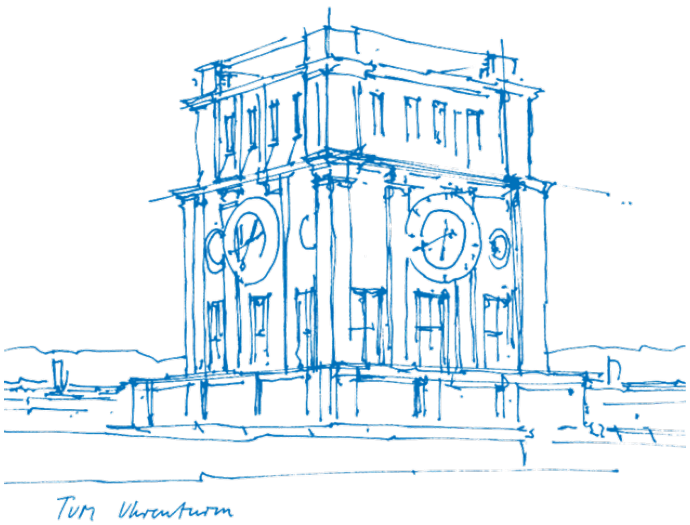
Chair of Integrated Systems
TUM School of Computation, Information and Technology
Technical University of Munich

Real-Time Processing of Forward Error Correction Coding for High-Datarate Free-Space Optical Satellite Communication on a Commercial Off-The-Shelf System on Chip

Jan Hildebrand



DLR Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

Chair of Integrated Systems
TUM School of Computation, Information and Technology
Technical University of Munich

Real-Time Processing of Forward Error Correction Coding for High-Datarate Free-Space Optical Satellite Communication on a Commercial Off-The-Shelf System on Chip

Jan Hildebrand

Thesis for the attainment of the academic degree

Master of Science (M.Sc.)

at the TUM School of Computation, Information and Technology of the Technical University of Munich.

Examiner:

Prof. Dr. sc.techn. Andreas Herkersdorf

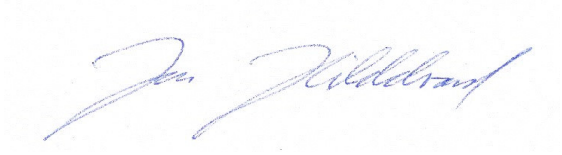
Supervisor:

Dr.-Ing. Thomas Wild
Benjamin Rödiger (DLR)
Anil Morab Vishwanath (DLR)

Submitted:

Munich, 15.10.2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.



Munich, 15.10.2024 Jan Hildebrand

Abstract

Free-space optical communication is a promising technology for the next generation of very-high data rate links between satellites and the Earth as well as between satellites themselves. The OSIRIS program of the German Aerospace Center's (DLR) Institute of Communication and Navigation aims to develop, test, and commercialize optical communication technologies for small satellites used in Earth observation and megaconstellations. Building on the success of DLR's OSIRIS4CubeSat (O4C), the world's smallest laser terminal, DLR is enhancing the O4C terminal to enable bidirectional optical communication links between CubeSats in the CubeISL project.

To improve the optical link performance, forward error correction (FEC) schemes are utilized to correct bit and burst errors introduced e.g. by the atmosphere. Processing of these forward error correction codes however is computationally demanding. While the processing of the FEC coding was performed in advance of a link in O4C, additional near-real-time processing is planned for CubeISL, requiring high throughput FEC coding on the inherently limited computational resources on the CubeSat. This thesis evaluates and optimizes the achievable FEC processing performance on the commercial off-the-shelf System-on-Chip (SoC) that is deployed on the CubeISL satellite. The FEC schemes for encoding and decoding were integrated, profiled and optimized on the SoC. A more than sevenfold increase in throughput is demonstrated, factors that need to be considered for near-real-time processing are discussed and a concept for further improvements is proposed.

Acknowledgements

First and foremost, I would like to thank my family for their ongoing support, without which i would have never been able to present this work.

I cannot fully express my gratitude to my supervisors at the German Aerospace Center (DLR), Benjamin Rödiger and Anil Morab Vishwanath, for their support and guidance. I am very grateful for the time they dedicated to me, even outside of office hours. A special thanks to Olman for his help with some of the uncountable toolchain issues I encountered.

I am also very grateful to Dr.-Ing. Thomas Wild for taking over the thesis supervision and providing immensely useful inputs to the direction of this work.

Lastly i want to thank my best friend Paul for his ongoing unconditional support and all the students at the Institute of Communications and Navigation who provided an irreplaceable working environment: Guillaume, Matti, Iker, Davide, Chiara, Salvatore, Aurora, Matteo, Thomas, Raphael, Luca and all others who contributed to this wonderful time i had at DLR.

Contents

List of Figures

1 Introduction

1.1 Free-Space Optical Communication

1.1.1 History of Free-Space Optical Communication

The concept of using light to communicate over long distances is not new. Humans have used rudimentary tools like fire/smoke to transmit information since the prehistoric times [3]. A prominent record describing the propagation of information via fire is the signalling of the fall of Troy around 1200 BC [4]. Over the centuries, various signaling methods like semaphores evolved to transmit more complex messages using light by improving message encoding and control flow protocols [5]. In 1880, Alexander Graham Bell invented the Photophone, which allowed for the transmission of sound on a beam of sunlight, which was modulated by a tiny mirror [6][7]. Through the invention of artificial electric light sources, the development of simple optical telecommunication systems was possible [8]. The Blinkgerät, a portable device using morse code, for example proved to be an invaluable ground-to-ground and ground-to-air communication device for the German army in the last months of the first world war [9]. The invention of laser (light amplification by stimulated emission of radiation) by Maiman in 1960 revolutionized Free-Space Optical Communication (FSOC) and led to the development of modern long range FSOC systems enabling very high bandwidth communications [10, 3].

1.1.2 Modern FSO communication systems for Space Applications

Only within a short period after the invention of the laser National Aeronautics and Space Administration (NASA) already started to explore the possibilities of using this novel technology for direct to earth (DTE) communications from spacecrafts [11].

Since then there have been huge advances in the development of FSOC systems for space applications. Today space based laser communication can be considered a fairly mature technology that has experienced significant growth in the recent years [12]. This applies not only to demonstration missions but also to operational deployments supporting high data rate DTE communication, optical inter-satellite links (ISLs) and commercial satellite communication networks. The European Data Relay Satellite System (EDRS) for example provides data relay services for low-earth orbit (LEO) satellites via optical ISLs from LEO to geostationary satellites with a datarate of up to up 1.8 Gb/s for commercial and scientific data [13].

Not only data directly related to space missions is routed over satellites, LEO megaconstellation like Starlink are using optical ISLs [14] and are already being used as an alternative to broadband internet access over physical networks in many rural households. LEO megaconstellations have the potential to be a key technology providing global 6G coverage in the future [15].

As data rate demands continue to grow, and the bandwidth of numerous users must be efficiently routed through aggregated links between satellites, communication terminals that support these very high data rates are essential. FSOC is a promising technology for this application.

1.1.3 Advantages of Free-Space Optical Communication systems

FSOC is as a promising alternative to traditional radio frequency (RF) systems to establish high data rate links for satellite communication. FSOC terminals offer several distinct advantages over their RF counterparts, making them an attractive solution for modern satellite networks.

Higher Data Rates One of the most significant advantages of FSOC is its ability to support much higher data rates than RF systems. Optical frequencies, typically in the near-infrared range, have significantly larger available bandwidth compared to RF frequencies. This wider spectrum allows FSOC terminals to achieve data rates on the order of several gigabits per second or even terabits per second, making them ideal for handling the increasing data demands of modern satellite networks [16]. In contrast, RF systems are limited by the relatively narrow spectrum allocations and increasing congestion within RF bands [17].

Smaller Form Factor and Reduced Mass FSOC terminals generally require smaller and lighter components than RF terminals for comparable data rates [18]. The shorter wavelengths used in FSO result in narrower beamwidths. This allows for the use of smaller apertures and optics to achieve the same directionality as larger RF antennas [19]. This reduction in size and mass is particularly beneficial for LEO satellites, where minimizing payload weight and volume is a key factor in optimizing launch costs, especially for mega-constellations.

Reduced Electromagnetic Interference FSOC is inherently immune to electromagnetic interference, as it uses light waves instead of radio waves. This characteristic is especially advantageous in space environments where multiple RF signals can interfere with each other, degrading communication quality [19]. The robustness against interference makes FSO terminals suitable for high-density satellite constellations where numerous links may operate in close proximity. This allows for greater reliability in communication between satellites, as well as between satellites and ground stations.

Increased Security Optical links offer enhanced security compared to RF systems due to their narrow beam divergence. FSO communication involves highly directional laser beams, which are difficult to jam, detect or eavesdrop on unless directly in the path of the beam [19]. This provides a natural layer of security, making FSO systems particularly appealing for secure data transmission in defense, government, and commercial applications.

Spectrum Licensing Advantages Unlike RF communication, which requires licensing and regulation of frequency bands by national and international authorities, FSO communication operates in the unregulated optical spectrum. This removes the need for costly and time-consuming spectrum licensing processes and allows for more flexible deployment of optical terminals on LEO satellites [2].

High Power Efficiency FSO systems can achieve high data rates with relatively low power consumption compared to RF terminals with the same data rate [2]. This is due to the use of narrow optical beams with higher power density. For LEO satellites with limited onboard power resources, this efficiency is a crucial advantage, enabling sustained high-throughput communication without excessive energy consumption.

1.1.4 Challenges of Free-Space Optical Communication systems

While FSOC presents several advantages over traditional RF systems, it also faces significant challenges that can affect its reliability and performance, particularly in dynamic satellite environments. The main challenges for FSOC, especially in LEO satellite networks, include Pointing, Acquisition, and Tracking (PAT), weather conditions such as cloud coverage, and the effects of atmospheric turbulence [20].

Pointing, Acquisition and Tracking FSO communication relies on highly directional laser beams for data transmission, making precise alignment between terminals crucial for maintaining a stable link. In LEO satellite systems, where both the satellites and ground stations or other satellites are in constant motion, achieving and maintaining accurate beam alignment is a significant challenge [2].

Weather and Cloud Coverage FSO communication is highly susceptible to weather conditions, particularly cloud coverage. Unlike RF signals, which can penetrate through clouds and precipitation to a certain level with limited attenuation, optical signals can be completely blocked by dense clouds. Even light cloud cover, fog, or mist can cause severe signal attenuation, reducing link availability and reliability [1].

For direct to earth communication, this presents a significant challenge, as unpredictable weather patterns can lead to intermittent or prolonged communication outages. To overcome this, ISLs play a crucial role in rerouting data to alternative ground stations that are free from cloud coverage. This ability to reroute traffic through other satellites helps mitigate the impact of cloud coverage and improves the overall reliability of FSOC systems in LEO satellite networks.

Atmospheric Effects Atmospheric turbulence is another major challenge for FSOC. As an optical beam passes through the Earth's atmosphere, it encounters varying temperature and pressure gradients, which cause fluctuations in the refractive index of the air. These fluctuations result in a range of effects, including beam wander, beam spread, and intensity fluctuations, collectively known as scintillation. Scintillation can cause rapid changes in the received signal strength, leading to data errors and reduced link quality. [21]

Forward Error Correction for Free-Space Optical Communication To mitigate the effects of atmospheric turbulences causing signal fades of up to several milliseconds, FEC techniques are essential in DTE FSOC systems. FEC allows for the detection and correction of errors introduced by atmospheric fades, ensuring reliable data transmission. By adding redundant information to the transmitted data, FEC enables the receiver to reconstruct the original message without requiring retransmissions.

The lack of an atmosphere in ISLs reduces the occurrence of burst errors. Nonetheless, variations in the pointing of each terminal and inhomogeneous wavefronts can still result in errors. In both DTE and ISL scenarios with high bit error rates, single-bit errors can occur due to low signal-to-noise ratios (SNRs). By correcting these bit and burst errors at the receiver, the required SNR can be significantly lowered, which may lead to reduced transmitter power requirements, smaller antennas, or improved link margins [22] [23].

1.2 OSIRIS Program at DLR

The development of compact and efficient laser communication terminals (LCTs) at the German Aerospace Center began with the OSIRIS program in the early 2010 years, culminating in the development of the world's smallest laser communication terminal for CubeSats, known as O4C. This terminal, which was launched on the CubeL satellite in 2021, demonstrated high data-rate DTE communication with 100 Mb/s [24]. Its compact design, occupying a volume of only 1/3 of a CubeSat unit, combined with a power consumption of below 9 Watt during operation, makes it ideal for LEO satellites requiring high data rates.

Building upon the success of O4C and responding to the growing need for high-speed communication in satellite constellations, DLR is extending the O4C terminal to enable optical ISLs between CubeSats in the CubeISL project. A possible optical ISL scenario is illustrated in Figure 1.1. CubeISL is designed to achieve data rates of 100 Mb/s over distances of up to 1,500 km. Additionally, it is planned to demonstrate an improved channel data rate of 1 Gb/s for the DTE link [2].

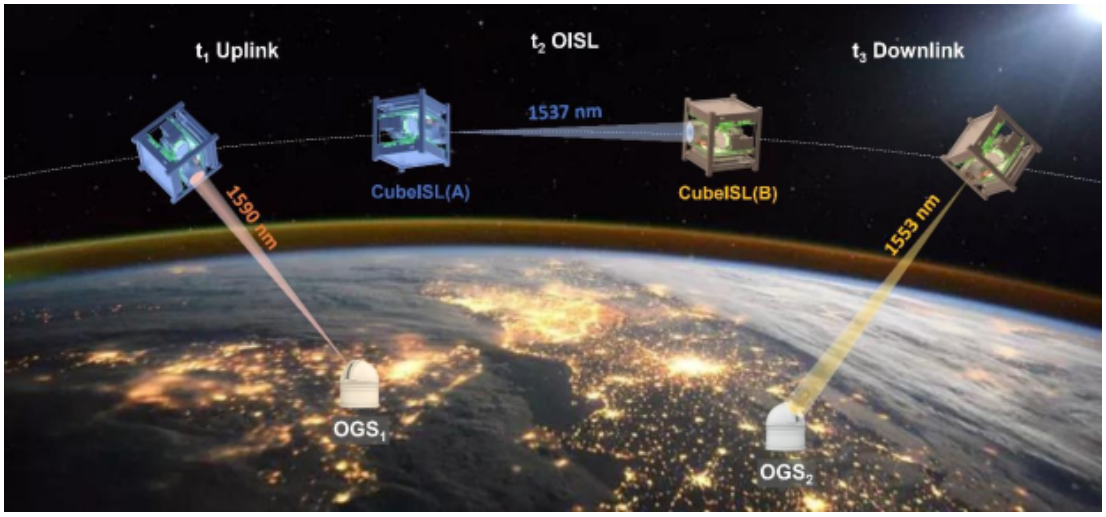


Figure 1.1 "Possible in-orbit use case with two CubeISL terminals, where the terminals are used to relay data between two optical ground stations. It consists of an uplink, optical inter-satellite link, and downlink" [1]. Image Source: DLR [1].

1.3 Forward Error Correction for OSIRIS4CubeSat / CubeISL

The O4C system supports only downlink communication, therefore only encoding of the FEC processing chain is required. This encoding process is performed offline in O4C, meaning that the data is prepared and encoded in advance of the downlink transmission. As a result, the encoding throughput does not need to match the data rate of the optical link. The encoding task in O4C is also not executed on the optical terminal itself, but rather utilizes free CPU computing capacity available on the GomSpace NanoMind Z7000software defined radio (SDR) platform, which is based on a Xilinx Zynq 7030 SoC.

However, when external processing is used, common satellite bus architectures can become a bottleneck at data rates exceeding 100 Mb/s. This is anticipated to be a critical issue for CubeISL, which is designed to support DTE communication links with planned data rates of up to 1 Gb/s [25].

Moreover, CubeISL introduces additional challenges for the data handling tasks as it requires decoding capabilities for its ISL. As decoding is generally more computationally intensive than encoding, additional computing resources become necessary to manage the FEC processing. Especially since CubeISL plans to move towards live encoding and decoding, where data is processed in near real-time during the communication link rather than being pre-encoded/post-decoded [25].

To address these increased processing demands and avoid interface bottlenecks, a dedicated Data Handling Unit (DHU) is incorporated into the CubeISL terminal. The DHU is responsible for managing all channel coding tasks, including both encoding and decoding, controlling the terminal, and interfacing with the satellite for telemetry and command (TM/TC) operations [25].

1.4 Contribution

The central research question of this thesis is to evaluate the achievable processing performance of the in-house developed FEC scheme on a commercial off-the-shelf System-on-Chip. This evaluation is focussing on following aspects:

- CPU-based implementation: Investigating whether the processing can be effectively performed on the CPU of the COTS SoC by reusing and modifying existing software modules. This would streamline the development process and greatly reduce the lead time for integration into the CubeISL system and future optical terminals.
- Real-time processing: Evaluate how the FEC processing can be optimized for real time processing to facilitate the live processing capabilities for CubeISL.
- Identification of bottlenecks: Identify the bottlenecks of the CPU based implementation and analyse possible improvements for future systems.

By analysing these aspects, this thesis contributes to the enhancement of CubeISL's data handling system and future projects of DLR enabling near real-time FEC processing for miniaturized LCTs and improving the overall efficiency of data transmission in satellite communication networks.

2 Literature Review

In this literature review section, we explore comparable FSOC systems, focusing on their approaches to Forward Error Correction (FEC). Additionally, we analyze comparable Radio Frequency (RF) systems to understand their FEC computing architectures.

2.1 Free-Space Optical Communication Systems for CubeSats

The integration of FSOC systems into CubeSats has garnered significant interest in recent years due to the advantages of high data rates, compact designs, and reduced regulatory burdens compared to traditional RF systems. The following section summarizes several key studies and missions in the field of FSOC systems specifically designed for CubeSats, analyses and compares their approach for the computation of FEC to the CubeISL mission.

Although many research institutes and companies are currently working on developing FSOC systems for CubeSats, only few have demonstrated links in orbit.

TeraByte InfraRed Delivery (TBIRD) Mission Launched to low-Earth orbit in May 2022, NASA's 3U sized TBIRD terminal successfully demonstrated laser communication downlinks of 100 Gb/s and 200 Gb/s from a 6U CubeSat to an optical ground station, achieving an error-free transfer of over 1 terabyte of data in a single pass [26]. This mission employs a combination of Reed-Solomon (RS) FEC and an automatic repeat request (ARQ) protocol to ensure reliable data transmission. They utilize transceivers developed for fiber telecommunication networks which handle the forward error correction (FEC) internally. Specifically, the downlink utilizes a RS(223,255) code. Due to the encapsulated processing of the FEC in the transceiver blackbox they do not have the possibility to use interleaving as a mechanism to mitigate atmospheric fading and revert to an ARQ protocol to guarantee error-free data transmission [27]. The FEC processing chain is therefore not comparable to CubeISL.

CubeCat developed by the Dutch Organization for Applied Scientific Research (TNO) and AAC clyde Space [28] is a compact and high-performance LCT specifically designed for CubeSats and small satellites. It facilitates bidirectional space-to-ground communication links, supporting downlink speeds of up to 1 Gb/s and uplink speeds of 200 Kb/s [29]. As the system is developed in alignment with the Consultative Committee for Space Data Systems (CCSDS) O3K coding standard which will be detailed in section 3.1, it features a very similar FEC encoding processing chain to CubeISL consisting of framing, RS Encoder and Channel Interleaver [29]. The underlying compute architecture processing this chain is however not published and the computation seems to be performed in advance of the link.

2.2 Forward Error Correction in RF link architectures on CubeSats

Although not directly comparable to optical channel coding requirements, the more common RF based communication systems for CubeSats can provide an overview of the different possible computing architectures for FEC on CubeSats. Zeedan and Khattab provide an in-depth analysis of the compute architectures utilized in RF communication systems for CubeSats [30]. Although they focus on the whole baseband architecture, FEC is an integral part of many baseband processing chains for satellite communication and the results and learnings of years of development can provide valuable insights on how to design FSOC based systems. They divide the utilized architectures in four categories:

- Custom SDR: Tailored and flexible SDRs designed specifically for the CubeSat mission, allowing reconfiguration and customization.
- Commercial SDR: Pre-built, commercially available SDR platforms adapted for mission needs, offering less customization but faster integration.
- Custom Hardware Design: Purpose-built hardware optimized for performance but lacks post-launch flexibility.
- Commercial Hardware Design: Off-the-shelf hardware components that are cost-effective but offer minimal optimization or flexibility.

As expected hardware based solutions offer the highest data rate for RF systems [30] but can not be considered for FSOC since commercial FEC solutions do not exist yet for CCSDS based CubeSat LCTs. Additionally a custom hardware design, although highly performant, would not only offer too little flexibility in exploring different coding designs for CubeISLs and future systems, but also result in a prolonged development time. This is why we need to look at designs from the custom SDR category which correlate most to the needs of the CubeISL mission.

While many custom SDR systems use Xilinx SoCs similar to the planned SoC for CubeISL, only few rely on the processor(s) of the SoC to perform FEC instead of the FPGA or an external commercial modem [30]. Maheshwarappa et al. for example propose a Custom SDR system for multi-CubeSat communications. They use one of the A9 processors of the Avnet Zedboards Xilinx Zync 7020 FPGA SoC for their FEC chain [31]. The proposed design features a similar FEC processing chain to CubeISL, consisting of an RS Encoder, Scrambler, Viterbi Convolutional Encoder and Interleaver. Similar to our approach their "goal is to keep as much functionality as possible in high level software." [32]. Although their processing chain is comparable, the analysed decoding data rate with a maximum of 19.2 kb/s is magnitudes lower than the 100 Mb/s target of CubeISL for live decoding. This can however not be compared directly as they published no individual performance measurement of the FEC and perform additional tasks such as frequency/time/phase correction and modulation/demodulation on the processor.

Learnings for FSO from RF Even in the more mature field of RF based systems for CubeSats the challenge of developing energy-efficient-high-speed modems remains [30]. Zeedan and Khattab propose, among other RF related improvements, use of FPGAs and improved coding algorithms [30]. The use of FPGA acceleration of the FEC coding is analysed in the outlook of this thesis.

2.3 Gap in the Literature

Although many communication systems (RF and FSO) exist, only few perform their FEC processing on the CPU(s) of a COTS System-on-Chip with high datarates.

In order to close this gap in research, this work investigates such a CPU based coding scheme implementation that would streamline the development process and greatly reduce the lead time for CubeISL and future LCTs developments for small satellites.

3 Coding for FSOC

3.1 Coding Standards for Free Space Optical Communication

In the field of space communications, the need for efficient and reliable data transmission has led to the development of various standards for optical communication.

CCSDS The Consultative Committee for Space Data Systems (CCSDS) communication standards are developed to ensure interoperability between international space agencies, facilitating reliable and high-performance communication links between spacecraft and ground stations. These standards support a wide range of optical communication scenarios, including ISL and DTE links, addressing challenges such as atmospheric disturbances. By setting frameworks for modulation, coding, and synchronization, the CCSDS optical standards enable more efficient data transmission, with significant advancements in high photon efficiency and high data rate communication systems [33]. Notably the DLR Institute of Communications and Navigation (KN) in Oberpfaffenhofen, Germany, has been leading the standardization effort for Optical On/Off Keying within the CCSDS [33].

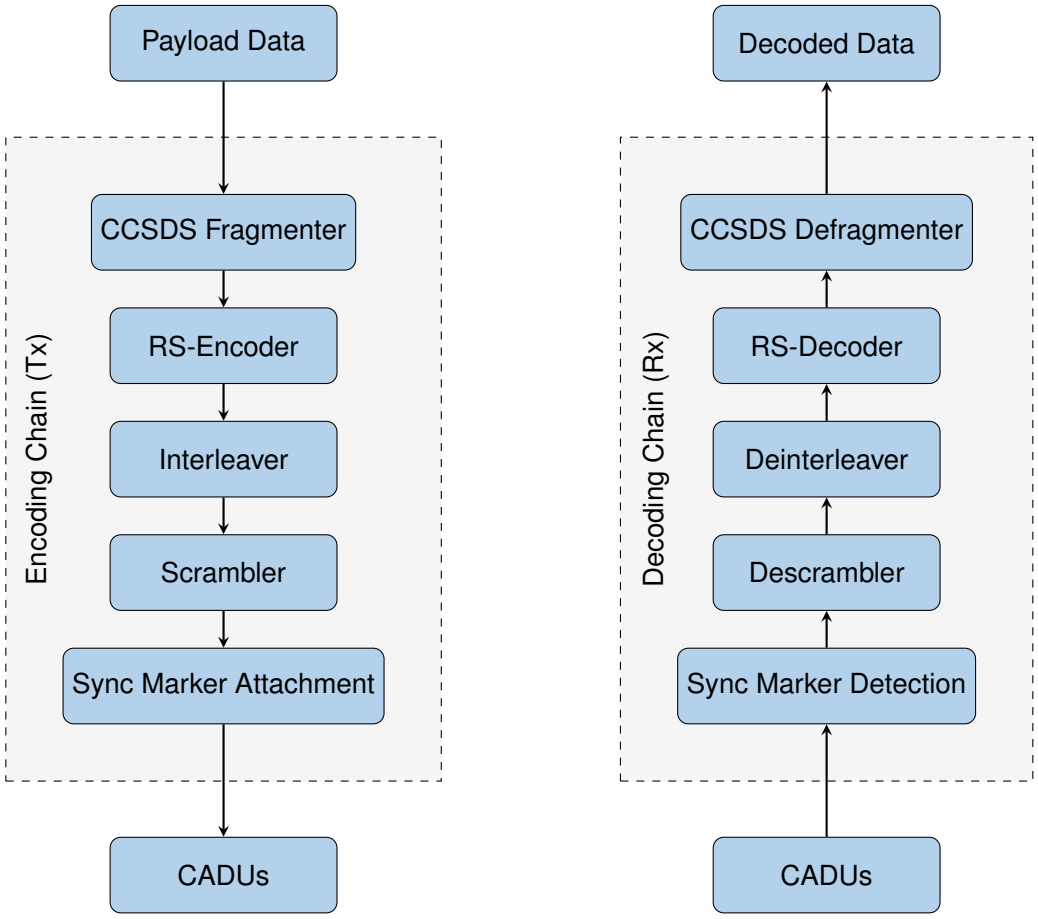
The implemented coding scheme in this work can be considered a preliminary version of the intended second release of the CCSDS Blue Book on Coding and Synchronization which will standardize Optical On/Off Keying modulation (O3K) [34]. Coding and Syncronization for on On/Off Keying has not been standardized in the first issue [35] of the Blue Books Optical CCSDS Standard [34]. A experimental specification for coding and synchronization similar to the used coding scheme in this work can be found in the Orange Book Optical High Data Rate Communication1064nm [36] developed by ESA and DLR which is based on the experience with the EDRS [33].

