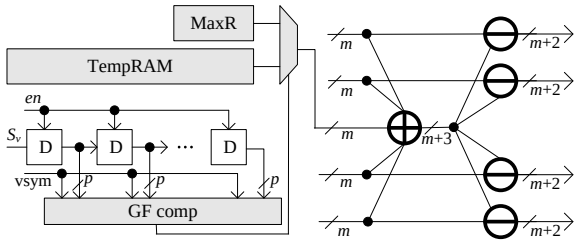

Figure 2.8: Proposed VNU Architecture

2.4.3 Top Decoder Architecture

Considering a nonbinary QC-LDPC code whose parity-check matrix H can be divided into $r \times t$ sub-matrices of dimension $s \times s$. Accordingly, H can be divided into t block columns. The top architecture of the proposed shuffled decoder, shown in Fig. 2.9, is a partly parallel architecture and hence has a similar top structure to other partly parallel architectures (see, for example, [62,65]). $M = r \times s$ CNUs process all rows concurrently. s VNUs process s columns concurrently. Two groups of barrel shifters, BS0 and BS1, implements the interconnection between VNU and CNU. The barrel shifter BS0 has s k-bit inputs and s k-bit outputs, where $k = m + \lceil \log_2 d_v \rceil$. The barrel shifter BS0 has s u-bit inputs and s u-bit outputs, where $u = \max(m, q)$. The channel message RAM has two elements: LLR RAM and its field symbol RAM. When a CNU needs to load messages from channel message RAM, the LLR value and its associated field symbol will travel through BS0 and BS1, respectively.

The decoding schedule of the proposed shuffled decoder is shown in Fig. 2.10. During the initial sort process, the CNU loads channel LLR messages to compute $\mathrm{CM}_{c,0}^{(0)}$. It takes $n_m + (d_c - 1)(X + 1)$ cycles to compute all X elements of $\mathrm{CM}_{c,0}^{(0)}$. Actually, the path construction (PCons) process can start two cycles later once

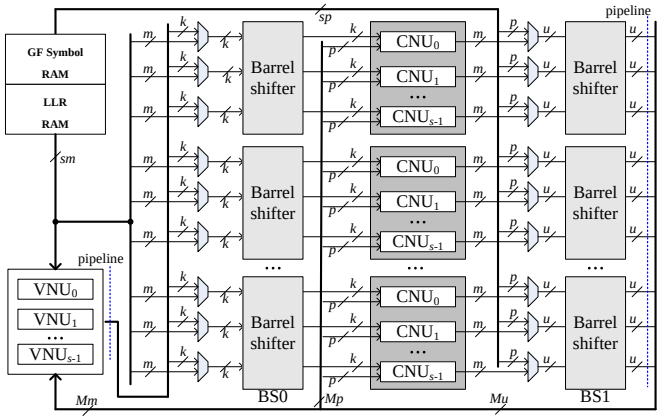

Figure 2.9: Proposed shuffled decoder architecture for NB QC-LDPC codes

$\mathrm{CM}_{c,0}^{(0)}(1)$ is written into RAM0 or RAM1. At the same time, the RMAG will begin to store the indexes compare results (SICR) to register A_i's or B_i's in RMAG once $\mathrm{CM}_{c,0}^{(0)}(1)$ is available. Since the path construction takes $2n_m$ cycles, the total number of cycles used before iteration 1 is $(d_c-2)(X+1)+3n_m+2$. After the initialization process, the shuffled decoder enters into regular iterations. Considering the processing of block column 0 during iteration 1, VNP updates the v-2-c messages within block column 0. The updated v-2-c messages ($q_{v,c}^{(1)}$, $qs_{v,c}^{(1)}$) are stored in the PS. VNP takes q cycles. The shuffle sort (SST) will begin once the VNP is finished. It takes only $1+1.5n_m$ cycles to compute $\mathrm{CM}_{c,1}^{(0)}$, because the RMAG eliminates the cycles used in index comparing. The Pcons process can start two cycles after the SST starts. The number of cycles used for processing one block column then is just $2+2n_m$, because SST and Pcons are conducted at the same time. The processing of the other block columns is the same of that of block column 0. The total number of cycles used for decoding one received word is $(d_c-2)(X+1)+3n_m+2+d_c(2+2n_m+v)I_{max}$.

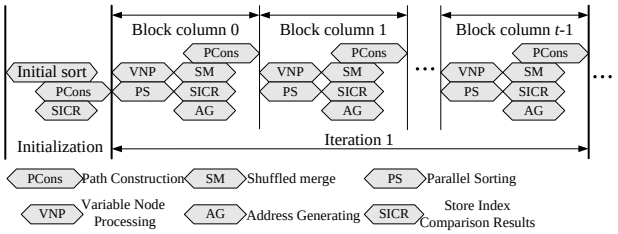

Figure 2.10: Decoding Schedule of Proposed Shuffled Decoder

2.4.4 Implementation Results

A shuffled decoder architecture for an (837,726) QC-LDPC code over GF(2^5) is implemented. n_m and X are set to 16 and 24, respectively, to ensure good decoding performance. Each LLR is represented by $w = 5$ bits with three integer and two fractional bits. This quantization scheme introduces little error performance degradation as shown in Fig. 2.2. Two stages of pipeline registers have been inserted as shown in Fig. 2.9. The memories in CNU and VNU are implemented with register files in order to improve the frequency that the decoder can work at. The decoder is synthesized with Cadence RTL Compiler using an SMIC 130nm library. The synthesis results are summarized in Table 2.1, where N_{cycle} denotes the total number of clock cycles required to decode one received word (assuming 15 iterations). Suppose the clock frequency for a decoder is f MHz, then the corresponding throughput of proposed shuffled decoder is $(fN_bR_{code})/N_{cycle}$, where N_b and R_{code} denote the equivalent code length counted by binary bit and code rate, respectively. The efficiency in Table 2.1 is defined by the throughput-to-gate-count ratio (Mbps/Million gates).

Implementations in [7, 15, 59] for the same (837, 726) nonbinary QC-LDPC code are also shown in Table 2.1 (the results presented in [7] are based not on synthesis

Table 2.1: Decoder Complexity Comparison for an (837, 726) LDPC Code over GF(32)

	[7]	[59]	[15]	proposed
Iterations	15	15	15	15
n_m	16	32	8	16
Schedule	layered	layered	layered	MSS
Process (nm)	N/A	180	90	130
Quantization bits	5	5	7	5
Frequency (MHz)	150	200	260	500
N_{cycle}	62240	53541	N/A	28215
Throughput (Mb/s)	10	16	29.0	64.3
Gate count (NAND)	1.27M	1.37M	3.28M	2.13M
Efficiency (Mbps/Mil gates)	7.87	11.67	8.84	30.18

but on estimation). The efficiency of the proposed decoder architecture is much higher than these previous works. Compare with [59], the efficiency of our work is almost 3 times of that in [59]. Even if the frequency of [59] doubles, our work is still 30% higher. The efficiency of our work is almost 4 times of that in [15] despite a more advanced technology used in [15].

We remark that both the throughput and the efficiency assume that the decoding takes 15 iterations. We adopt this definition for consistency, since the throughput in [7,15,59] is also defined in the same fashion. While this definition reflects the worst case instantaneous throughput of the decoder architecture, it does not account for the convergence behavior of the decoding algorithm if early termination is enabled. For the three codes in Fig. 2.3, the layered schedule reduces the required number of iterations by less than 20% than the MSS. If we were to define the throughput to be proportional to the number of iterations, our decoder architectures would still have better throughput than those in [7, 15, 59].

As shown in Table 2.1, the proposed decoder architecture needs much fewer clock cycles than those in [7, 59] based on the layered schedule. In addition to the reasons discussed in introduction part, the improved throughput is also attributed to two features of the proposed architecture. First, the RMAG reduces the number of cycles used in the shuffled sorting and ensures that the path constructor always gets some $\mathrm{CM}_{c,l}^{(k)}$ without waiting. Second, as shown in Fig. 2.10, part of the path construction and the shuffle sorting process can be carried out simultaneously to increase the throughput.

The proposed (837, 726) ($N_b = 4185$ bits) NB-LDPC decoder achieves a throughput of 64.3 Mb/s and has a 13.28 mm^2 silicon area under 130nm technology, while a (4608, 4096) binary LDPC decoder [65] achieves a throughput of 2.1 Gb/s and has a 1.92 mm^2 silicon area under 65nm technology. This comparison indicates that the decoding complexity and decoder architectures for NB-LDPC codes need to be further improved.

2.5 Conclusion

In this chapter, we propose the shuffled and modified shuffled schedule of the Min-Max decoding algorithm for NB-LDPC codes. Both the shuffled and modified shuffled schedule have a slightly better error performance and converge faster than the flooding schedule. Significantly reducing the complexity of check node processing, the modified shuffled schedule leads to higher throughput and smaller memory requirement while resulting in negligible degeneration in error performance and convergence speed. Moreover, an efficient shuffled decoder architecture based on modified

shuffled schedule for Quasi-Cyclic (QC) LDPC codes is also presented. With improved CNU and VNU, the proposed decoder architecture needs much fewer clock cycles to decoding a received word compared to state of arts design. The implementation of an (837, 726) LDPC decoder over GF(32) demonstrates the efficiency of proposed architecture.

Chapter 3

An Efficient Fully Parallel Decoder Architecture for Non-binary LDPC Codes

3.1 Introduction

Binary low-density parity-check (LDPC) codes are more and more popular in applications because of their capacity-approaching performance. In terms of performance, binary LDPC codes start to show their weaknesses when the codeword length is small or moderate, or when a higher order modulation is used. For these cases, nonbinary LDPC (NB-LDPC) codes over high order Galois fields have shown great potential [3, 4]. For instance, in [1], a rate-1/2 NB-LDPC code of length 84 over GF(64) is shown to perform 0.375dB better than a rate-1/2 binary irregular LDPC code of equivalent length 504 bits in [5] over binary input additive white Gaussian

noise (AWGN) channel. Over the QAM-AWGN channels, NB-LDPC codes with a field order greater than or equal to the size of constellation have the advantage that the encoder/decoder works directly with symbols. All mapping choices of the codeword symbols to the constellation points are equivalent and lead to the same performance. In [1], an NB-LDPC code over GF(256) performs 0.5dB better than a rate-1/2 binary LDPC codes of the equivalent length 1008 bits over QAM256-AWGN channel.

A significant obstacle to the application of NB-LDPC codes is that their decoding algorithms have high complexities. Hence, a lot of research effort has been spent on efficient decoding algorithms for NB-LDPC codes [1,6]. Among them, the EMS [1] and the Min-Max [6] algorithms draw a lot of attention because of their low computation and memory complexity. For an NB-LDPC code over GF(2^m), both the EMS and the Min-Max algorithms store only the n_m ($n_m \ll 2^m$) most reliable messages, thus reducing the memory requirement at the cost of small performance degradation. The check node processing of the EMS algorithm needs additions and comparisons, while the Min-Max algorithm needs only maximizations and comparisons in check node processing. The trellis-based check node processing (TBCP) algorithm [7] reduces the computational complexity of check node processing of the Min-Max algorithm by eliminating unnecessary check-to-variable messages.

Besides the Min-Max and EMS decoding algorithms, stochastic decoding [2,8,9] is another way to reduce the hardware complexity of NB-LDPC decoders while maintaining the decoding performance. Compared to conventional belief propagation decoding algorithms, the stochastic decoding algorithm has lower hardware

complexity [9]. The relaxed half-stochastic decoding algorithm optimized for NB-LPDC codes with variable node degree 2, called the RD2 algorithm, was proposed in [8]. The RD2 decoding algorithm reduces the decoding complexity by reducing the number of real multiplications significantly. An improved version of the RD2 algorithm, called the NoX decoding algorithm [2], is proposed to further reduce the computational complexity.

Recently, a considerable amount of research effort has been spent on efficient decoder architectures for NB-LDPC codes [9–17]. Existing NB-LDPC decoders still suffer from low throughput and large hardware complexity. For example, a (248, 124) NB-LDPC decoder over GF(32) [15] achieves a throughput of 47.69 Mb/s at the cost of 10.33 mm^2 silicon area under 90nm technology. An (837, 726) NB-LDPC decoder [16] over GF(32) achieves a throughput of 60Mb/s at the cost of 1.29M standard NAND gates using the 180nm technology. The FPGA implementation of a (192, 96) stochastic AMSA decoder [9] over GF(256) achieves a throughput of 64Mb/s at the frequency of 108MHz.

In this chapter, several improvements are proposed to reduce both the memory requirements and computational complexities of the Min-Max algorithm. A fully parallel decoder architecture based on the proposed decoding algorithm is also proposed. The main contributions of this chapter are as follows:

1. Based on the Min-Max algorithm, a reduced memory complexity trellis based check node processing (RTBCP) algorithm is proposed.
2. A simplified algorithm is proposed to reduce the computational complexity of variable node processing (VNP). As a result, compared with the RHS algorithm in [2], a stochastic decoder, the proposed decoding algorithm needs

fewer real multiplications but more real comparisons and finite field additions.

3. For each *a priori* message, all LLRs except several most reliable ones are approximated with a linear function. Two kinds of low complexity LLR generation units are also proposed for the approximation of the check-to-variable (c-to-v) LLR and *a priori* LLR, respectively. With 5-bit quantization scheme and $n_m = 32$, the areas of the two LGUs are 10.7% and 13.3%, respectively, of that of an SRAM which stores an LLR vector under a 90nm CMOS technology. A similar approach was proposed in [52] to approximate *a priori* LLR. The main differences between our work and that in [52] are as follows:

   - Besides the approximation of channel LLR vectors, we try to approximate check-to-variable LLR vectors.
   - A simplified variable node processing (SVNP) algorithm is proposed to compensate the performance degradation caused by LLR approximation.

4. A parallel check node unit (CNU) and a low-latency variable node unit (VNU) are proposed. Based on the proposed CNU and VNU, an efficient fully parallel decoder architecture is also proposed. A fully parallel NB-LDPC decoders based on GF(256) is implemented with 28nm CMOS technology. The decoder over GF(256) achieves a throughput of 546Mb/s and an energy efficiency of 0.178nJ/b/iter.

Since routing congestion tends to be challenging for fully parallel LDPC decoder architectures, the proposed decoder architecture is not suitable for very long LDPC codes. The proposed fully parallel decoder architecture is particularly advantageous for NB-LDPC codes over large fields, since the memory reduction will be more

significant when n_m is large.

The rest of this chapter is organized as follows. Section 3.2 reviews the TBCP algorithm. The RTBCP algorithm as well as the simplified variable node processing algorithm is proposed in Section 3.3. The parallel CNU architecture, the low-latency VNU architecture and the fully parallel decoder architecture are proposed in Section 3.4. The implementation results and comparisons are presented in Section 3.5. The conclusions are drawn in Section 5.6.

3.2 TBCP algorithm

3.2.1 Trellis based check node processing algorithm

Let GF(2^m) be a Galois field with 2^m elements. Let $\mathbf{H} = \{h_{i,j}\}$ be a sparse parity check matrix over GF(2^m) with M rows and N columns. We focus on regular non-binary LDPC codes, and hence assume $\mathbf{H}$ has constant row and column weights and is an array of sparse circulants over GF(2^m). Consider a check node c and a variable node v in the Tanner graph defined by $\mathbf{H}$. Let $\varepsilon(v)$ denote the set of check nodes adjacent to v, and $\tau(c)$ the set of variable nodes adjacent to c. For $a \in$ GF(2^m), let $L_v(a)$ be the *a priori* information of the variable node v concerning the symbol a.

For an LDPC code over GF(2^m), check node processing is the most complex part of the Min-Max algorithm [6, 7]. The forward-backward approach [6] is widely used in check node processing. However, both memory complexity and latency are high for the forward-backward approach when the check node degree is high. In [7], a TBCP algorithm is proposed to reduce the memory required by check

node processing. Let d_c be the check node degree of a check node c. The truncated variable-to-check (v-to-c) message from a variable node v to a check node c is $(\boldsymbol{\phi}_{v,c},\ \boldsymbol{\phi}^f_{v,c})$, where $\boldsymbol{\phi}_{v,c}$ is an n_m-dimension LLR vector and $\boldsymbol{\phi}^f_{v,c}$ is the associated n_m-dimension Galois field symbol vector. For the Min-Max algorithm, the first element of $\boldsymbol{\phi}_{v,c}$, $\phi_{v,c}(0)$, is always zero. The TBCP algorithm [7] first merges the incoming $d_c(n_m-1)$ nonzero LLRs in non-decreasing order as $x_c(0), x_c(1), \cdots$. Their associated Galois field symbols are $\alpha_c(0), \alpha_c(1), \cdots$, and they belong to variable nodes with indices $e_c(0), e_c(1), \cdots$. For $1 \le X \le d_c(n_m-1)$, a truncated message vector $\mathbf{m}_c = \{m_c(0), m_c(1), \cdots, m_c(X-1)\}$, where $m_c(i) = (x_c(i), \alpha_c(i), e_c(i))$ for $i = 0, 1, \cdots, X-1$, is used by the path construction (PC) algorithm [7], shown in Algorithm 6, to compute the updated c-to-v message $(\boldsymbol{\rho}_{c,v}, \boldsymbol{\rho}^f_{c,v})$.

Algorithm 6: Path Construction Algorithm [7]

input : $m_c(i)\ i = 0, 1, \cdots, X-1$; z_c; α_{sum}
output: $\rho_{c,v}(j), \rho^f_{c,v}(j)$ for $j = 0, \cdots, n_m - 1$
Initialization:
$\rho_{c,v}(0) = 0, \rho^f_{c,v}(0) = \alpha_{sum} \oplus z_c(v), i = 0, cnt = 1, \mathbf{P}_{c,0} = [0, 0, \cdots, 0]$
while $cnt < n_m$ **do**
  if $e_c(i) \neq v$ **then**
    $j = cnt$
    for $k = 0$ **to** $j - 1$ **do**
      $\alpha = \rho^f_{c,v}(k) \oplus z_c(e_c(i)) \oplus \alpha_c(i)$
      if $P_{c,k}(e_c(i)) \neq 1$ **and** $\alpha \notin \boldsymbol{\rho}^f_{c,t}$ **then**
        $\rho_{c,v}(cnt) = x_c(i); \rho^f_{c,v}(cnt) = \alpha$
        $P_{c,cnt}(e_c(i)) = 1$
        $P_{c,cnt}(s) = P_{c,k}(s)$ for $s \neq e_c(i)$
        $cnt = cnt + 1$
  $i = i + 1$

As shown in Algorithm 6, $z_c(v) = \phi^f_{v,c}(0)$ for $v \in \tau(c) = \{v_0, v_1, \cdots, v_{d_c-1}\}$ and

$\alpha_{sum} = \sum_{v=v_0}^{v_{d_c-1}} z_c(v)$. Algorithm 6 assumes a check node c and deals with variable node $v \in \tau(c)$. $\mathbf{P}_{c,cnt}$, which stores the constructed path, is a d_c-dimension vector over GF(2). $\oplus$ denotes addition over Galois field GF(2^m). The constructed n_m LLRs are picked from the sorted list $x_c(i)$, and stored in $\boldsymbol{\rho}_{c,v}$. Their associated Galois field symbols are stored in $\boldsymbol{\rho}^f_{c,v}$. X is set to $1.5n_m$ [7] so that at least n_m LLRs are generated for $\boldsymbol{\rho}_{c,v}$. Suppose L is the number of times that the codes within the for loop (lines 6 to 11) in Algorithm 6 are executed. In [7], $L = 2n_m$.

The TBCP algorithm [7] computes all n_m elements of a c-to-v message in serial. The LLR and corresponding Galois field symbol of the most reliable element are computed. The corresponding path information is also stored. Based on the path information of the first element, the TBCP algorithm computes the LLR and Galois field symbol of the second most reliable element. The third element is computed based on the path information of the previous two elements. The process repeats until all n_m elements of a c-to-v message are computed. The LLR vector of a c-to-v message is sorted in non-decreasing order once all n_m elements are computed.

3.3 Improved Decoding Algorithm for NB-LDPC Codes

3.3.1 RTBCP algorithm

In this section, a reduced complexity trellis based check node processing (RTBCP) algorithm is proposed to reduce the complexity of check node processing further. For a check node c, by observing the X LLR magnitudes of $\mathbf{m}_c$, it is found that these

LLRs come from the first few elements of all d_c connected v-to-c messages. As a result, one can store only n_v ($n_v < n_m$) elements for each truncated v-to-c message. Usually, n_v should satisfy that $d_c n_v > X$ where d_c is the check node degree. The detailed value of n_v can be determined by performance simulation. When d_c is large, n_v could be much smaller than n_m. Since only n_v elements are needed, the memory for v-to-c messages can be further reduced. Besides, normally, a length n_m parallel sorter [10] is used to store the n_m minimum LLRs and sort them in non-decreasing order. It will need a length n_v parallel sorter if only n_v elements are stored. Thus, the overall hardware cost is further reduced.

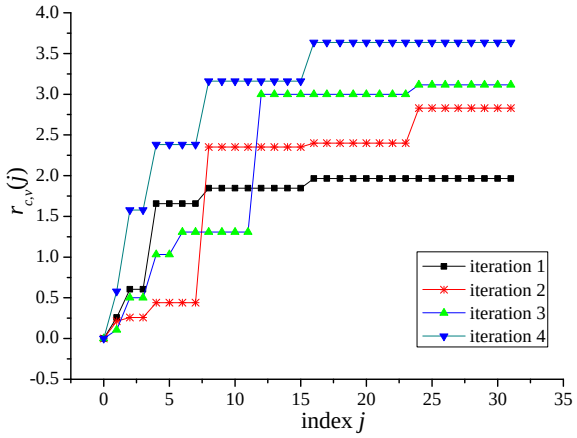

Figure 3.1: LLR evolution

Studying the magnitudes among the truncated n_m LLRs of a c-to-v message, it is found that the LLR vector can be approximated using a piece-wise linear function. In Fig. 3.1, we plot the LLR evolution of a randomly picked c-to-v message during the decoding of a (110, 88) QC-LDPC code over GF(256) [66] under BPSK-AWGN channel. The signal to noise ratio (SNR) is 3.9dB. These LLRs are computed using the PC algorithm in [7] with $n_m = 32$. As shown in Fig. 3.1, these LLRs demonstrate

piece-wise linearity during each iteration (a similar behavior is also demonstrated by the LLRs for other SNR values).

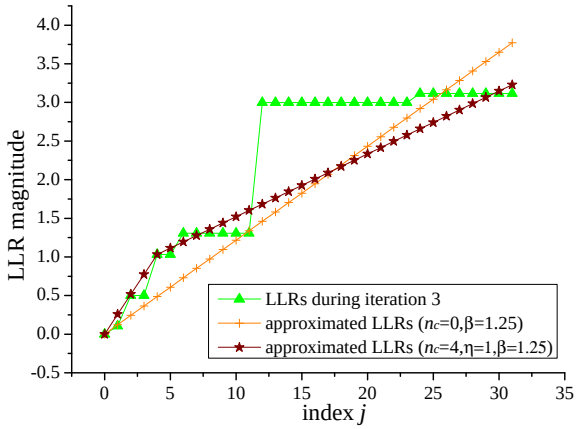

Figure 3.2: LLR approximation when $n_m = 32$

Since the n_m LLRs of $\boldsymbol{\rho}_{c,v}$ are generated in serial in Algorithm 6, we propose to store only $\theta_{c,v} = \rho_{c,v}(n_m - 1)$ and $M_{c,v} = \rho_{c,v}(n_c)$ during the while loop in Algorithm 6, where n_c is a parameter that can be determined by performance simulation. Once the while loop is finished, the approximated LLR vector are computed with the piece-wise linear function shown in Eq. (3.1), where η and β are scaling parameters to make the piece-wise interpolation more accurate. As shown in Fig. 3.2, we approximate all the elements of an LLR vector using a piece-wise linear function:

$$\hat{\rho}_{c,v}(j) = \begin{cases} \eta \frac{M_{c,v}}{F(n_c)} j & j \leq n_c \\ M_{c,v} + \beta \frac{\theta_{c,v} - M_{c,v}}{F(n_m)} (j - n_c) & j > n_c, \end{cases} \tag{3.1}$$

where $F(x) = 2^{\lceil \log_2 x \rceil}$. Note that the divisions in Eq. (3.1) can be implemented with bit shifting. If $n_c = 0$, then all LLRs are approximated with a linear function. When an LLR element is needed, it is computed using Eq. (3.1). This reduces the

memory required to store the LLR vector for c-to-v messages.

Based on the above discussions, an improved path construction (IPC) algorithm is also proposed as part of the RTBCP algorithm. The IPC algorithm differs from the PC algorithm in [7] at two aspects. First, the if block in Algorithm 6 (lines 7 to 11) is changed to that shown in Algorithm 7, where each path $p_{c,cnt}$ is represented by an integer index instead of a d_c-dimension binary vector. Besides, $p_{c,0} = d_c$ for the proposed IPC algorithm. For hardware implementations of the proposed IPC algorithm, it needs only t bits to store an integer index, where $t = \log_2 d_c + 1$ if d_c is a power of 2, and $t = \lceil \log_2 d_c \rceil$ if d_c is not a power of 2. Besides, the process to decide whether $x_c(i)$ is an element of $\boldsymbol{\rho}_{c,v}$ is also simplified. In addition, the number of elements in the truncated message vector $\mathbf{m}_c$ used in Algorithm 6, X, is reduced to n_m. For the proposed IPC algorithm, the number of loops of Algorithm 6, L, can be smaller than $2n_m$. Second, when the while loop of Algorithm 6 is finished, all n_m LLR elements are computed using Eq. (3.1).

Algorithm 7: Improved **if** block

if $p_{c,k} \neq e_c(i)$ **and** $\alpha \notin \boldsymbol{\rho}^f_{c,t}$ **then**
  if $cnt = n_c$ **then** $M_{c,v} = x_c(i)$
  $\theta_{c,v} = x_c(i); \rho^f_{c,v}(cnt) = \alpha$
  $p_{c,cnt} = e_c(i); cnt = cnt + 1$

3.3.2 LLR compression for *a priori* messages

Similar to the piece-wise linear approximation of the LLR vector of a c-to-v message, part of the LLR vector of a *a priori* message can also be approximated by its linear interpolation. Let $\mathbf{L}_v = (L_v(0), L_v(1), \cdots, L_v(n_m - 1))$ be the sorted LLR

vector estimated from the channel for a variable node v. The approximated LLR vector is $\hat{\mathbf{L}}_v = (\hat{L}_v(0), \hat{L}_v(1), \cdots, \hat{L}_v(n_m - 1))$, where $\hat{L}_v(k) = L_c(k)$ for $k \leq n_I$ and $\hat{L}_v(k) = L_c(n_I) + \Delta_v(k - n_I)$ for $k > n_I$. Here $\Delta_v = \beta_v \frac{L_v(n_m-1) - L_v(n_I)}{F(n_m)}$ and β_v is a scaling parameter. For hardware implementations, only n_I ($n_I < n_m$) LLRs and Δ_v are stored.

3.3.3 Simplified variable node processing algorithm

In this chapter, a simplified variable node processing (SVNP) algorithm is proposed in Algorithm 8, where d_v is the degree for a variable node v, and $(\boldsymbol{\rho}_{i,v}, \boldsymbol{\rho}^f_{i,v})$ for $i = 0, 1, \cdots, d_v - 1$ are d_v c-to-v messages sent to variable node v. $\boldsymbol{\rho}_{d_v,v} = \hat{\mathbf{L}}_v$, and $\boldsymbol{\rho}^f_{d_v,v} = \hat{\mathbf{L}}^f_v$, where $\hat{\mathbf{L}}^f_v$ is the corresponding Galois field symbol vector of the *a priori* LLR vector $\hat{\mathbf{L}}_v$ for a variable node v. $h_{0,v}, h_{1,v}, \cdots, h_{d_c-1,v}$ are d_v nonzero Galois field symbols associated with variable node v.

The FindLLR function returns $\rho_{w,v}(k)$ such that $h_{w,v}F_{i,j} = \rho^f_{w,v}(k)$ for $0 \leq w < d_c$. For $w = d_c$, the FindLLR function returns $\rho_{w,v}(k)$ such that $F_{i,j} = \rho^f_{w,v}(k)$. When $w < d_c$, if $h_{w,v}F_{i,j} \notin \boldsymbol{\rho}^f_{w,v}$, the FindLLR function returns $\gamma\theta_{w,v}$, where γ is a correction factor and $\theta_{w,v}$ is defined above. When $w = d_v$, if $F_{i,j} \notin \boldsymbol{\rho}^f_{w,v}$, $\theta_{w,v} = \hat{L}_v(n_m - 1)$. l_i for $i = 0, 1, \cdots, d_v$ are $d_v + 1$ integer parameters. The SORT function sorts $R_{i,v}$, which has at most $l_{sum} = \sum_{i=0}^{d_v} l_i$ ($l_{sum} \leq n_m$) LLR elements, in non-decreasing order and stores the n_v minimal LLRs and their corresponding Galois field symbols in $\boldsymbol{\phi}_{v,i}$ and $\boldsymbol{\phi}^f_{v,i}$, respectively.

The proposed SVNP algorithm first serially computes at most l_{sum} elements of a v-to-c message. Among them, only $n_v (n_v < n_m)$ most reliable elements are stored. Thus, both the memory requirement and the computation complexity are reduced.

Algorithm 8: SVNP algorithm

input : $(\boldsymbol{\rho}_{0,v}, \boldsymbol{\rho}^f_{0,v}), \cdots, (\boldsymbol{\rho}_{d_v,v}, \boldsymbol{\rho}^f_{d_v,v})$; $l_i, i = 0, 1, \cdots, d_v$;
$h_{i,v}, i = 0, 1, \cdots, d_v - 1$
output: $(\boldsymbol{\phi}_{v,i}, \boldsymbol{\phi}^f_{v,i}), i = 0, 1, \cdots, d_v - 1$
Initialization: $S_v = \emptyset$; $R_{i,v} = \emptyset, RS_{i,v} = \emptyset, i = 0, 1, \cdots, d_v - 1$;

```
for i = 0 to d_v do
    for j = 0 to l_i − 1 do
        if i < d_v then F_{i,j} = h^{-1}_{i,v} ρ^f_{i,v}(j) ;
        else F_{i,j} = ρ^f_{i,v}(j);
        if F_{i,j} ∉ S_v then
            push F_{i,j} into S_v;
            for w = 0 to d_v do
                t_w =FindLLR(F_{i,j}, ρ_{w,v}, ρ^f_{w,v});
            for w = 0 to d_v − 1 do
                push ((Σ_{b=0}^{d_v} t_b) − t_w) into R_{w,v};
                push F_{i,j} into RS_{w,v};
for i = 0 to d_v − 1 do
    (φ_{v,i}, φ^f_{v,i}) = SORT(R_{i,v}, RS_{i,v});
```

For the VNP algorithm in [1], all the elements of the incoming c-to-v messages are needed for a round of variable node processing. However, for each incoming c-to-v message, only part of its elements are needed for variable node processing when using the SVNP algorithm.

For a round of variable node processing of a variable node v, the computational complexities of the proposed SVNP algorithm and the VNP algorithm in [1] are dominated by real value additions and comparisons. Hence, we compare the numbers of real additions and comparisons of these two algorithms in Table 3.1. Since $n_v < n_m$ and $l_{sum} \leq n_m$, the numbers of real comparisons and additions of the VNP algorithm in [1] are more than $(3 - \frac{4}{d_v})\frac{n_m}{n_v}$ and $3 - \frac{4}{d_v}$ ($\frac{3d_v-4}{d_v} > 1$ when $d_v > 2$) times,

respectively, of those of the proposed SVNP algorithm.

Table 3.1: Computational complexity comparison between the proposed SVNP and the VNP algorithm in [1]

	[1]	SVNP
real comparisons	$(3d_v - 4)n_m \log_2(2n_m)$	$d_v n_v \log_2 l_{sum}$
real additions	$(3d_v - 4)2n_m$	$d_v 2l_{sum}$

The complexity reduction brought by the SVNP algorithm is not obvious according to Table 3.1. This is because $(3d_v - 4) = d_v$ when $d_v = 2$. Under this condition, the advantage of the SVNP algorithm depends on the specific values of n_v and l_{sum}, which in turn are selected by performance simulation. Thus, when $d_v = 2$ the advantage of SVNP algorithm may be code specific.

In Table 3.2, we compare the computational complexity per decoding iteration of the proposed improved decoding algorithm (IDA) with that of the RHS algorithm in [2], a stochastic decoder. For the proposed IDA, the computational complexity of the proposed IPC algorithm depends on the v-to-c messages. To be conservative, the maximal computational complexity of the IPC algorithm is assumed. As shown in Table 3.2, compared with the RHS algorithm, the proposed IDA needs fewer real multiplications but more real comparisons and additions over GF(2^m). When $l_{sum} < 2^m$, the proposed IDA needs fewer real additions than the RHS algorithm. l_{sum} could be smaller than 2^m, as its value is determined by the decoding performance simulation.

3.3.4 Numerical results

We compare the error performance of the proposed IPC and SVNP algorithms with that of the original PC and VNP algorithm for a (110, 88) NB-LDPC code over

Table 3.2: Computational complexity comparison between the improved decoding algorithm and the RHS algorithm in [2]

	RHS [2]	proposed IDA
real comparisons	0	$Nd_v n_v \log_2 l_{sum} + 2Md_c n_m^2$
real additions	$2Nd_v 2^m$	$2Nd_v l_{sum}$
real multiplications	$N(3d_v - 4)2^m$	$Md_c(n_m - 2)$
GF(2^m) additions	$M(2d_c - 2)$	$2Md_c n_m^2$

GF(256) [66] with variable node degree 2 and check node degree 10 under the BPSK-AWGN channel. In our simulations, $n_m = 32$, the maximal number of iterations is 30, and the flooding schedule is used. The resulting bit error rate (BER) and frame error rate (FER) are shown in Fig. 3.3 and Fig. 3.4, respectively, where PC-VNP denotes the decoding algorithm with the original PC algorithm and variable node processing algorithm in [1].

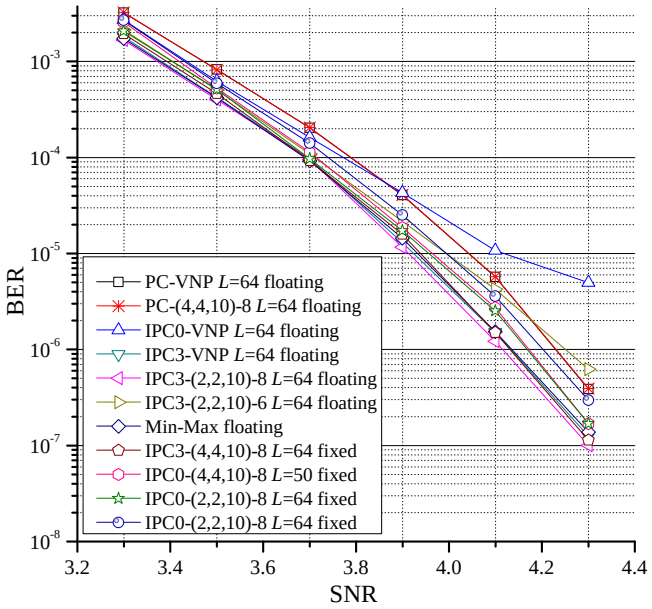

Figure 3.3: BER performance of the (110, 88) NB-LDPC code over GF(256)

In Fig. 3.3, L denotes the number of loops required by the corresponding IPC

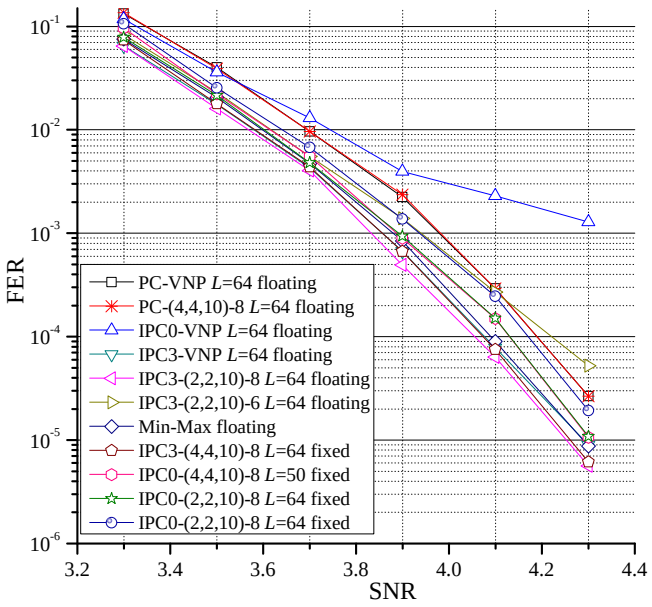

Figure 3.4: FER performance of the (110, 88) NB-LDPC code over GF(256)

algorithm. PC-(i, j, k)-ω denotes the decoding algorithm with the original PC algorithm and proposed SVNP algorithm with $l_0 = i$, $l_1 = j$, $l_2 = k$ and $n_v = \omega$. IPCe-VNP denotes the decoding algorithm that employs the proposed IPC algorithm with $n_c = e$ and the VNP algorithm in [1]. IPCe-(i, j, k)-ω denotes the decoding algorithm with the proposed IPC algorithm with $n_c = e$ and the proposed SVNP algorithm with $l_0 = i$, $l_1 = j$, $l_2 = k$ and $n_v = \omega$. For IPC0, $\beta = 1.25$. For IPC3, $\eta = 1.25, \beta = 1.75$. $\gamma = 1.25$ for all simulated algorithms. For all IPCe-(i, j, k)-ω algorithms, part of each *a priori* LLR messages are linearly approximated with $n_I = 4$ and $\beta_v = 1$. For fixed point simulations, a (4,1) quantization scheme is used, where four bits and one bit are used to represent the integer and fraction parts of an LLR, respectively.

Based on the results shown in Fig. 3.3, several observations are made as follows:

1. The BER performance of the PC-VNP and PC-(4, 4, 10)-8 are nearly the same.

The proposed SVNP algorithm does not introduce noticeable performance degradation.

