Interfacing Detectors and Collecting Data for Large-Scale Experiments in High Energy Physics using COTS Technology

Jörn Schumacher

PADERBORN UNIVERSITY
The University for the Information Society

Dissertation

Department of Computer Science
Paderborn University

Interfacing Detectors and Collecting Data for Large-Scale Experiments in High Energy Physics using COTS Technology

Jörn Schumacher
Author

Prof. Dr. Christian Plessl
Academic Supervisor

Contents

Abstract

Data-acquisition systems for high-energy physics experiments like the ATLAS experiment at the European particle-physics research institute CERN are used to record experimental physics data and are essential for the effective operation of an experiment. Located in underground facilities with limited space, power, cooling, and exposed to ionizing radiation and strong magnetic fields, data-acquisition systems have unique requirements and are challenging to design and build.

Traditionally, these systems have been composed of custom-designed electronic components to be able to cope with the large data volumes that high-energy physics experiments generate and at the same time meet technological and environmental requirements. Custom-designed electronics is costly to develop, effortful to maintain and typically not very flexible.

This thesis explores an alternative architecture for data-acquisition systems based on commercial off-the-shelf (COTS) components. A COTS-based data distribution device called FELIX that will be integrated in ATLAS is presented. The hardware and software implementation of this device is discussed, with a specific focus on performance, heterogenity of systems and traffic patterns. The COTS-based readout approach is evaluated in the context of the future requirements of the ATLAS experiment.

The main contributions of the thesis are an analysis of the ATLAS data-acquisition system with a focus on the readout system, a software architecture for the main application on FELIX hosts, a performance analysis and tuning based on computer science methods for central FELIX software components with respect to the requirements of the ATLAS experiment, a network communication library with a high-level software interface to utilize high-performance computing network technology for the purpose of data-acquisition systems, and an evaluation and discussion of ATLAS data-acquisition using FELIX systems as a case study for COTS-based data-acquisition in high-energy physics.

Zusammenfassung

Datenerfassungssysteme für Experimente in der Hochenergiephysik wie das ATLAS Experiment am europäischen Forschungsinstitut für Teilchenphysik CERN werden eingesetzt, um experimentalphysikalische Daten aufzuzeichnen und sind essenziell für den effektiven Betrieb eines Experiments. Solche Systeme sind oft in unterirdischen Einrichtungen untergebracht und haben begrenzten Zugang zu Strom, weniger Möglichkeiten zur Kühlung und sind zudem ionisierender Strahlung sowie magnetischen Feldern ausgesetzt. An ihre Entwicklung werden einzigartige Anforderungen gestellt und stellen eine Herausforderung dar.

Um mit den hohen Datenmengen, die ein Hochenergiephysikexperiment erzeugt, umgehen zu können und gleichzeitig den technologischen und umweltbedingten Anforderungen zu genügen bestehen diese Systeme traditionell aus individuell gefertigten elektronischen Komponenten. Solche eigenentwickelten Komponenten sind allerdings teuer zu entwickeln, schwer zu warten und in der Regel nicht sehr flexibel.

Diese Arbeit untersucht eine alternative Architektur für Datenerfassungssysteme die auf kommerziellen Standardkomponenten (commercial off-the-shelf, COTS) basiert. Eine in das ATLAS Experiment zu integrierende COTS-basierte Datenverteilungskomponente "FELIX" wird vorgestellt. Die Hardware- und Software-Implementierung dieses Geräts wird diskutiert, mit einem Schwerpunkt auf Leistungsfähigkeit, Heterogenität der Systeme und Kommunikationsmuster. Der COTS-basierte Ansatz wird im Rahmen der zukünftigen Anforderungen des ATLAS-Experiments bewertet.

Die Hauptbeiträge dieser Dissertation sind eine Analyse des ATLAS Datenerfassungssystems mit Schwerpunkt auf dem Datenauslesesystem, eine Software Architektur für die Hauptsoftwarekomponente von FELIX Systemen, eine Performanzanalyse und Geschwindigkeitsverbesserungen von zentralen FELIX Softwarekomponenten basierend auf Prinzipien und Methoden der Informatik unter Beachtung der Anforderungen des ATLAS Experiments, eine Softwarebibliothek für Netzwerkkommunikation die es erlaubt Netzwerktechnologie aus dem Bereich des Hochleistungsrechnen für die Zwecke von Datenerfassungssystemen zu nutzen, und letztlich eine Evaluation des neuen ATLAS Datenerfassungssystems mit FELIX als Fallstudie für COTS-basierte Datenerfassungssysteme in der Hochenergiephysik.

Acknowledgements

This dissertation is the product of more than three years of research. During this time, many people contributed with advice and support in one way or the other. I would like to express my sincere gratitude to all the individuals who helped me along the way. Without their help this dissertation would not have been possible.

First and foremost, I thank my academic advisor Professor Christian Plessl and my CERN supervisor Dr Wainer Vandelli. During my placement at CERN both of them provided me with guidance and advice as much as freedom to explore new ideas.

A large portion of the research presented in this thesis is based on the ATLAS FELIX project. I thank the FELIX development team for the productive and fun collaboration. It has always been a great joy to work as a member of this team.

I thank Emily, Noel, Sean, Tobias, and Will for their time and effort that they put into proofreading this dissertation, and Volker for his advice on graphics. Their comments were a tremendous help in improving my writing.

Finally, I thank my family, my wife Natalia and our daughter Stella, my parents Anton and Maria-Luise, and my brothers Volker and Tobias, for their endless support during the past years. They have been an enormous source of motivation.

Chapter 1

Introduction

1.1 Motivation

Data acquisition (DAQ) systems for high energy physics experiments are often implemented as complex distributed applications. The DAQ system of the ATLAS experiment [1] at the Large Hadron Collider [2] at CERN in Geneva, Switzerland consists of tens of thousands of applications running on thousands of nodes, in addition to a large amount of custom-designed electronic components. The DAQ system has to interface the ATLAS detector front-end electronics via dedicated custom optical links. The cost of building and maintaining an experiment like ATLAS is high, and thus a DAQ system needs to operate efficiently and record data at a high rate and high quality.

The high data rates in DAQ systems pose a challenge for computing and networking components, so traditionally many custom-designed electronic components, FPGAs, DSPs and so forth have been used to build early components of a DAQ chain close to the detectors. In the past years a trend has emerged in the high energy physics community to push the use of commercial off-the-shelf (COTS) components ever closer to the detectors and reduce the amount of custom electronics. The motivation this approach is the reduction of development time and cost as well as facilitating maintenance and operation of DAQ systems.

The ATLAS experiment is following this trend as well, and in this thesis I present the approach that the experiment is taking. The ATLAS approach is based on FELIX, a new central data distribution system that sits between the ATLAS detectors and data filters and processors and forwards data in both directions. A project like FELIX has to meet the various computational requirements of DAQ systems, e.g., to handle a high data throughput, low or fixed latencies for low-level data communication, high availability and so forth. The thesis shows the specifications of the FELIX system and explores how such a system can be implemented to meet the aforementioned requirements.

1.2 High Energy Physics and Accelerator Experiments

High energy physics (HEP), or particle physics, is the study of elementary particles [3]. Particle colliders are an important tool in HEP. A particle collider works by accelerating two beams of charged particles by the means of electromagnetic fields. The two beams cross in defined interaction points, where particles from the two crossing beams collide at high energies. The outcome of these collisions can be new particles which can be detected and analyzed by instruments in the interaction points. Particle colliders can be used to observe rare physics phenomena or unstable particles that decay quickly.

CERN

CERN (for *Conseil Européen pour la Recherche Nucléaire*, or *European Council for Nuclear Research*) is a European High Energy Physics research facility founded in 1964. The organization is located in the French/Swiss border region near Geneva, Switzerland. Currently CERN has 21 member states and more international collaborators.

The research at CERN concentrates on subatomic particles and includes topics such as testing predictions of the Standard Model of particle physics, supersymmetry, dark matter and others.

CERN is the home of many different experiments in High Energy Physics, many of which are based on various particle colliders. There are several linear accelerators (LINAC2, LINAC3, LINAC4), an antiproton decelerator to study antimatter, LEIR (Low Energy Ion Ring), the Proton Synchrotron and others. Most well known is the Large Hadron Collider (LHC), the world's biggest particle collider.

The Large Hadron Collider

The *Large Hadron Collider (LHC)* is a circular collider with a circumference of 27 km. It was constructed in the tunnel of the Large Electron-Positron Collider (LEP), an older collider that was the predecessor of the LHC. The tunnel is below ground, at an average depth of ca. 100 m. Two particle beams consisting of protons or lead ions are circulated in opposite directions in two vacuum tubes. The beams are bent using strong superconducting magnets. The LHC consists of 1232 dipole magnets, each 15 m in length, and 392 quadrupole magnets, each 5-7 m in length. To achieve superconductivity, the magents are cooled down to -273.1 °C using liquid helium. The beams cross at four points along the LHC circumference. The four LHC experiments ALICE, ATLAS, CMS, and LHCb are the collision points, where beam particles collide at high energies. The experiments use various detector technologies to measure the outcome of these collisions.

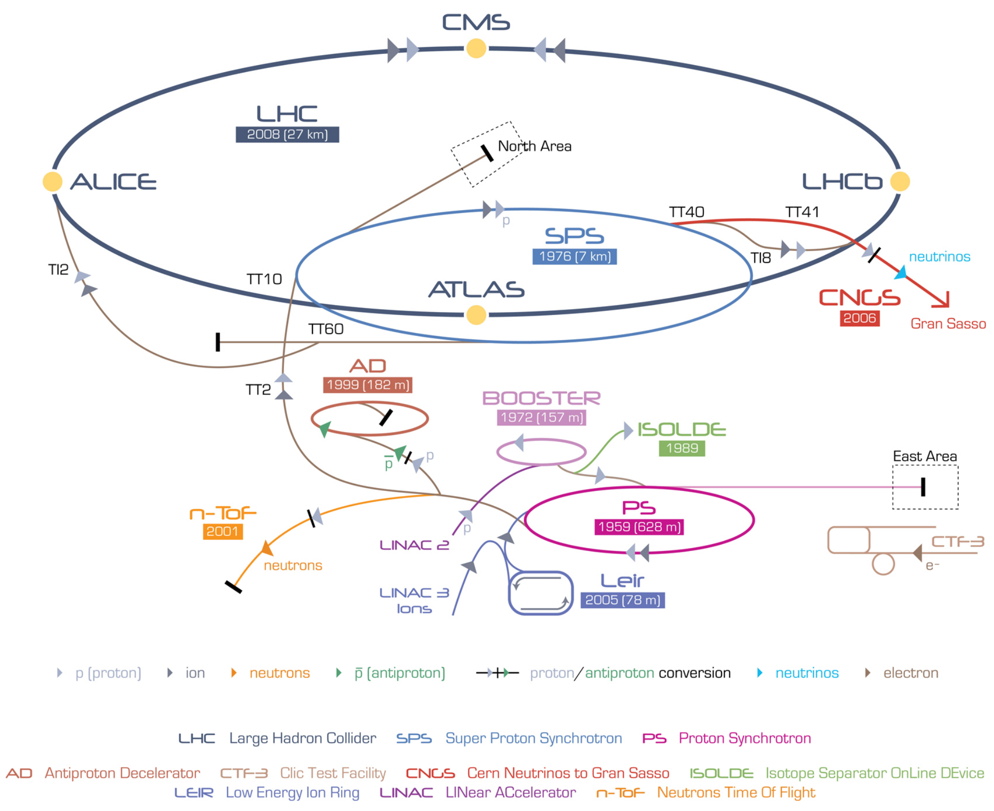


Figure 1.1: The accelerator complex at CERN. Particle beams pass through several accelerators before entering the LHC ring where they are accelerated to their final velocity. The LHC experiments ALICE, ATLAS, CMS, and LHCb are situated at four collision points along the circumference of the LHC. Image source: CERN.

Particles are accelerated to velocities close to the speed of light in several smaller accelerators before entering the LHC ring, see Figure 1.1. In the final beam, particles circulate the vacuum tubes in *bunches* of 1.15×10^{11} particles. An interaction of bunches in the counter-rotating beams in the collision points is called a *bunch crossing*. An important metric is the *bunch crossing rate (BCR)*, the amount of bunch crossings in each collision point per unit of time. The LHC operates at a bunch crossing rate of 40 MHz. Although the number of particles in a bunch is high, only around 20 particles will collide in a bunch crossing at the nominal mode of operation as of 2016.

The LHC and the four experiments are upgraded at irregular intervals, typically around every 3 years. The first upgrade took place in the so-called Long Shutdown 1 from 2013 to 2015; the next upgrade phase (Long Shutdown 2) is a planned 18-month period beginning in 2018. The upgrade projects aim to improve performance and reliability.

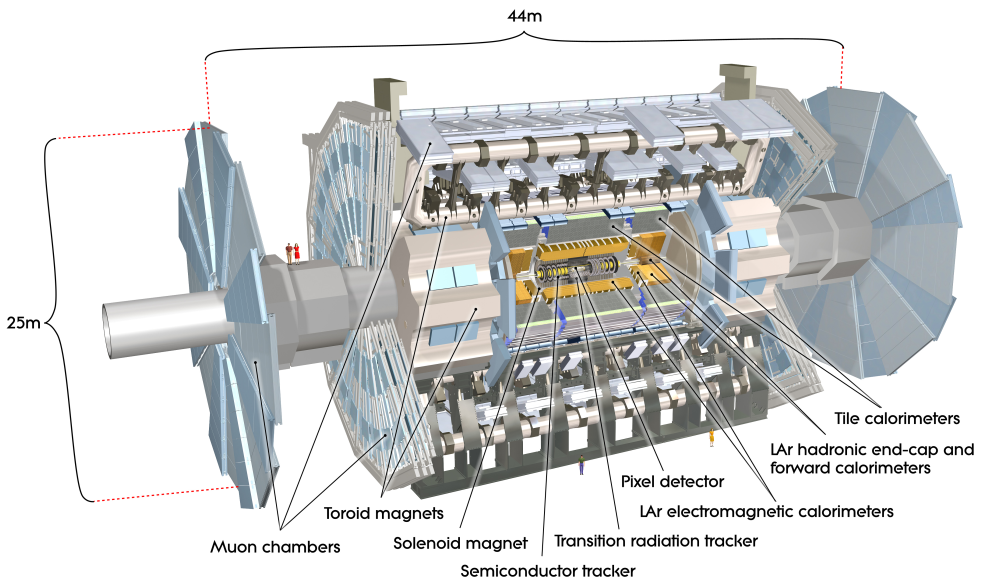

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 1.2: The ATLAS experiment. The two magnet systems are used to bend the trajectories of charged particles. The other systems are detectors that measure various properties of passing particles. Image source: ATLAS experiment.

The ATLAS Experiment

ATLAS [1] is, together with CMS, one of the two general-purpose particle physics experiments at the LHC. ATLAS is an international collaboration of more than 5000 scientists from institutions all around the world. As a general-purpose experiment, ATLAS has the goal to explore a wide range of particle physics phenomena. Specific goals include the test of predictions of the Standard Model of particle physics, the exploration of matter/antimatter asymmetry, or the exploration theories beyond the Standard Model.

The ATLAS experiment has a cylindrical design, see Figure 1.2. It is 44 m long, 25 m in diameter and weighs about 7000 t. The experiment is situated in a cavern approximately 100 m below ground. Next to the experiment cavern is a service cavern used for cooling systems, detector control and readout or maintenance purposes. An above-surface datacenter processes data generated by the experiment. A schematic of the experiment layout is shown in Figure 1.3. There are many environmental factors posing a challenge for data taking. Electronics in the detector and service cavern are exposed to radiation and magnetic fields and space is limited.

Different detectors measure properties of the particles that are created by the collision in the center of ATLAS. The properties that are measured are

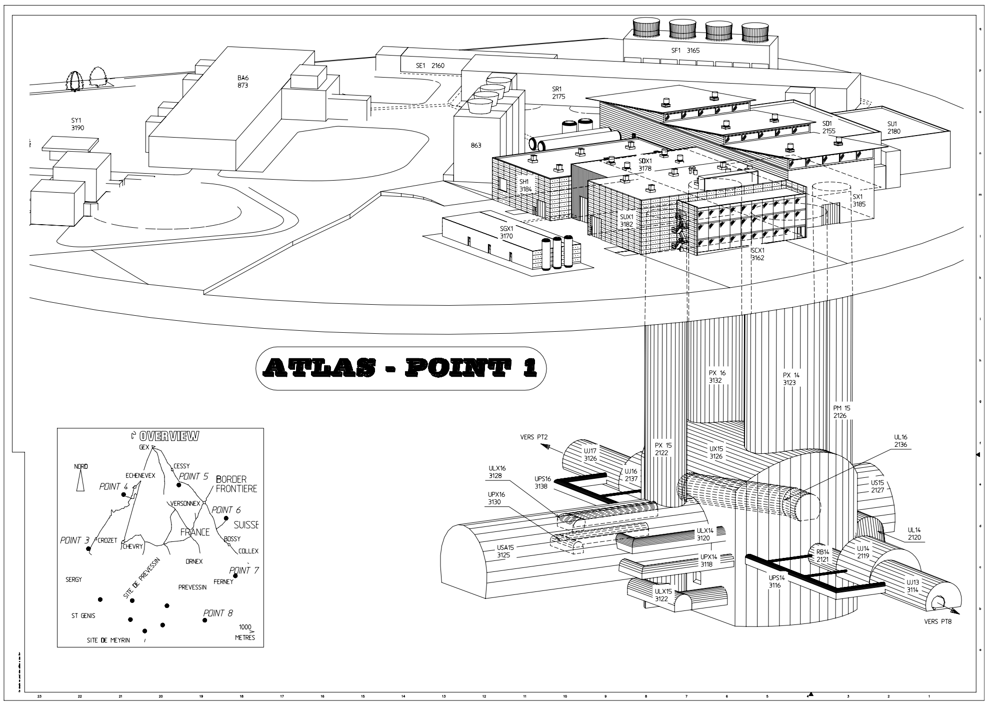


Figure 1.3: The ATLAS experiment cavern is situated approximately 100 m below ground level. A neighboring service cavern (USA15) contains detector readout components and parts of the ATLAS data-acquisition system, but also systems for cooling and machine control. The rest of the data acquisition system is situated in a datacenter above surface (SDX). Image source: [4].

the particle's trajectory, momentum, and total energy. A magnet system generates two overlapping magnetic fields that bend the trajectory of charged particles. Energy is measured by different Calorimeter systems. Charge and mass of the particles can be reconstructed from the other properties. Muons, which barely interact with matter, are detected by a special Muon spectrometer.

1.3 Trigger and Data Acquisition Systems

An experiment like ATLAS generates a large amount of data. In each bunch crossing, particles from the counter-rotating beams interact and might produce new particles or radiation as outcome of the collisions. The sum of all interactions in a bunch crossing and their outcomes is referred to as an *event*.

Definition 1.1 (Event). *The sum of all outcomes of all interactions between particles in a bunch crossing as measured and recorded by detector electronics is called an* event.

