GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Fakultät für
Physik

Bachelor's Thesis

Automatisierung des Stresstests der ITk-Pixel FELIX Auslesekette

Automatisation of the ITk-Pixel FELIX Read-out Chain Stress Test

prepared by

Timo Pospiech

from Goslar

at the II. Physikalischen Institut

Thesis number: II.Physik-UniGö-BSc-2024/03

Thesis period: 2nd April 2024 until 5th July 2024

First referee: Prof. Dr. Arnulf Quadt

Second referee: apl. Prof. Dr. Jörn Große-Knetter

Abstract

Diese Bachelorarbeit beschäftigt sich mit der Vereinfachung des Stresstests der ITk-Pixel Auslesekette. Um dies zu erreichen, wird zu Beginn das Wuppertaler Microservices Framework implementiert und zudem die Stress Test Software erweitert. Alle zur Verfügung stehenden Teile des Microservice Framework konnten implementiert werden. Zudem wurde ein Script geschrieben um die Bedienung der Microservices zu vereinfachen. Die Stress Test Software wurde auch erfolgreich erweitert. Es wird eine neue Version von YARR unterstützt, weitere Information der Frontend Register werden ausgelesen, und Scans mit externen Triggern werden unterstützt. Weiter wurden Tests durchgeführt, welche diese neuen Funktionen verwenden. Die Tests zeigten, dass der Wechsel von den alten Versionen von YARR und FELIX zu den neuen, die Leistung positiv beeinflusst.

Stichwörter: Auslesesystem, DAQ-Kette, Docker, Microservice Framework, Stresstest

Abstract

This bachelor thesis addresses how to simplify the stress test of the readout chain. This is accomplished by first implementing the Wuppertaler microservices framework and following that, expanding the features of the stress test software. All available parts of the microservices framework could successfully be implemented. Additionally, a script was developed to ease the use of the microservices. The stress test software was expanded to support a new YARR version, read additional information from the frontend registers and support external trigger scans. Further tests where performed where these new features are used. The tests reveal that switching from the old version of YARR and FELIX to the new ones, results in better performance.

Keywords: readout system, DAQ-Chain, Docker, microservices framework, stress test

Contents

1 Introduction

During the Long Shutdown 3 from Dec. 2025 to Feb. 2029, the LHC will be upgraded to the High Luminosity Large Hadron Collider (HL-LHC) [1]. And with the increase of luminosity comes an increase of data that the detectors generate. For that, it needs to be ensured that the new readout system is capable of handling these amounts of data before it gets deployed.

The Inner Detector at ATLAS is one of the components which will be upgraded. The new tracking detector is called the Inner Tracker. To ensure that it is ready, a large amount of testing of its various systems is needed. One of these systems is the readout chain, as it too will change in order to handle the increased data production. For that, the readout chain will employ fibre optic cables to provide the bandwidth needed to read the detectors out. At the receiving end of the fibre connection lays the FELIX system. Which is used to receive the optical signal and transfer it to the DAQ system. To be sure that the system is up to the task, it needs to be stress tested.

The current implementation of the stress test requires the handling of the frontend and data aggregator emulation as well as the DAQ software. The goal of this project is interfacing the overarching system to the microservice framework and use it for the current tests of bigger hardware systems. The microservice framework is capable of setting up the emulators, preparing the test data, configuring the system for the DAQ software, performing tests and collecting and evaluating the results.

Additionally, the application stress test software will be extended to include more features. This will allow performing a broader variety of tests with additional information to be gained.

2 Background

2.1 The Large Hadron Collider

The Large Hadron Collider (LHC) [2], is currently the biggest particle collider in the world. LHC is a circular particle accelerator of the synchrotron type. As such, it possesses a circumference of 26.7 km and is capable of providing a centre of mass energy of 14 TeV [2], but currently uses a centre of mass energy of 13.6 TeV. As the name states, the LHC collides hadrons against each other. The main type of hadrons used are protons. At the LHC, there are four experiments, two general purpose detectors ATLAS and CMS and two specialised detectors ALICE and LHCb.

2.2 The ATLAS Experiment

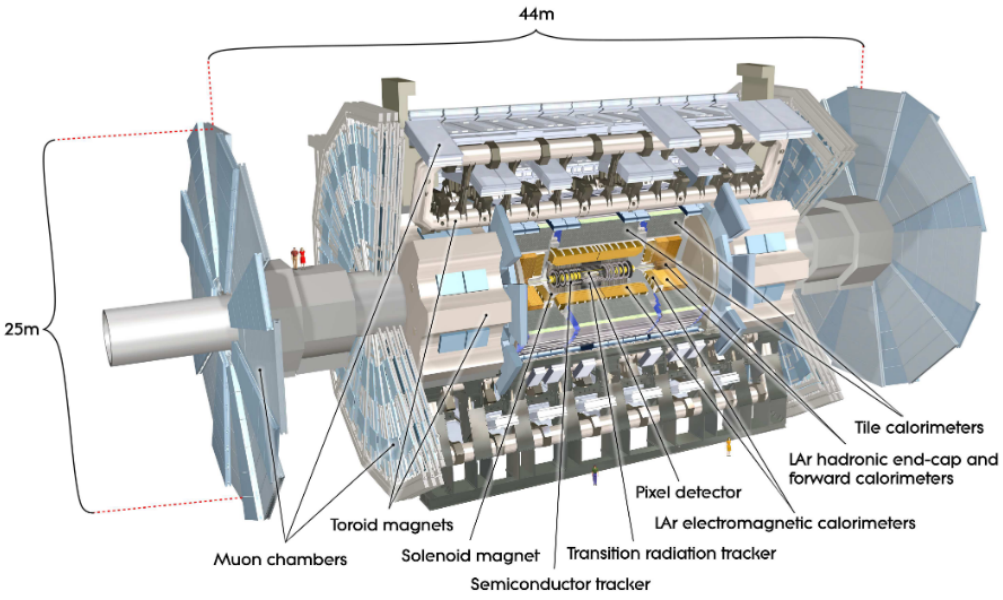


Figure 2.1: A cross-section of the ATLAS detector. ©CERN

Atlas is one of the four main experiments currently at the LHC. The Atlas detector comes in at a diameter of about 25 m, a length of 44 m and a mass of 7000 t. Particles created by the collision travel in all possible directions. Therefore, the detector covers most of the solid angle, except for directions occupied by the beampipe at high rapidities.

2.2.1 The Composition of ATLAS

The current Atlas detector [3] is composed of various layers, with each layer having their own purpose. The information of each layer is combined to identify particles and reconstruct an event. To ensure that as much of a solid angle as possible is covered, the detectors at Atlas are designed by having a barrel, to detect particles with low pseudo-rapidity and two endcaps to detect particles with high pseudo-rapidity.

Inner Detector

In the most inner part, right outside the beampipe lays the Inner Detector (ID) [4]. The ID is enclosed inside a solenoid magnet, which generates a magnetic field of 2 T, and is used to track charged particles. The track of a charged particle possesses information of the direction, momentum and charge of the particle and allows reconstruction of the origin vertex. For the later energy measurement, it is important that the energy loss at the ID is minimal. To achieve that, Atlas utilises silicon based strip and pixel detectors as well as a gaseous transition radiation tracker [3]. For the barrel region, there are three layers of pixel detectors stacked, followed by four layers of strip detectors, and finished with 73 planes of straw tubes interleaved with fibres for the transition radiation tracker. While each of the endcaps are composed of three stacks of pixel detectors, followed by 9 layers of strip detectors and ended with 190 planes of straw tubes.

Calorimeter

Following the solenoid magnet are the electro-magnetic calorimeter (ECal) and the hadronic calorimeter (HCal). The purpose of the ECal is to measure the energy deposition of produced electrons, positrons and photons. Mirroring that, the HCal measures the energy deposition of hadrons. The shape of the deposition infers to how much energy a particle had. The ECal induces an electromagnetic shower, while the HCal induces a hadronic shower. Atlas uses sampling calorimeters for both cases. Sampling calorimeters are made of two alternating layers of passive and active media, with the former inducing the particle shower and the latter generating a measurable signal. The ECal employs liquid Argon [5] as the active and Lead as the passive medium. For the HCal, liquid Argon is

still used as the active medium for the endcaps, but for the passive medium Copper is used. The barrel region uses steel as the passive medium and scintillating tiles for the active medium.

Muon Chamber

Due to the way muons interact with matter, they penetrate the calorimeters. So to detect and measure them, the outermost part of the ATLAS detector is used. It consists of the toroidal magnets and the muon chambers. The toroid magnets create a magnetic field, which, like in the ID, curves the trajectory of the muons. Inside the toroids magnetic field are the muon chambers, which track the muons trajectory. The muon chambers utilise monitored drift tubes for the barrel and cathode strip chambers for the endcaps.

The aforementioned structure of the ATLAS detector can be seen in Figure 2.1.

2.2.2 Data Readout at ATLAS

Inside the LHC, two proton bunches each containing 10^{11} protons collide every 25 ns. This leads, together with the sheer amount of readout channels, to the generation of huge amounts of data, which is too much to be processed. Due to that, we need to ensure that only events that interest us, such as ones, where an isolated muon is created, get read out fast.

Trigger

To ensure that only interesting data is saved, ATLAS utilises a multistep trigger system, which is spread out over its different subdetectors.

