Technische Universität München
Lehrstuhl für Nachrichtentechnik
Prof. Dr. sc. techn. Gerhard Kramer

Master's Thesis

Low Latency Polar FEC Chain Development in Software for 5G

Vorgelegt von:
Yadhunandana Rajathadripura Kumaraiah

München, November 2018

Betreut von:
Dr. Moritz Harteneck, Alexander Heinz, Rohde & Schwarz
Fabian Steiner M.Sc, Peihong Yuan M.Sc

Master's Thesis am
Lehrstuhl für Nachrichtentechnik (LNT)
der Technischen Universität München (TUM)
Titel : Low Latency Polar FEC Chain Development in Software for 5G
Autor : Yadhunandana Rajathadripura Kumaraiah

Yadhunandana Rajathadripura Kumaraiah
Schröfelhofstraße 14 WG 02/02
81375 München
yadhu.kumaraiah@tum.de

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

München, 21.11.2018

....

Ort, Datum (Yadhunandana Rajathadripura Kumaraiah)

Contents

List of Figures

List of Tables

Abstract

Software implementation of error correction and signal processing applications on general purpose processors has gained interest in recent times. Mainly due to latest technological developments in general purpose computing world. Implementations in software have an inherent advantage not being tied to one specific hardware architecture compared to FPGA or ASIC based implementations. They require much less development and maintenance effort compared to hardware implementations. For the device manufacturer software implementation provides platform flexibility in addition to reducing the cost of product.

In this thesis, we study the feasibility of developing a complete polar FEC chain of 5^{th} generation cellular mobile communication standard [4] in software. Specifically on general purpose processors. Thesis work attempts to achieve stringent latency requirements through software, algorithmic and platform specific optimizations. Many algorithms in FEC chain are optimized for hardware implementations. Direct implementation of these algorithms in software results in poor performance. To obtain best performance in terms of latency on general purpose processors, these algorithms are modified or reformulated to suit processor architecture and software implementation. Initially both encoding and decoding FEC chains are implemented naively without any optimization. Code profiling is performed on this naive implementation to identify the significant latency contributors. We split algorithms of significant latency contributing components into primitive operations. These primitive operations are optimized either with software optimizations or mapped to specialized functional units of a general purpose processor to achieve best performance. Specialized units include vector processing units (SSE, AVX and AVX2) and cache-prefetching units.

We concentrate on polar encoding and decoding FEC chain which are used to transmit and receive control information. Latency contributing components are identified. Algorithms of those components are reformulated to avoid or to reduce latency contributing operations. Major latency contributors in encoding FEC chain are the cyclic redundancy check (CRC) calculation, the polar code construction itself and polar encoding. For the

decoding FEC chain subblock deinterleaver, polar decoder, parity bit extraction and CRC calculation constitute the major bottlenecks. Algorithms of these components are reformulated to suit software requirements and implemented using efficient *vector processing instruction sets*. Algorithms are modified to reduce complexity and lookup tables are used to avoid complex computations. Other optimizations include function unrolling, avoiding superfluous copy operations, hints for the compiler for better instruction scheduling and block wise copying et cetera. At the end of both encoding and decoding chapter latency comparisons between naive and optimized implementations are presented. In decoding FEC chain chapter, latencies of decoder of this work and state of the art decoder are compared.

1. Introduction and Motivation

1.1. The Era of Computing and Wireless Communication

The ability to perform computations has evolved tremendously from the day the first computer was invented by Charles Babbage in the 19th century. By the end 19th century another important event occurred, in 1897 an Italian inventor and engineer Guglielmo Marconi demonstrated radio's ability to maintain continuous contact with ships in the English channel. A major breakthrough happened in the development of computers and wireless systems in 1948 when scientists at Bell Labs achieved groundbreaking results. Claude E. Shannon published his paper *"A mathematical theory of communication"*. John Bardeen, Walter Brattain, and William Shockley announced the invention of the *transistor effect*. These two landmark events paved way for the widespread adoption of computers and wireless communication systems in numerous applications. Since then, the telecommunication industry has grown manifold fueled by the advancements in RF and transistor fabrication techniques, miniaturization and very large scale integration. These technological advances made computing devices smaller, cheaper and more reliable. Recent advances in wireless communication have allowed not only short distance communication such as cellular communication but also deep space communication with billions of kilometers distance.

Today computing devices and wireless systems have become integral parts of our society. They allow communication between people even from remote areas. The invention of the internet has enabled people to have access to a world of information in their fingertips. Until recently, wireless devices were primarily used for information exchange between people. Today's wireless applications are entering new avenues such as industrial automation, telemedicine, Autonomous driving. These applications demand ultra-reliability and ultra-low-latency. Latest mobile communication standard 5G took a giant step towards providing service for such mission-critical applications. 5G has adopted several techniques to service stringent latency requirements. To name few, different OFDM numerologies, flexible frame structure et cetera. Traditionally, to achieve stringent latency requirements wireless communication stacks are implemented in hardware, specifically in

FPGAs/ASICs. Hardware implementations make use of implicit hardware-concurrency. However, hardware implementations come with inherent non-flexibility, huge cost and high development time. Due to the latest technological advancements in general-purpose computing, modern processors come with tremendous computation power. It is up to software engineer to efficiently harness this computational power. Modern processors come with special computing units to cater to specific application domains, namely vector processing units for signal processing applications. To achieve stringent latency requirements, it is very important that a software designer makes use of these additional processing elements and available optimization techniques. This work extensively makes use vector processing units through SSE, AVX and AVX2 instruction set extensions.

1.2. Polar Forward Error Correction (FEC) Chain Development in Software

Commonly high-performance signal processing and error correction applications are implemented in hardware either in FPGAs/ASICs. Hardware implementations allow these applications to achieve low latency and high throughput. Algorithms in these applications are developed targeting hardware implementations. These algorithms implemented in software without any reformulation or modification result in poor performance. Therefore algorithms need to be modified and reformulated to achieve expected performance in software.

Optimizations for hardware such as recursive formulation, reducing look-up tables (LUTs) and flip-flops are not always relevant in software. Let's try to understand conflicts in the optimizations targeted for hardware and software. Most of the encoder/decoder algorithms are formulated in a recursive form. In hardware implementations, recursive formulations are particularly useful since same the design can be replicated multiple times without significant effort and also with no performance compromise. However, in software implementations recursive implementation incur with significant overhead. Mainly due to a large number of branches, stack allocation/deallocation, and pipeline flushing. The next optimization steps in hardware targeted implementations are minimizing the required memory and flip-flops. The cost of hardware implementation depends on the amount of memory and number flips-flops required[5]. In contrast, general-purpose computing world can make use of off-the-shelve available cheap memory. Software designer should reduce the number of cache misses and branch miss-predictions[6]. In addition, software implementations should also avoid expensive operations such as multiplications, division, and modulus operations. If not, reformulate them by using inexpensive bitwise

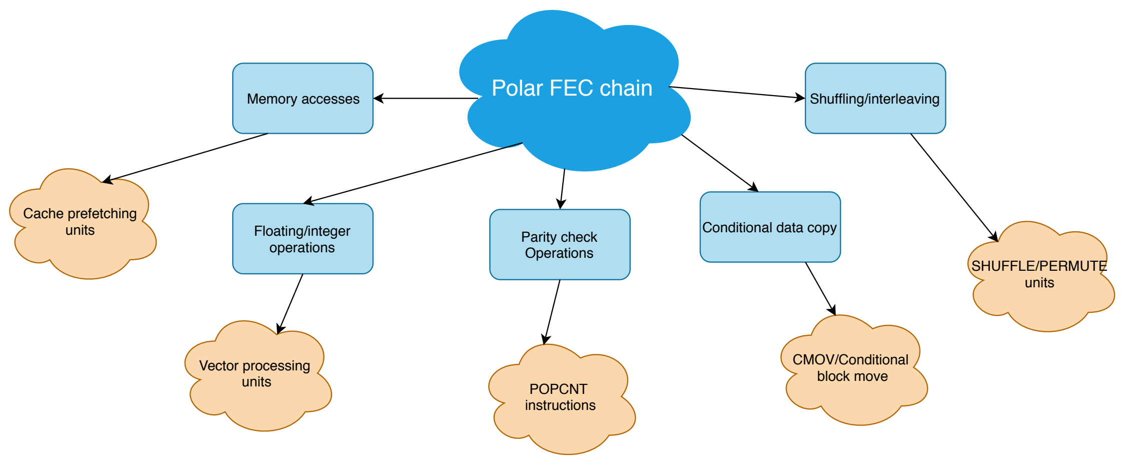

Figure 1.1.: Mapping FEC chain operations to particular functional units

operators.

It is clear from above-discussed details that algorithms directly implemented in software without modification or reformulation result in poor performance. To achieve the best performance in software algorithms must be broken down into smaller operations mapped to specific functional units of modern processors. For example, a huge number of floating point/integer operations can be mapped to specialized vector processing units (SIMD) which are specifically designed to perform these operations. Mapping to vector processing units allows data parallelism. If an algorithm requires a lot of memory accesses, software should make use of special cache prefetching units to hide memory access latency. If a huge amount of data must be moved/copied based on previous decisions, such operations must be mapped to conditional move instructions and so forth. Figure 1.1 gives visual illustration of mapping operations to specialized processing units/instructions.

This thesis work is a step in that direction. It tries to achieve maximum performance from modern general purpose processors to satisfy stringent latency requirements. The latest 3GPP standard has adopted polar codes for encoding and decoding control channel information [4]. Downlink control information (DCI) and Uplink control information (UCI) is transmitted using polar codes. In this work, polar encoding and decoding FEC chain are implemented in software. FEC chain algorithms are reformulated/optimized to suit a software implementation. FEC chain functionalities are broken down into primitive operations and mapped to specific computational units of general purpose processor. In this work, FEC chain implementation runs on a single processor core. The current implementation focuses on the off-the-shelf available AMD EPYC 7551P 32-Core general

purpose processor[7]. It comes with state-of-the-art instruction extensions namely `SSE`, `AVX`, `AVX2` `3DNow!` et cetera. These advanced instructions allow mapping of different operations in FEC chain implementation to specialized processing units to maximize performance.
Software implementation in this work makes extensive use of vector processing units, cache prefetching, branch prediction and compiler optimizations to achieve low latency. After the implementation, each component of the FEC chain performance of naive and optimized implementations are presented. At the end of encoding and decoding FEC chain chapters, overall latency of FEC chain is presented. Comparison between functional and optimized implementation is given to understand the importance of software optimizations. In decoding FEC chain chapter performance of implemented decoder is compared with state-of-the-art software decoder implementation[3].

Organization of the Thesis

Having described the overall problem and relevant motivation, in Chapter 2 the necessary background to develop polar FEC chain is provided. Including the required mathematical understanding of polar codes and fundamentals of modern computer architecture. In Chapter 3, Necessary details about different physical channels which use polar codes are presented. In Chapter 4, details of the polar encoding FEC chain are presented. Each component is analyzed for latency contribution. Both algorithmic and software optimizations are employed. In the end, the latency of naive and optimized implementations are compared. In Chapter 5, the details of polar decoding FEC chain are presented. Operation of different components in the FEC chain is given in detail. Each operation is optimized by mapping them efficiently to specific instructions/processing units. Decoder algorithm optimizations are presented and compared with state of the art software decoder implementations. Finally, at end of Chapter 5, the latency of the optimized decoding FEC chain is compared with the naive implementation.

2. Background

Polar codes were introduced by Arıkan in his seminal work [8]. They belong to the class of capacity achieving codes. In the past decade, polar codes have sparked a interest from both academia and industry alike, resulting in significant research work in improving performance. The 5$^{\text{th}}$ generation wireless systems (5G) standardization has adopted polar codes for uplink and downlink control information for the enhanced mobile broadband (eMBB). They are also considered as the potential coding schemes for two other frameworks of 5G, namely ultra-reliable-low-latency (URLLC) and massive machine-type communications (mMTC).
Polar codes achieve capacity asymptotically for binary input memoryless channel. Although they are the first theoretically capacity achieving codes with an explicit construction, capacity is approached only asymptotically. Their performance is suboptimal compared to LDPC (Low Density Parity Check Codes) or Turbo codes at short block lengths with *successive cancellation decoding (SCD).* In [9] the authors present an improved version of SCD called *successive cancellation list decoder (SCLD).*
The construction of polar codes involves the identification of channel reliability values. Information bits are placed in the K (number of information bits) high reliable bit indices out of N (block-length) positions and remaining bits are set to zero. These N bits are passed through a polar encoding circuit to get the encoded bits. Selection is of reliability indices is done based on the code length and channel signal-to-noise ratio. Due to varying code length and channel conditions in 5G systems, a significant effort has been put into identifying the reliable indices which have good error correction performance over different code length and channel conditions.

2.1. Background of Polar codes

This section introduces the foundations of the polar codes. In particular, about the frozen set design, encoding, and decoding. Different decoding algorithms are introduced. Mainly Successive Cancellation (SC) and Improved Successive Cancellation (Fast-SSC)[10]. Examples of encoding and decoding with different algorithms are presented for a better understanding.

The mathematical foundations of polar codes lies in the polarization effect of the kernel [8]. $\underset{\sim}{G} = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$ also called Arıkan matrix. Polar codes are (N, K) linear block codes of size $N = 2^n$ where n being a natural number. N is the block length of the code and K is the number of information bits. N-bit vector U contains K information and $N - K$ frozen bits which are set to known value mostly zeros. These bits are then multiplied with the generator matrix constructed from kronecker power[11] of Arikan kernel matrix. For example $n = 3$, block-length N becomes 8 hence the generator matrix is

$$\underset{\sim}{G}^{\otimes 3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

where $\underset{\sim}{G}^{\otimes n}$ denotes the n^{th} Kronecker power of $\underset{\sim}{G}$. The encoding process involves the multiplication of N-bit vector U consisting of K information bits and $N - K$ frozen bits with $\underset{\sim}{G}^{\otimes n}$.

2.1.1. Polar code construction

In polar coding, the first step is to identify the channel reliability values for a particular block length, this step is also called polar code construction. Basic idea is to produce a fraction of channels which are either completely noiseless or noisy out of N (block-length) independent copies of the given binary discrete memoryless channel. This process of creating extremal channels is called channel polarization. As $N \to \infty$, the fraction of noiseless channels approaches the capacity of the channel. Estimating reliability indices of channels is carried by considering the Bhattacharyya parameter [8]. Bhattacharyya parameter indicates the reliability of the individual channel.

For a generic binary-input discrete memoryless channel (B-DMC) which is represented as $W : \mathcal{X} \to \mathcal{Y}$ with input alphabet $\mathcal{X}$, output alphabet $\mathcal{Y}$ and transition probabilities given by $W(y|x), x \in \mathcal{X}, y \in \mathcal{Y}$.

Bhattacharyya parameter is given by

$$Z(W) \triangleq \sum_{y \in \mathcal{Y}} \sqrt{W(y|0)W(y|1)} \tag{2.1}$$

The Bhattacharyya parameter indicates how unreliable the channel is, It is easy see that $Z(W)$ takes values between $[0, 1]$ better the channel smaller is the $Z(W)$. Polarization for $N \to \infty$ creates channels with either $Z(W) \to 0$ or $Z(W) \to 1$.

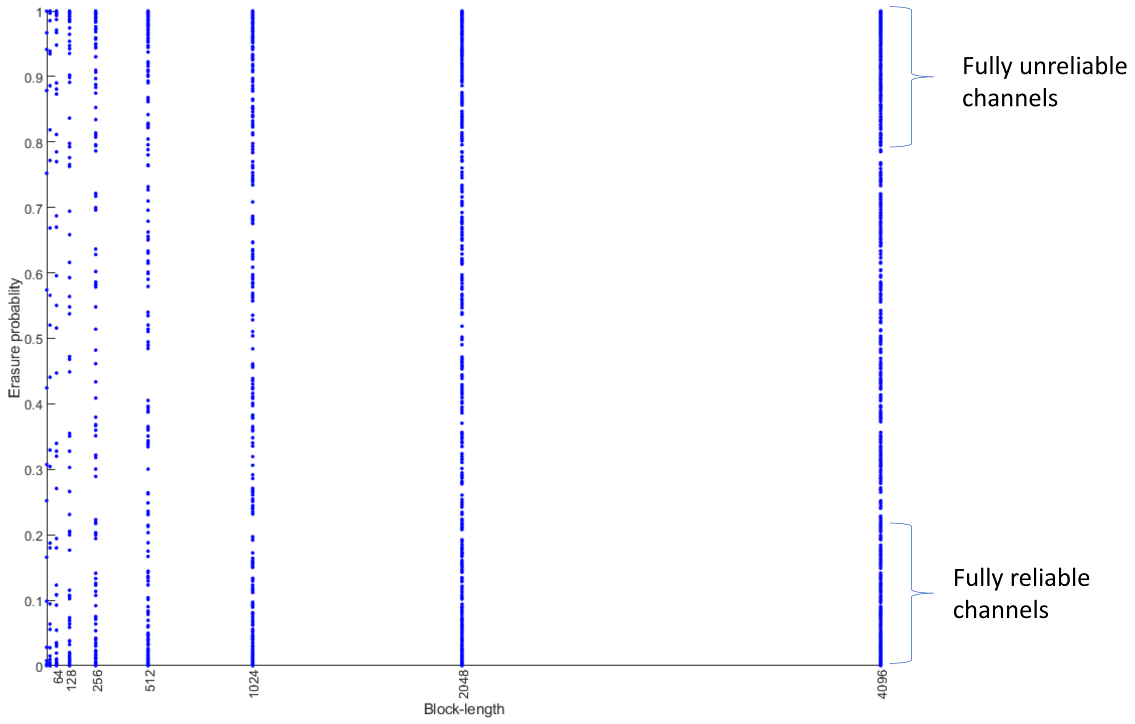

Figure 2.1.: Channel polarization example for binary erasure channel with $\epsilon = 0.4$

Figure 2.1 illustrates channel polarization for different block lengths for binary erasure channel with erasure probability $\epsilon = 0.4$. It can be seen that as block-length increases channels gets polarized to extremal channels (either completely reliable or unreliable).