The Coding and Synchronization protocols described in these standards correspond to the Data Link Layer of the OSI model. They handle the essential tasks of synchronization and channel coding within this layer, ensuring that data are reliably transmitted over an optical space link.

Beside the CCSDS standards the Space Data Association (SDA) standard, and the Enhanced Space Telemetry and Optical Link (ESTOL) standard are significant frameworks that address different aspects of optical communication.

3.2 Existing Encoding Chain

The existing encoding chain developed for the O4C terminal consists of a Fragmenter, Encoder, Interleaver and Scrambler written in C++. These components are described in more detail in the following section in order to analyse and understand challenges associated with computing these functions.



(a) Encoding Chain (TX) **(b)** Decoding Chain (RX)

Figure 3.1 Encoding and Decoding Chains. The segments are described in Section 3.2

3.2.1 Fragmenter

The Fragmenter is responsible to read in the payload data that shall be transmitted over the optical link from the eMMC and package it into so called Transfer Frames, adding necessary headers containing for example frame ID, frame counter, spacecraft ID, etc.

The CCSDS optical communication standards are designed around these Transfer Frames, which differ from packet-based systems like TCP/IP. While packet-based standards typically encapsulate data within small well-defined, fixed structures such as Internet Protocol (IP) packets, the CCSDS Transfer Frames focus on organizing and managing space communication in larger frames more suited for space missions. They are specifically designed to deal with the challenges posed by long distances, limited bandwidth, and high error rates in space communications.

The Transfer frames can vary in size depending on the mission and system requirements. They are structured to carry a wide variety of data (e.g., telemetry, command data, payload data) efficiently. Unlike packet-based systems that often rely on end-to-end connectivity with acknowledgment mechanisms (e.g., TCP/IP), CCSDS Transfer Frames are designed for environments where continuous connectivity cannot be guaranteed and return paths are limited in bandwidth. This is why Transfer Frames are designed to be sent without

expecting an immediate acknowledgment, making it more suitable for high-latency, high-error environments like space.

Since the communication is not packet-based and therefore lacking an immediate retransmission feature for individual frames, the created Transfer Frames need to be passed further through the forward error correction (FEC) coding scheme, ensuring that the data can be reconstructed accurately on the receiver side even if the signal quality is poor.

3.2.2 Reed-Solomon Encoder

After the payload data has been fragmented into CCSDS Transfer Frames it is encoded with by a Reed Solomon RS(n,k) encoder that calculates parity bits which can be used to detect and correct multiple errors on the receiver side depending on the variable code rate $R = (k/n)$. Where k is the number of information (data) symbols and n is the total number of symbols, which includes both the k data symbols and the $n - k$ parity symbols. Up to $t = \frac{n-k}{2}$ corrupted symbols can be corrected.

The data in the Transfer Frames is encoded into codewords of n symbols containing 8 bits each. The resulting data structure can be seen in Figure 3.2.

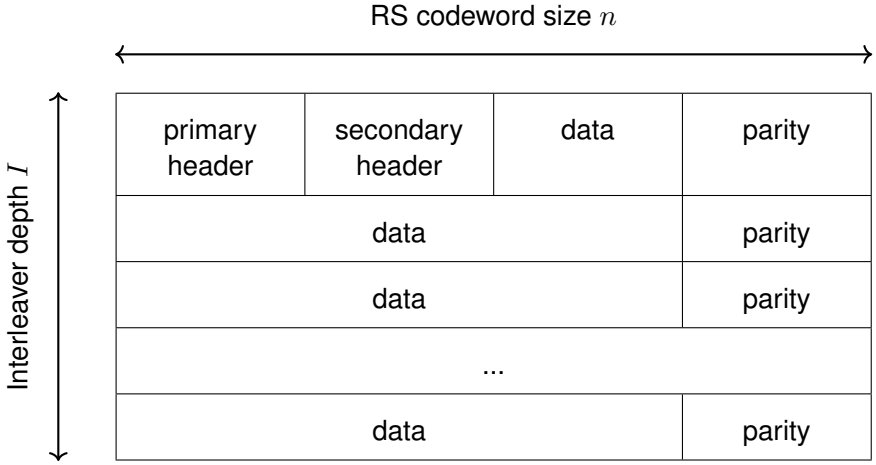


Figure 3.2 Transfer Frame Structure after RS-Encoding

3.2.3 Interleaver

To mitigate burst errors due to atmospheric fading described in paragraph 1.1.4 we need to achieve temporal diversity of the parity information. In order to achieve this, the encoded frames are interleaved as shown in Figure 3.3. The Interleaving is performed by writing the codewords into memory "row-wise" until the configured Interleaver depth I followed by reading and passing the data "column-wise" to the Scrambler which is the next module in the encoding chain. This process spreads out the individual codewords over a larger transmission time and therefore guarantees that only parts of a codeword are lost during a atmospheric signal fade of several milliseconds [37]. The lost parts of a single codeword can then be reconstructed with the parity information by the RS-Decoder when up to $t = \frac{n-k}{2}$ symbols are lost. If the data is not interleaved, several whole codewords would

be lost during a fade causing the loss of a complete Transfer Frame. The number of lost codewords without interleaving can be calculated with (3.1).

$$l = \frac{B * t_{fading}}{n}, \tag{3.1}$$

where B is the bandwidth of the communication channel (in bytes per second), t_{fading} is the duration of atmospheric fading (in seconds) and n is the codeword size (in bytes).

If for example an atmospheric fade of 10ms appears during a 100 Mb/s link 490.2 codewords would be lost.

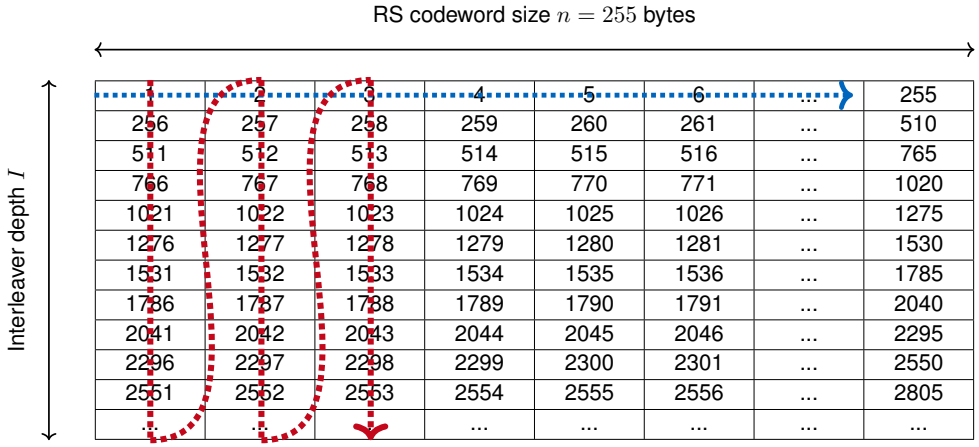


Figure 3.3 Representation of interleaving process of a transfer frame with RS(255,223), 8 bit symbol encoding. Writing into buffer (blue) and reading from buffer (red). The numbers represent the byte position in the buffer.

3.2.4 Scrambler

In CCSDS, there is no line coding, so DC balancing is achieved using a scrambler, which operates at the bit level. The scrambling is performed by XORing the data with a pseudo-random binary sequence (PRBS). In practice the scrambling is performed by generating the PRBS using the polynomial $x^8 + x^7 + x^5 + x^3 + 1$ once to save calculation time. The generated PRBS which repeats after 255 bits is then XORed with the data.

3.2.5 Syncmarker Attachment

The final step in the encoding chain is attaching the SyncMarker. As previously discussed, Transfer Frames for optical communication can become quite large due to the size of the Interleaver. If the sync markers are only added at the start of every Transfer Frame and therefore spaced far apart (e.g., with more than a million bits between markers), it can create synchronization challenges at the receiver. To address this, additional sync symbols within the Transfer Frame can be added.

The sync marker pattern is a fixed 4-byte sequence standardized by CCSDS: 0x1ACFFC1D.

3.2.6 Channel Access Data Units (CADUs) and Downlink

The generated data unit is defined as a channel access data unit (CADU). This CADU is stored in memory for the upcoming link. During the downlink, the CADUs are retrieved from memory, transfered to the FPGA, serialized and transmitted over the channel at a data rate of 100 Mb/s.

3.3 Decoding Chain

The decoding chain for the O4C essentially reverses the processes performed by the above described encoding chain to ensure accurate data reception and error correction over the optical link. Each component in the decoding process corresponds to an inverse function of its encoding counterpart. The chain consists of synchronization marker detection, de-scrambling, de-interleaving, decoding, and defragmenting to retrieve the original payload data.

Upon reception, the first step is to detect the attached synchronization markers. These markers, which were inserted periodically during the encoding process, help the receiver to align the data correctly for further processing.

The next stage involves the descrambler, which reverses the scrambling applied during transmission. In the encoding chain, the data is scrambled by XORing it with a pseudo-random binary sequence (PRBS). This scrambling transforms the data to ensure better transmission characteristics, particularly for achieving DC balance.

When the scrambled data is received, it is descrambled by XORing it again with the same PRBS that was used for scrambling. Due to the properties of XOR, applying the same operation (XOR with the same PRBS) twice results in the recovery of the original data. This process can be explained as follows:

- Let the original data be denoted as D, and the pseudo-random binary sequence as $PRBS$.
- During scrambling, the transmitted data becomes $D \oplus PRBS$, where $\oplus$ represents the XOR operation.
- At the receiver, the scrambled data is XORed again with the same PRBS: $(D \oplus PRBS) \oplus PRBS$.

Since XOR has the property that $A \oplus A = 0$ for any value A, and $D \oplus 0 = D$, the result of the operation is the original data:

$$(D \oplus PRBS) \oplus PRBS = D \oplus (PRBS \oplus PRBS) = D \oplus 0 = D$$

Thus, by applying the same PRBS a second time through XOR, the original data is perfectly restored.

After descrambling, the data undergoes **de-interleaving**. In this step, the interleaved code-words that were spread over the whole frame are reorganized back into their original form to be decoded by the RS decoder.

The **RS decoder** corrects any errors introduced during transmission using the Reed-Solomon (RS) error-correcting code. The parity bits added during encoding enable the

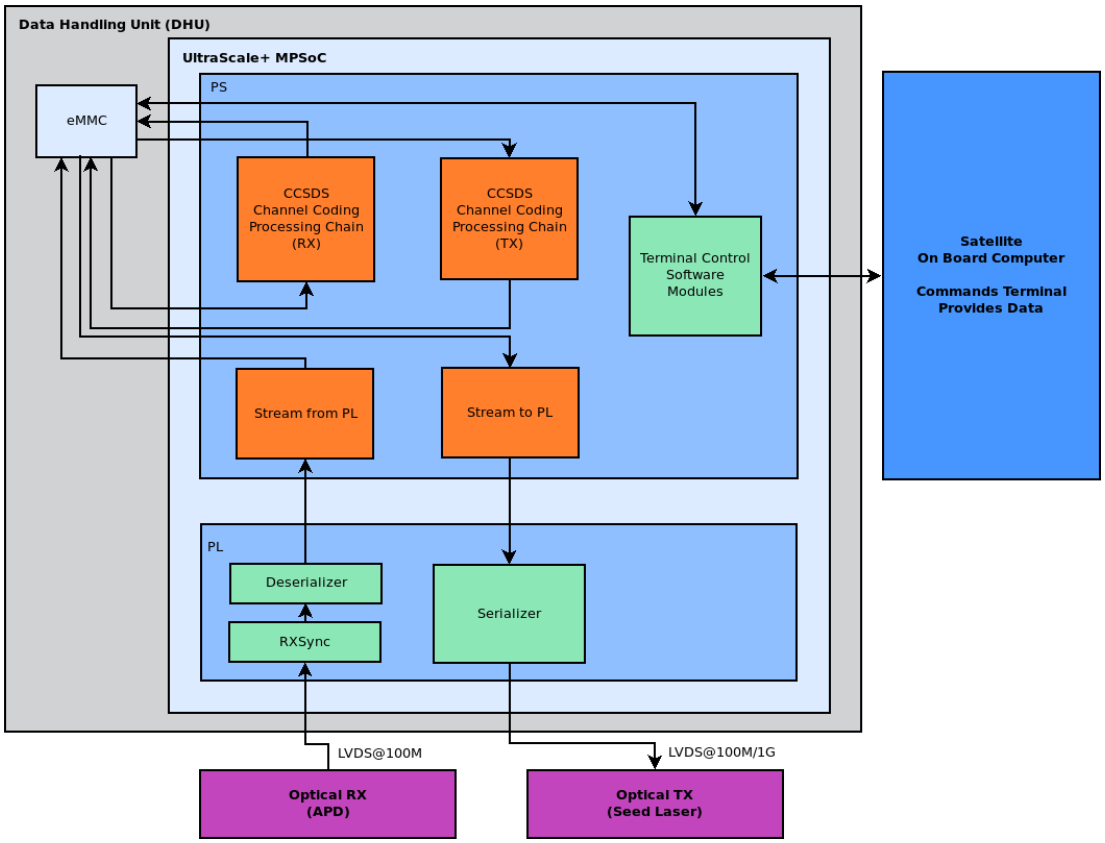


Figure 3.4 DHU Modules. Arrows are indicating dataflow for the offline coding case.

RS decoder to detect and correct errors in the received codewords. The combination of RS coding and Interleaver allows not only to correct single bit-flips but also the described longer burst errors due to atmospheric fading.

Finally, the **defragmenter** reverses the fragmentation process by extracting the payload data from the Transfer Frames. The headers that were added during encoding, such as the frame ID and spacecraft ID, are processed and removed, leaving the original payload data ready for storage in the eMMC.

In contrast to the encoding chain these modules have been implemented only for processing on the Electrical Ground Support Equipment (EGSE) RX [24] and have not been run on embedded hardware.

3.4 Required Changes for CubeISL

This section provides a more detailed description of CubeISL's DHU modules and a comparison to the existing O4C terminal. The comparison will mostly discuss the FEC coding components that are the main focus of this thesis and highlight the changes and developments that shall be conducted in this work.

Encoding/Decoding In the O4C system, the communication is limited to a DTE downlink, where only encoding on the satellite is required. However, in CubeISL an uplink shall

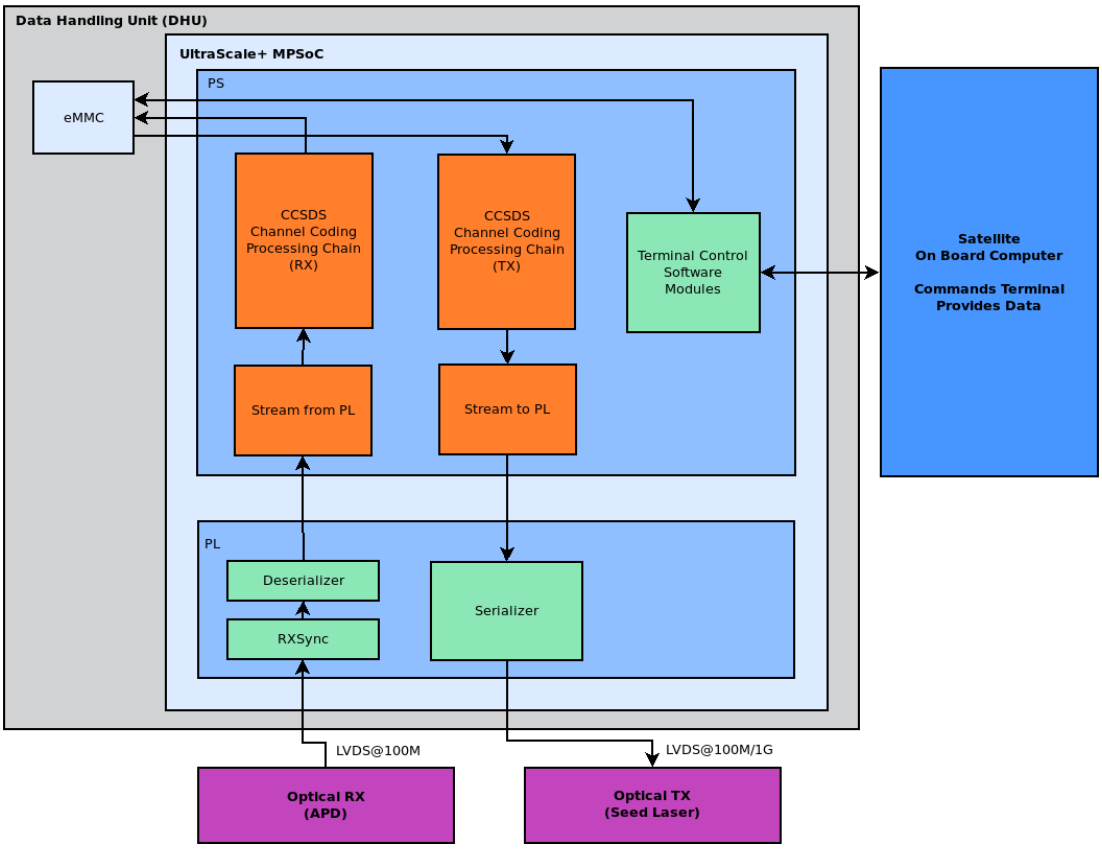


Figure 3.5 DHU Modules. Arrows are indicating dataflow for the live coding case.

be incorporated, which will introduce the need for onboard decoding, a process that is typically more computationally demanding than encoding. Currently, the decoding chain exists only for the ground station, this is adapted and optimized for the execution on the satellite in this work.

3.4.1 Offline vs. Live (Near-Real-Time) FEC Processing

The distinction between offline encoding in O4C and planned live coding in CubeISL is important. These approaches vary significantly in terms of data processing and performance considerations that are explored in this thesis.

Offline Coding In offline coding, all encoding processes are precomputed and stored in advance of transmission. This approach therefore does not require highly performant FEC coding chains because the coding throughput does not need to match the link speed. This is the case for the O4C system. The encoding chain of O4C, is not optimized for computational efficiency, as the preprocessed data is readily available before the optical downlink begins. For CubeISL it is however still highly advantageous to improve the performance of the encoding and decoding chain for the offline scenario. One reason is the difference in data rates: while the O4C system operates with a 100 Mb/s DTE link, CubeISL plans to use a significantly faster 1 Gb/s link. Without optimization, preparing/postprocessing encoded data for a link that is leveraging the maximum possible link duration to a single groundstation will take ten times longer for a 1 Gb/s link than for a 100 Mb/s link with the same FEC

coding chain throughput. Depending on the use case, an increased coding throughput could be highly favourable for scenarios where the timeliness of the data is critical.

Live Coding CubeISL aims to demonstrate near-real-time FEC processing. In the following sections also referred to as live coding. This approach does not necessarily imply strict real-time processing with minimal latency, but rather a system where data is processed during the transmission instead of in advance of a link. Such a system will enable for example applications like live-streaming of camera images captured by the satellite. Additionally, this would offer significant advantages for inter-satellite links, where data exchange between satellites in a network can occur with reduced delays.

Furthermore, CubeISLs goal of a bidirectional ISL necessitates simultaneous encoding and decoding on the satellite. It has to be analysed how fast this can be achieved while maintaining the functionality of the other components executed on CubeISL's DHU.

To achieve this near-real-time processing, FEC coding must be optimized for performance, ensuring that encoding and decoding operations can keep pace with the data throughput demands of live transmission scenarios.

3.4.2 CubeISL Modules and Interfaces to DHU

CubeISL is composed of three primary subsystems shown in Figure 3.6: the optical subsystem based on O4C, the erbium-doped fiber amplifier (EDFA) board, and the DHU. Each subsystem is equipped with its own dedicated mainboard, providing the necessary electronic and digital functionalities for that subsystem. This design adheres to the modular approach, allowing for the independent development, operation, and testing of each subsystem, ensuring that they function separately from one another [25].

The interfaces considering the Encoding and Decoding are towards the Seed Laser to transmit on-off keyed data, the avalanche photodiode (APD) for the receive path and to the on-board computer (OBC) of the satellite to get the data that should be send over the optical link and relay the data that was received over the optical link.

These interfaces are however not considered in the scope of this work. The important interfaces in the context of this work are inside of the DHU: The interface from the processing system (PS) to the programmable logic (PL) and the interface from the PS to the eMMC memory. Figure 3.4 shows the modules and interfaces inside of the DHU for the offline coding scenario whereas Figure 3.5 displays the interfaces for the live coding scenario.

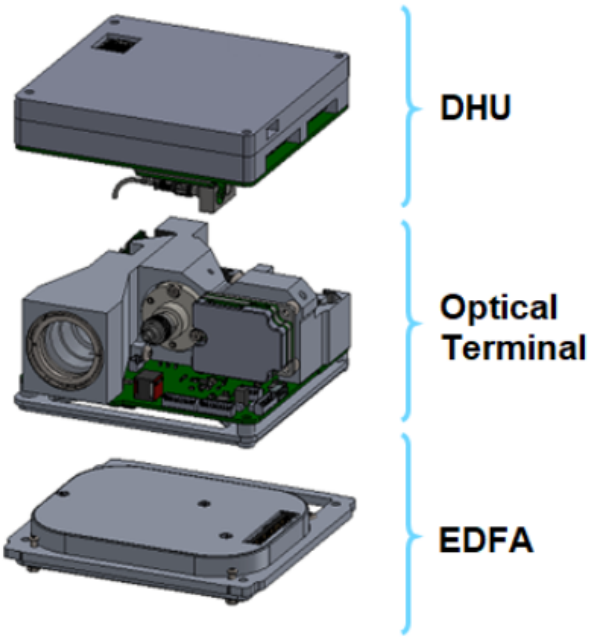


Figure 3.6 System Design of CubeISL. Source: DLR [2].

4 Development Setup and Methodology

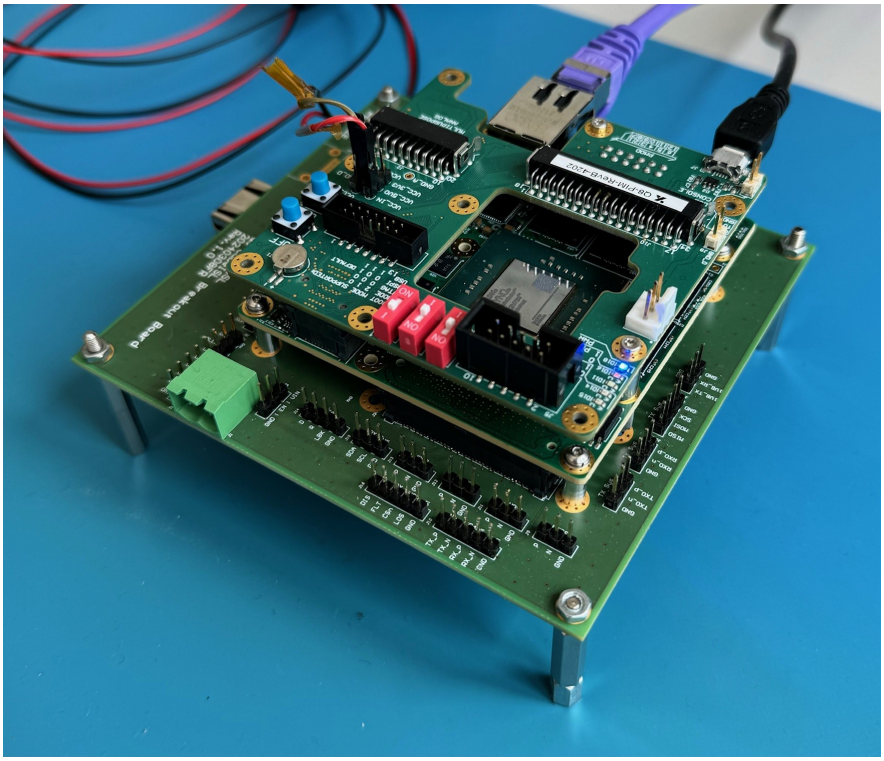


Figure 4.1 DHU Prototyping Setup. Xiphos Q8 (middle). The development header board from Xiphos (top) provides power and a Serial Interface to the Q8 over USB. The prototype breakout board on the bottom provides access to the DHU interfaces. Ethernet cable for remote access (purple).

4.1 Xilinx UltraScale+ SoC - the Hardware for the Data Handling Unit

The DHU for the CubeISL mission utilizes the Xiphos Q8S system, a COTS platform designed to offer scalable, high-performance processing capabilities within the constraints of small satellite missions [38]. The Q8S leverages a Multi-Processor System-on-Chip (MPSoC) architecture, which combines multiple processor cores with integrated programmable logic and various I/O interfaces, providing flexibility and robustness for the mission's data management needs.