2. The IPC0-VNP decoding algorithm shows an early error floor, while the BER performance of the IPC3-VNP is even better than that of the PC-VNP algorithm.
3. The BER performance of the IPC3-(2, 2, 15)-8 is better than that of the PC-VNP and IPC0-(2, 2, 10)-8 decoding algorithms. The decoding performance of IPC3-(2, 2, 15)-8 is close to that of original Min-Max floating algorithm.
4. The IPC0-(2, 2, 10)-8 decoding algorithm performs better than the PC-VNP and IPC0-VNP decoding algorithms.
5. Reducing the number of loops used by the IPC algorithm will worsen the corresponding decoding performance.
6. For decoding algorithms that employ the SVNP algorithm, n_v should be large enough to maintain decoding performance.

Both the IPC3-VNP and IPC3-(2, 2, 15)-8 decoding algorithms perform better than the PC-VNP algorithm. Since the propose LLR approximation can also be viewed as a non-uniform scaling technique, the improved decoding performance can be attributed to the non-uniform scaling of each element in an LLR vector. As is well known, for binary LDPC codes, the scaling technique has been used to improve the decoding performance of the Min-Sum algorithm [67].

The early error floor of the IPC0-VNP decoding algorithm may come from the inaccurate LLR approximation of the IPC0 algorithm. The performance of the IPC0-VNP algorithm is improved when the VNP algorithm is replaced with the proposed SVNP algorithm. On the other hand, the performance of the IPC3-VNP

algorithm and the IPC3-(2, 2, 15)-8 algorithm are almost the same. With proper value for each l_i, it seems that the SVNP is less sensitive to the LLR deviation caused by the LLR approximation of the IPC0 algorithm. A conceptual explanation is shown as follows.

Take the IPC0-(2, 2, 10)-8 decoding algorithm as an example, where $l_0 = 2$, $l_1 = 2$ and $l_2 = 10$. It is possible that the transmitted code symbol for variable node v lies in the first several elements in $\hat{\mathbf{L}}_v^f$, which is the corresponding Galois field symbol vector of the sorted LLR vector $\hat{\mathbf{L}}_v$. For IPC0 algorithm, the LLR approximation is not accurate enough. As a result, during variable node processing, the SVNP algorithm considers only the few most reliable symbols and their associated LLRs for each c-to-v message sent to variable node v. For these LLRs, the approximation error tends to be small. On the other hand, the SVNP algorithm considers more symbols from $\hat{\mathbf{L}}_v^f$. In this way, symbols with approximated LLRs, which may degrade the decoding performance, are excluded from variable node processing.

The proposed IPC and SVNP algorithms are also applied to a (372, 248) ($d_v = 4$) quasi-cyclic NB-LDPC (QC-NB-LDPC) code over GF(32) [68]. For both PC and IPC algorithms, $n_m = 8$. For IPC0-(1, 1, 1, 1, 6)-5 and IPC1-(1, 1, 1, 1, 6)-5 algorithms, $n_v = 5, n_I = 4$. As shown in Fig. 3.5, the BER performance of the IPC0-(1, 1, 1, 1, 6)-5 and IPC1-(1, 1, 1, 1, 6)-5 algorithms is nearly the same as that of the PC-VNP algorithm. Compared to original Min-Max floating algorithm, IPC1-(1, 1, 1, 1, 6)-5 has 0.1dB performance degradation. The BER and FER performance is shown in Fig. 3.5 and Fig. 3.6. For the fixed point simulation, a (4,1) quantization scheme is used, where four bits and one bit are used to represent the integer and faction parts of an LLR, respectively. For the (372, 248) code, the

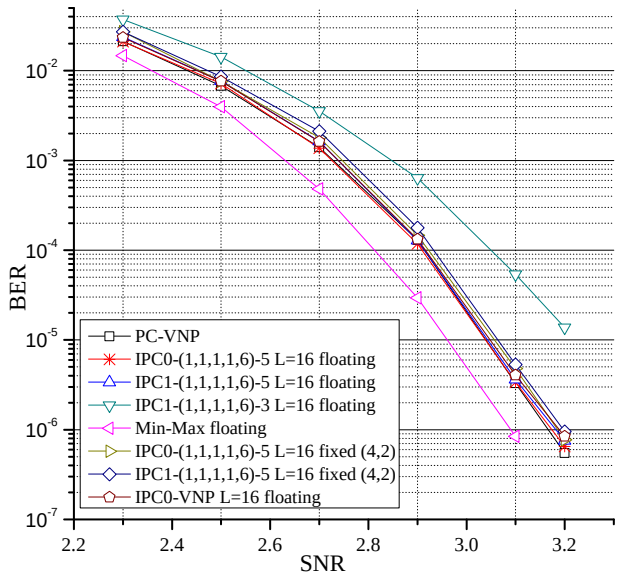

Figure 3.5: BER performance of the (372, 248) NB-LDPC code over GF(32)

IPC0-VNP algorithm does not show an early error floor.

3.4 Fully Parallel Decoder Architecture

3.4.1 Top decoder architecture

Suppose there are M rows and N columns in the parity-check matrix. As shown in Fig. 5.5, the proposed decoder architecture employs M CNUs and N VNUs. The main characteristics of proposed fully parallel decoder architecture are as follows:

1. Check node processing and variable node processing are interleaved. During check (variable, respectively) node processing, all rows (columns, respectively) of the parity check matrix are processed simultaneously.

2. For both c-to-v and v-to-c messages, each message element is transmitted in

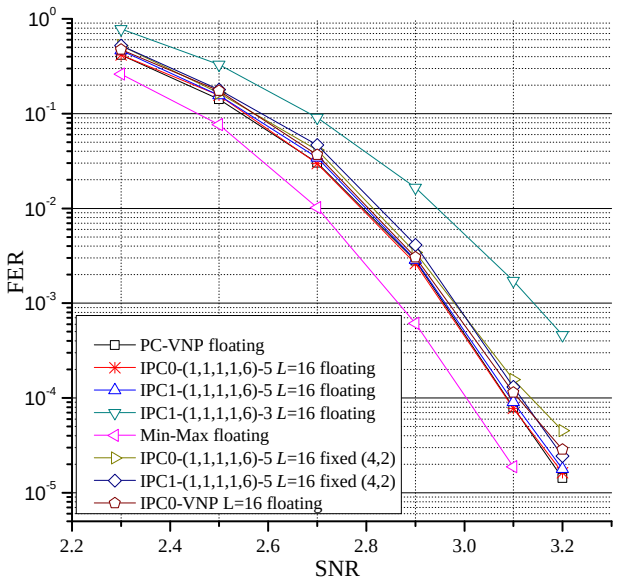

Figure 3.6: FER performance of the (372, 248) NB-LDPC code over GF(32)

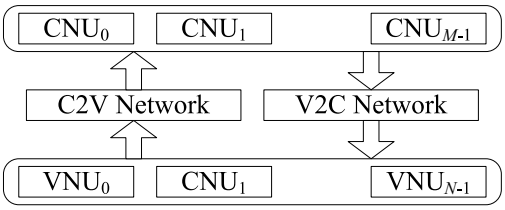

Figure 3.7: Proposed fully parallel decoder architecture

serial via the CNU-to-VNU (C2V) or VNU-to-CNU (V2C) interconnection networks, respectively. The C2V and V2C networks can be hard wires to connect CNUs and VNUs. If multiple quasi-cyclic NB-LDPC (QC-NB-LDPC) codes need to be supported, barrel shifters can be used instead.

3. For each c-to-v message, the LLR vector, which has n_m LLR elements, is not stored. Instead, only one or two LLRs are stored, and the others are computed on-the-fly.

4. The proposed fully parallel decoder architecture is suitable for moderate or

short length (around 10^3 bits) NB-LDPC codes over large fields.

3.4.2 Parallel CNU architecture

In this chapter, a parallel check node unit (CNU) is proposed for the proposed fully parallel decoder architecture. The top architecture of the proposed CNU is shown in Fig. 3.8, where m is the bit width of a Galois symbol, p is the number of the quantization bits, $r = \lceil \log_2 d_v \rceil + p$ is the number of the quantization bits of the LLRs sent to CNU, t is the bit width of the index. For a check node c, the proposed CNU is capable of computing all the corresponding c-to-v messages, which are sent from c, in parallel. For each c-to-v message, the n_m elements are computed in serial.

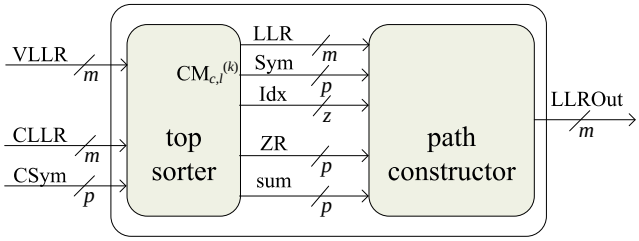

Figure 3.8: Parallel CNU architecture

As shown in Fig. 3.8, the parallel CNU for a check node c consists of d_c parallel sorters (PS), d_c path constructors (PC), a minimal LLR finder (MLF) and a truncated message register group (TMR). The proposed parallel CNU engages in both check node processing and variable node processing. During variable node processing, d_c v-to-c messages are sent to CNU in parallel. Take PS_i as an example, it receives each element of the incoming v-to-c message in serial. Meanwhile, PS_i sorts out the n_v minimum LLRs and their corresponding Galois field symbols from the received v-to-c message. During check node processing, the MLF unit computes the

truncated message vector $\mathbf{m}_c$. The X elements of $\mathbf{m}_c$ are computed in serial and stored in the TMR. At the same time, d_c path constructors compute d_c updated c-to-v messages in parallel based on the proposed IPC algorithm. The computing of $\mathbf{m}_c$ and d_c c-to-v messages are performed at the same time.

The architecture of the parallel sorter PS_i, shown in Fig. 3.9, is similar to that in [10]. The n_v sorted LLRs and their corresponding Galois field symbols are stored in LLR registers (LR) and symbol registers (SR), respectively. As shown in Fig. 3.9, the mode signal configures the function of the PS. During variable node processing, the PS acts as the parallel sorter in [10]. However, the PS acts as a shift register group during check node processing. During check node processing, if shift_i is enabled, the shifting operations of LR_i and SR_i are defined as $L_{i,j} = L_{i,j+1}$ and $S_{i,j} = S_{i,j+1}$, respectively, for $j = 1, 2, \cdots, n_v - 2$.

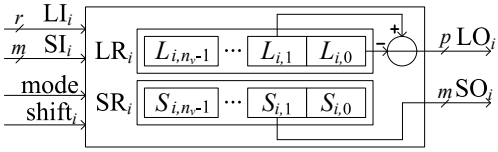

Figure 3.9: Parallel sorter architecture

The minimal LLR finder (MLF) unit computes the minimal LLR, the corresponding Galois field symbol and index based on its inputs messages. Suppose the number of input messages is 8, the macro architecture of the MLF unit is shown in Fig. 3.10. Each input message I_i in Fig. 3.10 consists of three parts: the LLR LO_i, the corresponding Galois field symbol SO_i, and the associated index idx_i. The MLF unit outputs the input message with the smallest LLR. The MLF unit is a tree of compare-and-select (CAS) units. Each CAS unit has two input messages and outputs the one with a smaller LLR.

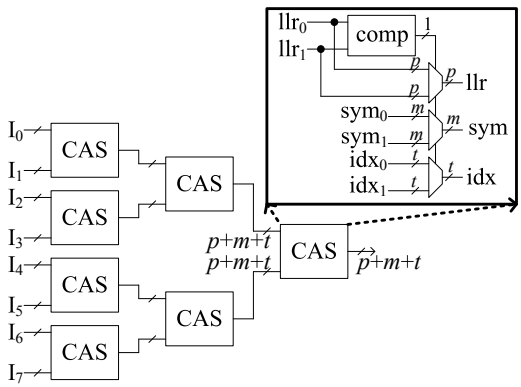

Figure 3.10: The macro architecture of MLF when the number of input is 8

It takes X cycles for the proposed MLF to compute all X elements of the truncated message vector $\mathbf{m}_c$. Once the minimal n_v LLRs are sorted out, $m_c(0) = (x_c(0), \alpha_c(0), e_c(0))$ is computed by the MLF unit. Meanwhile, if the output index $e_c(0) = k(0 \leq k < d_c - 1)$, then shift_k is enabled to shift both the LR_k and SR_k by one step at the next clock. Once LR_k and SR_k have been shifted, the MLF unit computes $m_c(1)$ and generates the corresponding shift signal. This repeats until all X elements of $\mathbf{m}_c$ are generated. Once an element of $\mathbf{m}_c$ is computed, it is stored in the TMR, which consists of n_m $(p + m + t)$-bit registers. The truncated message element $m_c(i)$ is stored in the i-th location. All the PC units within a CNU can access the message stored in any location of the TMR simultaneously.

The path constructor unit computes updated c-to-v message $(\boldsymbol{\rho}_{c,v}, \boldsymbol{\rho}^f_{c,v})$ based on the propose IPC algorithm. Take PC_i as an example, the architecture of the proposed PC unit is shown in Fig. 3.11, where sum $= \sum_{i=0}^{d_c-1} S_{i,0}$, and $s = \lceil \log_2 n_m \rceil$ is the width of the write and read address for the symbol register group (SRG) with n_m total locations. During the computing of the first element of a c-to-v message, the initial load (IL) signal equals 1 and ZS $= S_{i,0}$. For the computing of other

elements, ZS = $S_{I,0}$, where $I = e_c(j_i)$, and j_i is the location index of the truncated message element in the TMR that is read by PC_i. The index register group (IRG), which contains n_m t-bit registers, stores n_m paths: $p_{i,0}, p_{i,1}, \cdots, p_{i,n_m-1}$. Each path is represented by a t-bit integer as shown in Algorithm 7. The SRG, which contains n_m m-bit registers, stores the Galois field symbol vector, $\boldsymbol{\rho}_{c,v}^f$. The SRG also tests whether the input symbol (symIn_i) has already been stored. The proposed PC unit reads the truncated messages from the TMR and computes the corresponding updated c-to-v message in $2n_m$ cycles. Compared to the path constructor in [7], the hardware complexity of proposed PC unit is reduced for several reasons:

1. The memory used to store all n_m paths is reduced. In [7], it needs $n_m d_c$ bits to store all n_m paths. However, the proposed PC needs only $n_m t$ bits, where $t = \log_2 d_c + 1$ if d_c is a power of 2, and $t = \lceil \log_2 d_c \rceil$ if d_c is not a power of 2.

2. It is easier to determine whether $x_c(j_i)$ is an LLR element of the updated c-to-v message. As shown in Fig. 3.11, it only needs to compare whether two indices are the same. In contrast, the PC unit in [7] needs to first encode the t-bit index into a d_c-bit binary sequence and then do the bit-test operation.

3. Due to LLR approximation, the proposed PC does not need an LLR RAM to store the LLR vector of the updated c-to-v message. The PC shown in Fig. 3.11 adopts the approximation scheme with $n_c = 0$. Thus, only the maximal LLR $\theta_{c,v}$ is stored in the p-bit register, MR, shown in Fig. 3.11.

The SRG unit in CNU is moved to VNU in order to avoid the global connection wires between different CNUs. This will be discussed in Sections 3.4.3 and 3.4.4.

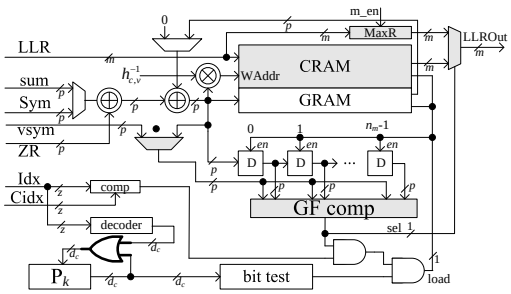

Figure 3.11: Path constructor architecture

For the proposed CNU, all PC units can access any element of the TMR simultaneously. The check node processing can be initiated once the n_v LLR elements are sorted out. From $m_c(0)$, each PC unit processes at most one element of $\mathbf{m}_c$ sequentially during each cycle. On the other hand, The MLF unit generates one new element of $\mathbf{m}_c$ during each cycle. As a result, the PC units and MLF unit can work simultaneously. Thus, it takes $L(L \leq 2n_m)$ cycles to finish a round of check node processing.

3.4.3 Low-latency VNU architecture

Many previous works [7, 15] on the NB-LDPC decoder architecture focus on the simplification of check node processing. However, variable node processing of current NB-LDPC decoders has not been carefully examined. The variable node processing for one layer takes $2n_m$ cycles for the VNUs in [7,15]. A low latency VNP algorithm is proposed in [69] to reduce the cycles from $2n_m$ to $L_{S-VN}+n_m$, where $L_{S-VN} < n_m$. However, the VNP algorithm in [69] works for the layered schedule. Usually, the variable node processing for different layers are performed in serial, which leads to reduced throughput.

In this chapter, based on the proposed SVNP algorithm, a low-latency VNU architecture is proposed to reduce the number of cycles used by variable node processing. The proposed low-latency VNU architecture works for the flooding or shuffled schedule. During an iteration, it takes only l_{sum} cycles to finish variable node processing for each variable node. For each c-to-v message sent to a variable node v, only a fraction of the n_m elements are used in variable node processing. Besides, it still needs a fraction of the n_m elements of each channel message during variable node processing.

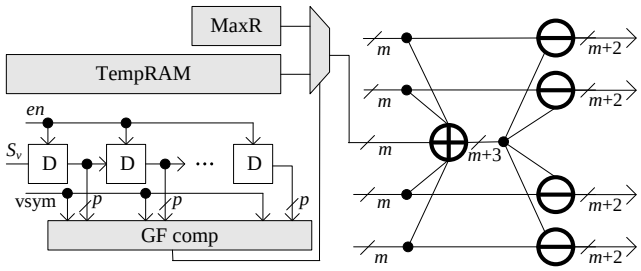

Figure 3.12: VNU architecture assuming $d_v = 2$

For a variable node v, suppose the variable node degree $d_v = 2$, the proposed VNU architecture is shown in Fig. 3.12. The VNU architecture for other d_v values is similar. The proposed VNU computes d_v temporary v-to-c messages ($R_{w,v}$, $RS_{w,v}$) for $w = 0, 1, \cdots, d_v - 1$ as shown in Algorithm 8. Each temporary v-to-c message has at most l_{sum} elements, which are serially sent to the corresponding parallel sorters in the CNU. As shown in Fig. 3.12, two symbol register groups, SRG_0 and SRG_1, store the Galois field symbol vector for the c-to-v messages sent to v. Take SRG_0 as an example, the input multiplexors select cnuSymIn_0 and cnuRA_0 as its Galois field symbol input and read address during check node processing. cnuSymIn_0 and cnuRA_0 are driven by symIn and RA in the corresponding PC unit shown in

Fig. 3.11, respectively. The Galois field symbol vector for the *a priori* message concerning v is stored in SRG_2. The output sa_i is the address of the input Galois field symbol in SRG_i. If the the input symbol has not been stored in SRG_i, $\text{sa}_i = n_m - 1$.

Based on the Galois field symbol vector stored in SRG, the proposed VNU performs variable node processing. Not all n_m elements of the LLR vector of a c-to-v message are stored. Instead, at most two LLR elements are stored since the use of the IPC algorithm. During variable node processing, each LLR element is computed on-the-fly based on Eq. (3.1). This LLR approximation approach reduces the memory required to store c-to-v messages. The LLR generation unit (LGU) computes the corresponding approximated LLR for c-to-v messages. The channel LLR generation unit (CGU) computes the approximated LLR for *a priori* message based on the approximation scheme proposed in Section 3.3.2. Suppose $l_{sum} = l_0 + l_1 + l_2$ for $d_v = 2$, variable node processing is carried out as follows:

1. The input multiplexors select vnuRA_0 as the read address of SRG0. The address signal, vnuRA_0, goes from 0 to $l_0 - 1$, and is increased by 1 each cycle. The corresponding Galois field symbol output is so_0. The input multiplexors of SRG1 and SRG2 select $\text{si}_{1,0}$ and $\text{si}_{2,0}$ as the symbol inputs, respectively. Here, $\text{si}_{1,0} = \text{so}_0 h_1/h_0$ and $\text{si}_{2,0} = \text{so}_0/h_0$, where h_0 and h_1 are the non-zero Galois field symbol in the v-th column of the parity-check matrix. The output multiplexor of SRG0 selects vnuRA_0 as the address input of LGU_0.

2. The output multiplexors of SRG1 and SRG2 select sa_1 and sa_2 as the address inputs of LGU_1 and CGU, respectively.

3. Similar read operations will be applied to SRG1 and SRG2 in serial. vnuRA_1 and vnuRA_2 will go from 0 to $l_1 - 1$ and 0 to $l_2 - 1$, respectively. The reading behavior of the SRGs during a round of variable node processing is shown in Fig. 3.13. Besides, $\text{si}_{0,1} = \text{so}_1 h_0/h_1$, $\text{si}_{0,2} = \text{so}_2 h_0$, $\text{si}_{1,2} = \text{so}_2 h_1$ and $\text{si}_{2,1} = \text{so}_1/h_1$, where so_1 and so_2 are the symbol output of SRG1 and SRG2, respectively.

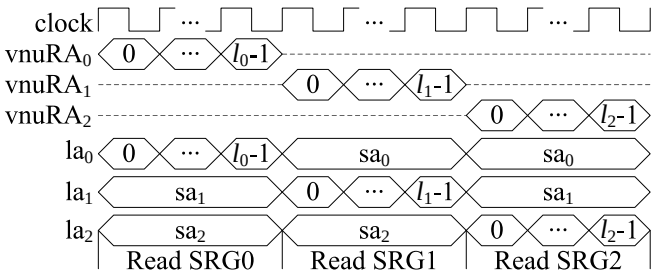

Figure 3.13: Reading behavior of the SRGs during the variable node processing

Suppose $n_m = 32$ and 5-bit quantization scheme is used. Based on the scaling factors proposed in Section 3.3.4, we propose two low complexity LGUs for $n_c = 0$ and 3, respectively. The LGU for $n_c = 0$ is shown in Fig. 3.14(a), where maxLLR is the maximum LLR stored during check node processing. The SC0 unit first multiplies the input by β, then divides the product by $F(n_m)$, where $F(n_m)$ is a power of 2. The division in SC0 is just bit shifting. The output of SC0 has 7 bits for the fraction part. In the proposed LGU, only one bit is kept for the fraction part. The ST0 unit returns the maximal value in the quantization range if the input is saturated. Otherwise, the output of ST0 is the same as the input. The LGU for $n_c = 3$, shown in Fig. 3.14(b), is more complex than that for $n_c = 0$ and needs both fixed-point multiplication and addition. $M_{c,v}$, shown in Algorithm 7, is another stored LLR. When the index is no greater than n_c, the SC1 unit multiplies

the input with η. Otherwise, the input of SC1 is multiplied with β. The product is then divided by $F(n_c)$ or $F(n_m)$ using bit shifting. The functionality of ST1 is similar to that of ST0. In addition, we also propose the CGU using the scaling parameters from Section 3.3.4. As shown in Fig. 3.15, the architecture of CGU is similar to that of LGU. The DEC unit in Fig. 3.15 generates the select signal for the multiplexer in the CGU.

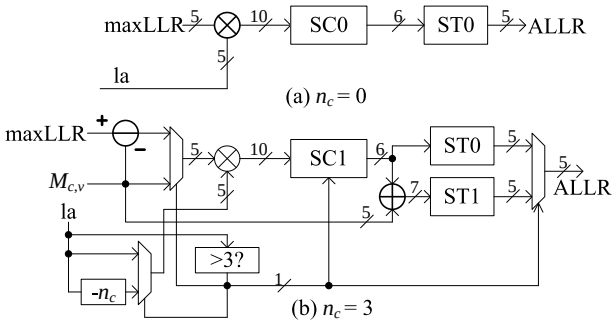

Figure 3.14: Architectures of the proposed LGUs

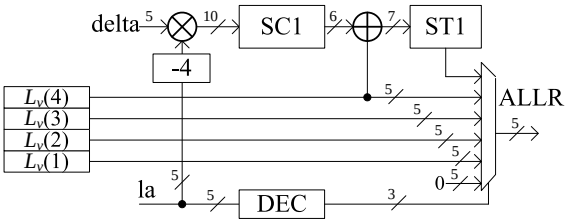

Figure 3.15: Architectures of the proposed CGU

Since $n_m = 32$ 5-bit LLRs need to be stored for the PC algorithm in [7], we also synthesize the proposed LGUs and compare the synthesis results with that of a (32×5)-bit SRAM module under a 90nm CMOS technology in Table 3.3. The SRAM is built with a register file using a memory compiler. LGU-0 and LGU-3 denote the LGU unit with $n_c = 0$ and $n_c = 3$, respectively, and CGU-4 denotes the CGU unit with $n_I = 4$. The areas of the proposed LGUs and CGU are only

a fraction of that of an SRAM storing 32 LLRs while maintaining the same clock rate.

Table 3.3: Comparisons of LGUs and CGU with a 32×5 SRAM.

	SRAM	LGU-0	LGU-3	CGU-3
Frequency (MHz)	400	400	400	400
area (μm^2)	7801	835	1031	1217
gate count	2763	295	365	431

3.4.4 Decoding schedule, decoder throughput, and interconnection

The decoding schedule of the proposed fully parallel decoder is simple. Take $d_v = 2$ as an example, at the beginning of decoding, each *a priori* message is loaded into SRG_2 in its corresponding VNU. The compressed LLR messages are stored in the CGU. At the same time, each message pair, which consists of an LLR and its related Galois field symbol, is sent to the corresponding PS unit through the V2C interconnection network. The loading of *a priori* messages takes n_m cycles. Check node processing begins once the loading of *a priori* messages is finished. The c-to-v messages and the compressed message vector $\mathbf{m}_c$ are updated at the same time, since the MLF unit generates the elements of $\mathbf{m}_c$ at a speed higher than the speed at which the elements of $\mathbf{m}_c$ are consumed by the PC unit. A round of check node processing takes L cycles.

Variable node processing starts once check node processing is finished, and takes l_{sum} cycles. In order to improve the decoder's clock frequency, P stages of pipeline registers are inserted in the VNU. Each element of the computed v-to-c messages is sent to the corresponding PS unit. For each v-to-c message, only the n_v minimum

LLRs and their corresponding Galois filed symbols are stored in a non-decreasing order in the corresponding PS unit. As a result, it takes $L + l_{sum}$ cycles to finish an iteration. The throughput T of the proposed fully parallel decoder is given by

$$T = \frac{NmRf}{N_I(L + l_{sum} + P)}, \tag{3.2}$$

where R is the code rate, f the clock rate, and N_I the number of iterations.

Since the SRGs of each PC unit are moved to the corresponding VNU to avoid global interconnection between different CNUs, the input control signals for the SRG during check node processing are transmitted form CNUs to VNUs through the C2V interconnection network. The output signal of the SRG that engages in the path construction is transmitted through the V2C interconnection network. As shown in Fig. 3.11, the control signals of the SRG during check node processing include: the write and read addresses, the symbol input, and the write enable signal. The output signals that go through the V2C network are exist_i and so_i. As a result, the bus width of each message that goes from CNUs to VNUs is $b_0 = \max(m+p, m+2s+1)$, where $s = \lceil \log_2 n_m \rceil$ is the width of the read and write address. The bus width of each message that goes from VNUs to CNUs is $b_1 = m+r+1$, where $r = \lceil \log_2 d_v \rceil + p$ is the number of bits used to represent an output LLR of VNU.

It is well known that the main obstacle of the fully parallel decoder architecture for binary LDPC codes is the routing congestion [70]. However, the routing congestion for the proposed fully parallel NB-LDPC decoder architecture is alleviated. The routing congestion is mainly determined by the number of the global interconnection wires that connect between CNUs and VNUs and by the numbers

of CNUs and VNUs instantiated in the decoder. For a binary (N_b, M_b) LDPC code with average variable node degree d_v, the fully parallel decoder in [70] needs M_b CNUs and N_b VNUs, and the number of edges in the corresponding Tanner graph is $N_b d_v$. As a result, the number of the global interconnection wire is $2N_b d_v p$, where p is the number of the quantization bits. For a NB-LDPC code over GF(2^m) with the same code length and code rate, the proposed fully parallel decoder will be easier to route since the numbers of CNUs and VNUs are reduced to M_b/m and N_b/m, respectively. Hence the number of the global interconnection wires is reduced to $N_b d_v(b_0 + b_1)/m$ for the proposed decoder architecture.

It has been shown that for large fields ($2^m \geq 64$), the best NB-LDPC codes decoded with belief propagation should be ultra sparse (cyclic codes, $d_v = 2$) [71]. The proposed fully parallel decoder architecture is especially suitable for such codes with moderate or short length. Suppose the length of a NB-LDPC code over GF(256) is 1024 bits. Let $n_m = 32$, $d_v = 2$ and $p = 5$. Based on the proposed fully parallel decoder architecture, the implementation of this 1024-bit NB-LDPC decoder has $\frac{1024}{8} \times 2 \times (19 + 15) = 8704$ global interconnection wires. On the other hand, the fully parallel LDPC decoder architecture in [70] for a 1024-bit binary LDPC code with $d_v = 2$ and $p = 5$ has $2 \times 1024 \times 2 \times 5 = 20480$ global interconnection wires. This comparison demonstrates that the proposed fully parallel decoder architecture for moderate or short NB-LDPC codes over large fields is feasible.

The proposed fully parallel decoder architecture not only works for QC-NB-LDPC codes, but also works for irregular NB-LDPC codes.

3.5 Implementation Results and Comparisons

In order to demonstrate the efficiency of our proposed fully parallel decoder architectures, we synthesize our NB-LDPC decoders using Design Compiler® Graphical (DCG) by Synopsys. Since DCG tightens timing and area correlation between synthesis and placement to 5% [72], the timing/area results obtained by using DCG are very close to those produced by place and route tools, such as IC Compiler®.

For power estimation, the PrimeTime® PX (PTPX) is employed. For the power estimation flow, the SPEF file generated by DCG is used to improve the accuracy of the power consumption results. The process of employing DCG synthesis is shown as follows:

1. Step 1: Perform the normal synthesis. Based on the cell area, determine the floorplan constraints, such as core area, pin locations and placement bounds and so on.

2. Step 2: Perform the DCG synthesis with these floorplan constraints.

3. Step 3: If the signal congestions predicted by DCG is not acceptable, revise the floorplan constraints and repeat Step 2. Otherwise, the DCG synthesis is finished.

Since the results in [15] are derived from place and route, we compare our DCG results with those in [15] in terms of energy efficiency (consumed energy per decoded bit) and area efficiency, where energy efficiency $= \frac{\text{power}}{\text{throughput}}$ and area efficiency $= \frac{\text{area}}{\text{throughput}}$. In this chapter, the (110, 88) NB-LDPC decoder is synthesized with DCG, and the power consumption is measured at the SNR point where BER is

around 1×10^{-6}. The physical results are shown in Table 3.4, where TGR denotes throughput to gate count ratio. Table 3.4 shows that our decoder architecture has better area efficiency and energy efficiency than those in [15].

The implementation results in [14] are derived from synthesis with Design Compiler using 180nm CMOS technology. Only the gate count (the gate count of memory is estimated) and throughput are provided in [14] and area and power are not available in [14]. In order to make a fair comparison, our fully parallel decoder is also synthesized under 180nm CMOS technology, and we compare the TGR of our decoder with those in [14]. The memory in [14] can be implemented with normal dual port SRAM. The memory of the our proposed decoder architecture can be implemented with the content addressable memory (CAM). Due to the lack of such a memory compiler, the memories in our proposed decoders are implemented with registers, and the gate counts of our decoder in Table 3.4 are the synthesis gate counts. Since registers require more area than memories generated from a memory compiler, our decoder architectures would have smaller gate counts than those in Table 3.4 if CAM modules were used. In terms of TGR, the proposed decoder architecture is better than those in [14].

The stochastic decoding algorithm for NB-LDPC codes is promising due to its low hardware complexity [9]. Besides, the stochastic decoding algorithm is a good candidate for fully parallel LDPC decoder architectures. The FPGA implementation of a (192, 96) NB-LDPC stochastic decoder over GF(256) [9] achieves a throughput of 65Mb/s, and it is projected [9] that a corresponding ASIC implementation can achieve a throughput of 698Mb/s, which would be 27% higher than the throughput of our proposed decoder on GF(256). In the chapter, however, we did not compare

our decoder architectures with that in [9] because the corresponding area, frequency and power results under ASIC implementation are not provided in [9].

3.6 Conclusion

In this chapter, a reduced memory complexity trellis based check node processing algorithm is proposed. An *a priori* message compression algorithm is also proposed to reduce memory requirement further. A simplified algorithm is also proposed to reduce the complexity of variable node processing. Based on the proposed algorithms, a fully parallel decoder architecture for NB-LDPC codes. The hardware efficiency of proposed fully decoder architecture is much higher than those of previous comparable decoder architectures in open literature.

Table 3.4: Comparisons with other decoder architectures.

	[15]	[14]	This work‡	
Code	GF(32)	GF(32)	GF(256)	GF(32)
Block Length	248	372	110	372
Code Rate	0.55	0.66	0.8	0.66
Process	90nm	180nm	28nm	180nm
Core Area (mm^2)	10.33 (0.99*)	-	1.289	-
Utilization	-	-	0.757	-
NAND Gate Count	1.92M	0.6M†	2.57M	4.1M
Frequency (MHz)	260	200	520	220
Iterations	10	10	10	10
Throughput (Mb/s)	47.69 (153.3*)	66	546	982
TGR (Mb/s/Million gate)	24.9 (79.6*)	110	212.4	239.5
Power (mW)	479	-	976	-
Energy Efficiency				
(nJ/bit)	10.06	-	1.78	-
(nJ/bit/Iteration)	1.006	-	0.178	-
Area Efficiency ($Mb/s/mm^2$)	4.62 (154.5*)	-	423.58	-

†The gate count of the memories in [14] is estimated, assuming that one memory bit takes the area of 1.5 NAND gates.
‡The implementation results of this work are not as accurate as those obtained from a place and route tool. In this work, the memories are implemented as registers , and the gate count of our decoder is obtained from DCG or normal synthesis.
* These results have been normalized to 28nm for comparison. Since the voltage of the process in [15] is not available, the energy efficiency has not been scaled.

Chapter 4

Efficient Error Control Decoder Architectures for Noncoherent Random Linear Network Coding

4.1 Introduction

Random linear network coding (RLNC) is an efficient technique for disseminating information in networks (see, for example, [39–42]). Due to its random linear operations, RLNC not only achieves network capacity with high probability in a distributed manner, but also provides robustness against varying network conditions [43]. Unfortunately, it is highly susceptible to errors due to noise, malicious or malfunctioning nodes, or insufficient min-cut [44]. As a result, error control is vital for RLNC.

Error control methods proposed for RLNC assume two transmission models. The

methods for the first model (see, for example, [45]) depend on and take advantage of the underlying network topology or the particular linear networking operations performed at various nodes. The methods for the other model (see, e.g., [44, 46]) assume that both the transmitter and the receiver have no knowledge of such channel transfer characteristics. The two models are referred to as coherent and noncoherent network coding, respectively. In this chapter, we focus on error control for noncoherent RLNC.

An error control code for noncoherent network coding [44], called a subspace code, is a set of subspaces. Information is encoded in the choice of a subspace spanned by a set of transmitted packets. A subspace code is called a constant-dimension code (CDC) if all subspaces are of the same dimension. CDCs lead to simplified network protocols due to the constant dimension. A class of asymptotically optimal CDCs, referred to as Kötter-Kschischang (KK) codes, has been proposed in [44]. A decoding algorithm based on interpolation for bivariate linearized polynomials is also proposed for KK codes in [44]. It was shown in [46] that KK codes correspond to lifting of Gabidulin codes, a class of optimal rank metric codes. As a result, KK codes can be decoded by the generalized decoding algorithm for the rank metric codes [46].

Motivated by KK codes, a new family of subspace codes, referred to as Mahdavifar-Vardy (MV) codes in this chapter, was proposed [47–49]. List decoding, which has been used to decode beyond the error correction diameter bound [50], can be applied to the decoding of MV codes. Using algebraic list decoding, it was shown [49] that MV codes can achieve a better tradeoff between rate and decoding radius than KK codes.

Error control for RLNC comes at the expense of additional computations needed for encoding and decoding. The complexities of existing decoding algorithms [44, 49, 51] for KK and MV codes are much higher than those of encoding, and are hence critical to applications of RLNC. Most previous works focus on theoretical aspects of network coding. For example, the decoding complexities of KK and MV codes were analyzed in [44, 46] and [47–49], respectively. However, theoretical analysis does not completely reflect how the decoding algorithms affect the hardware implementation results, such as area and throughput. For KK codes, decoder architectures based on the generalized decoding algorithm for rank metric codes [46] was proposed in [43]. Unfortunately, the rank metric decoder architectures in [43] suffer from limited throughput, long decoding latency and high area complexity. Besides, to the best of our knowledge, decoder architectures for MV codes and their hardware implementations have not been investigated in the open literature.

In this chapter, we focus on efficient architectures and their hardware implementations of interpolation based decoders for KK and MV codes. The main contributions of this chapter are:

1. The decoder of KK codes has two stages: interpolation and factorization. The generalized interpolation algorithm in [51] is used for the first stage since it is more efficient than Gaussian elimination [51]. For factorization, we propose a reformulated right division algorithm for linearized polynomials, which is suitable for hardware implementations.

2. The list decoder of MV codes also has two stages: interpolation and factorization. The generalized interpolation algorithm in [51] is used in the interpolation process. A linearized Roth-Ruckenstein (LRR) algorithm [53] is proposed

in [47] to solve the factorization problem for MV codes. In this chapter, we make a more detailed study on the LRR algorithm. For list size $L = 2$, we derive the equations used to compute all the information symbols and uncover the relation between two possible solutions. A matrix based LRR (M-LRR) algorithm, which is suitable for hardware implementations, is also proposed for factorization.

