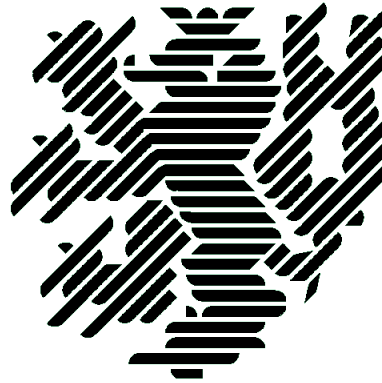


Performance Measurements of High-Bandwidth Transmission between FPGAs and Server Computers

Timo Göhring



UNIVERSITY OF WUPPERTAL
SCHOOL OF MATHEMATICS & NATURAL SCIENCES

A thesis submitted for the degree of
Bachelor of Science (B.Sc.)

Main Supervisor	Prof. Dr. Wolfgang Wagner
Second Supervisor	Prof. Dr. Christian Zeitnitz

24. September 2020



Contents

1	Introduction	1
2	Overview	3
2.1	ATLAS Experiment	3
2.2	Readout Chain	5
3	Setup for Data Transmission Tests	7
3.1	Data Generator	7
3.2	Network	8
3.3	Receiving on Computer	9
3.4	Histogramming	10
4	Testing Software and Procedure	11
4.1	Configuration	11
4.1.1	Handover of Parameters	11
4.1.2	StreamConfig	12
4.1.3	genConfig	13
4.2	Test procedure	14
4.2.1	Board Configuration	14
4.2.2	Start Generation	15
4.2.3	Stop Generation	15
4.2.4	Computer Configuration	15
4.3	Planned Tests	18
4.3.1	Data Rate and Packet Rate	19
4.3.2	Combining of Packet and Data Rate	19
4.3.3	Histogramming	21
5	Results	23
5.1	Data Rate	24
5.2	Packet Rate Limitations	26
5.2.1	Maximum Packet Rate	28
5.3	Combining Packet and Data Rate	29
5.4	Histogramming the Data	31
5.4.1	Receiving Data Streams	31
5.4.2	Multiple Data Streams	32
5.4.3	CPU Usage Scaling	34
6	Summary and Outlook	35
	Appendix	36

List of Figures

2.1	Overview of the ATLAS detector	3
2.2	Overview of a pixel module	4
2.3	Schematic of the pixel module converting a event to digital data	4
2.4	Much reduced schematic diagram of the ATLAS detector and server	5
3.1	Schematic diagram of the setup	7
3.2	Schematic of the Ethernet frame	9
3.3	Histogram examples	10
4.1	Flowchart of configuration files	12
4.2	General and stream configuration example	13
4.3	Buffer allocation in CPU cache	16
4.4	Numbering of CPU threads	17
4.5	Example of terminal output	18
4.6	Example of monitoring CPU usage with <code>htop</code>	18
4.7	Theoretical behavior of data and packet rate	20
4.8	Theoretical behavior of data and packet rate for different max. packet rates	21
5.1	Property of the data rate	24
5.2	Data rate for different paused clock cycles between payloads	25
5.3	Total packet rate for multiple data streams with a payload of 64 B	26
5.4	Maximum packet rate without packet loss	27
5.5	Unexpected step in maximum packet and data rate	28
5.6	Maximum packet rate for different payload sizes	29
5.7	Combined packet and data rate for 5 streams	30
5.8	Combined packet and data rate for 6 streams	30
5.9	CPU usage of receiving programs with two histogramming threads	33
A.1	Combined packet and data rate for 4 streams	36

List of Tables

5.1	Fit results for data rate in dependency of pause	26
5.2	Test of all 16 combinations of thread assignment in the receiving program	31
5.3	Results for two histogramming threads per Compute Complex	32
5.4	Results for two data streams per receiving program with different data rates	33
5.5	CPU usage of the receiving program for different payload sizes	34
A.1	Possible thread assignments	36

1 Introduction

The Large Hadron Collider (LHC) is the largest particle accelerator in the world with a circumference of about 27 km. It is located at the European Organization for Nuclear Research (CERN) in Switzerland and is used to study the properties of known elementary particles and search for new particles beyond the Standard Model of particle physics.

This is achieved by protons which are accelerated by the LHC in opposite directions in parallel so called beamlines. Each of these protons has an energy of 6.5 TeV per direction. The two beamlines cross each other at four different locations with a total collision energy up to 13 TeV. At every crossing point are cylindrically shaped detectors, the largest of these is the A Toroidal LHC ApparatuS (ATLAS) detector. The secondary particles from an proton-proton collision are measured by those detectors and the information of the detected particles can be used to reconstruct a collision event. This acquired data is used to test theories of particle physics [1].

In 2012 first findings on the Higgs Boson were published which is considered very important for the Standard Model and the understanding of the world. The data for this discovery was obtained during 2010 to 2012, also known as Run 1. The collision energy in this phase was 7 TeV and 8 TeV. After a first shut down of the LHC the data taking continued with proton-proton collisions at 13 TeV. A shut down from 2025 to mid 2027 is planned to increase the collected data even further. This upgrade is called the High-Luminosity LHC and enables more accurate measurements and observations of rare processes and increase our understanding of high energy physics. During the data taking after the upgrade in Run 4 and following the number of total measured collision increases by a factor of ten. Over the following ten years LHC aims to acquire a data set of about 4000 fb^{-1} [2].

The upgrade increases the peak luminosity to $7.5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ which is equivalent to about 200 proton-proton collisions per beam crossing by a crossing rate of 40 MHz. The detector also requires an upgrade to increase the data taking abilities. Some of the hardware therefore has to be replaced [3]. Hardware of the current and future ATLAS detector is briefly described in Chapter 2. The increase in data collected by the new detector also has to be handled. In order to analyze the physics with the measured data, it has to be transmitted to a computer and stored.

This thesis focuses on a possible hardware and software solution for the transmission respectively receiving of high data rates with a network between a Field Programmable Gate Array (FPGA) and a server computer. The used experimental setup for testing is shown in Chapter 3. The software and the performed tests are described in Chapter 4, they include the predictability of the data source and the determination of maximum receivable data rates for used CPU cores. The results of these tests are discussed in Chapter 5.

2 Overview

This chapter outlines the basic information about the ATLAS detector and the readout of the pixel detector. The path of the data from the front-end electronics in the detector to a server computer is described. A simplified version of this structure leads to the test setup which is used for performance measurements.

2.1 ATLAS Experiment

The ATLAS detector is one of the main particle detectors at the LHC. It was designed for the search of the Standard Model Higgs Boson but is not restricted to this. Different subsystems of the detector enable the precise measurements of particle tracks, electromagnetic and hadronic showers and muon properties. An overview of the detector is shown in Figure 2.1.

The measurements are performed by three main parts of the detector, which are placed around the beamlines to have high angular coverage to detect as many particles as possible. The inner system provides a good charged particle momentum resolution. The second layer is a electromagnetic calorimeter for electron and photon detection followed by an hadronic calorimeter for accurate jet and missing transverse energy measurements. The third layer is the muon chamber to identify and measure the muons [4].

The inner detector is immersed in a 2 T solenoidal field, which deflects charged particles depending on their charge. This allows for momentum measurements.

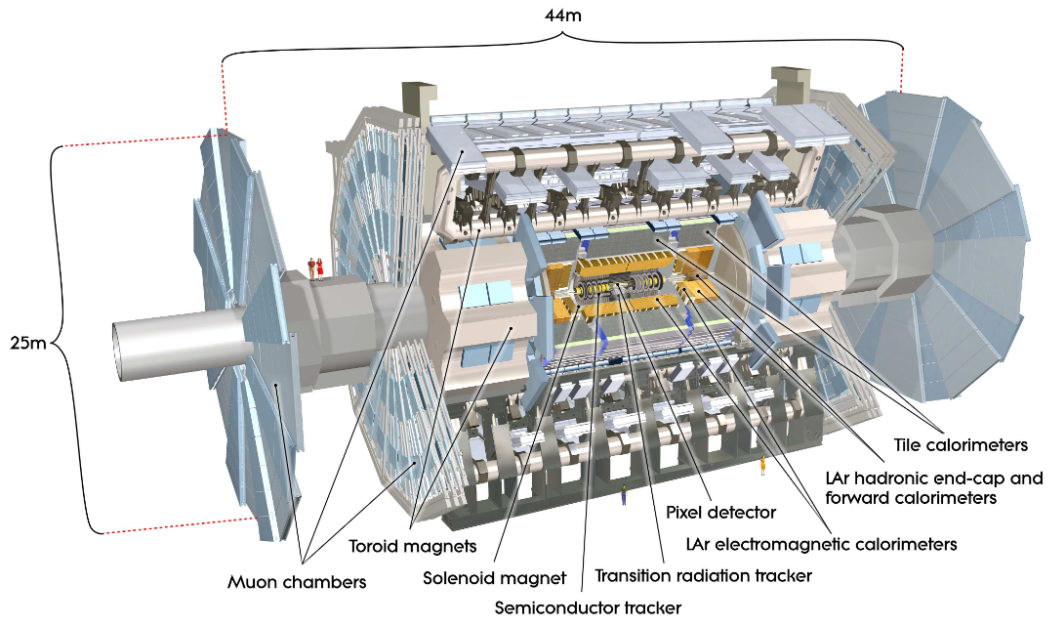


Figure 2.1: Overview of the ATLAS detector in a cut-away schematic [4].

For this thesis the relevant part is the innermost part of the inner detector, the pixel detector. This detector contains high resolution semiconductor pixels with a size of $50 \times 400 \mu\text{m}^2$ in the current detector which was driven by electronics design limitations at that time. If an ionizing particle passes through a pixel an electric charge is measurable. Each pixel acts basically like a diode. A pixel sensor with a chip on a printed circuit board form a pixel module as seen in Figure 2.2. The current pixel modules contain 46080 pixels and feature an area of about $63 \times 24 \text{ mm}^2$. Requirements for these pixel modules are fast detection and a radiation resistant structure [5].

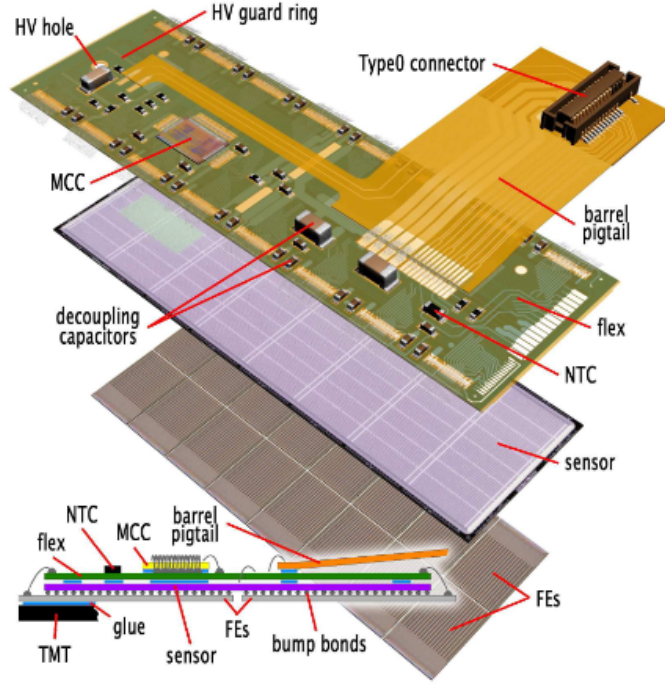


Figure 2.2: Overview of the currently used pixel module [5].

A schematic of the integrated parts on the module are shown in Figure 2.3. The analog signal from the sensor is converted to a digital signal by the readout chip.

In order to keep up with the higher radiation dose and more events at the same time from the High-Luminosity LHC new hardware for the pixel detector is required along with a reduction of the pixel size to $50 \times 50 \mu\text{m}^2$. This will also improve the spacial resolution of the measured data. The first prototype for the new pixel module called RD53A contains 400×192 pixels on a area of about $20 \times 12 \text{ mm}^2$. Because of the plan of increased amount of events per crossing and an increased trigger rate to a nominal frequency of 1 MHz, the

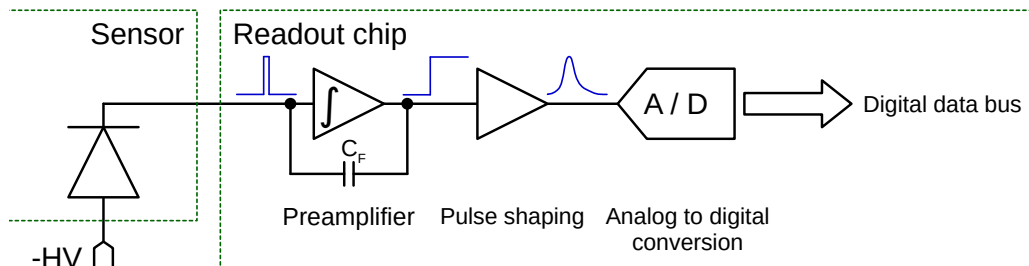


Figure 2.3: Schematic of the pixel module converting an event to digital data [6].

data rate is estimated to reach up to 5.12 Gbit/s per module. For the production module an array of 400×384 pixels is planned [3].