The first level is on the hardware itself. It provides a first filter for the event selection, such as if something is detected and if it possesses enough momentum. If an event passes the first level, it creates a Region of Interest (RoI) and gets read out to a nearby system, at which the level two trigger resides. It reconstructs the event with all available detector data inside the RoI at full granularity and precision. If the event passes through that, it gets passed to the last stage, the event filter. It is carried out at a local system in which offline analysis procedures are used.

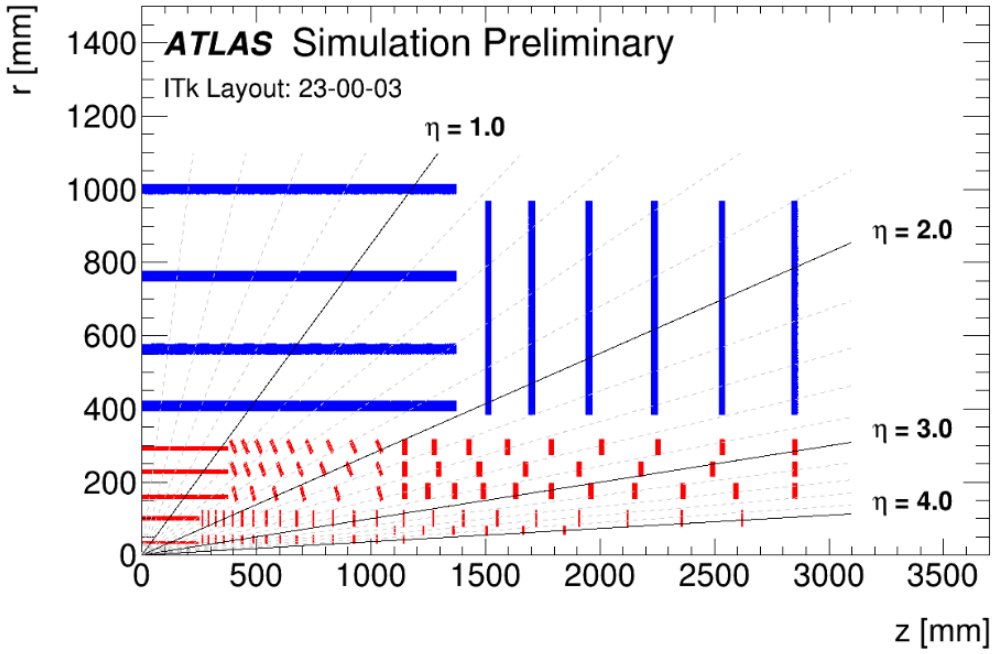


Figure 2.2: Cross-section of the ITk detector [6], in red the location of the Pixel detectors and in blue the location of the strip detectors. The dashed lines are the lines of constant pseudorapidity.

2.3 Inner Tracker

The new ATLAS Inner Tracker (ITk) is an important part of the ATLAS detector upgrade. It is the first detector a particle created in the collision passes through. The ITk will replace the current ID and be similar to it. It is built from Pixel and Strip detectors. Due to the increase in luminosity, it needs greater performance and durability than the current ID. For the Pixel side, hybrid pixel modules consisting of Pixel sensors made from high resistivity silicon and frontend read-out chips which are fabricated in CMOS technology will be used. The front-end chip is carried out within the framework of the RD53 foundation. The Pixel part of the ITk will approximately contain 1.4 million Pixels on 9400 modules with an area coverage of $13\,\mathrm{m}^2$. The strip sensors will be silicon micro-strip sensors. They are situated just outside the Pixel detector and will have an approximate silicon area of $165\,\mathrm{m}^2$ [7]. Like the rest of the ATLAS detector, the ITk is divided into a barrel and two endcaps. The barrel consists of 5 layers of pixels and 4 layers of strip detectors. The pixels close to the collision point are parallel with the beamline. These will track the events with $|\eta| < 1.4$. While those farther away will be angled to maximise the area covered. The endcaps consist of 5 layers of pixel detectors and 6 layers of strip detectors. The layout can be seen in Figure 2.2.

2.4 FPGAs

To perform computational tasks, one needs the necessary hardware. This is nowadays accomplished by application-specific integrated circuit (ASIC) or field programmable gate array (FPGA).

ASICs are chips which are focused on a particular task. They are made by printing the entire circuit directly on the board. This means that once they are made, the task which they perform can not be changed. They are cheap to manufacture in great quantities and boast great performance in their respective tasks.

An FPGA is a microchip which behaves in a way which is defined by the user, such as emulating a multitude of other hardware components or some completely original task. This is accomplished by uploading the desired firmware into the FPGA. The desired behaviour is described by the user using a hardware description language (HDL). The HDL describes concurrent processes, unlike normal sequential programming. Which is needed since the logic of the chip also operates fully in parallel. With the HDL description and the targeted FPGA, the software synthesises it into firmware. That firmware can then be uploaded onto the FPGA. The FPGA itself is composed of a multitude of look up tables (LUTs) arrays, flip-flops, full adders, routing channels and I/O pads. For more specialised applications the FPGA may contain further components to assist the task. LUTs receive multiple signal inputs and output a corresponding output signal for each possible combination of the inputs. The corresponding output can be defined by the user. With that, multiple LUTs can be arranged to behave like any logic circuit. This property makes them very flexible and useful. With this flexibility of having any logic circuit, the user can utilise the available resources of the FPGA in many different ways. FPGAs are typically used for development purposes as they are reprogrammable. This is done by changing the values of the LUTs and how they are connected to the routing channels. This is opposed to ASICs, as ASICs can not be changed after production.

2.5 Application Programming Interface

Many applications are built upon other applications, this is an important part of software development. For that, a standard of how to interact and use the application is necessary. To help the developer with this task, many applications provide an application programming interface (API). An API tells the developer how to communicate with the software.

There are many different kinds of API. One of them is the software library. Software

libraries tell the user how to use different instructions of the program. With the rise of the internet, Web APIs find more and more use. Web APIs are used for end clients to interact with a web server. They use the HTTP protocol in order to send messages in the JSON or XML format. The web server receives the HTTP request and sends responses depending on the type of content which is executed.

2.6 DAQ Chain

As part of the ATLAS upgrade, the data acquisition chain (DAQ chain) of the ITk described in Section 2.3 needs to be improved as well. In order to handle the increased data throughput, it is intended to deploy fibre optic cables. The components of the chain are explained further below. The chain consists of the frontend, which receives the signal from the sensor and sends that data to a nearby optoboard. There it gets turned into an optical signal and then sent off the ATLAS detector to the nearby counting room. There, a FELIX card, which is attached to a server, turns the optical signal back into an electrical signal and sends the data via the PCIe connection to the server. The server runs software that is used for analysing the data. The chain is also capable of the reverse, it can transfer command signals sent from the server to the pixel detector. A simplified overview of the setup DAQ Chain can be seen it Figure 2.3

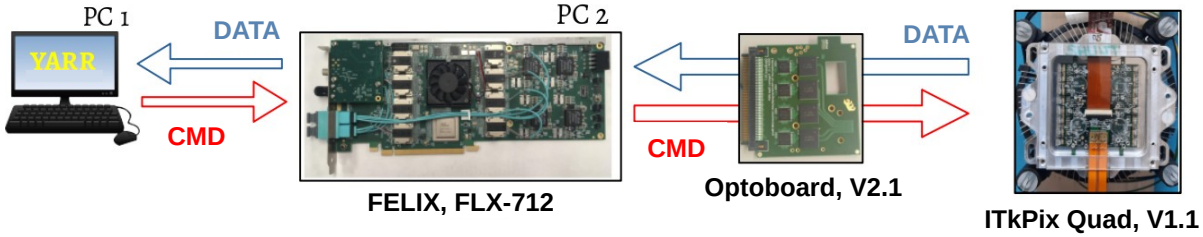


Figure 2.3: The simplified ITk Pixel DAQ chain setup.

2.6.1 Pixel Readout

For the readout of the Pixel detector, it is planned that the bare module, which consists of the sensor and the frontend (FE) chip, where each pixel of the sensor is bump bonded to the frontend chip directly, gets glued to a Printed Circuit Board (PCB). This is followed by the frontend chip being wire-bonded to the PCB. It is expected that a bandwidth of 3.97 Gbps is required for the innermost layers of the pixel detector [8].

When a charged particle passes through the pixel sensor, electron-hole pairs are created in the semiconductor material of the sensor, which induce a current in the pixel electrodes. This current gets read out by the frontend. The frontend possesses built-in circuity to integrate the current to get the injected charge and then determines how long the charge is over a set threshold to get the time over threshold (ToT). The ToT gets encoded as a binary signal and sent together with the position of the hit over up to four data lanes of 1.28 Gbps each of the detector.

A digital signal is a binary representation of the previous analogue signal. And then the data gets transferred over up to four data lanes of 1,28 Gbps.

The frontend is called ITkPix, and the prototype of it is called RD53A [9].

2.6.2 Optoboard

The Optoboard [10] is intended to be on the Patch Panel 1 on the Atlas detector, so that the data can be transferred quickly and with high signal integrity to it. An Optoboard consists of four GBCRs, four lpGBTs and one VTRX+. The GBCR receives the digital signal and is used to restore some signal integrity which has been lost during the transfer. From the GBCR, the signal gets passed to the corresponding lpGBT. In the operation mode chosen for ITk, the lpGBT receives up to 7 signal groups of input signals and aggregates them to an output link with a speed of up to 10.24 Gbps. The VTRx+ receives the electrical signal from the lpGBT and generates an optical signal, which it then transmits.