At the core of the Q8S system is the Xilinx Zynq UltraScale+ XCZU7EG MPSoC, which features a quad-core ARM Cortex-A53 Application Processing Unit (APU) clocked at up to 1.2 GHz. This APU is responsible for general-purpose computational tasks and supports

the software components of the DHU. The system also includes an ARM Mali-400 GPU, which can operate at speeds up to 600 MHz.

The Q8S is equipped with 4 GB of LPDDR4 DRAM with Error Detection and Correction (EDAC) capabilities, ensuring high reliability and data integrity, crucial for operations in the space environment. This memory is critical for the throughput optimization of software components, as it supports fast and efficient data processing. The system also includes two 256 MB of QSPI NOR Flash memories which are used to store four copies of the bootable image or different images. Additionally, two eMMC storage devices are included in the Q8S, each offering 128 GB of capacity on independent buses with power control features. The dual eMMC setup ensures redundancy and provides ample storage for mission data, including payload data, telemetry data, recorded scientific data and software updates.

In terms of interfacing, the Q8S offers Gigabit Ethernet, USB 2.0 and 3.1 Gen 1, and several general-purpose input/output (GPIO) lines, alongside specialized interfaces such as RS-232/422/485 and CAN bus. These interfaces can facilitate communication between the DHU and other satellite subsystems.

Other notable components include a Field-Programmable Gate Array (FPGA) with 504,000 system logic cells, 461,000 flip-flops, 274,000 lookup tables, and 1,728 DSP slices.

The Q8S systems small form factor and power consumption of 4-25 W (depending of use case) make it ideal for space applications like CubeISL where size, weight, and power efficiency are key considerations. Its robust design, which includes overcurrent protection, error detection, and radiation-hardened components, ensures reliable operation in the harsh environment of space.

With all CPUs stress tested at 100% and transferring a file from one eMMC to the other, i measured a system power consumption of 5.5 W. This is expected to rise further with the implemented FPGA modules.

4.2 Toolchain

In embedded system development, particularly for Xilinx FPGAs and SoC platforms such as the Zynq and Zynq UltraScale+, two primary approaches exist for building a Linux-based system: using the Xilinx-provided tools such as PetaLinux or the open-source Yocto Project. Both options have distinct characteristics, with differences in scope, flexibility, and control over the system configuration.

The Yocto Project is an open-source framework designed to build custom Linux distributions tailored to the specific needs of embedded hardware. It is platform-agnostic, supporting a wide range of architectures beyond Xilinx, including ARM, x86, and PowerPC. Yocto offers extensive customization capabilities, allowing for precise control over the operating system, kernel configuration, and user-space packages. The Yocto build system operates on a layered architecture, where each layer defines metadata, recipes, and configuration for building the software. The layers allow for flexibility in adding or removing software packages based on the needs of the project, ensuring that only the essential components are built into the final image. This modular approach makes it easier to manage system dependencies and version control as well as adhering to the modular development approach of the OSIRIS program. For Xilinx platforms, specific layers such as meta-xilinx

are available to provide support for Xilinx hardware. The primary strength of Yocto lies in its scalability and flexibility, making it suitable for creating minimal, lightweight systems or full-featured distributions depending on the target application. Yocto is highly extensible, capable of integrating custom recipes, which makes it ideal for projects requiring a high degree of control over the entire software stack. However, Yocto comes with a steep learning curve. Configuration and management of the layers can be very complex, and even small configuration mistakes can lead to significant build failures or debugging challenges. In addtion, the build process, which involves compiling all system components from scratch, is time- and resource-consuming for complex designs.

In contrast, Xilinx-provided tools, such as PetaLinux, are designed specifically for Xilinx hardware platforms. PetaLinux abstracts much of the complexity involved in configuring and building Linux systems for Xilinx devices, offering pre-configured kernels, device drivers, and Board Support Packages (BSPs) optimized for rapid deployment. It integrates with the Xilinx Vivado Design Suite for FPGA logic design and hardware-software interaction. However, PetaLinux offers less flexibility compared to Yocto and is primarily suited for projects where a streamlined, Xilinx-focused workflow is required, with limited customization options.

Due to the many advantages and the availability of hardware abstraction layers from Xiphos, the Yocto project is used.

The process began by setting up Yocto to build a custom Linux environment on a development computer. I created a custom Yocto layer for the encoding and decoding software. This custom layer was integrated and configured into a baseline yocto project template provided by Xiphos that already contained the Hardware support layers for the Xilinx and Xiphos specific hardware.

Once the Yocto project was configured with the necessary layers, the build process can be initiated to generate a custom Linux image for the SoC.

The generated image is a tar archive containing:

- The devicetree with a U-Boot header.
- The U-Boot bootloader .elf file.
- The firmware that initializes the hardware components during power-up and provides the necessary instructions to load and start the operating system.
- The compressed kernel image file.
- The UBI image file containing the root filesystem.
- The bitstream for the FPGA configuration.

Transfering and Flashing the Image After the image is successfully built, the next step is to transfer the image to the SoC. This can be accomplished via several ways, e.g. over a Serial connection. However, because the file size of the image is quite large I chose to use the secure copy protocol (SCP) to copy the image to the SoC over a network connection. The prototyping setup for this is shown in Figure 4.1.

In order to initiate this copy to the SoC it is imperative, that the SoC is already booted into a operating system with a running SCP server. The received Q8 was already shipped with a default image that satisfies these requirements, which is booted as soon as power

is supplied to the board. To safely connect the SoC to the network however, the network adapter and other parameters such as a root password need to be setup on this default image. This is done via a serial connection directly to the SoC. The serial port can be accessed over a universal serial bus (USB) to transistor-transistor logic (TTL) device which is integrated in the development board mounted on top of the SoC as seen in Figure 4.1.

Once the network settings as well as ssh certificates are configured correctly, the image can be copied to the eMMC of the SoC via SCP and secure shell (SSH) can be used to access the terminal of the SoC remotely. This enables remote development on the device.

The image can then be flashed onto one of the four possible bootable partitions on QSPI NOR Flash. This is done out of a booted Linux image with custom software provided by Xiphos. This software writes all the files to the device and can upgrade partial or complete images as well as the bitstream.

To boot into the newly flashed partition, the device is booted into the default partion/image by providing power to the SoC. From there it is possible to reboot into one of the other partitions.

These partitions originally exist to provide redundancy in case of corrupt images due to bit flips caused by radiation. During the development they provide the opportunity to store three different images (keeping the default image untouched). Since I have set up the device for remote access it is therefore possible to seamlessly share the device with other developers working on different software components.

4.3 Methodology

After the toolchain has been prepared the integration and optimizations can be performed. This section gives an overview over the main chapters that describe the optimization and analysis process.

The approach for optimizations adopted in this thesis follows a divide-and-conquer strategy, where individual components of the system are analyzed in isolation to identify performance bottlenecks. By first evaluating each subsystem separately, a clear understanding of the system's behavior can be established before integrating and assessing the full performance of the offline and live encoding scenarios.

Encoding and Decoding Chain Optimization (5): The initial step in the analysis was to isolate the encoding and decoding chain, which form the core of the DHU. The process was structured as follows:

- **Integration and Baseline Measurement (5.1, 5.2):** The existing encoding chain was integrated into the Yocto build system to measure its initial performance in the target environment. After the encoding chain was assessed, the same process was performed for the decoding chain to establish its baseline performance.
- **Profiling (5.3):** Profiling tools were implemented within both the encoding and decoding chains to identify the sections that demand the most processing power. This profiling step was essential to understand where the optimization efforts should be focused.

- **Optimization and Measurement (5.4):** Based on the profiling results, optimizations were applied to both the encoding and decoding chains. After each optimization, performance measurements were taken to evaluate the improvements and assess their impact.
- **Parallelization Analysis (5.5):** The possibility of parallelizing the encoding and decoding processes was evaluated to determine if this could further enhance throughput and performance. A thread pool was implemented for both chains to enhance processing efficiency. The parallelization was successfully carried out, and performance measurements confirmed significant improvements in throughput and overall system efficiency.

Performance Analysis of the Whole System (6): Once the encoding and decoding chains were optimized and parallelized, the performance of other key components involved in the offline encoding scenario was assessed. This included evaluating the throughput of the eMMC storage system, as its read/write speeds have a direct impact on system performance. Finally, the encoding and decoding chains were measured in an integrated scenario, where data was read from and written to the eMMC. These measurements provide a comprehensive view of the systems performance in the offline case.

Considerations for the future (7): To optimize the performance further the possibility of accelerating segments of the coding chains with the FPGA or graphics processing unit (GPU) of the SoC is analysed. In addition to the current system, the thesis explores considerations for the planned live encoding and decoding scenario. Key factors analyzed include how the processing resources between the encoder, decoder and other DHU components can be balanced to maintain data flow requirements during live transmissions while ensuring functionality of the other applications that will be running on the DHU.

4.3.1 Performance analysis on development computer

To accelerate the development process, I made the decision to implement and measure optimizations on a development computer first rather than directly on the SoC. Successful optimizations are then cross-compiled for the SoC. This approach offers the significant advantage of a faster compilation and measurement workflow, enabling quicker iterations and evaluations of the implemented changes. However, this strategy does come with a trade-off: the difference in computing architectures between the development computer and theSoC may result in different performance outcomes. The development OS is virtualized on a x86 architecture while the SoC is based on a ARM architecture.

Despite this limitation, the primary goal of using the development computer for performance measurements is to quickly assess the general magnitude of optimization potential rather than obtaining precise performance metrics. This allows for a more efficient exploration of various code modifications, with a focus on identifying promising changes that can later be tested directly on the SoC.

It is important to keep this distinction in mind throughout the thesis, as there will be measurements conducted on the development computer, as well as separate performance evaluations on the SoC itself. This distinction will be highlighted in the relevant sections when discussing performance results. Measurements considering execution directly on the SoC

will be referred to with the keyword target or SoC while measurements on the development computer will be referred to with the keywords development computer or personal computer (PC).

4.3.2 Parametrization of Coding Scheme

For all measurements the same parameters for interleaver depth $I = 3680$, RS codeword symbols $n = 255$ and RS information (data) symbols $k = 223$ are used to be able to compare the measurement results.

5 Optimization of Encoding and Decoding Chain Isolated

5.1 Encoding and Decoding Program structure and initial configuration

In order to understand how to optimize the coding chains, the existing implementation has to be analysed first. The implementation and core program flow is described in this section. This also makes it easier to understand optimizations taken in later sections of this work.

The existing coding chains for the O4C terminal and ground station use a configuration xml file to configure the parameters for the coding chains.

At the start of the execution this file is read containing not only the parameters but all necessary file paths. In the final implementation on the SoC no parameter is given to the program.

Core Functionality of the Encoding Chain

The encoding chain processes an input file and converts it into a series of encoded transfer frames, which are ultimately written into an output file. This process involves several steps, from reading the input data to encoding and assembling the final frames.

Reading Data The encoding process begins by reading the entire input file, which is stored in memory within a vector allocated on the heap. This vector is referred to as the *read-vector* in the following steps, it serves as the source of data for all subsequent operations.

Encoding a Transfer Frame Each transfer frame is created by encoding sections of the data from the read-vector. Although there are slight differences between start, intermediate, and end frames, the explanation below describes the encoding of a single intermediate frame, with distinctions for the start and end frames explained later.

1. **Creating the Write Buffer**: A new vector is allocated on the heap to store the encoded transfer frame, referred to as the write-buffer. This buffer will hold all data for the frame being encoded, including headers and parity information.
2. **Adding Headers**: The primary header is the first component added to the write-buffer. Following this, the secondary header is appended. These headers contain necessary metadata for the transfer frame.
3. **Filling the Write Buffer with Information Symbols**: The write-buffer is extended by $(k - \text{primary header size} - \text{secondary header size})$ bytes, where k represents the

size of the information symbols in one codeword. This amount of data is read from the read-vector and copied into the write-buffer. The bytes now present in the write-buffer are exactly the length of the specified number of information symbols k of one codeword.

4. **Reed-Solomon Encoding**: The write-buffer is then passed by reference to the RS(n,k) Encoder. The RS Encoder extends the write-buffer by $(n - k)$ bytes, calculates the $(n - k)$ parity bytes and appends the parity information to the write-buffer. At this point, the first RS codeword has been successfully encoded and stored in the write-buffer.
5. **Encoding Additional Codewords**: The encoding process repeats for $I - 1$ additional iterations, where I is the interleaver depth. For each iteration:
   - The write-buffer is extended by k bytes.
   - k data bytes are read from the read-vector and appended to the write-buffer.
   - The write-buffer and offset for the new codeword in the write-buffer is passed to the RS Encoder, which calculates and adds the necessary parity bytes.
6. **Interleaving**: Once the RS encoding is completed, the transfer frame undergoes an interleaving step. A new vector is created in the heap, called the interleaver-buffer. Conceptually, if each codeword in the write-buffer is viewed as a row, the data is read column-wise from the write-buffer and written to the interleaver-buffer. This ensures that the data is correctly interleaved across the frame.
7. **Scrambling**: The entire interleaved transfer frame is passed to the Scrambler, which XORs the data in the interleaver-buffer with a pseudorandom binary sequence. This scrambling process helps to mitigate potential patterns in the data.
8. **Inserting Sync Markers**: The sync markers are inserted at the required positions within the interleaver-buffer, ensuring proper alignment and synchronization of the encoded data when it is transmitted or stored.
9. **Creating the CADU**: The fully encoded and scrambled transfer frame, along with its sync markers, is referred to as a channel access data unit (CADU) in CCSDS [36]. The CADU is appended to the write-vector, which holds all the CADUs.

Differences for Start and End Frames

- **Start Frame**: The only distinction for the start frame is that it includes a larger secondary header, which contains additional information compared to the header in an intermediate frame. This extended header accommodates metadata specific to the initialization of the transmission, allowing the receiver to interpret the subsequent frames correctly.
- **End Frame**: In the case of an end frame, the primary difference lies in handling the remaining data in the read-vector. If there is less data left than what would normally fill an entire transfer frame, the frame is padded with zeroes. This padding ensures that the frame size matches the expected length for the encoding process, allowing the RS Encoder to correctly generate the necessary parity bytes. The padded data is treated as part of the information symbols and is encoded in the same manner as described for intermediate frames.

Final Output Once all the transfer frames have been encoded into CADUs, the write-vector containing all the CADUs is written to an output file, completing the encoding process.

5.2 Measurement of Baseline Performance

Once the encoding and decoding chains were successfully integrated on the SoC I took a baseline measurement on the SoC to obtain an approximate estimate of the throughput with randomly generated files.

Methodology

The data was generated by copying data from the linux device file */dev/urandom* with the *dd* low-level utility in Unix as shown in (5.1) for a 10 MiB file. This tool is used as it is provided within BusyBox, a lightweight software suite that provides a collection of Unix utilities in a single executable, designed for minimal environments like embedded systems. BusyBox is therefore often included in our base Yocto image for CubeISL and therefore available on the SoC.

```
$ dd if=/dev/urandom of=random_file.bin bs=1M count=10
```
(5.1)

The execution time of the encoding chain was then measured with the Unix *time* utility which measures execution time of a command as shown in (5.2) also provided by BusyBox.

```
$ time data-interface
```
(5.2)

The *time* utility outputs the user time, system time and real time that elapsed during the execution of the program. To get a baseline measurement of the throughput the elapsed real time, also called wall time, is used to calculate the throughput. Why real time is chosen and a brief description of the different elapsed times is described in further detail in Section 5.3.

The throughput calculation is performed as shown in Equation 5.3 and 5.4:

$$\text{Throughput Enc. [Mb/s]} = \frac{\text{Input Data Size [MiB]} \times 8 \times 1024^2}{10^6} \times \frac{1}{\text{Time [s]}} \tag{5.3}$$

$$\text{Throughput Dec. [Mb/s]} = \frac{\text{Enc. Data Size [MiB]} \times 8 \times 1024^2}{10^6} \times \frac{1}{\text{Time [s]}} \tag{5.4}$$

The input data size is provided in Mebibytes (MiB), where $1\,\text{MiB} = 1024^2$ bytes. The throughput is calculated in Mb/s because this is most commonly used to express and compare throughput between systems.

To measure the encoding and decoding, first a file is encoded and the encoded file is then decoded. Finally the decoded file is compared to the original file by comparing their md5 hashes. This measurement is performed with a 10 MiB file roughly resembling the size of a

high-resolution uncompressed photograph and a 100 MiB file to determine if larger files decrease the throughput. This could be a first indicator if processing data over a longer time period results in a decreased throughput on the system. Repeating the measurements produce highly similar results deviating only slightly from each other. Because this first baseline measurement is only performed to give an approximate throughput the measurement wasn't repeated a large number of times to calculate the average and variance. One Measurement result can be seen in Table 5.1 for the encoding and 5.2 for the decoding.

Baseline Throughput of Unoptimized Coding Chains on SoC

DUT, File Size	User Time [s]	System Time [s]	Real Time [s]	Throughput [Mb/s]
SoC, 10 MiB	11.18	0.08	11.27	7.44
SoC, 100 MiB	110.5	0.87	111.41	7.53

Table 5.1 Encoding Chain. Elapsed Time and Calculated Throughput. Measured on SoC. Baseline Measurement, No Optimizations.

DUT, File Size	User Time [s]	System Time [s]	Real Time [s]	Throughput [Mb/s]
SoC, 10 MiB	13.52	0.12	13.64	6.15
SoC, 100 MiB	125.17	1.11	128.24	6.54

Table 5.2 Decoding Chain. Elapsed Time and Calculated Throughput. Measured on SoC. Baseline Measurement, No Optimizations.

As seen the Throughput is not lower but rather marginally higher for the larger file. The reason for this is analysed in Section **??**. As the measured throughput values are very far from the goal of 100 Mb/s for the CubeISL live coding, optimizations are necessary to get as close to this target as possible. These baseline measurements already provide some first insight into the optimization potential.

The small difference between elapsed real and user times is indicating that the program is spending most of its time running in the user space and isn't frequently waiting on I/O or blocked by other processes. Since system scheduling delays should be minimal due to the very low resource use of the system during idle as seen in listing 5.1 this suggests the program is relatively efficient in terms of avoiding bottlenecks caused by external factors like disk I/O.

The primary focus for first optimizations is therefore on the programs user-space code. A deeper analysis by profiling the functions will provide more detailed insights on where bottlenecks are occurring.

```
Mem: 593328K used, 3456808K free, 17896K shrd, 712K buff, 533620K cached
CPU:   0% usr   0% sys   0% nic  99% idle   0% io   0% irq   0% sirq
Load average: 0.00 0.00 0.00 1/129 3776
```

Listing 5.1 Output of top Utility During Idle of SoC. No measurements and no other user software is running.

5.3 Profiling

To optimize the coding chains and identify potential bottlenecks, i decided to measure the execution time of each function within the chain. Accurate timing enables more targeted optimizations. The selection of an appropriate timing method is important, as it must provide precise measurements without introducing significant overhead that could distort the results.

Measuring Execution Time: Real Time vs. CPU Time

One of the key decisions in this process is how to measure the execution time of each function. When we want to measure the time spent on software components we need to distinguish between *real time* and *CPU time*.

- **Real time** refers to the real-world elapsed time between the start and end of a function's execution. It captures the entire duration, including periods when the program might not be actively using the CPU, such as when waiting for I/O operations or dealing with system-level scheduling delays.
- **CPU time**, on the other hand, refers to the amount of time the CPU actually spends executing the function's instructions. It is further divided into two categories: **user time** and **system time**.
  - **User time** accounts for the time the CPU spends running user-space processes, that is, the actual code of the function or application.
  - **System time** measures the time spent on executing kernel-space processes, such as system calls and other operating system-level operations.

  CPU time does not include any time during which the process is waiting for I/O or being swapped out of the CPU by the operating system.

For the encoding and decoding chain, I chose to use *real time* as the measurement method. The primary reason for this decision is that real time includes important factors such as disk I/O, system-level delays and other loads on the SoC. These affect the overall performance of the system and can have a significant impact on the throughput of the encoding and decoding chains. By measuring real time, it is ensured that all aspects of the functions execution are accounted for, providing a view of the system's performance under real-world conditions.

Method to Measure Real Time for Individual Coding Chain Segments

The measurement of real time is performed using the C++ standard library's high resolution clock, which provides high-precision timing. Several reasons contributed to this choice rather than using dedicated profiling tools like perf:

- **Ease of Use and Portability**: `std::chrono::high_resolution_clock` is part of the C++ standard library, making it highly portable across different platforms without the need for external dependencies or system-specific functions. This ensures that the same timing method can be applied consistently, whether on the development environment or SoC.

- **Unsuitability of perf on Virtual Machines**: While Linux's perf tool offers advanced profiling capabilities, it is not feasible in my development setup due to the limitations of the virtual machine environment. Profiling tools like perf require access to performance counters for most measurements that are not be exposed and supported in my virtualized environment.
- **Low Overhead**: The `std::chrono::high_resolution_clock` incurs very minimal performance overhead, ensuring that the act of profiling does not significantly influence the results. This makes it suitable for in-depth profiling of the system without skewing the performance characteristics of the chain being measured.

Measurement Selection for Encoding Chain

The execution time of several key subfunctions in the encoding chain are measured. This is achieved by saving timestamps before and after each function's execution, and calculating the time difference to obtain the execution duration. These measurements are then output over the standard output stream of the encoding chain. Below is an explanation of the specific subfunctions profiled and the rationale for measuring each one. An example output is given in Listing 5.2.

The overall encoding time provides a high-level view of the performance of the entire encoding process, from reading the file to preparing it for storage or transmission. The throughput value, calculated based on the file size and encoding time, serves as a measure of the systems efficiency, indicating how fast data can be processed through the encoding chain.

The time taken to read the file is crucial for understanding the I/O overhead in the encoding chain. Since file reading often introduces delays, especially with large files, measuring this helps determine if improvements to disk I/O performance could increase the overall encoding throughput.

The preparation of the start, intermediate, and end frames each contain all segments of the encoding chain. Intermediate frames are looked at in more detail, as they typically make up the bulk of the encoding time with larger file sizes. Identifying potential bottlenecks in this part of the process is essential for improving overall efficiency, so further granularity is added in profiling. By breaking down the time measurements of the encoding of intermediate frames into the subfunctions, we gain a deeper understanding of which operations are most time-consuming and where improvements can be made.

The average time and variance of the subfunctions in all intermediate frames is therefore measured for the RS encoder, Interleaver, Scrambler and Syncmarker Attachement. High variance could indicate performance variability, possibly caused by cache misses or memory bottlenecks.

Finally, the time to write the complete encoded data back to a file is measured to determine if I/O writespeeds are a potential bottleneck.

Measurement Selection for Decoding Chain

Similarly to the profiling of the encoding chain, I implemented profiling of the key subfunctions in the decoder.

Profiling of Unoptimized Encoding Chain

Following the implementation of the profiling mechanism, a baseline measurement on the development computer was conducted to assess the current performance of the encoding chain. This allows for the identification of key areas for optimization. The profiling was carried out using files of two examplary sizes: 100 MiB and 10 MiB. The results for the 100 MiB file are detailed in Listing 5.2, while the results for the 10 MiB file are available in the appendix in Listing A.1. Similar to the behavior observed in the baseline measurement on the target platform (Section 5.2), the throughput on the development PC for smaller files was slightly lower, which indicates the encoding process scales differently with file size, which is analyzed later in Section 6.2.

```
Filename: /home/jan/encode/random_data_100M.dat

File size = 100 MiB
Total encoding time = 13271.4 ms
Throughput = 63.21 Mbps

Elapsed time for reading the file: 75.7805 ms
Elapsed time for preparing the start frame: 100.887 ms
Elapsed time for preparing the intermediate frames: 12927.6 ms
Elapsed time for preparing the end frame: 100.85 ms

Elapsed time of subfunctions in preparing n=126 intermediate frames:
- Average elapsed time for encoding: 53.5612 ms
  - Variance for encoding: 13.1182 ms^2
- Deliver: average combined elapsed time of deliver(): 49.0452 ms
  - Interleaving:
    - Average elapsed time: 21.978 ms
    - Variance: 5.6214 ms^2
  - Scrambling:
    - Average elapsed time: 25.8684 ms
    - Variance: 6.21036 ms^2
  - ASM:
    - Average elapsed time: 0.373853 ms
    - Variance: 0.00918 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.82497 ms
    - Variance: 10.9456 ms^2

Elapsed time for writing whole vector to File/Memory: 65.8581 ms
```

Listing 5.2 Profiling Output of Unoptimized Encoding Chain. Running on development computer.

The **total unoptimized encoding throughput of 63 Mb/s on the development computer** is around 10 times faster than measured on the target SoC. Looking at the measurement times within the intermediate frame preparation, the functions that stand out for optimization are the Encoder, Scrambler, and Interleaver. The Encoder function, with an average elapsed time of 53.56 ms per frame, represents the highest portion of the overall processing time. Similarly, the Scrambler, with an average time of 25.87 ms and the Interleaver, with an average time of 21.98 ms, also contribute significantly to the overall duration, as they are repeated for every frame. Optimizations in these functions can contribute significantly to the optimization of total throughput of the encoding chain.

In contrast, functions such as the syncmarker insertion and writing to the new vector contribute minimally to the total encoding time, with both averaging less than 1 ms per frame. As such, these areas are not prioritized for optimization efforts.

Input/output operations, such as reading in the input file and writing the encoded data to a file, appear to have a negligible impact on overall performance. The profiling results indicate that file I/O occupies only about 1% of the total encoding time, independent of the processed file size (as the 10 MiB result shows). This suggests that I/O is not a bottleneck in the encoding chain and the focus of optimization efforts should be put on the computationally intensive functions.