2.2 Readout Chain

The acquired data has to be sent through a readout chain of processing units, where it is processed before the data can potentially be stored. The digital part of the front-end chip transmits the data to off-detector readout hardware from where the data is sent to a server via a network solution. A simplified schematic of the detector readout chain can be seen in Figure 2.4.

The software also has to be changed because of new features and the higher data rate from the pixel modules. In this thesis an alternative setup to ATLAS is used to test a possible software solution for data transmission. The hardware is reduced to a Field Programmable Gate Array (FPGA) for data generation and direct sending into the network. A server CPU is used for receiving and additional network hardware enables the communication between both.

For the readout chain it is very important to not lose any data. If a fraction of the data of a collision event is lost, the whole event might not be reconstructed by the analysis computer.

The used setup aims to transfer high data rates of up to 100 Gbit/s from the source to the destination without an occurrence of data loss. The widely-used Ethernet protocol is used as the networking technology. Data is transferred by dividing the data set into smaller portions and inserting additional information to enable a purposeful exchange of data. It is possible to lose packets and hence data at multiple points in a network. Usually lost data can be sent multiple times until it is successfully transmitted. This takes additional time and resources. The used software tries to reduce the effort for reliable data transmission by reducing data loss and therefore the necessity for resending data.

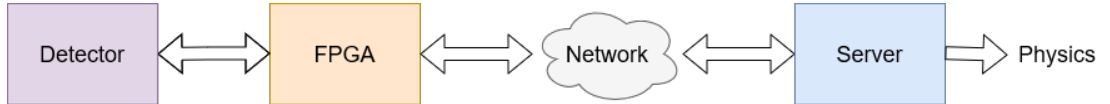


Figure 2.4: Much reduced schematic diagram of the readout chain from the ATLAS detector to a server.

3 Setup for Data Transmission Tests

In the used test setup, test data in a specific format is generated by a data generator within the FPGA and send out via the network by a network stack. At the computer this test data is received via a network card and checked by a test program. Figure 3.1 shows a block diagram of the setup. The generated data, also referred to as payload, is generated by an FPGA. In the test setup the FPGA VCU-128 by Xilinx is used [7]. Its connection to the network allows for up to 100 Gbit/s of data rate.

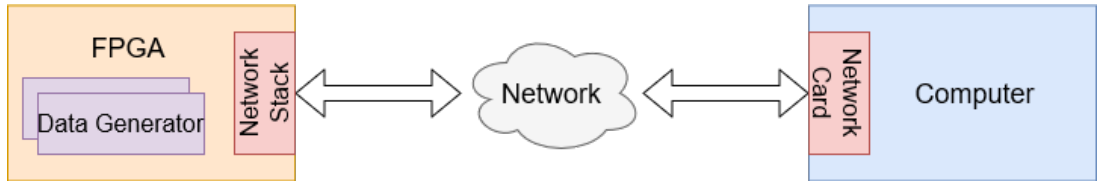


Figure 3.1: The schematic diagram of the used setup. The involved parts are described in the following sections in more detail.

3.1 Data Generator

A data generator can output the generated data with a fixed 160 MHz clock rate and 64 bit wide bus of the FPGA. Hence it can output up to 8 bytes of data per clock cycle and a theoretical total payload data rate of $64 \text{ bit} \cdot 160 \text{ MHz} = 10.24 \text{ Gbit/s}$. The payload is thus made of a given number of clock cycles n and 64 bit respectively 8 byte (i.e. $\text{payload} = 8n \text{ B}$). Assuming the bus can either output all 8 bytes or none, one can count the clock cycles with generated data as n and the clock cycles with no generated data with p per packet. p standing for paused clock cycles because no generating can be seen as equivalent to pausing data generation. $n + p$ is now the number of needed clock cycles per packet. The number of packets per second, the packet rate can therefore described as

$$\text{packet rate} = \frac{160 \text{ MHz}}{n + p}. \quad (3.1)$$

The data rate can be derived from the packet rate. Per packet the amount of data is the payload size hence

$$\begin{aligned} \text{data rate} &= \text{payload} \cdot \text{packet rate} \\ \text{data rate} &= \frac{64 \text{ b} \cdot n \cdot 160 \text{ MHz}}{n + p} \\ \text{data rate} &= \frac{n}{n + p} \cdot 10.24 \text{ Gbit/s}. \end{aligned} \quad (3.2)$$

This data rate needs two corrections. It is possible for the data generator to output a non complete data word in a clock cycle, for example 6 out of 8 bytes. As a result the last data word of a payload can be a non multiple of 8 bytes. Still the clock cycle of the data generator is an integer value thus causing for non multiple of 8 bytes a not fully used bus. Thus the number of needed clocks n has to be always rounded up. In the mathematical formula the ceiling function $\lceil n \rceil$ can be used.

Secondly, in the beginning of each generated payload the generator does not output any data for one clock cycle because of a subsequent module in the data generator. This adds one extra cycle to the paused clock cycles and p increases to $(p + 1)$. In the results (see Chapter 5) it can be seen that this addition describes the data generator correctly. The corrected generated data rate depending on size of the payload and number of paused clocks can now be described as

$$\text{data rate} = \frac{n}{\lceil n \rceil + p + 1} \cdot 10.24 \text{ Gbit/s} \quad (3.3)$$

with any payload size which is a multiple of a byte. For the pause p holds $p \geq 1$ because $p = 0$ is handled the same as $p = 1$ by the data generator. Thus the theoretical maximum data rate of 10.24 GBit/s can never be achieved.

For a given amount of data, which is consecutively generated, the number of paused clocks cannot be changed to a different value between payloads. Also the data generator does not stop until the end of the generated block is reached. This amount will be called block length and can be set via a parameter. When the end of this amount of data is reached, the last generated payload size is only the size of the remaining block length size. As a result the last generated payload can differ in size from all other payloads generated out of one block length.

The generated payload data is an increasing 16 bit counter. This leads to a repeating pattern of 65536 different values in the payload. This structure is important when histogramming the transferred data.

3.2 Network

A network is used to transfer data between devices, often over a physical distance. It can be described by a conceptual model called Open Systems Interconnection model (OSI model) [8]. It divides the network into seven layers from a physical layer to an application layer. The most relevant layers for this thesis are the transport oriented layers. For transmitting data the following protocols are used: 802.3 Ethernet, Internet Protocol version 4 (IPv4), User Datagram Protocol (UDP) and a custom Data ReadOut Protocol (DROP).

The most underlying protocol according to the OSI model is the Ethernet implementation. It fits into the first and second layer called physical and data link. Ethernet provides the basis to transfer data over a physical link, for example a cable. A packet, which is send with Ethernet contains additional information besides its payload. Destination and source address use additional 12 bytes of data. There are also two bytes used to specify the encapsulated protocol and a four byte check sum. This adds a total of 18 bytes of information to the payload. Finally Ethernet has a minimum payload size of 46 bytes so that the minimal packet size including header is 64 bytes. This restricts the maximum payload and packet rate for small payloads [9].

The implementation for the third layer of the OSI model is IPv4. The network layer is responsible for moving packets through a network and addressing devices. This protocol uses 20 bytes of header to enable its features. These bytes are additional information towards the Ethernet header [10].

On the fourth layer of the model, the transport layer, UDP can be used if the underlying layer uses IP. UDP provides port numbers to address specific functions at source and destination of a network. It works with 8 bytes of header to provide a minimum of protocol overhead. Due to a connection that does not require a circuit to be established between source and destination, there is no check if packets are transmitted in the right order or if packets are received at all [11].

The data transmission has no error correction. At this point DROP functions as counter of the transmitted packets. A 4 byte header uses 3¹ of its bytes for a packet counter to enable checking if packets are in the right order and not missing at all.

The total size of the header is 46 bytes. IPv4 and UDP header do not count towards the minimum Ethernet packet size. The DROP header of 4 bytes is going to be relevant later when measuring the data rate.

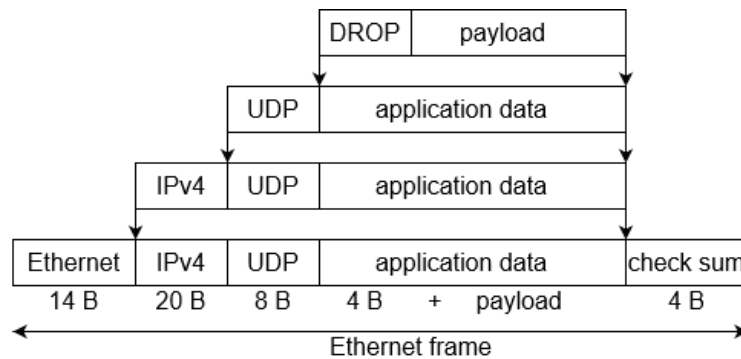


Figure 3.2: Schematic of the Ethernet frame with its various headers. In the following packet refers to this structure.

3.3 Receiving on Computer

The packets transmitted via the network are received by a network card in a server computer. A program can then take the packets and check for lost packets with the DROP header. After this, the received packets can be histogrammed. In the histogramming the occurrences of certain data words are counted, further details in Section 3.4. The used CPU on the server side of the setup is an AMD EPYC 7302 processor. It has 16 physical cores and 32 logic threads with a total of 128 MiB of L3 cache. This L3 cache is divided into 16 MiB that is shared and can be accessed by a pair of two physical cores. The CPU can hence be divided into 8 identical parts named Core Complex (CCX) by AMD [12]. This will be relevant for running software on the CPU.

A server with two of the same CPUs is used. These are set up so that one of them is reserved for the receiving program only, while the other is used by the kernel and all other running programs. The network card is connected to the reserved CPU. Per CCX one receiving program can be run. Each port number has a set queue number unique for each data stream. For each queue the network card triggers the interrupt which copies the

¹The fourth byte is reserved for future features.

received data into the cache of the CPU. From there the receiving program can process the data further. A receive thread counts the packets and histogrammer threads histogram the payload data.

3.4 Histogramming

When histogramming is enabled the histogramming thread generates data which is also written to the output file of the receiving program. This output can then be plotted with the following procedure. The receiving program counts the number of occurrences of a given word. The 16 bit value can be split into two 8 bit parts, each representing a decimal number between 0 and 255. These values can be plotted against each other and the number of occurrences of each pair can be counted in a histogram. Thereby a histogram has $256^2 = 65536$ possible points. To fill a histogram once $256^2 \cdot 2 \hat{=} 131\,072\text{ B} = 128\text{ kiB}$ are needed with 2 being the factor from splitting each counter value into two parts.

Thus a transmitted payload containing multiple counter values increases the count of given consecutive combinations of higher and lower 8 bits by one. Not received packets keep the number of occurrences of given values unchanged. Thus consecutive combinations of the 16 bit in the histogram have a lower value than expected. Since the histogram is filled multiple times, areas of consecutive combinations can overlap and cause substructures such as short lines and gradients in the occurrence. When consecutive packets are lost the area of lower occurrences is bigger and can be as big as multiple lines in the histogram. If all packets are received the increasing 16 bit counter results in fill up of the histogram from the bottom left to the top right corner. At one point in the histogram a step in the number of occurrences is to be expected because the counter of the data generator just stops at a given counter value. Hence the maximum and minimum of the number of occurrences differs by one when all data is received.

The histogram is a fast way to see if all packets are received and the variation of the number of occurrences can give an idea on the percentage of lost packets. Examples of the generated histogram are shown in Figure 3.3.

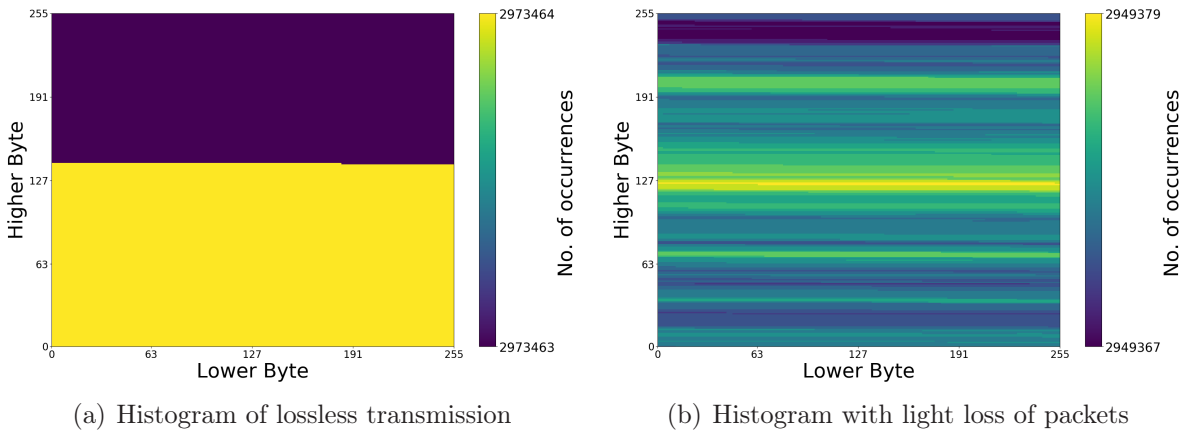


Figure 3.3: Examples of the described histogramming. Histogram (a) shows a transmission of packets with a payload of 2000 byte. No packets are lost during this transmission. In the middle the expected step is visible. Histogram (b) is made of payloads with the size of 1000 byte. The number of occurrences suggests that most of the data is received but some packets in the testing setup are lost.