2.6.3 FELIX

The FELIX PCIe card will receive the optical signal of the optoboard, transfer it back to an electrical signal and share the received data with the host system using the PCIe interface. The intent behind FELIX is, that it is capable of receiving signals of all kinds of detectors. This can be done due to the FELIX card being based on an FPGA. Depending on the connected detector, different firmware can be loaded on the FPGA. This allows for a FELIX card to be able of receiving the different signals and transfer the information to the host system. This is an advantage compared to the current system, which needs separate hardware to receive the signals of the different kind of detectors.

FELIX is already deployed at Atlas and as part of the upgrade, it is planned that more components of the detector switch to the FELIX system. In regard to that, FELIX is currently being further developed.

2.7 Software

The following sections detail the software used in this thesis.

2.7.1 YARR

YARR (Yet Another Rapid Readout) [11] is a software with the purpose of reading out various frontend chips, such as the ITkPix and the RD53A. It can also be used for testing the frontend chip. It sends triggers to the frontend, and then receives and decodes the response and fills the hits into histograms. By comparing the total number of triggers sent with the received amount of triggers, one can see the efficiency/loss of the tested frontend and by extent of the tested system.

Because YARR is not capable of connecting directly to FELIX, an intermediate software is used for the backend. felixcore and the newer felixstar fill that role. Later, felixstar is used to serve the detector data to a network for processing. They receive the data from the FELIX card and after processing it to the right format hand it over to YARR.

In order to perform scans in YARR, several files need to be provided. These files include configuration files for the controller, the connectivity and for what kind of test is performed. All of these files are easily readable in the JSON format. Depending on what type of test is being performed, the structure of these files needs to be adjusted.

The configuration file for the controller contains information on how the backend works, such as which felixstar port and settings to use. The configuration file for connectivity declares how the setup is built, such as which frontends are enabled and how the frontend groups and lpGBT groups are connected with the system and each other. This is needed so that YARR knows where it needs to send commands.

The scan configuration file first declares how the data is built and analysed, and then it specifies how the triggers get sent. This is done by using serval loops. These loops consist of the mask, the core and the trigger loop. In a digital scan, these loops contain information on how many times the triggers are supposed to be sent. Inside each mask loop, some pixels get masked so that they send no data. The core loop operates similar to the mask loop. During each run of the core loop certain columns get deactivated so that the pixels inside these columns send no data. This is followed by the trigger loop, where the frequency at which the triggers get sent as well as the amount of triggers is specified. Unlike during operation inside the Atlas detector, digital scans send bursts of 16 triggers in quick succession. This is to ensure that one of the 16 triggers captures the hits injected into the sensor via a test mechanism.

Another operation mode is the external trigger scan. There, the configuration file

specifies inside a single loop that external triggers get sent. That means, that YARR does not send triggers itself. When an external trigger scan is performed, YARR goes into an endless loop, and listens for events to decode until the user interrupts it. While YARR is inside the loop, the user sends their own triggers to the frontend.

In order to use YARR, the computer on which the software is run needs access to the system where the FELIX PCIe is. This allows YARR to run on the same computer as the FELIX PCIe, or on another computer which is connected to the same network, on which the FELIX PCIe is hosted.

2.7.2 Docker

Docker is a service hosting software where software is run inside "containers". It is a form of virtualisation, like virtual machines, as it emulates a whole computer with operating system, filesystem and system resources [12]. But it boasts greater performance than virtual machines. This is accomplished by sharing its resources directly with the host system, as well as natively communicating with the system kernel.

Docker can be separated into several parts, the dockerfile, the image and the container. In order to get an application started, it is necessary to get an image of said application. Inside a dockerfile is declared what operating system is used, which system resources are needed and which applications get started. Using that dockerfile an image gets created. This image is a representation of all the necessary files, applications and dependencies which are needed to run the application. Images are structured in layers, with layers being able to be shared between images. This makes images disk space efficient. Docker uses these images to create containers. That container is an instance of the image and runs the application. Docker containers share resources with each other and can be started/stopped as the user desires. With the resource and layer sharing, docker images typically only contain a single application. This makes it easy for the user to deploy services/applications as desired. For a user, Docker is easy to use as all they need is the image of the desired application, and all the dependencies, configurations and setup steps for the software are handled by Docker.

2.7.3 Wuppertal Microservices framework

The Wuppertal microservices framework is a collection of several Docker images for operating the ITk DAQ chain [13]. This collection of software is provided via docker images for each separate feature. This approach allows for the deployment of individual features as microservices.

Currently, there are several microservices already ready to be deployed.

The intended workflow of the microservices framework is, to generate a runkey with the desired test setup in terms of system and frontend configuration. That runkey gets put into a database. With the usage of the DAQ microservice, the desired runkey gets loaded, and with the specification of the runkey the test is performed. The results get loaded into another database.

Most microservices can be accessed with a web user interface (UI). Some microservices provide an API to make them available for the user and allows for easy integration into other software.

The modular nature of the microservices allows the user to deploy only the needed/desired microservices. This allows for ease of adaptability and integrating the microservices into many possible hardware setups.

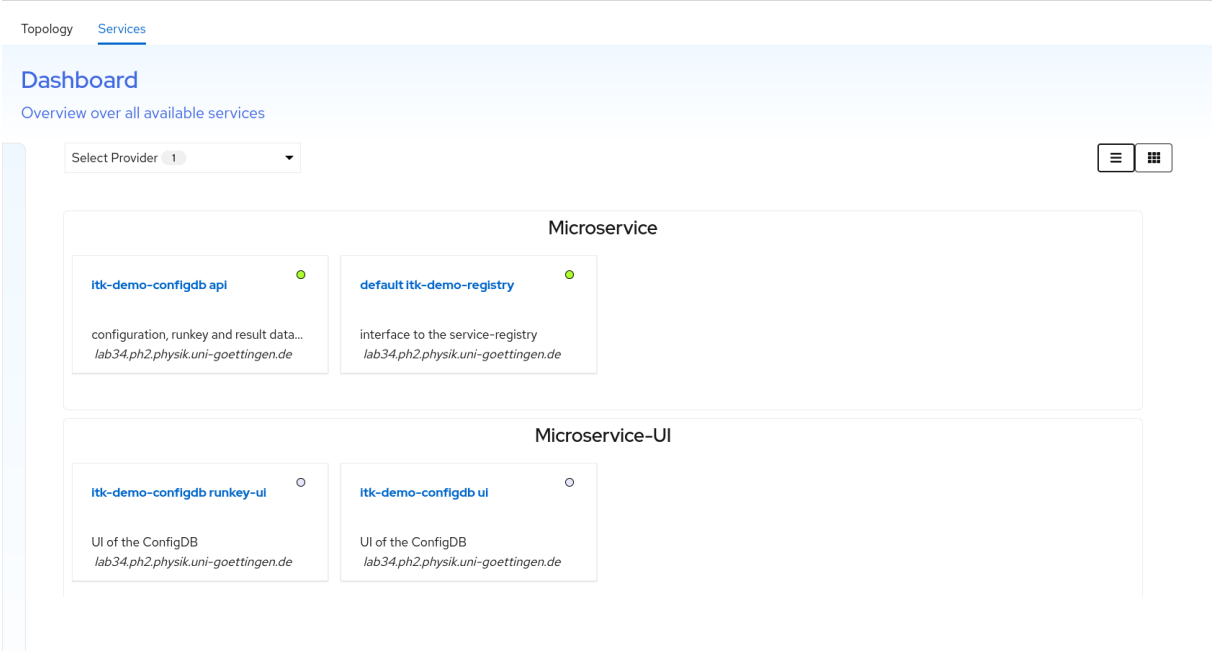


Figure 2.4: The dashboard of the WebUI. Shown are the deployed microservices as well as the access to other corresponding UI.

WebUI

The WebUI is a user interface which is accessible over a web browser. It shows an overview over the status of all deployed microservices, as well as easy access to their corresponding UI see Figure 2.4. This is advantageous, as the UI of some of the other microservices do not possess a static location. With the deployment of this microservice, it is easily accessible. Furthermore, it shows the currently loaded runkey and with that gives an

overview over the setup. The WebUI is hosted in a docker container and is exposed to the user via port 80 of the localhost.