3. A serial decoder architecture and an unfolded decoder architecture for KK codes are proposed for applications with moderate and high throughputs, respectively. Both architectures are implemented for KK codes over GF(2^8) and GF(2^{16}) to demonstrate their efficiency. To the best of our knowledge, this is the first efficient implementation of interpolation-based decoder for KK codes. Compared to the rank metric decoder architectures for KK codes [43], the proposed serial decoder architecture improves the throughput by 4.9 and 13.2 times, while its gate counts are only 56% and 76% of their respective counterparts in [43]. Moreover, for these two codes, the unfolded architecture achieves a throughput of 12.5Gb/s and 41.6Gb/s, much higher than the throughput of 214Mb/s and 134Mb/s of their respective counterparts in [43]. The throughputs per thousand NAND gates of our architectures are much higher and their latency much shorter than their counterparts in [43].

4. A serial list decoder architecture for MV codes is proposed. To the best of our knowledge, this is the first hardware implementation of MV decoders. An efficient architecture for solving equations over an extension field GF(q^{ml}) ($q > 2$ is moderate) is proposed. The proposed equation solver does not require

complicated inversion operations over GF(q^{ml}). Besides, an implementation of factorization that computes all L possible transmitted packets in parallel is proposed, where L is the list size for list decoding.

The rest of the chapter is organized as follows. Section 4.2 provides some related background about KK and MV codes. Our serial and unfolded decoder architectures for KK codes are proposed in Section 4.3. Section 4.4 presents the list decoder architecture for MV codes. Section 5.5 presents the implementation results, and conclusions are drawn in Section 5.6.

4.2 KK and MV codes

4.2.1 KK codes and its decoding algorithms

KK codes [44] constitute an important class of subspace codes with constant dimensions. A KK code over GF(2^m) is described by three parameters (m, n, k), where n is the dimension of the transmitted subspace and k is the number of information symbols over GF(2^m). A k-dimension information vector $\mathbf{u} = (u_0, u_1 \cdots, u_{k-1})^T$ is treated as a linearized polynomial [46] $u(x) = u_0 x^{[0]} + u_1 x^{[1]} + \cdots + u_{k-1} x^{[k-1]}$, where $x^{[i]}$ denotes x^{2^i} and $u_i \in$ GF(2^m). In this chapter, for all linearized polynomials $u(x)$, $\deg(u(x)) = \max\{j : u_j \neq 0\}$ denotes the degree of $u(x)$. The information vector $\mathbf{u}$ is encoded into n packets over GF(2^m): p_0, p_1, $\cdots$, p_{n-1}, where $p_i = (\beta_i, u(\beta_i))$ and β_0, β_1, $\cdots$, β_{n-1} are linearly independent over GF(2^m). Each packet consists of two elements from GF(2^m). After these n encoded packets are injected into the network, N potentially corrupted packets $(r_0(s), r_1(s))$'s are received, where $r_0(s), r_1(s) \in$ GF(2^m) for $s = 0, 1, \cdots, N-1$. Based on the received packets, the

KK decoder produces $\hat{\mathbf{u}} = (\hat{u}_0, \hat{u}_1, \cdots, \hat{u}_{k-1})^T$.

Algorithm 9: Interpolation algorithm for KK codes [51]

input : $(r_0(s), r_1(s)); s = 0, 1, \cdots, N-1$
output: $d(x, y)$ s.t. $d(r_0(s), r_1(s)) = 0,\ s = 0, 1, \cdots, N-1$

Initialization: $f_0(x, y) = x, f_1(x, y) = y$
for $s = 0$ **to** $N - 1$ **do**
  for $i = 0$ **to** 1 **do**
    $\Delta_i = f_i(v_s, w_s)$
    $O_i = \max(\deg(f_{i,x}(x)), \deg(f_{i,y}(y)) + k - 1)$
  $I_0 = \{i : \Delta_i \neq 0\}; I_1 = \{i : \Delta_i = 0\}$
  if $I_0 \neq \emptyset$ **then**
    $i^* \leftarrow \underset{i \in I_0}{\operatorname{argmin}}\{O_i\}$
    for $i \in I_0$ **do**
      if $i \neq i^*$ **then**
        $F_i(x, y) = \Delta_{i^*} f_i + \Delta_i f_{i^*}$
      else $F_i(x, y) = {f_i}^2 + \Delta_i f_i$
  if $I_1 \neq \emptyset$ **then**
    for $i \in I_1$ **do**
      $F_i(x, y) = f_i$
  $f_0(x, y) = F_0, f_1(x, y) = F_1$
$O_0 = \max(\deg(f_{0,x}(x)), \deg(f_{0,y}(y)) + k - 1)$
$O_1 = \max(\deg(f_{1,x}(x)), \deg(f_{1,y}(y)) + k - 1)$
if $O_0 \leq O_1$ **then**
  $d(x, y) = f_0(x, y)$
else $d(x, y) = f_1(x, y)$

The KK decoder in [44] consists of two stages. The first stage, called interpolation, finds a nonzero bivariate linearized polynomial $d(x, y) = d_x(x) + d_y(y)$ such that $d(r_0(s), r_1(s)) = 0$ for $s = 0, 1, \cdots, N-1$. The degrees of $d_x(x)$ and $d_y(y)$ are at most m and $m - k + 1$, respectively. The second stage, referred to as factorization,

obtains the transmitted information symbols by computing a linearized polynomial $\hat{u}(x)$ such that $d(x, \hat{u}(x)) = 0$. While the interpolation can be implemented by solving a system of linear equations via Gaussian elimination, a more efficient generalized interpolation algorithm in the ring of linearized polynomials has been proposed in [51] (the interpolation algorithm proposed in [44] is in fact a special case of this generalized interpolation algorithm). The generalized interpolation algorithm in [51] adapted to the interpolation problem for an (m, n, k) KK code is shown in Algorithm 9, where $f_i(x, y) = f_{i,x}(x) + f_{i,y}(y)$ is a bivariate linearized polynomial over GF(2^m). A rank metric decoder has also been proposed for KK codes in [46], and its hardware implementation has been investigated in [43]. Unfortunately the rank metric decoder architectures in [43] suffer from limited throughput, long decoding latency, and high area complexity.

The interpolation algorithm in [51] for KK codes parallels Koetter's interpolation algorithm for RS codes. The comparison between these two algorithms are shown in Table 4.1, where IRS denotes the interpolation algorithm for RS codes. As shown in Table 4.1, the IRS algorithm and Algorithm 9 are similar in their polynomial updating rules. The key difference lies in the fact that Algorithm 9 deals with linearized polynomial while the IRS algorithm deals with polynomials. Due to their differences in multiplications, interpolator architectures for RS codes are not applicable to Algorithm 9.

Table 4.1: Interpolation by Polynomials and Linearized Polynomials

Algorithms		IRS	Algorithm 9
Ring		Polynomials	Linearized Polynomials
Module	Basis	$x, y, y^2, \ldots, y^L$	$y_0 = x, y_1, y_2, \ldots, y_L$
	Monomials	$x^i y^j$	$x^{[i]} \circ y_j \stackrel{\text{def}}{=} y_j^{[i]}$
	Elements	$P = \sum_{i,j \geq 0} a_{i,j} x^i y^j$	$Q = \sum_{i,j \geq 0} b_{i,j} y_j^{[i]}$
Linear Functionals		$D_{r,s} P(\alpha, \beta) = \sum_k \sum_j \binom{k}{r} \binom{j}{s} a_{j,k} \alpha^{k-r} \beta^{j-s}$	$D(Q) = Q(\alpha, \beta_1, \ldots, \beta_L)$
Initialization		$f_{0,j} = y^j$	$g_{0,j} = y_j$
Update	No Update	$f_{i+1,j} = f_{i,j}, j \notin J = \{j : D_{i+1}(f_{i,j}) \neq 0\}$	same as IRS
	Cross-term	$f_{i+1,j} = D(f_{i,j^*}) f_{i,j} - D(f_{i,j}) f_{i,j^*}, j \in J, j \neq j^*$	same rule as IRS
	Order-raise	$f_{i+1,j} = D(f_{i,j})(x f_{i,j}) - D(x f_{i,j}) f_{i,j}, j = j^*$	$g_{i+1,j} = D(g_{i,j}) g_{i,j}^{[1]} - D(g_{i,j}^{[1]}) g_{i,j}, j = j^*$

4.2.2 MV codes and its list decoding algorithm

MV codes are similar to but different from KK codes [44]. To enable list decoding, different code constructions are proposed for different code dimensions in [47, 48].

For an l-dimensional MV code over GF(q^{ml}), where l is a positive integer that divides $q-1$, the equation $x^l - 1 = 0$ has l distinct roots $e_0 = 1, e_1, \ldots, e_{l-1}$ over GF(q). We first choose a primitive element γ of GF(q^{ml}) so that $\gamma, \gamma^{[1]}, \ldots, \gamma^{[ml-1]}$ form a normal basis of GF(q^{ml}), where $\gamma^{[i]} = \gamma^{q^i}$. We then construct elements $\alpha_i = \gamma + e_i\gamma^{[m]} + e_i^2\gamma^{[2m]} + \cdots + e_i^{l-1}\gamma^{[m(l-1)]}$ over GF(q^{ml}) for $i = 0, 1, \ldots, l-1$, where e_i's are the l distinct roots of equation $x^l - 1 = 0$ over GF(q). It is proved in [48] that the set $\{\alpha_i^{[j]} : i = 0, 1, \ldots, l-1, j = 0, 1, \ldots, m-1\}$ is a basis of GF(q^{ml}) over GF(q).

For an information vector $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$ over GF(q) and its corresponding linearized polynomial $u(x) = \sum_{i=0}^{k-1} u_i x^{[i]}$, let $u^{\otimes i}(x)$ denote the composition of $u(x)$ with itself by i times for any nonnegative integer i, where

$$u^{\otimes i}(x) \triangleq \begin{cases} x & i = 0 \\ u(x) & i = 1 \\ u(u^{\otimes(i-1)}(x)) & i > 1 \end{cases} \tag{4.1}$$

The information vector $\mathbf{u}$ is encoded into l packets $p_0, p_1, \ldots, p_{l-1}$, where

$$p_i = \begin{cases} (\alpha_0, u(\alpha_0), u^{\otimes 2}(\alpha_0), \ldots, u^{\otimes L}(\alpha_0)) & i = 0 \\ (\alpha_i, \frac{u(\alpha_i)}{\alpha_i}, \ldots, \frac{u^{\otimes L}(\alpha_i)}{\alpha_i}) & \text{otherwise} \end{cases} \tag{4.2}$$

and L is the desired list size. Each packet consists of $L+1$ elements in GF(q^{ml}). At

the receiver, N potentially corrupted packets $(r_0(s), r_1(s), \cdots, r_L(s))$'s are received, where $r_0(s), r_1(s), \cdots, r_L(s) \in \mathrm{GF}(q^{ml})$ for $s = 0, 1, \cdots, N-1$.

Similar to the decoding of KK codes, the list decoding of MV codes is divided into two stages: interpolation and factorization. The generalized interpolation algorithm is also capable of performing the interpolation for the list decoding of MV codes. The generalized interpolation algorithm adapted to the interpolation problem for an l-dimensional MV code is shown in Algorithm 10.

As shown in Algorithm 10, $f_i(x, y_1, \cdots, y_L) = f_{i,x}(x) + f_{i,y_1}(y_1) + \cdots + f_{i,y_L}(y_L)$ is a nonzero multivariate linearized polynomial, where $f_{i,x}$ and f_{i,y_j}'s $(j = 1, 2, \cdots, L)$ are linearized polynomials. The maximal degrees of $f_{i,x}$, f_{i,y_j} are $(l+t)L$, $(l+t)L - j(k-1)$, respectively, where $t < lL - L(L+1)\frac{k-1}{2m}$ is the dimension of error packets received. The output is a nonzero multivariate linearized polynomial $d(x, y_1, \cdots, y_L)$ that satisfies $d(r_0(s), r_1(s), \cdots, r_L(s)) = 0$ for $s = 0, 1, \cdots, N-1$. The interpolation step is finished in N iterations.

The factorization step finds at most L possible solutions of $u(x)$ for the following equation:

$$d(x, u(x), u^{\otimes 2}(x), \cdots, u^{\otimes L}(x)) = 0. \tag{4.3}$$

An LRR algorithm [47], shown in Algorithm 11, has been proposed to solve Eq. (4.3). Let Y be a variable in the ring $\mathcal{L}_q[x]$, where $\mathcal{L}_q[x]$ is the set of linearized polynomials with coefficients in $\mathrm{GF}(q)$. Since the output of Algorithm 10 is $d(x, y_1, \cdots, y_L) = d_x(x) + d_{y_1}(y_1) + \cdots + d_{y_L}(y_L)$, Eq. (4.3) is equivalent to

$$d(x, Y) = d_0(x) + d_1(x) \otimes Y + \cdots + d_L(x) \otimes Y^{\otimes L} = 0 \tag{4.4}$$

Algorithm 10: Interpolation algorithm for MV codes [51]

input : $(r_0(s), r_1(s), \cdots, r_L(s)); s = 0, 1, \cdots, N-1$
output: $d(x, y_1, \cdots, y_L)$ s.t. $d(r_0(s), r_1(s), \cdots, r_L(s)) = 0$, $s = 0, 1, \cdots, N-1$

Initialization: $f_0(x, y_1, \cdots, y_L) = x, f_i(x, y_1, \cdots, y_L) = y_i$ $i = 1, 2, \cdots, L$
for $s = 0$ **to** $N-1$ **do**
  for $i = 0$ **to** L **do**
    $\Delta_i = f_i(r_0(s), r_1(s), \cdots, r_L(s))$
    $d_j = \deg(f_{i,y_j}(y)) + j(k-1))$ for $j = 1, 2, \cdots, L$
    $O_i = \max(\deg(f_{i,x}(x), d_1, d_2, \cdots, d_L)$
  $I_0 = \{i : \Delta_i \neq 0\}; I_1 = \{i : \Delta_i = 0\}$
  if $I_0 \neq \emptyset$ **then**
    $i^* \leftarrow \underset{i \in I_0}{\operatorname{argmin}}\{O_i\}$
    for $i \in I_0$ **do**
      if $i \neq i^*$ **then**
        $F_i(x, y_1, \cdots, y_L) = \Delta_{i^*} f_i + \Delta_i f_{i^*}$
      else $F_i(x, y_1, \cdots, y_L) = \Delta_i f_i^{[1]} + \Delta_i^{[1]} f_i$
  if $I_1 \neq \emptyset$ **then**
    for $i \in I_1$ **do**
      $F_i(x, y_1, \cdots, y_L) = f_i$
  $f_0 = F_0, f_1 = F_1$
for $i = 0$ **to** L **do**
  $d_j = \deg(f_{i,y_j}(y)) + j(k-1))$ for $j = 1, 2, \cdots, L$
  $O_i = \max(\deg(f_{i,x}(x), d_1, d_2, \cdots, d_L)$
$d(x, y_1, \cdots, y_L) = f_0$
$O_{min} = O_0$
for $i = 1$ **to** L **do**
  if $O_i < O_{min}$ **then**
    $d(x, y_1, \cdots, y_L) = f_i$
    $O_{min} = O_s$

where $u(x)$ is the solution of Y and

$$d_i(x) = \begin{cases} d_x(x) & i = 0 \\ d_{y_i}(y_i)|_{y_i=x} & i > 0. \end{cases} \tag{4.5}$$

If the polynomial $d(x, Y)$ is divisible by $x^{[s]}$, then we define

$$d_{\downarrow s}(x, Y) = d'_0(x) + d'_1(x) \otimes Y + \cdots + d'_L(x) \otimes Y^{\otimes L}, \tag{4.6}$$

where $d'_i(x)^{[s]} = d_i(x)$.

Algorithm 11: LRR algorithm [47]

Procedure: LRR$(d(x, Y), k, \lambda)$

Global variables: $A \subseteq \mathcal{L}_q[x], u(x) = \sum_{i=0}^{k-1} u_i x^{[i]} \in \mathcal{L}_q[x]$
Call procedure initially with $d(x, Y) \neq 0$ $\lambda = 0$
if $\lambda == 0$ **then**
  $A = \emptyset$
$s \leftarrow$ largest integer s.t. $d(x, Y)$ is divisible by $x^{[s]}$
$H(x, \gamma) \leftarrow \frac{1}{x} d_{\downarrow s}(x, \gamma x)$
$Z \leftarrow$ set of all roots of $H(0, \gamma)$ in GF(q)
foreach $\gamma \in Z$ **do** $u_\lambda \leftarrow \gamma$
if $\lambda < k - 1$ **then**
  LRR$(d_{\downarrow s}(x, Y^{[1]} + \gamma x), k, \lambda + 1)$
else
  if $d(x, u_{k-1}x) == 0$ **then**
    $A \leftarrow A \cup u(x)$

As shown in Algorithm 11, the L possible solutions of $u(x)$ are stored in the set A. The original LRR algorithm is a high-level algorithm, the detailed expression of $d(x, Y)$ are not specified in [47].

4.3 Efficient KK decoder architectures

In this chapter we first propose a serial decoder architecture and an unfolded decoder architecture for KK codes.

4.3.1 Serial decoder architecture

In order to minimize the hardware cost, a serial decoder architecture is proposed in Fig. 4.1, where the widths of multi-bit buses are shown. The serial architecture consists of the following major parts: coefficient registers CXR_i and CYR_i, two interpolators $\mathrm{interpolator}_0$ and $\mathrm{interpolator}_1$, polynomial selection unit polySel, and a polynomial divider polyDiv, which implements the factorization step. Algorithm 9 updates two bivariate linearized polynomials: $f_i(x,y) = f_{i,x}(x) + f_{i,y}(y)$ for $i = 0$ and 1, where $f_{i,x}(x) = \sum_{j=0}^{N_x-1} \mathrm{CEX}_i(j)x^{[j]}$ and $f_{i,y}(y) = \sum_{j=0}^{N_y-1} \mathrm{CEY}_i(j)y^{[j]}$ are linearized polynomials in x and y, respectively. For $i = 0$ or 1, the coefficients of $f_{i,x}(x)$ and $f_{i,y}(y)$ are stored in CXR_i and CYR_i, respectively. CXR_i and CYR_i consist of N_x and N_y m-bit registers, respectively, since each element in $\mathrm{GF}(2^m)$ is represented by m bits. $N_x - 1$ and $N_y - 1$ are set to the maximal degrees of $f_{i,x}(x)$ and $f_{i,y}(y)$, respectively, during the interpolation process. Hence, $N_x = m + 1$, $N_y = m - k + 2$.

$\mathrm{Interpolator}_0$ and $\mathrm{interpolator}_1$ compute the updated coefficients for $f_0(x,y)$ and $f_1(x,y)$, respectively, and write the updated coefficients back to CXR and CYR during each cycle. Since $\mathrm{interpolator}_0$ and $\mathrm{interpolator}_1$ have the same circuitry, the architecture of $\mathrm{interpolator}_0$ is discussed in Sec. 4.3.1. After the interpolation is finished, the polySel unit selects $f_0(x,y)$ if $O_0 \leq O_1$, or $f_1(x,y)$ otherwise. Since

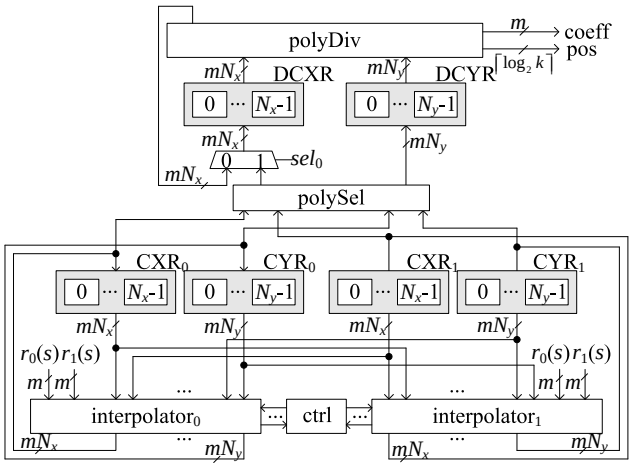

Figure 4.1: Serial KK decoder architecture

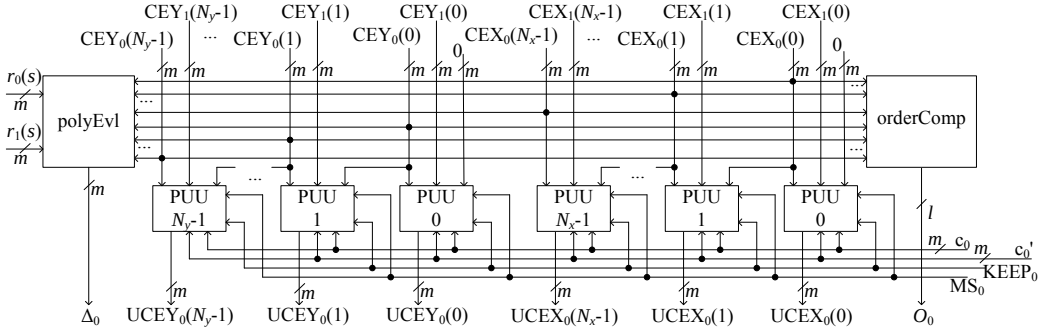

Figure 4.2: Architecture of interpolator$_0$

the polySel unit can be easily implemented with the orderComp unit (see Fig. 4.4) used for interpolator$_0$, the details of the polySel unit are omitted. The coefficients of the polynomial selected by the polySel unit will be stored in DCXR and DCYR, which consist of N_x and N_y m-bit registers, respectively. The polyDiv unit will then compute $\hat{\mathbf{u}}$ based on a reformulated right division algorithm described in Sec. 4.3.1.

Efficient interpolator architecture

The architecture of interpolator$_0$ is shown in Fig. 4.2. It computes the corresponding Δ_0 and O_0 as well as the updated polynomial coefficients for $f_0(x, y)$ as specified in Algorithm 9. Interpolator$_0$ consists of $N_x + N_y$ polynomial updating units (PUUs) and the orderComp and polyEvl units. Each PUU generates a new coefficient during each cycle. The polyEvl unit evaluates the corresponding linearized polynomial and generates Δ_0, and the orderComp unit generates O_0. The number of bits needed for O_0 is $l = \lceil \log_2 \max\{N_x, N_y + k - 1\} \rceil$.

To achieve high throughput, a fully parallel architecture shown in Fig. 4.3 is used for the polyEvl unit. In this work, all Galois field elements are represented with respect to a normal basis so that the q-exponentiation operation will be a cyclic shift. Suppose the normal basis representation for $b = \sum_{j=0}^{m-1} \gamma^{[j]} b_j \in \mathrm{GF}(2^m)$ is $(b_{m-1}, b_{m-2}, \cdots, b_0)$, where $b_j \in \mathrm{GF}(2)$ and $\gamma^{[j]}$'s constitute a normal basis. The corresponding normal basis representation for b^2 is $(b_{m-2}, \cdots, b_0, b_{m-1})$. In Fig. 4.3, the computation of $b^{[j]}$ is carried out by the S(j) unit, which cyclicly shifts its input by j positions and requires wiring only. The SUM$_0$ in Fig. 4.3 performs bit-wise XOR operations for their m-bit input messages. Finite field multiplications and additions are also used in the polyEvl unit. Additions over GF(2^m) are simply bit-wise XOR operations. In this work, we use the improved Massey–Omura normal basis multipliers proposed in our previous work [43, Sec. III-B].

The architecture of the orderComp unit is shown in Fig. 4.4. The IsZero unit tests whether its m-bit input message is zero. The two priority decoders in Fig. 4.4 compute $\deg(f_{0,x}(x)))$ and $\deg(f_{0,y}(y))$, respectively, and their outputs have $l_x = \lceil \log_2 N_x \rceil$ and $l_y = \lceil \log_2 N_y \rceil$ bits, respectively. In Fig. 4.4, k is the number of

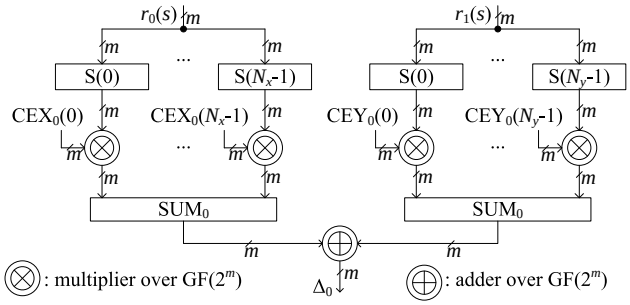

Figure 4.3: Architecture of polyEvl for interpolator$_0$

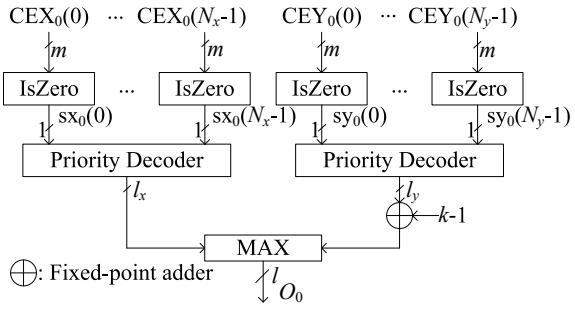

Figure 4.4: Architecture of orderComp for interpolator$_0$

information symbols. A fixed-point adder that performs integer addition is used. The MAX unit computes the maximum of its two inputs.

The two groups of PUUs in Fig. 4.2 — N_y PUUs on the left and N_x PUUs on the right — update $f_{0,y}(y)$ and $f_{0,x}(x)$, respectively. Since all PUUs have the same circuitry, the PUU that updates the coefficient of $x^{[j]}$ is shown in Fig. 4.5. In Algorithm 9, $f_0(x, y)$ is updated in three different ways: 1) $f_0(x, y) = f_0(x, y)^2 + \Delta_0 f_0(x, y)$ when $O_0 < O_1$ and Δ_0 is not zero; 2) $f_0(x, y) = \Delta_0 f_1(x, y) + \Delta_1 f_0(x, y)$ when $O_0 > O_1$ and Δ_0 is not zero; 3) $f_0(x, y)$ keeps unchanged when Δ_0 is zero. As a result, there are three different polynomial operations: 1) computing the square of a linearized polynomial; 2) multiplying a linearized polynomial with a constant; and 3) adding two linearized polynomials. The proposed PUU is configured to implement

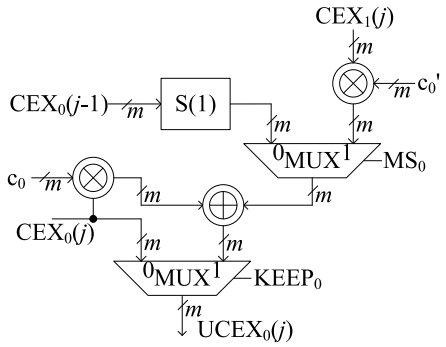

Figure 4.5: Architecture of the PUU that updates the coefficient of $x^{[j]}$

these three operations with two control signals MS_i and KEEP_i.

Taking Δ_0, Δ_1, O_0 and O_1 as inputs, the control unit computes the control signals

$$(\mathrm{KEEP}_0, \mathrm{KEEP}_1, \mathrm{MS}_0, \mathrm{MS}_1, c_0, c_0', c_1, c_1') = \begin{cases} (1,1,0,1,\Delta_0,\mathrm{X},\Delta_0,\Delta_1) & \text{if } \Delta_0 \neq 0, \Delta_1 \neq 0, O_0 \leq O_1 \\ (1,1,1,0,\Delta_1,\Delta_0,\Delta_1,\mathrm{X}) & \text{if } \Delta_0 \neq 0, \Delta_1 \neq 0, O_0 > O_1 \\ (0,1,\mathrm{X},0,\mathrm{X},\mathrm{X},\Delta_1,\mathrm{X}) & \text{if } \Delta_0 = 0, \Delta_1 \neq 0 \\ (1,0,0,\mathrm{X},\Delta_0,\mathrm{X},\mathrm{X},\mathrm{X}) & \text{if } \Delta_0 \neq 0, \Delta_1 = 0 \end{cases}, \tag{4.7}$$

where X in Eq. (4.7) is "don't care".

Reformulated right division algorithm

In [44], a recursive right division procedure is proposed to solve factorization problem. In this chapter, the right division procedure [44] is reformulated in a non-recursive manner. Let $a(x) = d_x(x)$ and $b(x) = d_y(y)|_{y=x}$. Let $\mathrm{lc}(a(x))$ denote the leading coefficient of $a(x)$. That is, if $a(x)$ has degree d, i.e., $a(x) = a_d x^{[d]} + a_{d-1} x^{[d-1]} + \cdots + a_0 x^{[0]}$, then $\mathrm{lc}(a(x)) = a_d \neq 0$. The reformulated right division algorithm is shown in Algorithm 12. The k messages symbols are recovered

within at most k iterations.

Algorithm 12: Reformulated right division algorithm

Input: $a(x)$ and $b(x), b(x) \not\equiv 0$
Output: $\hat{\mathbf{u}} = (\hat{u}_0, \hat{u}_1, \cdots, \hat{u}_{k-1})$

Initialization: $j = 0$; $\hat{u}_i = 0$, for $0 \leq i < k$
while $\deg(a(x)) \geq \deg(b(x))$ and $a(x) \neq 0$ **do**
  $d = \deg(a(x)), e = \deg(b(x)), q = d - e$
  $a_d = \mathrm{lc}(a(x)), b_e = \mathrm{lc}(b(x)), \hat{u}_q = (a_d/b_e)^{[m-e]}$
  $t(x) = (a_d/b_e)^{[m-e]} x^{[q]}$, $a(x) = a(x) - b(t(x))$
  $j = j + 1$
end while
if $j > k$ **or** $\deg(a(x)) > 0$ **then**
  return decoding failure
end if
return $\hat{\mathbf{u}}$

Efficient factorization architecture

A parallel polynomial divider that implements Algorithm 12 is shown in Fig. 4.6, where AX(j) and BX(j) denote the coefficients of $x^{[j]}$ for $a(x)$ and $b(x)$, respectively, UAX(j) the updated coefficients for $a(x)$, and coeff and pos a recovered information symbol and its position in the information vector, respectively. The COS unit finds the leading coefficient and the degree of a given linearized polynomial. The inv unit computes the inversion of the leading coefficient $\mathrm{lc}(b(x))$ of $b(x)$. The CS unit cyclicly shifts the m-bit input by $m - e$ positions, where e is the degree of $b(x)$. S(j) cyclicly shifts its input by j positions and hence requires wiring only. The LS unit has N_y m-bit inputs and N_x-1 m-bit outputs. As shown in Fig. 4.6, we have L(j+pos) = BX(j) for $j = 0, 1, \cdots, N_y - 1$. For other j, L(j)=0. In this work, a parallel inversion architecture is employed, and such an architecture for inversions

over GF(2^8) is shown in Fig. 4.7.

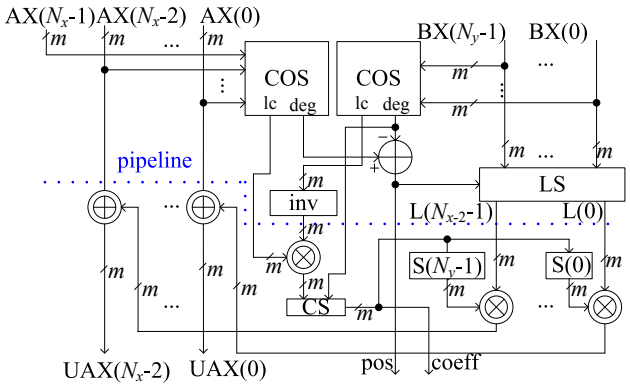

Figure 4.6: Architecture of the polyDiv unit

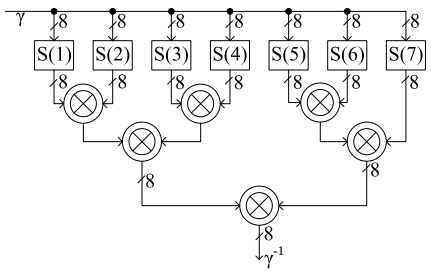

Figure 4.7: Parallel inversion architecture over GF(2^8)

Performance of the serial decoder architecture

We now consider the critical path delay (CPD), latency, and throughput of the serial architecture in Fig. 4.1. The critical path delay of the polyDiv unit is given by $T_{\text{COS}} + T_{\text{inv}} + T_{\text{mul}} + T_{\text{CS}} + T_{\text{mul}} + T_{\text{add}}$, where T_{COS} is the delay of the COS unit shown in Fig. 4.6 and T_{inv}, T_{mul}, and T_{add} are the delays of finite field inversion, multiplication, and addition, respectively. On the other hand, the critical path delay of an interpolator is given by $\max\{T_{\text{polyEvl}}, T_{\text{orderComp}}\} + T_{\text{ctrl}} + T_{\text{PUU}}$. The CPD of the polySel unit is negligible compared to those of the polyDiv unit and

the interpolator. When $m = 8$, the CPDs of the polyDiv unit and the interpolator are dominated by $5T_{\text{mul}}$ and $2T_{\text{mul}}$, respectively. In order to balance the CPDs between the polyDiv and interpolator, a stage of pipeline registers is inserted in the polyDiv unit (indicated by the dotted line in Fig. 4.6).

For the serial architecture in Fig. 4.1, the interpolation needs N cycles, where N is the number of linearly independent received packets. The maximal value of N is set to $2n-k$, since the decoder will fail if the number of errors exceeds its correction capability when $N > 2n - k$. Without the pipeline registers, the factorization will finish in M ($M \leq k$) cycles, since the polyDiv unit recovers one non-zero information symbol during each cycle. The polySel takes one cycle. In the worst case, it takes at most $N + 1 + k$ cycles to generate $\hat{\mathbf{u}}$. With the pipeline registers inserted in the polyDiv unit as shown in Fig. 4.6, it takes $2M$ cycles to finish the polynomial division, and the overall latency becomes $N+1+2M$. Once the coefficients of $d(x, y)$, which is the output of interpolation, are loaded into CXR and CYR, the interpolator could start processing the following received packets while the polyDiv unit is still inferring the previous information vector. The throughput of the proposed serial decoder architecture is given by $\frac{fmk}{\max(N,2M,1)}$ Mb/s, assuming that the clock rate of the decoder is f MHz.

4.3.2 Unfolded decoder architecture

For modern network applications, a throughput well beyond several Gb/s is desirable. Since the serial architecture described in Section 4.3.1 may not meet such throughput requirements, an unfolded decoder architecture is also proposed for high throughput scenarios. As shown in Algorithms 9 and 12, both the interpolation and

right division algorithms contain a loop, and both algorithms will finish in limited iterations. As a result, an unfolded architecture is proposed in Fig. 4.8.

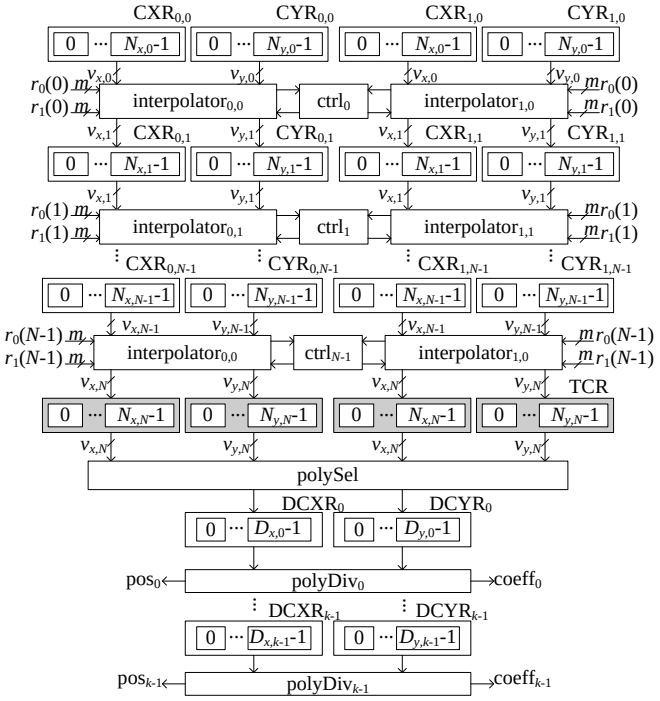

Figure 4.8: Unfolded decoder architecture

Compared with the serial decoder architecture, the unfolded architecture in Fig. 4.8 has N stages of interpolator and k stages of polyDiv, where N is the maximal number of received packets. For all received words, the N iterations of an interpolation process are distributed to N pairs of interpolators. interpolator$_{i,s}$ does the interpolation in iteration s of Algorithm 9 and passes the results to the next stage of interpolators. $N_{x,s}$ and $N_{y,s}$ denote the numbers of registers needed by interpolator$_{i,s}$. The polyDiv$_j$ unit implements iteration j of Alg. 12, and $D_{x,j}$ and $D_{y,j}$ denote the numbers of registers needed by polyDiv$_j$.

Based on Algorithm 9, at the beginning of iteration s, $(\deg(f_{i,x}(x)), \deg(f_{i,y}(y))) \leq$

$$(d_{x,s}, d_{y,s}) = \begin{cases} (s, 0) & \text{if } 0 \le s \le k \\ (s, s-k) & \text{if } k < s \le m \\ (m, m+1-k) & \text{if } s > m \end{cases}.$$ Similarly, we can show that the degrees of $a(x)$ and $b(x)$ at the beginning of iteration j of Alg. 12 are at most $p_{x,j} = m - j$ and $p_{y,j} = m - k + 1$, respectively, for $j = 0, 1, \cdots, k-1$. Thus, we can use $N_{x,s} = d_{x,s} + 1$ and $N_{y,s} = d_{y,s} + 1$ registers for interpolator$_{i,s}$ and $D_{x,j} = p_{x,j} + 1$ and $D_{y,j} = p_{y,j} + 1$ registers for polyDiv$_j$. These upper bounds help to reduce the hardware cost of the proposed unfolded decoder architecture.

The decoding latency of the unfolded decoder architecture is the same as that of the serial architecture, but the throughput is fmk Mb/s, which is higher than the serial architecture.

4.4 Efficient MV list Decoder Architecture

MV codes enable stronger error correction at the cost of higher computational complexity. As shown in [48, Fig. 1], MV codes under list decoding enhance the average decoding radius when packet rate is low. In this section, an efficient serial list decoder architecture for MV codes is proposed. Without loss of generality, we take $L = 2$ as an example to simplify the presentation. The decoder architecture for a different L is similar and can be easily obtained.