2.1.2. Encoding

After $N-K$ frozen bit positions have been found, they are set to zero and information bits are placed in remaining K positions. This N-bit vector U is multiplied with generator matrix obtained by the Kronecker power of Ar ıkan kernel matrix. Multiplying with generator matrix can also be represented as circuit form. Arıkan kernel matrix can also be represented in a circuit form as shown in Figure 2.2 also called butterfly circuit.
For an $n = 3$ told Kronecker product, block length N becomes 8 for such a case encoding circuit looks shown in Figure 2.3, which is a repeated application of the butterfly circuit. The read locations are the frozen bit indices which are set to zero, in remaining positions information bits are inserted. The output of the circuit is a code word which is transmit-

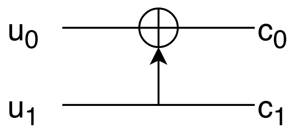

Figure 2.2.: Butterfly circuit representing Arıkan Kernel matrix

ted over the channel. Lets consider an example with $N = 8$ and $K = 4$, rate of this code is $R = K/N = 1/2$. As given in the figure frozen bit indices are $\{0, 1, 2, 4\}$ remaining indices contain information bits. Let the information bits, which needs to transmitted be {1,1,0,0}. After placing information bits at reliable channel positions the vector U becomes {0,0,0,1,0,1,0,0}. It is passed through the polar encoding circuit shown in Figure 2.3. Result at the output of encoder is {0,0,1,1,1,1,0,0}. These encoded bits are then transmitted over the channel.

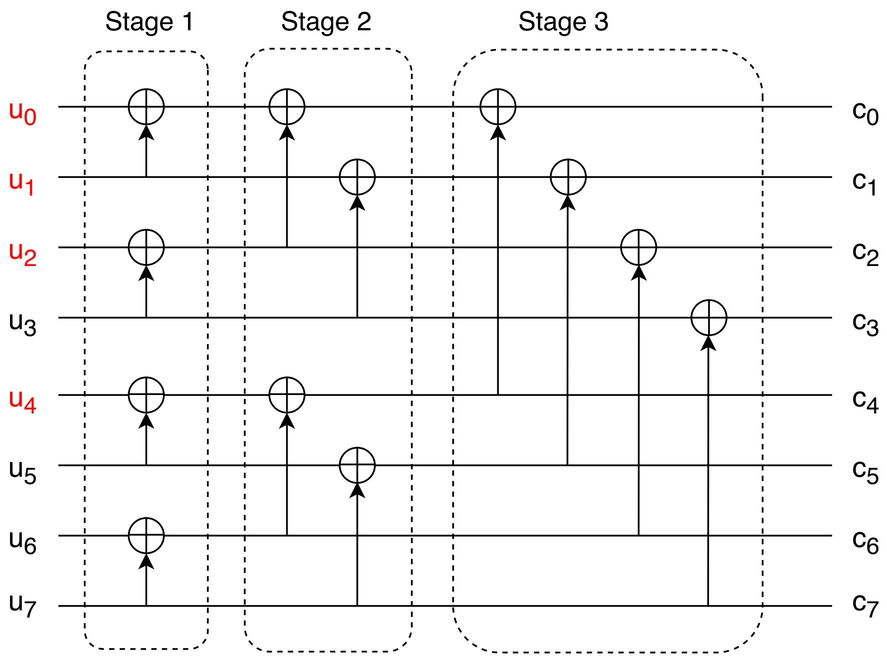

Figure 2.3.: Polar encoder in circuit form for $N = 8$

The encoding circuit is nothing but the recursive application of the transformation represented by the butterfly circuit shown in the figure 2.2. One butterfly unit can transform two uncorrelated bits (a, b) into two correlated output bits $(a \oplus b, b)$. This corresponds to a polarization into two channels. In the above example, the reliability of the u_1 is increased compared to the u_0. This operation recursively applied to the whole code word results in the circuit shown in Figure 2.3. Code word splits into two parts in stage-3, which again splits into two parts in stage-2 and so on, until one reaches to single source bit u_i in stage-1. So the process of polar encoding for $N = 8$ involves three stages of butterfly operations. Generally, for a given code length $N = 2^n$, the polar encoding consists of stages each with $N/2$ butterfly operations, which results in an encoding complexity of $O(N \log(N))$.

2.1.3. Decoding

As shown in section 2.1.2, repetitive application of butterfly operation during encoding introduces correlation between the source bits. At the receiver, this is exploited to estimate the transmitted codeword. Utilizing the high correlation between the source bits forms the central idea behind the basic polar decoding algorithm called *successive cancellation (SC)* decoding. This method sequentially decodes each of the bits and takes previously estimated value to account for estimating the next bit. This sequential decoding exploits the correlation between the source bits which as introduced during the polar encoding process. Due to the sequential nature of SC decoding, the decoding process has high latency. Improving the basic decoding algorithm has been the topic of researchers in academia and industry. These improvements are mainly directed towards two goals, first reducing decoding latency and second improving the error correction performance. Significant reduction in decoding latency is achieved by [12] and [13]. In these works, instead of decoding sequentially individual bit, special nodes are identified which can be decoded in parallel. Although polar codes are the first theoretically capacity achieving codes with explicit construction, their error correction performance at short block lengths is not comparable with that of LDPC or Turbo codes. This behavior can be better explained by the way decoding is performed. As presented earlier SC algorithm works by sequentially decoding individual bits and using information of previously decoded bits for estimating next bit. The issue with the algorithm is if the previously decoded bit is wrong, then there is no way of correcting this bit.

To overcome this problem [9] presents an improved version of the SC algorithm called *successive cancellation list* (SCL) Decoding. The basic idea is instead of deciding a value of bit u_i it takes both options, this results in two decoding paths for every bit, so to avoid exponential growth of complexity decoding candidates are restricted to L the list size. At the end of decoding, the most probable candidate is chosen from the list. Performance of polar codes with SCL is still not as good as LDPC or Turbo codes at small and moderate block lengths. Polar code concatenated with CRC as outer code beats the LDPC codes of similar block length [9]. SCL algorithm has a better error correction performance than SC, however, it comes with a increased complexity and high decoding latency. Due to these reasons, in this work concentrates on the Fast-SSC algorithm is considered although its error correction performance is inferior compared to SCL. Naive SCL algorithm is implemented, however, its optimization is considered for the future work.

A. Successive Cancellation Decoding (SC) The recursive SC decoder is basic algorithm presented in [8] for decoding polar codes. SC decoder is inherently sequential, the estimate of u_i, $\hat{u}_i$ is obtained by using channel observation y_1^N and all the previously decoded bits $\hat{u}_1^{i-1}$. If u_i is frozen bit, the decoder assigns $\hat{u}_i$ to known value (mostly zero). If u_i is an information bit, the decoder waits for all the previous bits to compute the decoding metric. In the following equations $W_N^{(i)}$ is a set of N binary-input synthesized channels with output $(y_1^N, \hat{u_1}^{i-1})$ and u_i as input, also represented as $W_N^{(i)} : \mathcal{X} \rightarrow \mathcal{Y^N} \times \mathcal{X}^{i-1}$, $1 \leq i \leq N$. These channels are synthesized from generic binary-input discrete memoryless channel also written as $W : \mathcal{X} \rightarrow \mathcal{Y}$ with input alphabet $\mathcal{X}$, input alphabet $\mathcal{Y}$. Synthesized through channel polarization as presented in [8].

Decoding metric can be one of the three different types of metrics shown below:
• log-likelihood ratio (LLR) where

$$L_N^{(i)}(y_1^N, \hat{u_1}^{i-1}) = \ln\left(\frac{W_N^{(i)}(y_1^N, \hat{u_1}^{i-1} | u_i = 0)}{W_N^{(i)}(y_1^N, \widehat{u_1^{i-1}} | u_i = 1)}\right); \tag{2.2}$$

• likelihood ratio (LR) where

$$LR_N^{(i)}(y_1^N, \hat{u_1}^{i-1}) = \left(\frac{W_N^{(i)}(y_1^N, \hat{u_1}^{i-1} | u_i = 0)}{W_N^{(i)}(y_1^N, \widehat{u_1^{i-1}} | u_i = 1)}\right); \tag{2.3}$$

• log-likelihood (LL) where

$$LL(y_1^N, \hat{u_1}^{i-1}) = \left[\ln\left(W_N^{(i)}(y_1^N, \hat{u_1}^{i-1} | u_i = 0)\right), \ln\left(W_N^{(i)}(y_1^N, \widehat{u_1^{i-1}} | u_i = 1)\right)\right]; \tag{2.4}$$

Decoding metrics computed from LLRs exhibit better numerical stability than those from LRs or LLs, so we have used the LLRs metric throughout this work. There are different ways to view and understand the operation of an SC decoder. In this work, decoding is viewed as a message passing algorithm on a binary tree with $\log(N)$ levels. Decoding is performed by traversing a tree from root to leaf node. The process of decoding involves check node (CN), variable node (VN) operations and threshold detection at the leaf node. The decoder receives an LLR value for every bit which needs to be decoded (including both frozen and information bits), hence for a code with block length N, SC decoder receives N LLR values. Decoding process estimates the bits $\hat{u}_i$ where $i = 1, 2..., N$. The decoding tree for $N = 8$ looks as shown in the Figure 2.4.

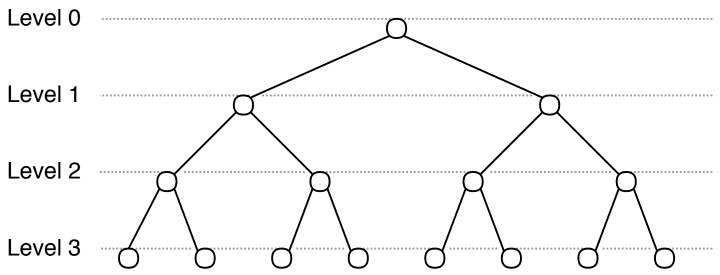

Figure 2.4.: Decoding tree

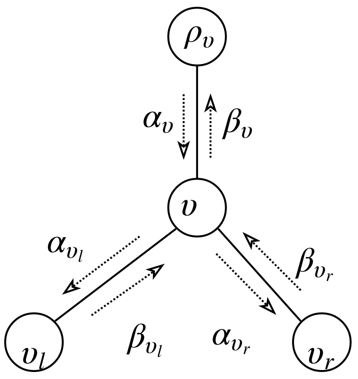

Figure 2.5.: Local Decoder

In a decoding tree, the messages to left child node are computed with CN and to the right are with VN operation. Figure 2.5 shows how the messages are exchanged in a local component decoder.

The CN and VN operations in LLR domain are given by the following equations:

• Check Node (CN) operation

$$\alpha_{v_l}[i] = \alpha_v[i] + \alpha_v[i + N_v/2] \tag{2.5}$$

• Variable Node (VN) operation

$$\alpha_{v_r}[i] = \alpha_v[i + N_v/2] + (1 - 2\beta_{v_l}[i]) * \alpha_v[i] \tag{2.6}$$

After decoding is done at both right and left child nodes the bits are combined at common parent node. the bit combining operation is given by the following equation.

$$\beta_v[i] = \begin{cases} \beta_{v_l}[i] \oplus \beta_{v_r}[i] & \text{if } i < N_v/2 \\ \beta_{v_r}[i] \end{cases}$$

In the Figure 2.5 and in equations (2.5), (2.6) α_v, β_v represent intermediate LLR values and estimated bits at the local decoder respectively.
Figure 2.6 gives an example of SC decoding for the block-length $N = 8$ and a number of information bits $K = 4$. The 16-bit quantized LLR values are an input to the decoder. In the figure, decoded bits and intermediate β_v are represented by black font color and computed intermediate α_v are indicated by green. The frozen pattern is provided below the leaf nodes. The frozen pattern indicates the position of information and frozen bits. One in frozen pattern indicates frozen bit, zero indicates information bit.

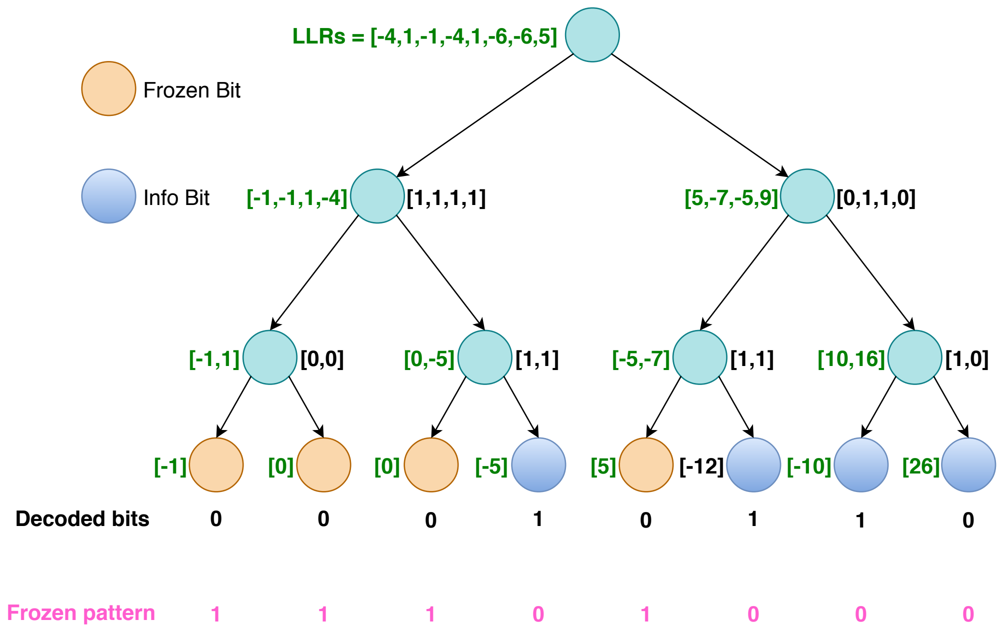

Figure 2.6.: SC decoding example

B. Improved Successive Cancellation Decoding (fast-SSC)

In the basic SC algorithm, decoding is performed sequentially, previously decoded values are used for decoding the present bit. Due to the sequential nature of the decoder, decoding latency is high. The works [12] and [13] try to identify special kind of nodes in a decoder tree which can be immediately decoded without traversing till the end of the tree. In [12], the authors try to identify the nodes which are associated with all information or frozen bits. These nodes are called rate-one ($R1$) and rate-zero ($R0$) nodes respectively. The $R1$ node can be decoded by taking hard decisions and a polar transform since there is no extra information which can be gained from traversing the tree. The decoding of $R0$ nodes is not necessary since none of them are information bits, so all the bits are set known value which is known at transmitter and receiver which are mostly set to zero.

Authors in [13] extend the idea presented in [12] by identifying two additional kinds of special nodes which can be decoded without traversing the tree single parity check (SPC) and repetition (REP) nodes. Both in [12] and [13], the node type is identified based on the frozen pattern at the component decoder. For an SPC node, only one frozen bit is present at the leftmost position. For a REP node, the frozen pattern contains one information bit at the rightmost position, remaining are frozen bits.
One such example, when frozen indices for $N = 8$ are $\{0, 1, 3, 4\}$. The full decoding tree of Figure 2.4 gets reduced to a tree with fewer nodes as shown in Figure 2.7. We can easily see that in the original decoder tree number of nodes were 15, in the pruned tree nodes are reduced to 7, which results in a significant reduction in the number of computations and decoding latency.

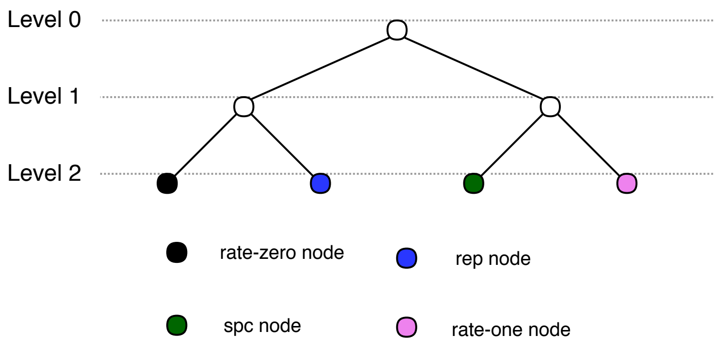

Figure 2.7.: Pruned Decoder Tree

2.2. Processor Architecture Background

To better understand the bottlenecks and optimizations performed in the software implementation of 5G FEC chain, it is necessary to understand the fundamentals of general purpose processors architecture. This section gives necessary background about cache memory systems, instruction pipelining, branch predictors, vector processing units and recursive function calling mechanism.

2.2.1. Cache memory

In the modern processors, a fast memory called cache is used to reduce the average access time of main memory also called RAM (Random Access Memory). The cache minimizes the number of accesses to RAM by storing frequently accessed data in it, hence avoiding huge penalty of reading data frequently from RAM which operates at a much lower frequency than the CPU. When a memory location is accessed for the first time it is copied from the RAM to the cache, future accesses to the same location is done via cache. This fast memory is placed between RAM and processor. In modern processors instead of single cache, multi-level caches are present. The main idea behind having multi-level caches is that if the data is not found in the first level then second level is checked if not then the third level until the last level, still, if the data is not found then RAM is accessed. This model significantly reduces the probability of accessing the RAM compared to having a single level cache. Complete memory hierarchy of the modern processors is shown in Figure 2.8 [14].