The measured variances indicate a slight difference in execution time for each frame, this could be due to scheduling on the development computer on which a lot of other applications are running. Execution of the profiling on the target device in section 5.5.5 reveals that these slightly high variances in encoding, interleaving and scrambling on the development computer can mostly likely be traced back to scheduling as execution on the target shows very low variances (See Listing A.12 in the appendix).

Profiling of Unoptimized Decoding Chain

Following the profiling of the encoding chain, a similar analysis was performed on the unoptimized decoding chain to evaluate its performance. The decoding chain was profiled using the encoded 100 MiB file produced by the encoding chain. The encoded file has a size of 114.91 MiB, and the total decoding time was measured at 14051.4 ms, yielding a **unoptimized decoding throughput of 68.60 Mb/s on the development computer**.

The profiling output (Listing A.2) shows the execution times for each significant function within the decoding chain. Notably, the RS Decoding function consumes the majority of the time, with an average elapsed time of 66.41 ms per frame. Followed by the deinterleaving and descrambling functions, which recorded average times of 21.87 ms and 15.23 ms, respectively. It has to be noted that this measured descrambling time is already the result of optimization of the scrambler, the reason for this is explained in Subsection 5.4.5.

The overall pattern is consistent with the findings in the encoding chain. Comparing the RS-Encoding and RS-Decoding time, as expected the time to decode is higher than the time to encode.

The Extracting Syncmarker and Extracting Frames functions, with average times of 0.36 ms and 4.83 ms, contribute relatively little to the overall decoding time. Their variances are notably small, except for frame extraction, which has an unusually high variance likely caused by memory access patterns in the frame extraction. However, like in the encoding chain, these components are not as critical to the overall performance, and thus optimization efforts should primarily focus on the core functions that dominate the processing time: the RS-decoding, deinterleaving, and un-Scrambling functions.

5.4 Optimization

This chapter discusses the first optimization steps that were applied based on the profiling results of the unoptimized coding chains. The different optimization iterations are

described and after each optimization batch, profiling measurements are taken to evaluate the improvements. The encoding chain was optimized first followed by the decoding chain.

Overview plots of the yielded performance improvements of the below discussed optimization iterations can be found in:

- Figure 5.1 for the throughput and Figure 5.2 for the profiling of the encoding chain.
- Figure 5.3 for the throughput and Figure 5.4 for the profiling of the decoding chain.

5.4.1 RS Encoder

Since the profiling indicated that the RS-Encoder consumed most processing time, this segment of the encoding chain was addressed first.

The existing implementation of the coding chain uses the libfec library by Phil Karn [39] for RS encoding and decoding. It is based on the Berlekamp-Massey algorithm, already optimized and provides CCSDS conformity without additional modifications.

Switching to a different, possibly faster, algorithm/library would be a possibility, but this shall be avoided because the libfec library is stable and reliable, with a proven track record in O4C. Moreover, it includes support for full CCSDS compliance which is something that most RS libraries for storage applications like the highly optimized Intel ISA-L library [40] don't provide. At the same time libfec can handle different other coderates outside of the CCSDS specification, making it a versatile library for possible experiments with different coding schemes.

The libfec library however contains an optimized CCSDS Encoder and Decoder. I switched to the optimized version of the RS encoder in libfec and performed profiling measurements on the development computer. This adjustment reduced the RS encoding time per frame from around 56 ms to 26 ms, as indicated by the profiling results in Listing A.3. However, with this version only the RS(255, 223) code is supported, which has to be considered for future experiments and links with different coderates.

5.4.2 Optimization of the Scrambler

The Scrambler function was optimized to improve performance by minimizing overhead associated with element access and improving iteration efficiency. Below are the changes made which showed an improvement and an explanation behind each optimization.

In the original implementation, the input data was accessed using the `at()` method within nested loops. The `at()` method, while safe due to bounds checking, adds additional overhead because it performs checks to ensure the index is valid for each access. This method can introduce unnecessary performance penalties, especially within loops that are executed repeatedly.

To optimize this and more, I made the following optimizations to the main part of the scrambler as shown in Listing 5.4 and 5.3:

1. **Use of Iterators Instead of `at()` :**
   Instead of accessing elements using `input.at(i*_n + j)` and `_prbs_sequence.at(j)` , the optimized code uses iterators to traverse the `input` and `_prbs_sequence` vectors. Iterators are more efficient because they directly

reference the memory location of the elements without performing bounds checks. This reduces the overhead in accessing elements repeatedly, particularly within a loop structure.

The `for` loop advances the iterators with each iteration, applying the XOR operation directly to the referenced elements. This optimization simplifies the loop and improves its performance.

2. **Pre-computation of Offset:**
   In the original code, the expression `i * _n` was recomputed for every iteration of the inner loop. In the optimized version, this value is pre-computed once as `offset` before entering the loop. By doing so, we reduce redundant computations and improve the loops efficiency, especially if the interleaver `_depth` grows larger.

```
for(uint16_t i=0; i < _depth; ++i) {
  for (uint8_t j=0; j < _n; ++j) {
    type a = input.at(i*_n + j) ^ _prbs_sequence.at(j);
    input.at(i*_n + j) = a;
  }
}
```

Listing 5.3 Section of Unoptimized Scrambler Source Code

```
for(uint16_t i=0; i < _depth; ++i) {
    uint16_t offset = i * _n;

    // Use iterators to access elements of the vectors, which is more
    efficient then at()
    auto inputIter = input.begin() + offset;
    auto prbsIter = _prbs_sequence.begin();

    for (; prbsIter != _prbs_sequence.end(); ++inputIter, ++prbsIter) {
      *inputIter ^= *prbsIter;
    }
}
```

Listing 5.4 Section of Optimized Scrambler Source Code

These optimizations collectively reduce the computatio time of the Scrambler for each transfer frame from around 26 ms to around 16 ms on the development computer as seen in the profiling output in Listing A.4. This optimization raises the overall throughput of the encoding chain on the development computer to approximately 100 Mbps.

5.4.3 Optimization of the Interleaver

To improve the Interleaver, following described optimizations resulted in a performance increase on the development computer.

1. **Direct Initialization of the Buffer:**
   In the original implementation (shown in Listing 5.5), the buffer `_interleaved_buffer` was initialized using the `resize()` method, which first constructs an empty vector and then resizes it. In the optimized version (shown in

Listing 5.6), the buffer is directly initialized with the required size using the constructor. By directly allocating the necessary memory during initialization, the optimized code reduces the time spent on memory management, particularly for larger buffer sizes.

2. **Removal of `at()` for Buffer Access:**
   In the original code, the `at()` method was used for accessing elements in the `_interleaved_buffer`. As already explained in the Scrambler optimization section (5.4.2), `at()` introduces bounds-checking overhead. As this method is used in a nested loop in the interleaver, the resulting performance penalty is significant. The optimized version accesses elements directly using the `[]` operator, which can be risky in some contexts. The `[]` operator is safe here, given the controlled loop structure and trusted index values.

```
std::vector<type> _interleaved_buffer;
_interleaved_buffer.resize(_length*_depth);

if (_isSet) {
  for (uint16_t i=0; i < _depth; ++i) {
    for (uint16_t j=0; j < _length; ++j)
      _interleaved_buffer.at(j*_depth+i) = buf.at(j+_length*i);
  }
```

Listing 5.5 Section of Unoptimized Interleaver Source Code

```
std::vector<type> _interleaved_buffer(_length*_depth);

if (_isSet) {
  for (uint16_t i=0; i < _depth; ++i) {
    for (uint16_t j=0; j < _length; ++j)
      _interleaved_buffer[j*_depth+i] = buf.at(j+_length*i);
  }
```

Listing 5.6 Section of Optimized Interleaver Source Code

These optimizations collectively decreased the interleaving time per transfer frame from approximately **22 ms** to **15 ms**, as observed in the profiling results in Listing A.5. While the improvements are not drastic, they offer a meaningful reduction in execution time and increase the total throughput of the encoding chain on the development computer to slightly more than **110 Mb/s**.

Replacing the `.at()` method in a shared buffer utility used by the interleaver further decreased the interleaving time per transfer frame to around 6 ms resulting in a total throughput of more than **130 Mb/s** on the development computer as seen in Listing A.6.

5.4.4 Compiler Optimizations

The performance of the encoding and decoding chains can be significantly influenced by the optimization settings applied during the compilation process. The used GNU Compiler Collection (GCC) offers several optimization levels that balance the trade-off between performance and binary size.

GCC Optimization Levels

GCC provides the following optimization levels, each with different impacts on performance and binary size:

- **-O0**: This is the default level, with no optimization applied. The compiler prioritizes fast compilation times and debugging ease, but the resulting binary is unoptimized, leading to lower performance, but the best view for debugging [41].
- **-O1**: At this level, GCC applies basic optimizations to improve runtime performance .e.g. function inlining.
- **-O2**: This level focuses more on performance improvements by enabling most optimization techniques that do not require speed versus size decisions, including e.g. instruction scheduling.
- **-O3**: The highest standard optimization level, `-O3` enables optimizations aimed at extracting maximum performance from the code. It includes all -02 optimizations plus additional ones such as aggressive inlining of functions. This level can result in larger binaries and longer compilation times but is intended to provide the best runtime performance.

Optimization Strategy

Since we are not tightly constrained by binary size in this project, the primary focus is on achieving the best possible runtime performance. To that end, we evaluated the three optimization levels `-O1` , `-O2` , and `-O3` by measuring the performance impact of each on both the encoding and decoding chains.

Importance of Architecture-Specific Optimizations

It is crucial to note that the effectiveness of these compiler optimizations can vary depending on the hardware architecture. The development environment is based on an x86 architecture, while the target device is an ARM-based SoC. These two architectures have significant differences in terms of instruction sets, memory access patterns, and processor design. As a result, the performance gains observed on the x86 development PC might not directly translate to the ARM SoC.

Measurement of Compiler Optimizations

Applying the compiler optimizations on the development computer resulted in a drastic improvement of throughput, as seen in the throughput overview of the optimization iterations in Figure 5.1. The throughput was increased from around **130 Mb/s without compiler optimizations** to over **235 Mb/s with level -O3 compiler optimizations** on the development computer. Where level -O1 already provided the majority of performance increase while levels -O2 and -O3 only resulted in minor changes

The compiler optimizations mainly affected the Scrambling and Interleaving as seen in Figure 5.2 plotting the subfunction times over the optimization iterations. Interleaving time was improved by a factor of 10 while Scrambling time was improved by a factor of more than 25. The RS-Encoder did not noticeably profit. The explanation for this is simple, the

libfec library was added as a seperately compiled Yocto recipe to the image with GCC compiler optimization already set to `-O2` during all measurements. The slight improvement probably originates from the fact that the measurement of the RS-Encoding subfunction measures also the setup and management of calling of the libfec library which was improved by the compiler.

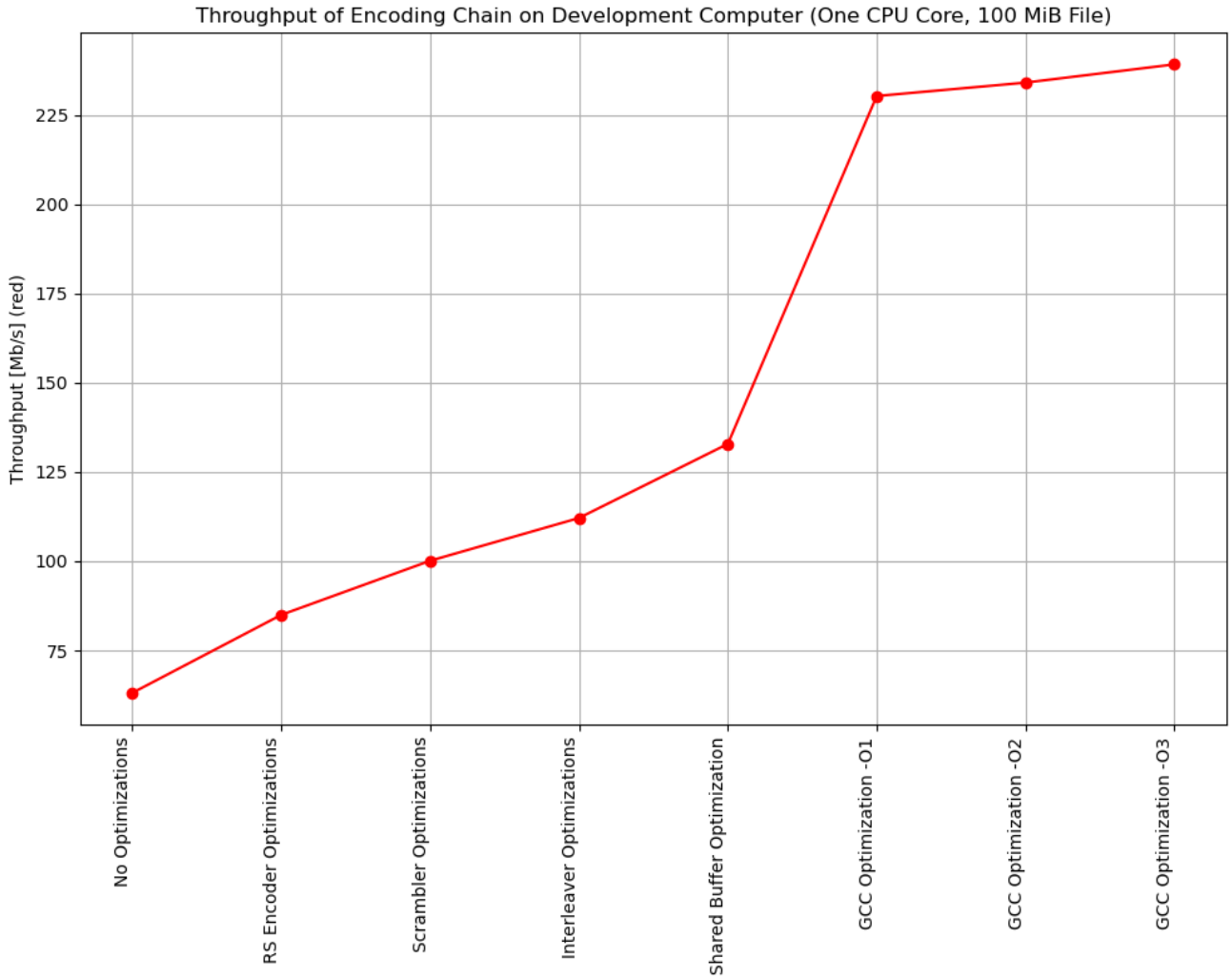


Figure 5.1 Total Throughput of Encoding Chain Optimization Iterations.

5.4.5 Decoding Chain Optimizations

The optimization steps taken for the decoding chain are very similar to the encoding chain, therefore this section only briefly describes the optimizations and focuses on the results. The profiling results for all paragraphs below can be found in the Appendix in Listings A.8 to A.11. Figure 5.4 plots the profiling results of the individual optimization iterations. Similarly Figure 5.3 provides an overview over the decoding chain throughputs of the optimization iterations below.

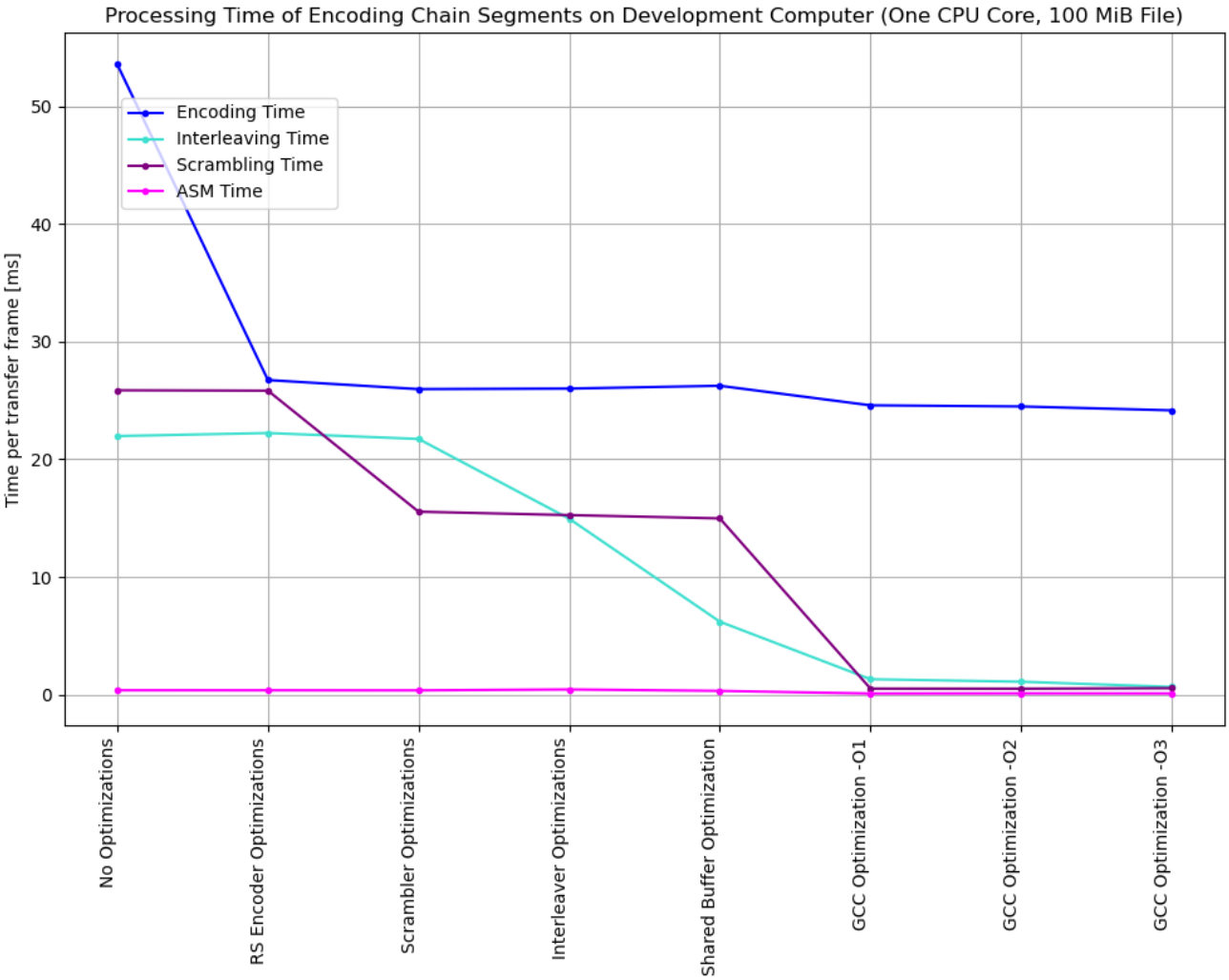


Figure 5.2 Impact of Optimization Iterations on Different Encoding Chain Subfunctions

Shared Buffer Utility Since the buffer utility that is also used in the decoding chain has already been optimized during the interleaver optimization, the impact of this optimization on the decoding chain is analysed first. As seen in Listing A.8 the deinterleaving time has been reduced from around 22 ms to 14 ms which is increasing the total decoding throughput slightly from around 69 Mb/s to 73 Mb/s. Other subfunctions are not affected.

RS-Decoder Optimization Switching to the optimized CCSDS RS-Decoder decreased the decoding time from around 66 ms to 48 ms. Resulting in a total decoding chain throughput of 81 Mb/s.

Deinterleaver Optimization The deinterleaver is almost identical to the interleaver. This means that the same optimizations can be applied. Replacing the `.at()` method to access elements in the `_deinterleaved_buffer` results in a decreased deinterleaving time of only 7 ms instead of 14 ms effectively halving the execution time of the deinterleaver. The total decoding chain throughput is 95 Mb/s after applying these optimizations.

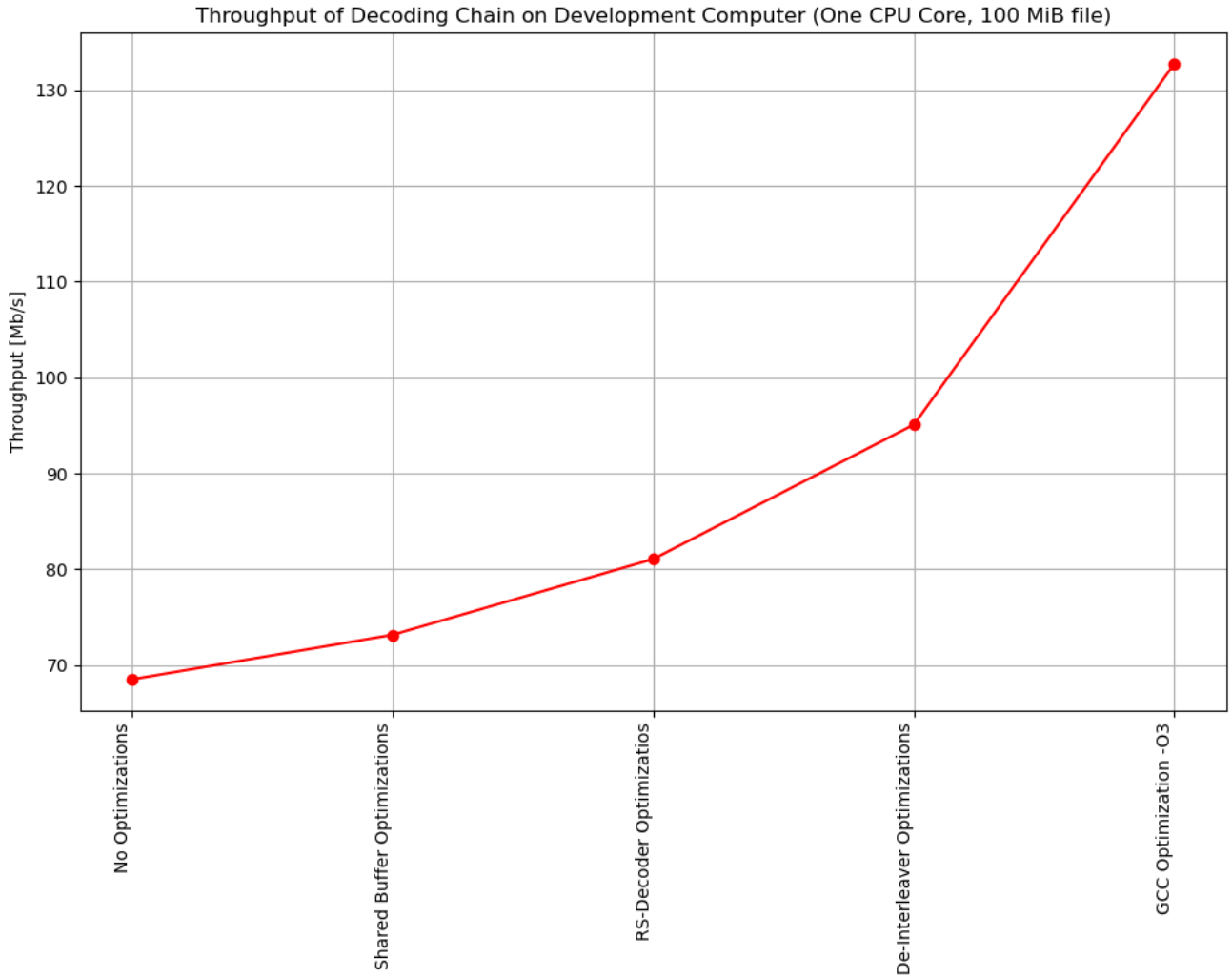


Figure 5.3 Total Throughput of Decoding Chain Optimization Iterations.

Descrambler Because of the intrinsic properties of the scrambler the descrambling is the exact same operation as the scrambling: XORing the data in the buffer. This is why the scrambler function is also used for the descrambling. Since we already optimized the scrambler this optimization is therefore already applied and also present in the unoptimized decoding chain measurements. A repeated analysis of the performance increase is unnecessary. As expected the unoptimized descrambling time of around 15 ms in the decoding chain seen in Listing A.2) is exactly the same as the optimized scrambling time in the encoding chain.