4.4.1 Serial list decoder architecture

A serial list decoder architecture for MV codes is shown in Fig. 4.9. When $L = 2$, three multivariate linearized polynomials, $f_0(x, y_1, y_2)$, $f_1(x, y_1, y_2)$, and $f_2(x, y_1, y_2)$,

participate in interpolation according to Algorithm 10, where $f_i(x, y_1, y_2) = f_{i,x}(x) + f_{i,y_1}(y_1) + f_{i,y_2}(y_2)$. As shown in Fig. 4.9, three groups of coefficient registers, CR_0, CR_1, and CR_2, store the coefficients of linearized polynomial f_0, f_1, and f_2, respectively. For each group of coefficient registers, CX, CY_1, and CY_2 store the coefficients of $f_{i,x}$, f_{i,y_1}, and f_{i,y_2}, respectively. The interpolator units perform the interpolation step and update corresponding coefficient registers according to Algorithm 10. Once the interpolation step is finished, the factorization unit computes two possible transmitted information vectors.

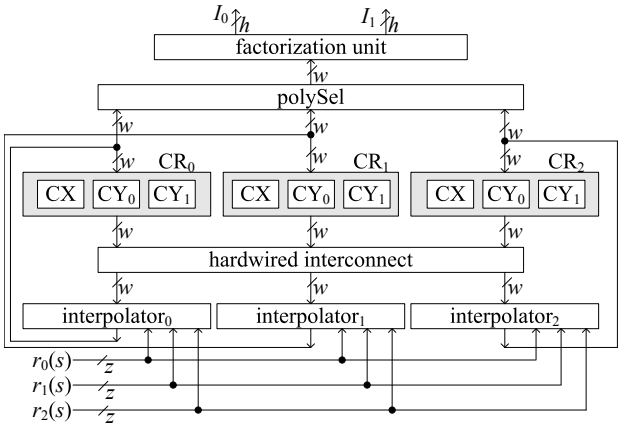

Figure 4.9: Unfolded decoder architecture

The coefficients of these multivariate linearized polynomials are elements in $\mathrm{GF}(q^{ml})$, where $q = 2^h$. Each coefficient is represented by an ml-dimension vector over $\mathrm{GF}(q)$. As a result, each coefficient is represented by a z-bit binary vector and stored in a z-bit register, where $z = mlh$.

The degree of each multivariate linearized polynomial may keep increasing during the interpolation step. Besides, within the decoding radius, there is an upper bound on the degree that each linearized polynomial can achieve. For an l-dimension MV

code over GF(q^{ml}), if a transmitted information vector can be correctly recovered, then the maximal values of $\deg(f_{i,x})$, $\deg(f_{i,y_1})$, and $\deg(f_{i,y_2})$ are $(l+t)L$, $(l+t)L-(k-1)$, and $(l+t)L-2(k-1)$, respectively, where $t < lL - L(L+1)\frac{k-1}{2m}$ is the dimension of error packets received. Let N_0, N_1 and N_2 denote the numbers of z-bit coefficient registers for CX, CY_1 and CY_2, respectively. Then, $N_0 = (l+t)L+1$, $N_1 = (l+t)L-(k-1)+1$, $N_2 = (l+t)L-2(k-1)+1$ and $w = (N_0+N_1+N_2)z$.

Similar to the KK decoders proposed in Section 4.3, all the finite field arithmetic operations for the decoding of MV codes are performed assuming a normal basis. Suppose an element $c = \sum_{i=0}^{ml-1} c_i\gamma^{[i]}$ is represented as $(c_{ml-1}, \cdots, c_1, c_0)$ under normal basis, where $\gamma^{[i]}$'s constitute a normal basis of GF(q^{ml}) and $c_i \in$ GF(q). We also define $c^{[-i]} \triangleq g$, where $g^{[i]} = c$. Then, under normal basis, $c^{[-1]}$ is represented as $(c_0, c_{ml-1}, \cdots, c_1)$.

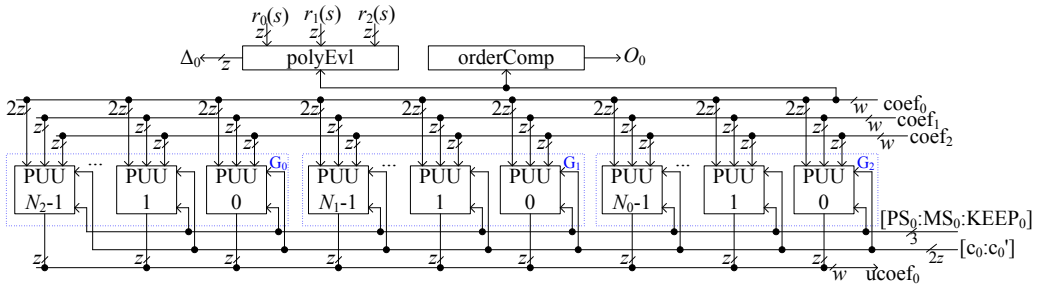

Figure 4.10: The architecture of interpolator$_0$ for the proposed MV decoder

4.4.2 Efficient interpolator architecture for MV codes

The proposed interpolator for the serial MV list decoder is similar to that of the KK decoders in Section 4.3. The interpolator architecture is shown in Fig. 4.10. Interpolator$_i$ updates the coefficients of the multivariate linearized polynomial f_i in Algorithm 10. Take interpolator$_0$ as an example. As shown in Fig. 4.10, interpolator$_0$

consists of polyEvl, orderComp and three groups of PUUs. The functions of these units are similar to those of an interpolator for the KK decoders in Section 4.3. As shown in Fig. 4.10, coef_0, coef_1 and coef_2 denote the coefficients of f_0, f_1 and f_2, respectively. The architectures of polyEvl and orderComp are similar to those of the KK decoders in Section 4.3. Let $f_{0,x} = \sum_{i=0}^{N_0-1} \text{CEX}(i)x^{[i]}$, $f_{0,y_1} = \sum_{i=0}^{N_0-1} \text{CEY1}(i)y_1^{[i]}$ and $f_{0,y_2} = \sum_{i=0}^{N_0-1} \text{CEY2}(i)y_2^{[i]}$. The architectures of polyEvl and orderComp of interpolator_0 when $L = 2$ are shown in Fig. 4.11 and Fig. 4.12, respectively, where all the multipliers are based on normal basis and z is the number of bits used to represent a coefficient.

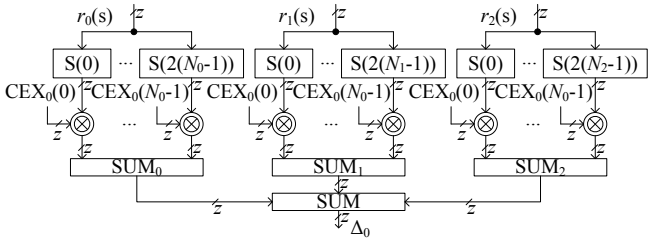

Figure 4.11: Architecture of polyEvl for interpolator_0 of the proposed MV decoder

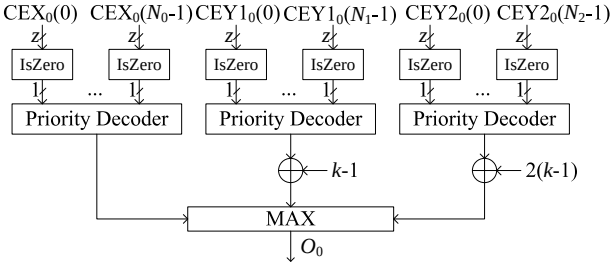

Figure 4.12: Architecture of orderComp for interpolator_0 of the proposed MV decoder

The PUU of interpolator_0, shown in Fig. 4.13, updates the coefficient of $x^{[j]}$ for $f_{0,x}$. The linearized polynomial f_0 is updated in four different ways: 1) $f_0 = \Delta_1 f_0 + \Delta_0 f_1$; 2) $f_0 = \Delta_2 f_0 + \Delta_0 f_1$; 3) $f_0 = \Delta_0 f_0^{[1]} + \Delta_0^{[1]} f_0$ and 4) f_0 remains

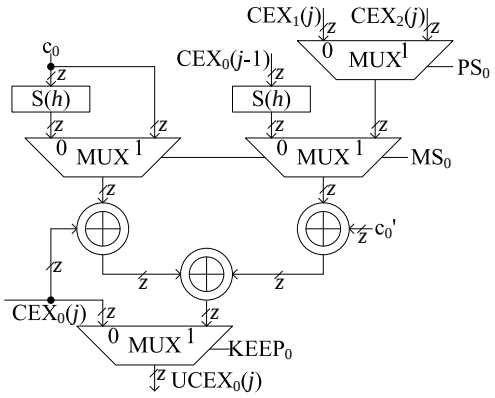

Figure 4.13: Architecture of PUU that updates $x^{[j]}$ for interpolator$_0$ of the proposed MV decoder

the same. The proposed PUU in Fig. 4.13 is configured to implement any of these updating operation with properly set control signals PS_0, MS_0 and KEEP_0.

4.4.3 Efficient factorization architecture for MV codes

Hardware efficient matrix based LRR algorithm

An LRR algorithm is proposed in [47] to solve the factorization problem for the list decoding of MV codes. For efficient hardware implementation of the factorization algorithm, more details about the LRR algorithm should be derived.

In this chapter, assuming $L = 2$, we derive the expression of linearized polynomial $d_{\downarrow s}(x, Y)$ in each iteration of Algorithm 11. We denote $d_{\downarrow s}(x, Y)$ as $d^{(i)}(x, Y)$ during the computation of the information symbol u_i. Based on $d^{(i)}(x, Y)$, the equation used to compute u_i is also derived. Here, Lemma 1 is given without proof since it is straightforward.

Lemma 1. *Let* $Y = \sum_{k=0}^{l} a_k x^{[k]}$ *be an element in the ring* $\mathcal{L}_q[x]$*, and* $\lambda \in GF(q)$*, then* $(Y^{[1]} + \lambda x)^{[i]} \otimes (Y^{[1]} + \lambda x)^{[j]} = Y^{[1+i]} \otimes Y^{[1+j]} + \lambda^2 x^{[i+j]}$.

Let Y_i denote $Y \otimes Y^{[i]}$. When $L = 2$, $d(x, Y)$ in Eq. (4.4) has the following general form:

$$\begin{aligned} d(x,Y) &= d_{0,0}x^{[0]} + \cdots + d_{0,n_0}x^{[n_0]} \\ &+ d_{1,0}Y^{[0]} + \cdots + d_{1,n_1}Y^{[n_1]} \\ &+ d_{2,0}(Y_0)^{[0]} + \cdots + d_{2,n_2}(Y_0)^{[n_2]}, \end{aligned} \tag{4.8}$$

where $n_0 = ml - 1$, $n_1 = n_0 - (k-1)$, and $n_2 = n_0 - 2(k-1)$. Then, $d^{(i)}(x, Y)$ has the following general form:

$$\begin{aligned} d^{(i)}(x,Y) &= d^{(i)}_{0,0}x^{[0]} + \cdots + d^{(i)}_{0,n_0}x^{[n_0]} \\ &+ d^{(i)}_{1,0}Y^{[0]} + \cdots + d^{(i)}_{1,n_1}Y^{[n_1]} \\ &+ d^{(i)}_{2,0}(Y_i)^{[0]} + \cdots + d^{(i)}_{2,n_2}(Y_i)^{[n_2]}. \end{aligned} \tag{4.9}$$

and $d^{(0)}(x, Y) = d_{\downarrow s}(x, Y)$. The equation to solve for u_0 is $d^{(0)}_{2,0}u_0^2 + d^{(0)}_{1,0}u_0 + d^{(0)}_{0,0} = 0$. The coefficients of $d^{(i+1)}(x, Y)$ can be derived from $d^{(i)}(x, Y)$. Let $D^{(i)}(x, Y) = d^{(i)}(x, (Y^{[1]} + u_i x))$, then

$$\begin{aligned} D^{(i)}(x,Y) &= d^{(i)}_{0,0}x^{[0]} + \cdots + d^{(i)}_{0,n_0}x^{[n_0]} \\ &+ d^{(i)}_{1,0}Y^{[1]} + \cdots + d^{(i)}_{1,n_1}Y^{[n_1+1]} \\ &+ d^{(i)}_{1,0}u_i x^{[0]} + \cdots + d^{(i)}_{1,n_1}u_i x^{[n_1]} \\ &+ d^{(i)}_{2,0}(Y_{i+1})^{[1]} + \cdots + d^{(i)}_{2,n_2}(Y_{i+1})^{[n_2+1]} \\ &+ d^{(i)}_{2,0}u_i^2 x^{[i]} + \cdots + d^{(i)}_{2,n_2}u_i^2 x^{[i+n_2]}. \end{aligned} \tag{4.10}$$

Eq. (4.10) can be simplified to be

$$\begin{aligned} D^{(i)}(x,Y) &= D_0^{(i)}(x) + D_1^{(i)}(Y) + D_2^{(i)}(Y_{i+1}) \\ &= D_{0,0}^{(i)}x^{[0]} + \cdots + D_{0,n_0}^{(i)}x^{[n_0]} \\ &+ D_{1,0}^{(i)}Y^{[1]} + \cdots + D_{1,n_1}^{(i)}Y^{[n_1+1]} \\ &+ D_{2,0}^{(i)}(Y_{i+1})^{[1]} + \cdots + D_{2,n_2}^{(i)}(Y_{i+1})^{[n_2+1]}, \end{aligned} \tag{4.11}$$

where $D_{0,j} = d_{0,j}^{(i)} + d_{1,j}^{(i)}u_i$ for $0 \le j < i$, $D_{0,j} = d_{0,j}^{(i)} + d_{1,j}^{(i)}u_i + d_{2,j}^{(i)}u_i^2$ for $j \ge i$, $D_{1,j} = d_{1,j}^{(i)}$ and $D_{2,j} = d_{2,j}^{(i)}$ for all j. Then, we have

$$d^{(i+1)}(x,Y) = D_{\downarrow s}^{(i)}(x,Y), \tag{4.12}$$

where s is the largest integer s.t. $D_0^{(i)}(x)$, $D_1^{(i)}(Y)$ and $D_2^{(i)}(Y \otimes Y^{[i+1]})$ are divisible by $x^{[s]}$, $Y^{[s]}$ and $(Y \otimes Y^{[i+1]})^{[s]}$, respectively. Once $d^{(i)}(x,Y)$ is determined, we obtain an equation about u_i, $d^{(i)}(x, u_i x) = 0$.

It is interesting to illustrate the root pattern of the factorization algorithm for MV codes. Since $L = 2$, u_i is a root of the following $i+1$ equations:

$$\begin{cases} d_{0,0}^{(i)} + d_{1,0}^{(i)}u_i = 0 \\ d_{0,1}^{(i)} + d_{1,1}^{(i)}u_i = 0 \\ \cdots \\ d_{0,i-1}^{(i)} + d_{1,i-1}^{(i)}u_i = 0 \\ d_{0,i}^{(i)} + d_{1,i}^{(i)}u_i + d_{2,0}^{(i)}u_i^2 = 0. \end{cases} \tag{4.13}$$

The first nonzero equation in Eq. (4.13) is used to solve u_i. The equation to derive u_0 is $d_{2,0}^{(0)}u_0^2 + d_{1,0}^{(0)}u_0 + d_{0,0}^{(0)} = 0$.

Lemma 2. *Consider a nonzero quadratic equation* $f(u) = au^2 + bu + c = 0$*, where* $a, b, c \in GF(q^{ml})$*. Here* $u \in GF(q)$*, where* $q = 2^h$*. If* $f(u) = 0$ *has two distinct roots, then* $a \neq 0$*, and* $b \neq 0$*. If* $f(u) = 0$ *has two identical roots, then* $a \neq 0$*, and* $b = 0$*. If* $f(u) = 0$ *only has one root, then* $a = 0$*,* $b \neq 0$*.*

The proof of Lemma 2 is omitted here since it is very straightforward. The roots generation of the factorization process is shown in Fig. 4.14. Without loss of generality, suppose the equation is $f_0(u_0) = 0$, derived from $d^{(0)}(x, Y)$, which has only one root u_0. $d^{(1)}(x, Y)$ is computed based on $d^{(0)}(x, Y)$ and u_0. The equation $f_1(u_1) = 0$ also has only one root u_1. The same root pattern repeats until $d^{(i)}(x, Y)$ is computed. Suppose $f_i(u_i) = 0$ has two different roots $u_{i,0}$ and $u_{i,1}$. As a result, $dl^{(i+1)}(x, Y)$ and $dr^{(i+1)}(x, Y)$ are computed based on $u_{i,0}$ and $u_{i,1}$, respectively. $dl^{(i+1)}(x, Y)$ and $dr^{(i+1)}(x, Y)$ are the linearized polynomials used to derive the equations about $u_{i+1,0}$ and $u_{i+1,1}$, which are two possible values of information symbol u_{i+1}, respectively. Then we give the following two lemmas which are proved in Appendix A and B, respectively.

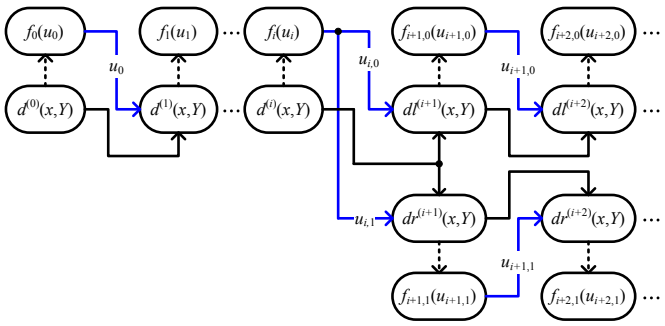

Figure 4.14: Root pattern

Lemma 3. *Both equations* $f_{i+1,0}(u_{i+1,0}) = 0$ *and* $f_{i+1,0}(u_{i+1,0}) = 0$ *have only one root, where* $f_{i+1,0}(u_{i+1,0}) = 0$ *and* $f_{i+1,0}(u_{i+1,0}) = 0$ *are derived from* $dl^{(i+1)}(x, Y)$

and $dr^{(i+1)}(x, Y)$*, respectively.*

Lemma 4. *The equation* $f_{i+j,0}(u_{i+j,0}) = 0$*, derived from* $dl^{(i+j)}(x, Y)$*, has only one root for* $j > 1$*. The same is true for* $dr^{(i+j)}(x, Y)$*.*

Based on the above discussions, for $L = 2$, the two possible output information vectors have the following general form

$$\begin{aligned} \mathbf{u}^{(0)} &= \{u_0, u_1, \cdots, u_{i,0}, u_{i+1,0}, \cdots, u_{k-1,0}\} \\ \mathbf{u}^{(1)} &= \{u_0, u_1, \cdots, u_{i,1}, u_{i+1,1}, \cdots, u_{k-1,1}\} \end{aligned} \tag{4.14}$$

where $0 \leq i \leq k-1$. It is also possible that factorization produces only one solution. Then $\mathbf{u}^{(0)} = \mathbf{u}^{(1)}$. In this chapter, a matrix based LRR (M-LRR) algorithm that is suitable for hardware implementation is proposed in Algorithm 13. Here, we assume $L = 2$.

Algorithm 13: M-LRR algorithm

input : $d(x, Y), n_0, n_1, n_2$
output: $\mathbf{u}^{(0)}, \mathbf{u}^{(1)}$

Initialization: $M^{(0)} = \mathbf{0}_{3\times n_0}, M^{(1)} = \mathbf{0}_{3\times n_0}$
for $i = 0$ **to** 2 **do**
  for $j = 0$ **to** n_i **do**
    $M^{(0)}_{i,j} = M^{(1)}_{i,j} = d_{i,j}$
for $i = 0$ **to** $k-1$ **do**
  $M^{(0)} = \text{shiftPow}(M^{(0)})$; $M^{(1)} = \text{shiftPow}(M^{(1)})$
  $u^{(0)}_i = \text{solver}_0(M^{(0)}, i)$; $u^{(1)}_i = \text{solver}_1(M^{(1)}, i)$
  $M^{(0)} = \text{matrixUpdate}(M^{(0)}, u^{(0)}_i, i)$
  $M^{(1)} = \text{matrixUpdate}(M^{(1)}, u^{(1)}_i, i)$

As shown in Algorithm 13, the coefficients of $d(x, Y)$, which are the output of interpolation, are first copied into the coefficient matrices $M^{(0)}$ and $M^{(1)}$. The

proposed M-LRR algorithm outputs at most two possible transmitted information vectors. The shiftPow function is shown in Algorithm 14. The solver_0 function in Algorithm 13 is shown in Algorithm 15. The solver_1 function is the same as solver_0 except that it returns r_1 when $f(u) = 0$ has two different roots. The matrixUpdate function shown in Algorithm 16 updates coefficient matrices based on Eq. (4.10).

The shiftPow function preprocesses the coefficient matrix so that at least one of $M_{0,0}$, $M_{1,0}$ and $M_{2,0}$ is a nonzero coefficient, where M is the input matrix. Take $M^{(0)}$ as an example, after preprocessing, the solver_0 function derives the coefficients of equation $f_i^{(0)}(u_i^0) = 0$ and solves this equation. It is possible that $f_i^{(0)}(u_i^0) = 0$ may have two distinct roots. However, only one root is selected as shown in Algorithm 15. The matrixUpdate function updates the coefficient matrix that will be used to compute the next information symbol.

Algorithm 14: shiftPow

input : M
output: M'

Initialization: $M' = \mathbf{0}_{3\times n_0}$
$i = 0$
for $j = 0$ **to** n_0 **do**
  if $M_{0,j} \neq 0$ *or* $M_{1,j} \neq 0$ *or* $M_{2,j} \neq 0$ **then**
    $i = j$
    break
for $s = 0$ **to** 2 **do**
  for $j = i$ **to** n_0 **do**
    $M'_{s,j-i} = M_{s,j}^{[-i]}$

Algorithm 15: solver$_0$

input : M, i
output: u_i

Initialization: $a = 0, b = 0, c = 0, a, b, c \in GF(q^{ml})$
$pow = 2$
for $j = 0$ **to** $i - 1$ **do**
  if $M_{0,j} \neq 0$ *or* $M_{1,j} \neq 0$ **then**
    $a = 0, b = M_{1,j}, c = M_{0,j}$
    $pow = 1$
    break
if $pow == 2$ **then**
  $a = M_{2,i}, b = M_{1,i}, c = M_{0,i}$
$f(u) = au^2 + bu + c$
solve equation $f(u) = 0, u \in GF(q) = \{0, 1, \cdots, q-1\}$
If $f(u) = 0$ has only one root r, then $u_i = r$
If $f(u) = 0$ has two roots r_0 and r_1, where $r_0 < r_1$, then $u_i = r_0$

Algorithm 16: matrixUpdate

input : M, u_i, i
output: M'

Initialization: $M' = \mathbf{0}_{3 \times n_0}$
for $j = 1$ **to** $i - 1$ **do**
  $M'_{0,j-1} = (M_{0,j} + u_i M_{1,j})^{[-1]}$
for $j = i$ **to** n_0 **do**
  $M'_{0,j-1} = (M_{0,j} + u_i M_{1,j} + u_i^2 M_{2,j})^{[-1]}$
for $j = 0$ **to** n_0 **do**
  $M'_{1,j} = M_{1,j}^{[-1]}$
  $M'_{2,j} = M_{2,j}^{[-1]}$

Efficient implementation of the M-LRR algorithm

Based on the proposed M-LRR algorithm, a parallel factorization architecture for list size $L = 2$ is proposed in Fig. 4.15. The proposed factorization architecture computes two possible transmitted information vectors $\mathbf{u}^{(0)}$ and $\mathbf{u}^{(1)}$ at the same time. It takes at most n_0 cycles to compute all possible information symbols. As shown in Fig. 4.15, after the interpolation step is finished, the output of polySel is loaded into the matrix coefficient register $M^{(0)}$ and $M^{(1)}$. The equation coefficients selector (ECS) unit computes the coefficients of $f(u)$ as described in Algorithm 15. The SV0 and SV1 units compute corresponding roots based on coefficients output from the ECS unit. The coefficient matrix update (CMU) unit implements both the shiftPow and matrixUpdate functions specified in Algorithms 14 and 16. Both ECS and CMU can be implemented with combinational logic.

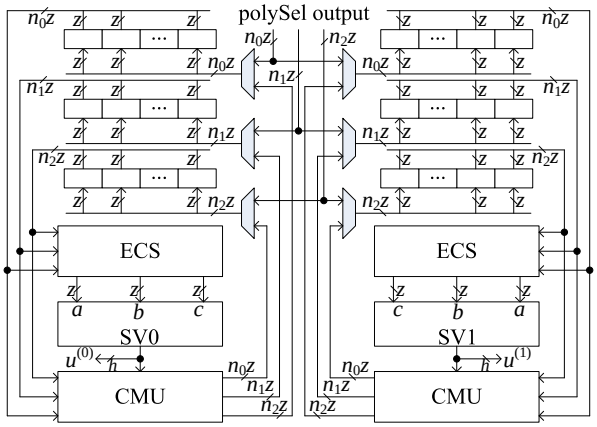

Figure 4.15: Architecture of factorization for MV decoder ($L = 2$)

The quadratic equation $f(u) = 0$ over GF(q^{ml}) is solved by enumeration. The architecture of the proposed SV0 unit is shown in Fig. 4.16. As shown in Fig. 4.16, the proposed SV0 consists of q equation checkers. The checker-i unit checks whether

$f(i) = 0$, where $i \in \mathrm{GF}(q)$. It outputs zero if $f(i) = 0$. The root selector (RS) unit chooses the final roots. The root selection rule is specified in Algorithm 15. The architecture of SV1 is almost the same as SV0 except the slight difference in the RS unit. The proposed SV0 architecture is suitable for small or moderate q. Both hardware cost and critical path will increase dramatically when q is large.

It needs at most $n_0 = ml - 1$ cycles to finish the factorization step. As a result, the worst case throughput is $\frac{fhk}{\max(N,n_0)}$Mb/s, where f is the clock frequency.

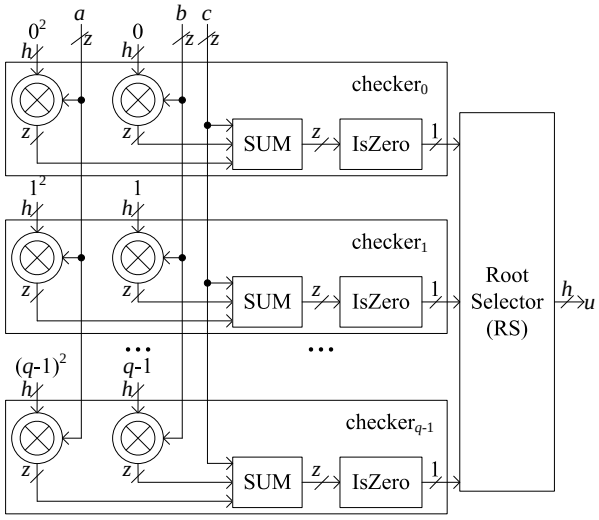

Figure 4.16: Architecture of SV0

Table 4.2: Hardware implementation results comparison.

code	KK codes				MV codes	
architecture type	rank metric decoder [43]		proposed serial		proposed unfolded	proposed serial
field	$GF(2^8)$	$GF(2^{16})$	$GF(2^8)$	$GF(2^{16})$	$GF(2^8)$	$GF(2^{16})$	$GF(4^6)$
gate count	71.1k	421.5k	40.2k	319.8k	274.8k	4390.9k	341.8k
Frequency (MHz)	455	276	400	333	400	333	322
Latency (ms)	0.47	2.72	0.053	0.13	0.053	0.13	0.047
Throughput (Mb/s)	214	134	1066	1777	12800	42666	193
Gate efficiency	3.0	0.32	26.5	5.56	46.5	9.72	0.56

4.5 Implementation Results

The two decoder architectures proposed in Section 4.3 are implemented for the two KK codes investigated in [43]: an (8, 8, 4) KK code over GF(2^8) and a (16, 16, 8) KK code over GF(2^{16}). The technology we used is a Free PDK 45nm process [73] which is the same as that used in [43]. In Table 4.2, we compare the implementation results of our decoder architectures with those in [43] in terms of gate count, frequency, throughput and latency. The gate count is measured by the two-input one-output NAND gate. The throughput of the proposed serial KK decoder architecture over GF(2^8) and GF(2^{16}) is 4.9 and 13.2 times of that of designs in [43], respectively, while the gate count is only 56% and 76% of that of the designs in [43], respectively. The unfolded decoder architecture achieves a throughput of 12.5Gb/s and 41.6Gb/s, respectively. The latency of the rank metric decoders in [43] over GF(2^8) and GF(2^{16}) is 8.6 and 20.9 times of the proposed serial and unfolded architectures.

The throughput per thousand NAND gates (throughput divided by the gate count in thousands) of the proposed serial and unfolded KK decoder architectures are much higher than the architectures in [43]. The throughput per thousand NAND gates of the unfolded architecture is higher than that of the serial architecture, because the upper bounds on $N_{x,s}$, $N_{y,s}$, $D_{x,j}$, and $D_{y,j}$ above reduce the hardware cost of coefficient registers, interpolators and the polyDiv units of the unfolded architecture.

The proposed serial list decoder architecture is also implemented for a 2-dimensional MV code over GF(4^6), where $k = 3, l = 3, L = 2$. As shown in Table 4.2, the proposed MV decoder achieves a throughput of 193Mb/s at the cost of 341.8k gates.

For KK and MV codes, the code parameters are restricted by the size of the finite

field. The decoding complexity of long KK and MV codes may increase significantly when the field size is large. In this chapter, we focus on KK and MV codes with small parameters to reduce complexity. However, for practical applications, the packet size is usually at the magnitude of several thousands of bytes. The proposed decoder architectures also work for long codes based on Cartesian products of KK or MV codes [43, 46]. For example, suppose the packet length is 1000 bytes, we can use the Cartesian product of 250 $(8, 8, 4)$ KK codes over GF(2^8) to get a code of length 2000 bytes. In this case, the evaluation of 250 polynomials are placed into a single packet. The decoding of long packets is just the decoding of these 250 KK codes. Based on the hardware complexity limit and throughput requirement, we can use one or multiple (8,8,4) KK decoders.

4.6 Conclusion

In this chapter, based on a generalized interpolation and a reformulated right division algorithms, an area efficient serial decoder architecture and a high throughput unfolded decoder architecture are proposed for KK codes. The implementation results show that the proposed decoder architectures are much more efficient than previously proposed rank metric decoder architectures. A serial list decoder architecture for MV codes is also proposed. An M-LRR algorithm is proposed for efficient implementation of the factorization step of the decoding of MV codes. The synthesis results demonstrate that the proposed list decoder architecture for MV codes is feasible for hardware implementation. Future works include developing efficient decoder architecture for fields with large sizes to achieve a good tradeoff between

error correction capability and hardware complexity.

Chapter 5

An Efficient List Decoder Architecture for Polar Codes

5.1 Introduction

Polar codes, recently introduced by Arıkan [18], are a significant breakthrough in coding theory. It is proved that polar codes can achieve the channel capacity of any discrete or continuous memoryless channel [18, 19]. Polar codes can be efficiently decoded by the low-complexity successive cancelation (SC) decoding algorithm [18] with a complexity of $O(N \log N)$, where N is the block length. To approach the channel capacity using the SC algorithm, polar codes require very large code block length (for example, $N > 2^{20}$ [20]), which is impractical in many applications. For short or moderate length, the error performance of polar codes under the SC algorithm is worse than that of Turbo or low-density parity-check (LDPC) codes [21].

Lots of efforts [21–28] have already been devoted to the improvement of error-correction performance of polar codes with short or moderate lengths. An SC list (SCL) decoding algorithm was proposed recently in [21], which performs better than the SC algorithm and performs almost the same as a maximum-likelihood (ML) decoder [21]. In [22–24], the cyclic redundancy check (CRC) is used to pick the output codeword from L candidates, where L is the list size. The CRC-aided SCL algorithm performs much better than the SCL algorithm at the expense of negligible loss in code rate.

In terms of hardware implementations of the SC algorithm, an efficient semi-parallel SC decoder was proposed in [20], where resource sharing and semi-parallel processing were used to reduce the hardware complexity. An overlapped computation method and a pre-computation method were proposed in [29] to improve the throughput and to reduce the decoding latency of SC decoders. Compared to the semi-parallel decoder architecture in [20], the pre-computation based decoder architecture [29] can double the throughput. A simplified SC decoder for polar codes, proposed in [30], reduces the decoding latency by more than 88% for a rate 0.7 polar code with length 2^{18}.

The investigation of efficient list decoder architectures for polar codes is motivated by improved error performance of the SCL and CA-SCL algorithms, especially for polar codes with short or moderate lengths. The tree search list decoder architecture for the SCL algorithm proposed in [31] is the first list decoder architecture for polar codes in the literature to the best of our knowledge. In this chapter, we propose the first hardware implementation of the CA-SCL algorithm to the best of our knowledge. Based on both algorithmic and architectural improvements, our decoder

architecture achieves better error performance and higher area efficiency compared with the decoder architecture in [31]. Specifically, the major contributions of this work are:

1. Message memories account for a significant fraction of an SC or SCL decoder [20, 31]. In this chapter, an area efficient message memory architecture is proposed. Besides, a new compression method for the channel messages is used to reduce the area of the proposed decoder architecture.

2. An efficient processing unit (PU) is proposed. For the proposed list decoder architecture, a fine grained PU profiling (FPP) algorithm is proposed to determine the minimum quantization size of each input message for each PU so that there is no message overflow. By using the quantization size generated by the FPP algorithm for each PU, the overall area of all PUs is reduced.

3. An efficient scalable path pruning unit (PPU) is proposed to control the copying of decoding paths. Based on the proposed memory architecture and the scalable PPU, our list decoder architecture is suitable for large list sizes.

4. A low-complexity direct selection scheme is proposed for the CA-SCL algorithm when a strong CRC is used (e.g. CRC32). The proposed direct selection scheme simplifies the selection of the final output data word.

5. For a (1024, 512) rate-$\frac{1}{2}$ polar code, the proposed list decoder architecture is implemented for list size $L = 2$ and 4, respectively, under a 90nm CMOS technology. Compared with the decoder architecture in [31] synthesized under the same technology, our decoder achieves 1.24 to 1.83 times area efficiency

(throughput normalized by area). Besides, the proposed CA-SCL decoder has better error performance compared with the SCL decoder in [31].

The rest of this chapter is organized as follows. In Section 5.2, polar codes as well as the SCL and CA-SCL algorithms are briefly reviewed. Two improvements of the CA-SCL algorithm are discussed in Section 5.3. The proposed list decoder architecture is described in Section 5.4. Section 5.5 shows the implementation and comparison results of the proposed list decoder architecture. The conclusions are drawn in Section 5.6.

5.2 Polar Codes and Its CA-SCL Algorithm

5.2.1 Polar Codes

A generation matrix of a polar code is an $N \times N$ matrix $G = B_N F^{\otimes n}$, where $N = 2^n$, B_N is the bit reversal permutation matrix [18], and $F = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$. Here $\otimes n$ denotes the nth Kronecker power and $F^{\otimes n} = F \otimes F^{\otimes(n-1)}$. Let $u_0^{N-1} = (u_0, u_1, \cdots, u_{N-1})$ denote the data bit sequence and $x_0^{N-1} = (x_0, x_1, \cdots, x_{N-1})$ the corresponding encoded bit sequence, then $x_0^{N-1} = u_0^{N-1} G$. The indices of the data bit sequence u_0^{N-1} are divided into two sets: the information bits set $\mathcal{A}$ contains K indices and the frozen bits set $\mathcal{A}^c$ contains $N - K$ indices.

5.2.2 SCL and CA-SCL Algorithms

List decoding was applied to the SC algorithm in [21] and the resulting SCL algorithm outperforms the SC algorithm. For a list size L, the SCL algorithm keeps at

Algorithm 17: SCL algorithm [21]

input : n, the received channel message y_0^{N-1}
output: $\hat{u}_0^{N-1}$

for $l = 0$ **to** $L - 1$ **do**
  for $\beta = 0$ **to** $N - 1$ **do**
    $P_{l,0}[\beta][s] = \Pr(y_\beta|s), s = 0, 1$
  for $\lambda = 0$ **to** n **do** $r_l[\lambda] = 0$
for $i = 0$ **to** $N - 1$ **do**
  for $\lambda = \phi^i$ **to** $n - 1$ **do** $r_l[\lambda] = l$
  foreach *survived decoding path* l **do**
    metricComp(l, i)
  if $i \in \mathcal{A}^c$ **then**
    foreach *survived decoding path* l **do**
      $\hat{u}_{l,i} = C_{l,n}[0][i \mod 2] = 0$
  else
    pathPruning$(P_{0,n}, \cdots, P_{L-1,n})$
  if $i \mod 2 == 1$ **then**
    foreach *survived decoding path* l **do**
      pUpdate(l, n, i)

Algorithm 18: metricComp(l, i) [21]

input : l, i

determine $(b_n^{(i)}, b_{n-1}^{(i)}, \cdots, b_1^{(i)})$ and $\phi^{(i)}$

for $\lambda = \phi^{(i)}$ **to** n **do**

  for $k = 0$ **to** $2^{n-\lambda}$ **do**

    if $b_\lambda^{(i)} = 1$ **and** $\lambda = \phi^{(i)}$ **then**

      $s = C_{l,\lambda}[\beta][0]$

      $P_{l,\lambda}[k][u]$

      $= G(P_{r_l[\lambda-1],\lambda-1}[2k], P_{r_l[\lambda-1],\lambda-1}[2k+1], s)$

      $= \frac{1}{2} P_{r_l[\lambda-1],\lambda-1}[2k][u \oplus s] \cdot P_{r_l[\lambda-1],\lambda-1}[2k+1][u]$ for $u \in \{0, 1\}$

    else

      $P_{l,\lambda}[k][u] = F(P_{l,\lambda-1}[2k], P_{l,\lambda-1}[2k+1])$

      $= \sum_{u'=0}^{1} \frac{1}{2} P_{l,\lambda-1}[2k][u \oplus u'] \cdot P_{l,\lambda-1}[2k+1][u']$

      for $u \in \{0, 1\}$

most L decoding paths and outputs L possible data words $\hat{u}_{0,0}^{N-1}, \hat{u}_{1,0}^{N-1}, \cdots, \hat{u}_{L-1,0}^{N-1}$, where $\hat{u}_{l,0}^{N-1} = (\hat{u}_{l,0}, \hat{u}_{l,0}, \cdots, \hat{u}_{l,N-1})$. A low complexity state copying scheme was proposed in [31] to simplify the copying process when a decoding path needs to be duplicated.