4 Testing Software and Procedure

In this chapter the used software is described in more detail. This contains the configuration of data source and destination as well as the methodology of handling the configuration. Also the supplied parameters are described and the needed parametrization to perform a test of a certain aspect of the networking system is discussed.

4.1 Configuration

The configuration of the data generators in the FPGA is done by a program called **CfgFreddie** (Configuration of Front-EnD Data Interface). This program can load different configuration files and depending on the parameters in the files, data streams with different properties can be created.

In the following three configuration files are used to parametrize the data streams, which are send over the network. The board configuration specifies the numbering of data streams. A start configuration sets the number of paused clocks between each generated payload in data streams and starts them. The last configuration file stops the started data streams.

On the receiving end the receiving program loads a configuration file of parameters, which sets the size of payload a data stream is divided into. A receiving program can work with up to 4 data streams in the latest implementation. This program is started before the start configuration for the data streams is loaded into **CfgFreddie**. The configurations can be loaded via the terminal with a given filename.

4.1.1 Handover of Parameters

As mentioned the configuration of the stream and the transmission are managed by **CfgFreddie** and a receiving program. Each configuration file has to be loaded individually. Doing this by hand would consume a lot of time and could be unreliable to due user errors. At this point an automated loading of the configuration files is handy. The used file for this task is fittingly named **runTests**. It is a bash script which loads all board configurations into **CfgFreddie** and the computer configuration files into the receiving programs. It also starts all given data streams and stops them again when the test is over.

A set of parameters has still to be set by hand to enable the testing of different configurations. These parameters are encoded into a file of the data interchange format JSON. It enables grouping of parameter into data objects and has easily understandable lists of key-value pairs.

This file has to be converted to the fitting configuration files, which **CfgFreddie** and the receiving program are able to interpret. At this point a python script named **genConfig** handles the generation of all needed configuration files with the parameters from the given JSON file.

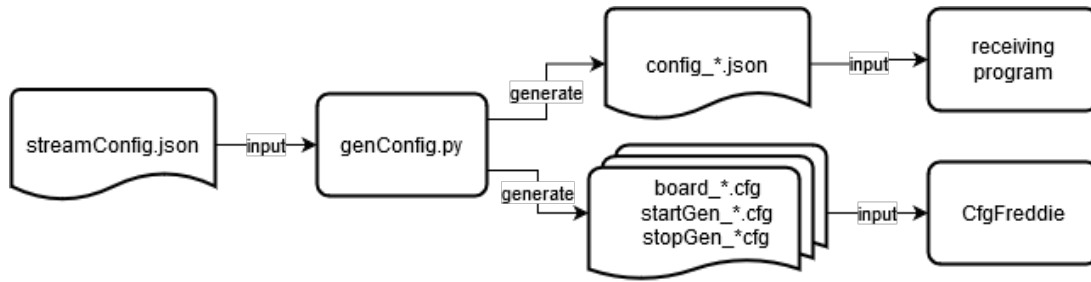


Figure 4.1: Flowchart of the involved programs and configuration files.

To start a test, the JSON file is passed to the `runTests` script. `runTests` calls the python script first and after the generation of all configuration files, they are loaded into the programs mentioned above.

The names of the files and their purpose is shortly summarized in the following enumeration, in which `*` indicates that the name also includes information to differentiate the configuration files for different boards or data streams.

`config_*.json` is used to configure the receiving program on the CPU.

`board_*.cfg` sets up the data generators of the board.

`startGen_*.cfg` starts the data generators from the board configuration.

`stopGen_*.cfg` ends the data generation by not starting a new block length.

4.1.2 StreamConfig

The **StreamConfig** is the most important file for testing. It is edited by hand and contains all important parameters in two JSON objects. The *general* object contains among other things the runtime duration and the enable statement argument for histogramming the received data.

The *board* object is a list of possible data generating boards. Each board contains a list of all configured data generators. These objects include more specific information about the data generation, the network and the histogramming configuration if latter is enabled. In Figure 4.2 an example for a **StreamConfig** is shown.

Some of the general parameters are not relevant for this thesis and therefore ignored, but are included to avoid as much hard-coded variables in the `genConfig` script as possible. The lists of board and stream objects also allows for a change of used hardware without changing the `genConfig` script.

The Boolean arguments have to be strings, because the configuration files are interpreted with *true* and *false* as strings and not with the Boolean arguments of JSON. This is important when changing parameters in the configuration file.

For testing it is important, that the number of used data streams can be changed by adding or removing objects from the *streams* list. Each stream can be different from another and has to differ in *stream* and *queue* value. The *pause*, *block-length* and *payload-size* are for configuring the data generators and the remaining values are important for the receiving program and histogramming. Each parameter is explained in the Section 4.2.


```

{
  "general":{
    "duration" : 300,
    "...",
    "histo" : "true",
    "rx-cpu-mask" : 1
  },
  "boards" : [
    {
      "name" : "vcu128-1",
      "board-ip" : "172.17.200.3",
      "streams" : [
        { "..."}
      ]
    }
  ]
}

{
  "streams" : [
    {
      "stream" : 1,
      "queue" : 1,
      "pause" : 1,
      "numFrames" : 1024,
      "frameSize" : 4096,
      "batchSize" : 128,
      "block-length" : 134217728,
      "payload-size" : 1000,
      "histo-cpu-mask" : 3
    },
    {...}
  ]
}

```

Figure 4.2: On the left is a general structure of the JSON file. The boards list contains a list of streams. A example for a stream is given on the right.

A list of streams contains in this example the parameters for only one data stream. This example results in the histogram of Figure 3.3 (b).

4.1.3 genConfig

In order to convert the given **StreamConfig** to the needed configuration files the python script **genConfig** is used. It parses the input from the JSON file and from there the values can be accessed. The script processes the parsed data and determines the necessary configuration file and writes to it.

The setup uses only one board but more than one can be configured and for each used board in the **StreamConfig** a board configuration file is created. Each file gets the fitting board-ip first and then the queue and stream numbers are passed to the configuration file. After this all data streams and their block length are set with a small delay between the stream configurations.

For the receiving program the given streams in **StreamConfig** have to be sorted. For this all streams are counted and it is determined to which configuration file they have to be passed to. This is possible, because the queue numbers are fixed to a certain Compute Complex and hence to a certain receiving program. If a computer configuration file has to contain at least information about one data stream the according JSON file is generated. The general parameters and fitting stream parameters are passed through to the file. Hence only needed configuration files are generated.

The data generation starting file is generated per board and sets the board-ip and the pause between each generated packet. Then the data generation is started. The different pauses to start with slower then intended data streams are calculated at this point. The reasoning for a slow data rate at the beginning of a test is preventing the loss of data on the receiving end. Reasons for this behavior are not clear. The last entered configuration of the data streams has the pause given in **StreamConfig** and is hence as fast as intended.

Lastly, the files to stop the data generation are generated. Similar to the starting configuration files it includes the board-ip and then every data stream of a board. It just changes the write command so that the data generation is not restarted again after the current block of data. The loading of these stop configurations is automated.

These configuration files can be read by **CfgFreddie** and the receiving program so that an automated configuration is possible.

4.2 Test procedure

In this section all relevant parameters and the configuration files are discussed. Understanding the configurable parameters leads to possible tests and limitations of the testing environment. Most interesting are the parameters for changing the properties of the data generator and the managing of the received data on the computer end of the network setup.

The **runTests** script checks every 10 seconds for running instances of the receiving program and loads the configuration for each data stream to stop if no receiving program is running anymore. After the termination of all involved processes the script copies the output from each receiving program, all configuration files, the script itself and the *StreamConfig*, which is explained in Section 4.1.2, into a result folder.

4.2.1 Board Configuration

It contains the IP address of the board it should configure and sets the mapping of the queue regarding the data streams. Setting these data streams requires a write command for each parameter. Between each change the configuration file has built-in sleep commands to avoid a too fast configuration. This is especially relevant for a high number of streams.

After that, the payload size for each stream is set. Once again by writing to a memory address by changing 16 bits to the payload value by loading its hex value to **CfgFreddie**. The relevant parameters are described in more detail in the following. The payload is explained in Section 4.2.4.

stream is an FPGA internal number to specify the data stream. The used board allows for 16 different data stream. Each stream also has to get a queue number for the network. A stream number can be used multiple times per board to fine-tune the data rate by adding multiple slower data streams. But this also would complicate the checking with a histogram.

Possible values: 0-15.

queue numbers have their own UDP port. The receiving program has one packet counter per queue number. They are chosen so that the queues 1, 2, 17 and 18 are written by the **genConfig** script to the configuration for the first receiving program². For the next receiving program the numbers 3, 4, 19 and 20 are used and so on.

For each test this is taken into account when writing the **streamConfig** file. If not stated differently, only one stream per receiving program is used. If multiple receiving programs are used, the queues are set up in a way that the receiving programs use the Compute Complexes in a consecutive order.

Possible values: 1-33.

²During testing the number of possible queues has been changed. This numbering enabled (re-)running tests with the numbering of old configuration files.

4.2.2 Start Generation

First the board-ip is set in `CfgFreddie` with this configuration file enabling the addressing of the board. Then the used data streams from this board are set for a first time with a pause of $(16 \cdot \text{pause} + 64)$ in order to reduce the data rate at the beginning. Writing to specified 32 bits blocks in the memory indicates if a new block-length of data has to be generated by changing certain bits to δ_{hex} . This sets the data generator to generate blocks. Writing a 0 effectively stops the data generation. The configuration file sets the pause between the packets two more times.

Having slower data generators at the beginning does not effect the important long term data transmission. It does only change the pause on the first few block lengths of generated data. After 200 ms the pause for the next block length is set to $(4 \cdot \text{pause} + 64)$. At an additional 200 ms later the pause is set a last time to the intended value of 1·pause.

The calculation for the needed pause values is done in the `genConfig` script in Section 4.1.3.

pause is the value for the pause parameter used in packet and data rate. It is measured in clock cycles and is given as a unsigned 16 bit value in the configuration, consequently dictating the maximum pause value. This makes really slow data rates possible because even the largest possible payload of 3400 bytes needs 425 clock cycles to be generated. The minimum pause is set by the FPGA. If 0 is entered into the configuration, the value is handled as 1. Thereby limiting the smallest value to 1. The pause is given by a 4 digit hex value limiting the maximum value.
Possible values: 1-65535, default: 1.

4.2.3 Stop Generation

While the receiving program has a set run time, the data generators generate data until they are explicitly stopped. The automated loading of the stop files for each board with active data generators stops all of them.

The configuration file for this sets once again the board-ip first. Then the same value as in the start configuration is set for each data generator with the difference being the highest digit is 0 to stop the data stream. As mentioned before, the remaining data to complete a block length is still generated and sent through the network. This can cause configuration problems for a following test if the data generator is still active, because of the time it takes to generate the remaining block length at very low data rates. As a result a new test cannot be immediately started afterwards.

4.2.4 Computer Configuration

Also the receiving end of the network has to know about the streams and how to handle the histogramming of the transmitted data. The configuration file is a JSON file, which is suitable for passing over multiple key-value pairs at once.

The parameters are structured into two groups (general values and data stream specific values). The first general parameter is the destination folder name for the output of the receiving program. A configurable folder name enables saving the test results in one folder and using a different name for each test. As the folder name the date and time of the test start is used. Because the start time for each part of the programs are slightly different, a general parameter allows for clear folder assignment. Second is the duration of the tests in

seconds. Each receiving program gets the same duration parameter and when the runtime duration is elapsed, the program stops and closes itself. The third relevant parameter is the concrete thread on the CPU the receiving part has to run on.

The remaining parameters are stream specific and can be different for every stream. Every stream has a queue number, so that the 3 byte DROP header can be differentiated from other data streams. Block length and payload size is also given to the receiving program. It is then able to identify the packets from the network which are addressed to the receiving program on a given port by packet size. From block length and payload size it is also possible to obtain the size of the last packet from the payload block.

The other parameters are a Boolean value if the received data should be histogrammed and parameter for handling of a buffer in the CPUs cache for the receiving program. This buffer consists of `numFrames` slots of `frameSize` sized memory blocks as shown in Figure 4.3.

frameSize is the size of a slot a packet is stored in. Most important is that the program can only allocate a whole or a half page of 4 kiB in the memory. This leads to frame sizes of 2048 or 4096 B. Because one CPU is only used for the receiving program, the memory allocation uses the CPU's cache first. Because the *frameSize* has fixed values it cannot be adjusted to the packet size. For payloads with a size of about 1000 B the frame size is set 2048 B and for larger packets to 4096 B. Using only the larger possible *frameSize* could result in a too fast fill up of the buffer for small packets because the packet rate is higher for smaller packets and only one packet can be stored per page in the buffer.

Possible values: 4096, 2048.

numFrames describes the number of slots that are allocated. The buffer is the total allocated part of the cache.

Because of the limited cache of the CPU [12], a maximum of 4 MiB is chosen as requirement for the buffer. In order to not exceed 4 MiB, the number of stored packets (*numFrames*) is limited to 2048 and 1024 respectively. The rest of the cache can be used by the histogrammer.