GCC Optimizations Due to the good results with the GCC optimization level -O3 and the similarity of the decoding chain to the encoding chain only the GCC optimization level -O3 was assessed for the decoding chain. Applying the GCC compiler optimizations drastically improved the performance similarly to the observed results on the encoding chain. Again the already optimized scrambler and interleaver are experiencing enormous increases in performance as seen in Figure 5.4. The total throughput of the encoding chain was in-

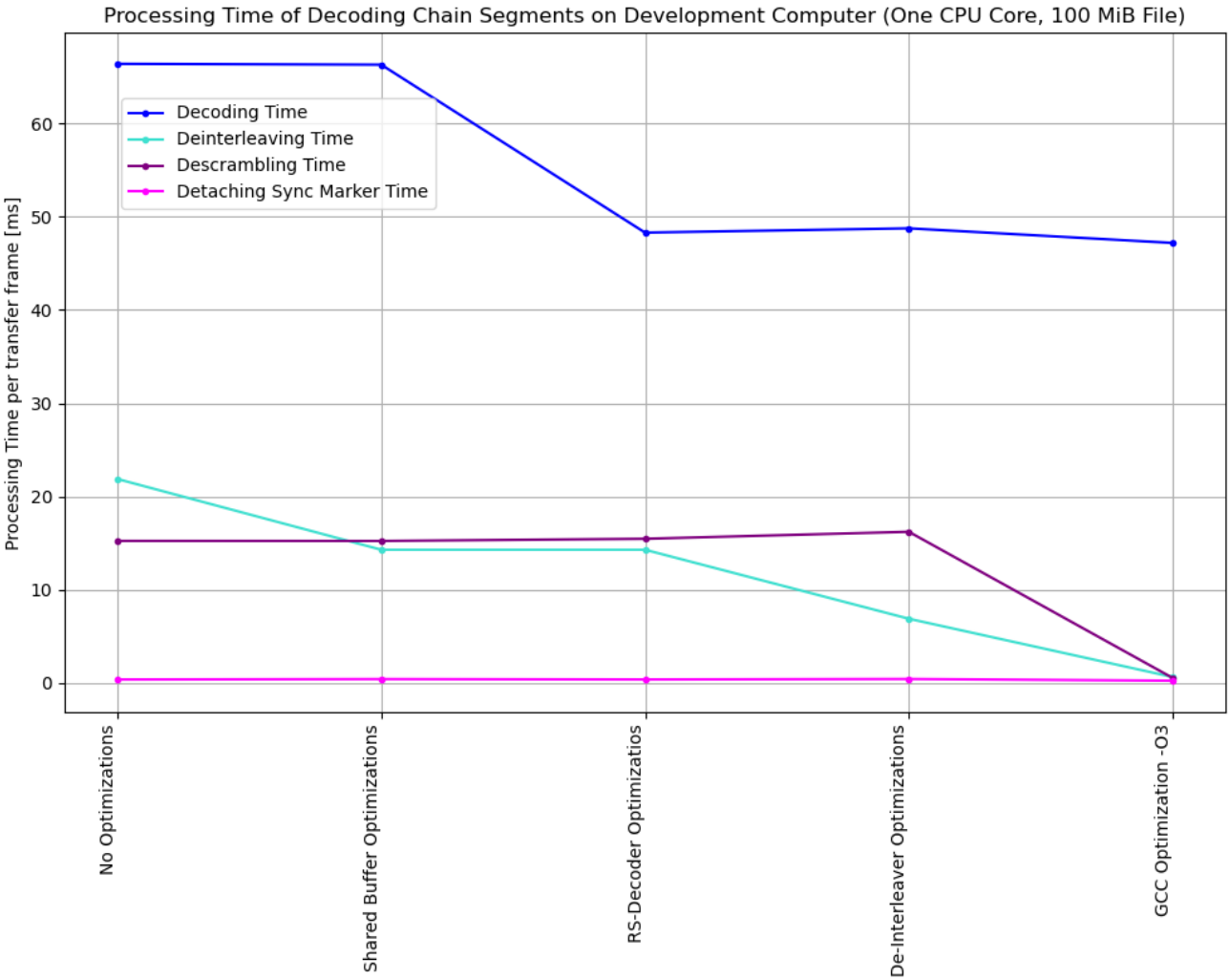


Figure 5.4 Impact of Optimization Iterations on Different Decoding Chain Subfunctions

creased from around **69 Mb/s without compiler optimizations** to over **130 Mb/s with level -O3 compiler optimizations** on the development computer.

5.4.6 Optimization Results on the SoC

The above performed optimizations on the encoding chain and decoding chain were cross-compiled, transferred and executed on the DHU prototype SoC. An overview comparison between baseline throughputs and optimized throughputs can be found in Table 5.3 and 5.4.

The profiling results from running the optimized coding chains on the SoC, despite being approximately ten times slower, closely mirror the outputs obtained from the development computer, demonstrating consistent behavior across both platforms.

With all optimizations a throughput of around **23 Mb/s was measured for the encoding chain**, which is almost four times as much as the baseline throughput of around 7.5 Mb/s. **For the decoding chain an optimized throughput of around 15 Mb/s was measured**, which equates to a 2.4x improvement to the baseline.

The decoding chain is significantly slower than the encoding chain, primarily due to the differences in the speed of RS encoding and decoding within the optimized coding chains. Initially, the encoding and decoding speeds were comparable when using the non-optimized RS functions from the libfec library. However, the encoder experienced a much greater performance improvement than the decoder from switching to the optimized libfec functions. This performance disparity is observed consistently on both the development computer and the SoC.

DUT, File Size	Throughput Unopt. [Mb/s]	Throughput Optim. [Mb/s]	Improvement
PC, 10 MiB	61.21	238.1	3.9x
PC, 100 MiB	63.21	239.4	3.8x
SoC, 10 MiB	7.44	23.25	3.1x
SoC, 100 MiB	7.53	23.15	3.1x

Table 5.3 Encoding Chain Throughput: Baseline vs. all optimizations before parallelization

DUT, File Size	Throughput Unopt. [Mb/s]	Throughput Optim. [Mb/s]	Improvement
PC, 100 MiB	(68.60)	132.9	n.A.
SoC, 10 MiB	6.15	15.0	2.4x
SoC, 100 MiB	6.54	15.8	2.4x

Table 5.4 Decoding Chain Throughput: Baseline vs. all optimizations before parallelization. Exception: Baseline Measurement on development computer already contains descrambler optimizations.

5.4.7 Bottleneck Identification

With the profiling output of the target device we can now determine the bottlenecks of the encoding and decoding chains running on the actual hardware. Looking at the output of the optimized encoding and decoding chains (Listing A.12 and A.13) it is obvious that the RS encoding and decoding times are still the largest contributors to the overall processing time. If we consider start-, end- and 11 intermediate frames, the RS encoding for all frames processes around 3370 ms out of a total processing time of around 3600ms, which is more then 90% of the execution time. A similar relation can be observed in the decoding chain. The RS encoder and decoder are therefore still the key bottlenecks and remain the main focus of the optimization efforts.

Possible Optimization

Since we only use one CPU core out of the four available cores on the SoC an idea is to parallelize the encoder and decoder. Before a detailed analysis is made if the encoder and decoder are parallelizable it needs to be determined if they would benefit from parallelization. If the encoder and decoder are memory bottlenecked parallelization may not lead to significant performance improvements as the threads/cores will spend a lot of time waiting for data from memory.

Analysis: Memory Bound or Instruction Bound

To determine if the CPU is not stalled by memory, while we execute the coding chains, but instead instruction bound, which would benefit from parallelization [42], we can analyse

the Instructions per Cycle (IPC). The IPC metric is a key indicator of how efficiently an application utilizes the CPU. It represents the average number of instructions executed per clock cycle [43].

The ARM A53 processor in our SoC has a 2-wide architecture meaning it can execute up to two instructions per cycle in some cases. Therefore the IPC can range from 0 to 2 for the A53. A high IPC suggests that the CPU is efficiently executing instructions, with minimal delays due to memory access or other stalls. This typically indicates that the application is instruction-bound, where performance is limited by the rate at which the CPU can execute instructions. Since only some instructions can be dual-issued (two instructions executed in one cycle) on the A53 reaching an IPC very close to 2 is unlikely.

On the other hand, a low IPC indicates that the CPU is not fully utilized. In this case, the application may be memory-bound, meaning performance is constrained by memory latency or bandwidth rather than the CPUs ability to execute instructions. When memory access times dominate, the CPU spends significant time waiting for data, resulting in lower IPC values [43].

By using the ARM CoreSight performance counters integrated in the processor, combined with the perf profiling tool, the IPC metric can be analysed for our encoding and decoding chain. This is explained in detail in Section 6.2.1.

The IPC during the execution of the encoding chain on the SoC was measured to be 1.02 for a 10 MiB file which suggests that our application is not memory bound [43]. (IPC measurement shown in Listing A.14)

The IPC measured during the execution of the decoding chain on the SoC was slightly lower but with a value of 0.95 it is still likely for our system that the application is not memory bound. (IPC measurement shown in Listing A.15)

Since the IPC metrics indicate a benefit of parallelizing the RS encoder and decoder, this is further analysed in the following section.

5.5 Parallelization

5.5.1 Analysis of Parallelizability of Encoder and Decoder

Since it has already been determined that the encoder and decoder are very likely not memory-bound, the next logical step in optimizing performance is to analyze if the encoding and decoding can be parallelized. This requires understanding which parts of the code can be executed concurrently without introducing data dependencies or conflicts.

Performing the encoding of a single codeword in parallel is not easily possible as this would require either a modification of the libfec library or a new implementation which is not feasible for the CubeISL development timeframe.

However, there is still an opportunity for parallelization at a higher level [44]. For the Encoder, each transfer frame contains I codewords as seen in Figure 3.2. Since there are no data dependencies between these individual codewords, the encoding process can be parallelized by distributing the encoding of multiple codewords across the available cores of the multicore processor. This allows the simultaneous encoding of several codewords, leveraging the full capacity of the processor to improve overall throughput.

A similar situation applies to the Decoder. Parallelization for the decoder can be achieved on a higher level as each received transfer frame contains I encoded codewords, which are also independent of one another. This independence enables the parallel decoding of multiple codewords across different processor cores.

This concept of dividing a large dataset into smaller chunks, on which the same operation is performed simultaneously by multiple processing units is also called data parallelism [45].

To parallelize the encoder and decoder I implemented a thread pool for data level parallelism of the RS coding that is explained in the next section.

5.5.2 Threading

To understand the concept of thread pools, it first has to be explained what a thread is.

A thread is a sequence of programmed instructions that can be executed independently. In a multi-threaded application, multiple threads run concurrently, sharing the same memory space but performing different tasks in parallel [46]. This approach is particularly beneficial when there are multiple independent tasks that do not depend on each others results, as they can be processed simultaneously on different cores of the CPU.

While threading enables parallelization, manually managing threadssuch as creating, starting, and destroying themcan be inefficient and error-prone, especially in systems where many tasks need to be executed repeatedly. This is where a thread pool becomes useful.

Thread Pools

A thread pool is a group of pre-instantiated threads that are ready to execute tasks. Instead of creating and destroying threads for each task, a thread pool keeps a fixed number of threads running and assigns them tasks from a queue. This reduces the overhead associated with frequent thread creation and destruction, and ensures better use of system resources by keeping threads busy.

In a thread pool, tasks are placed in a queue, and available threads pull tasks from the queue to execute them. When a task is completed, the thread becomes available to handle another task, making the process efficient and scalable. The use of a thread pool is particularly beneficial in applications where there are many short, independent tasks that can be executed in parallel.

One major advantage of using a thread pool is the reduction of overhead. Instead of creating and destroying threads for each task, which can be costly in terms of system resources, the thread pool maintains a fixed number of threads that are reused throughout the process. This avoids the performance penalty associated with frequent thread management and leads to more efficient execution.

Implementation of Threadpool

In the context of the encoding and decoding chain, a thread pool allows multiple codewords from a transfer frame to be processed concurrently. All codewords of a transfer frame

are added as tasks to the thread pool queue and are processed as soon as a thread is available.

A simple thread pool was implemented for this application using the thread class from the C++ standard library [47]. In this implementation, the thread pool is initialized with a specified number of worker threads that continuously fetch tasks from a shared queue. Tasks, such as encoding or decoding individual codewords, are stored as std::function<void()> objects, allowing for flexible task handling. The queue is protected by a mutex to ensure safe access, and a condition variable is used to notify threads when tasks are available. When the thread pool is shut down, all threads are properly joined to ensure a clean exit.

To accomodate parallelization of the encoding, the encoding chain structure as explained in Section 5.1 had to be changed.

The original implementation processed each of the I codewords in a transfer frame sequentially in following manner:

- The write-buffer is extended by k bytes.
- k data bytes are read from the read-vector and appended to the write-buffer.
- The write-buffer and offset for the new codeword in the write-buffer is passed to the RS Encoder, which calculates and adds the necessary parity bytes.

The encoding is not parallelizable in this form. To parallelize it, the dataflow was changed in the following way:

- The write-buffer is filled with all data bytes of one transfer frame first, leaving empty fields after each $n - k$ bytes for the k parity bytes of each codeword.
- A new thread pool is created by instantiating the implemented `ThreadPool` class. Directly passing the number of requested worker threads to the constructor.
- For each of the I codewords in the transfer frame a encoding task calling the RS Encoder is added to the end of the thread pool queue.
- The worker threads continuously fetch tasks from the shared thread pool queue.
- The main application waits for all tasks in the threadpool que to be completed.
- As soon as all codewords have been encoded by the worker threads, the ThreadPool is destructed and the transfer frame is passed to the interleaver.

5.5.3 Parallelization Results of Encoding

After implementation of the thread pool for the RS encoding, the encoding chain was executed and profiled on the development computer.

Thread Pool Overhead

To assess the overhead introduced by the thread pool, we can simply compare the performance of these two implementations:

- Optimized encoding chain with all optimizations from Section 5.4 without the thread pool implementation.

- Optimized encoding chain with all optimizations from Section 5.4 with the thread pool implementation for the RS encoding but only one thread allocated to the thread pool.

Allocating only one thread to the thread pool essentially confines the RS encoding to a single CPU core, as in the original setup. However, the overhead introduced by the thread pool implementation is still affecting the performance, enabling us to measure the impact of the overhead.

As seen in Figure 5.5 the overhead of the thread pool increases the time it takes to RS encode a single transfer frame from 24 ms to 31 ms. This leads to a throughput decrease from 239 Mb/s to 189 Mb/s.

Results

Increasing the number of worker threads in the thread pool highlights the benefits of parallelization. Doubling the threads from one to two nearly halved the RS encoding time per transfer frame from 31 ms to 16 ms, resulting in a throughput of 325 Mbps on the development computer. Further increasing the threads to three reduced the RS encoding time to 12.5 ms, achieving a throughput of **385 Mbps on the development machine**. However, the improvement from two to three threads was less significant, suggesting that the RS encoding may be becoming memory-bound on the development system.

Allocating more threads than physical CPU cores to the pool might benefit the performance if threads are waiting for I/O operations. By allocating more threads than cores, the system can continue to process other tasks while waiting for I/O operations to complete, improving overall throughput. If the RS coding threadpool is I/O bound, this would not provide a benefit as all threads would be waiting for the same memory resource. We therefore expect no additional improvement when more threads than the available three cores on the development computer are added to the pool.

Measurements show that when the number of threads in the thread pool exceeded the three available cores, throughput began to decrease notably. This degradation could be caused by increased context switching, meaning that the CPU spends more time switching between threads than performing useful work.

5.5.4 Parallelization of Decoding

Similar to the encoding chain, the decoding chain was parallelized by using a thread pool for RS decoding. The decoder did not require any changes to buffer access patterns, allowing the codewords of a transfer frame to be processed concurrently without large modifications, as seen in the Listings A.16 and A.17 in the Appendix showing the code for the implementation of the RS Decoding without and with the thread pool.

Thread Pool Overhead

The overhead for the decoding chain was measured in the same manner as for the encoding chain. This resulted in an increase of 10 ms in the RS decoding time per transfer frame, bringing it to 57 ms.

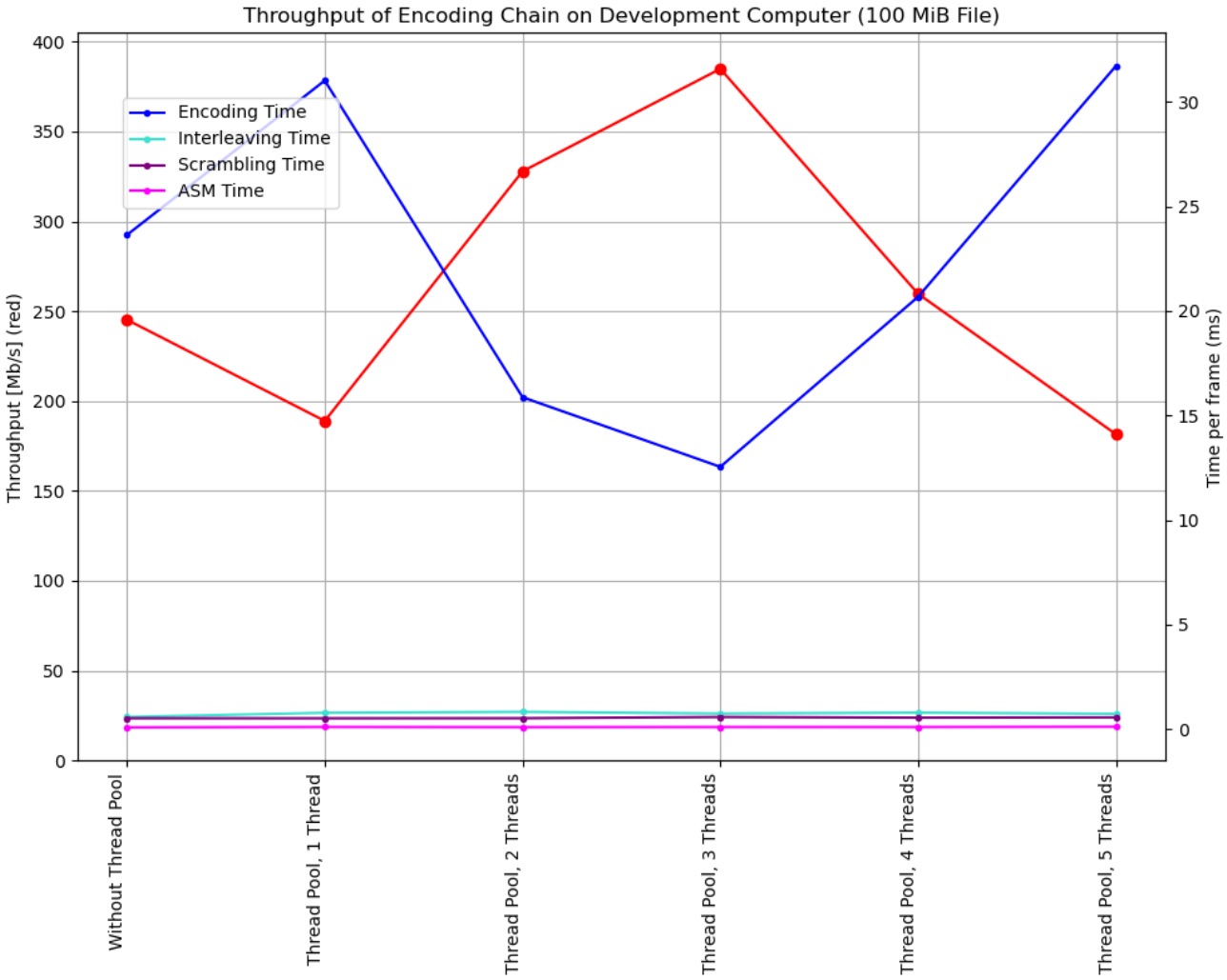


Figure 5.5 Throughput and Profiling of Encoding Chain. RS Encoding Parallelized.

Results

As shown in Figure 5.6, increasing the thread count produced similar results in throughput to those observed for the RS encoder. Increasing from one to two threads nearly halved the RS decoding time per transfer frame, reducing it from 57 ms to 29 ms. The step from two to three threads provided further improvement, though to a lesser extent, again suggesting that the RS decoding may be becoming slightly memory-bound. Using three threads resulted in a throughput improvement from around 130 Mb/s unparallelized to **275 Mb/s** with RS decoding parallelization on the development machine. Adding more threads than the three available cores to the pool again resulted in no improvement of performance.

5.5.5 Results on SoC

After transferring a newly created Yocto image containing all necessary additional libraries for the multithreaded implementation to the SoC, the performance impact of the parallelization of the RS coding was measured in the same way as on the development computer.

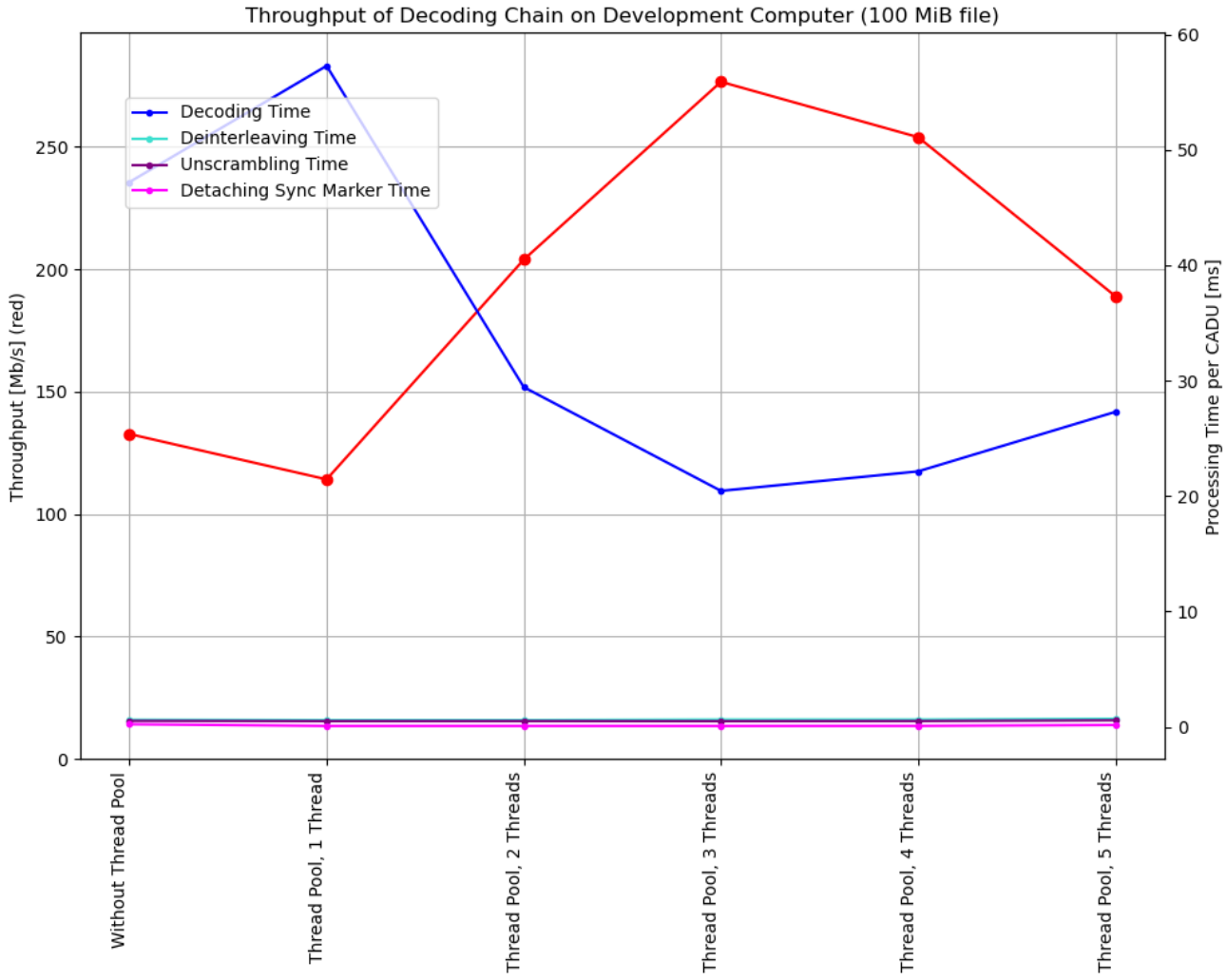


Figure 5.6 Throughput and Profiling of Decoding Chain. RS Decoding Parallelized.

As seen in Figure 5.7 for the encoding and Figure 5.8 for the decoding the results on the SoC are very similar to the results observed on the development computer.

Analysing the threading overhead as explained in Section 5.5.3 however no significant overhead of the threading implementation could be observed.

Adding more threads than the four available CPU cores on the SoC shows no drastic decrease in performance like observed on the development computer. This is suggesting that the significant decrease in performance on the development computer when adding more threads than cores was caused by a high cost of context switching in the virtual machine/hypervisor.

5.5.6 Conclusion of RS Coding Parallelization

The improvements of parallelizing the encoding and decoding are significant. A maximum throughput of **385 Mb/s for the encoding chain** and **275 Mb/s for the decoding chain** was measured on the development machine. On the SoC a maximum throughput of **63.3 Mb/s for the encoding chain** and **47.6 Mb/s for the decoding chain** was measured. An

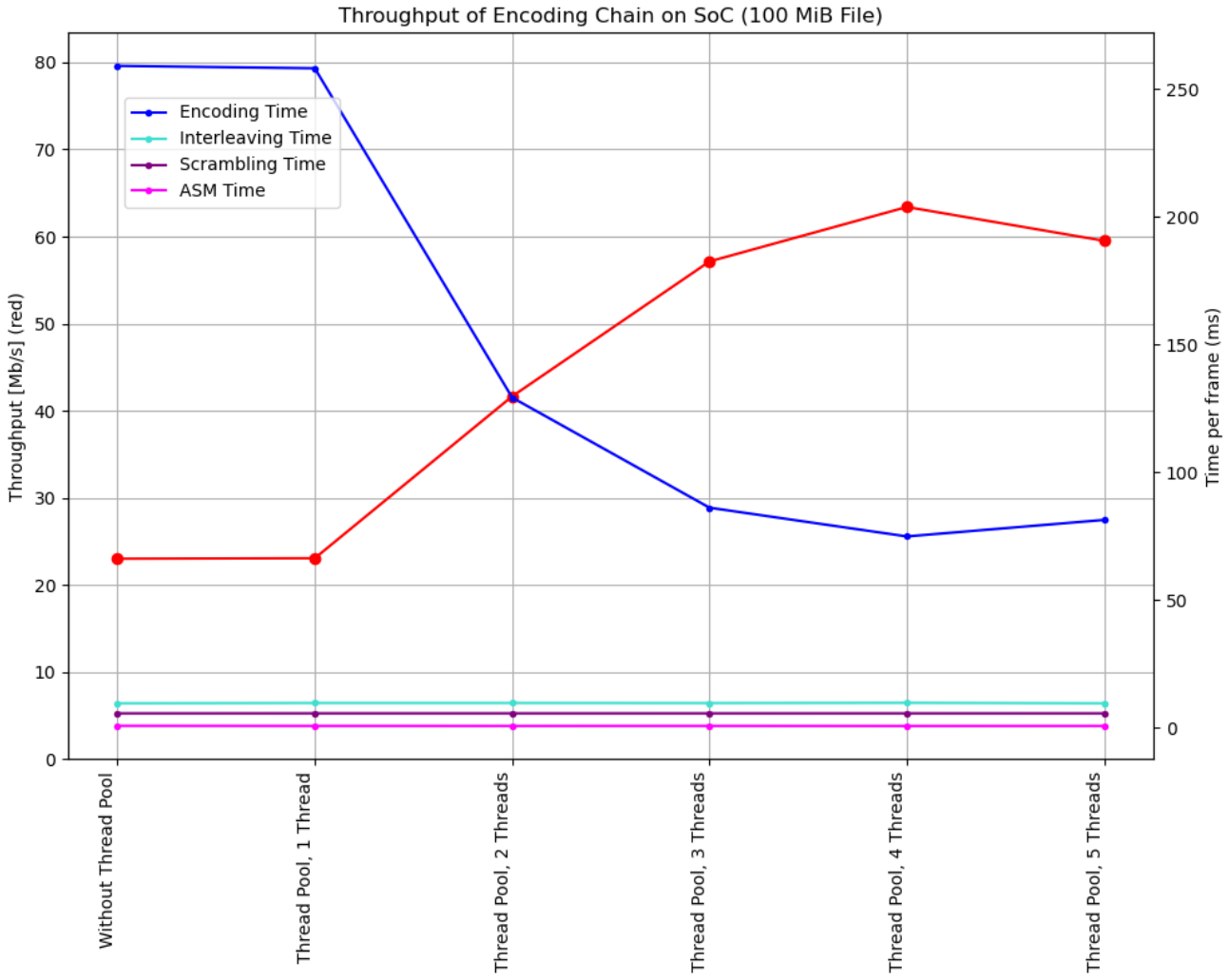


Figure 5.7 Throughput and Profiling of Encoding Chain on SoC. RS Encoding Parallelized.

overview of the improvements so far can be found in Table 5.5 for the encoding chain and Table 5.6 for the decoding chain.