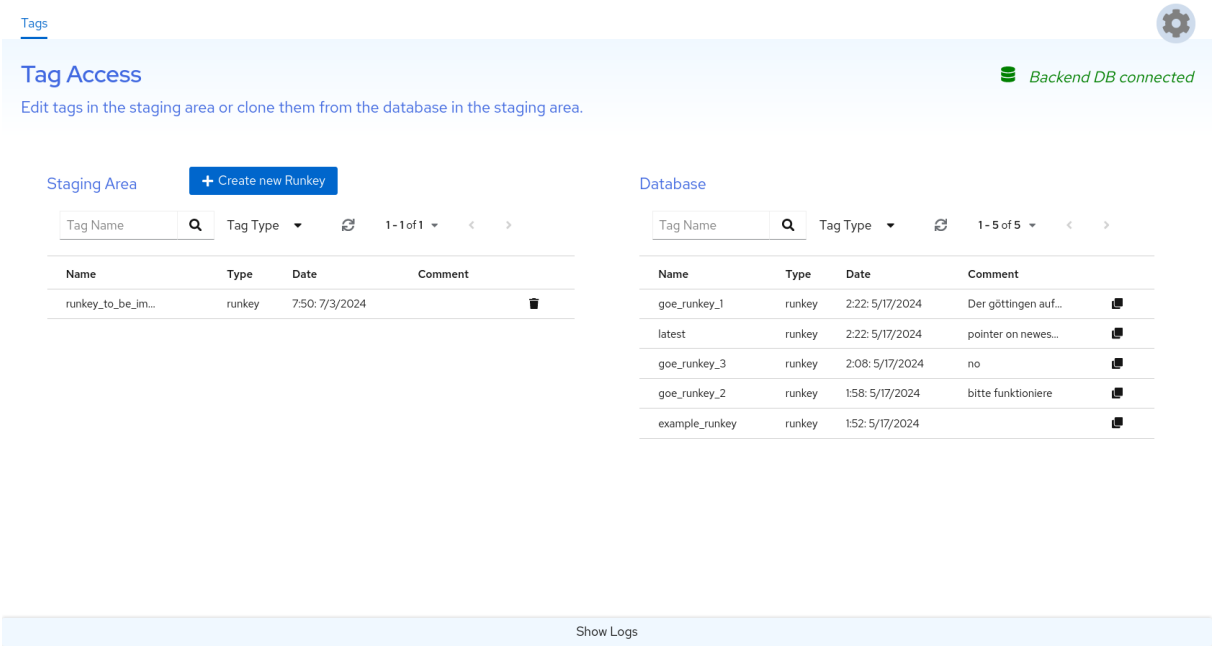


Figure 2.5: The WebUI of the runkey overview. On the left an overview over all staged runkeys. On the right all runkeys in the database are shown.

Runkey Database

The runkeys are an important part of the workflow. A runkey contains the information of the setup. How it is structured can be seen in Figure 2.6. In order to easily manage them, they get stored in a database. The runkey database has a UI, which allows for easy management of the runkeys see Figure 2.5. The runkey workflow is divided into two steps, the staging area and the database. A new runkey gets put into the staging area. There, the runkey can be modified. For that, one UI is provided. There, the user can create and modify runkeys in a list like fashion, but the UI also provides the possibility to create the runkey with drag and connect. After the runkey is finished, it can be pushed from the staging area to the database. Once a runkey is on the database, it can be no longer modified. Another UI is available for the overview of all the runkeys in the database. For all the aforementioned features, an API is provided. With the use of the API, the user can write their own software to handle the runkey generation for their used setup.

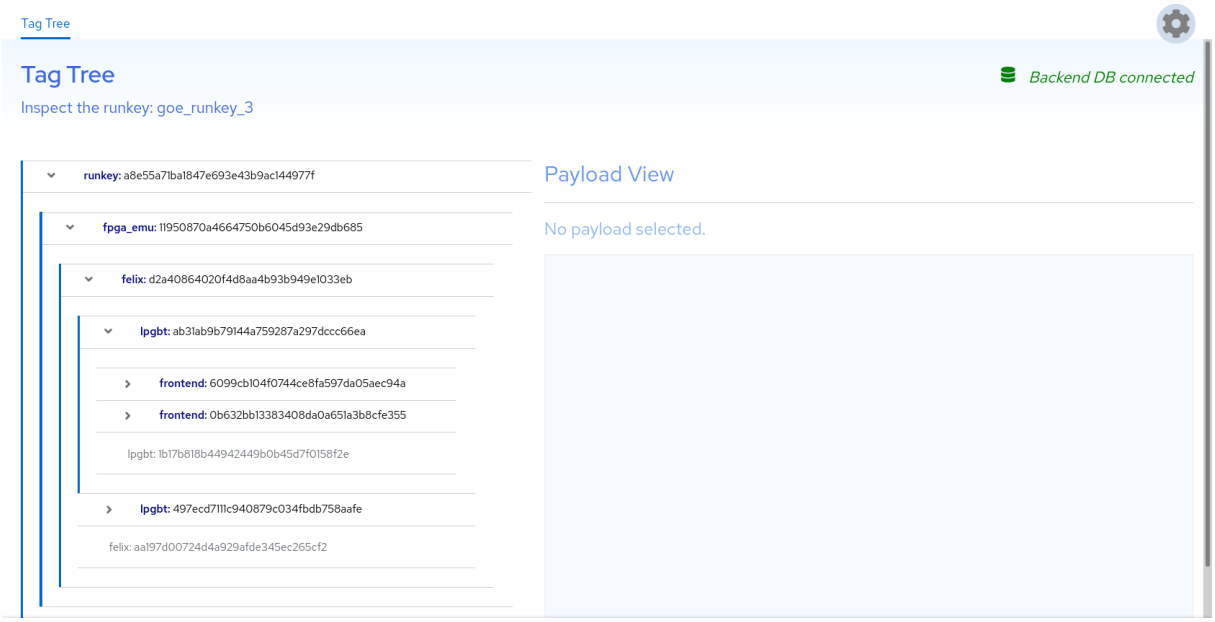


Figure 2.6: An overview over the runkey structure.

DAQ Container

With the service of the DAQ container, the test is performed. With the UI, a runkey can be selected, and based on the runkey a corresponding test gets performed. Further, the results get saved inside their own database.

Result Viewer

The result viewer allows for checking the results of the scans performed with the DAQ container. It loads the results from the database and provides the user a UI to work with the results, like sorting by the desired parameters.

FELIX container

The FELIX container possesses all of the necessary software which is needed in order to operate the FELIX card and perform the YARR scans. It also provides an API, with which FELIX can be accessed and managed.

3 Setup

One kind of test for hardware and software is the stress test. During a stress test, the tested application is pushed to its limits. This is usually done by putting the system under maximum load for an extended amount of time. This is done as during operation the system will be put in all kinds of different situations and it needs to be ensured that it does not fail.

Stress testing the ITk DAQ chain, as described in Section 2.6, poses various difficulties. Such as, how to support the necessary amount of optoboards and frontends to perform such tests and how to generate the data to perform the tests. In order to solve these problems, FPGAs are used to emulate the frontend chips and the optoboards. This approach is advantageous, as the flexibility of FPGAs, as well as the small size, allows for easy scaling and adaptivity. These are important features as changes can easily be implemented and tested.

3.1 Stress Test Hardware Setup

This setup is composed of two FPGAs which are used to both emulate the Optoboard as well as the frontend chip. As the frontend chip, the RD53A is emulated. The RD53A can be configured using registers. The boards used for the Optoboard and frontend emulator

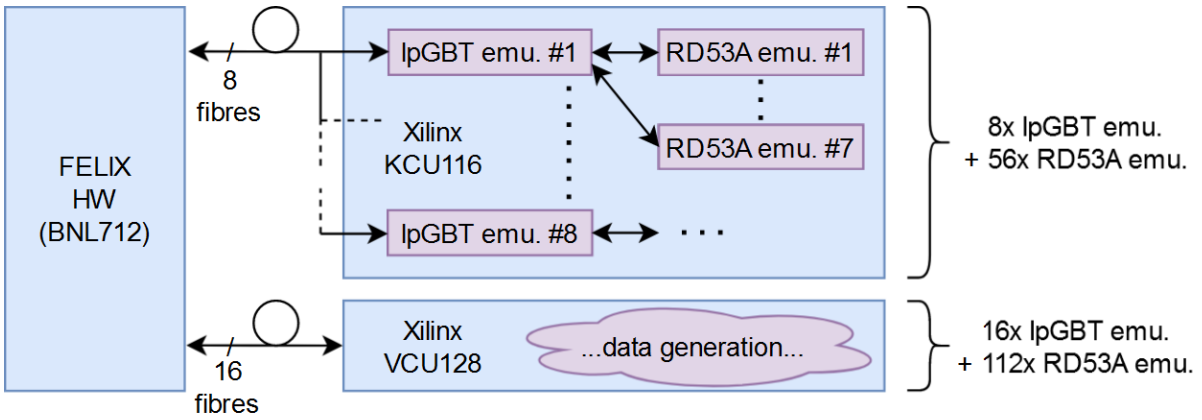


Figure 3.1: Schematic of the used setup.

are the KCU116 and VCU128, both from AMD. It is to note that because of the necessity of the high data speeds, it is important that the FPGAs are capable of reaching/surpassing the necessary transfer speed. The KCU116 and VCU128 are connected via multiple fibre optic cables to the FELIX PCIe. Together, the FPGAs emulate 24 lpGBTs with each lpGBT having 7 frontends connected as seen in Figure 3.1. This results in a total count of 168 frontends, which are available for testing the system. The FELIX PCIe contains an FPGA, on which the necessary firmware to receive signals from the RD53A is running. The PC, to which the FELIX PCIe is connected to, is running YARR. With YARR the tests are performed. Besides the fibre optic connections, the FPGAs are connected with an Ethernet connection to the host machine. This connection is used to program the FPGAs and write to the emulator registers.

3.2 Stress Test Software

The stress test software [14] allows for easy testing of multiple frontends using the stress test setup described in Section 3.1, for all kind of configurations. The software provides an easy way to perform YARR tests, as the management of the setup is condensed into a few simple scripts, instead of needing to handle YARR and felixcore and their configuration directly.