For $l = 0, 1, \cdots, L-1$ and $\lambda = 0, 1, \cdots, n$, let $P_{l,\lambda}$ be an array with $2^{n-\lambda}$ elements: $P_{l,\lambda}[j]$ contains two messages $P_{l,\lambda}[j][0]$ and $P_{l,\lambda}[j][1]$ for $j = 0, 1, \cdots, 2^{n-\lambda} - 1$. $C_{l,\lambda}$ has the same structure as $P_{l,\lambda}$: $C_{l,\lambda}[j]$ contains two binary partial sums $C_{l,\lambda}[j][0]$ and $C_{l,\lambda}[j][1]$ for $j = 0, 1, \cdots, 2^{n-\lambda} - 1$. The SCL algorithm with low complexity state copying [31] is reformulated in Algorithm 17. For the decoding of u_i, the SCL algorithm can be divided into the following parts:

- For each surviving decoding path l, compute the path metrics $P_{l,n}[0][0]$ and $P_{l,n}[0][1]$ using the recursive function metricComp(l, i) shown in Algorithm 18.

For $i = 1, 2, \cdots, N-1$, let $(b_n^{(i)}, b_{n-1}^{(i)}, \cdots, b_1^{(i)})$ denote the binary representation of index i, where $i = \sum_{j=0}^{n-1} 2^j b_{n-j}^{(i)}$. $\phi^{(i)}$ $(1 \leq \phi^{(i)} \leq n)$ in Algorithm 18 is the largest integer such that $b_{\phi^{(i)}}^{(i)} = 1$. When $i = 0$, $\phi^{(i)} = 1$. Based on the recursive algorithm for computing path metric in [21] and the low complexity state copying algorithm in [31], the path metric computation is formulated in a non-recursive way in Algorithm 18, where $\mathbf{r}_l = (r_l[n-1], r_l[n-2], \cdots, r_l[0])$ is the message updating reference index array for decoding path l. For decoding path l, $r_l[0] \equiv 0$, while all other elements are initialized with 0. Two types of basic operations, denoted as F and G operations, respectively, are employed in Algorithm 18.

- If u_i is a frozen bit, for each decoding path, the decoded code bit $\hat{u}_{l,i} = 0$, decoding path l will carry on with $\hat{u}_{l,i} = 0$. If u_i is an information bit, decoding path l $(l = 0, 1, \cdots, L-1)$ splits into two decoding paths with corresponding path metrics being $P_{l,n}[0][0]$ and $P_{l,n}[0][1]$, respectively. There are at most $2L$ paths after splitting, and $2L$ associated path metrics. The pathPruning function in Algorithm 17 finds the L most reliable decoding paths based on their corresponding path metrics.

- For each of the L surviving decoding paths, the pUpdate(l, n, i) function shown in Algorithm 19 [21] updates the partial sum matrices that will be used in the following path metric computation.

We make several observations about the path metric computation:

- When $i = 0$, $P_{l,1}, \cdots, P_{l,n}$ are updated in serial, and only the F computation is employed.

- For $i > 0$, $P_{l,\phi^{(i)}}, \cdots, P_{l,n}$ are updated in serial. The G computation is used when computing $P_{l,\phi^{(i)}}$, while the F computation is used for the other probability message arrays.

- The computation of $P_{l,\phi^{(i)}}$ is based on $P_{r_l[\phi^{(i)}-1],\phi^{(i)}-1}$, while the computation of $P_{l,\lambda}$ $(\lambda > \phi^{(i)})$ is based on $P_{l,\lambda-1}$.

Algorithm 19: pUpdate(l, λ, i) [21]

input : l, λ, i

if $\lambda == 0$ **then return**
$j = \lfloor i/2 \rfloor$
for $\beta = 0$ **to** $2^{n-\lambda} - 1$ **do**
  $C_{l,\lambda-1}[2\beta][j \mod 2] = C_{l,\lambda}[\beta][0] \oplus C_{l,\lambda}[\beta][1]$
  $C_{l,\lambda-1}[2\beta+1][j \mod 2] = C_{l,\lambda}[\beta][1]$
if $j \mod 2 == 1$ **then** pUpdate$(l, \lambda - 1, j)$

$$G(P_{r_l[\lambda-1],\lambda-1}[2k], P_{r_l[\lambda-1],\lambda-1}[2k+1], s) = P_{r_l[\lambda-1],\lambda-1}[2k][u \oplus s] + P_{r_l[\lambda-1],\lambda-1}[2k+1][u] \tag{5.1}$$

$$F(P_{l,\lambda-1}[2k], P_{l,\lambda-1}[2k+1]) = \max^*(P_{l,\lambda-1}[2k][u] + P_{l,\lambda-1}[2k+1][0], P_{l,\lambda-1}[2k][u \oplus 1] + P_{l,\lambda-1}[2k+1][1]) \tag{5.2}$$

$$F(P_{l,\lambda-1}[2k], P_{l,\lambda-1}[2k+1]) = \max(P_{l,\lambda-1}[2k][u] + P_{l,\lambda-1}[2k+1][0], P_{l,\lambda-1}[2k][u \oplus 1] + P_{l,\lambda-1}[2k+1][1]) \tag{5.3}$$

The path pruning function, pathPruning, finds the L most reliable paths, a_0, a_1, $\cdots$, a_L, and their corresponding decoded bits, $c_0, c_1, \cdots, c_L$, based on the path metrics. The path metrics of the surviving L decoding paths are the L largest ones among $2L$ input metrics. Once the surviving decoding paths are found, the partial sums and the reference index array of decoding path l will copy from decoding path a_l. The partial sum computation of decoding path l is carried on with the binary input c_l.

The pruning scheme in this chapter and the path pruning scheme in [28] both try to eliminate decoding paths that are less reliable. However, there are still some differences:

- The pruning scheme in [28] is used for successive cancelation stack (SCS) decoding algorithm as well as the SCH decoding algorithm, which is a hybrid of SCL and SCS decoding algorithms, whereas our pruning scheme is used for the SCL algorithm.

- For the SCL algorithm, suppose there are L decoding paths before the decoding of u_i, then the metrics of $2L$ expanded decoding paths are computed. The pruning scheme in this chapter finds the L largest metrics out of $2L$ metrics and keeps their corresponding decoding paths. For the pruning scheme in [28], a path will be deleted if its path metric is smaller than a dynamic threshold, $a_i - \ln(\tau)$, where a_i is the largest metric of candidate paths, and τ is a configuration parameter.

- For the path pruning scheme in [28], the number of deleted paths is not fixed and depends on the configuration parameter τ, while the number of deleted

paths is always L for the pruning scheme in this chapter.

The F and G operations in Algorithm 18 are in probability domain. The F and G operations in Algorithm 18 can also be performed over the logarithm domain [23]. For $u \in \{0, 1\}$, the resulting logarithm domain G and F computations are shown in Eq. (5.1) and Eq. (5.2), respectively, where $\max^*(x, y) = \max(x, y) + \log(1 + e^{-|x-y|})$. $\max^*(x, y)$ can also be approximated with $\max(x, y)$, resulting in the approximated F computation in Eq. (5.3).

In [22], the performance of the SCL algorithm is further improved by the adoption of CRC, which helps to pick the right path from the L possible decoded data words. In terms of the fixed point implementation, the CA-SCL algorithm is quite sensitive to saturation. For two decoding paths, it is hard to decide which is better if the metrics of both paths are saturated. In order to avoid message saturation, a non-uniform quantization scheme is proposed in [31]. If the channel messages ($P_{l,0}$) are all quantized with t bits, all the log-likelihood messages (LLMs) of $P_{l,\lambda}$ need to be quantized with $t + \lambda$ bits in order to avoid saturation.

5.3 Two Improvements of the CA-SCL Algorithm

In this chapter, two improvements of the CA-SCL algorithm are proposed. Firstly, for the i-th received bit y_i, there are two likelihoods, $\Pr\{y_i|0\}$ and $\Pr\{y_i|1\}$. Suppose $\Pr\{y_i|m\}$ ($m \in \{0, 1\}$) is the smaller one among the two likelihoods. For $j \in \{0, 1\}$, two log-likelihood messages (LLMs) are defined as

$$P_{l,0}[i][j] = \log \frac{\Pr\{y_i|j\}}{\Pr\{y_i|m\}}. \tag{5.4}$$

Thus one of the LLMs is always 0, and the other is always non-negative. For the proposed list decoder, only the non-negative LLM and its corresponding binary index s are stored. As shown in Fig. 5.1, Msg denotes the stored non-negative LLM, and its corresponding bit index is s. When $s = 0$, $P_{l,0}[i][0] = \text{Msg}$, $P_{l,0}[i][1] = 0$. When $s = 1$, $P_{l,0}[i][0] = 0$, $P_{l,0}[i][1] = \text{Msg}$. If t bits are needed to quantize a channel LLM, it takes $t + 1$ bits to represent two LLMs corresponding to a received bit y_i, while it takes $2t$ bits to store two LLMs directly.

Figure 5.1: Compressed channel message

Secondly, at the end of the CA-SCL decoding algorithm, the candidate data word that passes the CRC and has the greatest path metric is the output data word, which will incur additional comparisons. In this chapter, a simple direct selection scheme is proposed: we first calculate all L checksums in parallel and then scan from the checksum of data word 0 to the checksum of data word $L - 1$, if a data word passes the CRC, the scan process is terminated and the corresponding candidate data word is the final output one. When all L CRC checks fail, since the CRC checksum could be corrupted, a decoding failure is announced if re-transmission is possible; otherwise, pick a data word randomly and output.

The direct selection scheme reduces computational complexity at the expense of possible performance degradation. In this chapter, we give an estimation of the frame error rate (FER) degradation. Let w denote the number of the detectable errors for our CRC. Assume all the bits of the final L candidate data words are independently subject to a bit error probability, p_b. We calculate the increase in

FER, P_e, caused by the direct selection scheme instead of the ideal selection scheme, which always selects the transmitted data word if it is within the final L candidates. For each candidate data word, there are three probabilities:

- The probability that the candidate data word is the same as the transmitted one is givn by $p_1 = (1 - p_b)^K$.
- The probability that the candidate fails the CRC is denoted as p_2.
- The probability that the CRC identifies the candidate as the transmitted data word by mistake is denoted as p_3, and $p_3 \doteq \sum_{r=w+1}^{K} \binom{K}{r} p_b^r (1 - p_b)^{K-r} \doteq \binom{K}{w+1} p_b^{w+1} (1 - p_b)^{K-w-1}$.

Thus, $p_1 + p_2 + p_3 = 1$. We have

$$P_e \leqslant p_3 \frac{1 - p_2^L}{1 - p_2} + p_2^L - (1 - p_1)^L. \tag{5.5}$$

Note that p_b depends on the signal to noise ratio (SNR) and the list size L. For a specific SNR, in order to simplify our analysis, we can use $p_{b,\mathrm{SC}}$ to approximate p_b, where $p_{b,\mathrm{SC}}$ denotes the bit error probability of the SC algorithm. The probabilities, p_2 and p_3, are also approximated. Though approximated probabilities are employed when calculating P_e, the order of P_e still helps us in determining whether our direct selection scheme is applicable. For instance, when a strong CRC is used, i.e. large w, p_3 is small, leading to a small P_e. On the other hand, a higher data rate leads to a greater K and hence a greater P_e.

5.3.1 Numerical Results

For a rate 1/2 polar code with $N = 1024$, the FERs of the SC, SCL and CA-SCL algorithms are shown in Fig. 5.2, where SC denotes the floating-point SC algorithm. CS2-max and CS2-map denote the floating-point CA-SCL algorithm with $L = 2$ and the approximated F computation shown in Eq. (5.3) and the F computation shown in Eq. (5.2), respectively. CSi-max-j denotes the fixed-point CA-SCL algorithm with $L = i$ and non-uniform quantization scheme with $t = j$, where t is the number of quantization bits for channel probability message. Si-max-j denotes the fixed-point SCL algorithm with $L = i$ and non-uniform quantization scheme with $t = j$. For all simulated CA-SCL algorithms, a CRC scheme with a generator polynomial 0x1EDC6F41 is employed, and the direct selection scheme is employed to pick the final output codeword from L possible candidates.

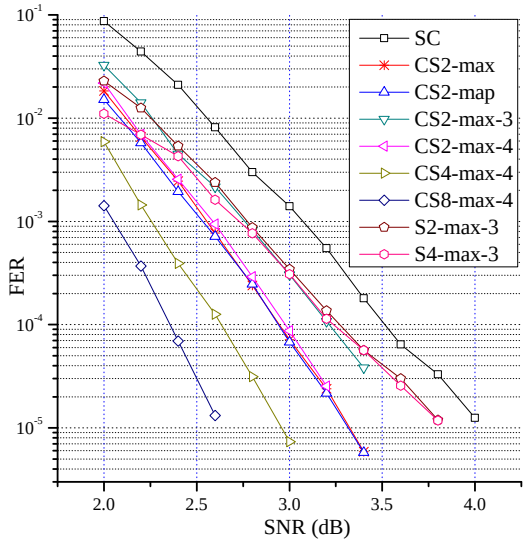

Figure 5.2: FER performance of a polar code with $N = 1024$

The simulated results show that:

- For the CA-SCL algorithm, the approximated F computation in Eq. (5.3) results in negligible performance degradation.

- When each channel LLM is quantized with 4 bits, the employment of the proposed non-uniform quantization scheme leads to negligible performance degradation. When each channel LLM is quantized with 3 bits, the resulting FER performance is roughly 0.2dB worse than that using 4-bit quantization.

- Using a larger list size leads to obvious performance improvement for the CA-SCL algorithm, whereas the SCL algorithm with $L = 2, 4$ has nearly the same performance, especially in the high SNR region. For polar codes with moderate block length (e.g. $N = 2^{11}, 2^{12}, 2^{13}$), similar phenomena have been observed in [22].

More simulation results on the proposed direct selection scheme are provided. There are three selection schemes employed in our simulations.

- The proposed direct selection (DS) scheme, which outputs the first data word that passes CRC.

- Ideal selection (IS) scheme, which always outputs the correct data word if it exists in the final list.

- Metric based selection (MS) scheme [22], which outputs the data word that has the maximal path metric among all data words that have passed CRC.

In Figs. 5.3 and 5.4, DSk, ISk and MSk denote the CA-SCL algorithms with list size $L = k$ under the direct, ideal and metric based selection schemes, respectively. The generation polynomial of the CRC16 used in our simulations is 0x1021.

As shown in Fig. 5.3, when code rate is 0.75 and CRC16 is used, the proposed direct selection scheme introduces early error floor for all simulated list sizes, while the metric based selection scheme performs nearly the same as the ideal selection scheme. When code rate is 0.5, as shown in Fig. 5.4, the direct selection scheme performs nearly the same as the ideal selection scheme with list size $L = 2$. When list size $L = 4, 8, 16$, the proposed direct selection scheme shows certain performance degradation compared with the ideal selection scheme, while the metric based selection scheme has little performance degradation. When CRC32 is used, the proposed direct selection scheme performs nearly the same as the ideal selection scheme for both code rates 0.5 and 0.75 [74].

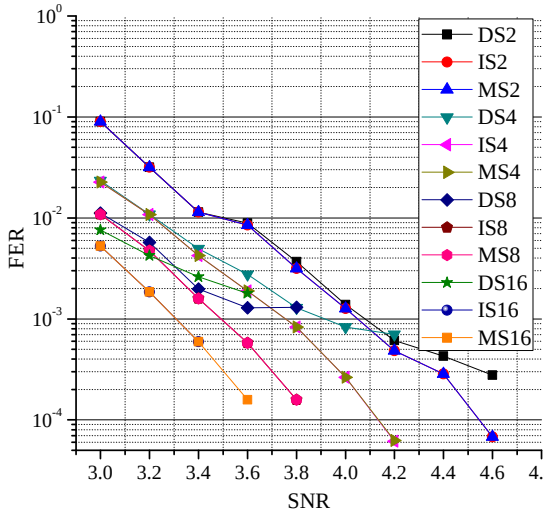

Figure 5.3: FER performances under CRC16 and rate 0.75

We also calculate the bound on the FER degradation for all simulated cases. We choose SNR = 3.6dB, since DS4, DS8 and DS16 begin to show an error floor at this SNR in Fig. 5.3. For the length 1024 polar code, the bit error probability p_b of the SC algorithm is 6.28×10^{-4} and 3.04×10^{-6} for rate 0.75 and 0.5, respectively. For

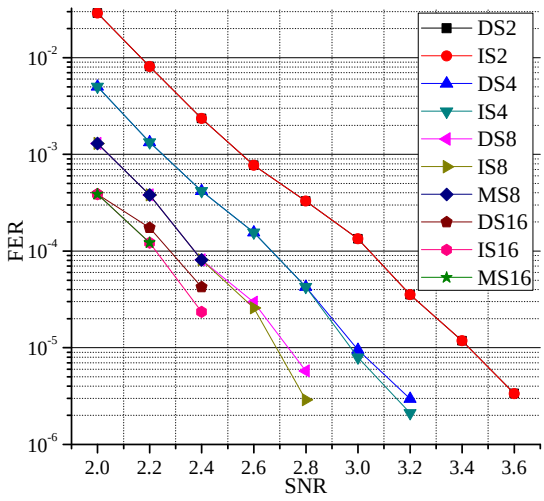

Figure 5.4: FER performances under CRC16 and rate 0.5

CRC16 and CRC32, $w = 2$ [75] and 4 [76], respectively. When CRC16 is used, for each simulated list size, the bound is around 10^{-2} and 10^{-10} for rate 0.75 and 0.5, respectively. When CRC32 is used, for each simulated list size, the bound is 10^{-4} and 10^{-17} for rate 0.75 and 0.5, respectively. It is found that the error degradation caused by our DS scheme is big when the corresponding P_e is big (e.g. 10^{-2}). On the other hand, when P_e is quite small (e.g. 10^{-17}), our DS scheme leads to little performance degradation.

Based on our calculation results, for a given CRC and code rate, P_e increases with the list size L. This observation indicates that the potential performance degradation caused by the DS scheme will increase when L increases. This is consistent with the simulation results shown in Figs. 5.3 and 5.4.

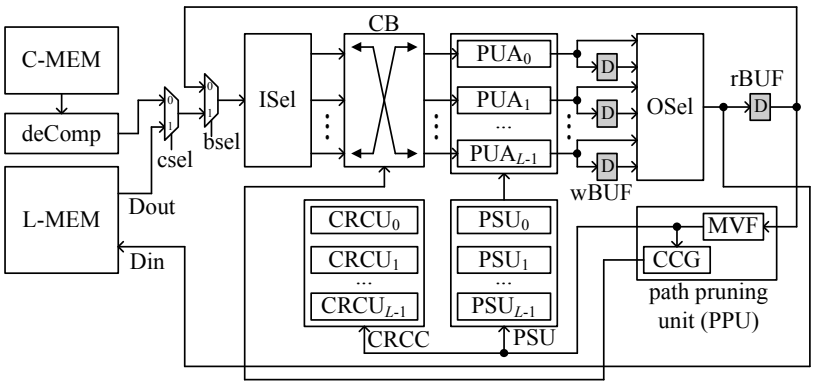

Figure 5.5: Top architecture of the list decoder

5.4 Efficient List Decoder Architecture

For the CA-SCL algorithm, we propose an efficient partial parallel list decoder architecture shown in Fig. 5.5. The proposed list decoder architecture mainly consists of the channel message memory (C-MEM), the internal LLM memory (L-MEM), L processing unit arrays (PUAs) (PUA_0, PUA_1, $\cdots$, PUA_{L-1}), the path pruning unit (PPU) and the CRC checksum unit (CRCU). These components are described in details in the following subsections.

5.4.1 Message Memory Architecture

The L-MEM stores all the inner LLMs used for metric computation. Since all the LLMs in $P_{l,\lambda}$ need to be quantized with $t + \lambda$ bits for $\lambda \geq 1$, the variable-size LLMs make the L-MEM architecture for the proposed list decoder nontrivial. In this chapter, an area efficient scalable memory architecture for L-MEM is proposed based on the nonuniform quantization:

- For $\lambda = 1, 2, \cdots, n$, since each LLM within $\mathbf{P}_\lambda = (P_{0,\lambda}, P_{1,\lambda}, \cdots, P_{L-1,\lambda})$ is

quantized with $t + \lambda$ bits, a regular sub-memory is created for storing LLMs in $\mathbf{P}_\lambda$.

- All n sub-memories are combined to a single memory.
- Due to the nonuniform quantization, the width of each sub-memory may be different. As a result, the concatenated L-MEM is an irregular memory with varying width within its address space. For the proposed memory architecture, the irregular L-MEM is split into several regular memories to fit current memory generation tools.

The proposed L-MEM is a mix of different types of memories, including SRAM, register file (RF) or register. Since SRAM and RF are more area efficient than a register, the proposed L-MEM architecture is better than the register based LLM memory in [31].

Suppose there are T processing units (PUs) in each PUA shown in Fig. 5.5, it consumes at most $4LT$ LLMs for one round of computation. For $\lambda = 1, 2, \cdots, n$, we store all the LLMs within $\mathbf{P}_\lambda = (P_{0,\lambda}, P_{1,\lambda}, \cdots, P_{L-1,\lambda})$ in a single memory as follows.

- When $2^{n-\lambda+1}L > 4LT$, it takes a sub-memory of $\frac{2^{n-\lambda-1}}{T}$ words, where each word has $4LT(t + \lambda)$ bits.
- When $2^{n-\lambda+1}L \leqslant 4LT$, it takes a sub-memory with only one single word, which has $2^{n-\lambda+1}(t + \lambda)L$ bits.

An example of the concatenation of $n = 6$ sub-memories, (S_1, S_2, $\cdots$, S_6), is shown in Fig. 5.6(a). For current memory compiler, it is hard to generate an irregular single

memory instance as shown in Fig. 5.6(a). For the proposed L-MEM architecture, the concatenated irregular memory is split into several regular memory instances as shown in Fig. 5.6(b), where additional dummy memories are added so that each instance is regular. For general cases, the irregular memory is divided into $\lambda_o = n - \log_2 T - 1$ regular instances. Depending on the number of words, each memory instance could be implemented with SRAM, RF or registers.

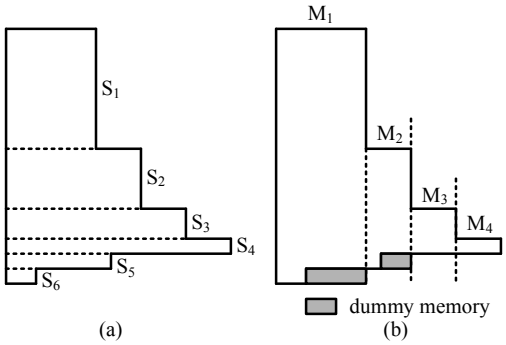

Figure 5.6: The split of an irregular LLM memory

Compared with the register based LLM memory, the proposed L-MEM architecture is more area efficient because:

- Some sub-memory instances can be implemented with SRAM or RF which is more denser than register based memory.
- As shown in Fig. 5.6(b), most of the LLMs are store in the largest memory instance M_1 which contains

$$N_w = n - \lambda_o + 1 + \sum_{\lambda=1}^{\lambda_o - 1} \frac{2^{n-\lambda-1}}{T} \tag{5.6}$$

words, where each word has $4LT(t+1)$ bits.

As shown in Eq. (5.6), N_w is inverse to the number of processing units, T, within a PUA. As a result, the area of the proposed L-MEM depends on T for a fixed block length $N = 2^n$ and t. Taking RF as an example, we show the comparison of area efficiency of RFs with different depths in Table 5.1, where area per bit (APB) denotes the total area of a memory normalized by the number of total stored bits. The total areas shown in Table 5.1 are from a memory compiler associated with a 90nm technology. As shown in Table 5.1, the RF with a larger depth has a smaller APB. Hence, given the same amount of bits, it takes a smaller area if those bits can be stored in a RF with a larger depth. For SRAM, the same phenomena have been observed.

Table 5.1: Area per Bit for RFs with Different Depth and Width 128 using TSMC 90nm CMOS technology

depth	8	16	32	64	128
total area (μm^2)	24331	27022	32308	42812	63811
APB (μm^2)	23.7	13.1	7.9	5.2	3.89

The C-MEM can be implemented with a simple regular memory with $\frac{N}{2T}$ words, where each word has $2T(t+1)$ bits. Due to the compression of the channel message, each compressed channel message is de-compressed into two LLMs by the deComp unit in Fig. 5.5 before being fed to the PUs. The deComp unit can be implemented with multiplexors.

5.4.2 Processing Unit Array

The G and approximated F computations shown in Eq. (5.1) and Eq. (5.3), respectively, are used in the metric computation. These two types of basic operation can be performed with a PU [31, 74]. The hardware complexity of the proposed PU is

determined by p, which is the width of an input LLM.

Due to the non-uniform quantization of the LLMs belonging to different message arrays, for each PU, the number of quantization bits, p, for each input LLM should be large enough so that no overflow will happen. According to the fixed point implementation of the CA-SCL algorithm, the quantization of $P_{l,n}$ ($l = 0, 1, \cdots, L-1$) needs the most binary bits, which is $t + n$. For each PUA, it is unnecessary to employ T PUs with $p+1 = t+n$. In this chapter, a fine grained PU profiling (FPP) algorithm, shown in Algorithm 20, is proposed to decide p for each PU.

Algorithm 20: FPP Algorithm

input : $n, t, \lambda_o = n - \log_2 T - 1$
output: $p[0], p[1], \cdots, p[T-1]$

for $j = 0$ **to** $T - 1$ **do**
  $p[j] = t + \lambda_o - 1$
for $\lambda = \lambda_o + 1$ **to** n **do**
  for $j = 0$ **to** $2^{n-\lambda} - 1$ **do**
    $p[j] = t + \lambda - 1$

Table 5.2: Bit width of LLM Inputs of $\mathrm{PU}_{l,j}$ when $n = 10$, $T = 8$ and $t = 4$

j	0	1	2	3	4	5	6	7
$p[j]$	13	12	11	11	10	10	10	10

Table 5.3: Area comparison between fine grained PU array and regular PU using TSMC 90nm CMOS technology

n	10				15			
T	8	16	32	64	32	64	128	256
CPD (ns)	0.555				0.588			
regular PU array area (μm^2)	27650	55259	113902	225418	150951	308640	602509	1212359
fine grained PU array area (μm^2)	19280	34131	59048	101377	104434	190615	334927	594048
area saving	30%	38%	48%	55%	30%	38%	44%	51%

For the j-th PU of PUA_l ($l = 0, 1, \cdots, L-1$), each LLM input is quantized with $p[j]$ bits. The proposed FPP algorithm is based on the observation that only $2^{n-\lambda} < T$ PUs are needed when computing the updated $P_{l,\lambda}$ with $\lambda > \lambda_o$. Thus, in the proposed PUA_l, only $\text{PU}_{l,0}$, $\text{PU}_{l,0}$, $\cdots$, $\text{PU}_{l,2^{n-\lambda}-1}$ are enabled for the computing of $P_{l,\lambda}$. Based on the proposed FPP algorithm, each PUA can finish the metric computation without any overflow at the cost of less area consumption. As shown in Algorithm 20, the bit width of the LLM inputs of a PU is determined by n, T and t. One example is shown in Table 5.2, where $n = 10$, $T = 8$ and $t = 4$.

The area saving due to the proposed fine grained profiling algorithm also depends on T, n and t. For the proposed list decoder architecture, there are L identical PU arrays, where each array contains T PUs. In Table 5.3, we compare the area of a regular PU array with that of an array where the input message width of each PU is determined by the fine grained profiling algorithm. As shown in Table 5.3, the area of PU arrays is reduced by 30% to 55% depending on the number of PUs with an array and the block length $N = 2^n$. Here, each channel message is quantized with $t = 4$ bits.

Metric Computation Schedule

For the proposed L-MEM, each data word is capable of storing $4TL$ LLMs. Moreover, each word is equally divided into L consecutive parts, where the l-th part stores the LLMs corresponding to decoding path l. The metric computation schedule is almost the same as that of the partial parallel SC decoder in [20] except that L PUAs work simultaneously for L decoding paths, respectively.

When a data word needs to be updated, the write mismatch would happen since

L PUAs generate only $2LT$ updated LLMs during one clock cycle. These L PUAs need to read two consecutive data words from L-MEM in order to generate $4TL$ LLMs. For the proposed list decoder architecture, as shown in Fig. 5.5, L write buffers (wBUFs) are employed to store half of $4TL$ LLMs generated by L PUAs. Once the remaining LLMs are computed, the output selection (OSel) module formats these LLMs in the way that these LLMs are stored in the L-MEM.

Since all the LLMs belonging to $\mathbf{P}_\lambda = (P_{0,\lambda}, P_{1,\lambda}, \cdots, P_{L-1,\lambda})$ with $\lambda > \lambda_o$ are stored in a single data word in L-MEM and the computing of LLMs belonging to $\mathbf{P}_{\lambda+1}$ can only take place once $\mathbf{P}_\lambda$ are updated, an additional clock cycle is needed to read out the LLMs within $\mathbf{P}_\lambda$ that have been just written into the L-MEM. This will increase the delay and decrease the throughput of the proposed list decoder. As shown in [20], the bypass buffer, rBUF, is used to temporarily store the messages written into the L-MEM and eliminate the extra read cycle.

5.4.3 Path Pruning Unit

For the CA-SCL algorithm, once the path metric computation of decoding step i is finished, each current decoding path splits into two sub decoding paths. However, the list decoder keeps at most L decoding paths. For the proposed list decoder architecture, a path pruning unit (PPU) is proposed to prune the split decoding paths in an efficient way. As shown in Fig. 5.5, the proposed PPU contains two sub modules, the maximum value filter (MVF) and the crossbar control signals generator (CCG). The MVF generates L path indices $a_0, a_1, \cdots, a_{L-1}$ and L associated decoded bits $c_0, c_1, \cdots, c_{L-1}$. For a current decoding path l, both the path metric and partial sum computations will be based on the LLMs and partial sums within decoding path a_l,

and the decoded data bit for $u_{l,i}$ is c_l. a_l and c_l for $l = 0, 1, \cdots, L-1$ are used to control the copying of partial sums and checksums.

Table 5.4: Comparison of ASIC implementation results using TSMC 90nm CMOS technology

	metric sorter [31]					proposed MVF				
L	2	4	8	16	32	2	4	8	16	32
CPD (ns)	0.45	0.85	1.8	4.1	9.6	0.54	1.25	2.25	3.7	5.2
area (μm^2)	1995	9199	47119	241633	1392617	1580	8401	30814	96979	319498
area saving	–					20%	8%	34%	59%	77%

Maximum Values Filter

Taking list size $L = 8$ as an example, the proposed MVF architecture, shown in Fig. 5.7, consists of a bitonic sequence generator (BSG) and a stage of compare and select (CAS) modules. The BSG has 16 inputs $(D_0, D_1, \cdots, D_{16})$ and 16 outputs $(S_0, S_1, \cdots, S_{16})$. Each input or output consists of three parts: the path metric, the associated list index and decoded bit. The width of each input and output is $z = x_1 + x_2 + 1$, where $x_1 = t + n$ is the number of bit used to quantize a path metric and $x_2 = \log_2 L$ is the number of bits used to represent a list index.

Each stage of the BSG consists of $\frac{L}{2}$ increase-order sorters (ISs) and $\frac{L}{2}$ decrease-order sorters (DSs), which are shown in Fig. 5.8(a) and Fig. 5.8(b), respectively. Both the IS and DS have two inputs and two outputs. For $k = 0, 1$, $SI_k = (LR_k, l_k, b_k)$, $SO_k = (LR'_k, l'_k, b'_k)$ and LR_k, l_k and b_k denote the path metric and its corresponding list index and decoded bit. The IS reorders the inputs such that path metric $LR'_0 \leq LR'_1$. The output of the comp-max module is 1 when $LR_0 > LR_1$. The DS reorders the inputs such that $LR'_0 \geq LR'_1$ and the output of the comp-min module is 1 when $LR_0 < LR_1$.

The BSG reorders the inputs based on the magnitude of path metrics. Let $LS_r(r = 0, 1, \cdots, 15)$ denotes the associated path metric of output S_r, the path metrics of the 16 outputs satisfy:

$$LS_0 \leq LS_1 \leq \cdots \leq LS_7, \tag{5.7}$$

$$LS_8 \geq LS_9 \geq \cdots \geq LS_{15}. \tag{5.8}$$

It is proved in [77] that the 8 maximum values among LS_i's are $\max(LS_r, LS_{8+r})$ for $r = 0, 1, \cdots, 7$. Hence, a stage of CAS modules is appended at the outputs of BSG shown in Fig. 5.7, where CSA_r takes S_r and S_{r+8} as inputs. This stage of CAS modules produce the outputs $O_t = (a_l, c_l)$ for $l = 0, 1, \cdots, L-1$. As shown in Fig. 5.8(c), the CAS module compares the path metrics of its two inputs and selects the corresponding list index and bit value whose associated path metric is larger.

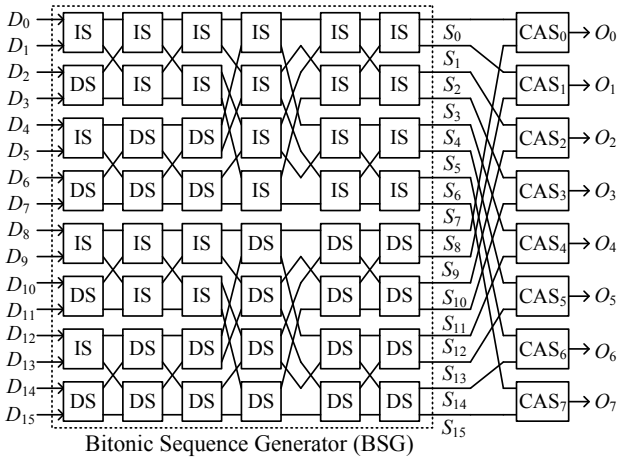

Figure 5.7: Maximum values filter architecture

The metric sorter in [31] has the same function as that of the proposed MVF. We compare the proposed bitonic sorter based MVF module with the metric sorter [31]

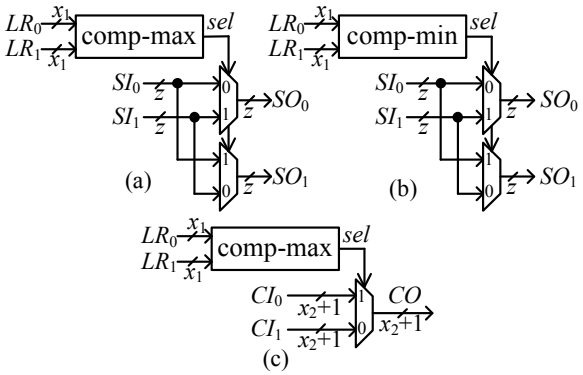

Figure 5.8: (a) Architectures of IS (b) Architectures of DS (c) Architectures of CAS ($z = x_1 + x_2 + 1$)

in terms of area and critical path delay (CPD) under different list sizes when both modules are synthesized under the TSMC 90nm CMOS technology. As shown in Table 5.4, the proposed MVF module is more suitable for large list sizes. For list size $L = 2$ to 32, the proposed MVF achieves 8% to 77% area saving. The proposed MVF architecture achieves area saving because the comparator dominates the area for the metric sorter and the MVF modules. For list size L, the metric sorter needs $N_{MS} = L(2L-1)$ comparators, while the proposed MVF module needs $N_{MVF} = 1 + 2 + \cdots + \log_2 L = \frac{L}{2}((\log_2 L)^2 + \log_2 L + 2)$ comparators. When L is large, $N_{MS}/N_{MVF} \approx \frac{4L}{\log_2 L}$. Clearly, our MVF module needs fewer comparators.

When $L = 2, 4, 8$, compared with the metric sorter, the proposed MVF has longer CPD while achieving area saving. However, the longer delay for the MVF is inconsequential because it is not in the critical path for the decoder architecture when $L \leqslant 8$. When $L = 16, 32$, the proposed MVF is better than the metric sorter in terms of both area and CPD. Thus, the proposed MVF is more suitable for large list sizes.

Crossbar Control Signal Generator

Due to the lazy copy method [31], when decoding path l needs to be copied to decoding path l', instead of copying LLMs from path l to path l', the index references ($\mathbf{r}_l = (r_l[n-1], \cdots, r_l[0])$ shown in Algorithm 18) to LLMs of path l are copied to path l'. For decoding path l, when PUA_l is computing updated LLMs in $P_{l,\lambda}$, the crossbar (CB) module shown in Fig. 5.5 selects input LLMs from decoding path $r_l[\lambda - 1]$. The CB can be implemented with L-to-1 multiplexors.

The crossbar control signal (CCG) generator computes the control signals of CB, cc_0, cc_1, $\cdots$, $\text{cc}_L - 1$, where the l-th output of CB is connected to the cc_l-th input. Since the CCG is a direct implementation of the lazy copy scheme in [31], the details are omitted and can be found in our extended manuscript [74].

5.4.4 Partial Sum Update Unit and the CRC Unit

In this chapter, a parallel partial sum update unit (PSU) is proposed to provide the partial sum inputs to L PUAs when performing the G computation. Compared with the PSU in [20, 31], which needs $N - 1$ single bit registers for a decoding path, our PSU needs only $\frac{N}{2} - 1$ single register bits.

Take $N = 2^3$ as an example, the architecture of PSU_l, which computes the partial sums for decoding path l, is shown in Fig. 5.9, where stage_3 and stage_2 have one and two elementary update units (EUs), respectively. $r_{l,3,0}$, $r_{l,2,0}$, $r_{l,2,1}$ shown in Fig. 5.9 are single bit registers. $c_l = \hat{u}_{l,i}$ is the binary input of the PSU_l. There are three partial sum outputs: $b_{l,3}$, $b_{l,2}$ and $b_{l,1}$ with a width of 1, 2 and 4 bits, respectively. When the LLMs in $P_{l,\lambda}$ need to be updated with the G computation, $b_{l,\lambda}$ is the corresponding partial sum input. The architectures of PSU_l for other

code lengths can be derived from the architecture in Fig. 5.9. For a polar code with length $N = 2^n$, the corresponding PSU_l contains $n-1$ stages: stage_n, stage_{n-1}, $\cdots$, stage_2, where stage_j has 2^{n-j} EUs for $n \geq j \geq 2$.