DUT, File Size	Unoptimzed [Mb/s]	Optimized [Mb/s]	Parallelized [Mb/s]	Total Improvement
PC, 100 MiB	63.21	239.4	384.8	6.1x
SoC, 100 MiB	7.53	23.15	63.4	8.4x

Table 5.5 Encoding Chain Throughput Performance Improvements

DUT, File Size	Unoptimzed [Mb/s]	Optimized [Mb/s]	Parallelized [Mb/s]	Total Improvement
PC, 100 MiB	(68.60)	132.9	276.5	n.A.
SoC, 100 MiB	6.54	15.8	47.6	7.3x

Table 5.6 Decoding Chain Throughput Performance Improvements. Exception: Baseline Measurement on development computer already contains descrambler optimizations.

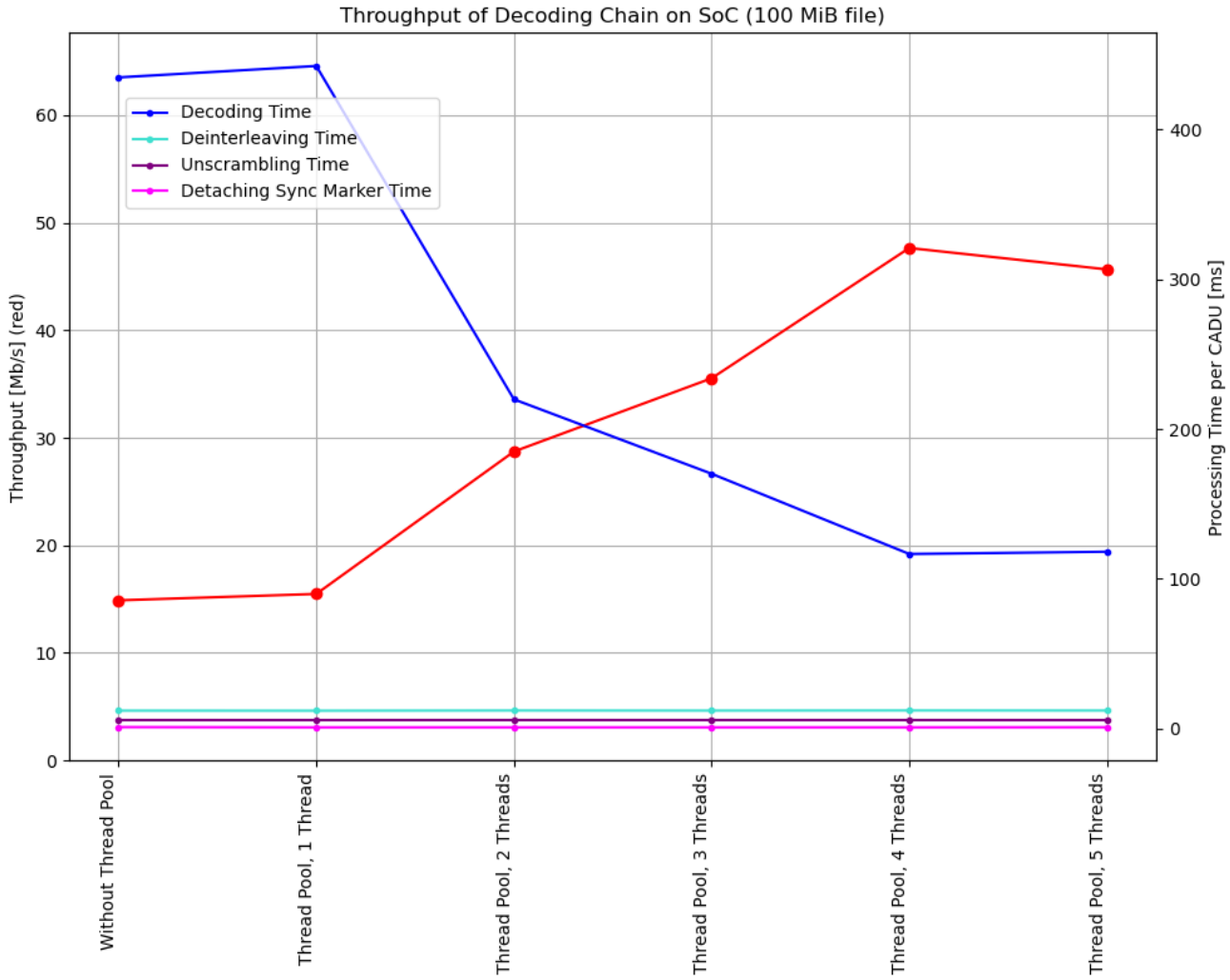


Figure 5.8 Throughput and Profiling of Decoding Chain on SoC. RS Decoding Parallelized.

In the next chapter the Encoding and Decoding Chain are tested on the SoC in the same way as they will be deployed on the CubeISL satellite, reading and writing data from/to the external eMMC memory.

6 Performance Analysis of the Whole System

After having analysed and optimized the processing of the coding chains individually, the coding shall be tested as close as possible to the final implementation of the system. A short explanation of the data flow for the offline and live coding scenario is given in the following paragraphs.

Offline Scenario For the offline scenario we need to distinguish between the encoding/decoding process and the transmission/reception process that happen at different times:

For the encoding/decoding process the user data that is supposed to be processed is already stored in eMMC memory. During encoding the data will be read from the eMMC encoded by the encoding chain and written back to eMMC memory. The encoding data is then ready for the transmission over the optical link. The same process applies to the decoding process. The received encoded data has been stored in the eMMC during the optical link. After the link the encoded data is read from the eMMC by the decoding chain. It then stores the decoded data back into the eMMC. The encoding and decoding in the offline scenario do not need to be performed at the same time.

For the transmission of data over the optical link, the prepared encoded data is streamed from the eMMCis to the PL, that relays it over low-voltage differential signaling (LVDS) to the Seed Laser as shown in Figure 3.4.

For the reception of data over the optical link, the received encoded data from the APD is simply streamed from the FPGA to the the eMMC memory where it is waiting to be decoded after the link.

Live Scenario For the future live coding scenario, the coding and transmitting/receiving will happen simultaneously: For the transmission of data over the optical link, the user data will be either fetched from the eMMC or directly received from the satellite by the encoding chain. The encoded data is then forwarded by the encoding chain to the PL which will further process the data and transmit it over LVDS to the Seed Laser as shown in Figure 3.5.

Inversely for the receiving of data over the optical link, the received encoded data will be streamed from the FPGA to the decoding chain with a buffer in between which is protecting against short drops in data rate of the decoder. The decoder then stores the data in eMMC memory or directly relays it to the satellite.

6.1 Offline Coding System Test

As the live coding scenario is a secondary goal for the mission and has not been completely implemented yet, only the offline scenario can be tested. For this the throughput of the interfaces is briefly discussed followed by the test of the complete system for offline encoding/decoding.

6.1.1 Throughput of Interfaces

For the offline coding scenario a constant read speed of 1 Gb/s has to be supported from the interface of the eMMC memory for data transmission. Similarly for receiving data, a constant write speed of 100 Mb/s + 15 Mb/s (for header and parity overhead of the encoded data) has to be supported by the interface to eMMC memory.

For the future live coding scenario a simultaneous read speed of 100 Mb/s and write speed of 100 Mb/s + 15 Mb/s overhead has to be supported by the interface to eMMC memory.

According to the Q8 user manual, the systems two eMMCs are supposed to provide read and write access speed of over 50 MB/s each, satisfying the requirement for live coding with a large margin.

For the offline transmission scenario a speedtest has already been conducted by Xiphos to confirm the required datarate from the eMMC to the FPGA.

6.2 Offline Coding Encoding/Decoding System Test

To test the coding chains combined with the eMMC memory on the SoC following test was conducted, to determine the consistency of the encoding/decoding throughput:

A random file with 10 MiB is created on eMMC0 with the `dd` utility (as in 5.1). This file is encoded by the optimized and parallelized encoding chain with four threads and stored in eMMC1. This encoded file is then decoded with the optimized and parallelized decoding chain with four threads which stores the decoded file in eMMC0. The encoding and decoding times including the reading and writing to eMMC are measured and stored. This process is repeated several times.

This process was also done for 100 MiB file sizes. The calculated average throughput and standard deviation can be found in Table 6.1. The observed throughputs are fairly constant with a standard deviation of less then 1 Mb/s. To detect outliers, the results were plotted in Figure 6.1. However it can be observed that the throughput for 10 MiB files is significantly lower than for 100 MiB files. To analyse this drop in performance for smaller files the measurements were repeated with random input file sizes between 1 and 100 MiB and plotted as shown in Figure 6.2. As observable the performance of the encoder drops significantly at file sizes below around 20 MiB.

This could be caused by following reasons. The parallelization of RS Encoding has only been implemented for intermediate frames, with smaller data sizes the impact of slower encoding of the start- and end frames becomes more influential. Aditionally he start- and endframe are always created. With the substantial size of transfer frames caused by large interleaver sizes, this generates a lot of overhead for very small user data sizes. A used interleaver depth of 3680 for example results in a tranfer frame size of 0.895 MiB. If files

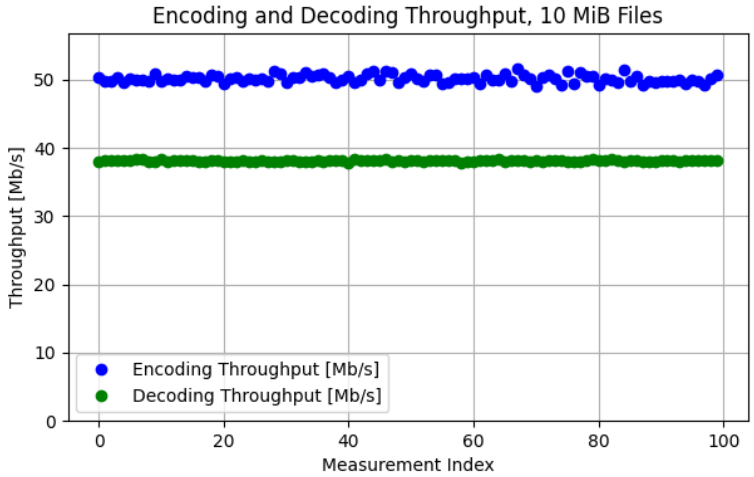


Figure 6.1 Encoding and Decoding Throughput. On SoC to/from eMMC storage.

smaller than 5 MiB shall be transmitted frequently over the link and grouping of individual files into e.g. a tarball is not viable this has to be studied further.

File Size	Operation	Avg. Throughput [Mb/s]	Std. Dev. [Mb/s]
10 MiB	Encoding	50.21	0.55
	Decoding	38.13	0.10
100 MiB	Encoding	62.86	0.11
	Decoding	48.25	0.26

Table 6.1 Encoding and Decoding on SoC. Average throughput and standard deviation to/from eMMC storage. Decoding throughput is based on the encoded file size.

6.2.1 Analysing Coresight Performance Counters with Perf

CoreSight is a debug and trace technology in ARM architectures, such as the ARM Cortex-A53, that enables non-intrusive performance monitoring by capturing data about the CPUs internal operations. It allows tools like perf to access performance counters, providing insights into various execution metrics without interfering with the normal operation of the system.

To utilize CoreSight on the SoC, I added the perf utility to the yocto image and then modified the kernel configuration to enable CoreSight by setting `CONFIG_CORESIGHT=y` and other CoreSight parameters in the kernel settings [48]. Once this configuration was in place, I tested access to the performance counters by running the perf selftest command, ensuring that perf could capture data from the CoreSight infrastructure.

When analysing the output of `$ perf stat` of the different optimization stages it can be observed e.g. for the encoding chain in Table 6.2 that the first optimizations yielded an improvement of the IPC due to the optimization of memory accesses and compiler optimizations. As expected the utilized number of CPUs is increasing with the parallelized versions. The number of utilized CPU cores for the "parallelized" encoding- and decoding chain applications is not approaching the total number of four available CPU cores because all coding chain segments apart from the RS coding are still running in a single thread.

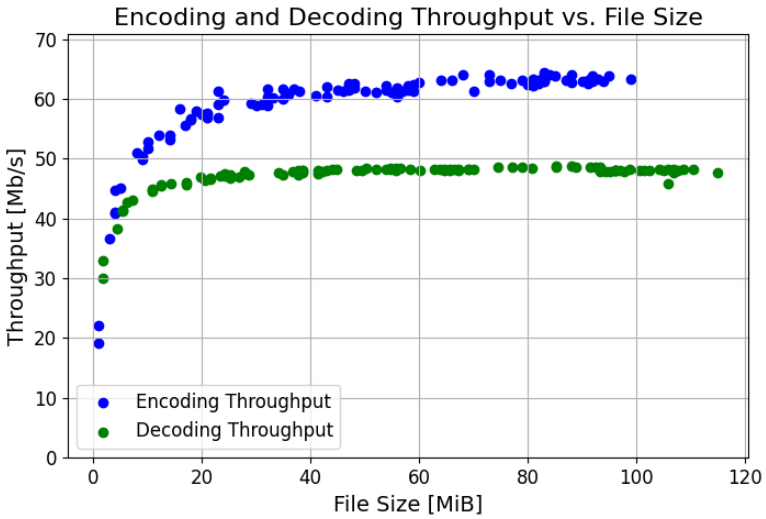


Figure 6.2 Throughput Plotted Over Filesize. On SoC to/from eMMC storage. Filesize for encoder is the input file size. Filesize for decoder is the encoded file size.

The shell output of `$ perf stat` of the different encoding-/decoding chain optimization stages can be found in the Appendix in Listings A.19 and A.20.

	Encoder		Decoder	
Stage	**CPUs Utilized**	**IPC**	**CPUs Utilized**	**IPC**
Unoptimized	0.883	0.85	0.997	i
Optimized	0.999	1.02	0.87	i
Parallelized	2.194	1.01	3.105	i

Table 6.2 Comparison of CPUs Utilized and IPC Across Different Optimization Stages. Execution on SoC.

7 Outlook and Considerations for the Future

7.1 Further Possible Optimizations

Even after parallelization, the RS encoding and decoding remain the bottlenecks of the coding chains, consuming most of the processing time of processing a transfer frame. Efforts to further optimize the encoding and decoding chains in the future should therefore focus on this segment of the coding chains. A possibility to optimize the RS coding further would be to move the workload to other available processing units available on the Xilinx UltraScale+ like the GPU or FPGA.

Guoyang Chen et al. analysed different computing units such as multicore CPUs, GPUs and FPGAs for Reed Solomon coding in storage systems [49]. Their experiments concluded that an implementation on FPGAs showed the most promising results [49].

A short analysis on the possibility of moving the RS coding onto the GPU or PL (FPGA) of the used Zynq UltraScale+ in CubeISL is given in the next sections.

7.1.1 Moving RS Coding workload to the GPU

Several other works demonstrated efficient coding on GPUs. For example from Suzuki et al. demonstrating 10-Gbps Reed Solomon decoding on a NVIDIA Tesla M40 GPU [50]. They also use a RS(255,223) configuration and parallelize the workload using data parallelism.

While the used Tesla M40 GPU is a high-performance GPU, the Mali 400 GPU on the Xilinx UltraScale+ is a low-power GPU focused on power efficiency and designed for graphics applications. It therefore does not support OpenCL, a framework for cross-platform parallel programming that allows developers to write code for CPUs, GPUs, and FPGAs, to accelerate computational workloads [51]. OpenCL is used by many works for the implementation that demonstrated high-speed RS coding on a GPUs for example by Guoyang Chen et al. [49]. Although it is possible to also perform mathematic computation with OpenGL [52], implementing RS coding on the GPU is less straightforward than with OpenCL.

In Conclusion lack of computional performance combined with the unavailability of OpenCL support make the Mali 400 a not very promising option for further optimization efforts.

7.2 Moving RS Coding workload to the FPGA

AMD/Xilinx offers IP Cores for RS Encoding [53] and Decoding [54]. These IP cores can even be configured to adhere to the CCSDS Standard and are available for the Zynq UltraScale+ family [54].

The maximum theoretical throughput of the IP cores for the UltraScale+ family can be calculated with the specifications from AMD:

7.2.1 Maximum Throughput of RS Encoder v9.0

To calculate the maximum data input rate D in Mb/s, we can use formula 7.1 provided by AMD [55]:

$$D[\text{Mb/s}] = F_{\text{max}}[\text{MHz}] * \text{Symbol\_Width}[\text{bits}] * \frac{k}{n} \tag{7.1}$$

Inserting the maximum clock frequency stated for CCSDS of 352 MHz for the Zync UltraScale+ family [56] and the parameters of 8 bit symbol RS(255,223) encoding with $n = 255$ and $k = 223$ a **maximum raw data input rate of 2462 Mb/s for the CCSDS RS encoder** can be calulated.

7.2.2 Maximum Throughput of RS Decoder v9.0

To calculate the maximum throughput D in Mb/s of the RS decoder IP core, we first need to calculate the Processing Delay PD [clockcycles] for one code block (without erasure decoding activated) with formula 7.2 provided by AMD [55].

$$PD = 2t^2 + 9t + 3 \qquad \text{where} \qquad t = (n/k)/2 \tag{7.2}$$

If we insert the codeword size $n = 255$ and information symbols $k = 223$ of the RS(255,223) code, we get a PD of 9 cycles. Since our $PD <= n$ the maximum throughput D for a given clock frequency F can be calculated with formula 7.3 provided by AMD [55].

$$D[\text{Mb/s}] = F[\text{MHz}] * \text{Symbol\_Width} \tag{7.3}$$

If we insert the maximum clock frequency for the CCSDS decoder of 380 MHz [57] and the 8 bit symbol width we can calculate a **maximum throughput of 3040 Mb/s for the RS decoder**.

7.2.3 Interface between PL and PS

To transfer the data from the PS to the PL and back for acceleration of the RS encoding and decoding, several high performance AXI direct memory access (DMA) interfaces could be utilized [58]. A concept of a possible implementation based on an example for FFT acceleration [59] is sketched in Figure 7.1 examplary for the decoder.

The integration into the dataflow of the existing decoding chain could be implemented as follows: The encoded codewords are stored in main memory by the decoding chain after deinterleaving, they are then transferred to the RS decoder in the FPGA using the AXI DMA interface. After the RS decoder decoded the codewords, they are sent back to the main memory over the same AXI DMA interface. The same would be done for the RS Encoder on a seperate AXI DMA Interface.

If we assume a AXI Frequency of 250 MHz and a 128 bit data width that is available in the high-performance AXI interfaces [58] that would result in a datarate of 32000 Mb/s per AXI interface. The maximum theoretical supported data rate to external memory (DDR4) is 2667 Mb/s for the Xilinx UltraScale+ Family, in theory more than enough in theory to support our datarate of 100 Mb/s. As the DDR4 memory is shared with the CPU it would have to be analysed if the data rate is sufficient for this implementation.

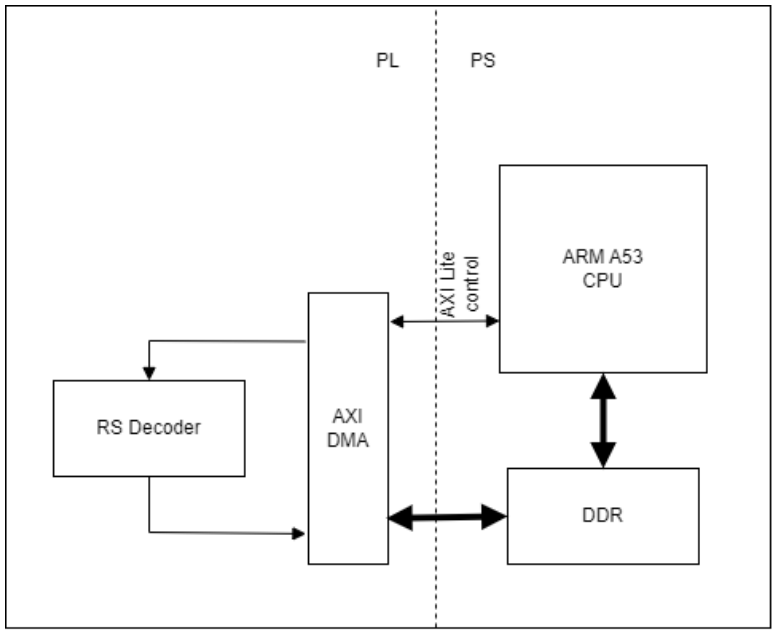


Figure 7.1 Concept of possible PS-PL AXI DMA Interface over DDR Memory. Exemplary for acceleration of the RS decoder.

7.2.4 Throughput of the Coding Chains with FPGA Acceleration

If the RS-Coding in the FPGA is implemented to be executed concurrently with the rest of the coding chain, the resulting throughput is whichever the slowest component's throughput is. To calculate the throughput of the coding chains without the RS-Coding, we can substract the time taken for RS encoding/decoding from the total encoding-/decoding chain time. The optimized coding chain profiling results for a 10 MiB file from Listing A.12 were utilized for this calculation. The total encoding chain time without RS encoding is 759.7 ms, resulting in a theoretical throughput of 107.8 Mb/s for the FPGA accelerated encoding chain. The total decoding chain time without RS encoding is 426.9 ms, resulting in a theoretical throughput of 223.9 Mb/s for the FPGA accelerated decoding chain (again referring to the encoded file size).

We can also estimate the throughput for a sequential implementation, meaning, that the CPU sends the data to the FPGA and waits until the processed data comes back. This would be more straightforward to implement but slower. To estimate how much slower, we can add the time it would take the FPGA based RS Encoder to encode a 10 MiB file to the total encoding chain time without RS-Encoding. The resulting total FPGA accelerated encoding time assuming no latency from the PL/PS Interface would be 793.8 ms resulting in a throughput of 103 Mb/s for the accelerated encoding. Similarly for the decoding chain a throughput of 210 Mb/s (referring to the encoded file size) can be estimated.

Accelerating the RS encoding and decoding in the PL could result in a significant benefit theoretically reaching CubeISLs 100 Mb/s goal for live encoding and decoding, at least when executed individually. The practical achievable throughput during simultaneous execution of the FPGA accelerated coding chains would have to be studied further. The theoretical maximum bandwidth of the AXI DMA interfaces is exceeding the required 100 Mb/s for bidirectional live coding by far. If it is decided to accelerate the RS encoding and decoding in the FPGA in the future a further detailed concept on the exact implementation of the multiple PS-PL interfaces is required.

If data rates exceeding 100 Mb/s are required for future missions, the scrambler and interleaver would be suitable components to be moved into the FPGA for further acceleration.

7.3 Impact of Corrupt Symbols on RS Decoder Performance

All measurements so far have been conducted without artificially corrupting symbols in the encoded file before decoding. The deinterleaving, descrambling as well as the frame extraction do not depend on introduced errors. Only the RS decoder might experience a performance decrease due to correcting symbols.

To get an estimate of the impact of introduced errors on the decoding performance, I performed a simple speedtest on the development machine for the used RS Decoder of libfec. Introducing 0, 8 and 16 corrupt symbols in the encoded data before decoding. This resulted in a performance decrease of 1.375x for 16 corrected symbols and 1.125x for 8 corrected symbols. The measurements can be found in the appendix in Listing A.18. Because the impact of the optimizations in chapter 5 had a similar effect on both the systems, it can be expected that the decrease in throughput by introduced bit errors behaves similarly on the SoC as on the development machine. To confirm this, detailed measurements of the impact of introduced errors on the decoding chain on the SoC are planned to be performed in the future.

7.4 Considerations for Live Coding

For the future live coding scenario, it is analyzed in this section if the latency can be reduced and how the processing resources between the encoder, decoder and other DHU components can be balanced to maintain data flow requirements. Furthermore it will be discussed how the time-critical functionality of other applications that will be running on the DHU during coding can be guaranteed.

7.4.1 Latency

In this section it is briefly assessed if the existing encoding and decoding chains could be changed in order to minimize latency for the live coding scenario.