In ATLAS, an event typically contains an average of 1.5–2 MB of data from all detectors. At a bunch crossing rate of 40 MHz the ATLAS experiment therefore generates more than 60 TB of data each second, a data rate that is challenging to record with modern storage systems. The vast majority of events represent well-known physical processes, so by filtering only interesting events the data rate can be reduced significantly. Such an event filtering system is called *trigger*.

Definition 1.2 (Trigger)**.** *A system that receives a stream of events as input and generates an ACCEPT or REJECT decision for each event is called a* trigger*.*

Triggers can be implemented in hardware, software, or a mixture of both. Multiple levels of triggers can be used with different implementations or strategies. A trigger can accept or reject an event based on the number of particles, type of particles, energy or momentum of particles, or other characteristics of an event.

The system that receives data from the detector sources and records an event stream on permanent storage is called *data acquisition system*.

Definition 1.3 (Data Acquisition System)**.** *A* data acquisition system (DAQ) *collects event data from detectors, receives trigger decisions from a trigger, and records accepted events on permanent storage.*

The systems in the detectors that interface with the DAQ are called *detector front-ends*. The main task of a DAQ system is to collect and relate data from different sources to events and finally write the event stream to a permanent storage. Often DAQ system also include entities to monitor the quality of measured data, data compression, or other relevant tasks.

Computer Engineering Challenges of DAQ systems

DAQ systems need to meet the requirements of the respective experiment. These can include, among others, the following:

High Data Rates In order to maximize the efficiency of the experiment and the number of recorded events, events are generated at a high rate. The front-end electronics, trigger, and DAQ system need to be able to handle an event stream at a high rate. The trigger is used to reduce the output event rate to a managable value.

High Availability Due to the cost of operation and maintenance, many experiments are run for long periods of time without a pause in data-taking. Stops in data-taking due to faulty components in a DAQ system can be costly, and maintenance can be complicated due to environmental factors like high radiation in the vicinity of the experiment. DAQ systems have to be designed for high availability.

High Throughput High resolution readout of experiments at a large event rate requires the DAQ system to handle a large data throughput. This can pose a challenge to communication links and interconnects that require a high bandwidth.

Low Latency or Fixed Latency A new event in the ATLAS experiment is generated every 25 ns. Low-level readout electronics in a DAQ system need to operate at a low or even fixed latency in order to process information at this rate. Cable lengths and processing times of components need to considered, and part of the components will typically need to have real-time capabilities.

Heterogenity of Systems, Workloads and Requirements A DAQ system involves many different components with different purposes and characteristics. Some detectors might generate more data than others so that work is unevenly distributed. A calibration run will have different requirements than a data-taking run. Some systems like detector control are critical for operation while other systems are redundant or can be deactivated temporarily.

Radiation and Magnetic Fields The environment has a big impact on the design of a DAQ system. Experiments like ATLAS produce radiation and strong magnetic fields and are cooled down to extreme temperatures. Electronics placed directly in the experimental cavern need to be designed to withstand these conditions.

Underground Access The placement below ground level complicates maintenance and restricts space, cooling, and power. This is challenging for any substantial installation of computing hardware.

Long Distances In ATLAS, cables connecting the service cavern that contains most of the readout components and the surface-level datacenter have to be about 150 m long. This distance is longer than typical distances in datacenters. Links that connect the various computing elements need to be able to cover such long distances.

Storage The filtered event stream has to be recorded and archived to a permanent storage for later analysis. The storage subsystem has to support a write speed to handle the incoming event stream and provide large enough space to store experiment data.

1.4 Contribution

Traditional DAQ systems are often built using many custom electronic components. The development and production of custom electronics requires careful planning and takes great cost and effort, but the challenges like high data-rates and throughput and latency requirements could render custom electronics the only viable option for the early stages of a DAQ system.

The unique challenges of DAQ systems for high energy physics experiments and the workloads, data access patterns, and performance requirements that differ in many aspects from classical high-performance computing scenarios, present an interesting area of study for computer science. In this thesis I explore the use of COTS components and computer science methods and principles for DAQ systems. Specifically, I contribute the following developments:

- An analysis of the current ATLAS DAQ system and outline points where a new DAQ system based on COTS components can improve data taking.
- As part of the ATLAS/TDAQ FELIX developer team I introduce a new DAQ system for the ATLAS experiment based on the FELIX project that maximizes the use of PC components and software over custom electronics. The new system in the DAQ chain based on computer engineering principles enables a more scalable, fault-tolerant and uniform system design. My personal contribution to the project is the development of many software components of the FELIX project.
- I show computer science methods and techniques to develop software components that meet the performance requirements of a DAQ system like ATLAS.
- I present NetIO, a general purpose network communication service that provides users with implementations for high-level communication patterns on top of high-performance fabrics. NetIO supports different HPC interconnects like Infiniband natively. It is used in FELIX and tuned for the foreseen workloads of the project, but the library is designed to be of general purpose and can be used outside the scope of ATLAS.
- Using a prototype of the FELIX system I evaluate and discuss the future ATLAS DAQ system with respect to the computational challenges presented in Section 1.3.

The FELIX project is a joint development effort that involves tens of developers from various institutes. I contributed the initial software stack for the FELIX project. Among other things I implemented the low-level tools to operate the FELIX PCIe I/O card, the main FELIX data processing application, code for processing data packets at a high rate with minimal latency and the FELIX network I/O stack (the NetIO library). Furthermore I contributed to the hardware selection of the FELIX PC platform. The FELIX project also delivers a custom PCIe board to connect to the ATLAS detector electronics and firmware for the FPGA on this board that manages low-level communication with detectors and the host PC. I did not participate in the development of the PCIe card and its firmware.

I published several results of my research as scientific papers and also contributed as co-author to other papers in the context of the ATLAS DAQ system:

- Jörn Schumacher et al., *"FELIX: a High-Throughput Network Approach for Interfacing to Front End Electronics for ATLAS Upgrades"* [Schumacher et al., 2015a]. Presented in April 2015 at the 21st International Conference for Computing in High-Energy Physics (CHEP 2015) in Okinawa, Japan.

- Jörn Schumacher et al., *"Improving packet processing performance in the ATLAS FELIX project"* [Schumacher et al., 2015b]. Presented in June 2015 at the 9th ACM International Conference on Distributed Event-Based Systems (DEBS 2015) in Oslo, Norway.

- Jörn Schumacher et al., *"High-Throughput Network Communication with NetIO"* [Schumacher et al., 2016]. Presented in October 2016 at the 22nd International Conference for Computing in High-Energy Physics (CHEP 2016) in San Francisco, USA.

- Andrea Borga et al., *"Evolution of the ReadOut System of the ATLAS experiment"* [Borga et al., 2014]. Presented in June 2014 at the 3rd International Conference on Technology and Instrumentation in Particle Physics (TIPP 2014) in Amsterdam, The Netherlands.

- Andrea Borga et al., *"A new approach to front-end electronics interfacing in the ATLAS experiment"* [Borga et al., 2016]. Presented in September 2015 at the Topical Workshop for Electronics in Particle Physics (TWEPP 2015) in Lisbon, Portugal.

- Julia Narevicius et al., *"FELIX: The New Approach for Interfacing to Front-end Electronics for the ATLAS Experiment"* [Narevicius et al., 2016]. Presented in June 2016 at the 20th Real Time Conference (RT 2016) in Lisbon, Portugal.

- Kai Chen et al., *"FELIX: a PCIe based high-throughput approach for interfacing front-end and trigger electronics in the ATLAS Upgrade framework"* [Chen et al., 2016]. Presented in September 2016 at the Topical Workshop for Electronics in Particle Physics (TWEPP 2015) in Karlsruhe, Germany.

1.5 Outline

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the evolution of the ATLAS DAQ system and introduces a new DAQ architecture based on the FELIX project for the upcoming ATLAS upgrades. The implementation of the FELIX system and its hardware, firmware and software components is discussed in Chapter 3. Chapter 4 describes how data in FELIX are encoded and transmitted over the PCIe bus and how the data can be efficiently decoded software. In Chapter 5 the network communication service NetIO is introduced and its implementation is discussed. Comparative performance measurements are given. Chapter 6 presents system benchmarks of a FELIX prototype in realistic scenarios. Concluding remarks are given in Chapter 7.

Chapter 2

From Custom to COTS Components: Evolution of the ATLAS Data-Acquisition System

2.1 LHC Upgrade Program

The LHC and its experiments consist of many different subsystems that are constantly maintained and upgraded. The upgrades have the aim of improving the LHC performance, for example collision energy and intensity. As a consequence, experiments have to be upgraded to cope with the new conditions. This includes not only the detectors themselves, but also the data acquisition systems, which are presented with an increased and more challenging workload in terms both of event processing rate and complexity. Upgrades are therefore computer engineering challenges with new requirements for computing, networking and storage. The upgrade periods also present an opportunity for redesigning DAQ architectures or deploying new components, and can be seen as a convenient break from data taking for research and development of computing and processing infrastructures.

The LHC is planned to be operated for decades and thus the upgrades are planned a long time in advance. 3- to 4-year run phases are typically followed by 1- to 2-year shutdown phases during which systems can be upgraded or replaced.

The LHC schedule is shown in Figure 2.1. LHC Run 2 is scheduled to continue until the end of 2018, followed by a second long shutdown phase (LS2) of two years. In 2021 the LHC is scheduled to restart and run continuously until 2023. The period from 2015 to 2023 (Run 2 and 3) are referred to as LHC Phase I. With these upgrades the LHC is planned to exceed its peak design luminosity, an operational parameter indicating the intensity of

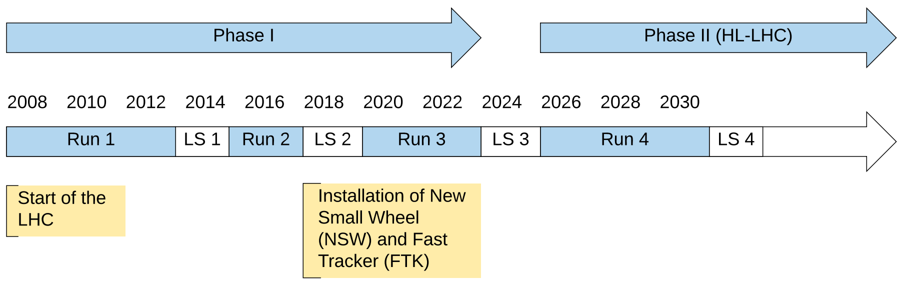


Figure 2.1: A timeline for operation and maintenance periods of the LHC. The beginning of Phase II marks a major upgrade to the collider.

the collisions, in Run 3. ATLAS will be upgraded during the long shutdown phases. Major upgrades in LS2 are the installation of a new muon detector, the New Small Wheel (NSW) [5], and a new low-level trigger component, the Fast Tracker (FTK) [6].

Phase I is followed by a third long shutdown phase (LS3) which marks the beginning of Phase II. The LHC will be subject of a major overhaul with the goal of increasing its design luminosity by a factor of 10. LHC Run 4 is scheduled from 2026 to 2030, followed by a one-year long shutdown phase (LS4) in 2031. Run 5 is planned to start in 2032.

2.2 LHC Run 1 (2009 – 2013)

In this section I present an overview of the ATLAS trigger and data-acquisition system during the first LHC run from 2009 to 2013. The descriptions are based on the publication [7]. The ATLAS trigger and data-acquisition system consisted and still consists of many individual hardware and software components which are operated full-time during data taking. A schematic overview of the system is depicted in Figure 2.2. The peak collision energy reached during run 1 was 8 TeV.

The overall architecture of the system invovles three layers of triggers: the L1 trigger, the L2 trigger and the event filter. While the L1 trigger is built purely from custom-desigend electronic components, the L2 trigger and event filter are based on server PCs. Data fragments of events accepted by the L1 trigger are buffered in a layer of server PCs called the Read-Out System (ROS) until the L2 trigger and event filter decisions are made. Finally, accepted events are sent to a set of storage servers where they are recorded on disk. The individual components will be discussed in more detail in the sections below.

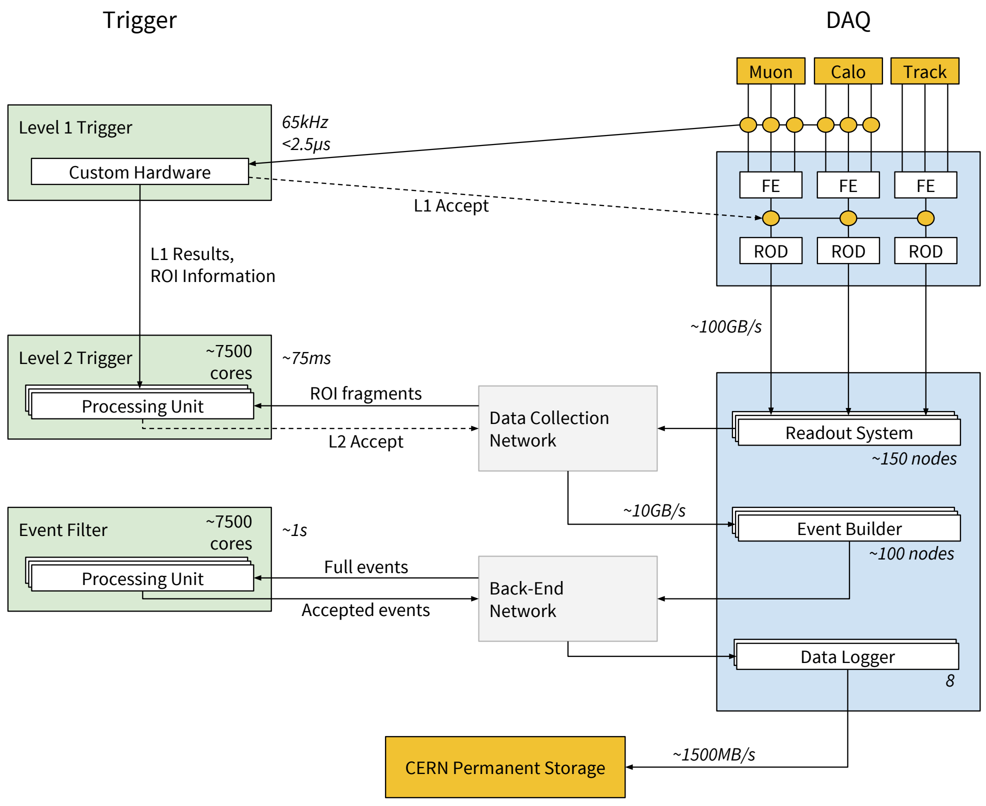


Figure 2.2: The ATLAS TDAQ system during LHC Run 1. The trigger part is on the left, data acquistion on the right.

Detector Readout

The first layer of the ATLAS data acquisition system consisted of the Read-Out Drivers (RODs). These are detector-specific systems receiving data from the various ATLAS detectors. They performed data manipulation tasks like compression, reformatting as well as data aggregation. The ATLAS front-end electronics and RODs communicated over custom, non-uniform and detector-specific optical links. The RODs were typically implemented as rack-mounted VMEbus [8] modules. VMEbus is a widespread bus used in high energy physics.

The RODs also received ACCEPT signals for accepted events from the first ATLAS trigger level via the TTC system (see below for descriptions of the trigger and TTC). Accepted events were forwarded to the 151 ROS PCs. The ROS system buffered event fragments and forwarded them on request to the higher-level data processors. RODs and ROS PCs were connected via optical point-to-point links running the S-Link [10] protocol, the so-called ReadOut Links (ROL). Each ROL was operated at a data rate of either 160 or 200 MB/s.

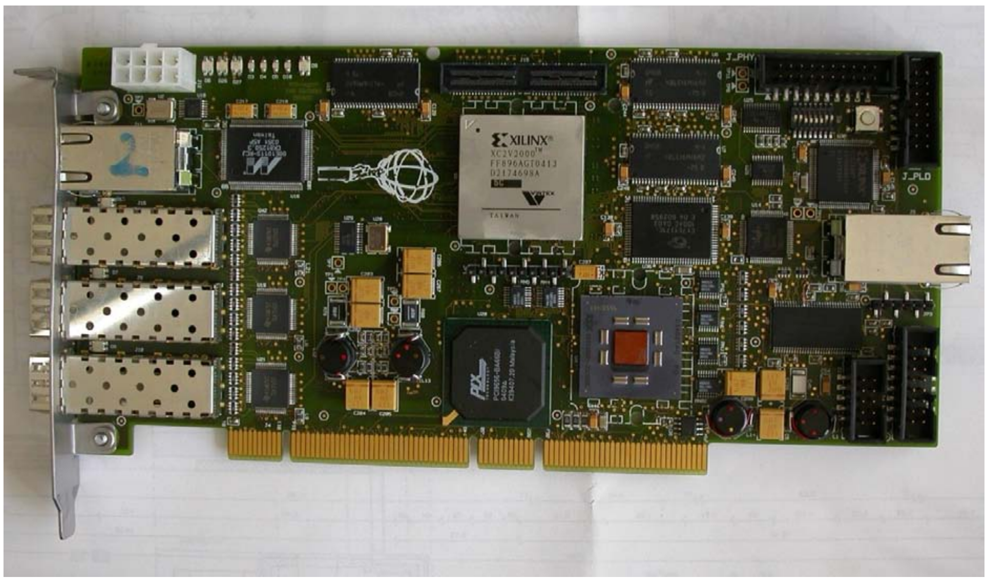


Figure 2.3: The ROBIN card. Image source: [9]

The ROS PCs were equipped with custom PCI cards called Read-Out Buffer Input (ROBIN) [11] to interface with the ROLs (Figure 2.3). Each ROBIN connected to up to three ROLs. The majority of the ROS PCs housed four ROBIN cards and thus connected up to 12 ROLs. An onboard chip for simple data processing tasks was also mounted on the ROBIN. The ROS PCs were mapped to the detectors and subdetectors of the ATLAS experiment. A ROS PC for example might have received event fragments from a few neighboring cells of the Liquid Argon calorimeter. The mapping was organized in a way that particles traversing through neighboring cells in a detector would lead to event fragments being sent to the same ROS. This minimized the number of ROS PCs storing information and therefore the number of messages being sent in the DAQ system.

TTC

The Timing, Trigger and Control (TTC) [12] system fulfilled multiple functions. First, the system delivered an accurate clock signal synchronous to the LHC bunch crossing rate to the readout electronics of the ATLAS DAQ system (timing). Second, the system delivered the ACCEPT information of the L1 trigger to the RODs, which forwarded accepted events to the ROS system (trigger). Third, control information, like a BUSY signal that temporarily pauses the trigger, was forwarded to all trigger components using the TTC system (control).

The TTC system was implemented with optical fibres, custom electronics, and a simple protocol where small data packets were transmitted at a 40 MHz clock rate. The TTC system had a required timing accurary in the order of nanoseconds.

Event Building and Data Flow

An event building system collected the event fragments from the ROS PCs and assembled the full event data structure. The application that built the full events is called SubFarm Input (SFI). The DataFlow Manager (DFM) application orchestrated the assignments to the SFIs and acted as a load balancer. The DFM also communicated *clear* commands from the High Level Trigger (see below) to the ROS system, which in turn could free old event data fragments.