When bit index i is even, c_l is stored in $r_{l,n,0}$ and other registers keep their current values unchanged. When bit index i is odd, bit registers in stage_n, stage_n, $\cdots$, $\text{stage}_{\phi(i+1)}$ are updated with their corresponding input. When decoding path index $l \neq a_l$, the updated partial sums of decoding path l should be computed based on the bit registers in PSU_{a_l}. The switch network (SW) shown in Fig. 5.9 selects the corresponding bit register value from PSU_{a_l}. The width of the input signal $B_{l,j,k} = \{r_{0,j,k}, r_{1,j,k}, \cdots, r_{L-1,j,k}\} \backslash \{r_{l,j,k}\}$ is $L-1$ bits.

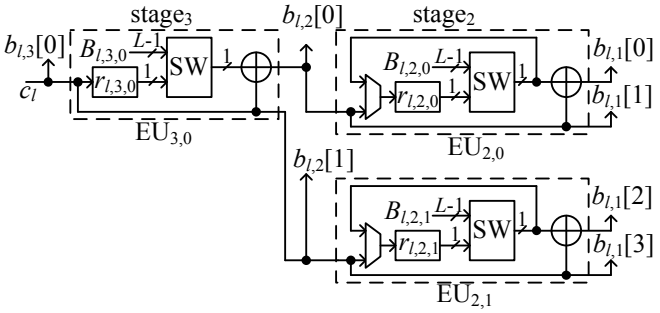

Figure 5.9: PSU architecture

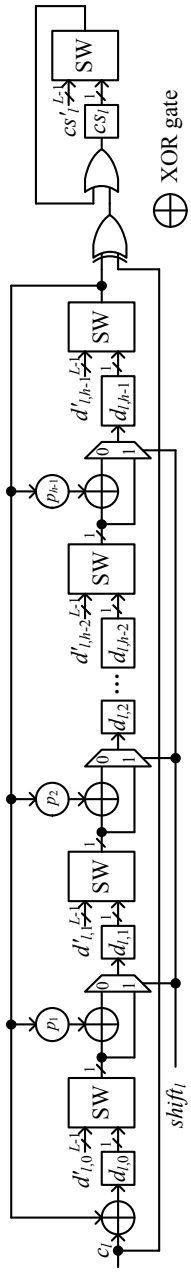

Figure 5.10: Architecture of the proposed CRC unit

The CRC unit (CRCU) checks whether a data word passes the CRC. Suppose an h-bit CRC checksum is used, the architecture of the CRCU_l for decoding path l is shown in Fig. 5.10, where the generation polynomial for the CRC checksum computation is $p(x) = x^h + p_{h-1}x^{h-1} + \cdots + p_1 x + 1$. The proposed CRCU_l is based on a well known serial CRC computation architecture [78]. If the polynomial coefficient $p_k = 0$, the corresponding XOR gate and multiplexer are removed. During the decoding of the first $K - h$ information bits, the control signal $shift_l = 0$ and CRCU_l computes the h-bit checksum of these information bits. The checksum is stored in bit registers $d_{l,0}, d_{l,1}, \cdots, d_{l,h-1}$ shown in Fig. 5.10. When a frozen bit is being decoded, $d_{l,0}, d_{l,1}, \cdots, d_{l,h-1}$ will not be updated. Once the checksum computation is finished, the checksum is compared with the remaining h decoded information bits, and the control signal $shift_l = 1$. The checksum and the remaining h code bits are compared bit by bit. The comparison result is stored in the register cs_l. The decoded codeword for decoding path l passes the CRC only if $cs_l = 0$. The SW module shown in Fig. 5.10 is the same as that used in the partial sum computation unit PSU_l. When $l \neq a_l$, the SW module selects $d_{a_l,k}$ for $k = 0, 1, \cdots, h-1$.

Table 5.5: Implementation Results With $R' = 0.468$ and $R = 0.5$

	proposed architecture				[31]†				[31]‡	
algorithm	CA-SCL				SCL					
list size L	2		4		2		4		2	4
total number of PUs LT	16	32	32	64	16	32	32	64	128	256
channel message quantization bits t	4								3	
process	TSMC 90nm								UMC 90nm	
frequency (MHz)	500	500	454	476	699	757	684	694	459	314
total area (mm^2)	0.406	0.553	0.810	1.132	1.114	1.174	2.181	2.197	1.60	3.53
N_C	3200	2816	3200	2816	3200	2816	3200	2816	2592	2592
latency (ms)	6.4	5.63	7.04	5.91	4.57	3.71	4.67	4.05	5.64	8.25
throughput (Mbps)	$160R'$	$181R'$	$145R'$	$173R'$	$224R$	$275R$	$219R$	$252R$	$181R$	$124R$
area efficiency (Mbps/mm^2)	$394R'$	$327R'$	$179R'$	$152R'$	$201R$	$234R$	$100R$	$114R$	$113R$	$35R$
normalized area efficiency	1.83	1.30	1.67	1.24	1	1	1	1	-	

‡Original synthesis results based on a UMC 90nm technology in [31].
†Synthesis results based a TSMC 90nm technology, provided by the authors of [31].

5.4.5 Decoding Cycles

For the proposed list decoder, pipeline registers can be inserted in the paths that pass through the MVF. Let N_C denote the number of cycles spent on the decoding of one data word. For list decoder architectures based on partial parallel processing [20],

$$N_C = 2N + \frac{N}{T}\log_2\frac{N}{4T} + n_p RN, \tag{5.9}$$

where N, T, n_p, R denote the block length, the number of PUs per decoding path, the number of pipeline registers inserted in the path pruning unit and the code rate, respectively.

The corresponding throughput $TP = \frac{fNR}{N_C}$, where f is the frequency of the list decoder. The latency $T_D = \frac{N_C}{f}$.

5.4.6 Scalability of the Proposed List Decoder Architecture

For our list decoders, in term of error performance, it is desirable to use large list sizes since a larger L leads to more performance gain for the CA-SCL decoding algorithm. For the current list decoder architecture in [31], two problems arise when L increases.

- The message memories of the list decoder in [31] are built with registers due to the non-uniform quantization of the logarithm domain messages. Besides, the message memories dominate the whole decoder area. As a result, the memory area of the list decoder is linearly proportional to list size L. For a larger list size, the list decoder architecture in [31] will suffer from large area and high power consumption due to its register based memory.

- As shown in Table 5.4, when the list size grows, the metric sorter suffers from large area and long critical path delay, which results in a slower clock frequency of the list decoder. If multiple pipelines are inserted in the metric sorter, the number of cycles for decoding one codeword also increases as shown in Eq. (5.9).

For our list decoder architecture, these two issues are solved as follows.

- The proposed memory architecture is more area efficient compared to register based memory. Besides, the proposed memory architecture offers a tradeoff between data throughput and memory area. The register based memory [31] remains almost unchanged when the number of PUs changes. However, for the proposed memory architecture, the number of PUs affects the depth-width ratio of the message memories. Hence, the area of message memory can be tuned by varying the number of PUs. Reducing the number of PUs will increase the depth of message memories, which is more area efficient. On the other hand, reducing the number of PUs will also increase the number of cycles used on decoding one codeword and decrease the data throughput.

- When the list size increases, the proposed MVF is more area efficient and has a shorter critical path delay compared with the metric sorter [31].

As shown in Eq. (5.6), the depth of the largest LLM memory instance will increase when N increases. Hence, the area efficiency will be improved when N increases. As a result, our list decoder architecture is more suitable for large block length N.

5.5 Implementation Results

In this chapter, our list decoder architecture has been implemented with list size $L = 2$ and 4 for a rate 1/2 polar code with $N = 1024$. For each list size, two list decoders with the numbers of $T = 8$ and 16 PUs, respectively, are implemented and synthesized under a TSMC 90nm CMOS technology. For the L-MEM within each of our list decoder, each sub memory is compiled with a memory compiler if its depth is large enough. Otherwise, the sub memory is built with registers. For all implemented decoders, each channel LLM is quantized with 4 bits in order to achieve near floating point decoding performance. For our list decoders with $L = 2$ and 4, one stage of pipeline registers is used. Since the synthesis results in [31] were based on a UMC 90nm technology, the authors of [31] have generously re-synthesized their decoder architecture using the TSMC 90nm technology. We list both synthesis results from [31] and the re-synthesized results provided by the authors of [31] in Table 5.5. To make a fair comparison, we focus on the re-synthesized results.

The implementation results in Table 5.5 show that:

- The decoder architecture in [31] has a higher throughput than our list decoder architecture. The reason is that the decoder architecture in [31] employs register based memory while the proposed list decoder architecture employs register file (RF) based memories. The read and write delays of an RF are larger than those of a register based memory.

- On the other hand, our list decoder architecture is more area efficient. For list decoders with the same L and T values, compared with the decoder of [31], our list decoder architecture achieves 1.24 to 1.83 times of area efficiency.

Our list decoder is implemented for the $N = 1024$ polar code because the same block length is used in [31]. For larger block length or larger list size, our advantage in area efficiency is expected to be greater due to more area efficient LLM memory.

Since the CA-SCL algorithm helps to select the correct one from L possible decoded codewords [22], the decoding performance of the CA-SCL algorithm is better than that of the SCL algorithm with the same list size in [31]. As shown in Fig. 5.2, the proposed CA-SCL decoders in Table 5.5 outperform the SCL decoders in Table 5.5. We note that the number of PUs has no impact on the error performance of the SCL and CA-SCL decoders.

As shown in Fig. 5.2, for the CA-SCL algorithm, increasing the list size results in noticeable decoding gain according to our simulations. As shown in [21, Fig. 1], increasing the list size of the SCL algorithm leads to negligible decoding gain especially in high SNR region. For the CA-SCL algorithm, the choice of list size L depends on the tradeoff between error performance and decoding complexity. Better error performance can be achieved by increasing the list size L. For the SCL algorithm, we need to find the threshold value L_T, where little further decoding gain is achieved by employing a list size $L > L_T$. For the SCL algorithm, the feasible list size should be no greater than L_T and satisfy the error performance requirement.

Due to the serial nature of the successive cancelation method, the SC based decoders and its list variants suffer from long decoding latency. In terms of throughput, the throughput of SC based decoders is expected to be lower than BP based decoders, since the BP algorithm for polar codes has a much higher parallelism. On the other hand, the BP algorithm for polar codes still suffers from inferior finite length error performance [26,79]. Current simulation results [26] show that the error

performance of the BP algorithm for polar codes is similar to that of SC algorithm, but worse than those of the SCL and CA-SCL algorithms.

5.6 Conclusion

In this chapter, an efficient list decoder architecture has been proposed for polar codes. The proposed decoder architecture achieves higher area efficiency and better error performance than previous list decoder architectures.

Chapter 6

A High Throughput List Decoder Architecture for Polar Codes

6.1 Introduction

Polar codes [18] are a significant breakthrough in coding theory, since polar codes can achieve the channel capacity of binary-input symmetric memoryless channels [18] and arbitrary discrete memoryless channels [19]. Polar codes of block length N can be efficiently decoded by a successive cancelation (SC) algorithm [18] with a complexity of $O(N \log N)$. While polar codes of very large block length ($N > 2^{20}$ [20]) approach the capacity of underlying channels under the SC algorithm, for short or moderate polar codes, the error performance of the SC algorithm is worse than turbo or LDPC codes [55].

Lots of efforts [23, 24, 55] have already been devoted to the improvement of error performance of polar codes with short or moderate lengths. An SC list (SCL)

decoding algorithm [55] performs better than the SC algorithm. In [23, 24, 55], the cyclic redundancy check (CRC) is used to pick the output codeword from L candidates, where L is the list size. The CRC-aided SCL (CA-SCL) decoding algorithm performs much better than the SCL decoding algorithm at the expense of negligible loss in code rate.

Despite its significantly improved error performance, the hardware implementations of SC based list decoders [31–34] still suffer from long decoding latency and limited throughput due to the serial decoding schedule. In order to reduce the decoding latency of an SC based list decoder, M ($M > 1$) bits are decoded in parallel in [35–37], where the decoding latency can be reduced by M times ideally. However, for the hardware implementations of the algorithms in [35–37], the actually achieved decoding latency reduction is less than M due to extra decoding cycles on finding the L most reliable paths among $2^M L$ candidates, where L is list size. A software adaptive SSC-list-CRC decoder was proposed in [38]. For a (2048, 1723) polar+CRC-32 code, the SSC-list-CRC decoder with $L = 32$ was shown to be about 7 times faster than an SC based list decoder. However, it is unclear whether the list decoder in [38] is suitable for hardware implementation.

In this chapter, a tree based reduced latency list decoding algorithm and its corresponding high throughput hardware architecture are proposed for polar codes. The main contributions are:

- A tree based reduced latency list decoding (RLLD) algorithm over logarithm likelihood ratio (LLR) domain is proposed for polar codes. Inspired by the simplified successive cancelation (SSC) [30] decoding algorithm and the ML-SSC algorithm [54], our RLLD algorithm performs the SC based list decoding

on a binary tree. Previous SCL decoding algorithms visit all the nodes in the tree and consider all possibilities of the information bits, while our RLLD algorithm visits much fewer nodes in the tree and consider fewer possibilities of the information bits. When configured properly, our RLLD algorithm significantly reduces the decoding latency and hence improves throughput, while introducing little performance degradation.

- Based on our RLLD algorithm, a high throughput list decoder architecture is proposed for polar codes. Compared with the state-of-arts SCL decoders in [32,33,36], our list decoder achieves lower decoding latency and higher area efficiency (throughput normalized by area).

More specifically, the major innovations of the proposed decoder architecture are:

- An index based partial sum computation (IPC) algorithm is proposed to avoid copying partial sums directly when one decoding path needs to be copied to another. Compared with the lazy copy algorithm in [55], our IPC algorithm is more hardware friendly since it copies only path indices, while the lazy copy algorithm needs more complex index computation.

- Based on our IPC algorithm, a hybrid partial sum unit (Hyb-PSU) is proposed so that our list decoder is suitable for larger block lengths. The Hyb-PSU is able to store most of the partial sums in area efficient memories such as register file (RF) or SRAM, while the partial sum units (PSUs) in [31–33] store partial sums in registers, which need much larger area when the block length N is larger. Compared with the PSU of [32], our Hyb-PSU achieves an area saving

of 23% and 63% for block length $N = 2^{13}$ and 2^{15}, respectively, under the TSMC 90nm CMOS technology.

- For our RLLD algorithm, when certain types of nodes are visited, each current decoding path splits into multiple ones, among which the L most reliable paths are kept. In this chapter, an efficient path pruning unit (PPU) is proposed to find the L most reliable decoding paths among the split ones. For our high throughput list decoder architecture, the proposed PPU is the key to the implementation of our RLLD algorithm.

- For the fixed-point implementation of our RLLD algorithm, a memory efficient quantization (MEQ) scheme is used to reduce the number of stored bits. Compared with the conventional quantization scheme, our MEQ scheme reduces the number of stored bits by 17%, 25% and 27% for block length $N = 2^{10}$, 2^{13} and 2^{15}, respectively, at the cost of slight error performance degradation.

Note that the SSC and ML-SSC algorithms reduce the latency of the SC algorithm by performing it on a binary tree. Inspired by this idea, our RLLD algorithm performs the SC based list decoding algorithm on a binary tree. The low-latency list decoding algorithm [38] also performs the list decoding algorithm on a binary tree. Our work [80] and the decoding algorithm in [38] are developed independently. Both of our RLLD algorithm and the low-latency list decoding algorithm [38] try to reduce the number of visited nodes in the binary tree so that the decoding latency can be reduced. However, there are still some differences.

- Compared with the decoding algorithm in [38], our RLLD algorithm visits fewer nodes. Illuminated by the ML-SSC algorithm, our RLLD algorithm

processes certain arbitrary rate nodes [30] in a fast way.

- When a rate-1 node [30] is visited, our RLLD algorithm employs a less complex and hardware friendly algorithm to compute the returned constituent codewords.

- Our RLLD algorithm is based on LLR messages, while the decoding algorithm in [38] is based on logarithm likelihood (LL) messages, which require a larger memory to store.

In terms of hardware implementations, compared with state-of-arts SC list decoders [31–34,36,37], our high throughput list decoder architecture shows advantages in various aspects.

- For the high throughput list decoder architecture, LLR message is employed while LL message was used in [31, 32, 36, 37]. The LL based memories require more quantization bits and a larger memory to store. The area efficient memory architecture in [32] is employed to store all LLR messages. LLR messages were also employed in [33,34]. However, the register based memories in [33,34] suffer from excessive area and power consumption when N is large.

- Our list decoder architecture employs a Hyb-PSU, which is scalable for polar codes of large block lengths. The register based PSUs of the list decoders in [31–33] suffer from area overhead when the block length is large. Instead of copying partial sums directly, our scalable PSU copies only decoding path indices, which avoids additional energy consumption.

The proposed high throughput list decoder architecture has been implemented for several block lengths and list sizes under the TSMC 90nm CMOS technology. The

implementation results show that our decoders outperform existing SCL decoders in both decoding latency and area efficiency. For example, compared with the decoders of [33], the area efficiency and decoding latency of our decoders are 1.65 to 45 times and 3.4 to 6.8 times better, respectively.

The rest of the chapter is organized as follows. Related preliminaries are reviewed in Section 6.2. The proposed RLLD algorithm is presented in Section 6.3. The high throughput list decoder architecture is presented in Section 6.4. In Section 6.5, the implementation and comparisons results are shown. At last, the conclusion is drawn in Section 6.6.

6.2 Preliminaries

6.2.1 Polar Codes

Let $u_0^{N-1} = (u_0, u_1, \cdots, u_{N-1})$ denote the data bit sequence and $x_0^{N-1} = (x_0, x_1, \cdots, x_{N-1})$ the corresponding codeword, where $N = 2^n$. Under the polar encoding, $x_0^{N-1} = u_0^{N-1} B_N F^{\otimes n}$, where B_N is the bit reversal permutation matrix, and $F = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$. Here $\otimes n$ denotes the nth Kronecker power and $F^{\otimes n} = F \otimes F^{\otimes(n-1)}$. For $i = 0, 1, \cdots, N-1$, u_i is either an information bit or a frozen bit, which is set to zero usually. For an (N, K) polar code, there are a total of K information bits within u_0^{N-1}. The encoding graph of a polar code with $N = 8$ is shown in Fig. 6.1.

6.2.2 Prior Tree-Based SC Algorithms

A polar code of block length $N = 2^n$ can also be represented by a full binary tree G_n of depth n [30], where each node of the tree is associated with a constituent

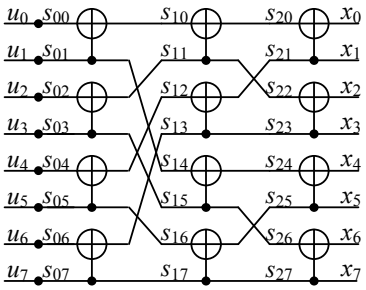

Figure 6.1: Polar encoder with $N = 8$

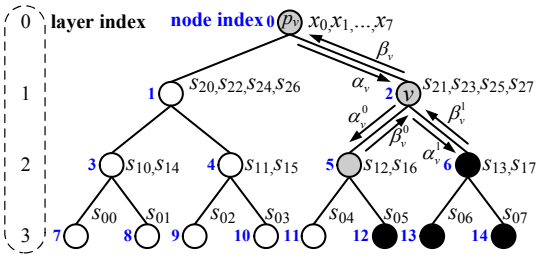

Figure 6.2: Binary tree representation of an (8, 3) polar code

code. For example, for node 1 shown in Fig. 6.2, the correspondent constituent code is the set $\{(s_{20}, s_{22}, s_{24}, s_{26})\}$, where each element $(s_{20}, s_{22}, s_{24}, s_{26})$ relates to the data word u_0^7 as shown in Fig. 6.1. The binary tree representation of an (8, 3) polar code is shown in Fig. 6.2, where the black and white leaf nodes correspond to information and frozen bits, respectively. There are three types of nodes in a binary tree representation of a polar code: rate-0 , rate-1 and arbitrary rate nodes. The leaf nodes of a rate-0 and rate-1 nodes correspond to only frozen and information bits, respectively. The leaf nodes of an arbitrary rate node are associated with both information and frozen bits. The rate-0, rate-1 and arbitrary rate nodes in Fig. 6.2 are represented by circles in white, black and gray as shown.

The SC algorithm can be mapped on G_n, where each node acts as a decoder for its constituent code. The SC algorithm is initialized by feeding the root node with

the channel LLRs, $(Y_0, Y_1, \cdots, Y_{N-1})$, where $Y_i = \log(\Pr(y_i|x_i = 0)/\Pr(y_i|x_i = 1))$ and $(y_0, y_1, \cdots, y_{N-1})$ is the received channel message vector. As shown in Fig. 6.2, the decoder at node v receives a soft information vector α_v and returns a constituent codeword β_v. When a non-leaf node v is activated by receiving an LLR vector α_v, it calculates a soft information vector α_v^0 and sends it to its left child. Node v first waits until it receives a constituent codeword β_v^0, and then computes and sends a soft information vector α_v^1 to its right child. Once the right child returns a constituent codeword β_v^1, node v computes and returns a constituent codeword β_v. When a leaf node v is activated, the returned constituent codeword β_v contains only one bit $\beta_v[0]$, where $\beta_v[0]$ is set to 0 if leaf node v is associated with a frozen bit; otherwise, $\beta_v[0]$ is calculated by making a hard decision on the received LLR $\alpha_v[0]$, where

$$\beta_v[0] = h(\alpha_v[0]) = \begin{cases} 0 & \alpha_v[0] > 0, \\ 0 \text{ or } 1 & \alpha_v[0] = 0, \\ 1 & \alpha_v[0] < 0. \end{cases} \tag{6.1}$$

From the root node, all nodes in a tree are activated in a recursive way for the SC algorithm. Once β_v for the last leaf node is generated, the codeword x_0^{N-1} can be obtained by combining and propagating β_v up to the root node.

The SSC decoding algorithm in [30] simplifies the processing of both rate-0 and rate-1 nodes. Once a rate-0 node is activated, it immediately returns the all zero vector. Once a rate-1 node is activated, a constituent codeword is directly calculated by making hard decisions on the received soft information vector as shown in Eq. (6.1). The ML-SSC decoding algorithm [54] further accelerates the SSC decoding algorithm by performing the exhaustive-search ML decoding on some resource

constrained arbitrary rate nodes, which are called ML nodes in [54]. For an ML node with layer index t, the constituent codeword passed to the parent node p_v is

$$\beta_v = \underset{\mathbf{x}\in\mathcal{C}}{\operatorname{argmax}} \sum_{i=0}^{2^{n-t}-1} (1-2\mathbf{x}[i])\alpha_v[i], \tag{6.2}$$

where $\mathcal{C}$ is the constituent code associated with node v.

6.2.3 LLR Based List Decoding Algorithms

For SCL decoding algorithms [31, 34, 55], when decoding an information bit u_i, each decoding path splits into two paths with $\hat{u}_i$ being 0 and 1, respectively. Thus $2L$ path metrics are computed and the L paths correspond to the L minimum path metrics are kept. The list decoding algorithm in [31, 55] are performed either on probability or logarithmic likelihood (LL) domain. In [34], an LLR based list decoding algorithm was proposed to reduce the message memory requirement and the computational complexity of LL based list decoding algorithm. For decoding path l ($l = 0, 1, \cdots, L-1$), the LLR based list decoding algorithm employs a novel path metric

$$\mathrm{PM}_l^{(i)} = \sum_{k=0}^{i} D(L_n^{(k)}[l], \hat{u}_k[l]), \tag{6.3}$$

where $D(L_n^{(k)}[l], \hat{u}_k[l]) = 0$ if $h(L_n^{(k)}[l])$ equals to $\hat{u}_k[l]$. Otherwise, $D(L_n^{(k)}[l], \hat{u}_k[l]) = |L_n^{(k)}[l]|$. Here $L_n^{(k)}[l] \triangleq \frac{W_n^{(k)}(y_0^{N-1}, \hat{u}_0^{k-1}[l]|0)}{W_n^{(k)}(y_0^{N-1}, \hat{u}_0^{k-1}[l]|1)}$ and $y_0^{N-1} = (y_0, y_1, \cdots, y_{N-1})$ is the received channel message vector.

6.3 Reduced Latency List Decoding Algorithm

6.3.1 SCL Decoding on A Tree

Similar to the SSC decoding algorithm, the SC based list decoding algorithms [31,55] can also be performed on a full binary tree G_n [38, 80]. The SCL decoding is initiated by sending the received channel LLR vector to the root node of G_n. As shown in Fig. 6.3, without losing generality, each internal node v in G_n is activated by receiving L LLR vectors, $\alpha_{v,0}, \alpha_{v,1}, \cdots, \alpha_{v,L-1}$, from its parent node v_p and is responsible for producing L constituent codewords, $\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1}$, where $\alpha_{v,l}$ and $\beta_{v,l}$ correspond to decoding path l for $l = 0, 1, \cdots, L-1$. Suppose the layer index of node v is t, $\alpha_{v,l}$ and $\beta_{v,l}$ have 2^{n-t} LLR messages and binary bits, respectively, for $l = 0, 1, \cdots, L-1$.

Once a non-leaf node v is activated, it calculates L LLR vectors, $\alpha_{v_\mathcal{L},0}$, $\alpha_{v_\mathcal{L},1}$, $\cdots$, $\alpha_{v_\mathcal{L},L-1}$, and passes them to its left child node $v_\mathcal{L}$, where

$$\alpha_{v_\mathcal{L},l}[i] = f(\alpha_{v,l}[2i], \alpha_{v,l}[2i+1]) \tag{6.4}$$

for $0 \leq i < 2^{n-t-1}$ and $l = 0, 1, \cdots, L-1$.

Here $f(a,b) = 2\tanh^{-1}(\tanh(a/2)\tanh(b/2))$ and can be approximated as:

$$f(a,b) \approx \text{sign}(a) \cdot \text{sign}(b) \min(|a|, |b|). \tag{6.5}$$

Node v then waits until it receives L codewords, $\beta_{v_\mathcal{L},0}, \beta_{v_\mathcal{L},1}, \cdots, \beta_{v_\mathcal{L},L-1}$, from $v_\mathcal{L}$. In the following step, node v calculates another L LLR vectors, $\alpha_{v_\mathcal{R},0}$, $\alpha_{v_\mathcal{R},1}$,

$\cdots$, $\alpha_{v_{\mathcal{R}},L-1}$, and passes them to its right child node $v_{\mathcal{R}}$, where

$$\begin{aligned} \alpha_{v_{\mathcal{R}},l}[i] &= g(\alpha_{v,l}[2i], \alpha_{v,l}[2i+1], \beta_{v_{\mathcal{L}},l}[i]) \\ &= \alpha_{v,l}[2i](1 - 2\beta_{v_{\mathcal{L}},l}[i]) + \alpha_{v,l}[2i+1] \end{aligned} \tag{6.6}$$

for $0 \leq i < 2^{n-t-1}$ and $l = 0, 1, \cdots, L-1$.

At last, node v waits until it receives L codewords, $\beta_{v_{\mathcal{R}},0}, \beta_{v_{\mathcal{R}},1}, \cdots, \beta_{v_{\mathcal{R}},L-1}$, from $v_{\mathcal{R}}$. It then calculates $\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1}$ and passes them to its parent node v_p, where

$$(\beta_{v,l}[2i], \beta_{v,l}[2i+1]) = (\beta_{v_{\mathcal{L}},l}[i] \oplus \beta_{v_{\mathcal{R}},l}[i], \beta_{v_{\mathcal{R}},l}[i]), \tag{6.7}$$

for $0 \leq i < 2^{n-t-1}$ and $l = 0, 1, \cdots, L-1$.

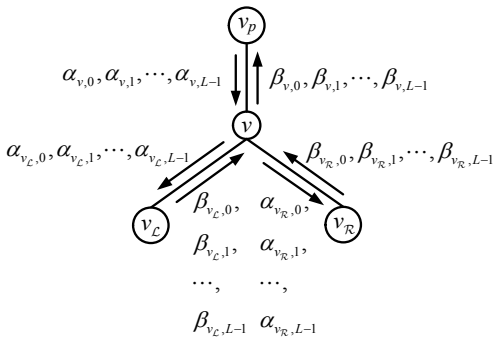

Figure 6.3: Node activation schedule for SC based list decoding on G_n

For $l = 0, 1, \cdots, L-1$, PM_l is the path metric associated with decoding path l and is initialized with 0. When a leaf node v associated with an information bit is activated, decoding path l splits into two paths with $\beta_{v,l}$ being 0 and 1, respectively. Note that the layer index of a leaf node is n, hence $\alpha_{v,l}$ and $\beta_{v,l}$ have only one LLR and binary bit, respectively, when node v is a leaf node. For the SCL decoding, $2L$

expanded path metrics are computed, where

$$\mathrm{PM}_l^j = \mathrm{PM}_l + D(\alpha_{v,l}, j), \tag{6.8}$$

for $j = 0, 1$ and $l = 0, 1, \cdots, L-1$. $D(\alpha_{v,l}, j) = 0$ if $h(\alpha_{v,l})$ equals j. Otherwise, $D(\alpha_{v,l}, j) = |\alpha_{v,l}|$. Suppose the L minimum expanded path metrics are $\mathrm{PM}_{a_0}^{j_0}$, $\mathrm{PM}_{a_1}^{j_1}$, $\cdots$, $\mathrm{PM}_{a_{L-1}}^{j_{L-1}}$, which correspond to the L most reliable paths, then $\beta_{v,l} = j_l$ for $l = 0, 1, \cdots, L-1$. Decoding path a_l will be copied to decoding path l before further partial sum and LLR vector computations. For each decoding path l, path metric is also updated with $\mathrm{PM}_l = \mathrm{PM}_{a_l}^{j_l}$. When a leaf node v associated with a frozen bit is activated, $\beta_{v,l} = 0$ for $l = 0, 1, \cdots, L-1$ are passed to its parent node v_p. The updated path metric $\mathrm{PM}_l = \mathrm{PM}_l + D(\alpha_{v,l}, 0)$. Note that the SCL algorithm on a tree described above is equivalent to the SCL algorithms in [31, 55].

6.3.2 Proposed RLLD algorithm

In this chapter, a reduced latency list decoding (RLLD) algorithm is proposed to reduce the decoding latency of SC list decoding for polar codes. For a node v, let I_v denote the total number of leaf nodes that are associated with information bits. Let X_{th} be a predefined threshold value and X_0 and X_1 be predefined parameters. Our RLLD algorithm performs the SC based list decoding on G_n and follows the node activation schedule in Section 6.3.1, except when certain type of nodes are activated. These nodes calculate and return the codewords to their parent nodes while updating the decoding paths and their metrics, without activating their child nodes. Specifically:

- When a rate-0 node v is activated, $\beta_{v,l}$ is a zero vector for $l = 0, 1, \cdots, L-1$.
- When a rate-1 node v with $I_v > X_{th}$ is activated, $\beta_{v,l}$ is just the hard decision of $\alpha_{v,l}$ for $l = 0, 1, \cdots, L-1$. For a well constructed polar code, we observe that the polarized channel capacities of the information bits corresponding to rate-1 nodes with $I_v > X_{th}$ are greater than those of the other information bits. Hence, for rate-1 nodes with $I_v > X_{th}$, our RLLD algorithm considers only the most reliable candidate codeword for each decoding path due to a more reliable channel.
- When a rate-1 node v with $I_v \leqslant X_{th}$ is activated, the returned codewords are calculated by the proposed candidate generation (CG) algorithm.
- Let t denote the layer index of node v. When an arbitrary rate node v with $I_v \leqslant X_0$ and $2^{n-t} \leqslant X_1$ is activated, each decoding path splits into 2^{I_v} paths. From now on, such an arbitrary rate node is called fast processing (FP) node. The proposed metric based search (MBS) algorithm is used to calculate the returned codewords.

When performed on a binary tree, the SCL algorithms in [31, 55] do the path expanding and pruning as well as the updating of path metrics when a leaf node is activated. In contrast, our RLLD algorithm do the path expanding and pruning as well as updating of path metrics when a certain intermediate node is visited. Thus, our RLLD algorithm visits fewer nodes.

When a rate-1 node with $I_v > X_{th}$ or a rate-0 node is activated, ideally, PM_l is updated with $\mathrm{PM}_l + \Delta_{v,l}$ for $l = 0, 1, \cdots, L-1$, where $\Delta_{v,l} = \sum_{i=0}^{I_v-1} D(\alpha_{v,l}[i], \beta_{v,l}[i])$. For each rate-1 node with $I_v > X_{th}$, $\Delta_{v,l} = 0$ since $\beta_{v,l}$ is the hard decision of

$\alpha_{v,l}$. However, for a rate-0 node, $\Delta_{v,l}$ could have a non-zero value. For our RLLD algorithm, $\Delta_{v,l}$ is also set to be 0 for each rate-0 node, since the resulting performance degradation is negligible. By setting $\Delta_{v,l}$ to be 0, we no longer need to calculate $\alpha_{v,l}$ sent to a rate-0 node.

Proposed CG Algorithm

When a rate-1 node with $I_v \leqslant X_{th}$ is activated, ideally, we should consider 2^{I_v} candidate codewords for each decoding path. Since there are at most L codewords from the same decoding path that could be passed to the parent node, it is enough to find only the L most reliable codewords among 2^{I_v} candidates for each decoding path. When I_v is large (e.g. $I_v \geqslant 32$), finding the L most reliable codewords is computationally intensive and lacks efficient hardware implementations. For our RLLD algorithm, we considers only the $W (W < L)$ most reliable codewords among 2^{I_v} candidates for each decoding paths. In this chapter, W is set to be 2, since it results in efficient hardware implementations at the cost of negligible error performance lost.

When $W = 2$, the proposed CG algorithm, shown in Alg. 21, is used to calculate the codewords passed to the parent node. Besides, the CG algorithm also outputs L list indices, $a_0, a_1, \cdots, a_{L-1}$, which indicate that decoding path a_l needs to be copied to path l. Suppose the layer index of such a rate-1 node v is t. For each decoding path l, there are $2^{I_v} = 2^{2^{n-t}}$ candidate codewords that could be passed to the parent node v_p. However, our CG algorithm considers only the most reliable codeword $\mathcal{C}_{v,l,0}$ and the second most reliable codeword $\mathcal{C}_{v,l,1}$. In order to find these

two codewords, each candidate codeword $\mathcal{C}_{v,l,j}$ is associated with a node metric

$$\mathrm{NM}_l^j = \sum_{k=0}^{I_v-1} m_k |\alpha_{v,l}[k]| \tag{6.9}$$

for $j = 0, 1, \cdots, 2^{I_v}-1$, where $m_k = 0$ if $\mathcal{C}_{v,l,j}[k]$ equals $h(\alpha_{v,l}[k])$ and 1 otherwise. As a result, the smaller a node metric is, the more reliable the corresponding candidate codeword is. Based on Eq. (6.9), $\mathcal{C}_{v,l,0} = h(\alpha_{v,l})$ is the hard decision of the received LLR vector $\alpha_{v,l}$. $\mathcal{C}_{v,l,1}$ is obtained by flipping the $k_{M,l}$-th bit of $\mathcal{C}_{v,l,0}$, where $k_{M,l}$ is the index of the LLR element with the smallest absolute value among $\alpha_{v,l}$.

Each decoding path splits into two paths and has two associated candidate codewords. Alg. 21 calculates $2L$ expanded path metrics PM_l^j for $l = 0, 1, \cdots, L-1$ and $j = 0, 1$ to select L codewords passed to the parent node. The $\min_L$ function in Alg. 21 finds the L smallest values among $2L$ input expanded path metrics. Once $\beta_{v,l}$ for $l = 0, 1, \cdots, L-1$ are computed, decoding path a_l is copied to decoding path l before further operations.

Algorithm 21: The proposed CG algorithm

input : $\alpha_{v,0}, \alpha_{v,1}, \cdots, \alpha_{v,L-1}$
output: $\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1}$; $a_0, a_1, \cdots, a_{L-1}$
for $l = 0$ **to** $L-1$ **do**
  $k_{M,l} = \underset{k \in \{0,1,\cdots,I_v-1\}}{\operatorname{argmin}} |\alpha_{v,l}[k]|$
  $\mathrm{NM}_l^0 = 0$; $\mathcal{C}_{v,l,0} = h(\alpha_{v,l})$
  $\mathrm{NM}_l^1 = |\alpha_{v,l}[k_{M,l}]|$; $\mathcal{C}_{v,l,1} = \mathrm{Flip}(\mathcal{C}_{v,l,0}, k_{M,l})$
  $\mathrm{PM}_l^j = \mathrm{PM}_l + \mathrm{NM}_l^j$ for $j = 0, 1$
$(\mathrm{PM}_{a_0}^{b_0}, \cdots, \mathrm{PM}_{a_{L-1}}^{b_{L-1}}) = \min_L(\mathrm{PM}_0^0, \mathrm{PM}_0^1, \cdots, \mathrm{PM}_{L-1}^1)$ **for** $l = 0$ **to** $L-1$ **do**
  $\beta_{v,l} = \mathcal{C}_{v,a_l,b_l}$; $\mathrm{PM}_l = \mathrm{PM}_{a_l}^{b_l}$

Proposed MBS Algorithm

When an FP node is activated, each current decoding path expands to 2^{I_v} paths, each of which is associated with a candidate codeword. Similar to the CG algorithm, the proposed MBS algorithm calculates L codewords passed to the parent node and L path indices, $a_0, a_1, \cdots, a_{L-1}$. The calculation of returned codewords are shown as follows.

- For each candidate codeword $\mathcal{C}_{v,l}^{j}$, calculate its corresponding node metric NM_l^j for $j = 0, 1, \cdots, 2^{I_v} - 1$ and $l = 0, 1, \cdots, L - 1$.
- Calculate $2^{I_v}L$ expanded path metrics PM_l^j for $l = 0, 1, \cdots, L - 1$ and $j = 0, 1, \cdots, 2^{I_v} - 1$.
- Find L expanded path metrics among $2^{I_v}L$ ones. The correspondent candidate codewords are passed to the parent node v_p.