In the existing encoding chain the interleaving is only started once all I codewords of a transfer frame have been RS encoded. In a live coding scenario this would result in a very high latency of transmitted data.

During studying the existing encoding chain I noticed that it would be possible to immediately start interleaving and scrambling of the first k collumns, before the RS encoding has even started. This is because all the information symbols are already available (assuming a very high possible user input data rate to the encoding chain). This means that we could already start streaming data to the FPGA before a whole transfer frame has been processed. However for the last $n - k$ symbols we need to wait until all I rows have been RS encoded. To visually retrace this, Figure 3.2 showing the transfer frame structure and Figure 3.3 showing the interleaving process can be helpful.

For the decoding chain however the RS decoding can only start as soon as nearly the complete transfer frame has been received. To be exact only as soon as $I * (n - 1) + 1$ symbols of the transfer frame have been received. This is visually represented in Figure 7.2.

This means that the Latency of the link inherently depends on the Interleaver depth I for our coding scheme.

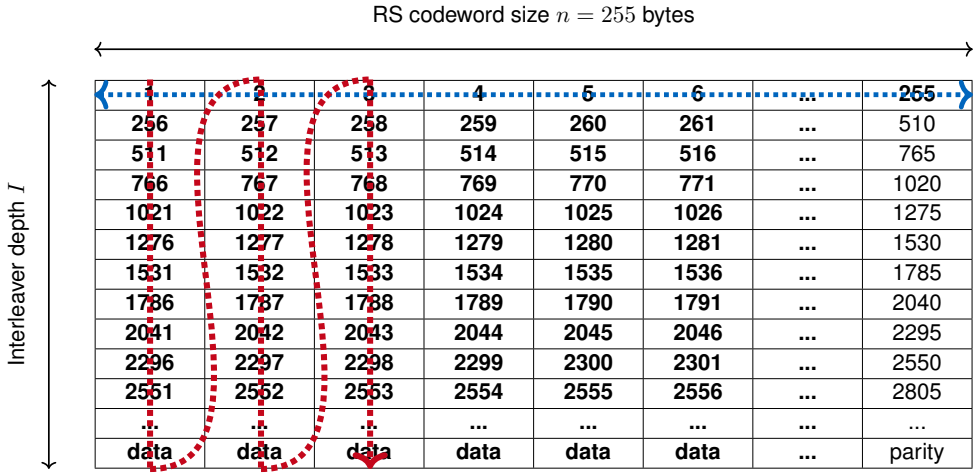


Figure 7.2 Deinterleaving Visualization. In bold text are the symbols of a transfer frame that need to be received until RS decoding can be started. Order of received encoded data (red). Symbols needed for the RS decoding of the first codeword (blue).

7.4.2 Idle frames

In order to guarantee a constant channel data rate, idle frames have already been planned for CubeISL before this work started. The concept of idle frames is summarized briefly in the following paragraph.

If the FPGA receives data at a lower data rate than the specified channel data rate from the encoding chain, it sends uniquely identifiable idle frames in between actual encoded data to sustain the channel data rate. On the receiver these idle frames are detected and dropped in the FPGA. Only the actual data is forwarded from the FPGA to the decoding chain.

This makes it possible to perform live coding on a higher channel data rate than supported by the encoding and decoding chain.

7.4.3 Simultaneous Encoding and Decoding

The decoding chain needs to operate at exactly the same or a higher datarate as the encoding chain for the bidirectional ISL link as it needs to be able to consume the encoded data at least at the same rate as the encoding chain produces data.

It is important to note that throughput for the decoding has been defined as *encoded file size* divided by *decoding time* for the so far taken profiling measurements.

To match the encoding throughput with the decoding throughput, the decoding throughput is redefined as *unencoded file size* divided by *decoding time* in this section.

To guarantee a balanced throughput of the decoding chain than for the encoding chain, a static allocation of threads to the encoding and decoding thread pools can be used because the observed performance of the chains was reasonably consistent as shown in chapter

6. To protect against short drops in data rate e.g. due to context switching to other DHU applications (Section 7.4.4) a buffer can be utilized.

Measurements

To provide an estimation of expected throughput for the simultaneous execution in the live decoding scenario, i performed measurements with the optimized and parallelized encoding and decoding chains from Section 5.5 running simultaneously on the SoC. The encoding chain is reading the input data from eMMC0 and writing the encoded data to eMMC0. The decoding chain is reading data that was encoded in advance of the measurement from eMMC1 and writing it to eMMC1.

For the file size a 10 MiB randomly generated file is chosen. This file size closely represents the size of an assumed uncompressed picture captured by a 5 megapixel sensor with 16 bits of colour depth per pixel. The same input file is used for all measurements. To balance the throughput different numbers of tc_{enc} thread pool threads for the encoding chain and tc_{dec} for the decoding chain are tested. The average rounded down throughputs measured are shown in Table 7.1.

Throughput (enc/dec) [Mb/s]	**Decoding Threads**					
		1	**2**	**3**	**4**	**5**
Encoding Threads	**1**	22/12	22/22	22/30	19/32	19/30
	2	35/12	35/21	29/27	29/29	27/27
	3	45/12	44/21	39/27	34/30	33/28

Table 7.1 Simultaneous execution of parallelized encoding and decoding chain. In each cell the throughput [Mb/s] corresponding to the input data size is given for the encoding/decoding.

A maximum balanced throughput of 29 Mb/s can be observed when utilizing two threads for the encoding chain and three threads for the decoding chain. To ensure that the decoder of the receiving satellite can keep up with the output data rate of the encoder of the transmitting satellite, a thread allocation of $tc_{enc} = 1$ and $tc_{dec} = 3$ might be better suited for the application. This would result in a user data rate for the bidirectional link of 22 Mb/s.

Assuming that the maximum number of 16 correctable symbols per codeword are corrupt and that the decoding performance is decreased by the estimated factor from Section 7.3, the decreased decoding performance would theoretically be 26.6 Mb/s. This is higher than the 22 Mb/s of the encoder, satisfying the requirement that the decoding must be faster than the encoding for the bidirectional link, including a margin of more than 4 Mb/s.

The **resulting user data rate of 22.0 Mb/s for the bidirectional link** would result in a data rate of 25.6 Mb/s transmitted over the optical link due to header- and parity overhead for the chosen coding parameters. This is **already 25% of the 100 Mb/s** channel data rate for bidirectional live coding on CubeISL.

7.4.4 Considerations for Other Software Components on the DHU

The DHU of CubeISL will also accommodate other Software modules. During a link following additional time-critical tasks need to be handled by the DHU besides the coding chains: 32 bytes of data have to be received and stored from an external microcontroller at 2 kHz. Additionally telemetry will be sent at 1 Hz to the satellite bus.

The low data rates of these applications suggest that the impact on the coding performance will be marginal. We need to however ensure that these tasks run in a predictable manner and are scheduled at the required time. For this the scheduling of the Linux system needs to be analysed. The Linux scheduling system is preemptive, meaning that CPU access can be taken away from threads by the kernel [60]. It also supports real-time policies [61]. If the time critical tasks running beside the coding application are configured to be real-time scheduled with a higher priority than the coding workload, they will interrupt the processing of the coding chains when necessary, enabling them to be run in a predictable manner, even if the coding chains use all processing resource in the rest of the time. Depending on the needed execution time of these tasks, it has to be made sure that the buffer in front of the decoding chain is large enough to buffer incoming data during the execution of the other tasks. This is because the real-time scheduled threads are run by the scheduler without interruption until they are completed or "preempted by a higher priority thread that is ready to run" [61].

8 Conclusion

The primary goal of this thesis was to evaluate the achievable processing performance of the forward error correction coding scheme for the CubeISL laser communication terminal that is planned to be launched into low-earth orbit in 2025. This evaluation focused on three key aspects:

- Optimization of the FEC coding chains and analysis whether the processing can be effectively performed on the CPU of the commercial off-the-shelf System-on-Chip.
- Identication of the considerations that need to be taken into account for the planned live FEC coding scenario in CubeISL.
- Identification of possible improvements of the FEC coding chains for future developments of laser communication terminals for small satellites.

Through optimization of the software and implementation of data level parallelism for the the Reed-Solomon encoding, i was able to demonstrate an improvement of more than 8x in encoding performance, resulting in an average encoding chain throughput of 62.86 Mb/s for 100 MiB files and 50.21 Mb/s for 10 MiB files, with standard deviations of 0.11 Mb/s and 0.55 Mb/s respectively. The testing was conducted using the exact same system setup that will be deployed on the CubeISL satellites in space, ensuring that the performance measurements accurately reflect the performance of the deployed system.

Similarly for the decoding chain i was able to demonstrate an improvement of more than 7x in decoding performance, resulting in an average decoding throughput of 48.25 Mb/s for 100 MiB files and 38.13 Mb/s for 10MiB files, with standard deviations of 0.26 Mb/s and 0.10 Mb/s respectively.

Regarding the live coding scenario for bidirectional inter-satellite links, a scheme for balancing the throughputs of the simultaneous encoding and decoding was proposed. The determined maximum user data rate for simultaneous encoding and decoding that is ensuring the dataflow requirements of ISL live coding is 22 Mb/s. This results in a data rate of 25.6 Mb/s transmitted over the optical link which is already 25% of the 100 Mb/s channel data rate for bidirectional live coding of CubeISL.

Future research needs to confirm the expected impact of varying introduced error rates on the decoding performance of the SoC.

To optimize the throughput further in the future, it is proposed to accelerate the RS coding in the FPGA. By accelerating only the RS coding segment of the coding chains in the FPGA, a throughput that would saturate the 100 Mb/s ISL channel data rate of CubeISL was estimated.

In conclusion, this work provides optimized FEC encoding- and decoding chain implementations that will be deployed in orbit. It demonstrates that CPU based processing, although not highly performant on its own, can be utilized for FEC coding in missions with constrained development timelines, where rapid implementation is crucial.

A Appendix

A.1 Listings

```
Filename: /home/jan/encode/random_data_10M.dat

File size = 10 MiB
Total encoding time = 1371.67 ms
Throughput = 61.21 Mbps

Elapsed time for reading the file: 4.97123 ms
Elapsed time for preparing the start frame: 101.015 ms
Elapsed time for preparing the intermediate frames: 1159.83 ms
Elapsed time for preparing the end frame: 98.5869 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 55.8359 ms
  - Variance for encoding: 32.4195 ms^2
- Deliver: average combined elapsed time of deliver(): 49.5053 ms
  - Interleaving:
    - Average elapsed time: 23.0717 ms
    - Variance: 4.31124 ms^2
  - Scrambling:
    - Average elapsed time: 25.9015 ms
    - Variance: 0.68748 ms^2
  - ASM:
    - Average elapsed time: 0.417618 ms
    - Variance: 0.00812136 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.11443 ms
    - Variance: 0.0076373 ms^2

Elapsed time for writing whole vector to File/Memory: 7.21033 ms
```

Listing A.1 Profiling Output of Unoptimized Encoding Chain. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat_encoded

Encoded File size = 114.91 MiB
Total decoding time = 14051.4 ms
Throughput = 68.60 Mbps

Number of CADUs in the File: 129
  - Extracting Syncmarker:
    - Average elapsed time: 0.356004 ms
    - Variance: 0.0037474 ms^2
  - Un-Scrambling:
    - Average elapsed time: 15.2285 ms
    - Variance: 1.39221 ms^2
  - Deinterleaving:
    - Average elapsed time: 21.8699 ms
    - Variance: 1.17005 ms^2
  - Decoding:
    - Average elapsed time: 66.4131 ms
    - Variance for decoding: 27.8099 ms^2
  - Extracting Frames:
    - Average elapsed time: 4.82896 ms
    - Variance: 991.6 ms^2

Elapsed time for writing whole vector to File/Memory: 7.21033 ms
```

Listing A.2 Profiling Output of Unoptimized Decoding Chain. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat

File size = 100 MiB
Total encoding time = 9860.29 ms
Throughput = 85.07 Mbps

Elapsed time for reading the file: 66.4583 ms
Elapsed time for preparing the start frame: 93.5795 ms
Elapsed time for preparing the intermediate frames: 9572.26 ms
Elapsed time for preparing the end frame: 68.8667 ms

Elapsed time of subfunctions in preparing n=126 intermediate frames:
- Average elapsed time for encoding: 26.7364 ms
  - Variance for encoding: 5.67765 ms^2
- Deliver: average combined elapsed time of deliver(): 49.2551 ms
  - Interleaving:
    - Average elapsed time: 22.2345 ms
    - Variance: 4.94429 ms^2
  - Scrambling:
    - Average elapsed time: 25.8351 ms
    - Variance: 3.55207 ms^2
  - ASM:
    - Average elapsed time: 0.371984 ms
    - Variance: 0.0148964 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.813456 ms
    - Variance: 9.48189 ms^2

Elapsed time for writing whole vector to File/Memory: 57.1609 ms
-------------------------------------------------------
Filename: /home/jan/encode/random_data_10M.dat

File size = 10 MiB
Total encoding time = 1012.87 ms
Throughput = 82.82 Mbps

Elapsed time for reading the file: 4.91323 ms
Elapsed time for preparing the start frame: 75.7265 ms
Elapsed time for preparing the intermediate frames: 845.999 ms
Elapsed time for preparing the end frame: 80.2787 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 25.9308 ms
  - Variance for encoding: 0.980408 ms^2
- Deliver: average combined elapsed time of deliver(): 51.09 ms
  - Interleaving:
    - Average elapsed time: 26.0833 ms
    - Variance: 185.624 ms^2
  - Scrambling:
    - Average elapsed time: 24.5859 ms
    - Variance: 0.661041 ms^2
  - ASM:
    - Average elapsed time: 0.34427 ms
    - Variance: 0.00552876 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.0764851 ms
    - Variance: 0.000236903 ms^2

Elapsed time for writing whole vector to File/Memory: 5.84029 ms
```

Listing A.3 Profiling Output of Encoding Chain. RS-Encoding optimized. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat

File size = 100 MiB
Total encoding time = 8368.58 ms
Throughput = 100.24 Mbps

Elapsed time for reading the file: 66.3262 ms
Elapsed time for preparing the start frame: 64.5401 ms
Elapsed time for preparing the intermediate frames: 8093.64 ms
Elapsed time for preparing the end frame: 79.2545 ms

Elapsed time of subfunctions in preparing n=126 intermediate frames:
- Average elapsed time for encoding: 25.9671 ms
  - Variance for encoding: 4.43152 ms^2
- Deliver: average combined elapsed time of deliver(): 38.3289 ms
  - Interleaving:
    - Average elapsed time: 21.7338 ms
    - Variance: 4.32815 ms^2
  - Scrambling:
    - Average elapsed time: 15.5488 ms
    - Variance: 2.80019 ms^2
  - ASM:
    - Average elapsed time: 0.365528 ms
    - Variance: 0.0110541 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.68081 ms
    - Variance: 6.36756 ms^2

Elapsed time for writing whole vector to File/Memory: 64.3652 ms
-------------------------------------------------------
Filename: /home/jan/encode/random_data_10M.dat

File size = 10 MiB
Total encoding time = 847.727 ms
Throughput = 98.95 Mbps

Elapsed time for reading the file: 4.7439 ms
Elapsed time for preparing the start frame: 66.0913 ms
Elapsed time for preparing the intermediate frames: 712.518 ms
Elapsed time for preparing the end frame: 57.7609 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 26.7181 ms
  - Variance for encoding: 2.60952 ms^2
- Deliver: average combined elapsed time of deliver(): 37.914 ms
  - Interleaving:
    - Average elapsed time: 21.7015 ms
    - Variance: 1.27577 ms^2
  - Scrambling:
    - Average elapsed time: 15.6523 ms
    - Variance: 1.03193 ms^2
  - ASM:
    - Average elapsed time: 0.433036 ms
    - Variance: 0.0271715 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.127108 ms
    - Variance: 0.00451838 ms^2

Elapsed time for writing whole vector to File/Memory: 6.13668 ms
```

Listing A.4 Profiling Output of Encoding Chain. RS-Encoding and Scrambler optimized. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat

File size = 100 MiB
Total encoding time = 7474.32 ms
Throughput = 112.23 Mbps

Elapsed time for reading the file: 72.1155 ms
Elapsed time for preparing the start frame: 55.8102 ms
Elapsed time for preparing the intermediate frames: 7230.84 ms
Elapsed time for preparing the end frame: 51.8156 ms

Elapsed time of subfunctions in preparing n=126 intermediate frames:
- Average elapsed time for encoding: 26.0125 ms
  - Variance for encoding: 30.1097 ms^2
- Deliver: average combined elapsed time of deliver(): 31.3375 ms
  - Interleaving:
    - Average elapsed time: 14.9529 ms
    - Variance: 1.76055 ms^2
  - Scrambling:
    - Average elapsed time: 15.2602 ms
    - Variance: 5.01271 ms^2
  - ASM:
    - Average elapsed time: 0.441492 ms
    - Variance: 1.23815 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.682937 ms
    - Variance: 6.82638 ms^2

Elapsed time for writing whole vector to File/Memory: 63.2861 ms
-------------------------------------------------------
Filename: /home/jan/encode/random_data_10M.dat

File size = 10 MiB
Total encoding time = 734.375 ms
Throughput = 114.23 Mbps

Elapsed time for reading the file: 4.0284 ms
Elapsed time for preparing the start frame: 56.7792 ms
Elapsed time for preparing the intermediate frames: 615.965 ms
Elapsed time for preparing the end frame: 52.0273 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 25.6002 ms
  - Variance for encoding: 0.768909 ms^2
- Deliver: average combined elapsed time of deliver(): 30.3177 ms
  - Interleaving:
    - Average elapsed time: 15.0834 ms
    - Variance: 0.574437 ms^2
  - Scrambling:
    - Average elapsed time: 14.8061 ms
    - Variance: 0.100016 ms^2
  - ASM:
    - Average elapsed time: 0.3569 ms
    - Variance: 0.00749651 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.0712799 ms
    - Variance: 2.23035e-05 ms^2

Elapsed time for writing whole vector to File/Memory: 5.38291 ms
```

Listing A.5 Profiling Output of Encoding Chain. RS-Encoding, Scrambler and Interleaver optimized. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat

File size = 100 MiB
Total encoding time = 6306.54 ms
Throughput = 133.01 Mbps

Elapsed time for reading the file: 56.3389 ms
Elapsed time for preparing the start frame: 48.5673 ms
Elapsed time for preparing the intermediate frames: 6104.52 ms
Elapsed time for preparing the end frame: 43.5939 ms

Elapsed time of subfunctions in preparing n=126 intermediate frames:
- Average elapsed time for encoding: 26.2508 ms
  - Variance for encoding: 247.708 ms^2
- Deliver: average combined elapsed time of deliver(): 22.1747 ms
  - Interleaving:
    - Average elapsed time: 6.20307 ms
    - Variance: 0.828936 ms^2
  - Scrambling:
    - Average elapsed time: 14.9815 ms
    - Variance: 2.27539 ms^2
  - ASM:
    - Average elapsed time: 0.322204 ms
    - Variance: 0.00704598 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.667942 ms
    - Variance: 8.08307 ms^2

Elapsed time for writing whole vector to File/Memory: 52.2914 ms
-------------------------------------------------------
Filename: /home/jan/encode/random_data_10M.dat

File size = 10 MiB
Total encoding time = 613.011 ms
Throughput = 136.84 Mbps

Elapsed time for reading the file: 3.57832 ms
Elapsed time for preparing the start frame: 48.7927 ms
Elapsed time for preparing the intermediate frames: 506.939 ms
Elapsed time for preparing the end frame: 46.5692 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 25.0169 ms
  - Variance for encoding: 1.75316 ms^2
- Deliver: average combined elapsed time of deliver(): 21.0692 ms
  - Interleaving:
    - Average elapsed time: 6.08075 ms
    - Variance: 0.0525514 ms^2
  - Scrambling:
    - Average elapsed time: 14.6107 ms
    - Variance: 0.15906 ms^2
  - ASM:
    - Average elapsed time: 0.308753 ms
    - Variance: 0.00484899 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.0689885 ms
    - Variance: 2.34435e-05 ms^2

Elapsed time for writing whole vector to File/Memory: 6.27608 ms
```

Listing A.6 Profiling Output of Encoding Chain. RS-Encoding, Scrambler, Interleaver and Buffer optimized. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat

File size = 100 MiB
Total encoding time = 3503.53 ms
Throughput = 239.43 Mbps

Elapsed time for reading the file: 82.0856 ms
Elapsed time for preparing the start frame: 29.0888 ms
Elapsed time for preparing the intermediate frames: 3307.4 ms
Elapsed time for preparing the end frame: 27.9303 ms

Elapsed time of subfunctions in preparing n=126 intermediate frames:
- Average elapsed time for encoding: 24.163 ms
  - Variance for encoding: 0.679382 ms^2
- Deliver: average combined elapsed time of deliver(): 2.07968 ms
  - Interleaving:
    - Average elapsed time: 0.659839 ms
    - Variance: 0.118778 ms^2
  - Scrambling:
    - Average elapsed time: 0.551049 ms
    - Variance: 0.011331 ms^2
  - ASM:
    - Average elapsed time: 0.102059 ms
    - Variance: 0.00428072 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.766732 ms
    - Variance: 9.56298 ms^2

Elapsed time for writing whole vector to File/Memory: 56.6021 ms
-------------------------------------------------------
Filename: /home/jan/encode/random_data_10M.dat

File size = 10 MiB
Total encoding time = 352.301 ms
Throughput = 238.11 Mbps

Elapsed time for reading the file: 5.79378 ms
Elapsed time for preparing the start frame: 27.5278 ms
Elapsed time for preparing the intermediate frames: 290.627 ms
Elapsed time for preparing the end frame: 21.9807 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 25.0656 ms
  - Variance for encoding: 3.46425 ms^2
- Deliver: average combined elapsed time of deliver(): 1.38482 ms
  - Interleaving:
    - Average elapsed time: 0.643156 ms
    - Variance: 0.02254 ms^2
  - Scrambling:
    - Average elapsed time: 0.542474 ms
    - Variance: 0.00750377 ms^2
  - ASM:
    - Average elapsed time: 0.124591 ms
    - Variance: 0.0114934 ms^2
  - Writing to new vector:
    - Average elapsed time: 0.0745996 ms
    - Variance: 1.61304e-05 ms^2

Elapsed time for writing whole vector to File/Memory: 5.81956 ms
```

Listing A.7 Profiling Output of Encoding Chain. RS-Encoding, Scrambler, Interleaver and Buffer optimized. GCC compiler optimizations -O3. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat_encoded

Encoded File size = 114.91 MiB
Total decoding time = 13158.9 ms
Throughput = 73.25 Mbps

Number of CADUs in the File: 129
  - Extracting Syncmarker:
    - Average elapsed time: 0.400279 ms
    - Variance: 0.0264286 ms^2
  - Descrambling:
    - Average elapsed time: 15.2219 ms
    - Variance: 0.795661 ms^2
  - Deinterleaving:
    - Average elapsed time: 14.2801 ms
    - Variance: 0.328345 ms^2
  - Decoding:
    - Average elapsed time: 66.3229 ms
    - Variance for decoding: 25.754 ms^2
  - Extracting Frames:
    - Average elapsed time: 5.53992 ms
    - Variance: 1500.5 ms^2
```

Listing A.8 Profiling Output of Decoding Chain. Shared Buffer Optimization. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat_encoded

Encoded File size = 114.91 MiB
Total decoding time = 11877.1 ms
Throughput = 81.16 Mbps

Number of CADUs in the File: 129
  - Extracting Syncmarker:
    - Average elapsed time: 0.359569 ms
    - Variance: 0.00808776 ms^2
  - Descrambling:
    - Average elapsed time: 15.4577 ms
    - Variance: 1.44035 ms^2
  - Deinterleaving:
    - Average elapsed time: 14.2799 ms
    - Variance: 0.880184 ms^2
  - Decoding:
    - Average elapsed time: 48.3052 ms
    - Variance for decoding: 11.4822 ms^2
  - Extracting Frames:
    - Average elapsed time: 5.73393 ms
    - Variance: 1779.71 ms^2
```

Listing A.9 Profiling Output of Decoding Chain. Shared Buffer and RS-Decoder optimized. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat_encoded

Encoded File size = 114.91 MiB
Total decoding time = 10126.8 ms
Throughput = 95.19 Mbps

Number of CADUs in the File: 129
  - Extracting Syncmarker:
    - Average elapsed time: 0.409946 ms
    - Variance: 0.0341784 ms^2
  - Descrambling:
    - Average elapsed time: 16.2058 ms
    - Variance: 3.63222 ms^2
  - Deinterleaving:
    - Average elapsed time: 6.87626 ms
    - Variance: 0.16451 ms^2
  - Decoding:
    - Average elapsed time: 48.7654 ms
    - Variance for decoding: 13.9696 ms^2
  - Extracting Frames:
    - Average elapsed time: 6.00465 ms
    - Variance: 1950.6 ms^2
```

Listing A.10 Profiling Output of Decoding Chain. Shared Buffer, RS-Decoder and Interleaver optimized. Running on development computer.

```
Filename: /home/jan/encode/random_data_100M.dat_encoded

Encoded File size = 114.91 MiB
Total decoding time = 7255.61 ms
Throughput = 132.85 Mbps

Number of CADUs in the File: 129
  - Extracting Syncmarker:
    - Average elapsed time: 0.229256 ms
    - Variance: 0.241893 ms^2
  - Descrambling:
    - Average elapsed time: 0.524823 ms
    - Variance: 0.0140949 ms^2
  - Deinterleaving:
    - Average elapsed time: 0.622525 ms
    - Variance: 0.0276521 ms^2
  - Decoding:
    - Average elapsed time: 47.2004 ms
    - Variance for decoding: 11.6571 ms^2
  - Extracting Frames:
    - Average elapsed time: 4.93047 ms
    - Variance: 1919.39 ms^2
```

Listing A.11 Profiling Output of Decoding Chain. Shared Buffer, RS-Decoder and Interleaver optimized. GCC compiler optimizations -O3. Running on development computer.