The scans of the perform_scan software are performed using YARR. Based on a central configuration file, the software performs the test automatically. Test events get generated and uploaded to the frontend, this is followed by starting the FELIX backend and with the successful start of the backend, YARR gets started, and a scan is performed. In addition to the results from YARR, the application provides further results such as how many triggers got lost during the scan. The central config. file contains information about which YARR and FELIX backend versions to use, which controller and connectivity needs to be used, what frontends are to be tested and where to put the test results.

Another set of applications the software provides is for handling the test setup. This includes the ability to read and write to the registers of the RD53A emulator, loading configurations to the lpGBT and RD53A, initialising the components of the setup, creating and uploading custom hitmaps to the RD53A emulator and resetting the high-speed link if needed. It is to be noted, that because the RD53A is emulated the registers to which the applications read/write are not the RD53A registers as implemented in the real chip. Instead, they write to custom registers, called the frontend registers and the board registers. These change the frontend emulator and board behaviour, respectively.

The software bundle also provides a Python library. It provides the features of the

previously described applications in the form of classes and methods. This allows for further automatisation and implementation of custom behaviour. This gives the user the power to write their own Python scripts to perform automated scans.

4 Stress Test Automatisation Development

4.1 Setting up the Microservices Framework

There are multiple reasons for adding the microservices framework to the stress test setup. The usage of the microservice framework allows to easily configure the frontends as well as handling the test results within a Graphical User Interface. Furthermore, the framework continues to be developed. With the help of Docker, updates to the testing software like YARR are swiftly integrated and do not need to be performed manually. Furthermore, it contains a database, with which the test result can be more easily organised and used. Using a uniform system allows for easier comparison with other testing sites. Moreover, scaling the system will be easier to handle. Due to the many advantages and features of the microservice framework, the decision was made that it gets adapted into the test system in Göttingen.

For implementing the microservices, scripts are written in order to get the system running. The first script sets the environment variables and checks if the necessary dependencies, such as Docker, are installed. Another script is to get the microservices running. This is a Docker compose file. A Docker compose file contains the information about how to start the microservices, as well as how to connect the network of the microservices so that they communicate with each other. The containers are connected by defining an internal network which they use. Inside the docker compose file, all currently available microservices are implemented. As new microservices get added, the compose file can be easily extended to integrate them without an issue.

Another script created for this thesis is for the generation of runkeys for the test setup which is used. The runkey generation script utilises the API of the runkey database microservice. The API is based on HTTP Requests, so the script sends the necessary request to generate the runkey. It was decided that Python is used, due to the need of handling HTTP requests. The requests library for Python provides functions to work with such structures. As the runkey is a representation of the setup, the structure of the

script mirrors the structure of the setup. The setup can be represented as nested loops. These loops are for the two FELIX connections, the 12 lpGBT links for each connection, and the 7 frontends for each lpGBT lane. The scripts generates the runkey with the corresponding data for each component, such as how the component is related to its peers and what data they contain. The script can easily be modified if a change of the test setup happens or the user wants to test only a part of the setup.

4.2 Extension of the Stress Test Software

Part of the work time was dedicated to extend the features of the stress test software. The extended features include more information about the tested frontend, support to perform external trigger scans and making the software compatible with new versions of YARR.

4.2.1 More Frontend Information

More information about the frontend allows the user to observe more about the behaviour of the whole system. This assists in getting a better understanding of what is happening inside the system and at what point errors could occur. Before the thesis working time, many parameters were gathered by the emulated registers of the RD53A, but are not conveniently available. For that reason, several functions are implemented so the data can be read out and provide useful feedback for the user. The chosen parameter to provide the user are the number of triggers lost during downlink and uplink, the occupancy of the link, the transfer time and the amount of false positive and false negative hits.

The separation of lost triggers into **uplink** and **downlink loss** allows one to understand where the loss of triggers occurs. This is done by comparing the total amount of sent triggers/events from YARR with the amount of triggers each frontend has sent. The difference between these two quantities shows the number of triggers the board has not received. The **link occupancy** shows how much the system is under load. It is calculated by dividing the amount of data packages sent by the total number of packages, which could have been sent. The values for the total packages and data packages are gathered from the frontend registers of the RD53A. Inside the frontend register, the number of packages is saved as a 64-bit integer. However, due to technical constraints, only 32 bits at a time can be read out. In order to get the number of packages which were sent, the first 32 bits and latter 32 bits get read separately and then combined. This needs to be done for the total packages sent and the data packages sent. Total packages include the data packages and idle packages. The **transfer time** gives the user the information about how long

a frontend operates. It is calculated by dividing the total amount of data sent by the data transfer speed. The data transfer speed for the RD53A is 1.28 Gbps which needs to be multiplied by the link efficiency to get the data transfer rate. The total amount of data sent in bits, is the total amount of frames sent times 66, as a frame consists of two packages of 32 bits and a 2-bit header. The total amount of frames consists of data frames and register frames. The data frames can be calculated by dividing the total amount of packages sent by two. The register frames are calculated by dividing the amount of data frames by the nRatio. The nRatio is the ratio between data frames and registers frames. The registers of the RD53A store the currently used nRatio and it gets read out using the frontend registers. And the **false positive** and **false negative hits** allow a deeper understanding of what kind of error occurs. **False positive hits** are the amount of pixels where YARR reads out hits at places where no hit should be present. **False negative hits** are the counterpart, they give the amount of reported missing hits in places where hits should be present.

4.2.2 External Triggers

One other subject is the adding of support for external triggers. During external trigger operation, YARR does not send any triggers to the frontend. Instead, a trigger generator on board level generates triggers for all frontends. When the triggers are generated by the board, they get sent evenly spaced with a fixed frequency. The frequency is given indirectly as the number of bunch crossings between two triggers. The trigger generator supports a fixed and an indefinite amount of trigger generation. During that mode of operation, YARR only reports about the amount of received triggers and does its usual event analysis. This method of operation eliminates the possibility that triggers get lost during the downlink. Furthermore, in basic operation YARR operates two streams for downlink and uplink separately. If the two streams are not in sync with each other YARR may miss some events, despite them being sent successfully. With external triggers this part is not necessary any more and with that another source of errors is eliminated. Another advantage is that it allows for more control in what pattern triggers are sent and guarantees a fixed trigger rate.

5 Results

With the extension from Chapter 4 it is possible to further analyse the setup as described in Chapter 3. During the test of the system, the difference between the old and new versions of YARR and the FELIX backend will be compared. The first version of YARR will be referred to as YARRRebase and the second as YARRFork. For the FELIX backend the old version is felixcore and felix-star is the new version. During testing the firmware of the FELIX FPGA was upgraded, as such comparisons between the FELIX firmware versions will be made.

5.1 Link Occupancy and Trigger Loss

The measurement for the link occupancy and lost triggers are done on the first frontend of the first link of the KCU116. To see how the data amount affects the system, an external trigger scan is used. For consistency and stability reasons, the high-speed link is reset in between hit changes. The tests are performed by the perform_scan program, which is executed inside loops of a bash script, where between loops the parameters are adjusted.

In this test series, the amount of triggers sent by the board trigger generator is fixed at 512000 triggers. The wait time between triggers is varied. The triggers are generated on the KCU116 and the wait time is the number of empty bunch crossings between two triggers. The time between two bunch crossings is 25 ns. The test is performed a total number of three times.

The tested setups are YARRFork with felixcore and YARRFork with felix-star. YARRRebase can not be compared with the other setups, because it would continue to send triggers during the external trigger scan. This behaviour is unexpected from YARRRebase.

Figure 5.1 displays the number of received triggers against the number of bunch crossings between triggers. Received triggers refer to the number of events which YARR received and decoded. An event is the data the frontend generates when the frontend receives a trigger. In Figure 5.2, the link occupancy against the wait time is shown. There the collected data together with the theoretical values are presented.

The theoretical link occupancy can be calculated and evaluates to

$$\frac{\text{produced data per second}}{\text{data readout speed}} = \frac{\frac{(\text{hits}+1)}{2} \cdot 66\,\text{bit} \cdot (25\,\text{ns} \cdot (\text{BC wait time} + 1))^{-1}}{1.28\,\text{Gbps} \cdot (\frac{\text{nRatio}}{\text{nRatio}+1})}\,. \tag{5.1}$$

The RD53A sends event data in blocks of 32 bit. In the worst case, the number of blocks which are produced equals the number of hits with an additional header block. A frame consists of 66 Bits and can carry two blocks. The timespan over which the data is produced corresponds to the time of a bunch crossing times the number of bunch crossings waited. The readout speed needs to be adjusted as service frames are produced every nRatio frames. This results in only nRatio/(nRatio + 1) of the bandwidth being available for the readout of the data.