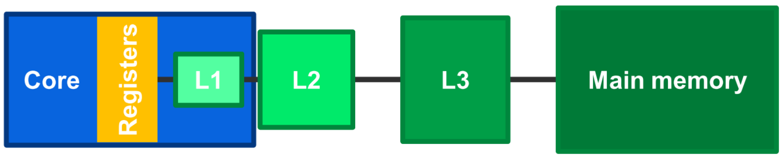

Figure 2.8.: Memory Hierarchy

Above figure shows processor architecture with three level caches namely L1, L2, and L3. In the order of increasing access latency, reducing cost and increasing size. L1 cache is fastest, costliest and smallest among all caches. Data is mapped to either memory or registers. If the available registers are not enough in such a case data is stored in memory. If the data is not found in all cache levels then it results in *cache-miss* which causes processor instruction execution to stop until data is fetched from RAM. Whenever the memory location is accessed for the first time it always results in *cache-miss*. Modern processors provide special instructions to avoid these compulsory cache misses, these are called cache prefetch instructions which allow a programmer to fetch data from the cache before it is accessed, hence hiding the memory access latency. Some other software techniques to reduce cache misses are reusing the allocated memory as much as possible and bit packing/unpacking to reduce the required memory. In this work, all of the above-mentioned techniques namely using prefetch instructions (`prefetch`) provided by AMD EPYC processor, reusing the allocated memory and bit packing/unpacking are used reduce the memory access latency. AMD EPYC processor used in this work has 3MB L1 cache, 16MB L2 cache, and 64MB L3 cache.

2.2.2. Instruction pipelining and branch predictors

Traditionally processors were designed to follow the steps fetch, decode, execute, memory finally write-back and then fetch the next instruction. Although these steps are sufficient to solve any problem at hand, it is very inefficient in terms of hardware utilization. In instruction fetch phase, all modules except fetch module are idle. Similarly, during the other phases module processing the current phase of an instruction is active remaining modules are idle. To overcome underutilization of hardware resources modern processors implement instruction pipelining concept, where if the current instruction is in decoding phase the next instruction will be concurrently fetched by the fetch module. A pipelining mechanism increases the instruction throughput by significantly reducing Cycles per Instruction (CPI). Example of sequential and pipelined execution is shown in figure 2.9 [15].

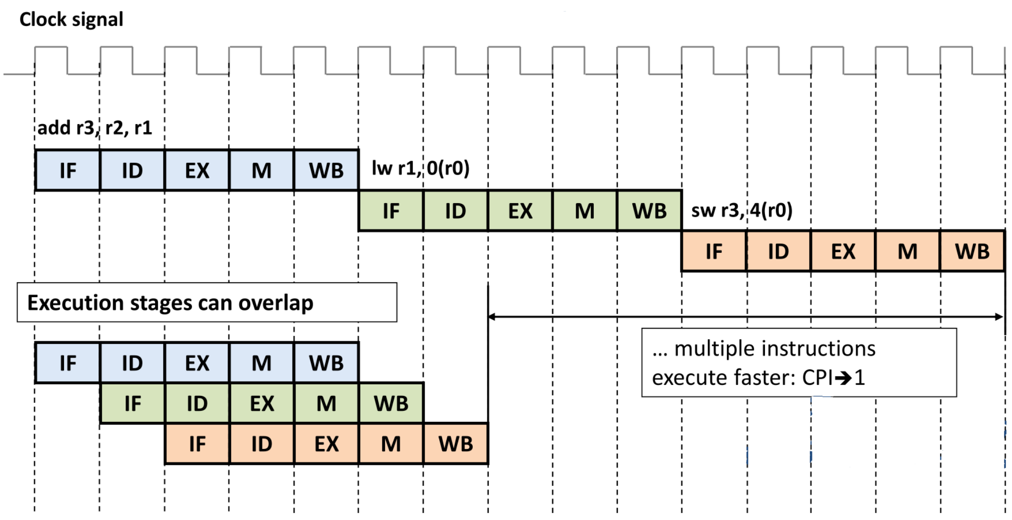
Clock signal
add r3, r2, r1
IF
ID
EX
M
WB
lw r1, 0(r0)
sw r3, 4(r0)
Execution stages can overlap
... multiple instructions execute faster: CPI→1

Figure 2.9.: Instruction pipelining

The example shown in Figure 2.9 assumes only five phases of instruction execution. Modern processors divide instructions execution to nineteen plus phases, which allows running processor at much higher frequency due to reduced critical path delay. Maximum advantage of pipelining can only be exploited when there are no pipeline stalling or flushing which happen when there is a data dependency, cache misses or branch instructions. Major contributors to pipeline stalling are cache misses and branch instructions. As explained previous section, *cache-miss* can be reduced by using a combination of different optimization techniques. Next culprit is branch instructions, whether to branch or not is decided only the at execution stage. By the time branching is decided, many of the future instructions are already fetched if the decision is to jump then all the prefetched instructions must be flushed which introduces stall in the pipeline. To overcome this issue branch predictor are designed to pro-actively fetch instructions from correct address, hence avoiding flushing of the pipeline. Branch predictors function by storing the previous decisions on the branching whether it was taken or not, hence requires the correct previous state to proactively fetch future instructions. This method reduces the pipeline caused due to looping type of code, by reducing pipeline flushes. For the scenarios where there are no looping instruction just if or if-else constructs branch predictors fail to correctly fetch the future instructions. These kinds of scenarios can be minimized by avoiding branch instructions wherever possible and by providing hints to compiler built-in macros to reduce the branching by better placement of assembly instructions (kind of instruction scheduling). One such macro is

```
long __builtin_expect(long EXP, long C);
```

which tells the compiler to generate code in such a way that the code which is more frequently executed is just after the branch instruction to minimize pipeline flushing. Code snippet 2.1 shows the typical usage.

Listing 2.1.: Branching hints to compiler

```
#define likely(expr) __builtin_expect(!!(expr), 1)

if (likely(a > 1)) {
          //Frequently executing part, most of the cases a > 1
        ...
} else {
        //Rarely executing part, rarely a < 1
        ...
}
```

Another feature provided by modern processors is conditional move instruction (`CMOV`). The compiler intelligently maps “if” statements to conditional move instructions. `CMOV` copies a particular value to a register or memory based on the flags set. It is not vulnerable to branch-prediction failure since no branch instructions are generated, hence avoiding pipeline flushing. Listing 2.2 and 2.3 illustrates the feature.

Listing 2.2.: Naive method

```
uint32_t bitMask = 0;

if(CRCLENGTH == 6) //If Crc6 needs be calculated, then change the mask.
  bitMask = 0x3F;
else
  bitMask = 0x7FF;
```

Listing 2.3.: With `CMOV`

```
uint32_t bitMask = 0x7FF; //Initializing with CRC11 mask.
if(CRCLENGTH == 6) //If Crc6 needs be calculated, then change the mask.
  bitMask = 0x3F;
```

Both codes achieve same results, however in Listing 2.2 a pipeline flush will happen due to branch-misprediction, whereas for Listing 2.3 compiler identifies conditional move construct and generates `CMOV` potentially avoiding pipeline flush. Optimizations such as minimizing branches, using built-in macros and using constructs which help the compiler to identify pattern are utilized in this work.

2.2.3. Vector Processing Units

Vector processing units are special kinds of multiple computational elements that perform the same operation on multiple data points simultaneously. Machines with vector processing units exploit data level parallelism but not concurrency. Same instruction operates on multiple data points, in other words, there are simultaneous computations but there is a single process. These special kinds of instructions are also called SIMD (Single Instruction Multiple Data) in Flynn's taxonomy of parallel computers [16]. These instructions are particularly useful when the same operation needs to be applied to the set of data, for example scaling a vector by constant. SIMD units can be thought of same processing unit replicated multiple times which are operated by a single instruction. The Figure 2.10 illustrates the concept of SIMD and how single instruction pool operates on multiple data points.

Modern x86 processors from AMD and Intel provide different SIMD extensions named Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX) with register size of 128-bits, Advanced Vector Extensions 2 (AVX2) with register size 256-bits and the latest AVX-512 with register size 512-bits. In this work, an AMD EPYC processor is used provides SSE, AVX, and AVX2 instructions. Listing 2.4 shows vector addition implemented naively. Listing 2.5 illustrates same operation using vector processing units.

Listing 2.4.: Naive vector addition

```
        int16_t vec1[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
        int16_t vec2[] = {17,18,19,20,21,22,23,24,25,26,27,28,29,30
                                                        ,31,32};
        for(auto i = 0; i < 16; i++)
                vec1[i] = vec1[i] + vec2[i];
```

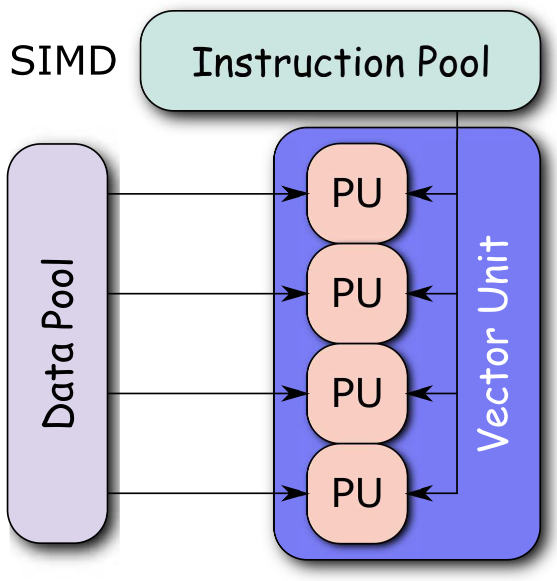

Figure 2.10.: Vector processing units [1]

Listing 2.5.: SIMD Addition

```
int16_t vec1[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int16_t vec2[] = {17,18,19,20,21,22,23,24,25,26,27,28,29,30
                                   ,31,32};

__m256i reg1 = _mm256_load_si256 ((__m256i*)vec1);
__m256i reg2 = _mm256_load_si256 ((__m256i*)vec2);
reg2 = _mm256_add_epi16 (reg1, reg2);
_mm256_store_si256 ((__m256i*) vec1, reg1);
```

2.2.4. Recursive function calling mechanism

Most of the encoder and decoder implementations are implemented through recursion. The recursive function is a function which calls itself. It is a powerful tool in computer science which can be used for solving many interesting problems [17]. However when it comes to performance, recursive problems solving fall behind algorithms which use loops to solve the same problem. Every time a recursive function is called, a new stack frame is allocated, local data is pushed to call stack and execution must branch to the beginning of the function. Branching and pushing data to a call stack are expensive operations. In the case of polar codes, both encoding and decoding are implemented as recursive functions. To reduce the latency of FEC chain encoding/decoding implementations are unrolled to avoid recursions. For the encoder, unrolling is carried out by manually implementing multiple inline functions. In decoder implementation unrolling is carried out by using template concept of C++. Although unrolling increases code size, for the application in hand latency is of at most importance than code size. Unrolling of the encoder and decoder implementations significantly improved the latency of the FEC chain. Listing 2.6 shows recursive implementation of factorial calculation. Listing 2.7 illustrates same operation implemented with unrolled implementation.

Listing 2.6.: Recursive implementation

```
int fact(int n) {
        if(n > 1)
        return n * factorial(n - 1);
        else
        return 1;
}
```

Listing 2.7.: Unrolled implementation

```
template <unsigned Nv>
inline int fact() {
        return Nv * fact<Nv - 1>();
}

template <>
inline int fact<0>() {
        return 1;
}
```

3. Polar Codes in 5G

Polar codes are a class of capacity-achieving codes introduced by Arıkan in 2009 [8]. 5^{th} generation wireless systems (5G) has adopted polar codes as the channel coding candidates for uplink and downlink control information for the enhanced mobile broadband (eMBB) [4]. This chapter explains the details about different types of 5G physical channels which use polar codes for channel coding, the purpose of these physical channels and used modulation formats. This chapter also presents the generic encoding FEC chain of the polar codes and explains the details about FEC chain parameters such as adopted polar code block-length sizes, Cyclic Redundancy Check (CRC) type, size.
Generic polar FEC chain is shown in Figure 3.1. This FEC chain is configured with the parameters of a particular physical channel as specified in [4] and [18].

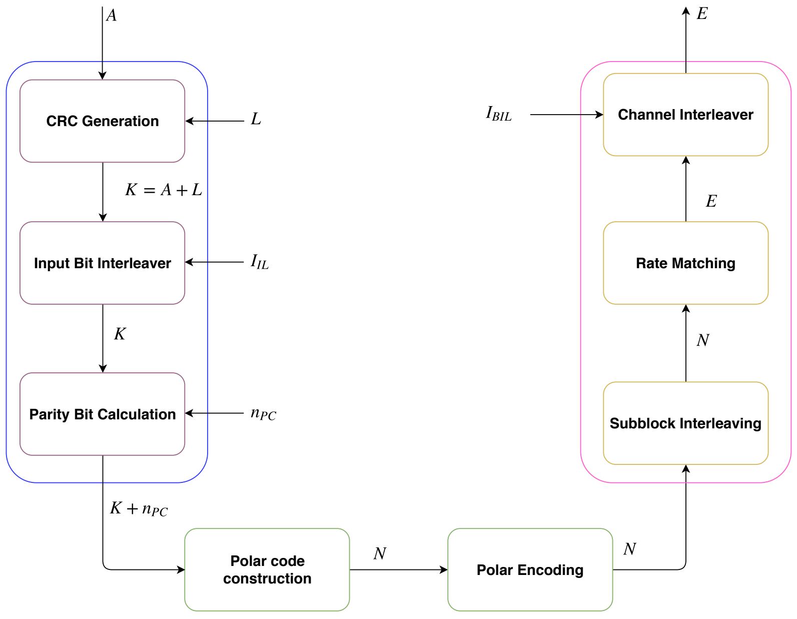

Figure 3.1.: Generic Encoding FEC of 5G

The polar FEC chain receives A information bits from the upper layer. These bits need to be transmitted through a code of length E bits. Size of A depends on the physical channel and size of uplink or downlink control information. Value of E depends on the scheduling and resource allocation parameters and it is configured from higher layers. After receiving

A information bits, a L-bit CRC is attached to the information bits resulting in $K = A+L$ bits. The value of L is determined based on the physical channel, size of A and E. After attaching CRC bits, K bits are interleaved by input bits interleaver. The input bits interleaver is configured with the parameter I_{IL}. I_{IL} can take either 0 or 1. This module is enabled when $I_{IL} = 1$ otherwise input bits interleaver is disabled. Next component in the FEC chain is Parity Check (PC) bits calculation. The number of PC bits are configured with the parameter n_{PC}. n_{PC} value is selected based on the physical channel, size of A and E. Another parameter of PC component is n_{pc}^{wm} which indicates how many parity bits need to be placed in the rows of minimum hamming weight in the polar code generator matrix [4]. After parity bits calculation, the polar code construction component identifies $K + n_{PC}$ reliable indices for placing $A + L$ bits and n_{PC} positions for parity bits. The parameter $N = 2^n$ is the block-length of polar code. Value of n is calculated based on the values of E, K and n_{max}. where n_{max} is configured based the physical channel. In 5G polar FEC chain, five different block-lengths are supported as given by $\mathcal{N} = 32, 64, 128, 256, 512, 1024$. The encoded bits go through sub-block interleaving and rate matching to obtain E bits from N bit codeword. Next configurable parameter is I_{BIL}, it configures the channel interleaver. It can take either 0 or 1. $I_{BIL} = 1$ enables channel interleaver otherwise it is disabled. Details of the different physical channels and their FEC chain parameters are presented in the following sections.

3.1. 5G Physical Channels

This section presents details of the 5G physical channels which use polar codes. The 5G standard adopted polar codes for uplink and downlink control channels. Uplink control channels carry information about channel quality indicators, acknowledgments. In downlink control channels carry resource allocation information, uplink power control instructions and the information required for the user equipment (UE) to access the network. Following sections explain each of these uplink and downlink control channels and their polar FEC chain parameters.

3.1.1. Physical Broadcast Channel (PBCH)

In downlink, polar coding is applied to PBCH which carries the essential information required for the UE to access the network. PBCH carries network information such as system bandwidth, current system frame sequence. The polar FEC chain parameters of PBCH are fixed. In other words, payload size (A) of PBCH is always 56 bits. Other fixed parameters of PBCH are $E = 846$, $L = 24$, $n_{max} = 9$, $I_{IL} = 1$, $I_{BIL} = 0$, and

$n_{PC} = n_{pc}^{wm} = 0$. Modulation format used for PBCH is always QPSK. PBCH is explained in more detail in [4].

3.1.2. Physical Downlink Control Channel (PDCCH)

The PDCCH is another downlink control channel which uses polar codes. Resources requested by the UE are assigned by the base station. This resource allocation information is transmitted via PDCCH channel. PDCCH also carries information related to uplink power control, downlink resource grant and system paging information [18]. The PDCCH contains a message called Downlink Control Information (DCI) which carries all the control information of UE. Payload size of PDCCH is not fixed. It varies based on the format of DCI, As a consequence, values of A, N and E vary. Type of DCI is configured from the higher layer. Except A, N and E, other parameters of the PDCCH polar FEC chain are same as PBCH. Complete details about different DCI formats in PDCCH are presented in Section 7.3 of [4].