Trigger

Three trigger levels were used for ATLAS in LHC Run 1. The *Level-1 trigger (L1)* was a low-level trigger that had events at the bunch crossing rate of 20 MHz as input. During operation the accept rate could reach 65 kHz (but the DAQ system was designed for an L1 accept rate of up to 75 kHz). Due to limited buffering capabilities in the front-end electronics a trigger decision had to be made within a few microseconds to decide whether the acquired event needs to be preserved or discarded. The ATLAS L1 trigger had a maximum latency of 2.5 μs, including the delay introduced by signal transit times in cables. The system was implemented in dedicated electronic components that were situated in the service cavern close to the ATLAS experiment to minimize latency. Trigger decisions were made using information from the calorimeters and muon detectors. Track reconstruction was not possible at this stage due to the strong latency requirements. Upon an L1 accept signal data were pushed from the front-ends to the ROS systems and were buffered there.

A second low-level system, the *Region-of-Interest-Builder (RoIB)*, computed region-of-interest information, i.e., location information about interesting features of detected signals. Together with the L1 accept signal this information was distributed to the *Level-2 trigger (L2)*.

L2 trigger decisions were based on partial event data. The L2 trigger received the information from the RoIB and requested data from the ROS PCs corresponding to the geographical information in the region-of-interest. The L2 trigger was composed of three different types of nodes: L2 supervisor nodes (L2SV), L2 processing units (L2PU) and L2 result handlers (L2RH). The L2 supervisor nodes were equipped with PCI cards called FILAR [13] to receive the RoIB data. Events were assigned by the L2SV nodes to one of the L2PU nodes, which ran algorithms to decide on the acceptance of the event. The L2 trigger in LHC Run 1 was operated at a maximum accept rate of circa

6 kHz. The L2 decision was sent to the L2SV nodes. The L2 trigger consisted of 768 servers in the end of 2011. Events that were accepted by the L2 trigger were then built and passed to the *event filter (EF)*.

The event filter was the last trigger level in ATLAS. Its algorithms were based on full event data. The accept rate of the event filter was up to 300 Hz. The event filter consisted of two types of applications. The event filter data-flow components (EFD) buffered the full event data. The event filter processing units (EFPU) ran the actual trigger algorithms on the event data. Each event filter node ran one EFD and multiple EFPUs. EFD and EFPUs communicated via shared memory. In total the event filter consisted of 630 servers in 2011 by the end of LHC Run 1.

Networks

In the ATLAS TDAQ system there were three separate networks: a control network, a data collection network, and a back-end network. The main purpose of the control network was run control and monitoring while the data collection and back-end networks were used for high-volume event data traffic. The networks are shown in Figure 2.4.

The control network was, for redundancy reasons, backed by two core routers. Each server was connected to both routers via 1 GbE links, either directly or via concentrator switches. All servers in the ATLAS DAQ system were connected to the control network.

The data collection network connected the ROS PCs with the L2 trigger and event builder. It had a similar topology as the control network and was backed by two core routers. The aggregated bandwidth was however much higher. ROS PCs and L2 trigger nodes were connected to each core router via aggregator switches. Event builder nodes and L2SV nodes were connected to each core router via direct 1 GbE links. The two core routers were interconnected via four 10 GbE links.

The back-end network was also built using two core routers to distribute data between the event builder, event filter and storage nodes. Event builder nodes were connected using 1 GbE links to each router. Storage servers and event filter processing nodes used aggration switches and 1 GbE links to each core router.

Storage

Accepted events were written to hard disk drives in a data logging farm consisting of six nodes. The application that received data from the event filter and wrote it to disk was called SubFarm Output (SFO). One SFO was running per data logging node. The event streams were only temporarily stored in the SFO nodes. A script moved the recorded data to CASTOR [14], a centralized

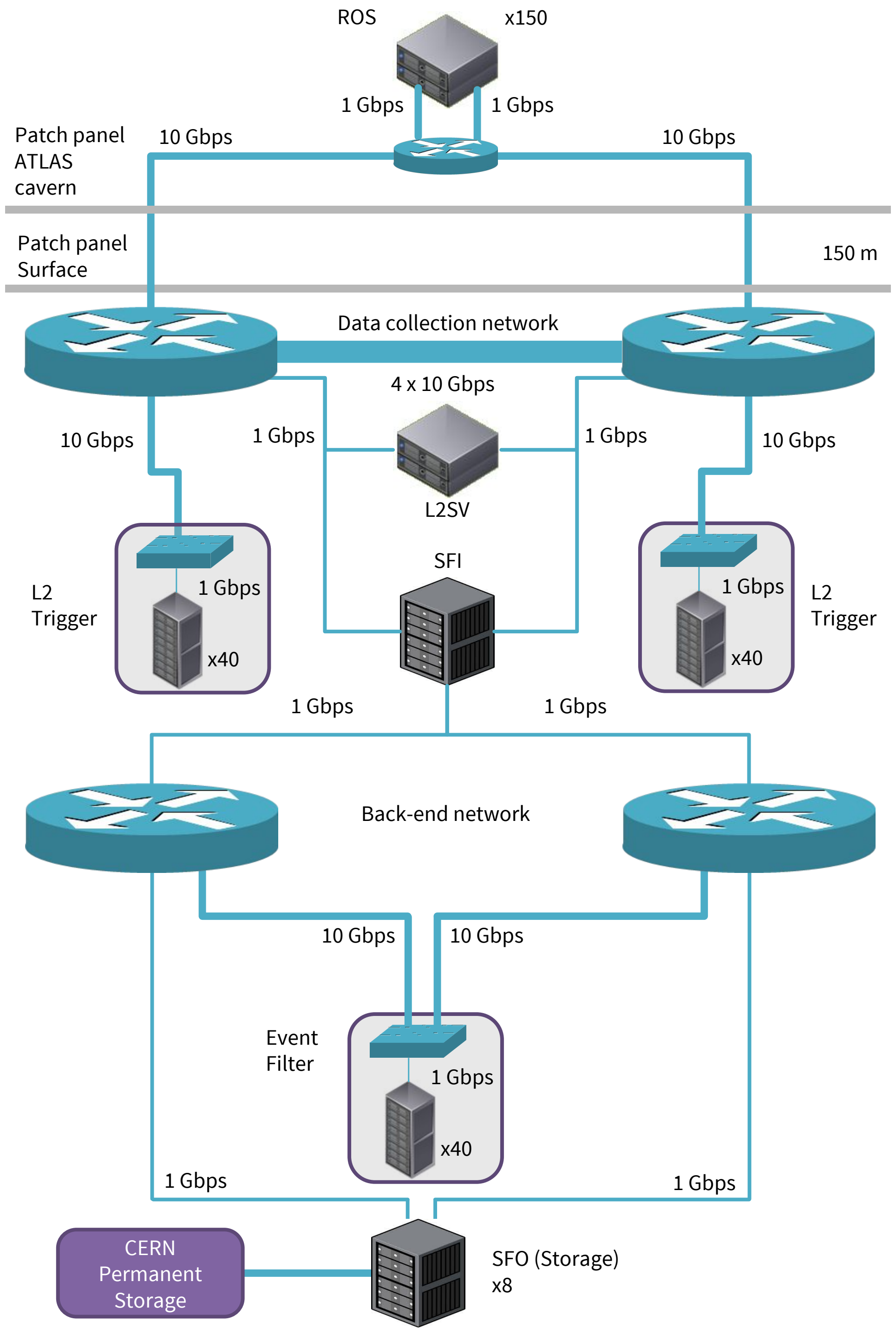


Figure 2.4: The ATLAS DAQ networks in Run 1. Not shown is the control network that each server is connected to.

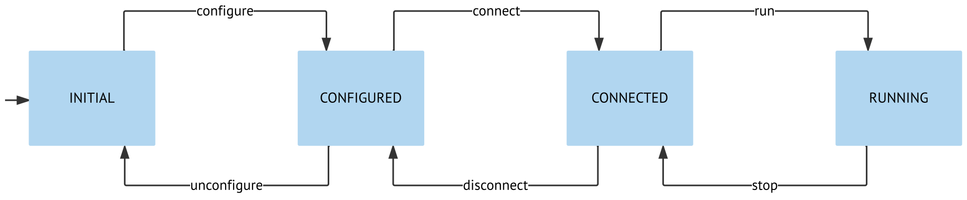


Figure 2.5: The run control state machine.

storage facility at CERN that was designed for archiving large amounts of data.

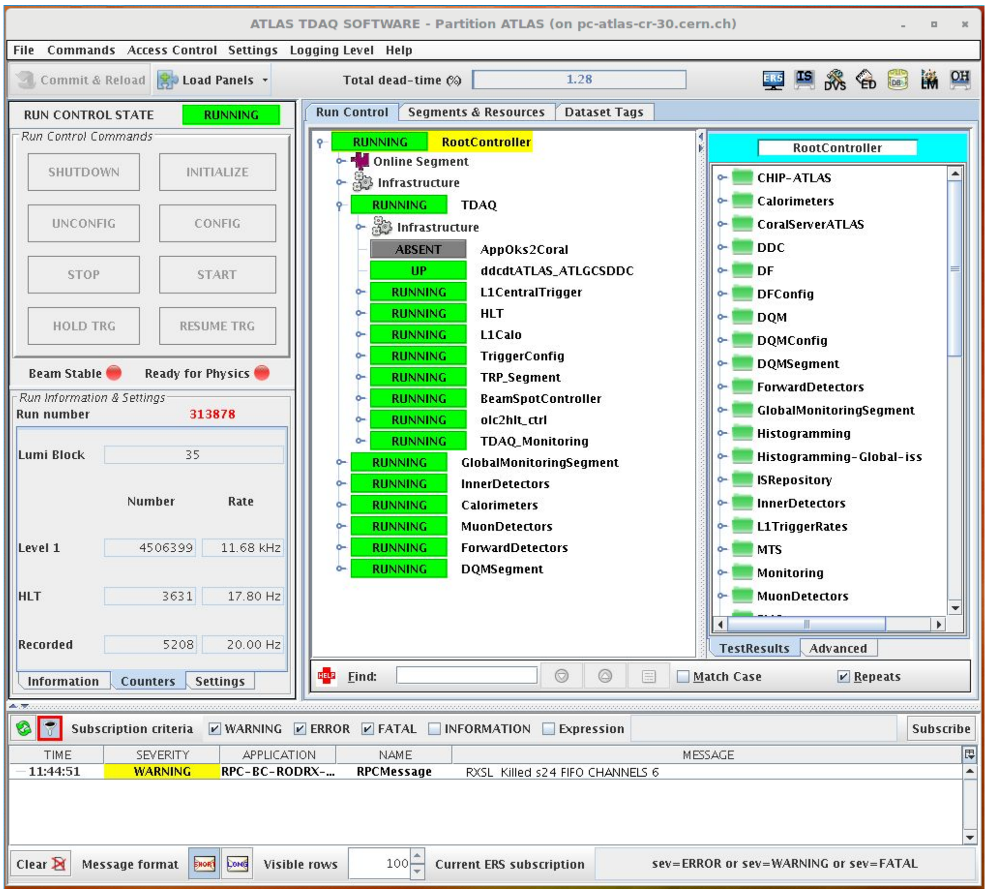
Run Control

The complete ATLAS online computing system including data acquisition, trigger, detector control, monitoring, calibration etc. consisted of many applications operating on thousands of nodes. Together they formed a large distributed computing system that needed to be operated, configured and monitored. To manage the state of each single component, each application implemented a state machine interface. Each application could be in one of the states INITIAL, CONFIGURED, CONNECTED, or RUNNING (plus several sub-states), see Figure 2.5. Similar applications were grouped together, and each group was assigned to a controller application, which itself implemented the state machine interface. A state transition in the controller was passed to the applications that were managed by the controller. Controllers themselves could be grouped and managed by other controllers, and so a hierarchy of state machines could be formed. On the top of this hierarchy was a single controller, the root controller. By bringing the root controller to the state RUNNING the entire distributed system was brought to data taking mode. Figure 2.6 shows a screenshot of the graphical user interface that is used to operate the state of all run control applications.

To communicate state changes and other information among the applications, an interprocess communication framework was needed. In ATLAS, CORBA [15] was used for this purpose. Services and frameworks for logging, error reporting and configuration were provided to the applications and allow centralized control of the computing farm to the operator in the control room.

2.3 Run 1 Performance Data

During the operation of the ATLAS experiment in the first LHC run from 2009 to 2013 many operational data were gathered. These give an insight



Run number 313878
Lumi Block 35

	Number	Rate
Level 1	4506399	11.68 kHz
HLT	3631	17.80 Hz
Recorded	5208	20.00 Hz

```
RUNNING  RootController
  Online Segment
  Infrastructure
  RUNNING  TDAQ
    Infrastructure
    ABSENT   AppOks2Coral
    UP       ddcdtATLAS_ATLGCSDDC
    RUNNING  L1CentralTrigger
    RUNNING  HLT
    RUNNING  L1Calo
    RUNNING  TriggerConfig
    RUNNING  TRP_Segment
    RUNNING  BeamSpotController
    RUNNING  olc2hlt_ctrl
    RUNNING  TDAQ_Monitoring
  RUNNING  GlobalMonitoringSegment
  RUNNING  InnerDetectors
  RUNNING  Calorimeters
  RUNNING  MuonDetectors
  RUNNING  ForwardDetectors
  RUNNING  DQMSegment
```

```
RootController
  CHIP-ATLAS
  Calorimeters
  CoralServerATLAS
  DDC
  DF
  DFConfig
  DQM
  DQMConfig
  DQMSegment
  ForwardDetectors
  GlobalMonitoringSegment
  Histogramming
  Histogramming-Global-iss
  ISRepository
  InnerDetectors
  L1TriggerRates
  MTS
  Monitoring
  MuonDetectors
```

TIME	SEVERITY	APPLICATION	NAME	MESSAGE
11:44:51	WARNING	RPC-BC-RODRX-...	RPCMessage	RXSL Killed s24 FIFO CHANNELS 6

sev=ERROR or sev=WARNING or sev=FATAL

Figure 2.6: The graphical user interface that is used by operators in the ATLAS control room.

into the performance characteristics of the trigger and data acquisition system. Of particular interest here is a dataset that describes the load on the different ROS PCs generated by the L2 trigger. The distribution of requests in one specific time-interval[1] is shown in Figure 2.7.[2] Clearly the load on the ROS PCs was distributed unevenly. The reason for this becomes clear when looking at the request rate per L2 trigger algorithm. The L2 trigger system made accept or reject decision based on a few tens of algorithms. These algorithms were based on different properties of the recorded data. Figure 2.8 shows the number of requests on the ROS PCs for two L2 algorithms. The algorithm in Figure 2.8a was based on particle trajectory reconstruction and thus requested data from the Pixel, SCT and TRT ROS PCs. The algorithm in Figure 2.8b, however, was based on information from the calorimeters and

[1]The time-interval in which the operating conditions of a particle collider are constant is called *lumi-block*, because the collider delivers a constant luminosity in this time-interval.

[2]Data were recorded during run number 0209025 and lumiblock 176.

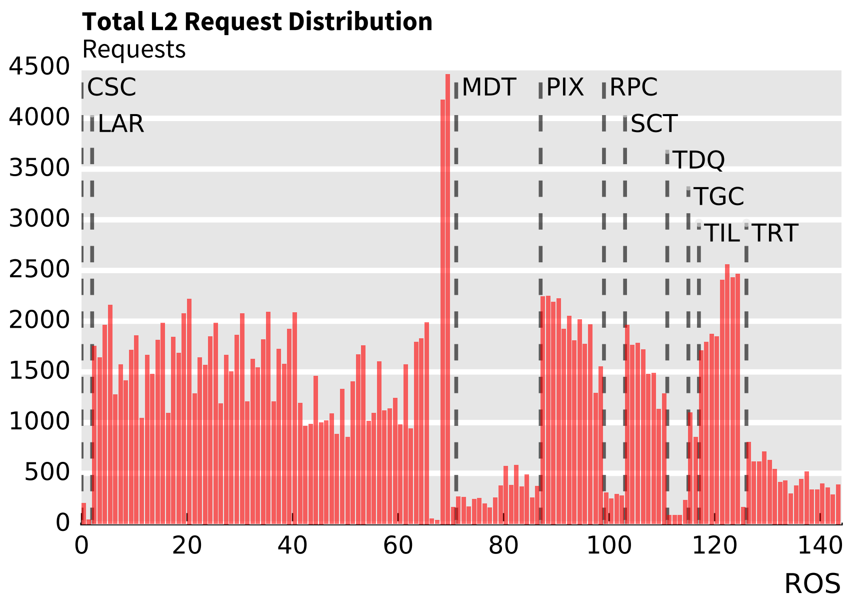


Figure 2.7: Distribution of L2 requests on the 151 ROS PCs. Data were recorded under typical run conditions.

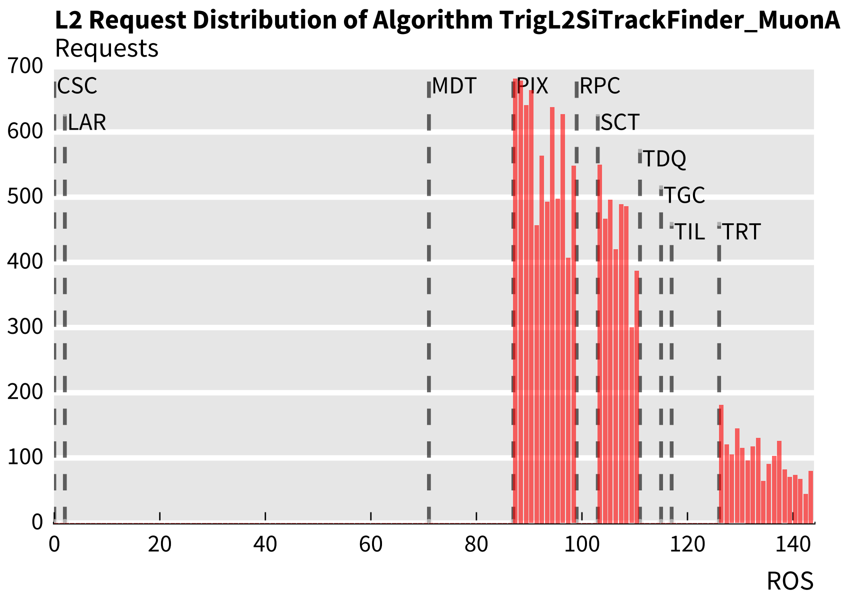
therefore requested data from the respective calorimeter ROS PCs. Which algorithms were used and how much weight was given to their output were configuration parameters of the ATLAS trigger system.

The ROS request distribution shows not only significant differences between ROS PCs of different detectors, but also between ROS PCs of a single detector (e.g. Liquid Argon (LAr)). As a consequence the assignment of ROLs to ROS PCs had to be balanced when the operational parameters of the LHC changed and more data were transmitted on the ROLs. Since the ROS PCs were connected to the RODs via point-to-point S-Links, the Run 1 architecture did not allow dynamic load balancing. The only possibility to balance the system load was to statically reassign ROLs to ROS PCs. This required a physical intervention on the ROS PCs. This had to be done during Run 1 for example for the Pixel detector. On the other hand systems with a low load had spare resources that are not used, which decreased the efficiency of the system. It is thus desirable to strive for an even load distribution.