To calculate the node metric, we propose a new method with low computational complexity. In the literature, two methods can be used: the direct-mapping method (DMM) shown in Eq. (6.9) and the recursive channel combination (RCC) [37]. In terms of computational complexity, the former needs $2^{I_v}(2^{n-t}-1)L$ additions, where $N = 2^n$ and t is the layer index of an FP node v. The RCC needs $(\sum_{i=1}^{n-t-1} 2^i 2^{2^{n-t-i}} + 2^{I_v})L$ additions. Compared to the DMM, the RCC approach needs fewer additions. For our RLLD algorithm, we want to compute these 2^{I_v} node metrics in parallel. However, the parallel hardware implementations of the DMM and RCC algorithms require large area consumption. This will be discussed in more detail in Section 6.4.3.

In this chapter, a hardware efficient node metric computation method, which takes advantage of both the DMM and the RCC, is proposed. The proposed method,

referred to as the DR-Hybrid (DRH) method, is shown in Alg. 22, where $\mathcal{C}_{v,l}^{j}[2i : 2i+1] = (\mathcal{C}_{v,l}^{j}[2i], \mathcal{C}_{v,l}^{j}[2i+1])$, and r is represented by a binary tuple of length two, i.e. $r = r_0 + 2r_1$. In our method, the RCC approach is used to calculate $\theta_{l,i}$ first. Then, the DMM is carried out.

Algorithm 22: DR-Hybrid method

```
   for l = 0 to L − 1 do
   |  /* ----------RCC----------                                   */
1  |  for i = 0 to 2^{n−t−1} − 1 do
2  |  |  for r = 0 to 3 do
3  |  |  |  θ_{l,i}[(r_0, r_1)] = (1 − 2r_0)α_{v,l}[2i] + (1 − 2r_1)α_{v,l}[2i + 1];
   |  /* ----------DMM----------                                   */
4  |  for j = 0 to 2^{I_v} − 1 do
5  |  |  NM_l^j = Σ_{i=0}^{2^{n−t−1}−1} θ_{l,i}[C_{v,l}^j[2i : 2i + 1]].
```

The DRH method needs $4 \times 2^{n-t-1} + 2^{I_v}(2^{n-t-1} - 1)$ additions. Take $X_0 = 8$ and $X_1 = 16$ as an example, the DMM, RCC and DRH methods need 3840, 864 and 1824 additions. Though our DRH method needs more additions than the RCC, it results in a more area efficient hardware implementation when all 2^{I_v} node metrics are computed in parallel, since the RCC method needs more complex multiplexors.

Once we have $2^{I_v}L$ node metrics and corresponding candidate codewords, $2^{I_v}L$ expanded path metrics $\mathrm{PM}_l^j = \mathrm{PM}_l + \mathrm{NM}_l^j$ for $l = 0, 1, \cdots, L-1$ and $j = 0, 1, \cdots, 2^{I_v} - 1$ can be computed. The next step is selecting L returned codewords and their corresponding expanded path metrics.

Ideally, we should find the L minimum expanded path metrics, which correspond to the L most reliable codewords, among $2^{I_v}L$ ones. However, directly finding the L minimum values from $2^{I_v}L$ ones is computationally intensive and lacks efficient

hardware implementations. A bitonic sequence based sorter [32] (BBS) with $2^{I_v}L$ inputs is able to fulfill this task. Such a BBS takes $2^{I_v-1}L(\sum_{i=1}^{s-1} i)+2^{I_v-2}L$ compare-and-switch (CS) units [32], where each of them has one comparator and two 2-to-1 multiplexors and $s = \log_2(2^{I_v}L)$. For example, when $I_v = 8$ and $L = 4$, such a BBS needs 23296 CS units. In order to simplify the hardware implementation, a two-stage sorting scheme was proposed in [37], where the first stage selects q ($q < L$) smallest node metrics from 2^{I_v} ones for each decoding path. The second stage selects the L smallest metrics from the Lq expanded path metrics produced by the first stage. Compared with the direct sorting scheme [32, 36], the hardware implementation of the two-stage sorting scheme is more efficient at the cost of certain error performance degradation.

In this chapter, our MBS algorithm employs the two-stage sorting scheme and improves the first stage in the following two aspects:

- Instead of using a fixed q, our MBS algorithm employs a dynamic $q_{I_v,L}(q_{I_v,L} \leqslant L)$, which is a power of 2 and depends on both I_v and L.
- An approximated sorting (ASort) method is used to select out $q_{I_v,L}$ metrics from 2^{I_v} ones. Though these sorted metrics are not always the precisely $q_{I_v,L}$ smallest among 2^{I_v} one, our ASort method leads to an efficient hardware implementation.

Our ASort method is illustrated as follows:

- When $2^{I_v} \leqslant 2L$, the BBS with $2L$ inputs and L outputs is used to select the $q_{I_v,L}$ minimum node metrics from 2^{I_v} ones.
- When $2^{I_v} > 2L$, all 2^{I_v} node metrics are divided into $q_{I_v,L}$ groups as follows:

$$\underbrace{\mathrm{NM}_l^0, \cdots, \mathrm{NM}_l^{m-1}}_{\text{group 1}}, \cdots, \underbrace{\mathrm{NM}_l^{(q_{I_v,L}-1)m}, \cdots, \mathrm{NM}_l^{q_{I_v,L}m-1}}_{\text{group } q_{I_v,L}}.$$

Here $m = \frac{2^{I_v}}{q_{I_v,L}}$. The two minimum node metrics of each group are first computed. The BBS computes the minimum $q_{I_v,L}$ node metrics among $2q_{I_v,L}$ ones.

After the first stage of sorting, the number of expanded path metrics N_e could be $2L, 4L, \cdots, L \times L$. The second stage of sorting is the same as that in [37]. A binary tree of $2L$-L BBSs are employed to sort out the final L minimum expanded path metrics. Take $N_e = 4L$ as an example, there are $4L$ extended path metrics: $\mathrm{PM}_{l_0}^{j_0}$, $\mathrm{PM}_{l_1}^{j_1}$, $\cdots$, $\mathrm{PM}_{l_{4L-1}}^{j_{4L-1}}$, then $\mathrm{PM}_{l_0}^{j_0}$, $\cdots$, $\mathrm{PM}_{l_{2L-1}}^{j_{2L-1}}$ and $\mathrm{PM}_{l_{2L}}^{j_{2L}}$, $\cdots$, $\mathrm{PM}_{l_{4L-1}}^{j_{4L-1}}$ are applied to two $2L$-L BBSs, respectively. Thus, total $2L$ metrics are selected out. Then the $2L$-L BBS is employed again to generated the final L minimum extended path metrics: $\mathrm{PM}_{l'_0}^{j'_0}$, $\mathrm{PM}_{l'_1}^{j'_1}$, $\cdots$, $\mathrm{PM}_{l'_{L-1}}^{j'_{L-1}}$.

6.3.3 Discussions on the Parameters of Our RLLD Algorithm

For our RLLD algorithm, the returned codewords from rate-1 nodes with $I_v > X_{th}$ are obtained by making hard decisions on the received LLR vectors. The other rate-1 nodes are processed by our CG algorithm. Note that both the hard decision approach and our CG algorithm could cause potential error performance degradation since ideally we should consider 2^{I_v} candidate codewords for each decoding path. With more rate-1 nodes (decreasing X_{th}) being processed by the hard decision approach, the decoding latency could be reduced at the cost of more error performance degradation.

The choices of X_0 and X_1 are tradeoffs between implementation complexity and

achieved decoding latency reduction. Ideally, we want X_0 and X_1 to be as large as possible so that more data bits could be decoded in parallel. The number of adders needed by Alg. 22 is proportional to $2^{I_v}2^{n-t}$ in terms of hardware implementations. Thus, for practical implementations, we could choose only realistic values for X_0 and X_1.

For the two step sorting scheme of our MBS algorithm, we want $q_{I_v,L}$ to be as small as possible so that the sorting complexity could be minimized. However, reducing $q_{I_v,L}$ could degenerate the resulting error performance, since ideally we need to consider the L most reliable candidate codewords for each decoding path. As a result, the selections of $q_{I_v,L}$ are tradeoffs between sorting complexity and error performance.

6.3.4 Comparison with Related Algorithms

If we perform the SC based list decoding algorithms [31, 55] on a tree, then all $2N-1$ nodes of the tree will be activated. For our RLLD algorithm, denote n_a as the number of activated nodes. Then we have $n_a < 2N-1$, where n_a is determined by the block length N, the code rate, the locations of frozen bits and the parameters X_0 and X_1. X_0 and X_1 are used to identify all FP nodes. The reduction of the number of activated nodes will transfer into reduced decoding latency and increased throughput. Take the (8, 3) polar code in Fig. 6.2 as an example, suppose $X_0 = 1$ and $X_1 = 2$, then only 5 nodes (node 0, 1, 2, 5, 6) need to be activated by our RLLD algorithm, whereas the previous algorithms [31, 55] need to activate all 15 nodes.

The CA-SCL decoding algorithm was also performed on a binary tree in [38]. Compared with the low-latency list decoding algorithm [38], our RLLD algorithm

employs the proposed MBS algorithm to process FP nodes, while FP nodes were processed by activating its child nodes in [38]. Our MBS algorithm results in decreased decoding latency at the cost of potential error performance loss. Besides, our RLLD algorithm takes a simpler approach when a rate-1 node is activated. When a rate-1 node is activated, a Chase-like algorithm was used to calculate the L codewords passed to the parent node in [38]. Compared to the Chase-like algorithm, our CG algorithm has lower computational complexity and is more suitable for hardware implementation due to the following facts:

(1) The Chase-like algorithm in [38] was performed over log-likelihoods (LL) domain while our method is performed over LLR domain. Compared with our LLR based method, it takes more additions to calculate related metrics for the Chase-like algorithm.

(2) For each decoding path, the Chase-like algorithm considers $1 + \binom{c}{1} + \binom{c}{2}$ candidate constituent codewords, where $c = 2$ in [38]. In contrast, our method considers only two constituent codewords.

(3) In order to find the L best decoding paths and their constituent codewords, the Chase-like algorithm creates a candidate path list. The final L candidates are determined by inserting and removing elements from the list. The Chase-like algorithm is suitable for software implementations. However, the hardware implementations of the Chase-like algorithm has not been discussed in [38]. On the other hand, with a bitonic based sorter [32] (BBS), the L most reliable decoding paths can be decided in parallel for our CG algorithm.

6.3.5 Simulation Results

For an (8192, 4096) polar code, the bit error rate (BER) performances of the proposed RLLD algorithm as well as other algorithms are shown in Fig. 6.4. In Fig. 6.4, CSx denotes the CA-SCL decoding algorithm with $L = x$, where CRC-32 is used. Rx-y denotes our RLLD algorithm with $L = x$ and $X_{th} = y$. The values of $q_{I_v,L}$'s under different list sizes and I_v's are shown in Table 6.1. For all simulated algorithms, the additive white Gaussian noise (AWGN) channel and binary phase-shift keying (BPSK) modulation are used. For all simulated RLLD algorithms, $X_0 = 8$ and $X_1 = 16$ for practical implementations.

Table 6.1: The Values of $q_{I_v,L}$'s under Different List Sizes and I_v's

	I_v	1	2	3	4	5	6	7	8
	2	2	2	2	2	2	2	2	2
	4	2	4	4	4	4	4	4	2
L	8	2	4	8	8	8	8	4	2
	16	2	4	8	8	8	8	8	2
	32	2	4	8	8	8	4	4	2

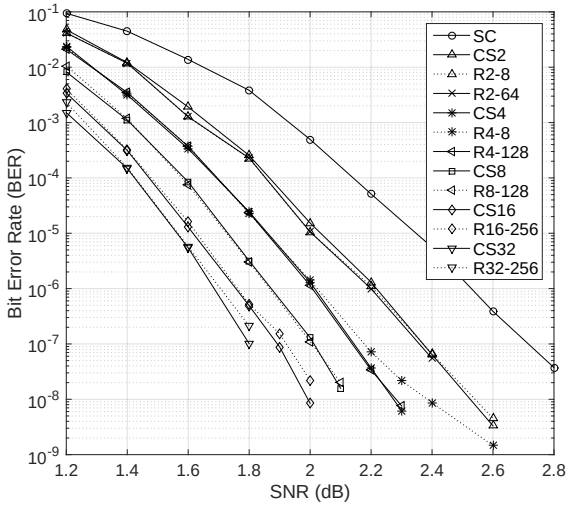

Figure 6.4: BER performance for an (8192, 4096) polar code

Based on the simulation results shown in Fig. 6.4, we observe that R2-8 performs nearly the same as CS2 and R2-64. When the list size increases, compared with CS4, R4-8 shows obvious error performance degradation when BER is blow 10^{-7}. The degradation is reduced by increasing X_{th} to 128, as we observe that R4-128 performs nearly the same as CS4. When the list size further increases (e.g. $L = 16$ and 32), at low BER level, the error performance degradation shows again even $X_{th} = 256$. As shown in Fig. 6.4, R16-256 and R32-256 are worse than CS16 and CS32 when BER is below 10^{-5} and 10^{-6}, respectively. Note that for the (8192, 4096) polar code in this chapter, I_v of a rate-1 node is at most 256.

Depending on the specific list size, we predict that our RLLD algorithm will show performance degradation compared to the CA-SCL algorithm at certain BER values even when all rate-1 nodes are processed by the proposed CG algorithm. Nevertheless, for the (8192, 4096) polar code, our RLLD algorithm can still show obvious advantage in terms of error performance compared with the SC algorithm. The root causes of the error performance degradation of our RLLD algorithm may be as follows:

(1) For our RLLD algorithm, when a rate-1 node with $I_v \leqslant X_{th}$ is activated, only the two most reliable constituent codewords are kept. When list size L is large, there may not be enough candidate codewords to include the correct codeword, since our CG algorithm could miss certain good candidate codewords.

(2) When a rate-1 node with $I_v > X_{th}$ is activated, only the most reliable candidate codeword is considered for each decoding path, which could also cause error performance degradation.

(3) During the first sorting stage of our MBS algorithm, when $2^{I_v} \geqslant L$, $q_{I_v,L}$ is

selected to be no greater than L for certain I_v values for efficient hardware implementation. As a result, we may lose certain good candidate codewords due to the limitation on $q_{I_v,L}$.

6.4 High Throughput List Polar Decoder Architecture

6.4.1 Top Decoder Architecture

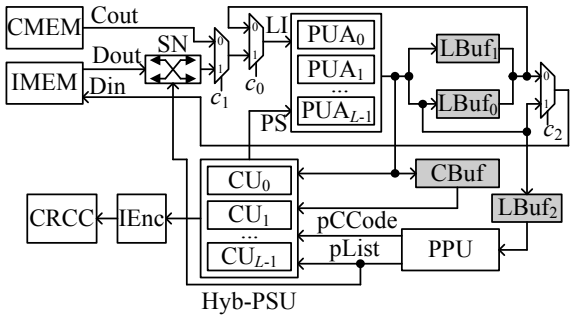

Figure 6.5: Decoder top architecture

In this chapter, based on the proposed RLLD algorithm, a high throughput list decoder architecture, shown in Fig. 6.5, for polar codes is proposed. In Fig. 6.5, the channel message memory (CMEM) stores the received channel LLRs, and the internal LLR message memory (IMEM) stores the LLRs generated during the SC computation process. With the concatenation and split method in our prior work [32], the IMEM is implemented with area efficient memories, such as register file (RF) or SRAM. The proposed architecture has L groups of processing unit arrays (PUAs), each of which contains T processing units [20] (PUs) and is capable of performing either the f or the g computation. The hybrid partial sum unit (Hyb-PSU) in Fig. 6.5

consists of L computation units, CU_0, CU_1, $\cdots$, CU_{L-1}, which are responsible for updating the partial sums of L decoding paths, respectively. The path pruning unit (PPU) in Fig. 6.5 finds the list indices and corresponding constituent codewords for L survival decoding paths, respectively.

Both our high throughput list decoder architecture in Fig. 6.5 and that in [32] employ a partial parallel processing method. Besides, both architectures contain a channel message memory and internal message memory. However, compared to the architecture in [32], the major improvements of our list decoder architecture are:

(a) Instead of LL messages, our high throughput list decoder architecture employs LLR messages, which result in more area efficient internal and channel message memories.

(b) The PPU in Fig. 6.5 implements our CG and MBS algorithms, while the PPU in [32] is just a sorter which selects L values among $2L$ ones. Due to the proposed PPU, our decoder architecture achieves much higher throughput that the that in [32].

(c) Our list decoder architecture employs a novel Hyb-PSU, which is more area and energy efficient than that in [32]. Our Hyb-PSU is based on the proposed index based partial sum computation algorithm. When a decoding path needs to be copied to another one, our Hyb-PSU avoids copying partial sums directly by copying only decoding path indices. In contrast, the PSU in [32] copies path sums directly, which incurs additional energy consumption. Our Hyb-PSU stores most of the partial sums in area efficient memories, while the PSU in [32] stores all the partial sums in area demanding registers. Hence, our Hyb-PSU is scalable for larger block lengths.

6.4.2 Memory Efficient Quantization Scheme

For an SC or SCL decoder, the message memory occupies a large part of the overall decoder area [20, 32]. An SCL decoder needs a channel message memory and an internal message memory. For an LLR based SCL decoder, the channel memory stores N channel LLR messages. The internal message memory stores Ln LLR matrices: $P_{l,t}$ for $l = 0, 1, \cdots, L-1$ and $t = 1, 2, \cdots, n$, where $P_{l,t}$ has 2^{n-t} LLR messages.

For a fixed point implementation of our RLLD algorithm, it is straightforward to quantize all LLRs in the internal memory with Q bits. In this chapter, a memory efficient quantization (MEQ) scheme is proposed to reduce the size of the internal memory. $f(a, b)$ in Eq. (6.5) has the same magnitude range as those of a and b, while the magnitude range of $g(a, b, s)$ in Eq. (6.6) is at most twice of those of a and b (s is either 0 or 1). Since $P_{0,t}, P_{1,t}, \cdots, P_{L-1,t}$ are computed based on $P_{0,t-1}, P_{1,t-1}, \cdots, P_{L-1,t-1}$, for a decoding path l, the LLRs in P_{l,t_1} may need a greater magnitude range than that of the LLRs in P_{l,t_2}, where $t_1 > t_2$. Suppose each channel LLR is quantized with Q_c bits, the proposed MEQ scheme is as follows:

(1) Suppose all LLRs within the internal memory are quantized with Q_m bits, determine the minimal Q_m such that the error performance degradation of the fixed point performance is negligible.

(2) Let $t_1, t_2, \cdots, t_r$ be r integers, where $t_1 \leqslant t_2 \leqslant \cdots \leqslant t_r \leqslant n$ and $r = Q_m - Q_c$. Denote $\mathbf{P}_t = (P_{0,t}, P_{1,t} \cdots, P_{L-1,t})$. Suppose LLRs associated with $\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_{t_1}$ are quantized with Q_c bits and the remaining LLRs are quantized with Q_m bits. Decide the maximal t_1 such that the resulting fixed point error performance degradation is negligible. Once t_1 is decided, suppose the LLRs within $\mathbf{P}_{t_1+1}, \mathbf{P}_{t_1+2}, \cdots, \mathbf{P}_{t_2}$ are

all quantized with $Q_c + 1$ bits, find the maximal t_2 such that the corresponding error performance degradation is negligible. In this way, $t_3, \cdots, t_r$ are decided in a serial manner so that $\mathbf{P}_{t_i+1}, \mathbf{P}_{t_i+2}, \cdots, \mathbf{P}_{t_{i+1}}$ are quantized with $Q_c + i$ bits for $1 \leqslant i \leqslant r-1$, and $\mathbf{P}_j$ are quantized to Q_m bits for $j > t_r$.

With the proposed MEQ scheme, the number of bits saved for the internal memory is

$$N_B = \sum_{j=1}^{r+1} \sum_{t=t_{j-1}+1}^{t_j} L2^{n-t}(Q_c + j - 1), \tag{6.10}$$

where $t_0 = 0$ and $t_{r+1} = n$ are introduced for convenience.

In order to show the effectiveness of our MEQ scheme, the error performances of our RLLD algorithm with the proposed MEQ scheme are shown in Fig. 6.6, where the RLLD algorithm with our MEQ scheme is compared with the floating-point CA-SCL decoding algorithm, floating-point RLLD algorithm, and RLLD algorithm with a uniform quantization scheme for three different polar codes, (1024, 512), (8192, 4096) and (32768, 29504) with $X_{th} = 32, 128, 1024$, respectively. For all fixed-point decoders, each channel LLR is quantized with $Q_c = 5$ bits. For the RLLD algorithm with uniform quantization, each LLR in the internal memory is quantized with $Q_m = 7$ bits. Since $Q_m - Q_c = 2$, we need to determine two integers, r_1 and r_2, for our MEQ scheme. When $N = 2^{10}$, 2^{13} and 2^{15}, $(r_1, r_2) = (1,2)$, $(3,4)$ and $(4,5)$, respectively. As shown in Fig. 6.6, the performance degradation caused by our MEQ scheme is small. Compared with the uniform quantization, the proposed MEQ scheme reduces the number of stored bits by 17%, 25% and 27% for $N = 2^{10}$, 2^{13} and 2^{15}, respectively.

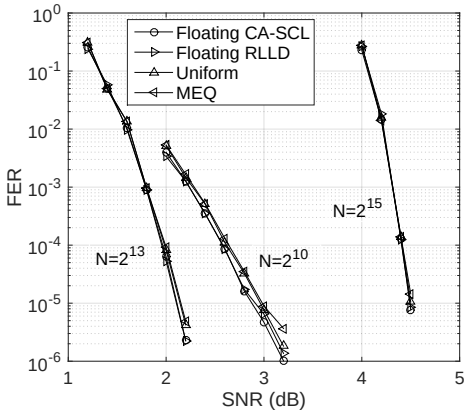

Figure 6.6: Effects of the proposed MEQ scheme on the error performances

6.4.3 Proposed path pruning unit

When a rate-1 node with $I_v \leqslant W_T$ or an FP node is activated, each decoding path splits into multiple ones and only the L most reliable paths are kept. The PPU in Fig. 6.5 implements our CG and MBS algorithms, and is responsible for calculating L returned codewords, $\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1}$, and L path indices, $a_0, a_1, \cdots, a_{L-1}$. For $l = 0, 1, \cdots, L-1$, decoding path l copies from decoding path a_l before further decoding steps.

Take $L = 4$ as an example, the proposed PPU is shown in Fig. 6.7, which can be easily adapted to other L values. Our PPU in Fig. 6.7 has two types of node metric generation (NG) units, NG-I and NG-II, which compute the node metrics for a rate-1 node and an FP node, respectively. NG-I$_l$ and NG-II$_l$ correspond to decoding path l. For decoding path l, the expanded path metrics PM$_l^j$'s are obtained by adding the node metrics to the path metric PM$_l$, which is stored in the path metric registers (PMR) and initialized with 0.

When a rate-1 node is activated, NG-I$_l$ outputs two node metrics for l =

$0, 1, \cdots, L-1$. After $2L$ expanded path metrics are computed, a stage of metric sorter (MS_{2L-L}) selects the L minimum metrics and their corresponding codewords from $2L$ ones. The metrics sorter MS_{2L-L} implements the $\min_L$ function in Alg. 21 and can be constructed with a BBS. When an FP node is activated, L NG-II modules implement the first part of our two-stage sorting scheme. For each decoding path, $q_{I_v,L}$ node metrics and their correspondent codewords are computed. The tree of metric sorters sort the L minimum metrics among $q_{I_v,L}L$ ones. This is achieved by $\log_2 q_{I_v,L}$ stages of metric sorters when $q_{I_v,L}$ is a power of 2. The output expanded path metrics of the last stage of metric sorter are saved in the PMR. The corresponding codewords of the selected L expanded path metrics are also chosen. The related circuitry is omitted for simplicity.

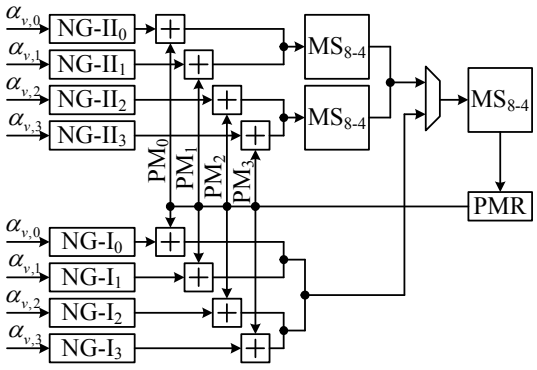

Figure 6.7: The proposed architecture for PPU

The micro architecture of NG-I$_l$ is shown in Fig. 6.8. The most complex part of NG-I$_l$ is finding the minimum LLR magnitude and its corresponding index among the LLR vector $|\alpha_{v,l}| \triangleq (|\alpha_{v,l}[0]|, |\alpha_{v,l}[1]|, \cdots, |\alpha_{v,l}[I_v - 1]|)$. Since the node metric of the most reliable candidate codeword is always 0, we need to compute $\mathrm{NM}_l^1 = |\alpha_{v,l}[k_{M,l}]|$ in Fig. 6.8, which is the node metric of the second most reliable candidate

codeword, with a corresponding index $k_{M,l}$. For our list decoder architecture, for each decoding path, at most T LLRs are computed in one clock cycle, since we have only T PUs per decoding path. The Min-1 unit in Fig. 6.8 is capable of finding the minimum value, mLLR, and its corresponding index, mIdx, from at most T parallel inputs. When $I_v \leqslant T$, $\mathrm{NM}_l^1 = \mathrm{mLLR}$ and $k_{M,l} = \mathrm{mIdx}$. $\mathcal{C}_{v,l,0} = h(\alpha_{v,l})$ in Fig. 6.8 is the hard decision of $\alpha_{v,l}$, which is the most reliable candidate codeword. The second most reliable candidate codeword is obtained by flipping the $k_{M,l}$-th bit of $\mathcal{C}_{v,l,0}$.

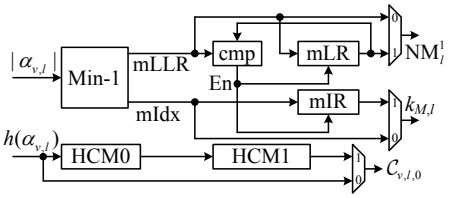

Figure 6.8: Hardware architecture of the proposed NG-I$_l$

When $I_v > T$, suppose T is a power of 2, then I_v can be divided by T. During each clock cycle, only T LLRs are fed to NG-I$_l$, and the minimum value and its corresponding index are computed in a partial parallel way. The minimum value and associated index of the first T inputs are stored in mLR and mIR, respectively. The minimum value of the second group of T inputs is compared with the current value stored in mLR, and is stored in mLR if it is smaller than the current value of mLR. This repeats until the whole LLR vector $\alpha_{v,l}$ is processed. At last, the minimum value of $|\alpha_{v,l}|$ and its index are stored in mLR and mIR, respectively. The hard decoding of $\alpha_{v,l}$ is stored in the hard decoded constituent codeword memory (HCM0), and is copied to HCM1 when the second most reliable constituent codeword is computed.

The micro-architecture of NG-II$_l$ under $X_0 = 8$ and $X_1 = 16$ is shown in Fig. 6.9,

where the block MUX4T256 includes 256 4-to-1 multiplexers. Our NG-II$_l$ consists of two parts, where the first part calculated 2^{I_v} node metrics, NM_l^0, NM_l^1, $\cdots$, $\mathrm{NM}_l^{2^{I_v}-1}$, based on Alg. 22. The second part implements the first stage sorting of our MBS algorithm. For $L = 4$, when $2^{I_v} > 2L$, the 2^{I_v} metrics are first divided into four groups. The Min-2 [81] block is modified slightly to find the two minimum node metrics and their associated indices for each metric group. The MS_{8-4} block calculates the final output metrics. When $2^{I_v} = 2L = 8$, the MS_{8-4} blocks work directly on the $2L = 8$ expanded path metrics. When $2^{I_v} \leqslant L$, the expanded path metrics are output directly. As shown in Figs. 6.7 to 6.9, our PPU has long critical path delay, since there are many levels of logic from the inputs to outputs. Pipelines should be used to improve overall decoder frequency.

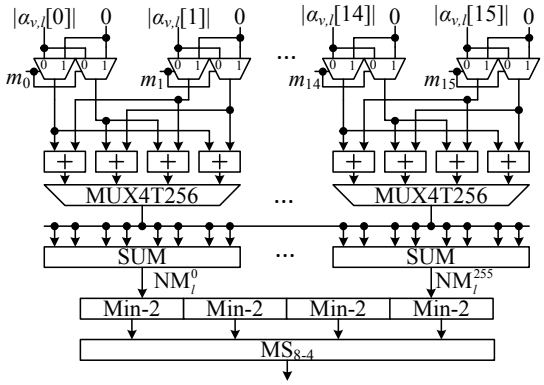

Figure 6.9: Architecture of NG-II$_l$

Based on the DMM method in Eq. (6.9), the node metric computation part needs $2^{I_v}(2^{n-t}-1)L$ adders and $2^{I_v}2^{n-t}L$ 2-to-1 multiplexers, where $N = 2^n$ and t is the layer index of an FP node v. Based on the RCC method, it takes $(\sum_{i=1}^{n-t-1} 2^i 2^{2^{n-t-i}} + 2^{I_v})L$ adders, $2^{I_v+1}L$ $2^{2^{n-t-1}}$-to-1 multiplexers and $4 \times 2^{n-t-1}L$ 2-to-1multiplexers. In contrast, based on our DRH method, it takes $4 \times 2^{n-t-1} + 2^{I_v}(2^{n-t-1}-1)$ adders,

$2^{I_v}2^{n-t-1}$ 4-to-1 and $4 \times 2^{n-t-1}$ 2-to-1 multiplexers. Table 6.2 compares hardware resources needed by the DMM, RCC and DR-Hybrid methods when $X_0 = 8$, $X_1 = 16$, and $\alpha_{v,l}[j]$ $(0 \leq j < 2^{n-t})$ is a 6-bit LLR. As shown in Table 6.2, the DRH method requires the smallest total area. Besides, the implementations based on DMM, RCC and DRH have roughly the same critical path delay.

Table 6.2: Hardware resources needed by different methods per list

	DMM	RCC	DRH
# of adders	3840	864	1824
# of MUX_{2-1}	4096	32	32
# of MUX_{4-1}	0	0	2048
# of MUX_{256-1}	0	512	0
total area (# of NANDs)	313,967	1,673,810	229,449

6.4.4 Proposed hybrid partial sum computation unit

For the list decoder architectures in [31, 32], all partial sums are stored in registers and the partial sums of decoding path l' are copied to decoding path l when decoding path l' needs to be copied to decoding path l. The PSU in [31] and [32] needs $L(N-1)$ and $L(\frac{N}{2}-1)$ single bit registers to store all partial sums, respectively. Thus, for large N, the register based PSU architectures in [31, 32] are inefficient for two reasons. First, the area of the PSU is linearly proportional to N. For large N (e.g. $N > 2^{15}$), the area of PSU is large since registers are usually area demanding. Second, the power dissipation due to the copying of partial sums between different decoding paths is high when N is large.

Proposed Index Based Partial Sum Computation Algorithm

In order to avoid copying partial sums directly, an index based partial sum computation (IPC) algorithm is proposed in Algorithm 23, where $p_l[z]$ ($l = 0, 1, \cdots, L-1$ and $z = 0, 1, \cdots, n$) is a list index reference. $C_{l,z}$ for $l = 0, 1, \cdots, L-1$ and $z = 0, 1, \cdots, n$ are partial sum matrices [32, 55]. $C_{l,z}$ has 2^{n-z} elements, each of which stores two binary bits.

For our RLLD algorithm, once a rate-0, rate-1 or an FP node sends L codewords to its parent node, the partial sum computation is performed after decoding path pruning. Let t denote the layer index of such a node v. Let $(B_{n-1}, B_{n-2}, \cdots, B_0)$ denote the binary representation of the index of the last leaf node belonging to node v, where B_{n-1} is the most significant bit. Let $t_e = n - j$, where j is the smallest integer such that $B_j = 0$. If $B_j = 1$ for $j = 0, 1, \cdots, n-1$, $t_e = 0$. Once $\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1}$ are calculated, decoding path l' may need to be copied to path l before the following partial sum computation. Under this circumstance, the index references are first copied, where $p_{l'}[z]$ is copied to $p_l[z]$ for $z = t, t-1, \cdots, 0$. The lazy copy algorithm was proposed in [55] to avoid copying partial sums directly. However, the lazy copy algorithm is not suitable for hardware implementation due to complex index computation. The PSU in [32] copies partial sums directly.

Micro Architecture of the Proposed Hybrid Partial Sum Unit

Based on our IPC algorithm, a Hyb-PSU is proposed with two improvements. First, some partial sums are stored in memory, while others are stored in registers. Second, instead of partial sums, only list index matrices are copied. These two improvements reduce the area and power overhead of partial sum computation unit when N is large.

Algorithm 23: Index Based Partial Sum Computation (IPC) Algorithm

input : $t_e, t, (\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1})$
output: $C_{l,t_e}[j][0]$ for $l = 0, 1, \cdots, L-1$ and $j = 0, 1, \cdots, 2^{n-t_e}$

for $l = 0$ **to** $L-1$ **do**
  for $j = 0$ **to** $2^{n-t} - 1$ **do**
    if *v is the left child node of its parent node* **then**
      $C_{l,t}[j][0] = \beta_{v,l}[j]$; $p_l[t] = l$
    else $C_{l,t}[j][1] = \beta_{v,l}[j]$
if *v is the left child node of its parent node* **then exit**
for $l = 0$ **to** $L-1$ **do**
  for $z = t-1$ **to** t_e **do**
    for $j = 0$ **to** 2^{n-z-1} **do**
      $v_0 = C_{p_l[z+1],z+1}[j][0]$; $v_1 = C_{l,z+1}[j][1]$
      if $z == t_e$ **then**
        $C_{l,z}[2j][0] = v_0 \oplus v_1$; $C_{l,z}[2j+1][0] = v_1$
        $p_l[z+1] = p_l[z] = l$
      else
        $C_{l,z}[2j][1] = v_0 \oplus v_1$; $C_{l,z}[2j+1][1] = v_1$
        $p_l[z+1] = l$

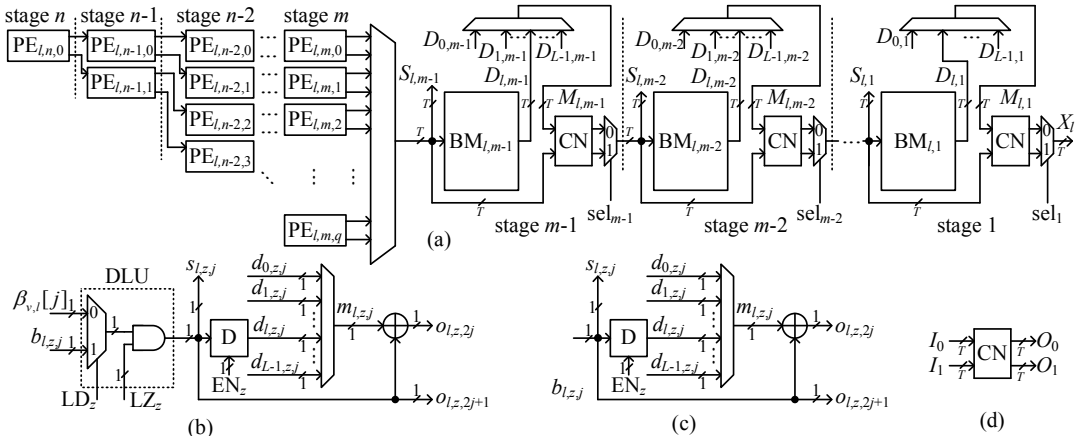

Figure 6.10: (a) Top architecture of CU_l. (b) Type-I PE. (c) Type-II PE. (d) Inputs and outputs of the CN.

The Hyb-PSU consists of L computation units, CU_0, CU_1, $\cdots$, CU_{L-1}, where the micro architecture of CU_l is shown in Fig. 6.10(a) and is described as follows.

(a) For block length $N = 2^n$, CU_l consists of n stages, where m is an integer and the first $n - m + 1$ stages are a binary tree of the type-I and type-II unit processing elements (PEs) shown in Figs. 6.10(b) and 6.10(c), respectively. Stage z ($z \geqslant m$) has 2^{n-z} PEs. Each of the remaining $m - 1$ stages has the same circuitry.

(b) Two types of PEs are used in the PE tree in Fig. 6.10(a). Suppose the maximal length of a constituent codeword that is returned from a rate-0, rate-1 or FP node is 2^{μ}, then stage z ($z \geqslant n - \mu$) employs only the type-I PEs. The remaining stages in the PE tree employ the type-II PEs.

(c) Compared with the type-II PE, the type-I PE has an extra data load unit (DLU). For $\mathrm{PE}_{l,z,j}$ within stage z ($j = 0, 1, \cdots, 2^{n-z} - 1$), the binary outputs, $o_{l,z,2j}$ and $o_{l,z,2j+1}$, are connected to $b_{l,z-1,2j}$ and $b_{l,z-1,2j+1}$, respectively. The wired connections are not shown in Fig. 6.10(a) for simplicity.

(d) $\mathrm{BM}_{l,z}$ ($z \leqslant m - 1$) is a bit memory with $c_w = \frac{2^{n-z}}{T}$ words, where each word

contains T bits. T is the number of processing elements belonging to a decoding path in a partial parallel list decoder. For our memory compiler, if c_w is greater than a threshold value, then $\mathrm{BM}_{l,z}$ is implemented with an RF. If c_w is even greater than another threshold value, then $\mathrm{BM}_{l,z}$ is implemented with an SRAM.