3.1.3. Physical Uplink Control Channel (PUCCH)

In uplink, PUCCH contains Uplink Control Information (UCI) similar to DCI in the downlink. UCI carries channel state information, acknowledgments, scheduling request. The payload size of PUCCH varies based on the PUCCH formats [18]. PUCCH uses different channel coding techniques depending on payload size. When payload size $A \geq 12$ polar codes are used. PUCCH polar FEC chain parameters also vary depending on the values of A and E.

• There are three different cases for PUCCH polar FEC chain parameters based on the values of A and E as presented below.

- *Case 1.* $A \geq 20$
  $L = 11$, $n_{max} = 10$, $I_{IL} = 0$, $I_{BIL} = 1$, and $n_{PC} = n_{pc}^{wm} = 0$.

- *Case 2.* $12 \leq A \leq 19$ *and* $E - A \leq 175$
  $L = 6$, $n_{max} = 10$, $I_{IL} = 0$, $I_{BIL} = 1$, and $n_{PC} = 3$, $n_{pc}^{wm} = 0$.

- *Case 3.* $12 \leq A \leq 19$ *and* $E - A \geq 175$
  $L = 6$, $n_{max} = 10$, $I_{IL} = 0$, $I_{BIL} = 1$, and $n_{PC} = 3$, $n_{pc}^{wm} = 1$.

PUCCH uses $\pi/2$-BPSK or QPSK depending on the PUCCH format [18].

3.1.4. Physical Uplink Shared Channel (PUSCH)

PUSCH is another uplink channel which transmits UCI. In PUSCH, LDPC codes are used for encoding user data and polar codes for control information. The UE transmits UCI in PUCCH, if UE is not transmitting any user data to the base station. When UE is transmitting user data through PUSCH, UCI is also transmitted with PUSCH using the same modulation parameters. In other words, if PUSCH is using 64-QAM then same modulation technique is applied to UCI. Allowed modulation formats for PUSCH are QPSK, 16-QAM, 64-QAM, and 256-QAM as presented in Section 6.3.1.2 of [18].

The parameters of the polar FEC chain for different physical channels of 5G are summarized in Table 3.1.

Table 3.1.: Channel parameters [2]

	PUCCH/PUSCH			PDCCH/PBCH
	$A \geq 20$	$12 \leq A \leq 19$		
		$E - A \leq 175$	$E - A \geq 175$	
n_{max}	10			9
I_{IL}	0			1
I_{BIL}	1			0
L	11	6		24
n_{PC}	0	3		0
n_{pc}^{wm}	0	0	1	0

4. Encoding FEC Chain

In the 5G standard, polar codes are used in the downlink to encode downlink control information (DCI) over physical downlink control channel (PDCCH) and for Master Information Block (MIB) in the physical broadcast channel (PBCH). In the uplink, to encode uplink control information (UCI) over the physical uplink control channel (PUCCH) and physical uplink shared channel (PUSCH). In this work, notations introduced in 3GPP technical specification [4] are used.

This chapter presents the details of the polar encoding FEC chain in 5G with a block diagram. Future sections will explain the functionality and potential latency contribution of individual components in the FEC chain. Each of these individual components is extensively profiled to identify expensive operations and latency contribution. After identifying the bottlenecks, both algorithmic and software optimization techniques are employed. Algorithm optimizations include reformulation of the problem to avoid expensive operations, encoder tree pruning using lookup tables etc. Huge latency reduction is achieved through software optimizations as well. Some of the major software optimization methods are unrolling an encoder function, exploiting data parallelism with SIMD, avoiding exponentially complex operations and finally reformulation of polar code construction to avoid expensive remove, erase and copying operations.

Figure 4.1 represents the complete polar FEC chain for PBCH and PDCCH. In general, A bits have to be transmitted with a code of length E bits. L CRC bits are added to the information bits, resulting in $K = (A + L)$ bits. The Resulting K bits are then passed through an input bit-interleaver. Interleaved bits are concatenated with parity bits. In the next step information bit indices are identified and the information bits are inserted in those positions to obtain a vector $\boldsymbol{u}$ of length N, where $N = 2^n$. Encoding is performed with a mother code with parameters (N, K). Encoding is performed through $\boldsymbol{d} = \boldsymbol{u}\boldsymbol{G_N}$, where the generator matrix $\boldsymbol{G_N} = \boldsymbol{G}^{\otimes \boldsymbol{n}}$ obtained by n^{th} Kronecker product of Arıkan matrix. The codeword $\boldsymbol{d}$ is passed through a subblock interleaver. It divides the codeword into 32 blocks and performs interleaving. The interleaving pattern is given in Figure 4.2. The next step involves rate matching, which maps mother code block-length N to rate matching size E bits. Rate matching can be repetition, puncturing or shortening. This decision is taken based on the value of E, N, and K. when $E > N$

repetition is applied. For repetition, some parts of the code word are repeated to create E bits from N bits. For the case $E < N$ either shortening or puncturing is applied. In this mode, bits are discarded to create E bits from N. Finally, channel interleaving is performed to improve the error correction performance for higher order modulations. This chapter dives into the implementation details of each block in an algorithmic level with small code snippets whenever necessary. It also analyzes the bottlenecks and presents solutions through different optimization techniques. All the latency measurements in this section are performed on AMD EPYC processor running at 1.6 GHz with Turbo disabled. Turbo mode allows the processor to dynamically increase the frequency when the load is high, To get accurate measurements, Turbo mode is disabled.

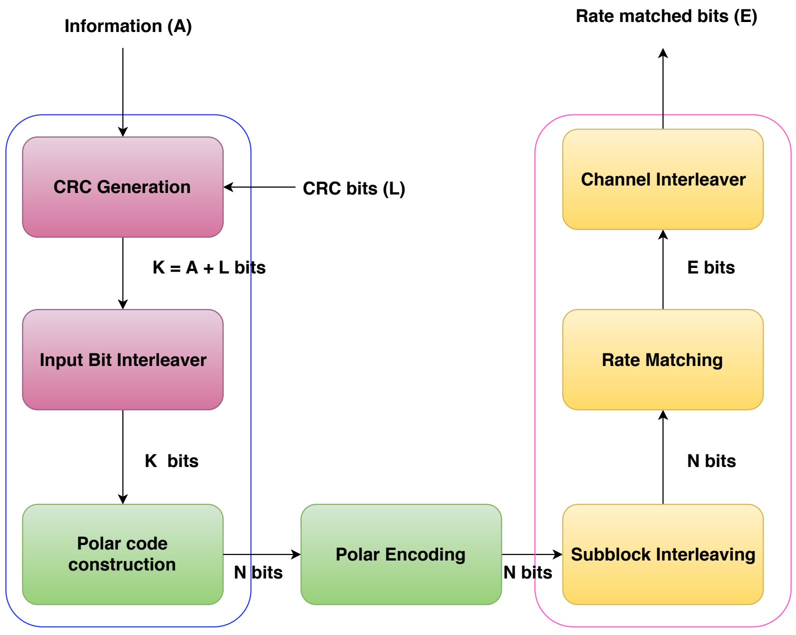

Figure 4.1.: Polar Encoding FEC chain for PDCCH/PBCH

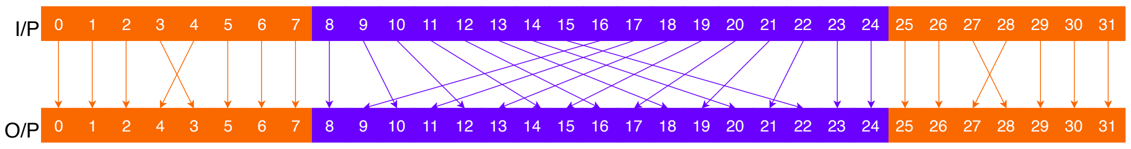

Figure 4.2.: Subblock Interleaver for PDCCH/PBCH

4.1. Data Packing and Unpacking Operations

Typically in software implementations, for clarity and ease implementation each bit of information is represented with 32-bit or 64-bit integers. Due to the presence of only one bit of information in each integer, if 1024 bits need to be encoded or decoded then 1024 integers are involved in encoding or decoding process. However, this isn't the case in hardware implementations since each bit can be processed in hardware description languages (HDL). Representing one bit of information using a 32 or 64-bit integer has the following disadvantages.

- Increased memory footprint: For 1024 bits, $64 \cdot 1024$ bits memory need to be allocated. It is equivalent to 8 kilobytes. Allocating and initializing this memory can introduce significant latency.
- Results in more cache misses: If more memory is allocated, more data needs to accessed from RAM which can result in more cache misses.
- Serializes encoding/decoding: General purpose processors have a data path width of 64-bit. If each bit is represented using a 64-bit integer, we are not using the capability of processing 64 bits simultaneously. Instead, each bit is processed sequentially. This can make encoding or decoding sequential although the processor is capable of processing multiple bits in parallel.

To avoid the above disadvantages and to enable data parallelism, this work tries to pack multiple bits into a single integer. Although packing multiple bits to a single integer has advantages, for some operations such as bitwise interleaving accessing each bit efficiently is very important. To exploit the advantages of bit packing as well as the advantages of accessing each bit separately, it is necessary to convert between the two. This is where the power of SIMD instructions in modern processors comes into play. These processors come with special hardware instructions which help to efficiently pack and unpack data. Data bits are used in packed format when data parallelism needs to be exploited and in unpacked format when certain operations require bits to be accessed individually. These pack/unpack instructions are very efficient and have low latency. Details of the AMD EPYC processor's instructions with corresponding latencies are provided in [19].
Some instructions which are used for fast packing and unpacking are:

Listing 4.1.: Sample packing/unpacking instructions

```
        int _mm_movemask_pi8(__m64 a);
        int _mm_movemask_epi8(__m128i a);
        __m256i _mm256_unpackhi_epi8(__m256i a, __m256i b);
        /** many more **/

```

Listing 4.2.: Sample packing/unpacking example

```
template<>
inline int8_t packBits<8>(int8_t s[]) {
        __m64 v8 = _mm_set_pi8(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7]);
        uint8_t idx = _mm_movemask_pi8(_mm_slli_si64(v8, 7));
}
```

Listing 4.2 shows an example of bit packing. It receives vector of size 8, containing 8-bit integers each representing a bit. Returns a one bit-packed one 8-bit integer.

4.2. CRC Calculation

In the polar FEC chain as shown in the Figure 4.1, an L-bit CRC is calculated for A information bits and attached to the message. The number of CRC bits (L) varies for different physical channels. In the downlink, for the payload of PBCH/PDCCH, 24-bit CRC is used. The Uplink Control Information (UCI) uses a 6-bit or 11-bit CRC based on the value of A. For $12 \leq A \leq 19$ and $A \geq 20$ 6-bit CRC and 11-bit CRC are used respectively. The polynomials for different CRC values are shown below [4].

$$g_6(x) = x^6 + x^4 + 1 \tag{4.1}$$

$$g_{11}(x) = x^{11} + x^{10} + x^9 + x^5 + 1 \tag{4.2}$$

$$g_{24}(x) = x^{24} + x^{23} + x^{21} + x^{20} + x^{17} + x^{13} + x^{12} + x^8 + x^4 + x^2 + x + 1 \tag{4.3}$$

Information bits concatenated with CRC increases the error correction performance of polar codes significantly. CRC is used for selecting the correct codeword out of potential candidates in the list. With CRC aided decoding, polar codes performance better than LDPC and Turbo codes at short block-lengths. To reduce the latency of encoding FEC chain CRC needs to be calculated very efficiently. A naive implementation of CRC calculation uses a shift register. This method calculates the CRC sequentially for one bit at a time as given in [20]. As explained in Section 4.1, it is very inefficient to process bits sequentially. Algorithm in [21] is adapted to calculate the CRC for the polynomial in (4.3) using lookup table based approach. This algorithm calculates the CRC blockwise with the help of the lookup table. In other words, divide the data into blocks of B-bits, read the corresponding CRC value from a lookup table and combine the individual CRC's of the blocks in a predefined way to create a CRC for complete data. Data bits are divided into blocks of 8-bits and packed into 8-bit integers. The CRC value corresponding to 8-bit integer is read from the lookup table and combined with CRC of a previous 8-bit integer. This process continues until CRC is calculated for all data bits. If the number of data bits are not multiple of 8 then zero's are appended at the MSB position. Table 4.1 presents the latency values of naive and optimized CRC calculation methods for payload of 41-bits. There is a significant improvement in the optimized method compared to naive implementation.

Table 4.1.: CRC24 calculation latency comparison

	Naive	Optimized
Latency (μs)	7.7	0.016

After CRC attachment total number of information bits become $K = A + L$, where A is data bits and L is CRC size.

4.3. Input Bit Interleaver

Information bit stream of length A is attached with CRC (L) and is interleaved by the input bit interleaver to create a distributed-CRC. This step is necessary for the early termination of decoding at the receiver if CA-SCL (CRC aided SCL) is used. The interleaving pattern is designed in such a way that every CRC remainder bit is placed after its relevant information bits. This helps to discard some paths in the list decoding process if the CRC calculated from the previously decoded information bits doesn't match the CRC bit. In other words, the input bit interleaver distributes CRC to ease decoding when a list decoder used through early termination. The interleaving pattern is calculated at runtime since it depends on the number of information bits (K). In this part of the implementation, there is not much optimization performed in this work since interleaving needs to carried out sequentially and also due to the fact that K is not very large. Complete details of the input bit interleaver and interleaving pattern calculation can be found in [4].

4.4. Polar Code Construction

Polar code construction is the process of identifying information and frozen bit position, i.e K out of N positions. This step determines the error correction performance of polar codes. There are many methods in the literature to construct polar codes. Arıkan [8] proposed to use the Bhattacharyya parameter as reliability metric for Binary Erasure Channels (BEC) then deriving reliability values using Monte Carlo simulation. For other channels, Mori and Tanaka [22] use more accurate density evolution (DE) methods but it suffers huge complexity. Tal and Vardy proposed Gaussian Approximation (GA) to reduce the complexity of DE with approximations [23]. Still, the GA method has a high computational complexity which scales linearly with code block-length, therefore, it is unacceptable for varying SNR, block-length and code rate. In use cases such as 5G, where the channel is continuously varying. it is not feasible to construct polar codes on the fly due to stringent latency requirements of both encoder and decoder. The polar code construction in 5G takes a suboptimal approach, instead of constructing polar codes for every different SNR, block-length and code rate, construction is carried out in such a way that the constructed code performs sufficiently good over a large range of SNR, block-length and code rate. 5G polar code construction method is based on the contribution from Huawei which uses a β-expansion method with universal partial order (UPO) property of channel reliability as presented in [24].

The 5G standard has adopted five different polar code block-lengths. Block-length sizes $\mathcal{N}$ are by $\mathcal{N} = \{32, 64, 128, 256, 512, 1024\}$. For each of the block lengths, reliability indices values are specified in [4]. The polar code construction also depends on the rate matching mode since it affects the reliability of bit indices. The polar code construction is straightforward when rate matching output E greater than or equal to block-length N. In such a case, code construction involves selection of K most reliable indices for information bits remaining positions are frozen since bit reliability not affected by rate matching. Example 4.4 shows such a case. However when rate matching output size E is smaller than block-length N the selection of reliability indices becomes more involved which is described briefly in the next paragraph.

Polar Code Construction Example for $N = 32, K = 16$ and $E = N$

Let's take an example with $N = 32$ channel reliability values are extracted from the reliability table provided in [4] and are given by

$$Q_0^{31} = \{0, 1, 2, 6, 3, 7, 9, 16, 4, 8, 11, 17, 13, 19, 20, 26, 5, 10, 12, 18, 14, 21, 24, 27, 15, 23, 22, 28, 25, 29, 30, 31\}$$

The bit positions in reliability array Q_0^{31} are ordered in increasing order of their reliability. For encoding with parameters $N = 32$ and $K = 16$, K most reliable indices in the array i.e last 16 indices are used as information bit positions.

$$Q_I^K = \{5, 10, 12, 18, 14, 21, 24, 27, 15, 23, 22, 28, 25, 29, 30, 31\}$$

For any other case, when $E < N$ either puncturing or shortening is performed during rate matching. Empirically it is been observed for polar codes that at low rates puncturing works better and shortening for high rates [25]. In 5G, it is not uncommon to have scenarios with rate matching output E is less than block-length N. In such scenarios, some bits need to be discarded in the rate matching stage through puncturing or shortening. When encoded bits are discarded in the rate matching stage, the reliability of bit channels get affected, identifying reliable bits by taking effect of rate matching procedure makes polar code construction complex in terms of time. The naive implementation of reliability indices selection algorithm provided in [4] is carried out in C++ as shown in algorithm 1. Upon code profiling of encoder FEC chain implementation, it was found that polar code selection algorithm is the most time-consuming part among all the encoding FEC chain stages.

The following algorithm gives a simplified picture of functional implementation to select information bit indices by taking the effect of rate matching. Notations used in the algorithm are the same as the ones specified in 5G standard [4].

$J(n)$	: Subblock interleaver pattern for a particular block-length N.
E	: Rate matcher output size.
N	: Mother code block length.
K	: Number of information bits.
$Q_0^{N\text{-}1}$	: Reliability indices array for block-length N in ascending order of reliability.
$\overline{Q}_I^N$	: Information bit positions.