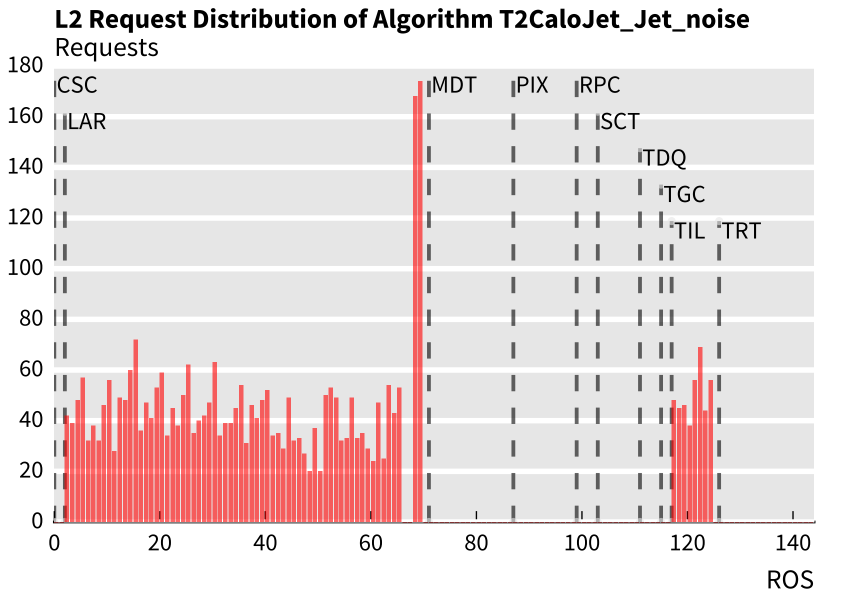
2.4 LHC Run 2 (2015 – 2018)

During the long shutdown phase in 2013/14 (Long Shutdown 1, LS1) the LHC was halted for maintenance operations. The bunch crossing rate was increased to the nominal LHC rate of 40 MHz.

Due to various factors also the average size of an event increased. This in-



(a) TrigL2SiTrackFinder MuonA



(b) T2CaloJet Jet noise

Figure 2.8: Distribution of L2 requests per L2 algorithm. Shown are the request patterns of two different L2 algorithms. The algorithm in (a) uses information from the trackers, the algorithm in (b) is based on information from the calorimeters.

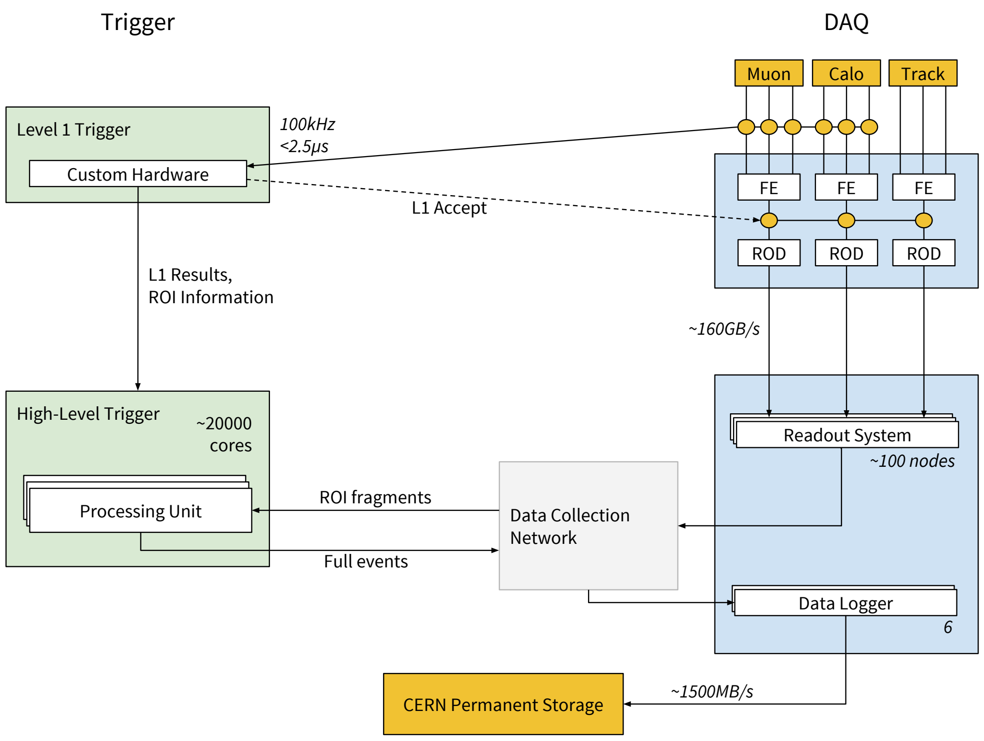


Figure 2.9: The ATLAS TDAQ system during LHC Run 2. L2 trigger and event filter have been merged to a single high-level trigger.

evitably increases the performance requirements on the ATLAS DAQ system, which makes an update necessary.

At the same time also the operational requirements of the DAQ were adjusted: the L1 target rate was increased from 75 kHz to 100 kHz, the target output rate/disk storage was increased to 1 kHz. Due to installation of new detector and trigger systems the number of Read-Out Links was increased from 1600 to 1800.

Three major components of the ATLAS DAQ system were upgraded in preparation for Run 2: the ROS PCs, the L2 trigger and event filter which were merged to a single high-level trigger, and the data collection network (Figure 2.9). Minor upgrades included a replacement of the RoIB and the SFOs. The upgrades described in this section are based on [16].

Detector Readout

Because of new performance requirements due to the increased incoming data rate as well as concerns for hardware obsolescene,[3] the ROS system experienced a major upgrade during LS1. A new S-Link interface, the RobinNP [16], was developed. The RobinNP is a PCIe Gen-1 x8 card. Via three QSFP modules it interfaces with up to 12 S-Links. The firmware of the card was redesigned and rewritten. Major processing tasks are now performed on the host system's CPU as opposed to the onboard chip on the ROBIN. This decision was taken with the future possibilty of future CPU upgrades in mind, which would allow a cheap and easy way of increasing ROS performance should the need arise. Up to two RobinNP cards are mounted in upgraded server PCs. The new servers have a 2U height profile, as opposed to 4U in the old ROS, allowing for a much denser system.

Trigger

One of the most significant changes in the LS1 upgrade was the redesign of the high-level trigger. In the first LHC run the high-level trigger was split into the L2 trigger and the event filter, with the event builder in between. In run 2 the L2 trigger and event filter are merged into a single high-level trigger. The new, unified trigger is executed on a single computer farm. The new design is much simpler, and also more efficient due to the removal of the communication step between the two trigger levels.

At the same time the HLT farm's compute capabilities were also expanded by the installation of new hardware. The number of available cores was increased from about 15,000 to more than 20,000 in Run 2.[4]

Another trigger-specific upgrade was the already mentioned increase of the L1 rate from 75 kHz to 100 kHz.

Networks

During LS1 the links connecting the ROS with the core routers of the data collection network were updated to from 1 GbE links to 10 GbE links. The overall architecture with two interconnected core routers as network backbone persisted. However, the routers were changed to operate in a bonding mode to form a router cluster for increased reliability and capacity. If one router fails data acquisition can continue at the same rate. Each ROS is equipped with four 10 GbE links and connected to both core routers. Each HLT rack is connected via two bonded 10 GbE links, while individual HLT servers are connected via 1 GbE links (see Figure 2.10).

[3]The ROBIN card was based on the PCI standard (predecessor to PCIe), which at the same time was slowly phased out of commercially available server hardware.

[4]During the course of Run 2 the farm size was further increased to a total number of 37,000 cores in 2017.

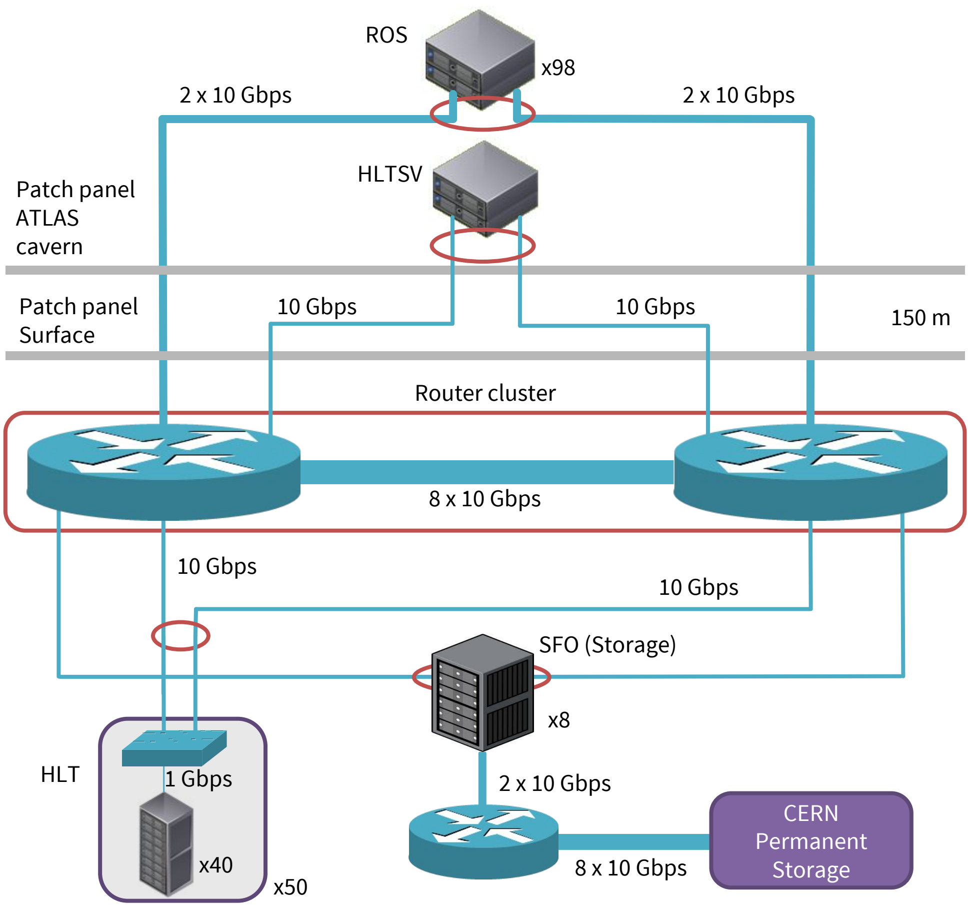


Figure 2.10: The ATLAS data collection network in Run 2. Two redundant core routers are the back-end of the network.

The back-end network became obsolete due to the merge of L2 trigger and event filter and was removed.

2.5 LHC Run 3 and beyond

The analysis of operational data presented in Section 2.3 made clear that the system architecture of Run 1 and Run 2 inherently leads to an unbalanced load distribution on the ROS PCs due to the static mapping of ROS PCs to detectors and sub-detectors. Moreover, a system based on static point-to-point links does not scale well. To reduce the load on a specific ROS machine and rebalance the load to a new node the ReadOut Links have to be reassigned. In case of a system failure of ROS or ROD the corresponding ReadOut Links become unavailable for recording. RODs and ROSs are therefore single points of failure.

Quality-of-service control where one traffic type is prioritized over another can be important. For example, a detector control data stream like a critical temperature sensor has a high priority since it is essential for a safe operation of the experiment. With the point-to-point S-Links of Run 1 and Run 2, a quality-of-service control is only possible on a per-link basis. A network with multi-path switching is more flexible in this regard and can also tolerate link failure to some extent.

Many of the low-level DAQ components from the Run 1 and Run 2 architecture like the RODs are custom, purpose-built electronic devices. Development and maintenance of these components are significant cost factors in the operation of the ATLAS experiment. Shifting requirements due to the LHC upgrade programs create the need for an ongoing development effort. Since the LHC experiments are planned to be operated for many decades, the continued supply of spare parts can also pose a problem for maintenance of these systems. Software, on the other hand, is relatively cheap and easy to develop and maintain. Computer technology has thus far undergone a constant, steady evolution, as can be seen by the continued validity of Moore's Law.

In this thesis I explore the potential of DAQ systems that make use of commercial off-the-shelf (COTS) components and computers early in the DAQ chain for ATLAS and similar HEP experiments from a computer science point of view. From the above mentioned points one could derive the following key ideas for such DAQ systems, and the ATLAS DAQ system in particular:

1. Scalability, load-balancing, failure-tolerance and quality-of-service can be improved by using dynamic switched networks over static point-to-point links early in the DAQ chain
2. Development and maintenance cost and effort can be reduced by pushing the usage of COTS technology and computer engineering methodologies closer to the detector.
3. Software solutions are preferred over hardware solutions where reasonably possible.

These key concepts enable the application of insights, techniques and knowledge from computer science to aid the development of DAQ systems for high-energy physics applications.

2.6 Centralized Data Distribution with FELIX

FELIX, for FrontEnd LInk eXchange, is an ATLAS project that incorporates the above points with the aim to provide a central, commodity data distri-

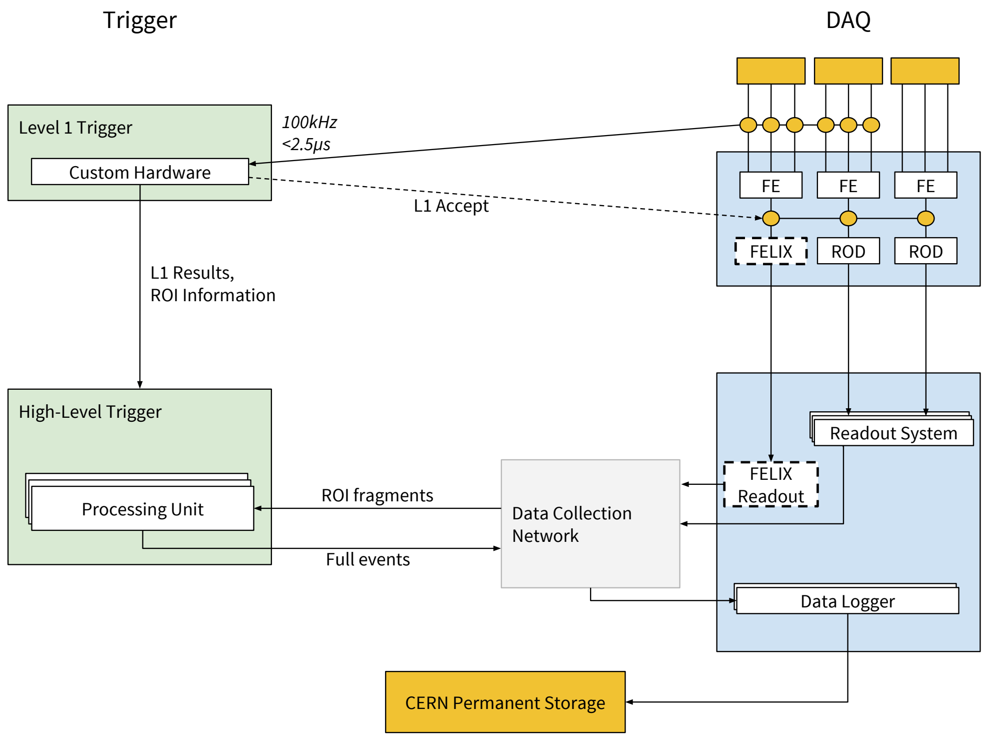


Figure 2.11: The architecture of the ATLAS trigger and data-acquisition system as planned for LHC Run 3. The FELIX Readout components implement the functionality of both the ROD and ROS in the old system.

bution layer early in the DAQ chain. FELIX consists of PC-based nodes that sit between the ATLAS detector front-end electronics and the network peers implementing, most likely in software, functions previously performed in detector-specific RODs. The new FELIX-based ATLAS trigger and DAQ system is shown in Figure 2.11. The project is a joint development effort of several institutes.[5]

The front-ends are connected to FELIX nodes via radiation-hardened optical links (called the Versatile Link [17]), on top of which the GigaBit Transceiver (GBT) [18, 19] protocol is employed. The Versatile Link and the GBT protocol are CERN projects providing a uniform, reliable, radiation-hardened link for high-energy physics experiments. The ATLAS experiment as well as other LHC experiments are phasing out custom link protocols in favour of

[5]At the time of this writing the FELIX development team consists of members from Argonne National Laboratory (USA), Brookhaven National Laboratory (USA), CERN (Switzerland), NIKHEF (Netherlands), Paderborn University (Germany), Radboud University Nijmegen (Netherlands), Royal Holloway University of London (UK), University of California, Irvine (USA), and the Weizmann Institute of Science (Israel).

the Versatile Link and GBT. In the following the combination of the Versatile Link and the GBT protocol will be simply referred to as *GBT link*.

On the back-end side, FELIX nodes connect via a switched high-performance network to the ATLAS DAQ system, detector control system, as well as detector specific calibration and monitoring systems. All these systems are implemented as software application running on servers connected to the FELIX network. The RODs, which where previously implemented as dedicated electronic components, can be implemented in software and the functionality can be merged into the ROS functionality. With all traffic flowing via FELIX systems there is now one single way for DAQ and auxiliary systems to communicate with detector front-end electronics.

FELIX will also interface with the existing TTC system as described earlier. FELIX receives TTC streams and can distribute the information downstream to detector front-end electronics, as well as upstream to the DAQ and other systems. TTC data that are sent downstream via the GBT links are forwarded with a fixed, low-latency delay to ensure accurate timing.

2.7 Related Work

The original ATLAS trigger and data acquisition system that was in use during the first LHC run from 2009 until 2013 is described in [7], its evolution to the system that is currently used in the second LHC run (from 2015 until 2018) is described in [20]. The Data Collection software framework used in ATLAS TDAQ is presented in [21], which is one of the first examples of common communication and message passing frameworks in ATLAS.

The LHCb project currently uses a combination of a low-level trigger and a high level trigger, similar to the ATLAS DAQ architecture in Run 2. The plan for the 2018 upgrade is to completely remove the low-level trigger and therefore read out the detector at the bunch crossing rate of 40 MHz. The foreseen detector readout system is based on GBT links and custom electronic component using the ATCA standard. As an alternative, a proposal for PCIe-based readout is also being investigated [22,23]. The system uses the PCIe40, an FPGA PCIe board developed by the LHCb collaboration. In this scenario the PCIe40 boards would be housed directly in the event builder PCs.

ALICE plans to deploy GBT links in their detectors and is developing a common interface between detector electronics, trigger and DAQ, called the CRU (Central Readout Unit) [24]. The implementation of the CRU is based on PCIe and the PCIe40 board from LHCb.

The CMS experiment upgraded their DAQ system for the current LHC Run 2. In the CMS architecture, low-level custom electronics connect to the PC-based DAQ system using a subset of the TCP/IP protocol over Ethernet links that is implemented in FPGAs [25,26].

Chapter 3

Architecture of a COTS-based Read-Out Switch

3.1 Overview

One of the core concepts that form the basis of the work presented in this thesis is the use of commercially available components for the purpose of detector readout. The FELIX project applies this principle at least in two distinct aspects.

First, the FELIX devices, based on commercial PC technology, minimize the need for custom components. Some custom hardware is unavoidable, for example detector links and TTC connections are very specific to high-energy physics and require custom developed hardware and firmware components. In FELIX connections to detectors and TTC system are implemented on a PCIe card with an on-board FPGA.