5.2 Single Frontend Digital Scan

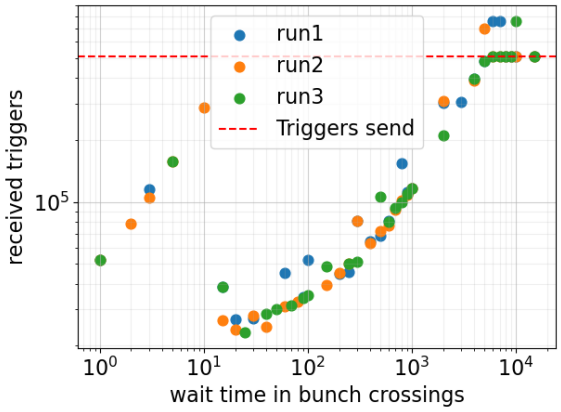
Another test series focused on the difference in performance for the different test setups. This is done by performing digital scans on a single frontend. The selected fronted is the first frontend of the first link of the KCU116. For the type of scan, digital scans are chosen. The high-speed link is reset between the switch of the YARR and FELIX backend versions. The test is carried out by the perform_scan program of the stress test software.

The test is performed 50 times for the new FELIX FPGA firmware and 10 times for the old FELIX FPGA firmware. The number of total lost triggers and the number of triggers which got lost in the downlink can be seen in Figure 5.3. Figure 5.4 shows the amount of false negative and false positive hits of the masurement. The boxes in both graphs represent the minimum and maximum measurement with the average being marked with a black line.

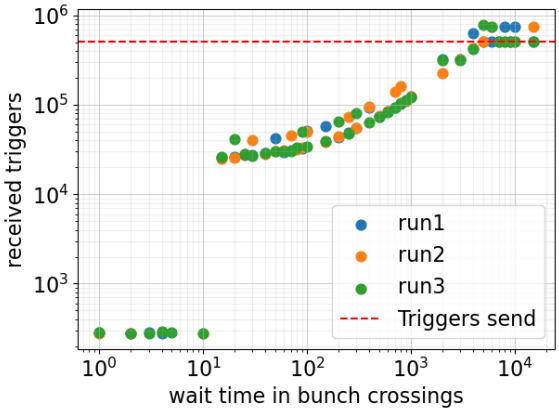
5.3 Scalability

To test how the system scales, the amount of enabled frontends is increased. For this, the test is performed on the first two links of the KCU116. This is followed by the third and fourth link, then all eight links of the KCU116. Afterwards, the first 8 links of the VCU128 are added, and the last test is performed by adding the remaining links of the VCU128.

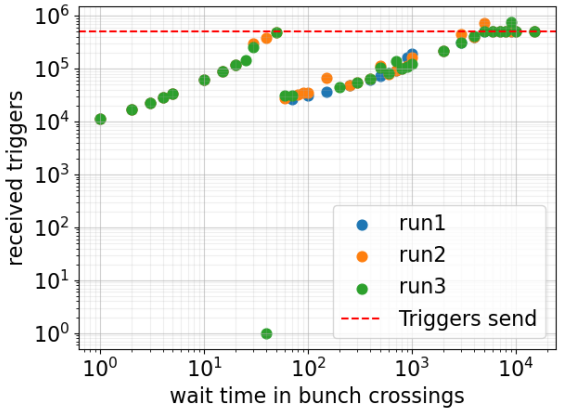
To perform the test the perform_scan program is used with a hit count per event of 10 and the main configuration file is adjusted to include more and more links. The test is performed a total of two times.



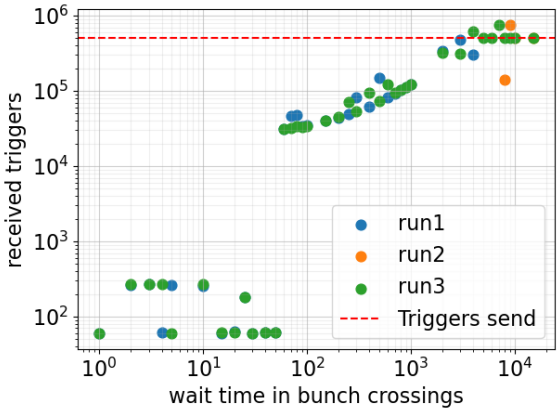
(a) felixcore 10 hits



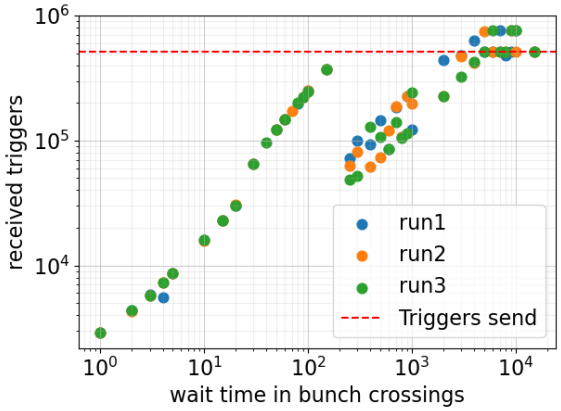
(b) felix-star 10 hits



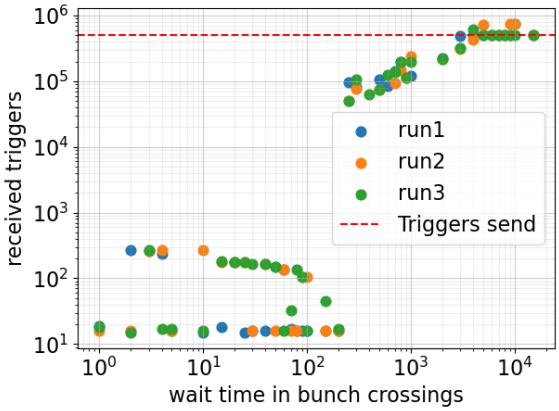
(c) felixcore 50 hits



(d) felix-star 50 hits

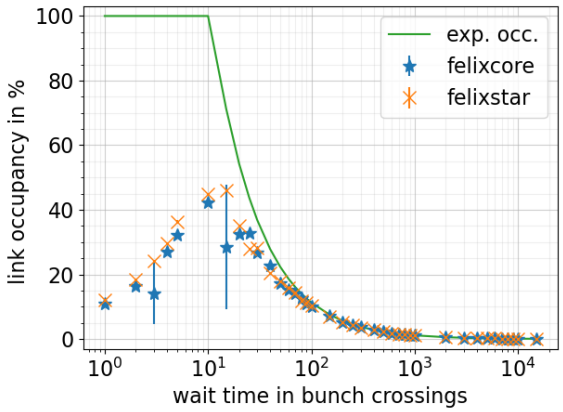


(e) felixcore 200 hits

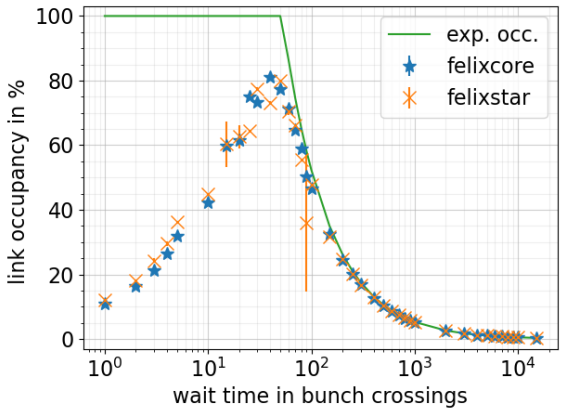


(f) felix-star 200 hits

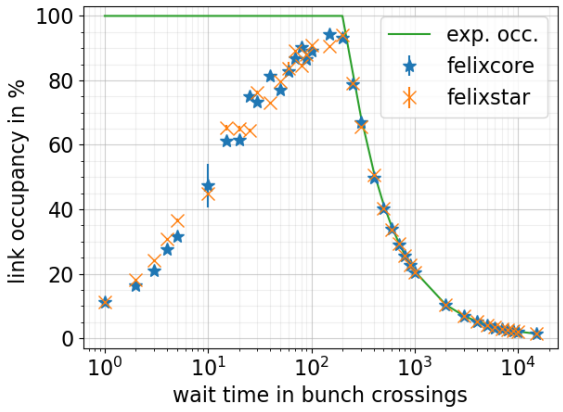
Figure 5.1: Received triggers against the wait time of felixcore and felix-star for N hits. With same coloured dots corresponding to the same run.



(a) Link occupancy against trigger wait time for 10 hits.



(b) Link occupancy against trigger wait time for 50 hits.



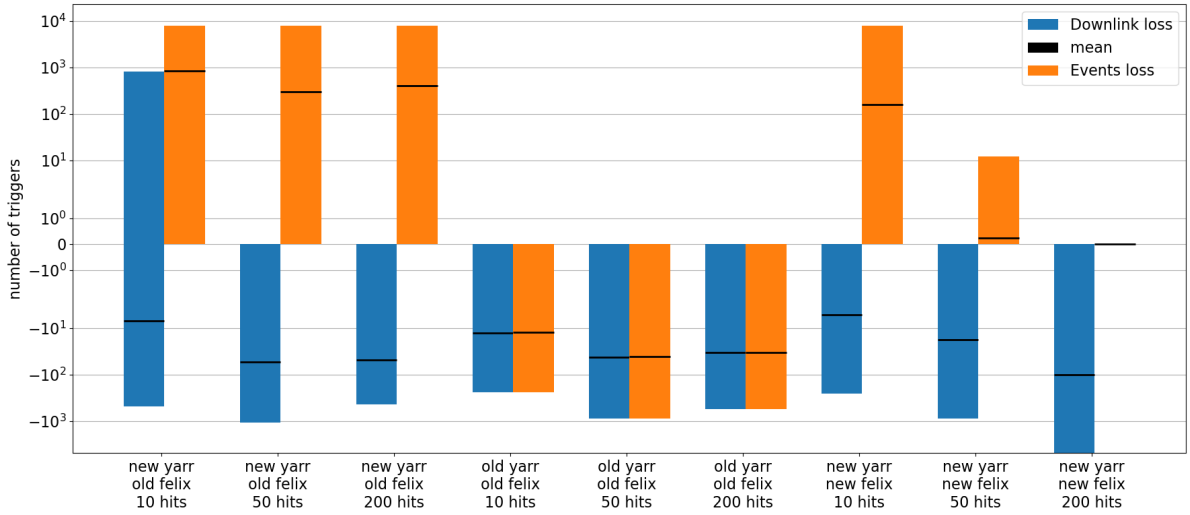
(c) Link occupancy against trigger wait time for 200 hits.