Algorithm 1 shows how the information bit indices are selected by taking rate matching into account. Finding and removing incapable bits due to rate matching is an expensive operation. As it can be seen in the algorithm in lines 4, 7, 11 and 15 subblock inter-

Algorithm 1: Polar code construction [4] Section 5.4

```
Data: $Q_0^{N\text{-}1}$, $N$, $K$, $E$ and $J(n)$
Result: $\overline{Q}_I^K$
$\overline{Q}_F^N = \emptyset$ ;
if $\big(E < N\big)$ then
    if $\big((K \cdot 16) \leq (E \cdot 7)\big)$ then
        $J_{sorted}(n) = sort(\{J(0), J(1), J(2)....J(N-E)\})$;
        if $\big(E \geq (0.75 \cdot N)\big)$ then
            $size = \lceil\big[(3 \cdot N \cdot 2 - E \cdot 4)/8\big]\rceil$;
            $\overline{Q}_F^N = J_{sorted}(n) \cup \{0, 1, 2, ..., size-1\}$
        end
        else
            $size = \lceil\big[(9 \cdot N \cdot 4 - E \cdot 16)/64\big]\rceil$;
            $\overline{Q}_F^N = J_{sorted}(n) \cup \{0, 1, 2, ..., size-1\}$
        end
    end
    else
        $\overline{Q}_F^N = \{J_{sorted}(E), J_{sorted}(E+1), ...J_{sorted}(N-1)\}$
    end
end
$frozenSize = \big|\overline{Q}_F^N\big|$ ;
$infoSize = N - frozenSize$ ;
$Q_I^N = Q_0^{N\text{-}1}$ ;
for $i = 0$ to $frozenSize$ do
    $iterator = remove(Q_I^N, \overline{Q}_{F,i})$ ;
    $erase(Q_I^N, iterator)$ ;
end
$startIdxInfo = N - K - n_{PC}$ ;
$\overline{Q}_I^K = \{Q_{I,startIdxInfo}, Q_{I,startIdxInfo\ +\ 1}, ..., Q_{I,END}\}$
```

leaving pattern is also taken into account for identifying the incapable bits due to the presence of subblock interleaver between rate matching and encoder. Due to presence of time consuming operations such as *sorting*, *set_union*, *search*, *remove* and *erase*. The contribution of this function highest among all the components of the FEC chain. In terms of latency, for a scenario with $E = 846, N = 1024, K = 130$ puncturing is required. For these parameters, the polar code construction, encoding, subblock interleaving rate matching and channel interleaver takes 411μs, only polar code construction contribution is 377 μs.

Algorithm 2: Proposed optimized polar code construction

```
    Data: $Q_0^{N\text{-}1}$, $N$, $K$, $E$ and $J(n)$
    Result: $\overline{Q}_I^K$
 1  $\overline{Q}_F^N = \emptyset$ ;
 2  if $\big(E < N\big)$ then
 3      if $\big((K \cdot 16) \leq (E \cdot 7)\big)$ then
 4          $J_{sorted}(n) = sort(\{J(0), J(1), J(2)....J(N-E)\})$;
 5          if $\big(E \geq (0.75 \cdot N)\big)$ then
 6              $size = \big\lceil\big[(3 \cdot N \cdot 2 - E \cdot 4)/8\big]\big\rceil$;
 7          end
 8          else
 9              $size = \big\lceil\big[(9 \cdot N \cdot 4 - E \cdot 16)/64\big]\big\rceil$;
10          end
11          $\overline{Q}_F^N = J_{sorted}(n) \cup \{0, 1, 2, ..., size-1\}$ ;
12          $frozenSize = length(\overline{Q}_F^N)$ ;
13          $mode = puncturing$ ;
14      end
15      else
16          $frozenSize = N - E$ ;
17          $mode = shortening$ ;
18      end
19  end
20  $Q_I^N = Q_0^{N\text{-}1}$ ;
21  for $i = 0$ to $frozenSize$ do
22      if $mode == puncturing$ then
23          $index = lookUpTable[\overline{Q}_{F,i}]$;
24      end
25      else
26          $index = lookUpTable[J[i+E]]$ ;
27      end
28      $Q_I^N[index] = \text{INVALID}$ ;
29  end
30  $idx = K$ ;
31  for $i = size(Q_I^N)$ to $0$ do
32      $i = i - 1$ ;
33      if $Q_I^N[i] \neq INVALID$ then
34          $\overline{Q}_I^K[idx] = Q_I^N[i]$ ;
35          $idx = idx - 1$ ;
36      end
37  end
```

Let's analyze the complexity of each the operations. Sorting is $\mathcal{O}\big((N-E)\log(N-E)\big)$ complex and of Set_union is again $\mathcal{O}\big((N-E)\log(N-E)\big)$ complex [26]. The block-length N is derived using E and K, so $(N-E)$ is small compared to N. Next operation in the algorithm is *remove* and *erase*. These functions are directly used from the standard C++ library. After deciding rate matching type (shortening/puncturing) and identifying incapable bit indices, these locations must be frozen, this requires traversing through reliability indices array and removing incapable bit locations. *remove* and *erase* functions are used to perform this operation. *remove* function searches through a reliability array for incapable bit index and removes the element. *erase* operation erases the memory allocated to removed element and resizes the array. The complexity of the remove operation is $\mathcal{O}(N)$. The *remove* function has to search through all the elements of an array for every frozen value and have to move the elements to overwrite the removed position, the size of the array is N, it can be as large as 1024. The erase operation has to deallocate the memory and resize the container. *remove* and *erase* together are $\mathcal{O}\big(N^2\big)$ complex. Above complexity-analysis is supported from latency measurements of *remove* and *erase* functions. It is found to be 318 μs. Avoiding these operations is critical to reducing the latency of the FEC chain.

In this work, the algorithm is reformulated to avoid searching, copying and memory deallocation while removing incapable bit indices. To avoid search operations, a lookup table is built whose values indicate the position of particular reliability value. After identifying the position it is marked as removed instead of removing. Marking has two advantages first one is avoiding memory deallocation and copying, the second one is keeping the same order of elements which is particularly useful for using the same lookup table for finding the next incapable bit index. After all the incapable bit indices are marked as removed, only the unmarked elements are considered as reliable bit positions for placing information bits.

The next optimization is avoiding copying subblock interleaving pattern to frozen indices array in case of shortening. Instead, a subblock interleaving pattern is directly used from the lookup table to mark the reliability indices as removed. In addition to above-mentioned optimizations, minor ones such as avoiding dynamic memory allocation instead reserving required memory in advance and employing pointer operations to avoid copying are performed. Finally, information bit positions are obtained from iterating the reliability table from the end (since indices are sorted in ascending order of reliabilities) and extracting K unmarked positions. These optimizations reduced the latency of polar code construction from 377 μs to 15 μs.

Algorithm 2 presents the optimized reformulation of algorithm 1 without erase, remove,

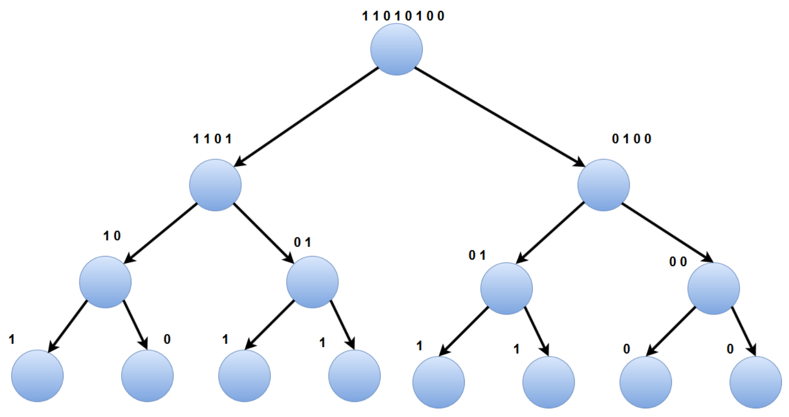

Figure 4.3.: Encoding tree

redundant copying operations and other minor optimizations.
Table 4.2 comparison of latency for reliability indices selection function, for the FEC chain parameters $I_{IL} = 1, n_{max} = 10, n_{pc} = 0, n_{pc}^{wm} = 0, I_{BIL} = 0, E = 846, K = 130$.

Table 4.2.: Latency comparison: Information bit positions selection

	Naive	Optimized
Latency (μs)	377	15

4.5. Polar Encoding

Once the information bit indices are identified, K bits are inserted to reliable positions and remaining $(N - K)$ are set to zero. Next step in the FEC chain is encoding. It can be carried out by multiplying N-bit vector with a generator matrix obtained by the n^{th} Kronecker power of Arikan matrix $\underset{\sim}{G} = \left[\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right]$ where n is such that $N = 2^n$. State of the art direct row vector multiplication with matrix algorithm is $\mathcal{O}\big(N^{2.38}\big)$ complex [27]. However in case of polar codes generator matrix follows a regular structure, hence it is shown that encoding can be reduced to a recursive structure with a complexity of $\mathcal{O}(N \log N)$ [8]. There are different ways to visualize encoder, one is through the encoding circuit and another is through the tree structure. The latter is suitable when encoding is performed in hardware where a group of bits processed in parallel. Since this work focuses on implementation/optimization for software, the tree structure is considered. Example encoding visualized as a binary tree for $N = 8$ is illustrated in figure 4.3. Every node in a tree splits i-bit to $i/2$-bit vector and performs XOR of 0 to $\frac{i}{2} - 1$ bits with $\frac{i}{2}$ to $i - 1$ bits. This process continues till bit vector length becomes one.

As it can be observed from the tree, the same operation is performed at every node only the size of a vector is different, due to the regular structure encoding problem fits the recursive algorithmic form. Algorithm 3 shows a naive implementation of recursive encoder. In the algorithm, each bit is represented as one integer and each bit is processed serially, hence parallelism is not exploited. Upon profiling the implementation it also identified that 8 and 9 are also the bottlenecks since copying is involved. One more issue with the algorithm is recursive implementation. Although the encoding can be easily implemented in this way, each recursive function call in software is expensive, since it requires a new stack frame to be allocated. Background about the overhead of recursive function calling is presented in the previous chapter.

Algorithm 3: Naive polar encoder

```
   Data: $u_0^{N-1}$,$N$,$dIndex = 0$
   Result: $y_0^{N-1}$
1  function recursiveEncode($u_0^{l-1}$,$l$):
2      if $l == 1$ then
3          $y_{dIndex} = u_0$ ;
4          $dIndex = dIndex + 1$ ;
5      end
6      else
7          $len = \frac{l}{2}$ ;
8          $P_0^{len-1} = u_0^{len-1}$ ;
9          $Q_0^{len-1} = u_{len}^{l-1}$ ;
10         for $i = 0$ to $length - 1$ do
11             $P_i = P_i \oplus Q_i$;
12         end
13         recursiveEncode($P_0^{len-1}$,$len$) ;
14         recursiveEncode($Q_0^{len-1}$,$len$) ;
15     end
```

To avoid the disadvantages mentioned above, the following optimization techniques are considered.

• **Data parallelism** To avoid the serial processing of bits and hence to improve the parallelism factor, the method described in Section 4.1 is used, i.e, multiple bits are packed to a single integer. In this particular instance, every 64 bits are packed into 64-bit integers so that 64-bits can be processed in parallel with a 64-bit processor. This results in a parallelism factor ($\mathcal{P}$) of 64. Packing also helps to further increase the parallelism factor $\mathcal{P}$ with state of the art SIMD processing units of modern processors. SIMD instructions can process 256-bit or 128-bit in a single instruction which results in a parallelism factor ($\mathcal{P}$) of 256 with AVX and 128 with SSE instructions.

- **Avoiding the copy operations:** The encoding in Algorithm 3 splits N-bits into two $\frac{N}{2}$-bit vectors and copies them to temporarily allocated variables. Code profiling pointed out that these copying operations are the bottlenecks. In an optimized algorithm, instead of copying, C++ pointers concept is used to calculate the index where the next block of vector starts and this index is passed to the next node for further processing.

- **Unrolling the encoder:** Recursive implementation has significant overhead due to the huge number of recursive function calls which require new stack frame allocation and branching. To avoid this overhead, the encoder implementation is unrolled. In other words, new inline functions are defined for each vector size. An advantage of these functions is that they will not require a new stack frame to be allocated when an inline function is called. They make use of stack frame from the calling function. However, this requires a separate inline function for every vector size. Having a different function for every vector size also has advantages, which allows us to use SIMD instructions whenever vector size can fit SIMD registers otherwise normal instructions are used.

- **Pruning the encoder tree:** As shown in the Figure 4.3, the encoding process can be represented as traversal of a binary tree. During encoding when traversing the tree towards the leaf node bit vector size becomes less than 8 bits. In such a scenario 4/2/1 bits of an integer needs to be accessed. In standard processors smallest unit of data accessed in software is an 8-bit integer. To access 4/2/1 bits, masking operations are needed. The number of nodes in a binary tree which accesses 4/2/1 bits are huge. Therefore a significant number of masking operations are needed, which introduce quite some overhead. Pruning of tree at the level where the bit vector size is 8, avoids this overhead in addition to reducing the number of nodes to be traversed in a binary tree. Pruning is done by building a lookup table containing an encoded value for every combination of the 8-bit vector. Value is read from a lookup table for encoding when the bit vector size is 8. The lookup table has 256 values, one encoded value for every combination of 8-bit vector.

Pruning of the tree had a significant latency improvement, it can be better understood by taking an example. In a scenario where $N = 1024$, with unpruned tree number of nodes to be traversed for encoding, is 2047 nodes, out of these nodes contain 1024 1-bit, 512 2-bit, and 256 4-bit nodes. With pruned tree 4/2/1 bit nodes are not present, the number of nodes needs to be traversed reduces to 255 from 2047 (87% reduction). Pruning avoids masking operations in addition to reducing the nodes in a tree.

Example of a pruned unrolled encoder containing also the tree traversal is shown in Figure 4.4. Inline function names for different bit vector size are also shown in the figure. One can see that tree traversing ends at $bitMult8$ function due to pruning. Tree traversal flow is represented with an orange line in the figure.

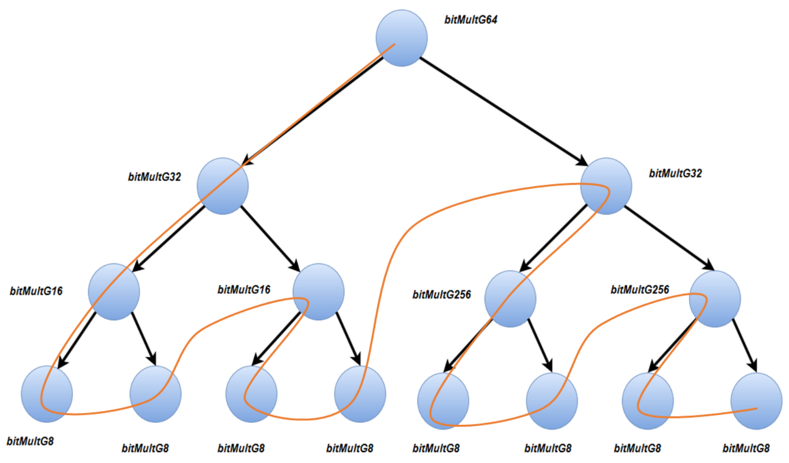

Figure 4.4.: Pruned unrolled encoder tree

Sample code snippet of node operation with SIMD instructions is shown in listing 4.3.

Listing 4.3.: Node operation using SIMD instructions

```
inline void bitMultG512(uint8_t *s, uint8_t *dEncoded)
{
        //Here s is 64 bytes, Divided to 32 bytes each.
        uint8_t *s1 = (uint8_t *) s;
        uint8_t *s2 = (uint8_t *) (s + 32);
        __m256i result;
        __m256i temp1 = _mm256_loadu_si256((__m256i*)operand1);
        __m256i temp2 = _mm256_loadu_si256((__m256i*)operand2);
        result = _mm256_xor_si256(temp1,temp2);
        _mm256_storeu_si256((__m256i*)destn, result);
        bitMultG256((uint8_t *) s1, dEncoded);
        bitMultG256((uint8_t *) s2, dEncoded);
}
```

Table 4.3.: Latency comparison: polar encoding for 1024 bits

	Naive	Optimized
Latency (μs)	34	0.244

4.6. Sub-block Interleaver

Subblock interleaver divides the block of N bits into 32 subblocks, each containing $\frac{N}{32}$ bits. These subblocks are interleaved as shown in Figure 4.2. Functionally, subblock interleaving is implemented with algorithm presented in [4]. Computation complexity of interleaving indexes is huge due to the use of multiplication, division and modulus operations. For a subblock interleaver block, minor optimizations such as avoiding expensive operations such as multiplication/division/modulus and replacing them with right or left shift operations are performed.

4.7. Rate Matching

Rate matching block calculates E bits out of N bits. Value of E relative to N determines the mode of rate matching. There can be three modes namely repetition, shortening and puncturing. For the scenarios where $E \geq N$ repetition is applied otherwise either puncturing or shortening needs to be done. Through empirical measurements, it is found that for high rates shortening performs better and puncturing for low rates. Rate matching is performed by a circular buffer as given in [2]. Major optimization in rate matching block is avoiding copy operations instead change the pointers for cases shortening/puncturing and repetition. Minor optimizations include using compiler primitives as hints to reduce the branchings and copying data blockwise instead of element-wise for repetition mode of rate matching. Blockwise copying is efficient since it can employ SIMD instructions and also it avoids usage of modulus operation in repetition mode.