Possible values: 1024, 2048.

batchSize is the number of frames which are taken per batch. It is necessary because the receive thread has to handle stored packets. Taking one packet at a time would lead to very high computation overhead. That is why packets are handled in batches. The number of frames per batch is set by *batchSize* and is one eighth of *numFrames*. This is chosen as a middle ground between taking as many packets at once and being able to free the handled buffer that is occupied by the packets frequently. At all times a part of the allocated buffer should be free to store new packets. If the buffer

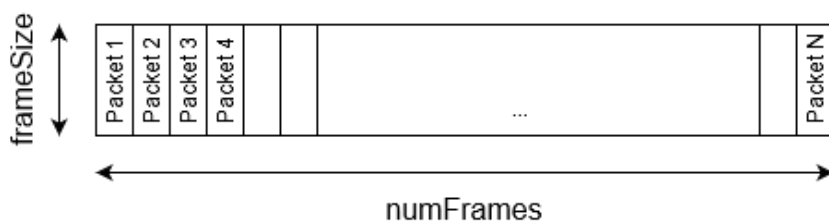


Figure 4.3: Schematic structure of the allocated buffer in the CPU cache.

would be completely filled, packets would be discarded and lost.
Possible values: 128, 256.

duration is the parameter that sets the runtime of the receiving programs in seconds.
Possible values: 1 - ($2^{31} - 1$).

payload is the size of the generated payload per packet. It is important for the packet rate and data rate. Very small payloads result in small packets and can lead to very high packet rates. For many tests the payload is often set to 2000 byte³ if the payload size is not important for a test. The maximum value is given by the used eXpress Data Path which is integrated into the Linux kernel. When changing this value too high the kernel states that the maximum transmission unit is 3498 byte and therefore the maximum of 3400 byte is chosen. Hence the sum of payload and header can fit into one transmission unit.
possible values: 1 - 3400.

block-length is the parameter which sets the generated block length in bytes. *block-length* is a 64 bit value resulting in a wide range of possible values. Changing the block length only changes the size of the last packet which is generated within a block length and hence the average packet size. The default value of 134217728 B reduces the percentage of different sized packets but is also generated relatively fast reducing the time it takes to stop a data generator.
Possible values: 1 - ($2^{64} - 1$) default: 134217728 ($= 2^{27}$).

rx-cpu-mask sets the logical thread of CPU the receive thread has to run on. There is one receive thread per receiving program and per Compute Complex only one program runs. The logical thread for the receive thread therefore only has to be chosen out of 4 available logical threads. A numbering for the threads in a Compute Complex is set as in Figure 4.4.
Possible values: 0 - 3, default: 1.

histo-cpu-mask is the number of the thread a histogrammer has to run on. A receiving program has two additional threads. Interrupts of the linux kernel capture the transmitted packets, this is fixed to thread 0 for each receiving program. A statistic thread is responsible for a console output and is fixed to thread 2. The default configuration leads to an assignment of one part of the receiving program per logical thread setting the default for *histo-cpu-mask* to the remaining thread 3.
Possible values: 0 - 3, default: 3.

³During early testing 2000 B has been a functional size and has been kept since.

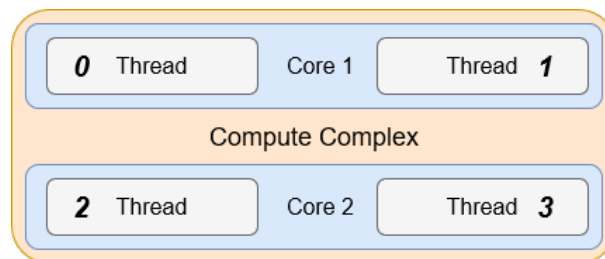


Figure 4.4: Schematic of a Compute Complex in the CPU. The physical and logical threads are numbered the same for all Compute Complexes.

4.3 Planned Tests

In this section the possible tests are discussed. After an editing of the `StreamConfig` file the test can be started with a single command line input. As a launch parameter the name of the `StreamConfig` can be passed. Running the `runTests` opens for each receiving program a tab in the terminal. First the runtime is displayed and then all queues of the receiving program. This output is split into a absolute value column and a rate column. Each column is split further into the interval and total information of each recorded quantity. These four recorded quantities are the number of received data and packets and the number of missed packets and batches in which packets are lost. The interval output every ten seconds is shown in Figure 4.5.

```
status (running time: 70.001s, interval time: 10.000s ):
Socket 0:
      value (interval/total) | rate (interval/total)
received data   : 12749369972 / 84715853612 | 10199.358 / 9681.665 Mbps
received packetes : 4244203 / 28201493 | 424.415 / 402.872 kpps
missed packetes  : 0 / 0 | 0.000 / 0.000 mpps
missed batches   : 0 / 0 | 0.000 / 0.000 mbps
```

Figure 4.5: Example of terminal output from the receiving program. The units are bits per second (bps), packets per second (pps) and batches per second (also bps) with metric unit prefixes. The example shows a data stream with payload size of 3000 B and the default pause of one clock cycle.

Each time the amount of received data, received packets, lost packets and the number of batches where a packet loss occurred are listed. In addition the time average for the complete test is output. It is important to consider that starting the data generators with a lower data rate as final lowers the time average. In addition the receiving program is started before the data generators. For a short moment the receiving program does not get any data causing the average data rate to be even lower. This order cannot be changed otherwise packets would be send before the receiving program is ready. The average of time intervals is not effected when ignoring the first few intervals.

The CPU usage of the receiving program can be seen with `htop` [13]. The display style of the information is changed by the built-in interface. `htop` displays the information of both CPUs in the computer thus the numbering of the threads. One row represents one CPU core with both its logical threads. The first two displayed lines are a Compute Complex. Each following pair of lines form a different Compute Complex.

The performed tests can roughly be split into three main parts of the setup. First is an understanding of the data generators to predict their properties when setting the payload size and pause parameters. Second is the limitation of the network solution and the consequences for the data rate and packet size. Third is the processing of the transmitted data on the CPU and how the computing ability depends on the data.

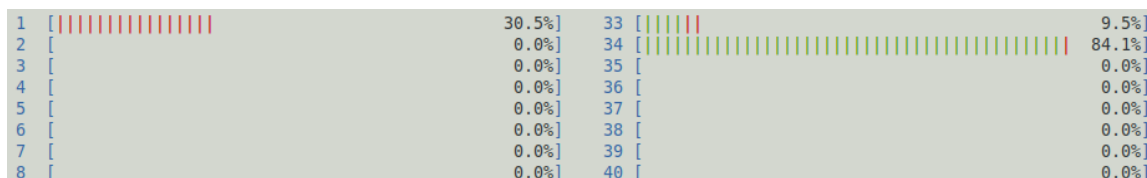


Figure 4.6: Example of monitoring the CPU usage with `htop`. Shown are half of the cores of the used CPU. Green indicates a user process, red the kernel [13, see CRT.c].

4.3.1 Data Rate and Packet Rate

At first the theoretical dependencies of the data rate are tested. This includes different pause clocks for a certain payload and different payloads. Larger payloads reduce the ratio between the payload data rate and network header. Equation 3.3 also suggests for very high and very low pauses a small relative difference for differently sized payloads.

Additional testing is necessary to measure the behavior for non multiple payloads of 8 byte. An additional byte in payload theoretically increases the data rate, but because of the data generator needing sometimes one more clock to generate a one byte larger payload, the rate decreases for this payload size. Hence the bus of the data generator is not completely used for certain payloads.

If a pause value is needed, it can be calculated by rearranging Equation 3.2 and ceiling the term because the pause only can be an integer and rounding down has the chance to increase the packet rate to much. Ceiling the pause results in slightly slower data streams.

$$\text{pause} = \left\lceil \frac{10.24 \text{ Gbit/s}}{\text{data rate}} \cdot n - (\lceil n \rceil + 1) \right\rceil \quad (4.1)$$

The packet rate which is proportional to the data rate can also be measured. The aforementioned pause equation also influences the packet rate because it is dependent from the pause as seen in Equation 3.2.

For example four data streams with a combined data rate of 40 Gbit/s created by a payload of 500 B have a packet rate of 10×10^6 p/s. p being the abbreviation for packets as a unit. Ten million packets per second could become a problem for the network card or the receiving program because of the high rate and each packet needing a certain computation time. The linear relationship also ten-folds the packet rate if the payload would be reduced to 50 B.

When operating the system at very high packet rates, the limitations of the network card and the receiving program will emerge. By reducing the packet rate until no packet loss appears, a stable upper limit can be estimated. Hence the packet rate has to be artificially limited. This estimated packet rate can be used for the next type of tests.

4.3.2 Combining of Packet and Data Rate

If the data rate and packet rate is plotted as a function of the payload size with a maximum packet rate, the data rate and packet rate behave as follows. The pause is as small as possible but has to change at one point because of the necessity to floor the packet rate. This can be seen in an example in Figure 4.7.

At one given small enough payload the packet rate has to be floored by introducing a pause between the packets. Otherwise the packet rate could ask too much of the used hardware. This results in a increasing data rate for increasing payload sizes. Because the possible packet rates are quantized a given packet rate cannot be reached for 7 out of 8 payloads. Only 1 of 8 payloads is divisible by 8 without a remainder.

The pause value needed to floor the packet rate depends on the payload size. For sufficient large payloads the packet rate is low enough, so that the pause can be the minimum value of one clock cycle. Decreasing the packet size until the floored packet rate is reached, the pause is still the minimum of one clock cycle. For eight more payload sizes the pause of one is enough because all these payload sizes need the same amount of clock

cycles to be generated. At the payload size for which another step in packet rate would occur, a pause of two clock cycles is needed. The reason is that the data generator needs one clock cycle less to generate the smaller payload at this point but by adding an extra artificial pause cycle, the number of clock cycles stays constant and hence the packet rate is constant. This is caused by the properties of the data generator.

Because of the constant packet rate, the data rate decreases linearly as seen in Equation 3.2 and this is tested. The described relations can be seen in Figure 4.7. It is also a comparison between the uncorrected and corrected theory of the data generators.

For sufficient large payload sizes the minimum pause of one clock cycle is enough to keep the packet rate lower than the maximum packet rate. For these payloads a term in the denominator cancels and the packet rate becomes constant for 8 B intervals. As an example, one uses the payload sizes from 800 to 808 B. The payload of 800 B needs 100 clock cycles to be generated and all other payloads from this interval need 101 clock cycles. Thus the packet rate is constant for 801 to 808 B and has a step at payload 800 B.

Over many intervals the packet rate increases for smaller packets still as function of payload^{-1} . If the packet rate stays constant in an interval then Equation 3.2 still implies that for increasing payloads the data rate increases. The data rate shows the same steps for increasing payload because it is connected to the packet rate via a factor. This behavior of the data rate is tested too.

At the payload size for which no artificial paused clock cycles are needed, the packet rate also decreases in steps because for a constant data rate less packets with a larger payload are needed to transmit the same amount of data.

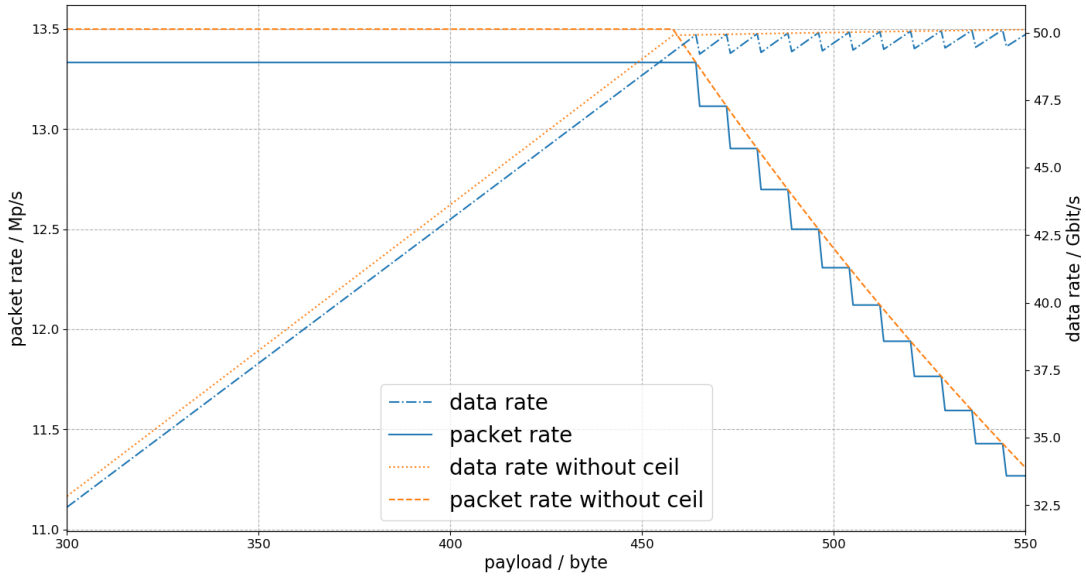


Figure 4.7: Theoretical behavior of data and packet rate for a maximum packet rate of 13.5 Mp/s. The data generation consist of 5 streams. Shown are both theoretical descriptions that are worked out before as a continuous function for better visualization.