Second, FELIX functions like a switch that connects multiple endpoints (detectors and DAQ systems) and routes data between them (Figure 3.1). This characteristic of FELIX makes it easy to implement DAQ functions in software running on COTS servers as opposed to dedicated electronics, because the connection to detector front-ends that might require custom devices is already handled by FELIX. An example are the the RODs, which were typically implemented as VME modules in LHC run 1 and 2. With FELIX, the ROD functionality in Run 3 can be provided by software components running on server PC.

3.2 Detector Connectivity

The Versatile Link [17] is an optical link technology developed at CERN targeted at high-energy physics experiments. The nature of these experiments requires the link and electronics to be radiation-resistant. The Versatile Link is a full duplex link.

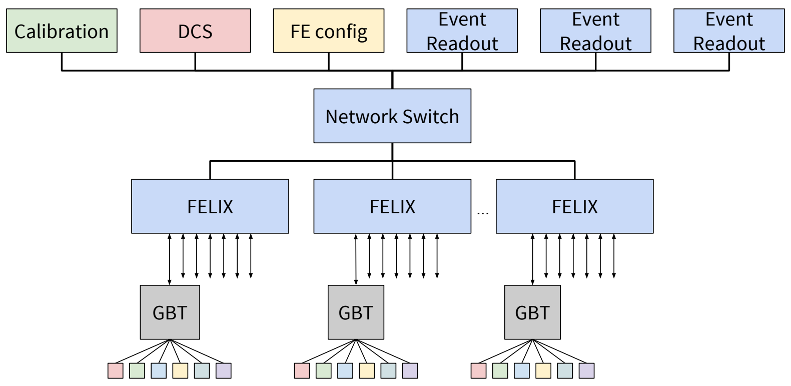


Figure 3.1: FELIX devices connect to multiple DAQ system endpoints.

The Gigabit Transceiver (GBT) protocol [18,19] is used on top of the Versatile Link. A widely available chip, the GBTx, allows the serialization of data streams according to the GBT protocol. The GBT protocol supports a standard mode which includes forward error correction using a Reed-Solomon code, and a wide mode with increased bandwidth but without forward error correction. The protocol uses virtual lanes, so-called e-links, to handle multiple parallel streams on a single, physical link. At the bunch crossing rate of 40 MHz the GBT protocol transmits packets called GBT frames of 120 bits, of which 80 bits (standard mode) or 112 bits (wide mode) are usable for user data. An e-link can be 2, 4, 8, or 16 bit wide, meaning that 2, 4, 8, or 16 bits of a GBT frame are utilized. A single GBT link can contain between 5 and 40 e-links in standard mode, or between 7 and 56 e-links in wide mode. In addition two special 2 bit e-links are part of the GBT frame headers. These are used as channels for the detector control system and for configuration of the GBTx chip itself. E-links have a net bandwidth of 80, 160, 320, or 640 Mbps. An illustration of the GBT protocol is shown in Figure 3.2.

The FELIX project decided to make two additions to the pure GBT protocol. The first addition is the definition of a packet encoding protocol on top of the standard e-link information. E-links are stream-based as opposed to packet-oriented protocols and therefore packet boundaries would be lost without an additional encoding. FELIX supports an 8B/10B encoding as well as an High-Level Data Link Control (HDLC) encoding. Packet boundaries are transmitted as out-of-band characters in the respective encodings.

The second addition is a third operation mode next to the standard and wide GBT modes. The third mode does not support e-links, but instead uses an 8B/10B encoding to carry a single packet stream on the link. This mode provides more bandwidth and is thus more efficient than the other modes, but sacrifices forward error correction and e-links. This third mode is refered to as full mode, and is intended for applications not requiring radiation-hard

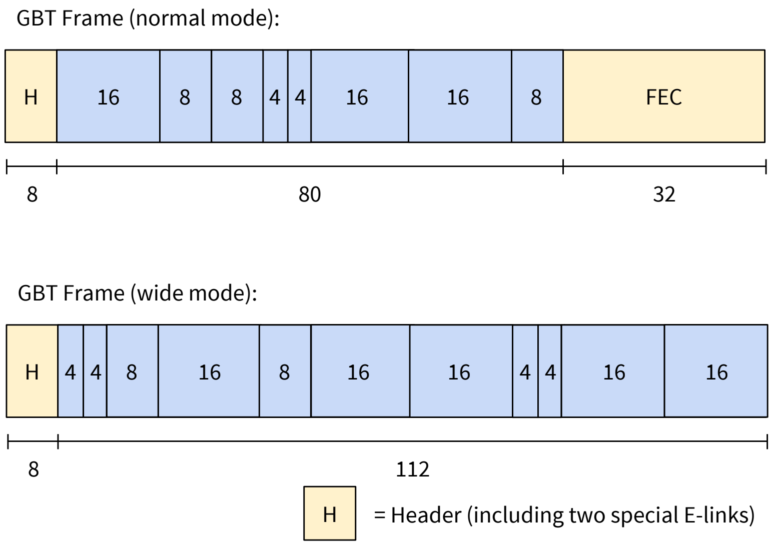


Figure 3.2: Illustration of the GBT Frames in normal and wide mode. The 80 bit or 112 bit payload field is divided in 2, 4, 8, or 16 bit E-links.

links. An example of this would be trigger electronics that are hosted in the ATLAS service cavern, and thus not in a radiation area.

3.3 The Detector Link Interface Card: Hardware, Firmware and Low-Level Software Tools

The PCIe card in the FELIX PC that connects to the detector links is simply called the FLX card. It is currently available in three versions (also see Table 3.1):

FLX-709 This card is based on the Xilinx VC-709 [27] development kit. The card hosts a Xilinx Virtex-7 VX690T FPGA and has four cages for SFP+ transceivers. It connects the host PC via a PCIe Gen-3 x8 interface. The FLX-709 only supports up to four GBT links, but the hardware is commercially available and relatively inexpensive, which makes it a good board for development and test setups.

FLX-710 The FLX-710 is based on the HiTech Global HTG-710 [28] PCIe board with the Xilinx Virtex-7 X690T FPGA. The card has two CXP sockets to connect two 12-channel full-duplex CXP transceivers. It can therefore support up to 24 GBT links. The board also has a PCIe Gen-3 x8 interface. The FLX-710 was the initial development board used by the FELIX project.

	FLX-709	**FLX-710**	**FLX-711**
Board	Xilinx VC-709	HiTech Global HTG-710	Custom
FPGA	Xilinx Virtex-7 XV690T	Xilinx Virtex-7 X690T	Xilinx Ultrascale XCKU115
PCIe Interface	Gen-3 x8	Gen-3 x8	Gen-3 x16
Links	4	24	48
TTC	via TTCfx FMC	via TTCfx FMC	onboard connector

Table 3.1: Specifications of the three generations of FLX cards.

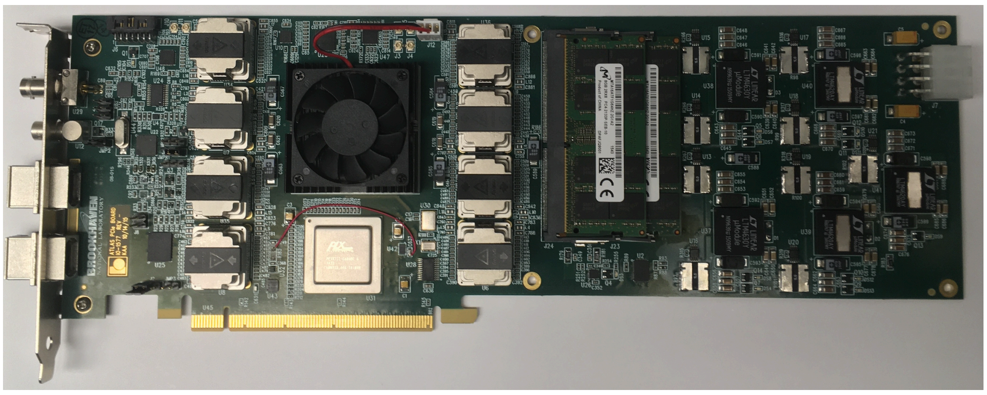


Figure 3.3: Hardware prototype of the FLX-711 card.

FLX-711 The FLX-711 (Figure 3.3) is based on a custom PCIe board that was originally designed for use in the LAr detector community.[1] The FLX-711 has a Xilinx Kintex Ultrascale XCKU115 FPGA, a PCIe Gen-3 x16 interface and supports up to 48 optical links in full-duplex. The FLX-711 is a candidate for the final card that will be used in the Run 3 FELIX installation.

Multiple FLX cards can be installed in a FELIX PC to increase link density. The firmware in the FLX card's FPGA contains the logic to interface between detector links and the host PC. The firmware can be divided into six major components (see Figure 3.4): a) the GBT interface, b) the Central Router, which handles data forwarding from and to e-links, c) a DMA engine and interface to the PCIe bus called *Wupper*, d) configuration and register map, e) TTC interface and f) a housekeeping module to configure and control clock resources and other peripherals on the FLX card.

[1]The original use of the card is for testing the Liquid Argon Trigger Digitizer Board (LTDB) which is a sub-system of the ATLAS L1 trigger.

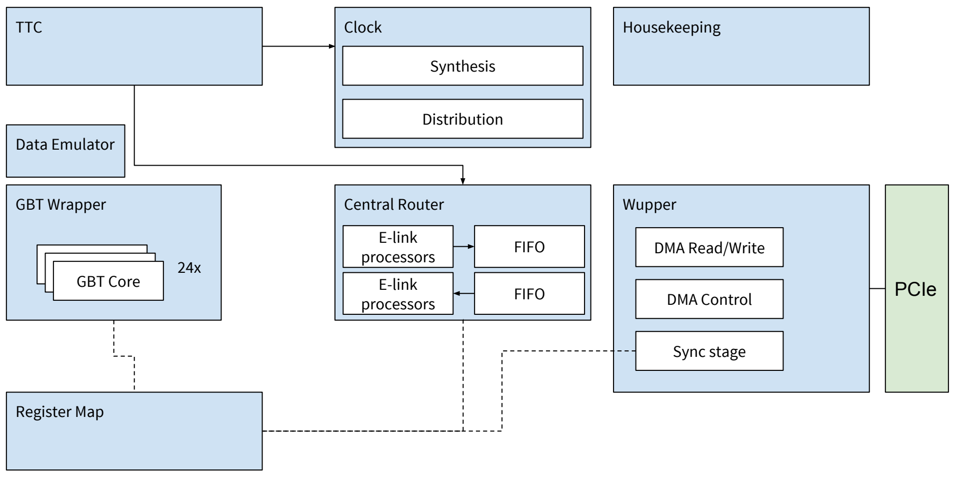


Figure 3.4: Block diagram of the FLX card firmware.

The GBT Interface

The GBT interface is a wrapper around an HDL softcore provided from the CERN GBT team. It manages the encoding and decoding of GBT frames (in the case of standard and wide mode) and the operation of the optical transceivers.

The Central Router

The Central Router encodes and decodes the packet stream on top of the GBT streams, and encodes and decodes blocks that are transmitted over the PCIe bus (see chapter 4 for a description of the encoding format). Per e-link two FIFOs are maintained for transmission of data in both directions, from the GBT links and to the GBT links. For performance reasons the Central Router will encode the variable-length packets coming from the detector links into one or multiple fixed-size 1 kB blocks. These blocks are then transferred via the PCIe bus. The Central Router also forwards TTC information to the GBT links.

DMA Engine and Interface to the PCIe Bus

As part of the development of the FLX card firmware a DMA engine called Wupper [29] for the PCIe Gen-3 hard block of the Xilinx Virtex-7 FPGA was developed. Wupper allows eight DMA transfers in parallel and a maximum transfer rate of 64 Gb/s. MSI-X is supported for the handling of interrupts.

Wupper supports two modes of operation: a single-transfer mode, and a continuous-transfer mode. In single-transfer mode a software application will request the DMA engine to perform a single transfer of data into a buffer in the host system memory. Once the transfer is complete, the DMA engine channel is free to be programmed with further DMA operations.

In the continuous mode on the other hand the Wupper engine treats the destination memory region of the transfer as a circular buffer and maintains a read- and write-pointer. Data can be fed continuously to the DMA engine by the FPGA logic. Wupper will write the incoming data to the destination buffer and update the write-pointer. User software can read the data from the buffer and will subsequently update the read-pointer. When the write-pointer reaches the read-pointer, Wupper will pause the transfer until the read-pointer is advanced. When the end of the buffer is reached, a wrap-around occurs and reading or writing continues at the beginning of the buffer. The continuous mode is useful to continuously transfer data to the host system without the need of reprogramming the DMA engine for every block of data, thus making the usage of the PCIe bus more efficient.

Configuration and Register Map

The FLX card parameters are controlled via a central register map. The register map has to be synchronized with software tools. To ease development a Python script is used to generate a VHDL register map as well as C source-code from a central register description file.

The TTC System: Trigger, Timing and Control

A connection to the TTC system is physically established via a FPGA mezzanine card (FLX-709, FLX-710) or via a dedicated on-board optical connector (FLX-711). The 40 MHz TTC clock is cleaned and extracted in the FPGA and is used to operate the optical transceivers for the detector links at a synchronized clock. Furthermore the TTC information is forwarded to the detector front-ends and host system via the Central Router.

3.4 The FELIX Software Stack

The FELIX software stack (Figure 3.5) runs on top of a RedHat-based Linux operating system (Scientific Linux). The software reads and writes data from and to the detector links, manages network connections to DAQ nodes, and encodes and decodes data packets. The software stack consists of a main server component to route data traffic between detectors and DAQ, various support libraries, and command-line tools for operation, monitoring and diagnostics. FELIX connects to a high performance network via a standard PCIe network interface card on the same host PC.

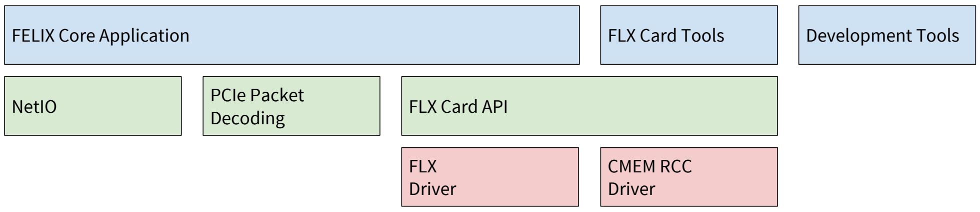


Figure 3.5: The FELIX software stack.

Drivers

Two kernel device driver modules are used to operate an FLX card. The first driver, which is simply called *flx*, maps memory regions assigned by the Linux kernel to the FLX card to the user address space. The memory regions are PCIe memory spaces that contain the FLX card's register map. Via ioctl operations a user application can request the driver to map the configuration register space of the FLX card to its own address space. Via direct manipulation of the mapped memory region the user application can configure and operate the card, start or stop DMA operations, or read status information. Furthermore, slow data transfer operations between the host system and the FLX card can be implemented by software writing to specific memory addresses which are read out by the FPGA firmware on the card. Giving user space applications direct access to the PCIe address space of the FLX card is more light-weight than encapsulating the accesses in the driver. The direct access method avoids context switches each time a register of the FLX card is accessed. However, this approach is only suitable for slow data transfers. Use cases requiring a higher throughput should use DMA transfers instead.

The flx driver is also responsible for handling MSI-X interrupts issued by the FLX card. Interrupts can be used to signal DMA transfer completions, full or empty FIFOs or other important events such as error conditions. The flx driver manages the handling of interrupts and notifies user space programs of the events. For debugging and diagnostic purposes the flx driver outputs basic operational data to the /proc/flx file in the proc file system.

The second driver in the FELIX software is *cmem_rcc*, which stands for contiguous memory (the appendix rcc is for historic reasons). To use the continuous mode of the Wupper DMA engine efficiently it is necessary to provide a large, contiguous block of system memory as a destination. On modern PC architectures memory is virtualized for user space applications, but devices on the PCIe bus typically operate with physical memory addresses.[2]

[2]There are architectures which use a IOMMU (input-output memory management unit) to provide virtual addresse also for peripherial devices. An example is Intel VT-d. However, not all modern CPUs include an IOMMU, and in order to maximize freedom in the choice of hardware for FELIX it is beneficial to opt for contiguous physical memory allocations over an IOMMU.

Thus, the normal memory allocation functionalities provided by the Linux kernel are not sufficient. The cmem driver closes this gap and allows the allocation of large regions of contiguous physical memory via ioctl system calls.[3]

Low-Level Tools

Several low-level tools with a command-line interface are available to configure and operate the FLX cards. The tools are listed in Table 3.2. The toolset is used by developers, testers and end users to test and debug the FLX card. The functionality of the tools is mostly implemented in a shared library libflxcard, which is also used by the FELIX core application (see below).

Core Application

The FELIX core application is the central process of a FELIX system. The application has the following functions:

1. Packet forwarding from the front-ends to the DAQ system: read and decode data packets from the FLX card and forward the packets to network endpoints based on dynamic routing rules.
2. Packet forwarding from the DAQ system to the front-ends: receive messages from network endpoints and write the contained packets to the e-links that are supplied in the message header.
3. Configure the FLX card, e-link configuration and operational parameters based on input from a configuration file.
4. Recover from a network endpoint failure. This could mean that data are temporarily routed to another endpoint, or that data transfers are resumed seamlessly after recovery of the network endpoint.
5. Gather statistics and performance metrics.
6. Report operational status information like the status of the detector links and warn in case of a hardware failure.

Figure 3.6 shows a diagram of the architecture of the FELIX core application. In the case of a system with multiple FLX cards, one application is run per card.

[3]In Linux kernel v3.5 the Contiguous Memory Allocator (CMA) was added, which provides similar functionality as the cmem driver. Since the Linux distribution Scientific Linux 6, which is very commonly used in high-energy physics, is still based on kernel series 2.6, the FELIX project uses cmem over CMA.

Name	Description
`FlxCard_scope`	An interactive debugging tool.
`flx-config`	Access to configuration options of the FLX card and the raw register map.
`flx-dma-stat`	Displays the status of the Wupper DMA engine.
`flx-dma-test`	Tests the functionality of the Wupper DMA engine.
`flx-dump-blocks`	Reads raw data from the card via DMA transfers and dumps the data into a file. Data come from GBT links or the internal FLX data generator.
`flx-i2c`	Reads from or writes to I^2C-connected devices on the FLX board.
`flx-info`	Displays generic information about the FLX cards, connected transceivers, link status or connected sensors.
`flx-init`	Initial configuration and clock setup. This program has to be executed once before data can be read from the card.
`flx-irq-test`	Tests the functionality of the MSI-X interrupts.
`flx-reset`	Resets the card or specific parts of the firmware to the initial state.
`flx-spi`	Similar to flx-i2c, this tool is used to communicate with devices connected via the SPI bus.
`flx-throughput`	A throughput benchmark to measure the transfer speed of the Wupper DMA engine.
`fdaq`	Read and decode GBT data from an FLX card and record the decoded data stream to a file. This application can be used as a standalone DAQ system for tests.
`fupload`	Transmit user-supplied data via GBT links.

Table 3.2: The low-level command line tools for the FLX card.