4.8. Channel Interleaver

The 5G standard specifies channel interleaver for higher order modulations for better error correction performance [28]. In the downlink, PBCH/PDCCH use only QPSK, hence channel interleaving is disabled using a parameter I_BIL. Setting it to zero disables the channel interleaver. For uplink channels such as PUCCH/PUSCH, higher modulations can be used. In those scenarios, channel interleaving is enabled. This work focuses mainly on PDCCH/PBCH channels so interleaving block contribution to latency is zero since it is disabled. There are no optimizations performed for this block in the FEC chain.

4.9. Miscellaneous optimizations

In addition to the above described major improvements, many micro-optimizations are performed. Few examples are
• Avoiding multiplication/division/modulus operations and achieving the same using bit-wise operators.
• Implemented approximate versions $\log_2(x)$ and $\exp(x)$ functions to reduce the number of floating point multiplications.
• Avoided jump functions to reduce flushing of the instruction pipeline.
• Using the compiler optimization primitives to reduce branches in the program.

4.10. Results Comparison

A. PBCH FEC chain

Parameters of the FEC chain are
$n_{max} = 9, I_{IL} = 1, n_{pc} = 0, n_{pc}^{wm} = 0, I_{BIL} = 0, L = 24, E = 864, K = 56$

Table 4.4.: Latency comparison: PBCH FEC chain

	Naive	Optimized
Latency (μs)	53	7

B. PDCCH FEC chain

In case of PDCCH reliability indices need to be identified at runtime due to varying A, N, and E. PDCCH latency gives an overall picture of employed optimizations. Parameters of the FEC chain are
$n_{max} = 9, I_{IL} = 1, n_{pc} = 0, n_{pc}^{wm} = 0, I_{BIL} = 0, L = 24, E = 423, K = 106$

Table 4.5.: Latency comparison: PDCCH FEC chain

	Naive	Optimized
Latency (μs)	162	19

To obtain overall picture of improvement from all the above optimizations it is worthwhile to look at worst case latencies of the FEC chain which is with maximum block size ($N = 1024$), with puncturing (requires more time to identify reliability indices as well as for rate matching) and with both input bit-interleaving and channel-interleaving. Following table gives a comparison between naive and optimized versions.
Worst case latency FEC chain parameters,
$I_{IL} = 1, n_{max} = 10, n_{pc} = 0, n_{pc}^{wm} = 0, I_{BIL} = 1, E = 846, K = 106$.

Table 4.6.: Worst case latency comparison: Polar FEC chain

	Naive	Optimized
Latency (μs)	451	40

4.11. Summary

In this section, we implemented the polar encoding FEC chain. Different components of the FEC chain are analyzed to understand the complexity and latency contributions. Through extensive analysis and code profiling, three major latency contributors are identified, namely CRC calculation, polar code construction, and polar encoding. All these components are optimized using algorithmic and platform-specific optimizations. The latency of CRC calculation is reduced by using lookup and exploiting the data parallelism. The biggest contributor to latency in the FEC chain is the polar code construction due to the presence of expensive search, remove and copy operations. The polar code construction algorithm is reformulated to reduce latency by avoiding above mentioned expensive operations. Polar encoding is another latency contributor. The encoder is optimized using a number of techniques. Significant optimizations include pruning encoder tree, implementing encoding with SIMD instructions, unrolling the recursive function and by avoiding superfluous copy operations. All the above-mentioned optimizations to FEC chain reduced the latency by 10x compared to naive implementation.

5. Decoding FEC Chain

In this chapter, the implementation and optimization details of 5G polar decoding FEC are presented including the challenges faced while achieving low latency decoding. In the decoding FEC chain, the decoder is the critical part due to inherent sequential nature of polar decoding. i^{th} bit is decoded by using all the previously decoded bits, hence i^{th} bit depends on 1 to $i-1$ bits. Due to the sequential decoding process, significant latency is introduced by the decoder. This chapter presents the optimization techniques employed to improve decoding FEC chain latency, which include both algorithmic and platform-specific optimizations. Each of these techniques is explained in a separate section. Every section talks about one particular component of the FEC chain, presents implementation details and employed optimization techniques. In this work, the FEC chain considered is part of the base station, therefore uplink control information is decoded at the receiver. PUCCH and PUSCH contain polar encoded information. Received signal after demodulation is quantized to 16-bit LLR (log likelihood ratio) values. Decoding is performed with LLR (Log-likelihood ratio) values rather than probabilistic likelihoods due to their numerical stability and low computational complexity. Receiver side FEC chain is a reverse of the operations performed at the transmitter. Figure 5.1 shows the receiver side polar decoding FEC chain.

5.1. Decoding Algorithms

The basic decoding algorithm is the successive cancellation (SC) and was developed by Arıkan in his seminal work on polar codes [8]. It achieves the symmetrical capacity of the binary memoryless channel through sequential decoding when block length is very large. However, due to the sequential nature, significant latency is introduced by the decoding algorithm. The latest 5G standard specifies a transmission time interval (TTI) of $125\mu s$ [29]. Within this TTI duration modulation/demodulation and encoding/decoding must happen. Therefore it is very important to efficiently perform FEC chain operations. This work concentrates on implementing the polar encoding/decoding in software and studies the feasibility of satisfying the strict latency requirements of 5G. Decoding through SC algorithm is represented as a binary tree. The decoding process is equivalent to traversing

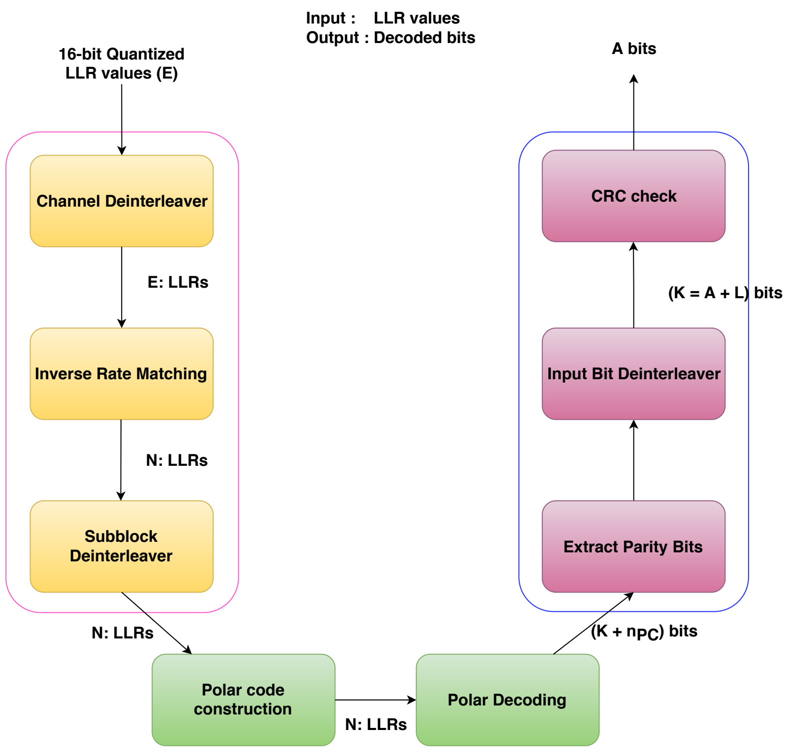

Figure 5.1.: Polar decoding FEC chain for PUCCH/PUSCH

a binary tree. The significant research effort has been devoted by both academia and industry to improve the decoding latency of the SC algorithm. Major improvement to SC algorithm, which reduced the decoding latency is identifying special kind nodes in a tree. These special nodes allow immediate decoding of multiple bits without requiring full tree traversal. Algorithms presented in [12] and [13] present such improvements, which identify special nodes specifically *Rate-0*, *Rate-1*, *RPC* and *SPC* nodes. *RPC* and *SPC* mean repetition and single parity check code respectively. Identification of special nodes requires finding particular patterns in the frozen bit locations in the constructed polar code. To gain full advantages of Fast-SSC (Fast Simplified Successive Cancellation) algorithm, special nodes must be identified efficiently. In this work, 5G RX FEC chain is implemented with the Fast-SSC algorithm, optimized in software and feasibility of achieving desired latency($< 50\mu s$) is analyzed.

5.2. Decoding Chain

This section briefly discusses the functionalities carried out by different blocks of the decoding FEC chain. Figure 5.1 shows the complete receiver side FEC chain. It is almost an inverse operation of encoding FEC chain except a few differences related to PUCCH and PUSCH which contain parity check bits (n_{PC}). The decoding FEC chain receives the Uplink Control Information (UCI) in form of 16-bit quantized E LLR values. Before passing the LLR values to the decoder, the following operations are performed to LLR values namely channel deinterleaving, inverse rate matching and subblock deinterleaving. These steps grouped by a pink rectangle in Figure 5.1. After these steps, the polar code construction is performed using the same optimized method as presented in the previous chapter. Polar code construction procedure outputs the information bit positions, from which frozen pattern can be obtained. The next step in the FEC chain is polar decoding, N LLR values, and the frozen pattern is passed to polar decoder, which outputs the decoded bits. Polar construction and decoding blocks are colored green the FEC chain figure. Using information bit positions obtained in the polar construction procedure $K + n_{PC} + L$ bits are extracted from N decoded bits. $K + n_{PC} + L$ bits contain n_{PC} parity bits, extracting these bits requires identifying the row of minimum weight from the generator matrix of polar code. Finally input deinterleaving is applied on the remaining $K + L$ bits to obtain concatenated information and CRC bits. Blocks representing Extracting parity bits and input bit deinterleaver are grouped with a blue rectangle.

5.3. Channel Deinterlever

The first operation after receiving the LLR values is channel deinterleaving, This is the inverse of interleaving operation done at the transmitter. Channel interleaving is performed to make transmission robust against burst errors. The authors of [28] analyze the error correction performance of polar codes for different channel conditions and constellations. It is found that error correction performance significantly deteriorates for constellations 16-QAM and higher. Channel interleaving isn't applied for downlink PBCH/PDCCH since the constellation is QPSK. In the case of PUCCH/PUSCH, higher constellations are used. Therefore channel interleaving is necessary. In 5G standard, the isosceles right triangle interleaver is adopted. Deinterleaving is carried out by writing LLR values to columns of triangular structure and reading LLR values in rows. Interleaver design is proposed by Qualcomm [28].
Vector processing instructions cannot be used for the implementation of interleaver due to irregular and nonuniform memory access, therefore interleaver just plain functional implementation. One optimization technique was to avoid new memory allocation and using already allocated memory. This avoids the overhead of dynamic memory allocation and initialization. Channel deinterleaving is one significant contributor to latency in polar decoding FEC chain since each of the LLR values needs to be processed sequentially.

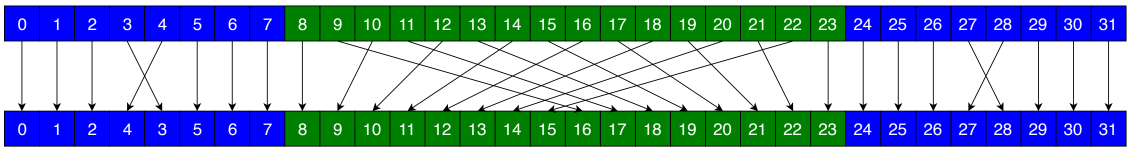

Figure 5.2.: Subblock deinterleaving pattern

5.4. Inverse Rate Matching

Inverse rate matching step maps the E LLR values to mother code block size N. Rate matching step has three modes puncturing, shortening, and repetition. The mode is selected based on rate matcher output size (E) and mother code size(N). If $E > N$ then repetition is performed, otherwise either puncturing and shortening is done. If $\frac{K}{E} > \frac{7}{16}$ shortening else puncturing is performed. Major optimization in inverse rate matching is utilizing SIMD capability for soft combining when $E > N$ and performing block-wise copying.

5.5. Sub-block Deinterleaver

After inverse rate matching, E values are mapped to N LLRs, which is always a power of two. Subblock interleaver/deinterlever divides block of N LLRs into 32 subblocks, each containing $\frac{N}{32}$ LLRs. Functionally, subblock deinterleaving can be implemented as an inverse of interleaving operation as presented in [4]. Upon measuring the latency contribution of subblock deinterleaver it was found to be taking $19\mu s$. Computation complexity of interleaving indexes is huge due to the use of multiplication, division and modulus operations.

If we look at figure 5.2, we can see that not all the values of LLRs are interleaved, only 18 positions out of 32. Calculating interleaving positions is expensive. Instead, they can be pre-calculated and stored in a lookup table. For the mother code size of 1024, with pre-calculated positions interleaving requires looping for 576 times. Modern processors with *AVX* and *AVX2* extensions provide special swizzle instructions, which allow shuffling, permuting and blending of vectors. These instructions process vector of values hence allow data-parallelism (Multiple data elements processed in parallel). To make use of swizzle instructions [30] for sub-block deinterleaving, it must be reformulated to fit into functionality provided by platform-specific SIMD instructions. It is divided into three parts, each one is independent of another. Part one and three are exactly the same operations. Each one is dealing with 8 sub-blocks and performing the operation marked

green in Figure 5.2. Part one and three are mapped to `permute` SIMD instructions. Part two deals with 16 subblocks, marked by blue in the figure. Part two operation is achieved with `blend` and `permute` SIMD instructions provided by AVX2 vector extension.
The code snippet in the listing 5.1 shows sample SIMD implementation of sub-block deinterleaving operation for a mother code size (N) 64.

Listing 5.1.: Vectorized Sub-block deinterleaving for $N = 64$

```
void subblockdeinterleaver64( int16_t y[], int16_t d[]) {
        //interleaving pattern precalculated.
        //v256_perm0,v256_perm1,v256_perm2,v256_perm3;

        //prepare part1
        v256_in = _mm256_loadu_si256((__m256i*)y);
        v256_out = _mm256_permutevar8x32_epi32 (v256_in,v256_perm0);
        _mm256_storeu_si256((__m256i*)d,v256_out);

        //prepare part2
        v256_in = _mm256_loadu_si256((__m256i*)(y + 16));
        v256_out = _mm256_permutevar8x32_epi32 (v256_in,v256_perm1);
        v256_in = _mm256_loadu_si256((__m256i*)(y + 32));
        v256_out2 = _mm256_permutevar8x32_epi32(v256_in,v256_perm2);
        v256_blended = _mm256_blend_epi32 (v256_out,v256_out2,0b11110000);
        _mm256_storeu_si256((__m256i*)(d  + 16),v256_blended);
        v256_out2 = _mm256_permutevar8x32_epi32(v256_out, v256_perm3);
        v256_out = _mm256_permutevar8x32_epi32(v256_in, v256_perm1);
        v256_blended = _mm256_blend_epi32(v256_out2,v256_out,0b11110000);
        _mm256_storeu_si256((__m256i*)(d + 32),v256_blended);
        //prepare part3, same as part1
        //....Same as part 1
}
```

Results latency optimization of sub-block deinterleaver for $N = 1024$ are given in table 5.1.

5.6. Decoder Optimization

This section discusses the details of decoder implementation and optimizations. The Fast-SSC algorithm is considered due to low complexity and the ability to parallelize

Table 5.1.: Latency comparison: Sub-block deinterleaver

	Naive	Optimized
Latency (μs)	19.7	0.47

the decoding operations. Fast-SSC is an improved version of the basic SC algorithm. It identifies the special nodes. Special nodes are decoded without traversing the full tree. These nodes are decoded efficiently using SIMD instructions. Decoding using SIMD instructions parallelizes operations which further reduces the latency. In addition to using SIMD instructions, many software specific optimizations are performed. Each of them is discussed in this section.

5.6.1. VN and CN Operations

Polar decoding with the Fast-SSC algorithm is equivalent to traversing a binary tree. Every node except the leaf node of a tree involves CN, VN and bit combination operations. The result of CN operation is passed to left child and VN operation result to the right child. The decoding result from both child nodes is combined and passed to the parent node. The number of nodes in a tree is exponential dependent on the block length of a code. For a block length of 1024 number nodes in a binary tree are 2047. Decoding requires 2047, CN, VN and bit combination operations, hence it is very critical to implement these operations efficiently. In this work, LLRs are quantized to 16-bit integers hence decoder is a 16-bit fixed point implementation. At each node, a vector of LLRs is received from the parent node. CN and VN operations are performed on the received vector and bit combination on decoded bits from child nodes.

Listing 5.2.: Vectorized CN operation

```
template<unsigned int NvDash>
void CnOp(int16_t alphaL[],int16_t demodLLRs[]) {
  __m256i temp1,temp2,mag_temp1,mag_temp2;
  _m_prefetch(alphaL);
  _m_prefetch(demodLLRs);
  for(unsigned i = 0; i < NvDash; i = i + 16) {
        temp1 = _mm256_loadu_si256((__m256i*)(demodLLRs + i));
        temp2 = _mm256_loadu_si256((__m256i*)(demodLLRs + NvDash + i));
        mag_temp1 = _mm256_abs_epi16 (temp1);
        mag_temp2 = _mm256_abs_epi16 (temp2);
        mag_temp1 = _mm256_min_epi16 (mag_temp1, mag_temp2);
```

```
      temp1 = _mm256_sign_epi16(temp1, temp2);
      temp1 = _mm256_sign_epi16(mag_temp1, temp1);
      _mm256_storeu_si256((__m256i*)(alphaL + i),temp1);
  }
}
```

Listing 5.3.: Vectorized VN operation

```
template<unsigned int NvDash>
void VnOp(int16_t alphaR[],int16_t demodLLRs[],int8_t betaL[]) {
  __m256i alphaLeft,alphaRight,beta;
  _m_prefetch(alphaR);
  _m_prefetch(demodLLRs);
  for(unsigned i = 0; i < NvDash; i = i + 16) {
    alphaLeft = _mm256_loadu_si256((__m256i*)(demodLLRs + i));
    alphaRight = _mm256_loadu_si256((__m256i*)(demodLLRs + i + NvDash));
    beta = _mm256_cvtepu8_epi16(_mm_loadu_si128((__m128i*)(betaL + i)));
    beta  = _mm256_slli_epi16(beta,15);
    beta = _mm256_or_si256(beta,_mm256_set1_epi16(1));
    alphaLeft = _mm256_sign_epi16(alphaLeft,beta);
    alphaRight = _mm256_add_epi16(alphaRight,alphaLeft);
    _mm256_storeu_si256((__m256i*)(alphaR + i),alphaRight);
  }
}
```

If the CN operation needs to perform on the received vector, the naive method would be to access each value from the vector and perform the same CN operation. Sequentially performing CN and VN operations is very inefficient. Size of the vector received at each node is always a power of 2. Due to this fact, CN and VN operations naturally fit vector processing units provided in the modern processors. In this work, both CN and VN operations are efficiently implemented using SSE and AVX instruction extensions provided by AMD EPYC platform. During CN and VN operations memory access is regular therefore data required in the future can be fetched to cache from main memory to reduce cache misses. Listing 5.2 and 5.3 show the efficient vectorized CN and VN operations. Bit combination operation is an XOR operation decoded bits from child nodes. Bit combination is again implemented using SIMD vector XOR operations.