Due to the quantization of the possible packet rates, a certain ranges of packet rates cannot be set as the floor value. For smaller maximum packet rates the payload size for which the packet rate decreases is offset to larger payload sizes. At a given payload size the different packet rates cannot be differentiated, this is the case when the necessary pause reaches one clock cycle. This can be seen in Figure 4.8.

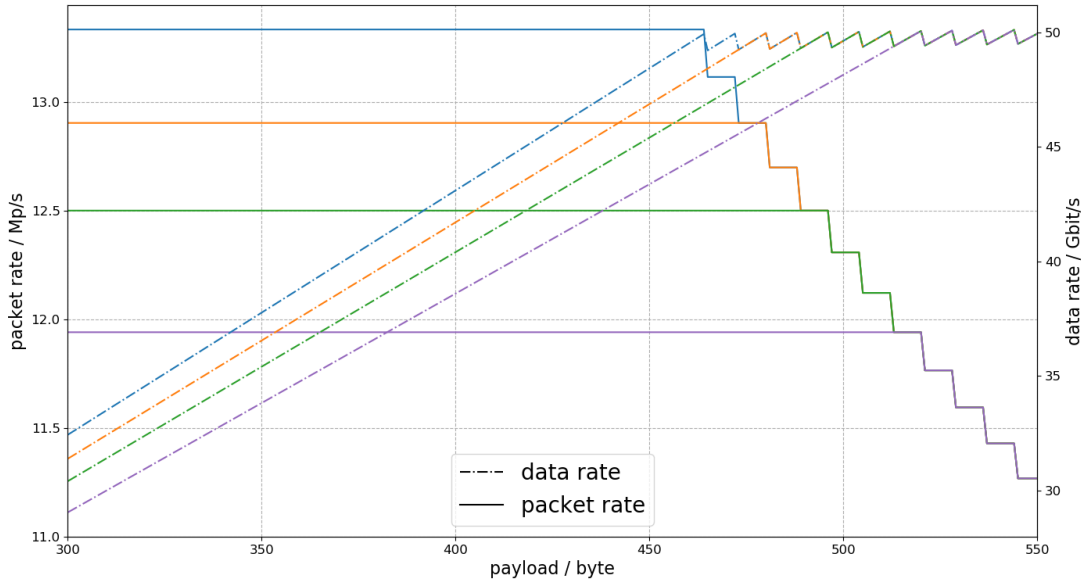


Figure 4.8: In this plot are shown different maximum packet rates of 13.5, 13, 12.5 and 12 Mp/s from the visually highest curve to the lowest curve. As expected, the different graphs are the same after a certain payload size.

Also a different amount of data streams can offset the curves. If more data streams are used the data increases but the maximum number of packets per second cannot be increased. Thus more payload data has to be send per packet. This is also tested.

Most interesting is the point at which the transition from a constant packet rate to a decreasing packet rate happens. The payload size at this point describes the minimum payload size for which a transmission of packet is possible without a need to introduce an artificial gap between packets in the network. For the application this result means that payload data has to exceed a certain size in order to not result in a too high packet rate. Sending each event fragment from a detector in a separate packet would not be possible at a certain point.

4.3.3 Histogramming

As described in Section 4.2 the receiving program has a receive thread and multiple histogrammers that can be assigned to different logical threads of the CPU. It is reasonable to assume, that not all possible assignments for the threads are useful in application because the possible occurrence of packet loss. The first test is combining the receive thread with one histogrammer and testing all 16 different possibilities of assigning the threads to a Compute Complex. Only the default configuration, which is used for all testing done before, has a known behavior at this point.

After this the configuration of two histogrammers in each receiving program is tested. Assuming both histogrammers behave the same and the numbering of them therefore permutes, 40 possible combinations are possible. Because of the observation from the test with one histogrammer, not all 40 combinations need to be tested. If there are combinations with two histogrammers and no packet loss remaining, these could be tested with a third histogrammer. If necessary, the third gets a lower data rate to histogram than the first two.

Alongside the packet loss of certain configurations, the CPU usage can be observed. The CPU usage of the receiving program still has to have a bit of overhead. At this point the scaling of the CPU usage for different payload sizes and data rates can also be monitored.

5 Results

Results of the aforementioned tests are presented and discussed in this chapter. If unexpected behavior is encountered, further tests are described and performed.

The first observation refers to the data rate which is displayed in the console output. Received data in an interval is counted with the 4 B DROP header included. Equation 3.3 for the data rate of the data generators does not take this into account. For each generated payload size of $\text{payload} = 8n$ bytes are counted $(8n + 4)$ bytes by the receiving program and are shown in the terminal output. This results in a final correction of the data rate and all derived quantities. These are listed in the following.

$$\begin{aligned} \text{data rate}_{\text{meas}} &= \frac{8n + 4}{8n} \cdot \text{data rate}_{\text{gen}} \\ \Rightarrow \text{data rate}_{\text{meas}} &= \frac{8n + 4}{8} \cdot \frac{1}{\lceil n \rceil + p + 1} \cdot 10.24 \text{ Gbit/s} \end{aligned} \quad (5.1)$$

$$\Rightarrow \text{packet rate} = \frac{\text{data rate [MB/s]}}{8n + 4} \quad (5.2)$$

$$\Rightarrow \text{pause} = \left\lceil \frac{10.24 \text{ Gbit/s}}{\text{data rate [MB/s]}} \cdot (8n + 4) - (\lceil n \rceil + 1) \right\rceil \quad (5.3)$$

This correction makes the payload data rate slightly smaller than the actual measured data rate. For payloads larger than 400 B this correction is less than 1%. Due to this the data rate is used to refer to the payload data rate from now on. This correction is also used for the plots because the plotted values are the output from the receiving program.

The second observation is about temporal fluctuations of the terminal output. Data rates of 10 Gbit/s fluctuate in the order of 0.01 %. This is a comparable small absolute deviation, but considering the pause value can be a thousandth of the payload size, the configurable data rate can be changed in almost equally small increments compared to the fluctuation. The error of the data rate is therefore estimated by the fluctuation. To increase the accuracy of the measurement multiple interval outputs are averaged.

This can have multiple reasons. First this can depend on the timing of receiving and processing of transmitted data. A batch of packets is processed at regular intervals at a constant data rate. Even though the interval statistics are an average over 10 seconds different numbers of packets can be processed.

Additionally the data rate at the beginning is slower than the final data rate and the time interval of the receiving program is started before the data generation. For example a hypothetical data stream with 10 Gbit/s, which runs for the first 10 seconds with half the data rate⁴, has over 10 minutes an average of about 9917 Mbit/s. Over an hour the average reaches 9986 Mbit/s. After 84 minutes the average differs by less than 0.1%, hence the time average needs a long run duration to be accurate. Thus the interval rates are more accurate than the average rates for shorter test runtimes.

⁴Half data rate is only an approximation for the start procedure of the data generation for this example.

Lastly the statistic thread takes a moment of computation time to output the information into the console. This also slows down the time of the statistic thread, between outputs of the statistic thread the elapsed time is 10s plus about 300 to 400 μ s. But this does not change the average value because the statistic thread uses the actual elapsed time. The output interval just shifts with time.

5.1 Data Rate

First measurements of the payload data rate are meant to prove the theoretical description of the data rate to ensure that in the following tests the data rate can be predicted correctly. The first test checks the payload dependency of the data rate.

When a certain packet size is set in the configuration file, the average payload size of the data streams is not always the set payload size. Because of the last packet which is generated in a block length, the average packet size is different when a payload size does divide the block length with a remainder. This effect is small for a block length of 128 MiB because a packet with the size of the remainder is rarely generated for all possible payload sizes. For a payload size of 2000 B in the configuration file the average generated packet size is about 1999.996 B, hence the one remaining packet does not significantly change the average. For smaller payload sizes this effect is even less relevant and is neglected for the results.

For the first test an interval of payload sizes around 2000 B is used. 256 paused clock cycles are used to approximately half the data rate to avoid the data rate limit of about 10.24 Gbit/s.

The payload sizes from 1993 to 2000 B need the same number of clock cycles $n = 250$ to be generated. For one byte smaller respectively bigger payload sizes a step in the data rate is expected. By choosing the interval of 1991 to 2002 B also payload sizes with $n = 249$ and $n = 251$ are covered therefore two steps are expected. The result of this test is shown in Figure 5.1.

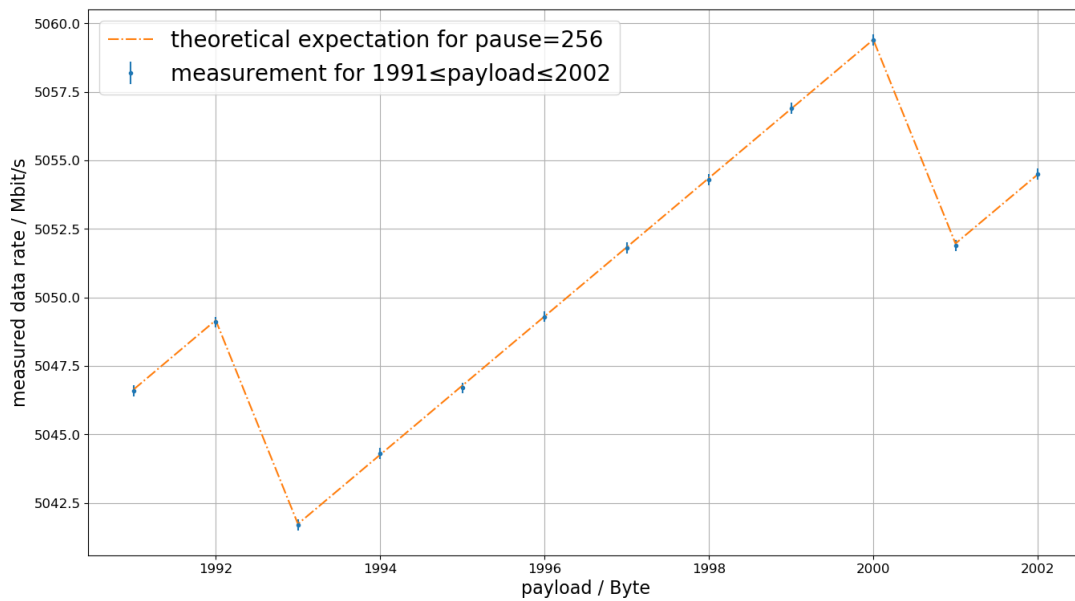


Figure 5.1: Property of the data rate for different payload sizes. The data rate shows the expected steps from the data rate having an 8 B bus.

The relative effect of the steps in the data rate is quite small, the change in data rate in a step is about 1% in this test. Larger payloads require more clock cycles and adding an extra cycle to a quite large number n is a relative small change. For very small payload sizes this effect becomes more relevant.

The number of payload sizes for which the data rate is slower after a step depends on the absolute data rate. In this test the data rate is for a by 8 dividable payload size like 1992 B higher than for 1995 B. As in Equation 5.1 the limit of high data rates behaves proportional to $\frac{8n+4}{8\lceil n \rceil + 8}$. Besides a smaller relative change for larger payloads this effect leads to more payload sizes with a slower data rate compared to a smaller payload size which is a multiple of 8 B.

Next is the data rate for different pause values in the configuration files. This is the second dependency beside the payload size of Equation 5.1. In this test the pause value is set in logarithmic steps and changed from one to 16384 clock cycles. For even higher pause values the data rate would be even smaller and very small data rates are not very relevant in this thesis.

The measured value for plotting of the data rate is acquired by averaging three interval data rates from the console output of the receiving program.

Equation 5.1 implies a similar data rate for different payload sizes by the minimum pause of one clock cycle. For very high pause values the determining term is pause^{-1} and hence the data rate decreases towards a very low data transmission. This factor also determines the global behavior.

The pause value is the parameter to set different data and packet rates for a given payload size. For this reason multiple fixed payload sizes are used. For larger payload sizes an offset to higher data rate is expected when neglecting the aforementioned effect. As a result of the effect different payload sizes can have similar data rates. That is why the curves are fitted in this test. The only free parameter in Equation 5.1 as fit is the payload size when expecting that the maximum theoretical output of the data generator is always 10.24 GBit/s. The result can be seen in Figure 5.2. Because of the absolute scale the error bars are relatively small.