Front-End to DAQ Path

A single thread is responsible for reading data from the FLX card. As mentioned before, data that are transferred from the FLX card are encoded in fixed-size 1 kB blocks. The thread reading from the FLX card assigns these blocks in a round-robin fashion to one of multiple block queues. For each block queue a worker thread is started which reads the blocks from the assigned block queue and processes the blocks. Within the worker thread the blocks will run through a pipeline with the following steps:

1. The block is copied from the queue into a 1 kB buffer. The buffer is pre-allocated in the worker thread and retrieved from a queue. At this point the memory in the cmem buffer can be made available for further data transfers by advancing the read pointer.

2. The next step is to decode the block into variable-length packets, called chunks. This step is crucial for overall system performance. The problem of decoding blocks into chunks efficiently is therefore discussed in more detail in chapter 4.

3. The third step in the pipeline is to gather stastics. This pipeline module counts every chunk that is processed in a central data structure that is shared among all worker threads. Thread synchronization is ensured by using atomic variables as counters.

4. In this step meta information is extracted from the chunks. This information can be used for routing of data packets in the next step. At the time of this writing only the e-link number is stored. In the future this could be extended to extract an event identifier or other information.

5. The meta information from the previous step is used to assign one or more network endpoints as destinations for the chunk.

6. The last step of the pipeline is to send the chunk to the network endpoints assigned in the previous step.

The implementation of steps 4 and 5 is discussed in chapter 5. The FLX card transfers blocks via a DMA transfer into a cmem-allocated buffer. Pointers to the blocks in the cmem buffer and later pointers to the chunks in the blocks are passed through the pipelines. In the last step the chunk data are copied for each destination endpoint into a large output buffer.

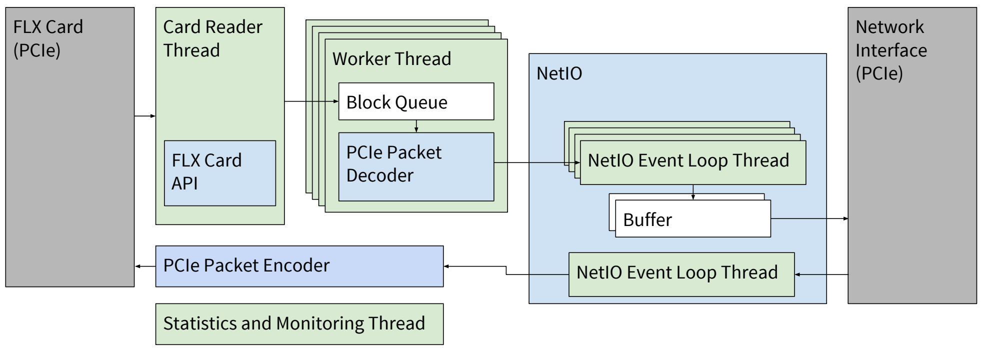


Figure 3.6: The architecture of the FELIX core application.

DAQ to Front-End Path

Traffic that is sent from the DAQ endpoints to GBT links is not performance critical and only requires a low throughput. Therefore a single thread is sufficient to receive incoming messages from network endpoints. Upon the arrival of a message, the message header is decoded and the message is written to the respective e-link. The software has to encode the data in 32 byte packets including a 2 byte header specifying the destination e-link prior to transferring the data to the card for proper handling by the FPGA.

3.5 Related Work

The PCIe40 board of the LHCb experiment is powered by an Altera Arria 10 FPGA and supports up to 48 GBT detector links and has a PCIe Gen-3 x16 [30]. The PCIe40 software stack uses a driver that exposes a bytestream interface. The driver uses I/O mapping of PCIe memory spaces in user space similar to the FELIX driver. Three streams are available: the main data stream including data from detector frontends, a metadata stream including size information about the data in the main stream, and an optional stream for TTC-like information. A key difference to FELIX is that the PCIe40 firmware will not transparently expose the GBT links, but instead combine data from different fibres that belong to the same event and output a single data stream. Furthermore the PCIe40 envisages a mechanism for detector specific data processing functionality like compression or data formatting within the PCIe40 FPGA, so PCIe40 devices are equipped with different firmware, depending on the connected detector. In the FELIX archicture these functions are implemented outside of FELIX in software FELIX clients, and thus all FLX cards have the same firmware.

The GBT-Based Expandable Front-End (GEFE) board is an FPGA-based GBT interface that is used in the CERN beam instrumentation group [24].

Chapter 4

Efficient Decoding of Detector Link Data Streams

4.1 Overview

Communicating efficiently with peripherial devices such as a network card or the FLX card is crucial for COTS-based DAQ systems like FELIX, as this will have a direct impact on overall system performance. This chapter focuses on the specific part of the FELIX software that handles the decoding of data packets transmitted over the PCIe bus by the FLX card. This piece of software plays an essential role in the overall performance of the FELIX application. First, I introduce the data encoding for the PCIe transfers and the software decoding algorithm. The performance optimizations and results of such a packet processing algorithm are then discussed. Finally, further analysis of the software will demonstrate that memory bandwidth is the primary bottleneck limiting the algorithm speed.

4.2 The PCIe Packet Format

As defined in the previous chapter, data coming over GBT links are organized as a stream of packets. Packets have variable length; their content is not standardized and depends on the source. Packet boundaries are defined by an 8B/10B encoding using 10B control symbols to mark the start and end of a packet, or an HDLC encoding. It is the task of the FLX card firmware to decode the packet stream and transmit packets, which are called *chunks* in FELIX terminology, over the PCIe bus into the host system's memory.

For technical reasons relating to the FPGA hardware and firmware as well as to ensure a high throughput, chunks are packed into fixed-size *blocks* of 1 kB in order to move them from the FLX card to the host system memory. Each block has a 4 byte header (see Table 4.1a) which contains a 2 byte start-of-block word, a 5 bit sequence number, and an 11 bit data stream iden-

Bit range	Description
0-10	Stream ID
11-15	Sequence Number
16-31	Start-of-Block Symbol (0xABCD)

(a) Block Header (4 byte)

Bit range	Description
0-9	Length in Byte
10	Reserved for length field extension
11	Chunk error bit
12	Truncation bit
13-15	Type field

(b) Subchunk Trailer (2 byte)

Table 4.1: The meta-data in block headers and subchunk trailers included in the packets transmitted over PCIe.

tifier. A single large chunk can span multiple blocks, or a single block can contain multiple small chunks. To fit into the fixed-size blocks, the variable-length chunks are split into so-called subchunks. Every subchunk ends with a 2 byte trailer, which contains the length encoded as 10 bit integer, a truncation bit, an error bit, and a 3 bit field indicating the type of this subchunk (Table 4.1b). The truncation bit is set when the FLX card receives a chunk that is longer than a configurable maximum size and indicates an error condition. The subchunk type can either be *first*, *last*, or *middle*, indicating this subchunk starts a new chunk, ends a chunk, or is in the middle of a chunk; *both*, indicating that this subchunk represents a full chunk that has not been split up; *null*, indicating that this subchunk does not carry data and is only used as padding to fill up the block; or *out-of-band*, indicating that the rest of the trailer is to be interpreted as an out-of-band signal. The block format is illustrated in Figure 4.1.

4.3 The Packet Decoding Algorithm

It is the task of the FELIX core application to decode the block stream and reconstruct the original chunks. The algorithm starts processing the subchunks at the end of a block. The subchunk trailer is read and a pointer to the data part of this subchunk is stored in a stack data structure. When a full chunk

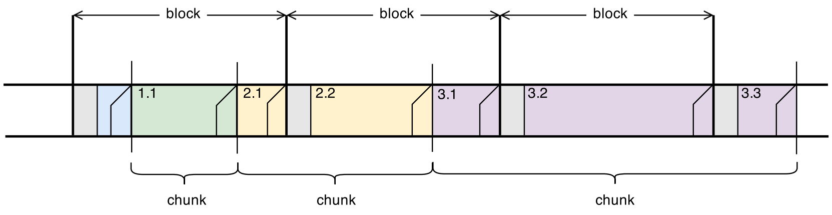


Figure 4.1: The block data format used to transmit data over the PCIe bus. The numbers indicate subchunks and their containing chunks, e.g., chunk 2 consists of two subchunks 2.1 and 2.2. Subchunk 1.1 is of type *both*, 2.1 is of type *first* and 2.2 is of type *last*. Each block starts with a 4 byte header (left-most rectangle in each block), each subchunk ends with a 2 byte trailer (slanted shape at the end of each subchunk).

has been read, the pointers on the stack are read in reverse order and stored in a data structure. Note that only pointers to the actual data are stored. Using scattered read and write routines (readv, writev on POSIX), the data can be copied into a consecutive memory region or, for example, passed on to a network card, enabling a zero-copy application design.

4.4 Profiling

For the performance measurements, I generated test data with mixed-size chunks and processed with the packet processing algorithm isolated and in-memory.

I used the Intel VTune Performance Analyzer utility [31] to perform an initial profiling of the algorithm execution. Several issues were revealed by the profile and could be fixed; details are disussed in Section 4.5.

As a next step, I used VTune to measure memory transactions while the benchmark was running. The results suggested a high CPI (clocks per instruction) of more than 2.5 in parts of the code as well as a large number of LLC (last-level cache) misses. The high number of LLC misses was expected since the benchmark was designed to read block data from main memory, as in a real-world scenario where data is copied to main memory via PCIe. The high CPI rate suggests that instructions are stalling and ILP (instruction-level parallelism) cannot be used effectively. This is an indication that the memory bandwidth of the test system is the performance bottleneck.

4.5 Optimizations

After the profiling gave some insight into the bottlenecks of the application, I iteratively optimized the code and re-evalued the performance with benchmarks after each optimization step. The first optimizations were guided by results of the VTune profile. The usage of STL containers could be improved by several modifications:

- By reserving memory upfront unnecessary memory allocations can be avoided. The vector data structure in C++ reserves memory dynamically as elements are added. When the maximum number of elements per vector is reached, the vector allocates a larger contiguous memory region and copies data from the old memory region to the newly allocated one. This is an expensive operation. Allocating a larger amount of memory upfront can reduce the number of memory allocations and copies.

- The C++11 standard `emplace_...()` methods were added to many data structures. With these calls objects can be directly allocated inplace in the data structures. This avoids the need for an additional copy.

- The algorithm uses a stack data structure to intermediately store subchunks (see Section 4.2). The stack data structure in the C++ standard library allows to use different container formats as back-end. I compared `std::vector` and `std::deque`, of which `std::deque` showed a higher performance.

Other optimizations involved the tuning of compiler options, the usage of NUMA-aware memory allocations and core-pinning to ensure that memory accesses are always local, and data prefetching using SSE intrinsics. These optimizations are represented by the "optimized" line in Figure 4.2.

Compiler option tuning In the optimized implementation I used the the following GCC compiler options:

`-O3`	This option enables full optimization. This option was already used in the benchmarks of the unoptimized algorithm implementation.
`-opt-prefetch`	The compiler will more aggressively try to prefetch memory before it is being accessed. This reduces the average time that an instruction needs to wait for data.
`-unroll-aggressive`	The compiler will more aggressively try to unroll loops.
`-march=native`	The compiler will generate code for the native CPU of the host system. This allows the compiler to use

all features of the current CPU. Generated code is not backward-compatible, i.e., it will not run on older CPUs. This is acceptable given that FELIX systems will have a well-defined hardware platform.

NUMA-aware memory allocations On a NUMA system with multiple processors a memory access of a thread to remote memory, i.e. memory that is not attached to the local CPU of the thread, can be costly. Using the library *libnuma* function `void numa_set_localalloc()` it can be ensured that all memory allocations using functions like `malloc` occur on local memory. Thus the cost of a remote memory access can be avoided.

Core pinning The task scheduler of the Linux operating system has the ability to move threads to any CPU core with free resources. This ensures a good resource utilization and avoids the oversaturation of individual CPU cores. However, the scheduler might decide to move a thread to a different NUMA node, which renders the previously described NUMA-aware local memory allocations ineffective. By pinning threads to a specific CPU these scheduler effects can be avoided. Again a libnuma function can be used: `int numa_run_on_node(int node)`.

Data prefetching When data is accessed by a CPU instruction it is possible that the instruction pipeline is stalled until the data is retrieved. Caches are used in CPUs to decrease the performance penalty of memory accesses. By explicitly fetching data into the caches before they are accessed, the number of cache misses can be reduced and the efficiency of an algorithm can thus be increased. Explicit prefetch instructions can ge generated by using intrinsic instructions of the compiler in the sourcecode. The GCC intrinsic instruction for this is `__builtin_prefetch`. Note that always a whole 64 byte cacheline is prefetched. Whenever a subchunk trailer is read in the optimized algorithm implementation, the cachelines that likely include the next subchunk trailer are prefetched. Since a block is read from back to front, the eight cachelines preceding the current trailer's cacheline are prefetched. It is likely that the next subchunk trailer is found within these cachelines.

Short chunks The runtime profile revealed that the usage of the stack data structure (see Section 4.3) is relatively expensive. In the case of subchunks of type *both*, which represent a whole complete chunk, the stack can be avoided entirely. Changing the implementation to omit the stack data structure in this case could improve the runtime significantly. The optimized algorithm directly yields a subchunk of type *both* to the caller without intermittently storing the data in the stack.

The packet processing benchmark results show that it takes much longer to process a block containing many small chunks than a block containing few but larger chunks (see Figure 4.2, "baseline"). This is expected since the amount of processing and data acesses increases when more trailers have to be parsed. On the other hand, the chance that a short chunk has to be split up in several subchunks is much smaller than for a large chunk. For example, 15 chunks with a size of 64 bytes fit into the 1020 bytes payload of a block without being split up into subchunks. As a result, only a single data pointer has to be stored for each of the 15 chunks. The situation with a data stream containing mostly small chunks is common enough to have a dedicated specialized implementation for this case. I introduced a new data type for short chunks that only contains a single data pointer. Construction of this object is significantly faster compared to a variable-length lists of pointers. As a result, the processing speed for blocks with short chunks is reduced significantly (see Figure 4.2, "optimized, new data type").

4.6 Benchmark Results

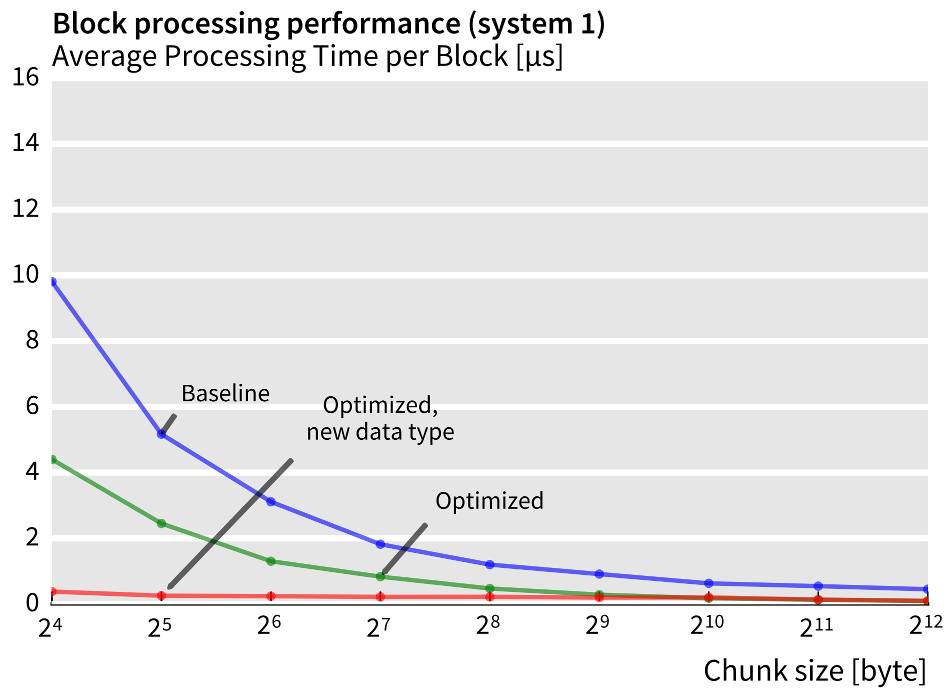
Benchmarks were performed on two different test systems, which are shown in Table 4.2. System 1 is a single-socket system with 4 cores and a modern CPU. System 2 is a dual-socket system with more cores than System 1, but slower memory.

To be able to handle the full load of an FLX card with 24 input links the packet processing software needs to be able to handle about 9.375 Mblocks/s, which is the maximum block rate for 24 links at a signalling rate of 3.2 Gb/s.

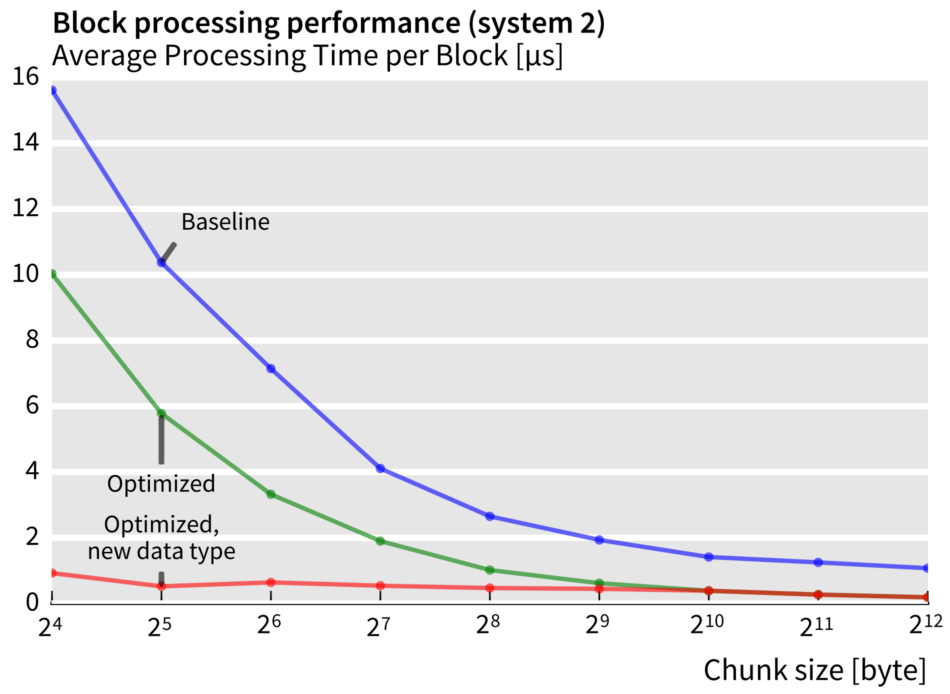
The optimizations discussed in the previous sections reduce the processing time per block significantly. The speedup is larger for small chunk sizes, an indication that the introduction of the new data type was successful. But also for larger chunk sizes the processing time could be reduced. Speedups of above 10x are achieved on both test systems.

Figure 4.2 shows a plot of the average processing time per fixed-size block. In the benchmark I assume that only chunks of equal size are stored in blocks. For each data point I generated 100 MB of chunk data and encoded in the fixed-size block encoding described earlier. Note that the average processing time for System 1 is in all cases better than for System 2. As will be discussed in the next section this can be attributed to the different memory speeds of the systems.