Figure 5.2: Link occupancy against trigger wait time for N hits per event for felix-star, felixcore and the theoretical link occupancy.

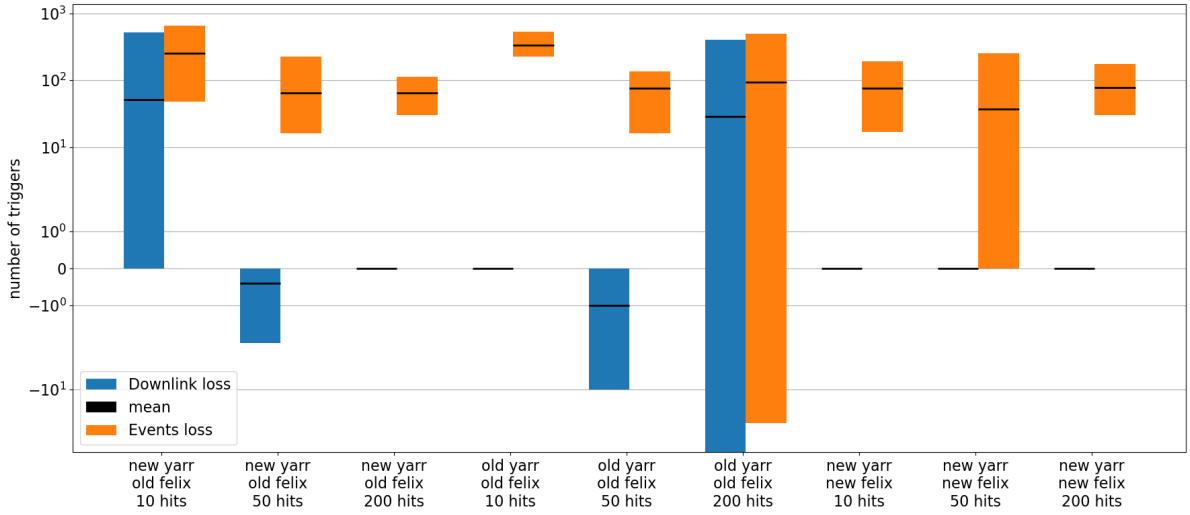
It is observed that independent of which FELIX backend is used when the test is performed for multiple frontends, sometimes the system would crash. And as such, not much data can be collected. The only successful measurement series are the measurements done with felixcore until the 8 links/56 frontends point.

The collected data consists of components of the total runtime of YARR. The runtime of YARR is split into four components. Configuration consists of the time it takes YARR to configure the selected frontends to perform a scan. The scan time relates to the time it takes to perform the scan itself. Analysis refers to the time it takes to analyse the results like producing the occupancy map for the test. And last the processing time is the time it takes to process the final results.

The runtime against the number of links is visualised in Figure 5.5.

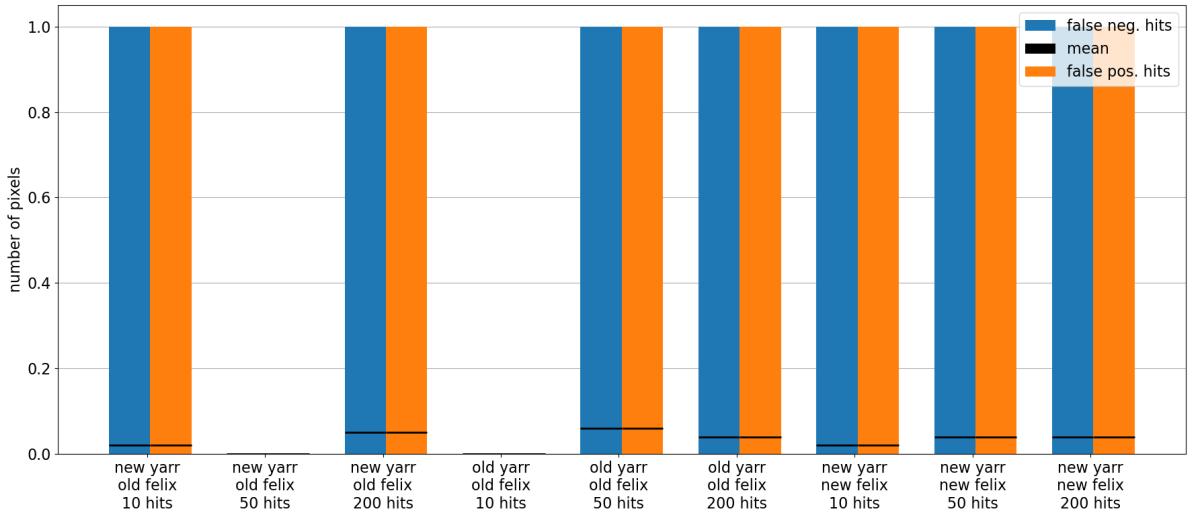


(a) After the FELIX firmware upgrade

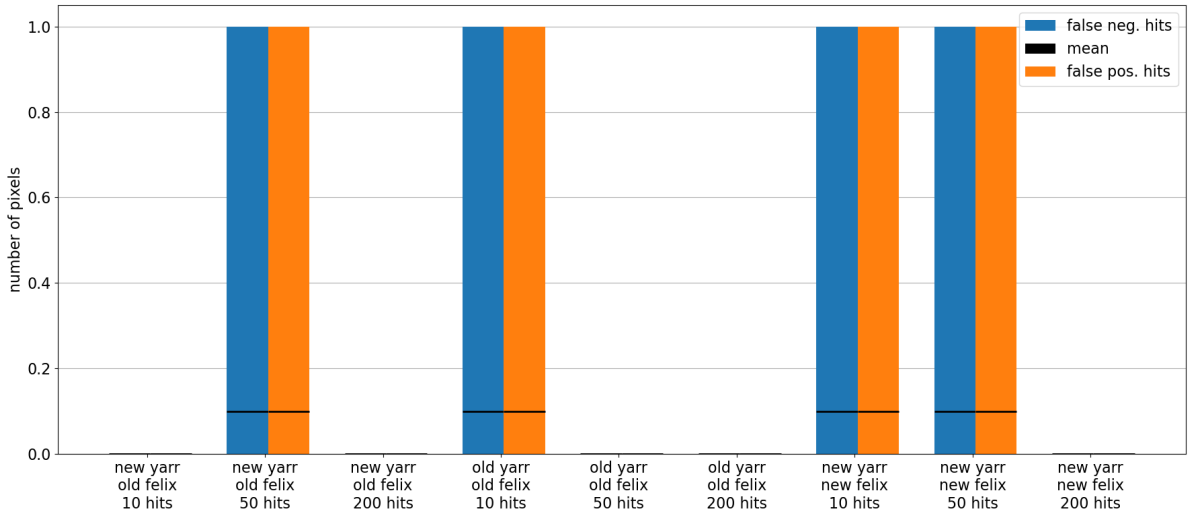


(b) Before the FELIX firmware upgrade

Figure 5.3: Box graph of the total lost triggers in orange and the loss on the downlink in blue. The top and bottom of the boxes correspond to the maximum and minimum values received in the multiple repetitions of the digital scan. The black line shows the average value obtained.

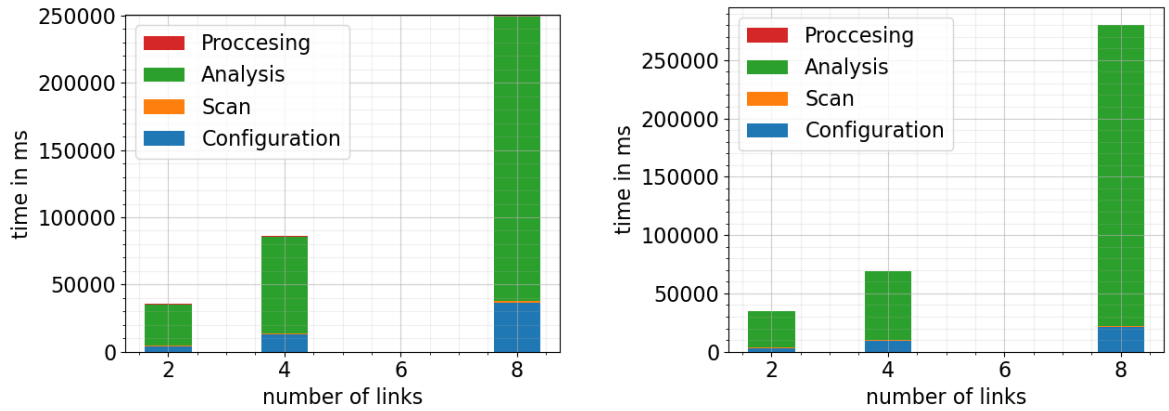


(a) After the FELIX firmware upgrade



(b) Before the FELIX firmware upgrade

Figure 5.4: The amount of misaligned pixels. In blue the number of false negative hits. In orange the number of false positive hits. The black line shows the average value obtained.