```
Filename: /home/root/pics/random_data_10M.dat

File size = 10 MiB
Total encoding time = 3608.63 ms
Throughput = 23.25 Mbps

Elapsed time for reading the file: 34.7463 ms
Elapsed time for preparing the start frame: 276.251 ms
Elapsed time for preparing the intermediate frames: 3045.2 ms
Elapsed time for preparing the end frame: 227.739 ms

Elapsed time of subfunctions in preparing n=11 intermediate frames:
- Average elapsed time for encoding: 258.991 ms
  - Variance for encoding: 0.00363899 ms^2
- Deliver: average combined elapsed time of deliver(): 17.7221 ms
  - Interleaving:
    - Average elapsed time: 9.49342 ms
    - Variance: 0.000122346 ms^2
  - Scrambling:
    - Average elapsed time: 5.57476 ms
    - Variance: 3.82563e-05 ms^2
  - ASM:
    - Average elapsed time: 0.687224 ms
    - Variance: 0.0465886 ms^2
  - Writing to new vector:
    - Average elapsed time: 1.96674 ms
    - Variance: 11.691 ms^2

Elapsed time for writing whole vector to File/Memory: 24.5298 ms
```

Listing A.12 Profiling Output of Encoding Chain. RS-Encoding, Scrambler, Interleaver and Buffer optimized. GCC compiler optimizations -O3. Running on target SoC.

```
Filename: /home/root/pics/random_data_10M.dat_encoded

Encoded File size = 11.6706 MiB
Total decoding time = 6515.44 ms
Throughput = 15.03 Mbps

Number of CADUs in the File: 14
  - Extracting Syncmarker:
    - Average elapsed time: 0.937518 ms
    - Variance: 0.0772089 ms^2
  - Descrambling:
    - Average elapsed time: 5.61144 ms
    - Variance: 0.000583627 ms^2
  - Deinterleaving:
    - Average elapsed time: 11.9485 ms
    - Variance: 0.0465603 ms^2
  - Decoding:
    - Average elapsed time: 434.894 ms
    - Variance for decoding: 3403.4 ms^2
  - Extracting Frames:
    - Average elapsed time: 10.5966 ms
    - Variance: 161.854 ms^2
```

Listing A.13 Profiling Output of Decoding Chain. Shared Buffer, RS-Decoder and Interleaver optimized. GCC compiler optimizations -O3. Running on target SoC.

```
 Performance counter stats for 'data-interface':

      3656.466765      task-clock (msec)         #    0.999 CPUs utilized
             116      context-switches          #    0.032 K/sec
               0      cpu-migrations            #    0.000 K/sec
           10920      page-faults               #    0.003 M/sec
      4387164534      cycles                    #    1.200 GHz
      4456726478      instructions              #    1.02  insn per cycle
       412359144      branches                  #  112.775 M/sec
        13075245      branch-misses             #    3.17% of all branches

      3.659917918 seconds time elapsed
```

Listing A.14 Performance Counters During Execution of Optimized (Not Parallelized) Encoding Chain on a 10 MiB Input File. Output of perf stat command.

```
Performance counter stats for 'di-reassembler':

      6534.073395      task-clock (msec)         #    0.999 CPUs utilized
              138      context-switches          #    0.021 K/sec
                0      cpu-migrations            #    0.000 K/sec
             8358      page-faults               #    0.001 M/sec
       7840193986        cycles                    #    1.200 GHz
       7441598797      instructions              #    0.95  insn per cycle
        533275267      branches                  #   81.615 M/sec
         31122435      branch-misses             #    5.84% of all branches

      6.538910146 seconds time elapsed
```

Listing A.15 Performance Counters During Execution of Optimized (Not Parallelized) Decoding Chain on a Encoded 10 MiB Input File. Output of perf stat command.

```
for (uint32_t i=0; i < _depth; ++i) {
    _decoder_ptr->decode(&new_vector.at(i*_nn),
              derrlocs, 0);
  }
```

Listing A.16 Source Code of RS Decoding of One Transfer Frame

```
// Initialize Thread Pool
assert(_nr_threads>0);
ThreadPool pool(_nr_threads);

std::vector<std::future<void>> results;

// Add codeword processing tasks to thread pool
for (uint32_t i=0; i < _depth; ++i) {
  results.emplace_back(pool.enqueue([this, i, &derrlocs, &new_vector,
    decoder = _decoder_ptr.get()]() mutable {
    decoder->decode(&new_vector.at(i*this->_nn),
            derrlocs, 0);
  }));
}

// Wait until decoding of all codewords is done before extracting frames
for (auto &result : results) {
  result.get();
}
```

Listing A.17 Source Code of Parallelized RS Decoding of One Transfer Frame with Thread Pool

```
$ ./rs_error_speedtest
0 corrupted symbols:
Execution time for 10000 Reed-Solomon blocks using CCSDS decoder: 0.16 sec
decoder speed: 1.11619e+08 bits/s

8 corrupted symbols:
Execution time for 10000 Reed-Solomon blocks using CCSDS decoder: 0.18 sec
decoder speed: 9.80705e+07 bits/s

16 corrupted symbols:
Execution time for 10000 Reed-Solomon blocks using CCSDS decoder: 0.22 sec
decoder speed: 7.99297e+07 bits/s
```

Listing A.18 Comparison of Performance Metrics of the Different Optimization Stages. Output of perf stat.

```
perf stat output for encoding of 10 MiB file


# Unoptimized:
 Performance counter stats for 'data-interface':

      8460.156954      task-clock (msec)         #    0.883 CPUs utilized
            18113      context-switches          #    0.002 M/sec
                0      cpu-migrations            #    0.000 K/sec
            10963      page-faults               #    0.001 M/sec
      10116447012      cycles                    #    1.196 GHz
       8558289574      instructions              #    0.85  insn per cycle
       1059532066      branches                  #  125.238 M/sec
         15351102      branch-misses             #    1.45% of all branches

      9.584474151 seconds time elapsed

# Optimized:
 Performance counter stats for 'data-interface':

      3656.466765      task-clock (msec)         #    0.999 CPUs utilized
              116      context-switches          #    0.032 K/sec
                0      cpu-migrations            #    0.000 K/sec
            10920      page-faults               #    0.003 M/sec
       4387164534      cycles                    #    1.200 GHz
       4456726478      instructions              #    1.02  insn per cycle
        412359144      branches                  #  112.775 M/sec
         13075245      branch-misses             #    3.17% of all branches

      3.659917918 seconds time elapsed

# Optimized and Parallelized:
 Performance counter stats for 'encoding-chain':

      3674.312951      task-clock (msec)         #    2.194 CPUs utilized
              328      context-switches          #    0.089 K/sec
               18      cpu-migrations            #    0.005 K/sec
            10967      page-faults               #    0.003 M/sec
       4407041353      cycles                    #    1.199 GHz
       4460913738      instructions              #    1.01  insn per cycle
        412742648      branches                  #  112.332 M/sec
         13197553      branch-misses             #    3.20% of all branches

      1.674734467 seconds time elapsed
```

Listing A.19 Encoding Chain: Comparison of Performance Metrics of the Different Optimization Stages. Output of perf stat.

```
perf stat output for decoding of 10 MiB file


# Unoptimized:
 Performance counter stats for 'di-reassembler':

      10970.579402      task-clock (msec)         #    0.997 CPUs utilized
              1199      context-switches          #    0.109 K/sec
                 2      cpu-migrations            #    0.000 K/sec
              8428      page-faults               #    0.768 K/sec
       13161638136      cycles                    #    1.200 GHz
       11451183897      instructions              #    0.87  insn per cycle
        1172004902      branches                  #  106.832 M/sec
          28550000      branch-misses             #    2.44% of all branches

      11.009075206 seconds time elapsed

# Optimized:
 Performance counter stats for 'di-reassembler':

       6534.073395      task-clock (msec)         #    0.999 CPUs utilized
               138      context-switches          #    0.021 K/sec
                 0      cpu-migrations            #    0.000 K/sec
              8358      page-faults               #    0.001 M/sec
        7840193986                  cycles                    #    1.200 GHz
        7441598797      instructions              #    0.95  insn per cycle
         533275267      branches                  #   81.615 M/sec
          31122435      branch-misses             #    5.84% of all branches

       6.538910146 seconds time elapsed

# Optimized and Parallelized:
 Performance counter stats for 'decoding-chain':

       7212.456864      task-clock (msec)         #    3.105 CPUs utilized
             68577      context-switches          #    0.010 M/sec
                39      cpu-migrations            #    0.005 K/sec
              8699      page-faults               #    0.001 M/sec
        8642793103      cycles                    #    1.198 GHz
        7814195856      instructions              #    0.90  insn per cycle
         579667496      branches                  #   80.370 M/sec
          33556436      branch-misses             #    5.79% of all branches

       2.322930968 seconds time elapsed
```

Listing A.20 Decoding Chain: Comparison of Performance Metrics of the Different Optimization Stages. Output of perf stat.

Acronyms

APD	Avalanche Photodiode
ARQ	Automatic Repeat Request
CADU	Channel Access Data Unit
CCSDS	Consultative Committee For Space Data Systems
COTS	Commercial Off-the-shelf
CPU	Central Processing Unit
DHU	Data Handling Unit
DLR	German Aerospace Center
DMA	Direct Memory Access
DTE	Direct To Earth
DUT	Device Under Test
EDFA	Erbium-doped Fiber Amplifier
EDRS	European Data Relay Satellite System
EGSE	Electrical Ground Support Equipment
ESA	European Space Agency
FEC	Forward Error Correction
FPGA	Field-Programmable Gate Array
FSOC	Free-Space Optical Communication
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
IPC	Instructions Per Cycle
ISL	Inter-satellite Link
laser	Light Amplification By Stimulated Emission Of Radiation
LCT	Laser Communication Terminal
LEO	Low-earth Orbit
LVDS	Low-voltage Differential Signaling
MPSoC	Multi-Processor System-on-Chip
NASA	National Aeronautics And Space Administration
O4C	OSIRIS4CubeSat
OBC	On-board Computer
OGS	Optical Ground Station
OISL	Optical Inter-satellite Link
OSIRIS	Optical Space Infrared Downlink System
PAT	Pointing, Acquisition, And Tracking
PC	Personal Computer
PCB	Printed Circuit Board
PD	Processing Delay
PL	Programmable Logic
PRBS	Pseudorandom Binary Sequence
PS	Processing System
RF	Radio Frequency

RS	Reed-Solomon
SCP	Secure Copy Protocol
SDR	Software Defined Radio
SNR	Signal-to-noise Ratio
SoC	System-on-Chip
SSH	Secure Shell
TBIRD	TeraByte InfraRed Delivery
TM/TC	Telemetry And Command
TNO	Dutch Organization For Applied Scientific Research
TTL	Transistor-transistor Logic
U	CubeSat Unit (10x10x10 Cm)
USB	Universal Serial Bus

Bibliography

[1] J. Rosano Nonay, R. Rüddenklau, A. Sinn, B. Rödiger, C. Schmidt, and G. Schitter, "Horizontal link demonstration over 143 km with cubeisl: The world's smallest commercial optical communication payload for inter-satellite links," in *Small Satellite Conference 2024 - 38th Annual Small Satellite Conference*, ser. 38th Annual Small Satellite Conference. Small Satellite Conference 2024, Juli 2024. [Online]. Available: https://elib.dlr.de/205896/

[2] B. Rödiger, L. R. Rodeck, M.-T. Hahn, and C. Schmidt, "Transformation of dlr's laser communication terminals for cubesats towards new application scenarios," in *Small Satellites Systems and Services - The 4S Symposium 2024*, Mai 2024. [Online]. Available: https://elib.dlr.de/204536/

[3] A. B. Raj and A. K. Majumder, "Historical perspective of free space optical communications: from the early dates to today's developments," *IET Communications*, vol. 13, no. 16, Oct. 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/iet-com.2019.0051

[4] G. J. Holzmann, "Data communications: The first 2500 years." in *IFIP Congress (2)*, 1994, pp. 2–3.

[5] G. J. Holzmann and B. Pehrson, *The early history of data networks*. IEEE Computer Society Press, 1995.

[6] A. G. Bell, "The Photophone," *Science*, vol. os-1, no. 11, Sep. 1880. [Online]. Available: https://www.science.org/doi/10.1126/science.os-1.11.130

[7] D. L. HUTT, K. J. SNELL, and P. A. BÉLANGER, "Alexander graham bell's photophone," *Opt. Photon. News*, vol. 4, no. 6, p. 1, Jun 1993. [Online]. Available: https://www.optica-opn.org/abstract.cfm?URI=opn-4-6-20

[8] M. Garlinska, A. Pregowska, K. Masztalerz, and M. Osial, "From Mirrors to Free-Space Optical CommunicationHistorical Aspects in Data Transmission," *Future Internet*, vol. 12, no. 11, Oct. 2020. [Online]. Available: https://www.mdpi.com/1999-5903/12/11/179

[9] K. Möser, *Four Technical Artifacts of the Great War*, 2015.

[10] T. H. Maiman, "Laser applications," *Physics Today*, vol. 20, no. 7, pp. 24–28, 07 1967. [Online]. Available: https://doi.org/10.1063/1.3034397

[11] M. Kolker, "LASER COMMUNICATIONS," *Annals of the New York Academy of Sciences*, vol. 163, no. 1, pp. 118–143, Sep. 1969. [Online]. Available: https://nyaspubs.onlinelibrary.wiley.com/doi/10.1111/j.1749-6632.1969.tb13041.x

[12] M. Toyoshima, “Recent trends in space laser communications for small satellites and constellations,” in *2019 IEEE International Conference on Space Optical Systems and Applications (ICSOS)*, 2019, pp. 1–5.

[13] D. Calzolaio, F. Curreli, J. Duncan, A. Moorhouse, G. Perez, and S. Voegt, “Edrs-c the second node of the european data relay system is in orbit,” *Acta Astronautica*, vol. 177, pp. 537–544, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576520304707

[14] A. U. Chaudhry and H. Yanikomeroglu, “Laser intersatellite links in a starlink constellation: A classification and analysis,” *IEEE Vehicular Technology Magazine*, vol. 16, pp. 48–56, 2021.

[15] H. Xie, Y. Zhan, G. Zeng, and X. Pan, “Leo mega-constellations for 6g global coverage: Challenges and opportunities,” *IEEE Access*, vol. 9, pp. 164 223–164 244, 2021.

[16] J. Perdigues, H. Hauschildt, W. El-Dali, S. Mezzasoma, M. Politano, Z. Sodnik, and C. Vasko, “Hydron: the esa initiative towards optical networking in space,” in *2021 European Conference on Optical Communication (ECOC)*, 2021, pp. 1–4.

[17] A. Ramli, S. M. Idrus, and A. Supaat, “Optical wireless front-end receiver design,” *2008 IEEE International RF and Microwave Conference*, pp. 331–334, 2008.

[18] A. Cline, P. Shubert, J. McNally, N. Jacka, and R. Pierson, “Design of a stabilized, compact gimbal for space-based free space optical communications (fsoc),” vol. 10096, 2017.

[19] NASA, “State-of-the-art of small spacecraft technology - communications,” https://www.nasa.gov/smallsat-institute/sst-soa/soa-communications, Feb. 2024, accessed: 08 Sept. 2024.

[20] S. Rai and A. Garg, “Fso: Issues, challenges and heuristic solutions,” pp. 1162–1170, 2019.

[21] B. Rödiger, C. Menninger, C. Fuchs, L. Grillmayer, S. Arnold, C. Rochow, P. Wertz, and C. Schmidt, “High data-rate optical communication payload for CubeSats,” in *Laser Communication and Propagation through the Atmosphere and Oceans IX*, J. A. Anguita, J. P. Bos, and D. T. Wayne, Eds., vol. 11506, International Society for Optics and Photonics. SPIE, 2020, p. 1150604. [Online]. Available: https://doi.org/10.1117/12.2567035

[22] J. Wertz and W. Larson, *Space Mission Analysis and Design*, ser. Space Technology Library. Springer Netherlands, 1999.

[23] J. Rosano, “Concept study of optical inter-satellite links on cubesats,” Master’s thesis, 2021, kontakt DLR: Benjamin Rödiger (benjamin.rödiger@dlr.de). [Online]. Available: https://elib.dlr.de/142019/

[24] B. Rödiger and C. Schmidt, “In-orbit demonstration of the world’s smallest laser communication terminal - osiris4cubesat / cubelct,” in *Small Satellites Systems and Services - The 4S Symposium 2024*, Mai 2024. [Online]. Available: https://elib.dlr.de/204534/

[25] B. Rödiger, C. Fuchs, J. R. Nonay, W. Jung, and C. Schmidt, “Miniaturized optical intersatellite communication terminal cubeisl,” in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2021, pp. 1–5.

[26] C. Schieler, K. Riesing, B. Bilyeu, J. Chang, A. Garg, N. Gilbert, A. Horvath, R. Reeve, B. Robinson, J. Wang, S. Piazzolla, A. Tarif, and B. Keer, “The 200 gbps tbird mission: Results from 6u cubesat laser communications in low earth orbit,” in *Proceedings of SPIE Optics + Photonics*, no. 20230000434. MIT Lincoln Laboratory, Jet Propulsion Laboratory, NASA Goddard Space Flight Center, Terran Orbital, 2023, presented at SPIE Optics + Photonics 2023. [Online]. Available: https://ntrs.nasa.gov/api/citations/20230000434/downloads/Schieler%20paper%20spie2023-tbird-v4.pdf

[27] K. Riesing, C. Schieler, B. Bilyeu, J. Chang, A. Garg, N. Gilbert, A. Horvath, R. Reeve, B. Robinson, J. Wang, S. Piazzolla, A. Tarif, and B. Keer, “Operations and results from the 200 gbps tbird laser communication mission,” in *37th Annual AIAA/USU Conference on Small Satellites*, no. SSC23-I-03. Logan, UT: MIT Lincoln Laboratory, Jet Propulsion Laboratory, NASA Goddard Space Flight Center, Terran Orbital, 2023, presented at the 37th Annual Small Satellite Conference, August 2023. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2023/all2023/167/

[28] AAC Clyde Space, “Cubecat communication system,” https://www.aac-clyde.space/what-we-do/space-products-components/communications/cubecat, 2024, accessed: 2024-09-10.

[29] R. Saathof, A. J. Meskers, F. Van Kempen, G. Witvoet, W. Korevaar, D. De Lange, W. Crowcombe, E. Fritz, and M. Van De Pol, “In orbit demonstration plans for an optical satellite link between a CubeSat and a ground terminal at TNO,” in *Free-Space Laser Communications XXXIII*, H. Hemmati and D. M. Boroson, Eds. Online Only, United States: SPIE, Mar. 2021, p. 6. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11678/2583381/In-orbit-demonstration-plans-for-an-optical-satellite-link-between/10.1117/12.2583381.full

[30] A. Zeedan and T. Khattab, “A Critical Review of Baseband Architectures for CubeSats Communication Systems,” 2022, version Number: 1. [Online]. Available: https://arxiv.org/abs/2201.09748

[31] M. R. Maheshwarappa, M. Bowyer, and C. P. Bridges, “Software Defined Radio (SDR) architecture to support multi-satellite communications,” in *2015 IEEE Aerospace Conference*. Big Sky, MT: IEEE, Mar. 2015, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7119186/

[32] M. R. Maheshwarappa, M. D. Bowyer, and C. P. Bridges, “A reconfigurable SDR architecture for parallel satellite reception,” *IEEE Aerospace and Electronic Systems Magazine*, vol. 33, no. 11, pp. 40–53, Nov. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8566051/

[33] B. L. Edwards, R. Daddato, K.-J. Schulz, R. Alliss, J. Hamkins, D. Giggenbach, B. Robinson, and L. Braatz, “An update on the ccsds optical communications working group interoperability standards,” in *2019 IEEE International Conference on Space Optical Systems and Applications (ICSOS)*. IEEE, 2019, pp. 1–9.

[34] B. L. Edwards, “Latest status of the ccsds optical communications working group,” in *2022 IEEE International Conference on Space Optical Systems and Applications (ICSOS)*, 2022, pp. 1–6.

[35] “Ccsds recommended standard for optical communications coding and synchronization, ccsds 142.0-b-1,” Consultative Committee for Space Data Systems (CCSDS), Standard, August 2019, issue 1. [Online]. Available: https://public.ccsds.org/Pubs/142x0b1.pdf

[36] “Ccsds experimental specification for optical high data rate communication - 1064nm, ccsds 141.11-o-1,” Consultative Committee for Space Data Systems (CCSDS), Experimental Specification, Dezember 2018, issue 1. [Online]. Available: https://public.ccsds.org/Pubs/141x11o1e2.pdf

[37] D. Giggenbach, F. David, R. Landrock, K. Pribil, E. Fischer, R. Buschner, and D. Blaschke, “Measurements at a 61 km near-ground optical transmission channel,” vol. 4635, 04 2002, pp. 162–170.

[38] Xiphos, “Q8s specifications,” https://satcatalog.s3.amazonaws.com/components/460/SatCatalog_-_Xiphos_Systems_Corporation_-_Q8S_-_Datasheet.pdf, 2021, accessed: 20.05.2024.

[39] P. Karn, “Dsp and fec library,” https://www.ka9q.net/code/fec/.

[40] I. Corporation, “Intel intelligent storage acceleration library,” https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html, accessed: 10-05-2024.

[41] A. Documentation, “Gcc optimization options,” https://developer.arm.com/documentation/den0013/d/Optimizing-Code-to-Run-on-ARM-Processors/Compiler-optimizations/GCC-optimization-options, 2024, accessed: 2024-09-10.

[42] V. Pai, P. Ranganathan, H. Abdel-Shafi, and S. Adve, “The impact of exploiting instruction-level parallelism on shared-memory multiprocessors,” *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 218–226, 1999.

[43] B. Gregg, *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[44] H. Mohebbi, “Parallel SIMD CPU and GPU Implementations of BerlekampMassey Algorithm and Its Error Correction Application,” *International Journal of Parallel Programming*, vol. 47, no. 1, pp. 137–160, Feb. 2019. [Online]. Available: http://link.springer.com/10.1007/s10766-018-0574-x

[45] C. J. Hughes, *Data Parallelism*. Cham: Springer International Publishing, 2015, pp. 1–7. [Online]. Available: https://doi.org/10.1007/978-3-031-01746-9_1

[46] A. Williams, *C++ Concurrency in Action: Practical Multithreading*, ser. Manning Pubs Co Series. Manning, 2012. [Online]. Available: https://books.google.de/books?id=EttPPgAACAAJ

[47] T. U. of Munich, “C++ praktikum 10.2: Multi-threading in c++,” https://db.in.tum.de/teaching/ss21/c++praktikum/slides/lecture-10.2.pdf, 2021, accessed: 2024-08-14.

[48] C. Haitzler, “Coresight - perf,” https://docs.kernel.org/trace/coresight/coresight-perf.html, 2022, accessed: 01.10.2024.

[49] Guoyang Chen, H. Zhou, Xipeng Shen, J. Gahm, N. Venkat, S. Booth, and J. Marshall, “OpenCL-based erasure coding on heterogeneous architectures,” in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. London, United Kingdom: IEEE, Jul. 2016, pp. 33–40. [Online]. Available: http://ieeexplore.ieee.org/document/7760770/

[50] T. Suzuki, S.-Y. Kim, J.-i. Kani, T. Hanawa, K.-I. Suzuki, and A. Otaka, “Demonstration of 10-Gbps Real-Time ReedSolomon Decoding Using GPU Direct Transfer and Kernel Scheduling for Flexible Access Systems,” *Journal of Lightwave Technology*, vol. 36, no. 10, pp. 1875–1881, May 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8259283/

[51] K. Group. (2024) Opencl - open standard for parallel programming of heterogeneous systems. [Online]. Available: https://www.khronos.org/opencl/

[52] D. Göddeke, *GPGPU Basic Math Tutorial*, 2005, openGL Tutorial. [Online]. Available: https://wwwold.mathematik.tu-dortmund.de/papers/Goeddeke2005b.pdf

[53] AMD. (2024) Ip core - reed-solomon encoder. [Online]. Available: https://www.xilinx.com/products/intellectual-property/do-di-rse.html

[54] ——. (2024) Ip core - reed-solomon decoder. [Online]. Available: https://www.xilinx.com/products/intellectual-property/do-di-rsd.html

[55] ——. (2021) Reed-solomon encoder product guide (pg025). [Online]. Available: https://docs.amd.com/v/u/en-US/pg025_rs_encoder

[56] ——. (2023) Performance and resource utilization for reed-solomon encoder v9.0. [Online]. Available: https://download.amd.com/docnav/documents/ip_attachments/rs-encoder.html#zynquplus

[57] ——. (2023) Performance and resource utilization for reed-solomon decoder v9.0. [Online]. Available: https://download.amd.com/docnav/documents/ip_attachments/rs-decoder.html#zynquplus

[58] ——. (2023) High performance ps to pl axi interfaces. [Online]. Available: https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/High-Performance-PS-to-PL-AXI-Interfaces

[59] ——. (2023) Example design - zynq-based fft co-processor using the axi dma. [Online]. Available: https://adaptivesupport.amd.com/s/article/58582?language=en_US

[60] Baeldung, “Real-time process scheduling in linux,” https://www.baeldung.com/linux/real-time-process-scheduling, 2024, accessed: 01.10.2024.

[61] I. Red Hat, “Performance tunig guide - cpu scheduling,” https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-scheduler, 2020, accessed: 01.10.2024.