In a second experiment the influence of multithreading on the performance was analyzed. The FELIX core application uses multiple threads to decode the e-link block streams. In the experiment I emulated this scenario by starting multiple threads, each working on an independent buffer of test data. Similar to the previous experiment, each thread is given 100 MB of block data to process. Results for System 2 are shown in Figure 4.3. The



(a) System 1



(b) System 2

Figure 4.2: Average runtime per processed block for two different test systems and for different stages of the optimization process. Note that System 1 has faster memory modules.

	System 1	System 2
CPU Type	Intel Core i7-3770	Intel Xeon E5645
Architecture	Ivy Bridge	Westmere EP
CPU Clock Speed	3.40 GHz	2.40 GHz
Instruction Set Extensions	SSE4.1/4.2, AVX	SSE4.2
Nr of cores (real)	4	12
Nr of cores (Hyper-Threading)	4	24
Nr of CPUs	1	2
Memory	8 GB DDR3	24 GB DDR3
Memory Speed	1600 MHz	1333 MHz
Nr of Memory Modules	2	6 (3 per CPU)

Table 4.2: Specifications of the systems used for benchmarks. Note that the Core i7-3770 does support Hyper-Threading, but it was disabled during the benchmarks.

speedup is almost linear in the number of threads used, until the number of threads matches the number of real cores in the system. Using additional cores with HyperThreading does not present a significant advantage. With System 2 it is possible to process more than 10 Mblocks per second, which is roughly the minimum throughput threshold of 9.375 Mblocks/s for 24 links at 3.2 Gbps each, i.e., the amount of links foreseen to be connected per FLX card.

4.7 Memory Bandwidth Analysis

In this section I present a more in-depth analysis of the memory-access aspects of the implementation. First, I characterize the memory access pattern of the decoding algorithm and compare it to a memory benchmark with a similar access pattern. Second, a Roofline model analysis is performed to determine the bottleneck of the algorithm by theoretical means.

Memory Access Throughput

I used the PMBW [32] benchmark collection to characterize the test systems for different memory access patterns and test scenarios. PMBW allocates buffers of different sizes and processes these buffers using different routines with different memory access patterns. The benchmarks include several sequential scanning and random access routines. The results for System 1 can be seen in Figure 4.4. System 2 behaves similar but is slightly slower. For

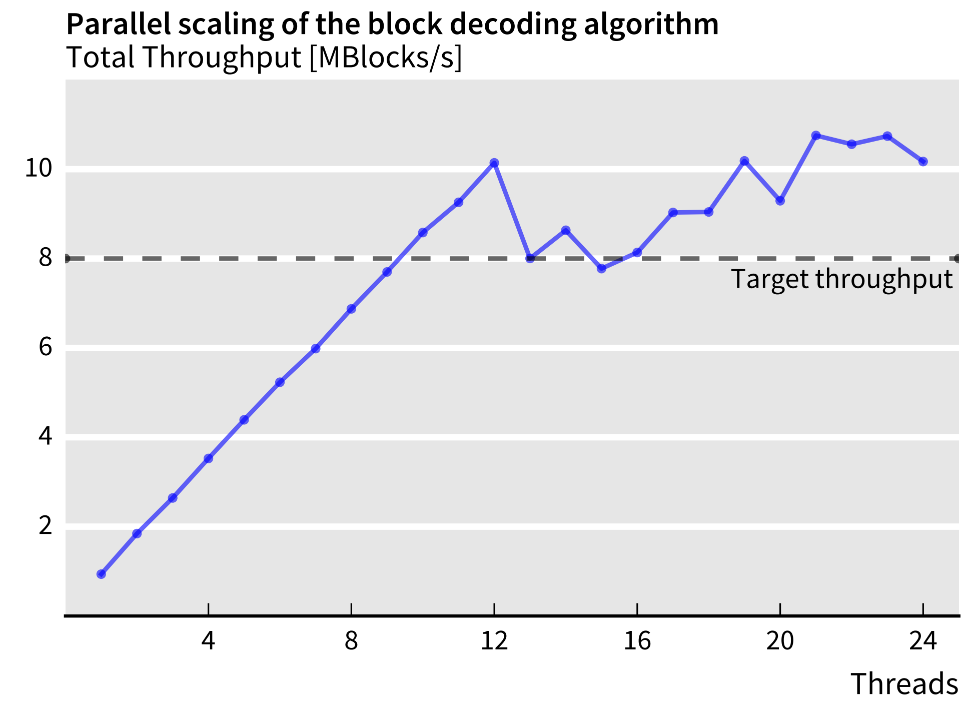


Figure 4.3: Overall block processing throughput for System 2 for different numbers of threads. The speedup is linear in the number of threads as long as HyperThreading is not used; the throughput saturates for more than 12 threads.

single-threaded scans from main memory 5–10 GiB/s were measured on System 2, compared to 10-20 GiB/s for the same scenarios on System 1.

The effects of caching are visible in the PMBW benchmarks: test scenarios with small buffer sizes benefit from the differently sized CPU caches. Caching can be ignored in FELIX though, since data are copied via PCIe to main memory. Therefore scenarios with large buffers that are fully stored in main memory will be used for the following discussion.

The memory access pattern in the packet processing algorithm consists of many short reads of 2 bytes for the subchunk trailers and fewer reads of 4 bytes for the chunk headers. Chunks are read sequentially. But, since only trailers and headers are processed, large parts of the data are skipped. This particular access pattern is similar to the Scan/Read scenarios with short data lengths in PMBW. The Scan/Read scenarios involve reads ("scans") of spatially sequential memory. These are depicted in Figure 4.4c.

One can see that better memory performance would be possible with a different memory access pattern, for example with reads of more than 16 bit, implying changing the data format of the block encoding. On the other hand, the current algorithm is significantly faster than scenarios with a completely "random" memory access.

The measured read bandwidth during the packet processing benchmark on System 1 was between 8 and 9 GB/s in the optimized version. This is slightly less than the peak bandwidth of ca. 11GB/s obtained by PMBW, single-threaded Scan/Read/32Bit/SimpleLoop for this access pattern. Bandwidth measurements were performed with Intel VTune.

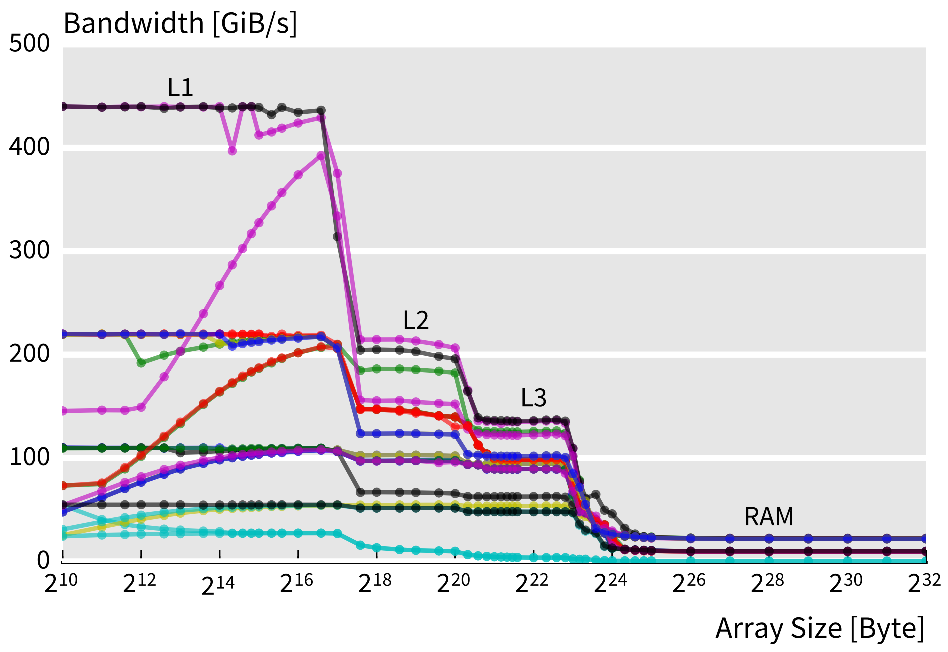
The memory access pattern results in a large number of cache misses. In a typical scenario with relatively short chunks, the majority of reads will be 16 bit reads for subchunk trailers. In most modern x86-based CPU architectures memory is always read in 64 byte cachelines. As a result, to process one chunk, a whole cacheline always has to be read even though only 2 bytes (the subchunk trailer) are used. The memory read efficiency is therefore only about 1/32 (assuming subchunks of at least 64 byte).

Roofline Model Analysis

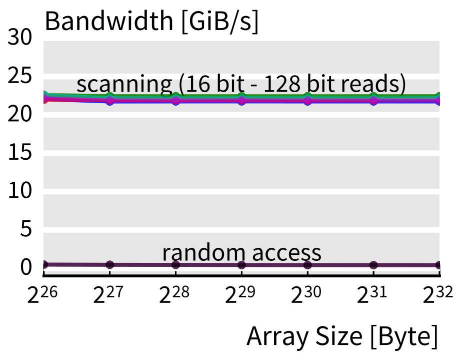
The Roofline model, as described in [33], is a modelling method used to describe the performance of an algorithm implementation in the context of limited memory bandwidth and computing speed. It is useful to identify bottlenecks and can give directions for optimization. In our case it is useful to support our hypothesis that the implementation of the packet decoding in FELIX is memory-bounded.

For the Roofline model, the performance P of an algorithm implementation is measured and related to its operational intensity I. The operational intensity is a property of the implementation and measures the average amount of instructions that are issued per byte read from memory. The measured performance P is then compared to two performance ceilings, the memory ceiling and the compute ceiling. Implementations with a low operational intensity are limited by the memory ceiling, whereas implementations with a high operational intensity are limited by the compute ceiling.

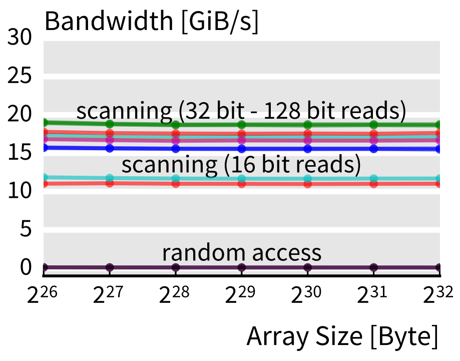
In the case of the packet processing algorithm I approximated the operational intensity by counting the number of operations that are needed to process one subchunk trailer, and dividing this number by the amount of memory that has to be read for the computation. The subchunk trailer is 2 byte long, but as indicated before a whole 64 byte cacheline must be read in order to process the 2 bytes. The operation count is estimated to be 6 operations per trailer, thus $I = 6\,\text{Ops}/64\,\text{byte} = 0.09375\,\text{Ops/byte}$. The memory ceiling is measured by the PMBW benchmarks and the performance ceiling is estimated as 2 Ops/Cycle per thread. This assumes pipelined integer operations with a 2-fold instruction-level parallelism. The Roofline model analysis of the packet processing algorithm for chunk of 64 bytes is depicted in Figure 4.5.



(a) All memory layers, 4 threads



(b) Main memory transfers, 4 threads



(c) Main memory transfers, 1 thread

Figure 4.4: An evaluation of test System 1's memory performance using PMBW. In (a) the effects of the different cache levels are clearly visible. The packet processing algorithm access data from main memory, which is shown in (b) and (c). Memory speed does not multiply with the number of threads used.

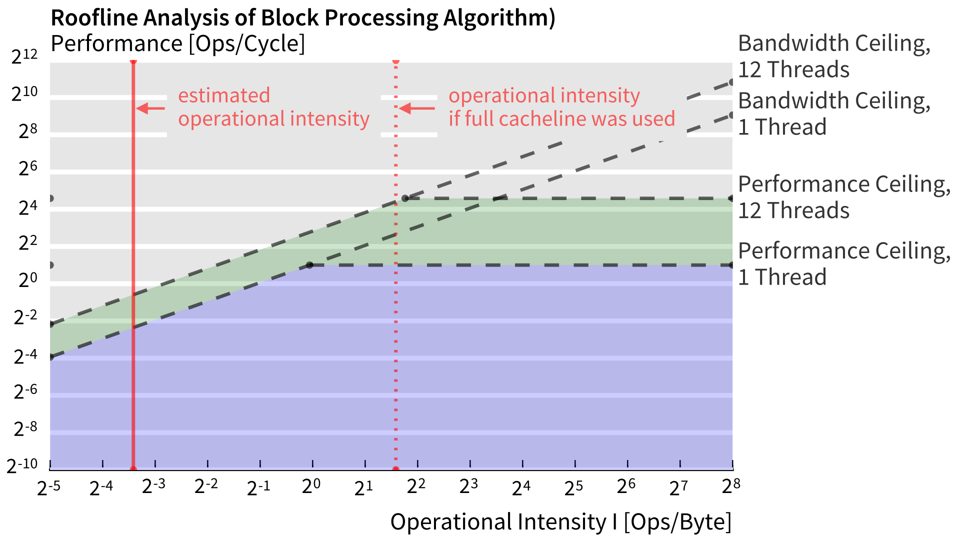


Figure 4.5: A Roofline model analysis of the packet processing algorithm on System 2 for 64 bytes long chunks. Due to the low read efficiency (only 2 of 64 bytes read are actually used for processing) the benchmark is limited by the memory speed.

According to the roofline model, the algorithm is clearly bounded by memory. This is expected since the algorithm is computationally not very demanding but has many memory accesses and cannot benefit from caches, and thus has a low operational intensity. An increased operational intensity would therefore increase the measured performance. This could, for example, be achieved by an improved data layout. If the FLX card would store subchunk trailers not interleaved with data but in a separate meta-data table, multiple subchunk trailers could be read at once when accessing a cacheline. This hypothetical scenario is indicated in Figure 4.5 by the dotted red line. According to the model, this optimization would shift the algorithm nearly into the compute-bound region. However, implementing this optimization would require support in the firmware of the FLX card. It is also not clear that the speedup would be as indicated by the model, since it would also require changes to the algorithm and, therefore, to the number of operations needed.

4.8 Conclusion

The implementation of packet processing algorithm compatible with the FELIX requirements requires several levels of optimizations. Advanced profiling tools were fundamental in achieving the necessary throughput performance. Furthermore, I demonstrated that the resulting algorithm is limited by the test system's memory bandwidth.

In order to independently validate this result, I performed a Roofline model analysis. This confirmed that the FELIX packet processing algorithm is memory-bounded. This analysis also provided additional insights on the Roofline model. While it is certainly useful, the Roofline model can only be seen as a first-order approximation, especially effective in classifying an implementation as memory-bound or compute-bound. Quantities like the operational intensity are hard to obtain, whether by measurement or just plain code analysis. Moreover, as today's CPUs get more and more complex and include features like ILP, pipeline architectures, or micro-ops, it is difficult to provide a good estimate of a CPU's peak performance.

4.9 Related Work

This chapter was originally published as part of the conference proceedings for the International Conference on Distributed and Event-Based Systems (DEBS) [Schumacher et al., 2015b].

Chapter 5

Fast Networking for DAQ Systems

5.1 Overview

Fast networking is crucial for COTS-based DAQ systems which compose of thousands of individual processes that need to communicate with each other. The definition of "fast" can vary for different use cases within a DAQ system: some subsystems might require high throughput and efficient link utilization, while others require low latencies to reduce communication delays as much as possible. Networking technologies including high performance fabrics, network topologies, software stacks and APIs are well researched topics in fields like high-performance computing (HPC).

Network infrastructures in online DAQ systems for high-energy physics (HEP) experiments however have fundamentally different requirements and require different methodologies and paradigms. The typical HPC use case for high-performance fabrics is large-scale computing with a single-program-multiple-data (SPMD) approach. The communication layer is often implemented with software layers like MPI [34], PGAS [35], or similar message passing or distributed shared memory schemes. Networking aspects as hardware like fabrics and switches, network topologies, and low-level protocols are similar in DAQ systems and HPC installations.

DAQ systems, however, are distributed systems with many different applications (see Chapter 2) and thus do not match the single-program-multiple-data (SPMD) paradigm well. Different networking software stacks compared to HPC are required for DAQ systems.

Also different requirements might impose implications on DAQ networks. A DAQ system, for example, has to be maintainable for decades due to the longevity of HEP experiments, which has an impact on the choice of hardware technology. Furthermore, DAQ system network infrastructures have to span relatively long distances of several hundred metres. For example, the

ATLAS read-out system is located underground, in a service cavern next to the experiment. The trigger farm is instead housed in a surface data-center. To connect systems in the two locations, distances of up to 150 m have to be bridged. Commonalities and differences between HPC and DAQ systems are listed in Table 5.1.

5.2 Networking in FELIX

FELIX has three principal network communication domains:

1. High-throughput communication from FELIX to DAQ network endpoints. Mainly this includes collision event data that are sent to the ROS and High-Level Trigger PCs. In this domain latency is not an issue, but the bandwidth requirements are relatively high. A single FELIX system might forward data at a rate in the order 100 Gb/s.

2. Low-latency communication from FELIX to DAQ endpoints. This includes operational data from detectors and sensors that are sent to the Detector Control System (DCS) system, detector calibration systems, or monitoring systems. The applications in this domain have small bandwidth requirements compared to the first domain, but latency can be critical for some applications. For the DCS for example it is critical that sensor information arrive within a fixed time frame to ensure that the system can react appropriately to unexpected events and ensure safety for the experiment. A typical latency requirement is a maximum latency in the order of a few hundred microseconds.

3. Low-latency communication from DAQ endpoints to FELIX. This domain mostly concerns configuration and calibration systems for detectors. Again bandwidth requirements are relatively low compared to

	HPC	**DAQ**
Performance	Low-latency	Low-latency, high-throughput
Communication Model	Message Passing, Shared Memory	Distributed System, Message Queue
Parallelism Model	SPMD	MPMD, Distributed System
Common Topologies	Mesh, Torus	Leaf-Spine
Longevity	Less than 10 years	20-30 years with periodic upgrades

Table 5.1: A comparison of networking in HPC and DAQ.

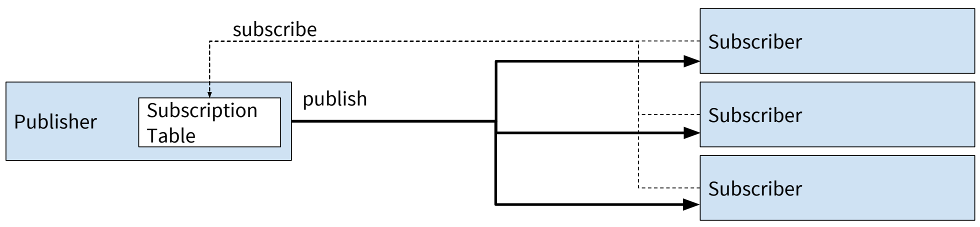


Figure 5.1: The publish/subscribe communication pattern. Publishers have no prior knowledge of subscribers. Subscribers are dynamically added to the subscription table by issuing subscribe requests.

the first domain. Configuration or calibration cycles however might require many iterations and therefore a short round-trip time. Again a latency in the order of a few hundred microseconds is desirable.