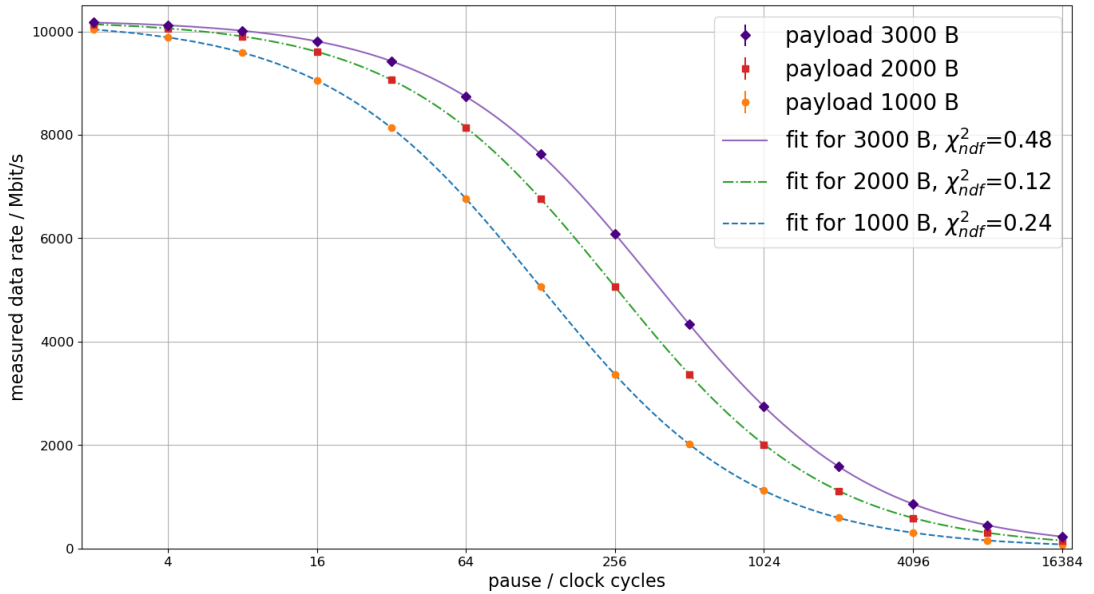


Figure 5.2: Data rate for different paused clock cycles between payloads. Tested are multiple payload sizes which offset the data rate and Equation 5.1 is used as fit function.

Table 5.1: Fit results for data rate in dependency of paused clock cycles. The payload of the fits is consistent with the set parameter in the configuration file.

payload /B	fit payload /B
1000	999.98(1)
2000	1999.98(2)
3000	2999.98(6)

The fit parameter suggests, that the effect of the average payload size which is generated in a block length can be seen in the measured results. The reduced chi-square also suggests that the error is overestimated, this could likely be solved by averaging more interval data rates than three.

The measurements above show that the description in the data rate equation is correct and pause, data and packet rate can successfully be predicted.

5.2 Packet Rate Limitations

The packet rate has an upper limit which can be seen by the following measurement. High packet rates can be reached by reducing the payload size per packet or by running multiple data streams where each one contributes to the total packet rate. A payload size of 64 B leads to a packet rate of up to 16 Mp/s per data stream.

When receiving such a data stream the receiving thread limits the packet rate which can be processed. At 16 Mp/s the right packet order or completeness cannot be considered. The console output for one data stream shows a limit of 3.88(1) Mp/s if the histogramming is enabled. When disabling the histogramming, the allocated buffer in the cache can be cleared more frequently. Processed data rate increases as a result and reaches 4.66(1) Mp/s. This is the maximum packet rate a receiving program can process and this rate seems to be handled by the network card just fine. To further increase the packet rate more streams are needed, this is shown in Figure 5.3.

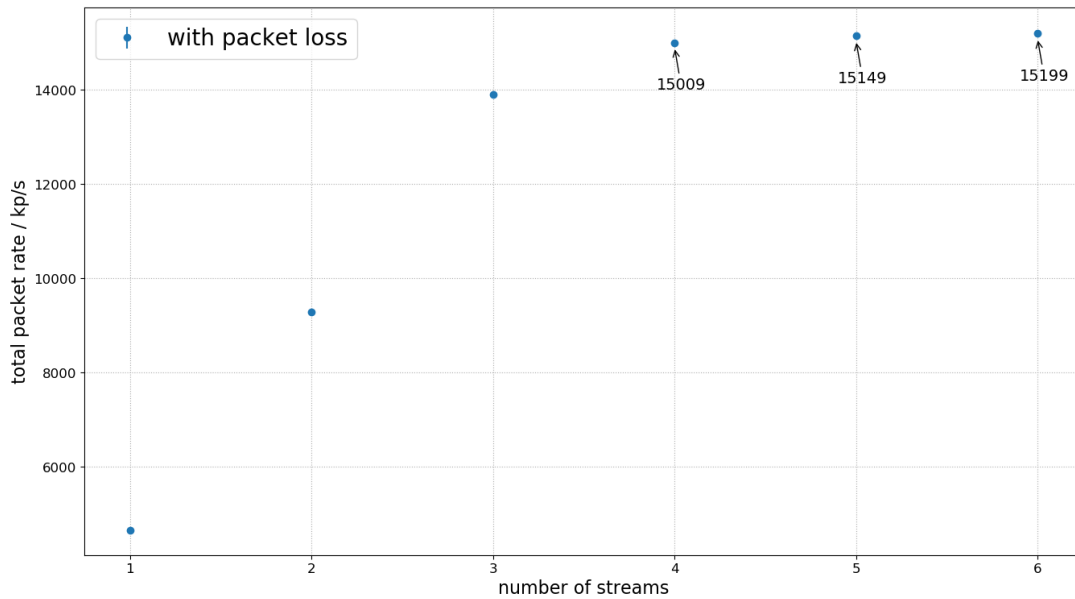


Figure 5.3: The total packet rate for a different amount of data streams with a payload of 64 B. A saturation in packet rate can be seen for 4 or more data generators.

For 3 or less data streams the total processed data rate is just a multiple of the maximum packet rate that can be processed by one receiving program. At a number of 4 streams something else limits the maximum packet rate. The network card is most likely the reason for this behavior.

To ensure that the packet rate can be set high enough for the following test, 5 data streams are used. To find the exact maximum packet rate a range of packet rates has to be set and the packet loss is observed. The highest possible packet rate without packet loss with enabled histogramming is estimated to 3.5 Mp/s based on the maximum transferable rate of 3.88(1) Mp/s from the previous section. To reach about 3.5 Mp/s with the minimum pause of one clock cycle a payload size of 350 B is required. Therefore this payload size is used for the following test.

Different packet rates with a constant payload size require different numbers of paused clock cycles, these are calculated with Equation 5.3. By transmitting 5 data streams at once, the reached packet rate can exceed 3.5 Mp/s. The packet rate interval is set from 10 Mp/s to about 15 Mp/s to ensure that the maximum packet rate without packet loss is covered. The runtime duration for the packet rates without a occurrence of packet loss is 60 min.

If the packet rate becomes unstable and packets are lost the transmitted data rate decreases further than the highest stable packet rate. This makes sense because once the packet rate cannot be processed completely by the network card, it starts to lose even more packets because of the time it takes to discard a packet to make place for new packets.

Even higher packet rates lead to more and more discarded packets. Therefore the received packet rate stays constant, the relative packet loss just increases. Both effects can be seen Figure 5.4.

The result of this part is that the maximum packet rate without packet loss for a payload of 350 B is about 12.2 Mp/s. This can be used for the next test, where this packet rate is set as an upper limit and the payload size is changed.

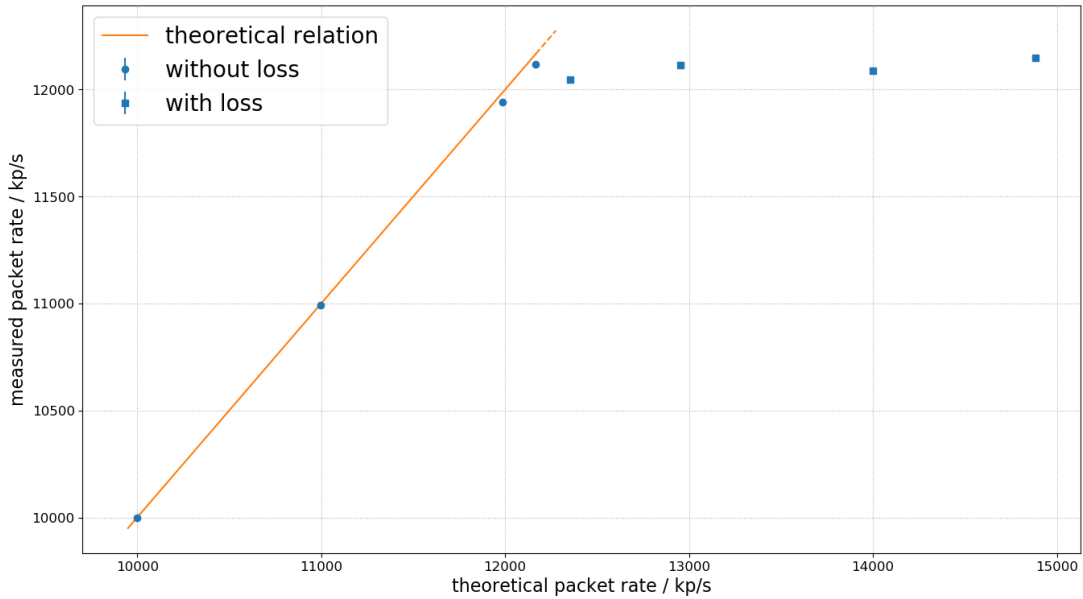


Figure 5.4: The first 4 data points do not show packet loss over 60 min. Higher packet rates lead to a loss of packets.

5.2.1 Maximum Packet Rate

A first test with the maximum packet rate set to 12.2 Mp/s and combining packet and data rate is shown in Figure 5.5.

This observation implies a dependency of the maximum packet rate on the payload size and requires unexpected further testing. The step occurs between a payload of 450 B and 500 B. For a payload of 466 B the total packet size (the payload plus the 46 B header) is 512 B. A first test for the packet sizes of 465 B and 466 B shows that the maximum packet rate suddenly changes.

A possible reason is a structure in the hardware or software of the network card which corresponds to 2^n . If a packet is smaller than a certain size it can be processed way faster than a minimal larger packet even though the network card only uses the header of the packet which does not change in size. This can result in sudden changes in the maximum packet rate the network card can handle.

To further investigate this finding, different payload sizes are set with a number of data streams so that the packet rate is too high for the network card and packet loss will occur. For a packet size of 128 B to 512 B also the payloads around each 64 B interval are measured. A performance difference of the network card for 64 B can result from the internal structure or processing of the network card. These 64 B intervals are not tested for larger payloads, because of the high number of necessary test.

For very small packets and therefore very high packet rates, the network card can only handle the first few data streams and loses all packets of the other data streams. This results in a overtaking of the receiving threads on the CPU and limits the maximum packet rate to the maximum the receiving program can handle as seen in Section 5.2. To overcome this the pause parameter of each data generator is adjusted to limit the total packet rate of all streams to 16 Mp/s.

Instead of an even higher packet rate for packet sizes below 192 B the data rate is lower than expected. This is an unexpected behavior. A possible reason could be the timing of the packets on a clock cycle level being different by introducing an artificial gap between

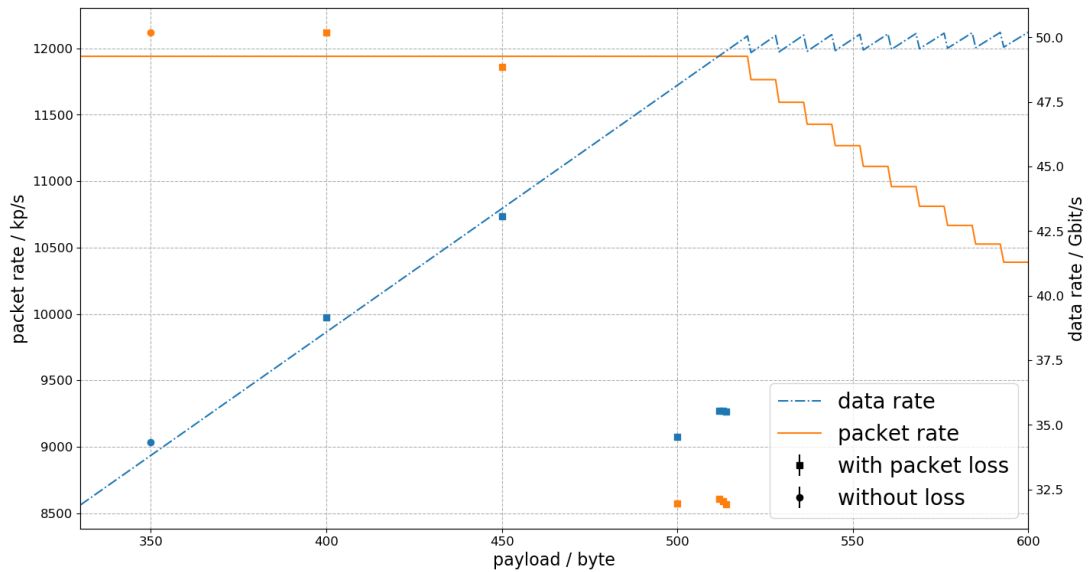


Figure 5.5: A very apparent step in the packet and data rate is observed. Because of rounding errors in the pause value to reach the maximum packet rate, the data points are off the theoretical line. The first two are too high, the third too low.

packets. Doing this the maximum packet rate for different payload sizes can be seen and an upper limit of the packet rate can be estimated for different payload sizes. The result of the tests is Figure 5.6.

Beside the packet rate step by a packet size of $256 \text{ B} + 64 \text{ B} = 320 \text{ B}$ all steps in the maximum packet rate are continuously linking to each other. The deviating step can not be explained. For larger packet sizes the continuous steps are also expected but this could be validated by further testing.

When estimating a maximum stable packet rate the largest occurring payload size has to be used. This is necessary because the maximum packet rate is a decreasing measure and the largest payload sets the upper limit. It is also important to keep the packet header in mind when estimating an upper limit, especially at the steps for 2^n .