(e) The connector module (CN) has two T-bit inputs and two T-bit outputs. The connections between the outputs and inputs are

$$\begin{cases} O_0[2j] &= I_0[j] \oplus I_1[j] & 0 \leqslant j < T/2 \\ O_0[2j+1] &= I_1[j] & 0 \leqslant j < T/2 \\ O_1[2j-T] &= I_0[j] \oplus I_1[j] & T/2 \leqslant j < T \\ O_1[2j+1-T] &= I_1[j] & T/2 \leqslant j < T \end{cases} \tag{6.11}$$

(f) For our Hyb-PSU, L computation units are needed. For each PE within CU_l, $m_{l,z,j}$ in Figs. 6.10(b) and 6.10(c) is the output of an L-to-1 multiplexer whose inputs are $d_{0,z,j}$, $d_{1,z,j}$, $\cdots$, $d_{L-1,z,j}$, where $L-1$ of them are from other computation units. For each CN, $M_{l,z}$ is the output of an L-to-1 multiplexer whose inputs are $D_{0,z}, D_{1,z}, \cdots, D_{L-1,z}$.

Computation Schedule of Our Hybrid Partial Sum Unit

Once the returned L codewords $\beta_{v,0}, \beta_{v,1}, \cdots, \beta_{v,L-1}$ are computed, the path pruning unit also outputs L indices $a_0, a_1, \cdots, a_{L-1}$, where a_l needs to be copied to decoding path l. For $l = 0, 1, \cdots, L-1$, $\beta_{v,l}$ is first loaded into stage t by the DLU in Fig. 6.10(b), and the output partial sums in Alg. 23 come out from stage t_e. For stage t, if $\beta_{v,l}$ is sent from a rate-0 node, then the control signal LZ_t is 0, since $\beta_{v,l}$ is a zero vector. Otherwise, $\mathrm{LD}_t = 0$ and $\mathrm{LZ}_t = 1$. For the other stages, $\mathrm{LD}_z = 1$

and $\mathrm{LZ}_z = 1$ $(z \neq t)$.

For all partial sums within the partial sum matrix $C_{l,z}$, we divide them into two sets: $\mathbf{C}^0_{l,z}$ and $\mathbf{C}^1_{l,z}$, where $\mathbf{C}^0_{l,z}$ consists of $C_{l,z}[j][0]$ for $j = 0, 1, \cdots, 2^{n-z}-1$ and $\mathbf{C}^1_{l,z}$ consists of the other partial sums within $C_{l,z}$. For each $C_{l,z}$, our Hyb-PSU stores only $\mathbf{C}^0_{l,z}$ in the registers or bit memory of stage z. As shown in Alg. 23, for $z = t-1$ to $t_e + 1$, $\mathbf{C}^1_{l,z}$ is computed in serial. At last, $\mathbf{C}^0_{l,t_e}$ is computed. For our Hyb-PSU, after loading the returned L codewords into stage t, for $z = t-1$ to $t_e + 1$, $\mathbf{C}^1_{l,z}$ is computed on-the-fly and passed to the next stage as shown in Fig. 6.10.

When $t_e \geqslant m$, $\mathbf{C}^0_{l,t_e}$ is computed in one clock cycle and is output from stage t_e, where $C_{l,t_e}[j][0]$ is set to $s_{l,t_e,j}$ produced by the type-I and type-II PEs for $j = 0, 1, \cdots, 2^{n-t_e}-1$. When $t_e < m$, $\mathbf{C}^0_{l,t_e}$ is computed in $2^{n-t_e}/T$ cycles, and T updated partial sums are computed in each clock cycles. Since decoding path a_l needs to be copied to path l, for $z = t, t-1, \cdots, t_e + 1$, the computation of $\mathbf{C}^1_{l,z}$ is based on $\mathbf{C}^0_{a_l,z+1}$ and $\mathbf{C}^1_{l,z+1}$. Hence, the multiplexers within stage z are configured so that $m_{l,z,j} = d_{a_l,z,j}$ for $z \geqslant m$. When $z < m$, $M_{l,z} = Q_{a_l,z}$.

Comparisons with Related Works

Compared to the partial sum computation architectures in [31, 32], the proposed Hyb-PSU architecture has advantages in the following two aspects.

(1) The proposed Hyb-PSU is a scalable architecture. The PSU architectures in [31,32] require $L(N-1)$ and $L(N/2-1)$ single bit registers, where $N = 2^n$ is the block length. Hence, they will suffer from excessive area overhead when the block length N is large. In contrast, the proposed Hyb-PSU stores $L(N-1)$ bits and most of these bits are stored in RFs or SRAMs, which are more area efficient than

registers.

(2) The architectures in [31,32] copies partial sums of a decoding path to another decoding path when needed, while our Hyb-PSU copies only index references. We define the copying of a single bit from one register to another as a single copy operation. When decoding path l' needs to be copied to path l, the PSU in [32] requires $N_1 = 2^{n-1} - 1$ copy operations, while our Hyb-PSU needs only $N_2 = (n + 1) \log_2 L$ copy operations. Since the value of L for practical hardware implementation is small, our lazy copy needs much fewer copy operations than direct copy.

In this chapter, when $L = 4$ and $T = 128$, for $N = 2^{13}$ and 2^{15}, the proposed hybrid partial sum computation unit architecture is implemented with $m = 3$ and $m = 5$, respectively, under a TSMC 90nm CMOS technology. Our partial sum computation unit consumes an area of 0.779mm^2 and 1.31mm^2 for $N = 2^{13}$ and $N = 2^{15}$, respectively.

To the best of our knowledge, those decoder architectures in [31,32,36,82] are the only for SC based list decoding algorithms of polar codes. However, in [31, 36, 82], the partial sum computation unit architecture was not discussed in detail and the implementation results on the PSU alone are not shown. Hence, we compare our proposed Hyb-PSU with that in [32]. When $L = 4$, the partial sum unit architecture in [32] for $N = 2^{13}$ and 2^{15} consumes an area of 1.011mm^2 and 3.63mm^2, respectively, under the same CMOS technology. All PSUs are synthesized under a frequency of 500MHz. Our Hyb-PSU achieves an area saving of 23% and 63% for block length 2^{13} and 2^{15}, respectively.

6.4.5 Latency and Throughput

For the proposed high throughput decoder architecture, the number of clock cycles, N_D, used on the decoding of a codeword depends on the block length, the code rate and the positions of frozen bits. For our RLLD algorithm, let N_V be the number of nodes (except the root node) visited in G_n. Let S_V denote the set of indices of visited nodes (except the root node). Let S'_V be a subset of S_V, where S'_V consists of rate-1 nodes with $I_v \leqslant X_{th}$ and all FP nodes. For $v_i \in S_V$, let t_i be the layer index of node v_i for $i = 0, 1, \cdots, N_V - 1$. Then

$$N_D = \sum_{i=0}^{N_V-1} (N_L^{(i)} + N_P^{(i)}) + N_C, \tag{6.12}$$

where $N_L^{(i)} = \lceil \frac{2^{n-t_i}}{T} \rceil$ is the number of clock cycles needed to calculate the LLR vectors sending to node v_i. $N_P^{(i)}$ is the number of clock cycles used by our PPU when v_i is activated. Note that decoding path splits only if node v_i is a rate-1 node with $I_v \leqslant X_{th}$ or an FP node. Hence, $N_P^{(i)} = 0$ if $v_i \notin S'_V$. If $v_i \in S'_V$, $N_P^{(i)} \neq 0$ and depends on the node type, X_{th}, $q_{I_v,L}$, T, L and the number of pipeline stages in our PPU. This will be discussed in more detail in Section 6.5.

Since our list decoder outputs x_0^{N-1} instead of u_0^{N-1}, we need to obtain u_0^{N-1} based on x_0^{N-1} before calculating the CRC checksum of the information bits. A partial-parallel polar encoder [83] can be used and the corresponding latency is N/T when T bits are fed to the encoder in parallel. For the computation of CRC, a partial parallel CRC unit [84] can be used, and the corresponding latency is also N/T. As a result, $N_C = \frac{2N}{T}$.

The latency of our decoder is $T_L = N_R/f$, where f is the decoder frequency.

Since we are using CRC for output final data word, we calculate the net information throughput (NIT) of our decoder, where NIT $= \frac{(NR-h)f}{N_D-N_C}$, where h is the CRC checksum length. Here, the latency due to the CRC checksum computation does not affect out decoder throughput, since our decoder can work on the next frame once our Hyb-PSU begins to output decoded codewords for current frame.

Table 6.3: Implementation Results for $N = 2^{10}, R = 0.5$

	proposed			[33]			[32]‡			[36]				[37]
L	2	4	8	2	4	8	2	4	8	2		4		4
Frequency (MHz)	423	403	289	847	794	637	507	492	462	500*	361†	400*	288†	500
Cell Area (mm^2)	1.92	3.69	6.95	0.88	1.78	3.85	1.23	2.46	5.28	1.06*	2.03‡	2.14*	4.10†	1.403
# of Decoding Cycles	337	371	404	2592	2649	2649	2592	2592	3104	1022		1022		1290
NIT (Mbps)	666	570	374	168	154	123	93	91	71	250*	180†	200*	144†	186
Latency (us)	0.79	0.92	1.21	3.06	3.34	4.16	5.11	5.26	6.72	2.04*	2.83‡	2.55*	3.54†	2.58
AE (Mbps/mm^2)	347	154	53	191	86	32	76	37	13	237*	88†	94*	35†	132

‡The decoder architecture in [32] has been re-synthesized under the TSMC 90nm CMOS technology. * These are the original implementation results based on a 65nm CMOS technology. †These are the scaled results under the TSMC 90nm CMOS technology.

Table 6.4: Implementation Results for $N = 2^{13}, R = 0.5$

	proposed			[33]†			[32]‡			[37]‡
L	2	4	8	2	4	8	2	4	8	4
Frequency (MHz)	416	398	289	847	794	637	467	434	434	434
Cell Area (mm^2)	3.12	5.86	11.06	6.48	12.73	28.04	3.97	7.93	17.45	7.02
# of Decoding Cycles	2146	2367	2576	20736	20736	20736	20736	20736	24832	11488
NTT (Mbps)	839	723	479	167	156	125	92	85	71	153
Latency (us)	5.16	5.94	8.91	24.48	26.11	32.55	44.40	47.78	58.56	26.47
AE (Mbps/mm^2)	268	123	43	26	12	4.6	23	11	4.1	21.79

†These results are estimated conservatively. ‡The decoder architectures in [32, 37] have been re-synthesized under the TSMC 90nm CMOS technology. The number of PU per decoding path is 128.

Table 6.5: Implementation Results for $N = 2^{15}, R = 0.9004$

	proposed			[33]†			[32]‡			[37]‡
L	2	4	8	2	4	8	2	4	8	4
Frequency (MHz)	367	359	286	847	794	637	398	389	389	389
Cell Area (mm^2)	4.56	8.56	16.45	25.68	50.41	111.08	8.59	17.54	34	15.5
# of Decoding Cycles	6070	6492	6895	96576	96576	96576	96576	96576	126080	63606
NIT (Mbps)	1949	1772	1323	258	242	194	121	118	90	180
Latency (us)	16.53	18.08	24.11	114.02	121.63	151.61	242.65	248.26	324.1	163.5
AE (Mbps/mm^2)	427	207	80	11	4.8	1.75	14	6.72	2.64	11.61

†These results are estimated conservatively. ‡The decoder architectures in [32, 37] have been re-synthesized under the TSMC 90nm CMOS technology. The number of PU per decoding path is 128.

6.5 Implementation Results and Comparisons

To compare with prior works, we implement our high throughput list decoder architecture for three polar codes with lengths of 2^{10}, 2^{13} and 2^{15}, respectively, and rates 0.5, 0.5 and 0.9, respectively. The last polar code is intended for storage applications. For each code, three different list sizes are considered: $L = 2, 4, 8$. All our decoders are synthesized under the TSMC 90nm CMOS technology using the Cadence RTL compiler. The area efficiency (AE) of a partly parallel decoder architecture depends on the number of PUs. In order to make a fair comparison with prior works in [32, 33, 37], the number of PUs for each decoding path of our implemented decoders is selected to be 64 when $N = 2^{10}$. When $N = 2^{13}$ and 2^{15}, the number of PUs per decoding path is 128 for our decoders. The list decoders in [85] are based on a line architecture, which always requires $\frac{N}{2}$ PUs.

A total of 3, 4 and 6 pipeline stages, respectively, are inserted in the PPU for decoders with $L = 2$, 4 and 8, respectively. The number of pipeline stages needed for our PPU is determined by the longest data path. For each $v_i \in S'_V$, if node v_i is a rate-1 node with $I_v \leqslant X_{th}$, $N_P^{(i)}$ depends on the number of PUs in a decoding path: when $I_v \leqslant T$, $N_P^{(i)} = 2$ for all our implemented decoders; otherwise, $N_P^{(i)} = 4$ for all our decoders, since the minimum value of a received LLR vector is calculated in a partial parallel way, which incurs extra clock cycles. When node v_i is an FP node, $N_P^{(i)}$ relates to $q_{I_v,L}$. Depending on the detailed value of $q_{I_v,L}$, we may use different data paths when computing the L minimum expanded path metrics. The locations of all pipelines are arranged so that a fewer number of clock cycles is needed when the $q_{I_v,L}$ is smaller. In Table 6.6, we list the detailed value of $N_P^{(i)}$ with respect to I_v and L.

The selection of X_{th} is a trade-off between AE and error performance. When increasing X_{th}, more rate-1 nodes will be processed by our CG algorithm. Hence, N_D increases and the resulting NIT decreases. Meanwhile, the corresponding error performance is better especially in high SNR region. Our high throughput list decoder architecture supports all X_{th} values. For all our implemented decoders, X_{th} is set to be large enough so that all rate-1 nodes are processed by our CG algorithm. In this setup, for each implemented decoder, N_D is maximized with respect to X_{th}, and hence the throughput of our decoder architecture in Tables 6.3, 6.4 and 6.5 is the *minimum* achieved by our decoders. For each code, the corresponding error performance is better than that of the RLLD with the MEQ in Fig. 6.6.

Table 6.6: $N_P^{(i)}$ with Respect to I_v and L

I_v	1	2	3	4	5	6	7	8
$L=2$	2	2	3	3	3	3	3	3
$L=4$	2	4	4	4	4	4	4	3
$L=8$	2	3	4	5	5	6	5	3

The implementation results are shown in Table 6.3, 6.4 and 6.5. The implementation results show that our decoders outperform existing SCL decoders [32, 33, 36] in both decoding latency and area efficiency. Compared with the decoders of [33], the area efficiency and decoding latency of our decoders are 1.65 to 45 times and 3.4 to 6.8 times better, respectively. The area efficiency and decoding latency of our decoders are 4.07 to 30 times and 5.5 to 13 times better, respectively, than the decoders of [32]. Compared with decoders of [37], our decoders improve the area efficiency and decoding latency by 1.16 to 17.8 times and 2.8 to 9 times, respectively. When $N = 2^{10}$, the area efficiency and decoding latency of our decoders are 3.9 to 4.4 times and 3.58 to 3.84 times better, respectively, than the decoders of [36].

Compared with the decoders of [36], our decoders would show more significant improvements in area efficiency and decoding latency when N is larger.

Based on the implementation results shown in Tables 6.3, 6.4 and 6.5, it is observed that when the block length is fixed, as the list size L increases, the area efficiency and decoding latency will decrease and increase, respectively, due to the following reasons:

- It takes more memory to store internal LLRs when L increases.
- The number of pipeline stages within our PPU will increase when L increases, which in turn increases the overall decoding clock cycles.

The latency reduction and area efficiency improvement of our decoders are due to the reduced number of nodes activated in the decoding. However, the area and frequency overhead of the proposed PPU somewhat dilute the effects due to decoding clock cycles reduction. For example, our decoder reduces the number of decoding cycles to approximately $\frac{1}{7}$ of that of the decoders in [33] for $L = 2$, 4 and 8. However, the reduction in decoding cycles does not fully transfer into the improvement in decoding latency and area efficiency. Based on our implementation results, take $L = 2$ as an example, the PPU occupies 61.99%, 40.16% and 25.40% of the area of the whole decoder, for $N = 2^{10}$, 2^{13} and 2^{15}, respectively. Compared with the decoders with $N = 2^{10}$ and 2^{13}, the effects on the area efficiency caused by the area overhead of PPU are smaller for decoders with $N = 2^{15}$. Keeping T unchanged, as N increases, the area of the PPU increases very slowly while the total area of all LLR memories is proportional to N. Hence, for larger N, PPU occupies a smaller percentage of the total area of a whole decoder. When list size L is fixed, as N

increases, the latency reduction and area efficiency improvement compared with other decoders in the literature will be greater.

6.6 Conclusion

In this chapter, a reduced latency list decoding algorithm is proposed for polar codes. The proposed list decoding algorithm results in a high throughput list decoder architecture for polar codes. A memory efficient quantization method is also proposed to save the size of message memories. The proposed list decoder architecture can be adapted to large block lengths due to our hybrid partial sum computation unit, which is area efficient. The implementation results of our high throughput list decoder demonstrates significant advantages over current state-of-art SCL decoders.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, efficient hardware decoder architectures for NB-LDPC codes, polar codes, MV and KK codes are presented. An algorithm-architecture co-optimization approach is employed.

In Chapter 2, the shuffled decoding algorithm and decoder architecture are presented. The shuffled decoding algorithm reduces the average number iterations. Implementations for the (837, 726) nonbinary QC-LDPC code show that the efficiency of the proposed decoder architecture is much higher than these previous works.

In Chapter 3, a fully parallel decoder architecture based on the is presented. A reduced memory complexity trellis based check node processing (RTBCP) algorithm is first proposed. A parallel check node unit (CNU) and a low-latency variable node

unit (VNU) are also proposed. Based on the proposed CNU and VNU, an efficient fully parallel decoder architecture is also proposed. A fully parallel NB-LDPC decoders based on GF(256) is implemented with 28nm CMOS technology. The decoder over GF(256) achieves a throughput of 546Mb/s and an energy efficiency of 0.178nJ/b/iter. Compared with the state-of-art NB-LDPC decoder architectures, the implementation results demonstrate that our fully parallel decoder architecture has obvious advantages in terms of throughput and area efficiency.

In Chapter 4, efficient decoder architectures for KK and MV codes are presented. A serial decoder architecture and an unfolded decoder architecture for KK codes are proposed for applications with moderate and high throughputs, respectively. Both architectures are implemented for KK codes over GF(2^8) and GF(2^{16}) to demonstrate their efficiency. Compared to the rank metric decoder architectures for KK codes [43], the proposed serial decoder architecture improves the throughput by 4.9 and 13.2 times, while its gate counts are only 56% and 76% of their respective counterparts in [43]. Moreover, for these two codes, the unfolded architecture achieves a throughput of 12.5Gb/s and 41.6Gb/s, much higher than the throughput of 214Mb/s and 134Mb/s of their respective counterparts in [43]. The throughputs per thousand NAND gates of our architectures are much higher and their latency much shorter than their counterparts in [43]. A serial list decoder architecture for MV codes is also proposed. To the best of our knowledge, this is the first hardware implementation of MV decoders.

In Chapter 5, we present the first hardware implementation of the CA-SCL algorithm to the best of our knowledge. An memory efficient memory partition method is employ to reduce the area of the message memories. A fine grained PU

profiling (FPP) algorithm is proposed to determine the minimum quantization size of each input message for each processing unit so that there is no message overflow. An efficient scalable path pruning unit (PPU) is proposed to control the copying of decoding paths. Based on the proposed memory architecture and the scalable PPU, our list decoder architecture is suitable for large list sizes. For a (1024, 512) rate-$\frac{1}{2}$ polar code, the proposed list decoder architecture is implemented for list size $L = 2$ and 4, respectively, under a 90nm CMOS technology. Compared with the decoder architecture in [31] synthesized under the same technology, our decoder achieves 1.24 to 1.83 times area efficiency (throughput normalized by area). Besides, the proposed CA-SCL decoder has better error performance compared with the SCL decoder in [31].

In Chapter 6, a tree based reduced latency list decoding algorithm and its corresponding high throughput hardware architecture for polar codes are presented. Our reduced latency list decoding algorithm reduces the number of nodes visited in a decoding tree. The proposed high throughput list decoder architecture has been implemented for several block lengths and list sizes under the TSMC 90nm CMOS technology. The implementation results show that our decoders outperform existing SCL decoders in both decoding latency and area efficiency. For example, compared with the decoders of [33], the area efficiency and decoding latency of our decoders are 1.65 to 45 times and 3.4 to 6.8 times better, respectively.

7.2 Future Work

For future work, the following point may be worthy to be looked into:

- **Low complexity low latency decoding algorithms and decoder architectures for NB-LDPC codes.** Compared to binary LDPC decoders, current NB-LDPC still suffer from excessive hardware complexity. For the practical application of NB-LDPC codes, efficient decoding algorithms still need to be investigated. Stochastic computation can reduce the computational complexity at the cost of long decoding latency. It is interesting to combine stochastic computation with current NB-LDPC decoding algorithms over real domain. It is also promising to explore the joint detection and decoding with NB-LDPC codes. Besides, efficient hard decoding algorithms for NB-LDPC codes still need to be investigated.

- **Efficient belief propagation decoding algorithms and decoder architectures for polar codes.** Lots of efforts have already been devoted to the research of efficient SC based decoding algorithms for polar codes. For applications that require soft output, the belief propagation decoding algorithms are essential. Current belief propagation decoding algorithms suffer from high computational complexity, high hardware complexity, inferior error performance due to inefficient message updating schedule. It is interesting to explore efficient schedules for belief propagation decoding of polar codes.

Bibliography

[1] A. Voicila, D. Declercq, F. Verdier, M. Fossorier and P. Urard, "Low-complexity decoding for non-binary LDPC codes in high order fields," *IEEE Trans. Commun.*, vol. 58, no. 5, pp. 1365–1375, May 2010.

[2] G. Sarkis, S. Hemati, S. Mannor and W. J. Gross, "Stochastic decoding of LDPC codes over GF(q)," *IEEE Trans. Commun.*, vol. 61, no. 3, pp. 939–950, Mar. 2013.

[3] M. Davey and D. J. C. Mackay, "Low-density parity check codes over GF(q)," *IEEE Commun. Lett.*, vol. 2, no. 6, pp. 165–167, Jun. 1998.

[4] C. Poulliat, M. Fossorier, and D. Declercq, "Design of non binary LDPC codes using their binary image: algebraic properties," in *Proc. IEEE Int. Symp. on Information Theory*, Seattle, USA, Jul. 2006.

[5] D. J. C. MacKay, "Online database of low-density parity check codes, Available: http:www.inference.phy.cam.ac.uk/mackay/codes/data.html."

[6] V. Savin, "Min-Max decoding for non-binary LDPC codes," in *Proc. IEEE Int. Symp. on Information Theory*, Toronto, Canada, Jul. 2008, pp. 960–964.

[7] X. Zhang and F. Cai, "Reduced-complexity decoder architecture for non-Binary LDPC codes," *IEEE Trans. VLSI Systems*, vol. 19, no. 7, pp. 1229–1238, Jul. 2011.

[8] G. Sarkis and W. J. Gross, "Efficient stochastic decoding of non-binary LDPC codes with degree-two variable nodes," *IEEE Commun. Lett.*, vol. 16, no. 3, pp. 389–391, Mar. 2012.

[9] A. Ciobanu, S. Hemati and W. J. Gross, "Adaptive multiset stochastic decoding of non-binary LDPC codes," *IEEE Trans. Signal Processing*, vol. 61, no. 16, pp. 4100–4113, Aug. 2013.

[10] A. Voicila, D. Declercq, F. Verdier, M. Fossorier and P. Urard, "Architecture of a low-complexity non-binary LDPC decoder for high order fields," in *Proc. IEEE Int. Symp. Commun. and Inf. Technologies (ISCIT)*, Sydney, Australia, Oct. 2007, pp. 1201–1206.

[11] J. Lin, J. Sha, Z. Wang, and L. Li, "Efficient decoder design for nonbinary quasicyclic LDPC codes," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 57, no. 5, pp. 1071–1082, May 2010.

[12] J. Lin and Z. Yan, "Efficient shuffled decoder architecture for nonbinary Quasi-Cyclic LDPC codes," *IEEE Trans. VLSI Systems*, vol. 21, no. 9, pp. 1756–1761, Sept. 2013.

[13] J. Lin, J. Sha, Z. Wang, and L. Li, "An efficient VLSI architecture for nonbinary LDPC decoders," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 57, no. 1, pp. 51–56, Jan. 2010.

[14] F. Cai and X. Zhang, "Relaxed Min-Max decoder architectures for nonbinary Low-Density Parity-Check Codes," *IEEE Trans. VLSI Systems*, vol. 21, no. 11, pp. 2010–2023, Nov. 2013.

[15] Y. Ueng, C. Leong, C. Yang, C. Cheng, K. Liao and S. Chen, "An efficient layered decoding architecture for nonbinary QC-LDPC codes," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 56, no. 2, pp. 385–398, Feb. 2012.

[16] X. Chen and C.-L. Wang, "High-Throughput Efficient Non-Binary LDPC Decoder Based on the Simplified Min-Sum Algorithm," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 59, no. 11, pp. 2784–2794, Nov. 2012.

[17] C. Zhang and K. K. Parhi, "A network-efficient nonbinary QC-LDPC decoder architecture," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 59, no. 6, pp. 1359–1371, Jun. 2012.

[18] E. Arıkan, "Channel polariztion: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Info. Theory*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

[19] E. T. E. Sasoglu and E. Arıkan, "Polariztion for arbitrary discrete memoryless channels," in *Proc. IEEE Information Theory Workshop*, 2009, pp. 144–148.

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

[21] I. Tal and A. Vardy, "List decoding of polar codes," in *Proc. IEEE Int. Symp. on Information Theory*, Jul. 2011, pp. 1–5.

[22] ——, "List decoding of polar codes," in *http://webee.technion.ac.il/people/idotal/papers/preprints/polarList.pdf*.

[23] K. Niu and K. Chen, "Crc-aided decoding of polar codes," *IEEE Commun. Lett.*, vol. 16, no. 10, pp. 1668–1671, Oct. 2012.

[24] H. S. B. Li 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.

[25] S. K. N. Goela and M. Gastpar, "On lp decoding of polar codes," in *Proc. IEEE Information Theory Workshop*, Aug. 2010, pp. 1–5.

[26] A. Eslami and H. Pishro-Nik, "On finite-length performance of polar codes: stopping sets, error floor, and concatenated design," *IEEE Trans. Commun.*, accepted.

[27] P. Trifonov, "Efficient design and decoding of polar codes," *IEEE Trans. Commun.*, vol. 60, no. 11, Dec. 2012.

[28] K. Chen, K. Liu, and J. Lin, "Improved successive cancellation decoding of polar codes," *IEEE Trans. Commun.*, vol. 61, no. 8, pp. 3100–3107, Aug. 2013.

[29] C. Zhang and K. K. Parhi, "Low-latency sequential and overlapped architectures for successive cancellation polar decoder," *IEEE Trans. Signal Processing*, vol. 61, no. 10, pp. 2429–2441, Mar. 2013.

[30] A. Alamdar-Yazdi and F. R. Kschischang, "A simplified successive-cancellation decoder for polar codes," *IEEE Commun. Lett.*, vol. 15, no. 12, pp. 1378–1380, Dec. 2011.

[31] A. Balatsoukas-Stimming, A. J. Raymond, W. J. Gross and A. Burg, "Tree search architecture for list SC decoding of polar codes," in *arXiv:1303.7127*.

[32] J. Lin and Z. Yan, "An efficient list decoder architecture for polar codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2015, to appear.

[33] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg, "LLR-based successive cancellation list decoding of polar codes," http://arxiv.org/abs/1401.3753v3, submitted to IEEE Trans. Signal Process. [Online]. Available: http://arxiv.org/abs/1401.3753

[34] ——, "LLR-based successive cancellation list decoding of polar codes," in *Proc. IEEE Int. Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Florence, Italy, May 2014, pp. 3903–3907.

[35] B. Li, H. Shen, and D. Tse, "Parallel decoders of polar codes," http://arxiv.org/abs/1309.1026v1, Sep. 2013.

[36] 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.*, to appear.

[37] C. Xiong, J. Lin, and Z. Yan, "Symbol-decision successive cancellation list decoder for polar codes," http://arxiv.org/abs/1501.04705, submitted to IEEE Trans. Signal Processing.

[38] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Increasing the speed of polar list decoders," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, Belfast, UK, 2014.

[39] R. Ahlswede, N. Cai, S. Li, and R. Yeung, "Network information flow," *IEEE Trans. Info. Theory*, vol. 46, pp. 1204–1216, Jul. 2000.

[40] P. A. Chou, Y. Wu, and K. Jain, "Practical network coding," in *Allerton Conf. on Comm., Control, and Computing*, Monticello, IL, oct 2003.

[41] T. Ho, M. Médard, R. Kötter, D. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Trans. Info. Theory*, vol. 52, no. 10, pp. 4413–4430, Oct. 2006.

[42] T. Ho, R. Kötter, M. Médard, D. R. Karger, and M. Effros, "The benefits of coding over routing in a randomized setting," in *Proc. IEEE Int. Symp. on Information Theory*, Yokohama, June-July 2003, p. 442.

[43] N. Chen, Z. Yan, M. Gadouleau, Y. Wang and B. W. Suter, "Rank metric decoder architectures for random linear network coding with error control," *IEEE Trans. VLSI Syst.*, vol. 20, no. 2, pp. 296–309, feb 2012.

[44] R. Kötter and F. R. Kschischang, "Coding for errors and erasures in random network coding," *IEEE Trans. Info. Theory*, vol. 54, no. 8, pp. 3579–3591, August 2008.

[45] N. Cai and R. W. Yeung, "Network coding and error correction," in *Proc. IEEE Information Theory Workshop*, Bangalore, India, oct 2002, pp. 20–25.

[46] D. Silva, F. R. Kschischang and R. Kötter, "A rank-metric approach to error control in random network coding," *IEEE Trans. Info. Theory*, vol. 54, no. 9, pp. 3951–3967, September 2008.

[47] H. Mahdavifar and A. Vardy, "Algebraic list-decoding on the operator channel," in *Proc. IEEE Int. Symp. Info. Theory*, Austin, USA, June 2010, pp. 1193–1197.

[48] ——, "Algebraic list-decoding of subspace codes," *http://arxiv.org/abs/1202.0338*.

[49] ——, "Algebraic list-decoding of subspace codes with multiplicities," in *Proc. 2011 Allerton Conf. Communications, Control and Computing*, Illinois, USA, Sep. 2011, pp. 1430–1437.

[50] V. Guruswami and M. Sudan, "Improved Decoding of Reed-Solomon Codes and Algebraic Geometry Codes," *IEEE Trans. Info. Theory*, vol. 45, no. 6, pp. 1757–1767, sep 1999.

[51] H. Xie, J. Lin, Z. Yan and B. W. Suter, "Linearized polynomial interpolation and its applications," *IEEE Trans. Signal Processing*, vol. 61, no. 1, pp. 206–217, Jan. 2013.

[52] G. He, G. Sarkis, S. Hemati, W. J. Gross and B. Bai, "Low-complexity channel-likelihood estimation for non-binary codes and QAM," *IEEE Commun. Lett.*, vol. 16, no. 6, pp. 801–804, Aug. 2012.

[53] M. Sudan, "Decoding of Reed-Solomon codes beyond the error-correction bound," *J. Complexity*, vol. 13, pp. 180–193, 1997.

[54] G. Sarkis and W. J. Gross, "Increasing the throughput of polar decoders," *IEEE Commun. Lett.*, vol. 17, no. 9, pp. 725–728, Apr. 2013.

[55] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Trans. Info. Theory*, 2015, [Online; DOI: 10.1109/TIT.2015.2410251].

[56] L. Barnault and D. Declercq, "Fast decoding algorithm for LDPC over GF(2^q)," in *Proc. IEEE Information Theory Workshop*, 2003, pp. 70–73.

[57] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Log-domain decoding of LDPC codes over GF(q)," in *Proc. IEEE Int. Conf. Commun*, Paris, France, Jun. 2004, pp. 772–776.

[58] D. Declercq and M. Fossorier, "Decoding algorithms for nonbinary LDPC codes over GF(q)," *IEEE Trans. Commun.*, vol. 55, no. 4, pp. 633–643, Apr. 2007.

[59] X. Chen, S. Lin and V. Akella, "Efficient configurable decoder architecture for nonbinary quasic-cyclic LDPC codes," *IEEE Trans. Circuits Syst. I: Reg. Papers*, 2012.

[60] C. Lin, K. Lin, H. Chan, and C. Lee, "A 3.33 Gb/s (1200, 720) low-density parity check code decoder," in *31st Eur. Solid-State Circuits Conf.*, Sep. 2005, pp. 211–214.

[61] M. Beermann, L. Schmalen, and P. Vary, "Improved decoding of binary and non-binary LDPC codes by probabilistic shuffled belief propagation," in *Proc. IEEE Int. Conference on Communications (ICC)*, Kyoto, Japan, Mar. 2011, pp. 1–5.

[62] L. Liu and C.-J. Shi, "Sliced message passing: High throughput overlapped decoding of high-rate low-density parity-check codes," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 55, no. 11, pp. 3697–3710, 2008.

[63] R. M. Tanner, D. Sridhara, A. Sridharan, T. E. Fuja, D. J. Costello, Jr., "LDPC block and convolutional codes based on circulant matrices," *IEEE Trans. Info. Theory*, vol. 50, no. 12, pp. 2966–2984, Dec. 2004.

[64] B. Zhou, Y. Y. Tai, L. Lan, S. Song, L. Zeng, and S. Lin, "Construction of Non-Binary Quasi-Cyclic LDPC Codes by Arrays and Array Dispersions," *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1652–1662, Jun. 2009.

[65] H. Zhong, W. Xu, N. Xie and T. Zhang, "Area-efficient Min-Sum decoder design for high-rate Quasi-Cyclic Low-Density Parity-Check codes in magnetic recording," *IEEE Trans. Magnetics*, vol. 43, no. 12, pp. 4117–4122, 2007.

[66] K. Kasai, M. Hagiwara, H. Imai and K. Sakaniwa, "Quantum error correction beyond the bounded distance decoding limit," *IEEE Trans. Info. Theory*, vol. 58, no. 2, pp. 1223–1230, Feb. 2012.

[67] M. R. Yazdani, S. Hemati, and A. H. Banihashemi, "Improving belief propagation on graphs with cycles," *IEEE Commun. Lett.*, vol. 8, no. 1, pp. 57–59, Jan. 2004.

[68] B. Zhou, J. Kang, S. Song, S. Lin, K. Abdel-Ghaffar, and M. Xu, "Construction of non-binary quasi-cyclic LDPC codes by arrays and array dispersions," *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1652–1662, Jun. 2009.

[69] Y. Tao, Y. S. Park and Z. Zhang, "High-throughput architecture and implementation of regular (2, d_c) nonbinary LDPC decoders," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, Seoul, Korea, May 2012.

[70] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 Low-Density Parity-Check code decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, Mar. 2002.

[71] X.Y. Hu and E. Eleftheriou, "Binary representation of cycle Tanner-graph GF(2^q) codes," in *in Proc. IEEE Int. Conference on Communications (ICC)*, Paris, France, Jun. 2004.

[72] "Design Compiler Graphical," www.synopsys.com/tools/implementation/rtlsynthesis/dcgraphical/Pages/default.aspx, 2013, [Online; accessed 29-July-2013].

[73] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An open-source variation-aware design kit," in *Proc. IEEE Int. Conf. Microelectron. Syst. Education (MSE07)*, San Diego, USA, Jun. 2007, pp. 173–174.

[74] J. Lin and Z. Yan, "An efficient list decoder architecture for polar codes," in *http://arxiv.org/abs/1409.4744*.

[75] C. A. Klein, *www2.ece.ohio-state.edu/~klein/ece766/766-10n.ppt*.

[76] R. Cideciyan and M. Gustlin, "Double burst error detection capability of ethernet CRC," in *http://www.ieee802.org/3/bj/public/jul12/cideciyan_01_0712.pdf*.

[77] K. E. Batcher, "Sorting networks and their applications," in *Proc. ACM spring joint computer conference*, Apr. 1968, pp. 307–314.

[78] S. M. Sait and W. Hasan, "Hardware design and VLSI implementation of a byte-wise CRC generator chip," *IEEE Trans. Consumer Electron.*, vol. 41, no. 1, pp. 195–200, Feb. 1995.

[79] N. Hussami, S. B. Korada, and R. Urbanke, "Performance of polar codes for channel and source coding," in *Proc. IEEE Int. Symp. on Information Theory*, Seoul, South Korea, Jun. 2009, pp. 1488–1492.

[80] J. Lin, C. Xiong, and Z. Yan, "A reduced latency list decoding algorithm for polar codes," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, Belfast, UK, October 2014, pp. 56–61.

[81] C.-L. Wey, M.-D. Shieh, and S.-Y. Lin, "Algorithms of finding the first two minimum values and their hardware implementation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 11, pp. 1549–8328, Dec. 2008.

[82] C. Zhang, X. Yu, and J. Sha, "Hardware architecture for list successive cancellation polar decoder," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, Melbourne, AU, Jun. 2014, pp. 209–212.

[83] H. Yoo and I.-C. Park, "Partially parallel encoder architecture for long polar codes," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, accepted.

[84] C. Cheng and K. K. Parhi, "High-speed parallel CRC implementation based on unfolding, pipelining, and retiming," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 53, no. 10, pp. 1017–1021, Oct. 2006.

[85] B. Yuan and K. K. Parhi, "Low-latency successive-cancellation polar decoder architectures using 2-bit decoding," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 4, pp. 1241–1254, Apr. 2014.

Vita

Jun Lin received the B.S. degree in physics and the M.S. degree in microelectronics from Nanjing University, Nanjing, China, in 2007 and 2010, respectively. From 2010 to 2011, he was an ASIC design engineer with AMD. During summer 2013, he was an intern with Qualcomm Research, Bridgewater, NJ. He is currently working toward the Ph.D. degree in the Department of Electrical and Computer Engineering, Lehigh University, Bethlehem.

His current research interests include low-power high-speed VLSI design, specifically VLSI design for digital signal processing and cryptography. He was a co-recipient of the Merit Student Paper Award at the IEEE Asia Pacific Conference on Circuits and Systems in 2008. He was a recipient of the 2014 IEEE Circuits & Systems Society (CAS) student travel award. Has also a recipient of the 2015 Lehigh University Doctoral Travel Grant for Global Opportunities.