5.6.2. Identifying Component Codes

The naive SC algorithm is purely sequential, hence decoder introduces significant latency in the FEC chain. With improvements such as [13] and [12] decoding, the tree can be pruned by identifying particular patterns in frozen bit positions. Pruning of a tree allows decoding of multiple bits in parallel. Component codes out of polar codes can be identified. These codes allow decoding without traversing the full decoder tree. Authors in [12] and [13] identify four such codes namely rate-0, rate-1, repetition and single parity codes. Decoding a codeword through component codes improves latency without compromising the error correction performance. However, to fully enjoy the fruits of decoding tree pruning, the implementation should be able to identify component codes efficiently. One simple functional way is to go through all frozen bits and search by comparing with predefined patterns. The naive way of searching for a pattern introduces significant latency in the decoding process. Processors with *AVX* and *AVX2* support contain registers which can store 256/128 bits in a single register. Frozen pattern array/vector contain either one or zero, one indicating a frozen bit and zero an information bit. Since one bit is enough to represent the type of bit position, frozen pattern can be stored by packing multiple bits type information to single 256bit register. Bit packing allows identifying a pattern by comparing with an integer vector using single SIMD instruction. For example, a mother code size $N = 256$ information about which position is frozen and which is not is stored in a single 256 bit SIMD register in a bit packed format. To check whether it is a rate-0, rate-1, SPC or RPC requires one SIMD comparison instruction. The snippet in listing 5.4 illustrates an example of identifying node type in bit packed frozen bits pattern.

Listing 5.4.: Checking rate-0 node

```
template<>
inline int identify_R0<256>(uint64_t s[]) {
        __m256i temp1 = _mm256_loadu_si256 ((__m256i*)s);
        __m256i temp2;
        temp2 = _mm256_set1_epi8 ((char)0xFF);
        __m256i pcmp = _mm256_cmpeq_epi64 (temp1, temp2);
        unsigned bitmask = _mm256_movemask_epi8(pcmp);
        return (bitmask == 0xffffffffU);
}
```

5.6.3. Decoding Rate-0 Code

The R0 code is a special kind of node in a decoding tree in which all the descendants represent frozen bit positions, in other words, the corresponding node's frozen pattern contains all ones. One such example is given in the background chapter. For such a node, we know that all the bits are frozen hence decoder can immediately decode values as zero. All the decoded bits corresponding to such a node are set to zero. Rate-0 node allows the decoder to avoid performing VN and CN operations at the subsequent child nodes in addition to decoding multiple bits simultaneously.

5.6.4. Decoding Rate-1 Code

A node is considered as an R1 node if all of its descendants in a decoder tree are information bits. In other words, the Rate one node contains no frozen bits. Decoding of Rate-1 nodes is performed without traversing till the end of a decoder. This avoids a significant number of VN and CN operation and function calls. However, decoding Rate-1 node is not as straightforward as Rate-0 node. Decoding is performed through threshold detection of all the LLRs and performing the polar transform on the result to obtain decoded bits.

Algorithm 4: Rate-1 node decoding algorithm

```
  Data: $\alpha_v^{N_v-1}$, $N_v$
  Result: $y_0^{N_v-1}$, $\beta_0^{N_v-1}$
1 function decodeR1($\alpha_v^{N_v-1}$,$y_0^{N_v-1}$, $\beta_0^{N_v-1}$):
2     for $i = 0$ to $N_v - 1$ do
3         $\beta_v[i] = \alpha_v[i] \geq 0$;
4     end
5     $y_0^{N_v-1} = polarTransform(\beta_0^{N_v-1})$ ;
```

Although R1 nodes avoid CN and VN operations, decoding is not parallel. The decoder needs to go through each of the LLRs to decode the bits and finally perform the polar transform. Both steps are costly operations. To improve the latency through data parallelism, threshold detection can make use of SIMD instructions. Threshold detection of a complete vector can be performed through single SIMD comparison instruction. This improves the parallelism factor to sixteen for 16-bit LLRs with AVX2 instructions. The code snippet in listing 5.5 presents an example where rate-1 node decoding is implemented using AVX instructions. Resulting parallelism factor is eight.

Listing 5.5.: Decoding rate-1 subcode

```
template<unsigned Nv>
void decR_1(int16_t demodLLRs[],int8_t beta[],int8_t decodedBits[]) {
  __m128i temp1,tempDecodedVec;
  _m_prefetch(beta);
  _m_prefetch(decodedBits);
  for(unsigned i = 0; i < Nv; i = i + 8) {
    temp1 = _mm_loadu_si128((__m128i*)(demodLLRs + i));
    tempDecodedVec = _mm_cmplt_epi16(temp1,zeros);
    tempDecodedVec = tempDecodedVec & _mm_set1_epi16(1);
    tempDecodedVec = _mm_packs_epi16(tempDecodedVec,_mm_setzero_si128());
    _mm_storeu_si128((__m128i*)(beta + i),tempDecodedVec);
    _mm_storeu_si128((__m128i*)(decodedBits + i),tempDecodedVec);
  }
  polarTransform<Nv>(decodedBits,decodedBits);
}
```

Next step in decoding rate-1 node is performing polar transform operation. It is equivalent to performing polar encoding. As explained in the previous chapter, the binary tree represents the encoding process. Efficient polar transform implementation makes use of the same optimizations techniques employed in encoding such as SIMD vectorization and lookup table techniques.

5.6.5. Decoding an RPC Code

Repetition code (RPC) is another type of component identified from polar code. It also allows decoding multiple bits without full tree traversal. A node is considered as RPC when only one of its rightmost descendent contains information and all remaining bits are frozen. A bit packed frozen pattern allows easy identification of RPC node. If frozen pattern at the node is equal to one then it is an RPC node. RPC node decoding is similar to simple repetition code decoding. Bit decoded by summing all LLR values and doing threshold detection of the result. The result of threshold detection is stored at information bit position and remaining bits at the node are set to zero.

$$\beta_v[i] = \begin{cases} 0, & \text{when } \sum_j \alpha_v[j] \geq 0; \\ 1, & \text{otherwise.} \end{cases}$$

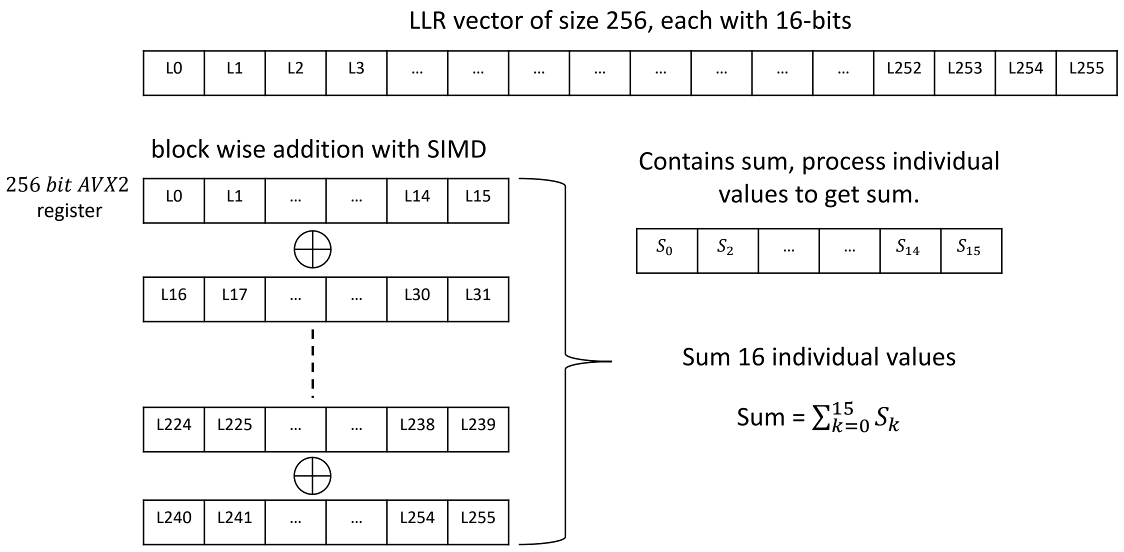
LLR vector of size 256, each with 16-bits
L0
L1
L2
L3
L252
L253
L254
L255
block wise addition with SIMD
256 bit AVX2 register
L14
L15
L16
L17
L30
L31
L224
L225
L238
L239
L240
L241
Contains sum, process individual values to get sum.
S_0
S_2
S_{14}
S_{15}
Sum 16 individual values
Sum = $\sum_{k=0}^{15} S_k$

Figure 5.3.: Block wise addition with SIMD instructions

Again, the RPC-node decoding process has room for improvement. Decoding requires summing of all LLRs. Summation operation efficiently performed with SIMD instructions through data parallelism instead of adding individual values. Figure 5.3 illustrates how blockwise addition is performed using SIMD instructions. This vectorization has a significant gain when the sum of huge LLR vector is required.

5.6.6. Decoding an SPC Code

Another type of constituent codes identified from decoding tree are single parity check (SPC) codes. SPC codes can also be decoded more efficiently without complete tree traversal. These nodes have a code rate $Nv - 1/Nv$. SPC nodes have only one frozen bit at the right most position remaining ones are information bits. Optimal ML decoding of SPC codes is performed with very low complexity. Similar to R1 code decoding SPC code requires threshold detection for all the LLRs. To achieve this efficiently same optimization of Rate-1 to employ threshold detection for whole vector is reused. For SPC decoding two more additional steps are required namely finding position of minimum magnitude LLR and calculating the parity of decoded bits. If the parity of decoded bits is not even then bit value at the position of lowest magnitude LLR is flipped. Final step in the decoding is to obtain final decoded bits values through polar transform. The steps described above are shown in the algorithm 5.

SPC code uses the same two operations (threshold detection and polar transform) as used in RPC decoding. This allows reusing of the same optimization techniques. Two

Algorithm 5: SPC decoding

```
Data: $\alpha_v^{N_v-1}$, $N_v$
Result: $y_0^{N_v-1}$, $\beta_0^{N_v-1}$
function decodeSpc($\alpha_v^{N_v-1}$,$y_0^{N_v-1}$, $\beta_0^{N_v-1}$):
    $j = \alpha_v[i]$ ;
    $parity = 0$ ;
    for $i = 0$ to $N_v - 1$ do
        $\beta_v[i] = \alpha_v[i] \geq 0$;
        if $j < |\alpha_v[i]|$ then
            $j = i$ ;
        end
        $parity = parity + \beta_i[i]$ ;
    end
    if $parity \neq 0$ then
        $\beta_v[j] = 1 - \beta_v[j]$;
    end
    $y_0^{N_v-1} = polarTransform(\beta_0^{N_v-1})$ ;
```

additional operations in SPC decoding are finding the position of minimum magnitude LLR and calculating the parity. The time complexity of finding a minimum magnitude LLR position is $\mathcal{O}(N)$. It can be reduced to $\mathcal{O}(N/8)$ by mapping search operation to SIMD instruction which processes vectors of size eight in parallel. SSE4.1 instruction `phminposu` comes to rescue. It processes vectors of size 128 bits, computes the minimum amongst the packed unsigned 16-bit vectors returns the position and its value. Parity calculation of the decoded bits also requires iterating through all bits obtained after threshold detection. To efficiently perform parity check calculation decoded bits are packed into a single integer. The number of set bits in an integer is obtained by the hardware instruction `popcnt`. If the number of set bits is odd then a bit in the lowest magnitude LLR position is flipped. These optimizations reduced the latency of SPC decoding to less than 50% of naive algorithmic implementation shown in 5.

5.6.7. Partial Unrolling of the Decoder

Recursive formulation of decoder makes it easy to implement decoding algorithm using dynamic programming. It also allows the reuse of hardware resources in FPGA/ASIC. Successive cancellation decoding algorithm for polar codes is formulated recursive form. However, this formulation has a huge disadvantage when it comes to software implementations. Every recursive function call in software requires new stack frame allocation and jump to a starting address of the function. Both operations are expensive in terms of pro-

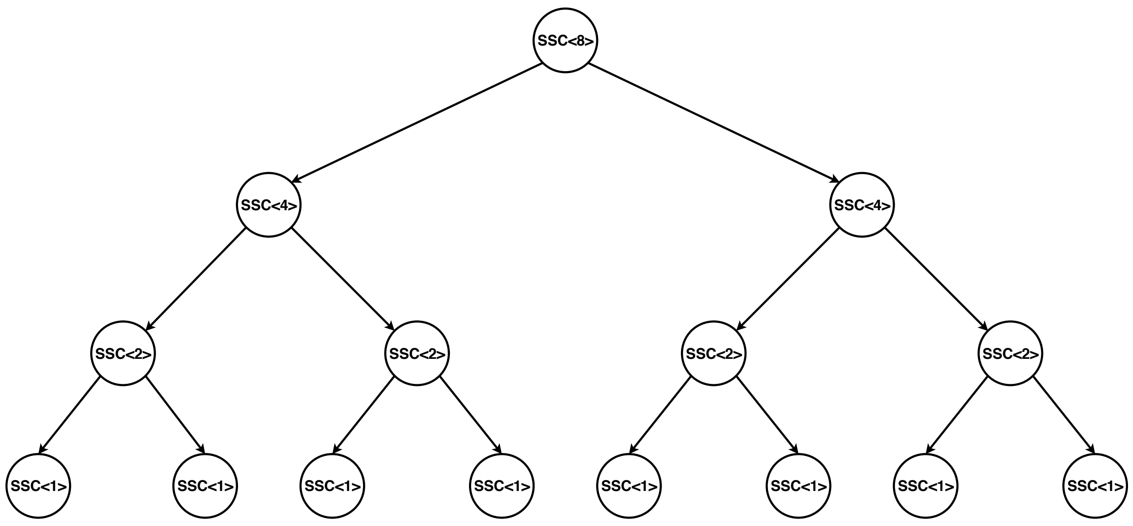

Figure 5.4.: Unrolled decoder

cessor cycles. Authors in [10] propose complete unrolling of the decoder, hence avoiding recursive function calls and branch instructions. With complete unrolling, number special nodes and their level in decoder tree are known at compile time due to the fixed frozen bit position. In polar FEC chain of 5G, a full unrolling of the decoder is not possible due to varying block-length and code rate requirements. Different code rate requirement makes it impossible to know frozen bit positions during compile time. Therefore it is necessary to dynamically identify the presence of special nodes in the decoder tree. In this work, partial decoding unrolling is done, i.e recursive function calls are completely avoided. Template concept of C++ language is used for auto unrolling of functions. However, branches are still present to check for the presence of special nodes in a tree. Partial unrolling also reduced the decoding latency considerably.

5.6.8. Cache Prefetching

A significant performance bottleneck in the modern processors is the access to memory. As presented in Chapter 2, Usage of caches reduces the number of accesses to main memory. However, due to the limited cache size, it might happen that the requested data is not present in the cache. The absence of data in cache results in an event called cache-miss. In this case, data is requested from main memory. Copying data from main memory to cache has a huge overhead. The cache-miss overhead can be reduced by dealing with less memory allocation. In other words, we should reuse the memory as much as possible. Some of the modern processors provide special nonblocking instructions which allow cache line prefetching. If the memory access required is known in advance, the prefetching instruction can be executed well in advance by software before accessing memory. These instructions if used efficiently can hide the memory access latency. In this work, decoder implementation is optimized for AMD EPYC platform, which provides cache prefetching instructions through `3dnow` extensions. Due to regular memory access in the polar decoder, prefetching instructions are used whenever possible to hide memory access latency.

5.6.9. Eliminating Superfluous Operations on β-Values

Every non-leaf node in the decoder performs bit combine operation to obtain β. Only half of the β is calculated by XOR operation remaining bits are copied unchanged. One optimization method is to eliminate these superfluous copy operations by choosing suitable memory layout for β values. If i is the size of β, only $\frac{i}{2}$ values are modified. After updating $\frac{i}{2}$ bits same aligned memory address is passed to the parent node. Since vector sizes are always powers of two, memory passed to the parent node is again implicitly aligned to SIMD vector size. This alignment of memory allows vectorization of bit combination operation at the parent node.