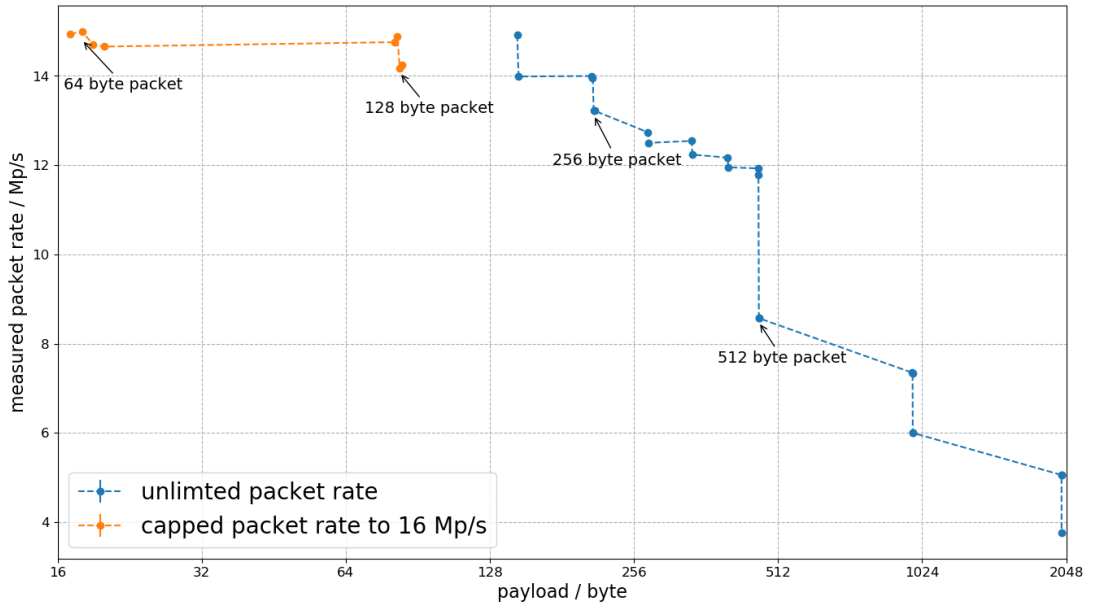


Figure 5.6: The maximum packet rate for different payload respectively packet sizes can be seen.

5.3 Combining Packet and Data Rate

With a new way to estimate a high but stable packet rate for different payload sizes the test combining the packet rate and payload rate can be repeated. A high but stable maximum packet rate is estimated to 8.5 Mp/s using Figure 5.6 for a payload of about 500 B. Lowering the estimation for the maximum packet rate shifts the payload size, for which a transmission is possible without an artificial pause, to larger payloads. This requires a new estimation of the maximum packet and so on.

This procedure ends at about 7 Mp/s at a payload size of about 900 B. With this parameters a new test is conducted. The payload is varied from 850 B to 975 B and the test runtime for each data point is 90 min. For 5 data streams the result is Figure 5.7.

For all tested points not a single packet loss occurred. Changing the number of data streams at the same maximum packet rate is no problem. For a lower number of data streams the data rate is lower and therefore the payloads smaller which are needed to reach the same packet rate. This is no issue because for smaller packet sizes the maximum packet rate is even higher. A test of this can be seen in Figure A.1.

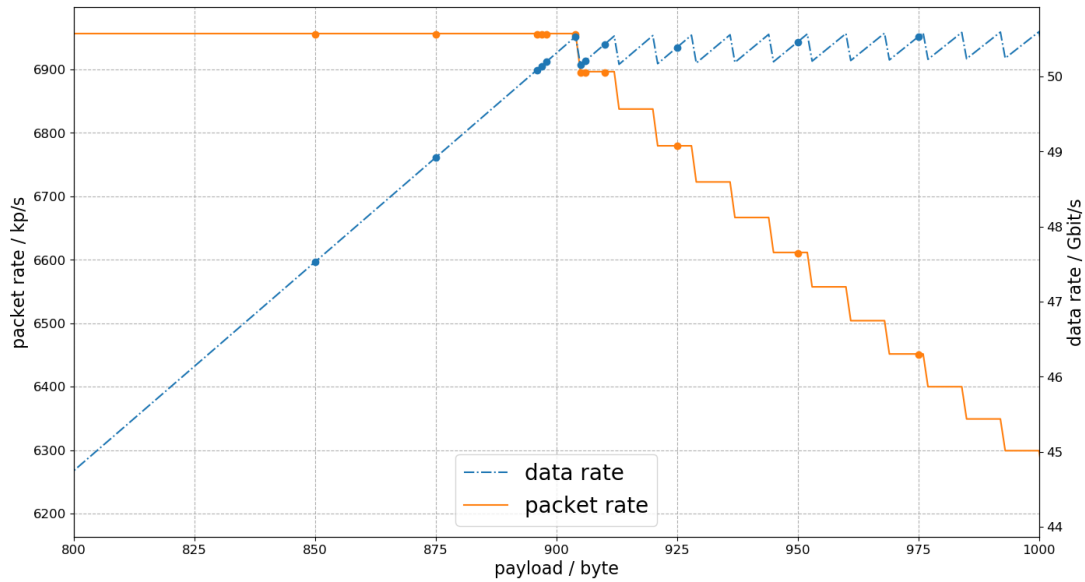


Figure 5.7: Combined packet and data rate for 5 streams and a combined maximum packet rate of 7 Mp/s.

With 6 data streams the payload rate can be further increased. The maximum packet rate is set to 5 Mp/s and the minimal payload that can be transmitted without a added pause shifts towards 1500 B. Each point is tested again for 90 min.

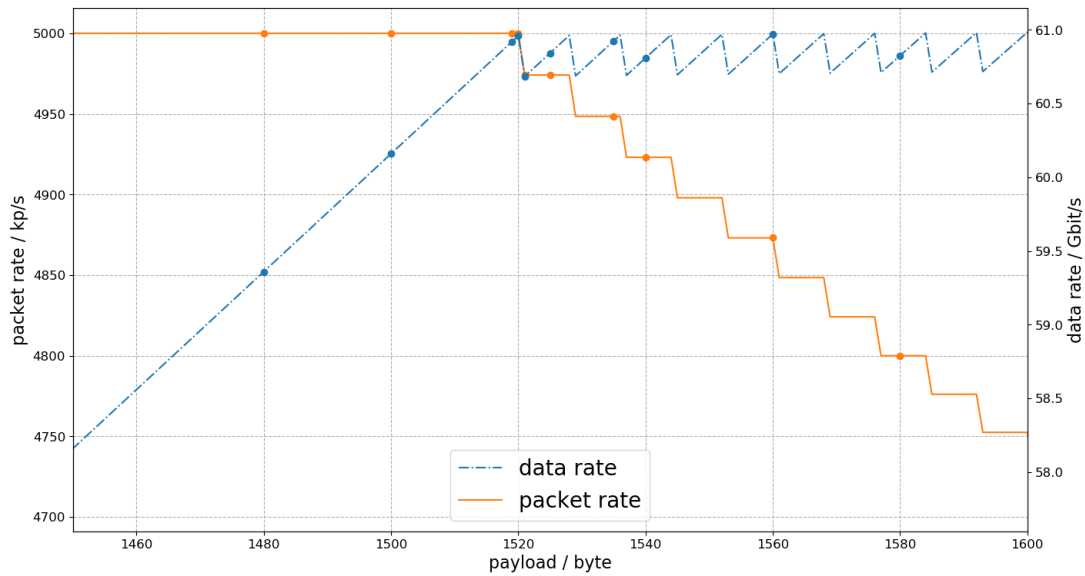


Figure 5.8: Combined packet and data rate for 6 streams and a combined maximum packet rate of 5 Mp/s. No packet loss occurred in this test.

This result shows that for ever higher transmitted data rates the minimum payload size per packet increases. This expectation is a strong information for the payload which is used. In addition only one constant payload size can be used for a data stream, which can not simulate a real usage scenario completely if the payload sizes can vary.

In order to reach high data rates the data has to be combined into as few packets as possible if the network card is the limiting factor in the system. Less than 3 data

streams should not be a problem for the network card. Choosing 3 data streams with small payload sizes are more of a problem for the receiving program on the CPU.

All tests are conducted over 90 min in order to complete the tests in a reasonable time. For the packet loss only a statistical statement can be made. For example at 7 Mp/s the total transferred packets over 90 min are about 40 billion packets. The average packet loss can hence be estimated lower than 1 in 40 billion (i.e. $< 2.5 \times 10^{-11}$). Longer tests with histogramming enabled which increases the resource usage of the CPU are performed in the next section.

5.4 Histogramming the Data

Also the processing of the received data on the CPU is very important. The first test is one data stream per receiving program and testing the possible distribution of the receiving and histogramming thread onto the four logical threads of a Compute Complex. For some combinations also adding a second or even third histogramming thread could be possible.

5.4.1 Receiving Data Streams

The `genConfig` file handles the `rx-cpu-mask` which sets the thread for the receiving thread as a general parameter meaning that if only one `StreamConfig` used to configure a test, the assigned number for the receiving thread is the same for all started receiving programs. Hence four tests with four possible histogrammer configurations are run to test all 16 possible combinations for one receiving and one histogramming thread.

Each test runs for 2 hours and the packet loss can be observed in the console output. The percentage of the lost packets is calculated with `missed packets` and `received packets` as seen in the Figure 4.5 of the console output. Result of the test is Table 5.2.

The parametrization of the data streams uses the default values of 2000 B for the payload size and one paused clock cycle.

Table 5.2: Test of all 16 combinations of thread assignment in the receiving program. For each combination of the mask numbers the sometimes fluctuating packet loss in percent are stated.

histo \ rx	0	1	2	3
0	~31	~9	~8	~12
1	0	~24	0	0
2	0	0	~17	0
3	0	0	0	~15

The packet loss for some combinations fluctuates a lot but an exact percent of packet loss is not relevant because every packet loss exceeding one percent is way to high for actual use. No packet loss means in this case that per combination no packet in about 28 billion transmitted packets got lost.

In the diagonal elements of Table 5.2 can be seen that setting the receiving and histogramming thread on the same logical core results in packet loss. This is to be expected because a histogrammer already uses about 85% of a logical thread and adding the receiving threads requires even more computation time.

The first line in the table is the histogrammer set on the same thread as the fixed interrupt thread. This also increases the CPU usage to such an extent that packet loss occurs. From this observation two rules can be made for not useful combinations.

Firstly, the histogrammer should not be on the same thread as a receiving thread. Secondly, the histogrammer should also not be on the thread for the interrupt (i.e. thread 0).

5.4.2 Multiple Data Streams

To increase the processed data per Compute Complex the received data rate per receiving program has to be increased in the setup. This is done by configuring two data streams instead of one per receiving program. The second data stream gets a second *histo-cpu-mask* so that in this test three parts of the program can be assigned to logical threads. The data streams are configured as before.

In the following the *cpu-mask* names are abbreviated as *rx*, *h1* and *h2*, the latter standing for first and second histogrammer. There are 40 combinations of *rx h1 h2* if both histogrammer considered to behave the same. From all possible combinations not all will be tested. The first and second histogrammer not assigned to the same logical thread is an additional and reasonable restriction for useful configurations because one histogrammer already has an high CPU usage on a core.

After applying the sensible restriction for all possibilities, six combinations are left. This is made comprehensible in Table A.1. The remaining combinations are *rx h1 h2* = 0 1 2, 0 1 3, 0 2 3, 1 2 3, 2 1 3 and 3 1 2. These are tested with two data streams with a payload size of 2000 B and a pause of one clock cycle therefore with a data rate of about 20 GBit/s. The result of this test with 60 min runtime is shown in Table 5.3.

Table 5.3: Test results for two histogramming threads per Compute Complex. The packet loss fluctuates for the most of the data streams and is calculated to a percentage per histogramming thread.

<i>rx h1 h2</i>	packet loss <i>h1</i> /%	packet loss <i>h2</i> /%
0 1 2	~ 11	0.6
0 1 3	~ 9	1.1
0 2 3	~ 32	~ 31
1 2 3	~ 30	~ 30
2 1 3	~ 4	0.002
3 1 2	~ 2	0.001

No configuration of the receiving program is able to receive about 20 Gbit/s and processing this data without packet loss. At a first view on the CPU usage displayed in *htop*, every test has at least one histogramming thread using 100% of a thread.

The combinations 0 2 3 and 1 2 3 where two histogrammers share a physical core of the CPU seem to perform comparatively worse than the other tested configurations. 0 1 2 and 0 1 3 lose less packets in a given interval but are still 3 to 4 times worse in packet loss than the remaining configurations 2 1 3 and 3 1 2.

In the following only 2 1 3 and 3 1 2 will be further tested because for receiving higher data rates than 10 Gbit/s they seem to be the most promising. For these configurations one histogramming thread shows more CPU usage than the other one as seen in Figure 5.9.