(a) Bar graph of the split runtime from YARRFork against the number of links scanned.

(b) Bar graph of the split runtime from YARRRebase against the number of links scanned.

Figure 5.5: YARR runtime split into its components for felixcore and different versions of YARR for the different number of links.

6 Discussion

6.1 Link Occupancy and Trigger Loss

By comparing Figure 5.1 with Figure 5.2 it can be seen that before the wait time reaches the point where the peak link occupancy is, that triggers get lost from YARRs perspective. This is probably because before the peak occupancy is reached that the rate at which the data is generated is too high for the readout system. This is supported by comparing the graphs of the different hit amounts for the same setup in Figure 5.2 and by the theoretical occupancy line. There a clear shift to the right can be detected. This means that for higher hit counts a longer readout time is needed to reach the maximum occupancy. Because only active pixels generate data, the increase of hit pixels results in an increased amount of data needing to be transferred.

Another thing to note is felixcore behaviour prior to reaching peak occupancy. It can be seen that the number of received triggers grows linearly with minimal deviation. But after reaching the peak occupancy a dip in the number of received triggers can be seen. One possibility could be that felixcore receives the events but the data itself could be faulty such as the reported pixel hit location being wrong. This behaviour would need more investigation before more can be discussed about it.

Noticeable is that the occupancy possesses a peak and it does not reach 100% in Figure 5.2. It would be expected, that the occupancy would be at 100% until the readout speed is enough for the data amount like the theoretical line. However, it is observed that the link occupancy builds up until the necessary readout speed is reached. One reason could be that the rate of data production is so high that the readout system can not create proper packages/frames to be read out. This would be a fault in the emulator.

Following the peak occupancy it can be seen that not all triggers are received. With increasing wait time more of the triggers are received. This suggests that not all the data gets read out and with higher wait time less loss occurs. This trend continues until 2000 bunch crossings wait time in between triggers. Starting from that point it seems that the time between triggers is enough that for all of the generated triggers a corresponding event is received.

Another noticeable thing is the jumps between the number of received triggers between the data points. It appears that the points build two separate lines. A possible explanation could be that the data buffers are not flushed properly in between two points of measurement. This means that some events, which were not received in a previous measurement get added in the current one. This is supported by the fact that starting from the point where all triggers are received some measurements receive more triggers than were sent. Another possibility could be that the trigger generator from the board malfunctioned and sent more triggers when ordered, but this is improbable due to the jumps occurring randomly.

Additionally to mention is that some points are missing. This is due to YARR failing to generate an occupancy map and with that the perform_scan program not being able to get further results from the frontend.

6.2 Single Frontend Digital Scan

In Figure 5.3a it can be seen that for the new firmware version, with felix-star, fewer triggers are lost. It seems that the newer versions of YARR and the FELIX backend perform better than the old ones. Another thing to note is the negative lost triggers from YARRRebase. That probably occurs due to the synchronisation error between the YARR transmit and receive parts in a digital scan. This means that the send trigger and receive event stream from YARR are misaligned. Furthermore, the high range of values from YARRFork with felixcore comes from the fact that during the measurements there are a few outliers where no triggers are received, but in general, most triggers are received. An anomaly is the negative amount of downlink loss, which translates to the frontend sending more triggers than YARR instructed. By comparing the means, it seems that this is the reason for the negative lost triggers for YARRRebase. It is to mention, that due to the other setups having lost more or equal than zero triggers, this implies the same number of triggers which appeared in the downlink got lost in the uplink. The most likely explanation is that the frontend register of the RD53A reports the number of send triggers wrong or the registers get read before the scan ended. Something further to mention is the performance for false positive and false negative hits as seen in Figure 5.4a. It can be seen that the amount of false positive and false negative hits is equal. This means that a single pixel gets misaligned. Further due to the low mean. It seems that the occurrence of these misaligned pixels is independent of the setup used.

When looking at the old firmware Figure 5.3b, the system seems to behave more like expected. For most setups, the lost events seem to be pretty consistent. It is to note, that

most of them do not reach zero. This means that for each measurement some triggers are lost. When looking at Figure 5.4, the amount of misaligned hits seems independent of which firmware version was used. The low amount of misaligned pixels for the old firmware is probably due to the dataset being smaller.

6.3 Scaling to Multiple Frontends

In Figure 5.5 it can be seen, that the configuration time seems to grow with the number of links enabled as one would expect. Further, most of the runtime consists of the analysis and configuration time, which grow the fastest. Another thing to note is that it seems that the time grows beyond linearly compared to the number of links, which is unexpected, as configuring and analysing should be independent from one frontend to another.

By comparing Figure 5.5a with Figure 5.5b, it can be seen that for a greater number of frontends YARRFork performs better than YARRRebase.

It is to be noted that due to the low sample size, together with the fact that the test could only be performed up to only 8 links, one should look at these results with caution.

The reason as to why the scaling to multiple frontends failed appears to be the FELIX backend. For the test performed with felixcore, it seems that stability issues occur. As the error occurred following the 8 link mark it looks like the links from the VCU128 are causing the problems. It is to be noted that the VCU128 itself seemed unstable, as while the board was idle, links seemed to disconnect and reconnect in an arbitrary pattern. To mitigate these effects, the high-speed link is reset in between runs. For felix-star, a different issue occurred. After scaling to multiple frontends it seemed that the bus resource allocation failed, which in turn made that scan itself fail. This could be an issue on the software side as well.

7 Conclusion

The implementation of the microservice framework can be seen as successful. It has to be noted, that because not all microservices are yet available the full extent of the system could not be tested. As for the available microservices, all of them could be implemented without much issue. For the future, as more microservices get released they will need to be added and the whole system tested. This also extends to the already implemented microservices. As they are still in development changes are bound to happen, so scripts like the runkey generator need to be maintained in case of a structural change.

The extension of the stress test software is also a success. With the support of the new YARR version, comparing the results with other institutes that have already switched to the new version is now possible. External trigger scans can now be performed. This gives the tester more control and options when performing tests. The extended results give a further overview of the system which allows the tester to pinpoint behaviour and regions of interest.

The test of the different backend and YARR versions was also a success. It can be seen, that felixstar and the new version of YARR outperform the old systems. with the results, one can be confident that moving from felixcore to felix-star benefits the performance of the system for the new firmware. But it is to note that, with the test performed on the system, some weird behaviour was observed which would warrant further investigation. These include, what is the reason that the board generate more triggers when ordered after the firmware update, Why does felixcore fail, when the multiple frontend scan reaches the VCU128, why does felixstar fail to allocate resources for multiple frontends, why do misaligned pixels occur and why does the system receive more triggers than are sent for the external trigger scan.

Bibliography

[1] I. Zurbano Fernandez, et al., *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report* **10/2020** (2020)

[2] L. Evans, P. Bryant, *LHC Machine*, JINST **3** (2008)

[3] ATLAS collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3** (2008)

[4] *ATLAS inner detector: Technical design report. Vol. 1* (1997)

[5] *ATLAS liquid argon calorimeter: Technical design report*, Technical Report CERN-LHCC-96-41 (1996)

[6] *Expected tracking and related performance with the updated ATLAS Inner Tracker layout at the High-Luminosity LHC*, Technical Report ATL-PHYS-PUB-2021-024 (2021)

[7] *Technical Design Report for the ATLAS Inner Tracker Strip Detector*, Technical Report CERN-LHCC-2017-005 (2017)

[8] ATLAS collaboration, *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*, Technical Report CERN-LHCC-2017-021, Cern (2017)

[9] ATLAS collaboration, *ATLAS ITk Pixel Detector Overview*, in *International Workshop on Future Linear Colliders* (2021)

[10] ATLAS ITK, *The Optosystem: validation and testing of the high-speed electro-optical conversion system for the readout of the ATLAS ITk Pixel upgrade*, JINST **19(04)**, C04015 (2024)

[11] T. Heim, *YARR - A PCIe based readout concept for current and future ATLAS Pixel modules*, J. Phys. Conf. Ser. **898(3)** (2017)

[12] S. Binet, B. Couturier, *docker & HEP: Containerization of applications for development, distribution and preservation*, J. Phys. Conf. Ser. **664(2)** (2015)

[13] *DeMi - ATLAS ITk Pixel System Test Microservices*, URL `https://demi.docs.cern.ch/`

[14] *ITk Pixel Stress Test*, URL `https://itk-pixel-stress-test.docs.cern.ch/userdoc/usage_instructions/sw_features/`

Danksagung

I want to thank Arnulf Quadt for giving me the opportunity to write my Bachelor Thesis under him. Also, I want to thank Jörn Große-Knetter, for making himself available to be the second referee and for showing me this interesting area of work in the bachelor börse. Furthermore, I want to thank Ali Skaf, for being available for questions and explaining things to me. And I want to especially thank Matthias Drescher for the numerous explanations, suggestions, ideas and help surrounding this thesis. And lastly, I want to thank my family for supporting me this far.

Erklärung nach der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:
Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 2. September 2024

(Timo Pospiech)