5.6.10. Decoder Tree Pruning

Decoding latency is further reduced by minimizing the number of nodes to be traversed in a decoder tree. This is achieved by pruning decoding tree intelligently at a particular level based on the given SNR and code rate. The level of pruning for a particular SNR and code can be determined through simulations. This information is included in the implementation to determine the pruning level during decoding. Here decoder tree is pruned irrespective frozen pattern type. The main idea is to adoptively prune the decoding tree depending on the SNR and code rate in hand. Tree pruning level is determined before the decoder starts. As decoding proceeds and reaches a particular level, the frozen pattern is checked whether it matches any special patterns such as R0, R1, RPC or SPC. If it matches any of these patterns then decoding is performed accordingly. If not further traversing the tree is avoided by decoding through threshold detection and then applying a polar transform. This is equivalent to performing a hard decision decoding at that node. two levels of pruning are investigated in this work namely 8 and 4. Pruning level is decided based on SNR and code rate.

Pruning at level 8 reduces the decoding latency from 5.3us to 4.0us and pruning at level 4 reduces latency to 5us from 5.3us (average values). Pruning comes at the expense of error correction performance. However, for the scenarios where the code rate is low and SNR is good, decoder tree can be safely pruned. Figure 5.6 shows the error correction

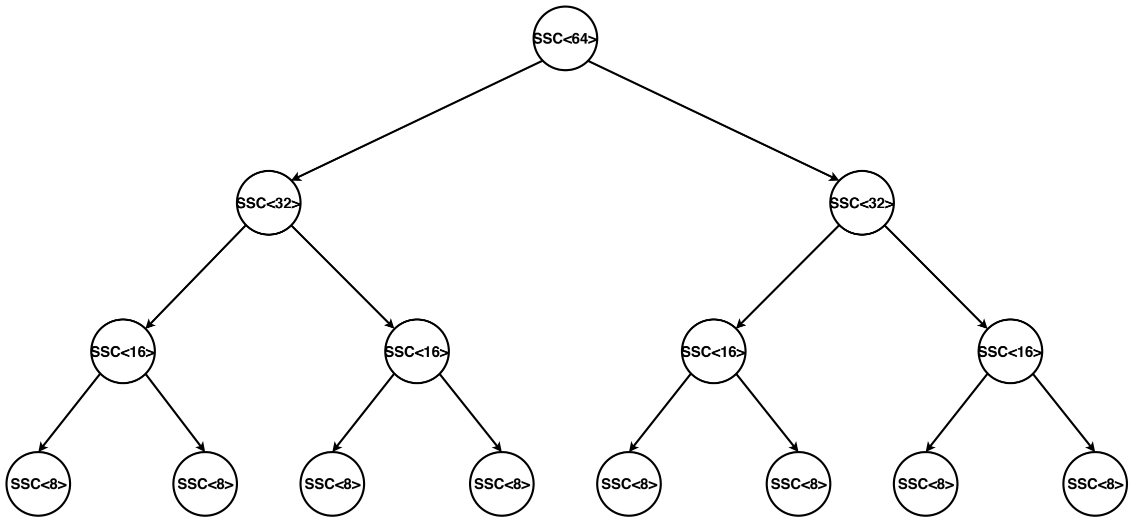

Figure 5.5.: Pruned decoder tree

performance versus the pruning level plot for a code rate of 0.5.

5.6.11. Decoding Latency Comparison

Following tables represent decoding latency representations

Table 5.2.: Latency comparison: Plain versus Optimized implementation

	Functional	Optimized
Latency (μs)	283.4	5.3

Table 5.3.: Latency comparison: this work versus state of the art [3]

	this work	[3]*
Latency (μs)	5.3	8

* Scaled according to frequency

Authors in [3] use Intel-i7 processor running at 3.1GHz for latency measurement. In this work, the AMD EPYC processor is running at 1.6GHz. Therefore the latency values in [3] are scaled by frequency factor for meaningful comparison.

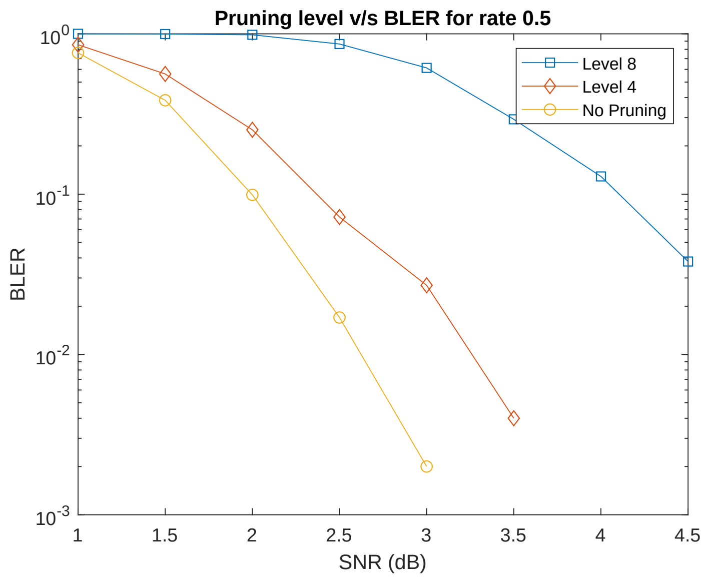

Figure 5.6.: Pruned code error correction performance for code rate 0.5

5.7. Extracting Parity Check Bits

Parity bits are useful for early termination when decoding performed with list algorithm. Polar decoder output is a transmitted codeword of mother code with a block-length N including frozen, information bits and parity check bits. As specified in the 5G standard [4] for PUCCH three parity bits are calculated ($n_{PC} = 3$) two of these bits are placed in the two least reliable positions out of $K + n_{PC}$, where $K + n_{PC}$ positions are the most reliable out of N. The remaining one bit ($n_{PC}^{wm} = 1$) is placed at a position which corresponds minimum hamming weight row of the polar code generator matrix. This position needs to be identified dynamically since it changes for different code rates. Identifying a minimum Hamming weight row requires information about the number of ones in each row of generator matrix. One way to know is by storing hamming weight of every row in a look-up table and read the values of particular rows. Another way is to exploit the unique structure of the polar generator matrix, i.e. number set bits in an integer representing the row number gives the hamming weight of a particular row [2]. For the latter method lookup table is not required. In the FEC chain implementation, latter method is implemented and it has the following advantages over the latter.

- Modern processors provide instructions to efficiently calculate the number of set bits in an integer. AMD EPYC platform provides `popcnt` instruction [19].
- Dynamically calculating reduces the number of irregular memory accesses. Result-

ing in reduced cache misses.

After extracting parity bits. these positions are marked as frozen to ease the extraction of information bits.

5.8. Extracting Information Bits

After extracting parity bits, the codeword of mother code with a block-length N contains frozen and information bits. To obtain only information bits, again at the receiver highest reliability indices must be identified for the same parameters E, K and N. The same algorithm presented in the encoding chain chapter is used for identifying reliability indices. The information bits are read from K most reliable positions out of N. There are two ways to extract information bits, One is to take most reliable positions, sort them, extract information bits in the order of increasing reliability. Another method is using the previously created frozen pattern, read bits from the location where it contains zero (means that position has information bit). Extensive profiling identified that sorting is more expensive than going through all N values and reading information bits from non-frozen locations. So the latter method is included in the FEC chain.

5.9. CRC calculation

For UCI, the 5G standard specifies different CRC sizes depending on the number of information bits(A). As specified in [4], if $A \geq 20$ $CRC11$ otherwise $CRC6$ is calculated. For CRC calculation algorithm [21] is used. It is adapted to calculate CRC blockwise for the blocks of size 8-bits with the help of prebuilt lookup tables for $CRC11$ and $CRC6$. Unlike PDCCH/PBCH CRC implementation, for CRC6 and CRC11 is a generic logic is implemented. Type of CRC is decided at runtime based on the value of A and it is calculated with the same generic logic.

5.10. Decoding FEC Chain Latency Results

$I_{IL} = 0, n_{max} = 10, n_{pc} = 3, n_{pc}^{wm} = 1, I_{BIL} = 1, E = 1692, K = 512.$

Table 5.4.: Latency comparison: Polar decoding FEC chain

	Naive	Optimized
Latency (μs)	391	40

5.11. Summary

In this section, the polar decoding FEC chain is implemented. Each component of the FEC chain is analyzed to understand the complexity and latency contribution. After extensive analysis and code profiling, three major latency contributors are identified namely sub-block deinterleaving, polar decoding and CRC calculation. Sub-block deinterleaving is optimized through algorithm reformulation and mapping interleaving operations to specialized SIMD instructions. Biggest latency contributor in the decoding FEC chain is polar decoder. A number of optimizations are performed to reduce decoder latency. Optimizations which resulted in major latency reduction are SIMD implementation of CN, VN and bit combination operations and component code decoding, unrolling the decoder, avoiding superfluous copying and cache prefetching. CRC calculation is optimized through a lookup table and parallel bit processing. Above optimizations reduced the latency of the FEC chain by 10x compared to the naive implementation.

6. Conclusion and Outlook

The objective of this work is to study the feasibility of developing polar FEC chain of 5G in software on general-purpose-processor while satisfying stringent latency requirements. In other words, all the components of encoder and decoder FEC chain are developed on general purpose AMD EPYC processor. The software satisfies latency constraint of less than 50μs. In the first part of the thesis, we provide necessary background about polar encoding/decoding and computer architecture. In the second part, we develop encoding and decoding FEC chains and optimize them to satisfy the necessary latency constraints.

To begin with, we provided necessary mathematical background about polar code construction, polar encoding, and decoding. Including different polar decoding algorithms. To understand FEC chain development in software it is necessary to know the basics of modern computer architecture. Computer architecture section talks about pipelining, cache memory and vector processing units in modern general purpose processors.

In the next chapter, we talk about the details of polar encoding FEC chain. In this chapter, we analyze the different components of the FEC chain to identify latency contributors. Each of these latency contributors is further studied to reformulate the algorithm to avoid costly operations. Algorithms are reformulated to fit into specialized functional units of modern processors such as vector processing units. Vector processing units allow data parallelism in addition to supporting very fast mathematical computations. The encoding, major latency contributors were polar code construction, CRC calculation, encoding, and rate matching. A wide range of optimization techniques is employed to reduce the latency both algorithmic and platform specific. Namely, reducing algorithm complexity, using lookup tables, compiler hints for better instruction scheduling, vector processing instructions for data parallelism and avoiding superfluous copy operations et cetera. Optimizations reduced the worst-case latency of the encoding FEC chain from 451μs to 40μs which is more than 10x reduction in latency.

For the decoding FEC chain again same steps as encoding chain are followed to identify the latency contributors. Major contributors in decoding FEC chain were channel deinterleaver, subblock deinterlever, polar decoder, parity bit extractor, and CRC calculation. Decoding FEC chain extensively uses SIMD, bit count, cache prefetching instructions to reduce latency. Subblock deinterleaving operation is divided into three primitive small operations which are implemented efficiently with `permute` and `blend` vector instructions. The polar decoder is optimized by implementing XOR, CN, VN, bit combination and frozen pattern identification operations using vector processing instructions. Parity bit extractor optimized by avoiding expensive remove and erase operations instead uses modified algorithm marking indexes and dynamically calculating hamming weights of generator matrix rows. Finally, for CRC calculation an algorithm based on lookup table is developed based on [21] which processes block of data bits to calculate CRC. These optimizations significantly reduced the latency of decoding FEC chain from 391μs to 40μs almost a 10x reduction in latency.

As an outlook, for the above stated decoding FEC chain, decoder is developed with *fast-SSC* algorithm. This algorithm has much lower error correction performance than similar block-length LDPC and Turbo counterparts. As part of this work, *CRC-Aided Successive Cancellation List* (*CA-SCL*)[9] decoding algorithm is also implemented, however, it is not optimized for software. *CA-SCL* ideally suits very low SNR scenarios such as mmWave communication. It has approximately $1.5dB$ gain over *fast-SSC* algorithm for $N = 2048$ and list size $L = 8$. Ideal continuation of this work would be to extend the decoding chain by incorporating *CA-SCL* algorithm to the FEC chain. It would be interesting to see the latency values of this algorithm, which has expensive *sort* and *copying* operations.

Bibliography

[1] Vadikus, “SIMD Units.”

[2] V. Bioglio, C. Condo, and I. Land, “Design of Polar Codes in 5G New Radio,” *ArXiv e-prints*, Apr. 2018.

[3] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, “Low-latency software polar decoders,” *J. Signal Process. Syst.*, vol. 90, pp. 761–775, May 2018.

[4] 3GPP, “Technical Specification Group Radio Access Network; NR; Multiplexing and channel coding,” Technical Specification (TS) 38.212, 3rd Generation Partnership Project (3GPP), April 2018. Version 15.1.1.

[5] Gisselquist Technology, “Minimizing FPGA Resource Utilization.”

[6] U. Drepper, “What Every Programmer Should Know About Memory,” tech. rep., Red Hat, Inc., 2018.

[7] www.amd.com, “AMD EPYC Procesor Datasheet.”

[8] E. Arikan, “Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels,” *IEEE Trans. Inf. Theory*, vol. 55, pp. 3051–3073, July 2009.

[9] I. Tal and A. Vardy, “List decoding of polar codes,” *IEEE Trans. Inf. Theory*, vol. 61, pp. 2213–2226, May 2015.

[10] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, “Fast polar decoders: Algorithm and implementation,” *IEEE J. Sel. Areas Commun.*, vol. 32, pp. 946–957, May 2014.

[11] Wikipedia, “Kronecker Product.”

[12] A. Alamdar-Yazdi and F. R. Kschischang, “A simplified successive-cancellation decoder for polar codes,” *IEEE Commun. Lett.*, vol. 15, pp. 1378–1380, December 2011.

[13] M. Hanif and M. Ardakani, "Fast successive-cancellation decoding of polar codes: Identification and decoding of new nodes," *IEEE Commun. Lett.*, vol. 21, pp. 2360–2363, Nov 2017.

[14] A. Herkersdorf, "Chip Multi Core Processors." Lecture Notes, Institute for Integrated Systems, Technische Universität München, 2017.

[15] A. Herkersdorf, "System on Chip Technologies." Lecture Notes, Institute for Integrated Systems, Technische Universität München, 2016.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* The Morgan Kaufmann Series in Computer Architecture and Design, Burlington, Massachusetts: Morgan Kaufmann, 2011.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms.* Cambridge, Massachusetts: The MIT Press, 2009.

[18] 3GPP, "Technical Specification Group Radio Access Network; NR; Physical channels and modulation," Technical Specification (TS) 38.211, 3rd Generation Partnership Project (3GPP), April 2018. Version 15.1.0.

[19] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," tech. rep., Technical University of Denmark, 2018.

[20] Wikipedia, "Cyclic redundancy check."

[21] D. V. Sarwate, "Computation of cyclic redundancy checks via table look-up," *Commun. ACM*, vol. 31, pp. 1008–1013, Aug. 1988.

[22] R. Mori and T. Tanaka, "Performance of polar codes with the construction using density evolution," *IEEE Commun. Lett.*, vol. 13, pp. 519–521, July 2009.

[23] I. Tal and A. Vardy, "How to construct polar codes," *IEEE Trans. Inf. Theory*, vol. 59, pp. 6562–6582, Oct 2013.

[24] G. He, J. Belfiore, I. Land, G. Yang, X. Liu, Y. Chen, R. Li, J. Wang, Y. Ge, R. Zhang, and W. Tong, "Beta-expansion: A theoretical framework for fast and recursive construction of polar codes," in *IEEE Global Telecommun. Conf*, pp. 1–6, Dec 2017.

[25] V. Bioglio, F. Gabry, and I. Land, “Low-complexity puncturing and shortening of polar codes,” in *IEEE Wireless Commun. and Netw. Conf. Workshop (WCNCW)*, pp. 1–6, March 2017.

[26] C++ Standardization Committee, “C++ reference.”

[27] F. L. Gall, “Powers of tensors and fast matrix multiplication,” *CoRR*, vol. abs/1401.7714, 2014.

[28] Qualcomm, “Interleaver design for Polar Codes,” Discussion/Decision 7.1.4.2.1.3, 3rd Generation Partnership Project (3GPP), May 2017. Meeting 89.

[29] www.sharetechnote.com, “5G - Frame Structure.”

[30] Intel, “Intel Intrinsics Guide.”

[31] T. Cover and J. Thomas, *Elements of Information Theory.* Wiley series in telecommunications, New York: John Wiley & Sons, 1991.

[32] K. Niu, K. Chen, J. Lin, and Q. T. Zhang, “Polar codes: Primary concepts and practical decoding algorithms,” *IEEE Commun. Mag.*, vol. 52, pp. 192–203, July 2014.

[33] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg, “Llr-based successive cancellation list decoding of polar codes,” *IEEE Trans. Signal Process.*, vol. 63, pp. 5165–5179, Oct 2015.

[34] G. Sarkis and W. J. Gross, “Increasing the throughput of polar decoders,” *IEEE Commun. Lett.*, vol. 17, pp. 725–728, April 2013.

[35] B. L. Gal, C. Leroux, and C. Jego, “Multi-gb/s software decoding of polar codes,” *IEEE Trans. Signal Process.*, vol. 63, pp. 349–359, Jan 2015.

[36] H. Ji, S. Park, J. Yeo, Y. Kim, J. Lee, and B. Shim, “Ultra-reliable and low-latency communications in 5g downlink: Physical layer aspects,” *IEEE Wireless Commun.*, vol. 25, pp. 124–130, JUNE 2018.

[37] G. Sarkis, *Efficient Encoders and Decoders for Polar Codes: Algorithms and Implementations.* PhD thesis, McGill University, Montreal, Canada, 2016.