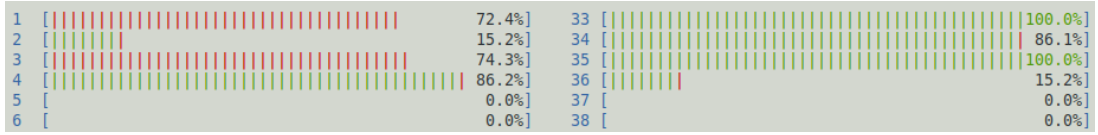


Figure 5.9: In the upper two rows the receiving program with configuration 2 1 3 is displayed and below configuration 3 1 2 in third and fourth row with 2×10 Gbit/s each.

A reason for this behavior is not clear and the high CPU usage cannot be changed by swapping the queue numbers nor by starting a second instance of the `runTests` script. Also adding a third data stream to the receiving program with a pause of 6000 clock cycles and a data rate of about 400 Mbit/s does not change the too high CPU usage of one the histogramming threads. The higher CPU usage seems to be a property of one histogramming thread running on thread 1 in a Compute Complex. Therefore the histogrammers do not behave the same but their numbering still permutes.

In order to reduce the packet loss the data rate is lowered. First one data generator is slowed with a pause of 34 clock cycles to 9 Gbit/s. For a total of 19 Gbit/s still packet loss occurs. The data rate is lowered until a point without packet loss over a longer period of time. Each following test in Table 5.4 was run for 12 h.

Configuration $rx\ h1\ h2 = 2\ 1\ 3$ shows that for a payload of 2000 B per packet the maximum processible data rate is about 15.5 Gbit/s. The relative packet loss is as low as 5×10^{-11} and could be potentially smaller. At this point a longer runtime duration is necessary to improve the statistical conclusion. The remaining configuration $rx\ h1\ h2 = 3\ 1\ 2$ has not the same relative packet loss for tested combinations of data rates as 2 1 3 as shown in Table 5.3. A reason for this unexpected observation is not known.

As a result the configuration $rx\ h1\ h2 = 2\ 1\ 3$ seems to be the best possible configuration for assigning threads from the receiving program if two data streams have to be processed. With a constant payload size of 2000 B a data rate of 15.5 MBit/s can be reached.

For the combination 2 1 3 a third data stream can be added. As seen in Figure 5.9 the CPU usage on the receiving thread is only about 15%. Three data streams with 2000 B payload are used, two streams with a data rate of 7.5 Gbit/s and one with 2 Gbit/s. The latter is assigned to the same logical thread on the CPU as the receiving thread. Over 1 h for the streams with 7.5 Gbit/s a relative packet loss of about 1.0×10^{-6} is seen. Lowering the data rate even further for these streams to 7 Gbit/s, hence a total rate of 16 Gbit/s, is not a huge improvement over the 15.5 Gbit/s for two data streams. At first sight three data streams are not really beneficial at least for a payload size of 2000 B. But further testing is needed to make a concluding statement.

Table 5.4: Test results for two data streams per receiving program with different data rates. $h1$ is always the histogramming thread with more CPU usage.

$rx\ h1\ h2$	data rate /Mbit/s		packet loss /packets		relative packet loss	
	$h1$	$h2$	$h1$	$h2$	$h1$	$h2$
2 1 3	8.0	8.0	318	0	1.5×10^{-8}	$< 4.6 \times 10^{-11}$
	7.5	8.0	0	0	$< 5.0 \times 10^{-11}$	$< 4.6 \times 10^{-11}$
	7.5	8.5	63	0	3.1×10^{-9}	$< 4.4 \times 10^{-11}$
3 1 2	8.0	8.0	237866	238583	1.1×10^{-5}	1.1×10^{-5}
	7.5	8.0	220558	237428	1.1×10^{-5}	1.1×10^{-5}
	7.5	8.5	223417	252848	1.1×10^{-5}	1.1×10^{-5}

5.4.3 CPU Usage Scaling

Until here only one payload size is used. Hence the CPU usage of the individual threads of the receiving program is measured for different payload sizes. Only one data stream per receiving program and a payload range from 250 B to 3000 B with a pause of 1 clock cycle is used to maximize the data rate. The receiving and histogramming thread get their default assignment to the logical threads of the CPU, results are in Table 5.5. The error of the CPU usage is estimated by the observed fluctuation.

Table 5.5: CPU usage of the receiving program for different payload sizes.

payload /B	CPU usage for each thread /%		
	interrupt	receive	histogramming
3000	27(2)	12(1)	86(1)
2000	38(2)	15(1)	86(1)
1000	43(2)	12(2)	82(1)
750	52(2)	22(2)	81(1)
500	64(3)	26(3)	79(1)
200	71(3)	38(3)	76(1)

The CPU usage for the interrupt thread depends heavily on the payload size. Higher payload sizes lead to a lower packet rate and decrease the usage. When comparing the measurement for 2000 B in Table 5.5 with the Figure 5.9 one can see about a doubling of CPU usage for the interrupt thread when doubling the data rate with two data streams as used for Figure 5.9. Therefore the CPU usage of the interrupt thread scales with the packet rate.

Comparing the receive thread with Figure 5.9 implies a dependency only on the payload size and not on the amount of data in a given time. For 10 and 20 Gbit/s the CPU usage is the same. Small payload sizes decrease the overhead for an additional histogramming thread on top of the receiving thread. The change in usage between 2000 B and 1000 B is likely from the necessary change of the parameters *numFrames*, *frameSize* and *batchSize*.

Lastly a histogramming thread scales with data rate. For higher data rates the usage increases and reaches a maximum of about 86% leaving a bit of overhead.

These results have to be taken into account when receiving multiple data streams per receiving program with a different payload size than 2000 B.

6 Summary and Outlook

In this thesis a working custom networking solution is studied regarding its performance. First the transmitted data has to be understood. For this a theoretical description of the payload data rate and network packet rate is successfully made. This is possible because the data is generated by an FPGA in a set pattern, this pattern can be configured to alter the data generation. A data stream can be configured from kbit/s to close to the theoretical maximum data rate of 10.24 Gbit/s and with a network stack that supports up to 100 Gbit/s multiple data streams at a time are possible.

It is observed that very high data rates are limited by the receiving end of the network. Because of the very high data rates this is expected. The result of the limitation is a loss of transmitted data which is not useful for the application and has to be reduced to a minimum. A maximum stable packet rate can be measured and is about 4 to 14 million packets per second depending on the payload size per Ethernet packet. Depending on a specified data rate the maximum achievable packet rate depends on the payload size. This payload size was measured for different data rates, as example for 60 Gbit/s is a payload size of at least 1520 B necessary to avoid packet loss over time. This would set a limitation to the structure of the transmitted data in a real application if similar hardware is used.

After receiving the data it is processed on a server CPU. A receiving program can be configured to use the logical threads of the CPU for different parts of the receiving program. When receiving and processing 10 Gbit/s on four logical threads on the CPU some limitations are given for the receiving program and found with testing. For most of the possible configurations a lossless processing is possible.

This changes when more than 10 Gbit/s have to be processed on four logical threads. Out of 40 possible configurations only few are able to process significantly more data. One configuration is able to process 15.5 Gbit/s with a relative packet loss of less than 5×10^{-11} .

For this thesis the test duration is limited by available time and especially the statistics of relative packet loss for longer runtimes could be improved. In addition only fixed payload sizes are tested. This does not represent real data with a variation in packet sizes to a full extend. A distribution in the payload size can be configured but requires the combination of different payload sizes with different data rates. Also one could analyze as example the structure of the payload which is output by an RD53A chip to have an indication for the size and variation of payload, which represents the real application data in a more precise way. If the variation of the packet size is relatively small no relevant difference is expected because the receiving program has no problem processing packet with different sizes at the same time. Lastly, the observed properties of data transmission depends on the used hardware. Changing hardware can solve existing and introduce new limitations to the setup.

Appendix

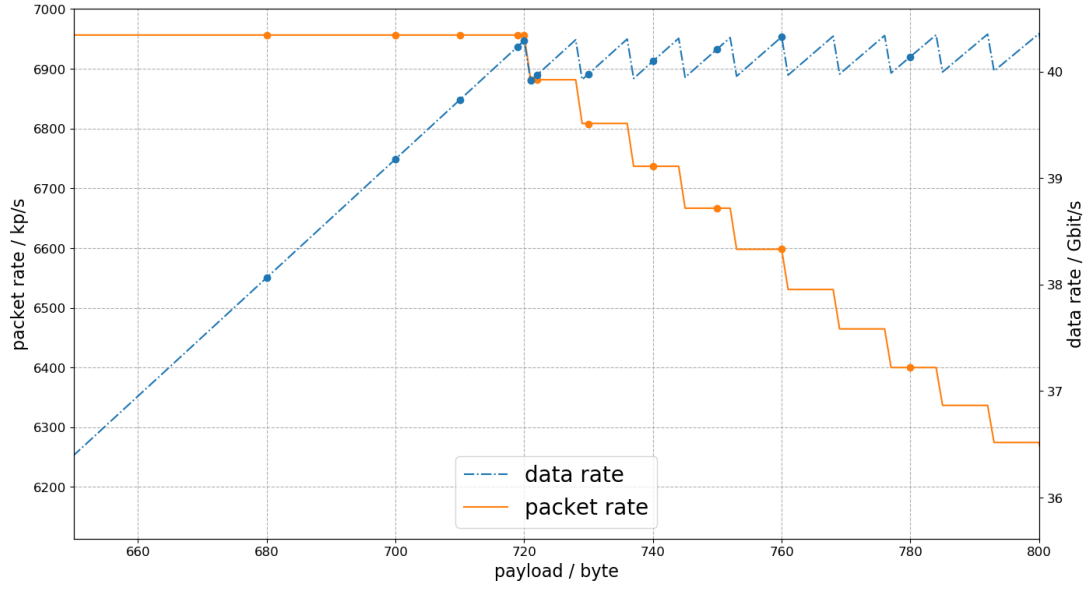


Figure A.1: Combined packet and data rate for 4 streams and a combined maximum packet rate of 7 Mp/s.

Table A.1: Possible thread assignments and which make sense for testing.

$rx\ h1\ h2$	$rx\ h1\ h2$	$rx\ h1\ h2$	$rx\ h1\ h2$
000	100	200	300
001	101	201	301
002	102	202	302
003	103	203	303
011	111	211	311
012	112	212	312
013	113	213	313
022	122	222	322
023	123	223	323
033	133	233	333

000: $h1 \neq h2$ and $h1, h2 \neq 0$

112: $h1, h2 \neq rx$

Glossary

ATLAS A Toroidal LHC ApparatuS. 1, 3, 5

CCX Core Complex (AMD Zen2 architecture). 9

CERN European Organization for Nuclear Research. 1

CfgFreddie Configuration program for the Front-End Data InterfacE. 11, 14, 15

DROP Data ReadOut Protocol. 8, 9, 16, 23

FPGA Field Programmable Gate Array. 1, 5, 7, 11, 14, 35

IPv4 Internet Protocolversion 4. 8, 9

LHC Large Hadron Collider. 1, 3, 4

OSI model Open Systems Interconnection model. 8, 9

UDP User Datagram Protoco. 8, 9, 14

Bibliography

- [1] Communications Education and Outreach Group. *LHC the guide*. February 2017. <http://cds.cern.ch/record/2255762/files/CERN-Brochure-2017-002-Eng.pdf>, retrieved 15.09.2020.
- [2] CERN. *The HL-LHC project*. <https://hilumilhc.web.cern.ch/content/hl-lhc-project>, retrieved 21.09.2020.
- [3] The ATLAS Collaboration. *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*. 15. June 2018. <http://cds.cern.ch/record/2285585/files/ATLAS-TDR-030.pdf?version=2>, retrieved 15.09.2020.
- [4] The ATLAS Collaboration et al. *The ATLAS Experiment at the CERN Large Hadron Collider*. 2008. <https://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08003/pdf>, retrieved 15.09.2020.
- [5] G Aad et al. *ATLAS pixel detector electronics and sensors*. 2008. <https://iopscience.iop.org/article/10.1088/1748-0221/3/07/P07007/pdf>, retrieved 15.09.2020.
- [6] Helmuth Spieler. *Semiconductor Detector Systems*. Oxford Science Publications, 2005. ISBN: 978-91-637-4473-0.
- [7] *VCU128 Evaluation Board User Guide*. 21. December 2018. https://www.xilinx.com/support/documentation/boards_and_kits/vcu128/ug1302-vcu128-eval-bd.pdf, retrieved 01.09.2020.
- [8] ITU Telecommunication Standardization Sector. *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model*. July 1994. <http://handle.itu.int/11.1002/1000/2820>, retrieved 31.08.2020.
- [9] Ieee standard for ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, 2018. <https://ieeexplore.ieee.org/document/8457469>, retrieved 31.08.2020.
- [10] University of Southern California Information Sciences Institute. *RFC 791*. September 1981. <https://tools.ietf.org/html/rfc791>, retrieved 31.08.2020.
- [11] J. Postel. *RFC 768*. August 1980. <https://tools.ietf.org/html/rfc768>, retrieved 31.08.2020.
- [12] J. De Gelas. *AMD Rome Second Generation EPYC Review: 2x 64-core Benchmarked*, pages 1–4. August 2019. <https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen>, retrieved 01.09.2020.
- [13] Hisham Muhammad. *htop*. <https://github.com/hishamhm/htop>, retrieved 06.09.2020.