Especially considering the high throughput demands for the collision event data streams the FELIX project aims to operate the network links at a high efficiency, i.e. to utilize a large fraction of the available link bandwidth. High link utilization enables a dense system (in terms of connected detector links per FELIX), which has a direct impact on the cost of the data acquisition system.

FELIX devices are decoupled from the DAQ system. While the DAQ applications are started and stopped using the run control state machine (see section 2.2), FELIX devices are always active. Control systems like the DCS need to receive data from the detectors and monitor their status at all times. In this sense a FELIX device has a purpose similar to network switches: providing transparent connectivity between multiple endpoints. But unlike in a network switch, FELIX has to translate between the GBT protocol and the network links. It not only forwards data packets, but actively has to maintain connections (e.g. TCP connections) to the DAQ nodes. This creates several requirements for the to-DAQ communication of FELIX systems:

1. A FELIX device needs to know which e-links are relevant for which nodes in the DAQ network and to forward data accordingly.

2. A FELIX device should forward data only to DAQ system applications if the receiving application is in the running state.

3. DAQ application crashes or hardware failures need to be handled. The connection between FELIX and the DAQ node needs to be re-established once the crashed application is restarted. If a DAQ server crashes, a spare server needs to be able to take over the input traffic of the crashed server.

In FELIX these requirements are met by using a publish/subscribe system for the to-DAQ communication. A publish/subscribe system, as illustrated in Figure 5.1, has two types of actors: subscribers, which receive data, and publishers, which publish data. The publisher does not have any prior knowledge of the subscribers. Instead, subscribers send a subscription request to the publishers. Publishers receive the subscription request and add the subscriber to a subscription table. When publishing a message, a publisher looks up the subscription in the subscription table and sends the message to all subscribers. A subscriber can specify in the subscription request a mask to filter the messages that it will receive.

Subscribers need to know to which application they need to subscribe. The publishers periodically broadcast information about their state and configuration, for example which GBT links are handled by this particular application. Subscribers receive these broadcast messages and can select the publisher with relevant data based on the received information.

In the FELIX system, FELIX devices are publishers that publish chunks from e-links. DAQ, DCS, configuration, calibration and other systems are subscribers. These subscribers can subscribe to the e-links they are interested in. The publish/subscribe communication meets the three requirements: 1) subscribers know which e-link are relevant and will only subscribe to relevant e-links, 2) subscribers are aware of their state and will only subscribe when they are in running state, and unsubscribe when they leave the running state, and 3) if a DAQ application fails, it will resubscribe after it has restarted, and if an application is started on a spare host, it will issue a new subscription request from there.

The frontend-facing direction in FELIX uses simple point-to-point communication. Applications send a message to a FELIX system, a header in the message includes the target e-link. FELIX will then send the data part of the received message on the specified e-link.

5.3 The NetIO Message Service

The FELIX network stack is provided by a separate library called NetIO. I designed and implemented NetIO as a generic message-based networking library that is tuned for typical use cases in DAQ systems. It supports four different communication patterns: low-latency point-to-point communication, high-throughput point-to-point communication, low-latency publish/subscribe communication, and high-throughput publish/subscribe communication. Therefore it covers every FELIX use case.

NetIO has a back-end system to support different network technologies and APIs. At the time of this writing two different back-ends exist. The first back-end uses POSIX sockets to establish reliable connections to endpoints. Typically this back-end is used to use TCP/IP connections in Ethernet net-

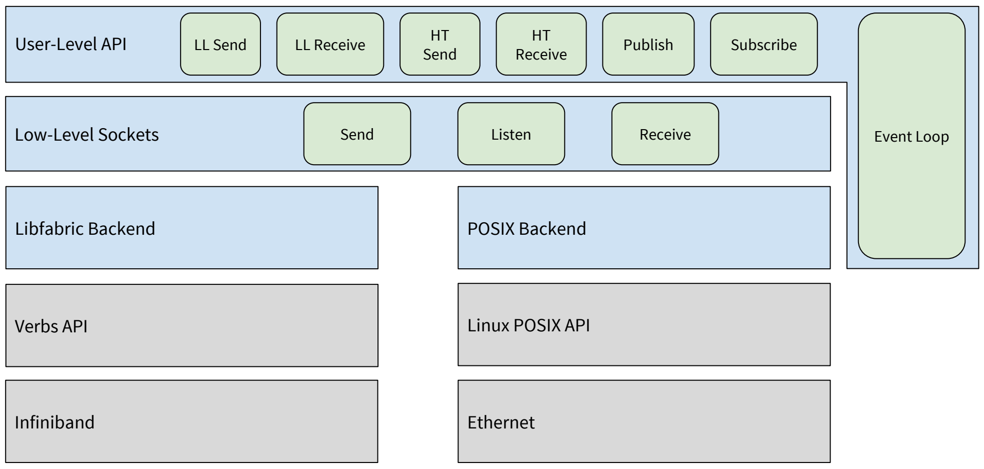


Figure 5.2: The NetIO architecture.

works. The second back-end uses libfabric [36] for communication. It is used for Infiniband and similar network technologies. Libfabric is a network API that is provided by the OpenFabrics Working Group.

The NetIO architecture is illustrated in Figure 5.2. There are two software layers within NetIO. The upper level contains user-level sockets. These are the sockets that application code interacts with. The different socket types are listed in Table 5.2.

The lower architecture level provides a common interface to the underlying network API. The common interface consists of three low-level socket types (a send socket, a listen socket and a receive socket), which are implemented by each back-end. The low-level sockets provide basic connection handling and simple transmission of messages between two endpoints. All higher level functionality like buffering, notification of user code via callbacks, or the publish/subscribe system are implemented in the user-level sockets. This maximizes code sharing among the back-ends, as only code that is specific to the underlying network technology is implemented in the low-level sockets.

Both architecture levels use a central event loop to handle I/O events like connection requests, transmission completions, error conditions, or timeouts. The event loop is executed in a separate thread. Its implementation is based on the epoll framework in the Linux kernel.

IP address and port are used for addressing network endpoints, even for back-ends that do not natively support this form of addressing. For the Infiniband back-end the librdma compatibility layer is used to enable addressing by IP and port.

5.4 User-level sockets

There are six different user-level sockets, of which four are point-to-point sockets (one send socket and one receive socket, each in a high-throughput and a low-latency version), and two publish/subscribe sockets (one publish and one subscribe socket). The publish/subscribe sockets internally use the point-to-point sockets for data communication.

A high-throughput send socket does not send out messages immediately but maintains a buffer in which messages are copied. Due to the buffering less, but bigger packets are sent on the network link. This approach is more efficient and yields a higher throughput. The average transmission latency of any specific message however is increased due to the buffering. A typical buffer size is 1 MB. Once a buffer is filled the whole buffer is sent out to the receiving end. Additionally a timer (driven by the central event loop) flushes the buffer at regular intervals to avoid starvation and infinite latencies on connections with a low message rate. A typical timeout interval is 2 s. A message is split if it does not fit into a single buffer. The original message is reconstructed on the receiving side.

A high-throughput receive socket receives buffers that contain one or more messages or partial messages. The messages are encoded by simply prepending an 8 byte length field to the messages. The high-throughput receive socket maintains two queues: a buffer queue, which contains unprocessed buffers that have been received from a remote, and a message queue, in which messages are stored that are extracted from buffers when they are processed. The high-throughput receive socket enqueues received buffers in the receive buffer queue. When user code calls `recv()` on a high-throughput receive socket, it will return the next message from the message queue. If the message queue is empty, the next buffer from the receive buffer queue is processed and the contained messages are stored in the message queue. When processing a receive buffer the contained messages are copied.

A low-latency send socket does not buffer messages. Messages are immediately sent to the remote process. Unlike for high-throughput send sockets there is also no additional copy: the message buffer is directly passed to the underlying low-level socket. These design decisions minimize the added latency of a message send operation.

A low-latency receive socket handles incoming messages by passing them to the application code via a user-provided callback routine, instead of enqueuing the messages in a message queue. This approach allows incoming messages to be processed immediately. In contrast to high-throughput receive sockets also no data copy is taking place, the receive buffer memory is passed to the user level code. After execution of the callback routine the receive buffer will be freed and accessing the memory by user-level code is an illegal operation. If necessary, a user can decide to copy the buffer in the the callback routine.

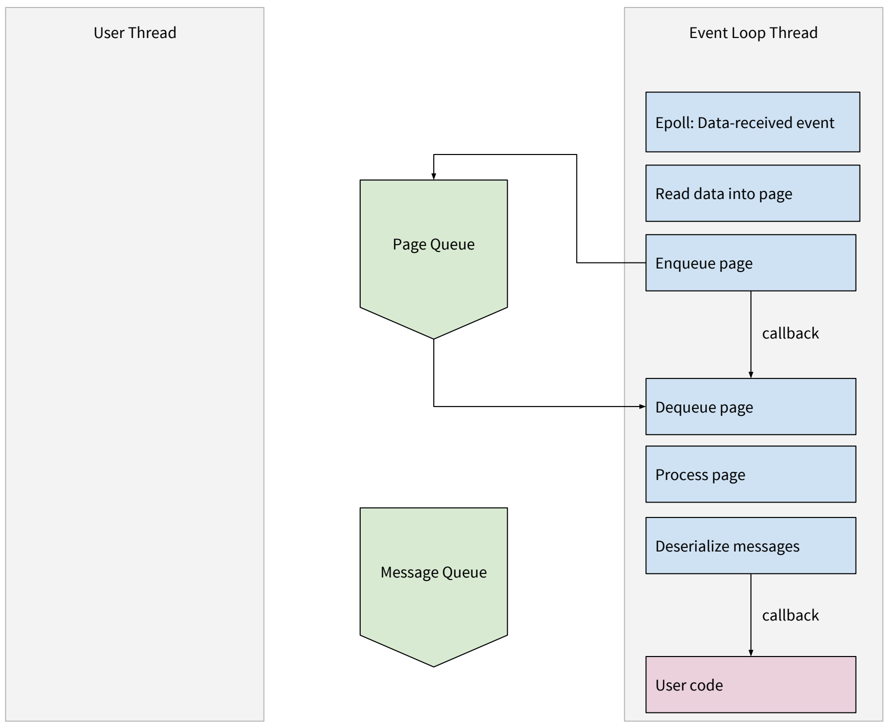


Figure 5.3: Processing of incoming messages in NetIO low-latency sockets. Note that all processing, including the execution of user code, is performed in the event loop thread.

High-throughput and low-latency receive sockets also differ in the way threading is involved in processing incoming messages. In both cases a buffer receive notification from a low-level receive socket is handled in the event loop thread. In high-throughput receive sockets the buffer is immediately pushed into the receive buffer queue, after which the event handler returns and the event loop thread is free to process further events. Parsing the buffer, extracting the messages and processing them with user code is done in the user thread. In low-latency sockets the event handler routine executed by the event loop thread will call the user-provided callback. Thus, all user code is executed by the event loop thread. The event handler will only return after the user callback is processed. This might block the event loop from processing further events for any amount of time. Users have to take care to implement sensible callback routines that do not block the event loop too long, or otherwise performance might degrade. The benefit of executing user code in the event loop thread is however that no latency is added by queuing of messages. Message processing in high-throughput and low-latency sockets is illustrated in Figures 5.3 and 5.4.

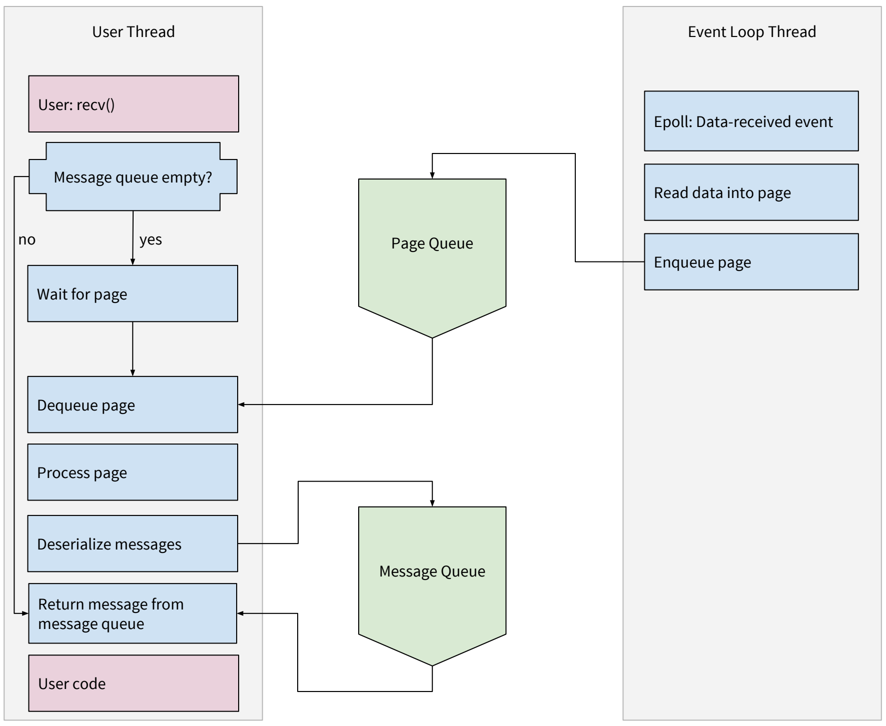


Figure 5.4: Processing of incoming messages in NetIO high-throughput sockets. After the data is received in the event loop thread, all processing is done in the user level thread. The event loop thread is freed up to process further incoming data.

5.5 Low-Level Sockets

The interface to the NetIO back-ends is provided to the user-level sockets by three types of low-level sockets: back-end send sockets, back-end listen sockets, and back-end receive sockets. Back-end listen and receive sockets are used on the receiving side of a connection. Back-end listen sockets open a port and listen for incoming connections by back-end send sockets. A back-end receive socket is created when a connection request arrives at a back-end listen socket. A back-end receive socket represents a single connection. A back-end send socket is used on the sending side of a connection. A back-end send socket can connect to a port opened by a back-end listen socket and send messages when the connection is established.

The back-end sockets provide callback entry points for user-level sockets that are called when a connection has been successfully established, a remote has disconnected, or data has arrived.

The low-level API also provides an interface for back-end buffers. These are buffers that are used by the back-end sockets and can be transmitted

Socket Type	**Description**
Low-Latency	
Send	A message that is posted is immediately sent to the remote endpoint without any delay.
Receive	A message that is received is immediately passed to the user code via a callback.
High-Throughput	
Send	Messages are copied into a large connection buffer instead of being sent immediately. The buffer is transmitted when it is full or after a timeout occurs.
Receive	When a message buffer is received the contained messages are copied into separate message datastructures and enqueued in a message queue. User code can read the received messages by calling `recv()` on the receive socket.
Publish/Subscribe	
Publish	When a message is published under a given tag, it matches the tag against a subscription table and send the message to all subscribed remote endpoints via either low-latency or high-throughput send sockets.
Subscribe	Sends subscriptions requests to publish sockets via low-latency send sockets and then receives messages via either a low-latency or a high-throughput receive sockets.

Table 5.2: The different types of user-level sockets in NetIO. An important feature in NetIO is the distinction between low-latency and high-throughput communication.

over the network. Back-ends might have special requirements on buffers. Libfabric for example requires that all buffer are previously registered in a central registry.

5.6 The POSIX Back-end

The POSIX back-end is straight-forward and uses the socket API that is defined in the POSIX standard. The back-end uses sockets of the SOCK_STREAM type, i.e., TCP/IP connections. The socket option TCP_NODELAY is set, which disables Nagle's algorithm. Nagle's algorithm can temporarily delay packet sends to reduce the number of TCP packets on the wire. The buffering capabilities of the user-level sockets however allow a more fine-grained control over packet delay, so Nagle's algorithm can be deactivated.

The POSIX socket API uses file descriptors to represent the sockets. These file descriptors are registered in the central event loop. Thus, when a connection request or a new message arrives, the corresponding sockets are informed and handler routines are executed. The POSIX sockets are configured to asynchronous, non-blocking mode, i.e., the O_NONBLOCK is set.

5.7 The FI/Verbs Back-end

Libfabric provides several communication modes to the user, for example reliable datagram (RDM) communication, reliable connection communication (which works like RDM but additionally provides message ordering), or RDMA. Libfabric be can used on top of several network stacks. For Infiniband the library utilizes librdma and libibverbs, for Intel OmniPath the native PSM2 interface can be used.

The NetIO FI/Verbs back-end uses the reliable connection communication model from libfabric. Libfabric also provides reliable datagram communication, but the reliable connection model preservers message order. Preservation of message order is important since a message can span multiple NetIO buffers (see section 5.4 on high-throughput sockets).

Libfabric provides so-called active and passive endpoints to manage connections. Passive endpoints listen to incoming connections, while active endpoints are the equivalents of sockets and are used to send and receive messages. The libfabric API is fully asynchronous, and connection management notifications are presented to the user as events that need to be handled. Each endpoint has an event queue in which connection events are stored. Libfabric allows to register a file descriptor with an event queue. When a new event arrives, the file descriptor becomes readable. The NetIO FI/Verbs back-end uses these event queue file descriptors and registers them in the central event loop.

POSIX sockets have internal buffers. When a message is sent on a POSIX socket, the data is copied into the internal buffer, from which the data is then sent to the remote process. The user-supplied buffer is usable again immediately after the send call. Similarly, a receiving POSIX receives data in an internal buffer, and a receive call will copy the data out of the internal buffer. The user does not need to supply a buffer in which data from the network can be received.

The FI/Verbs back-end is asynchronous and allows to send and receive messages without data copies; libfabric endpoints do not have internal buffers. When a message is sent, the user-supplied message buffer is used and no data is copied. To receive messages, a user needs to provide receive buffers. To manage the send and receive buffers, each active endpoint has a queue for completion events. Completions notify the user-space application of the result of the send or receive operation. After a send completion arrives, the corresponding send buffer can be reused for new send operations. After a receive completion arrives, the corresponding receive buffer is filled with a message from a remote host and can be processed. Like the connection management events the completion events can trigger a file descriptor. NetIO uses such completion file descriptors and registers them in the central event loop.

Libfabric requires send and receive buffers to be registered with the call `fi_mr_req`. The NetIO FI/Verbs back-end provides a data buffer interface that performs this registration step.

5.8 The Intel OmniPath Back-end

Intel OmniPath [37] is a recent fabric technology that is based on the TrueScale technology by the former QLogic company. On the software side OmniPath has a Verbs interface and is thus directly supported by NetIO via libfabric. OmniPath additionally provides a native API called PSM, which is also supported by libfabric. The libfabric PSM provider however currently does not support the reliable connection mode, which is needed for NetIO. NetIO on OmniPath therefore currently only works using the Verbs interface.

At the time of this writing the NetIO on OmniPath is still a work in progress and requires further investigation and development.

5.9 Benchmarks and Tests with NetIO

To evaluate the performance of NetIO I performed several experiments. As a reference point I use the ZeroMQ [38] library to compare NetIO against. ZeroMQ is a library that gained popularity in the HEP community and is used in several projects in the LHC and the LHC experiments. ZeroMQ provides point-to-point communication as well as a publish/subscribe system. Benchmarks are performed between two nodes connected via a single switch. The benchmark system configuration is described in Table 5.3. The systems are equipped with Mellanox ConnectX-3 VPI network interface cards, which can be operated 40G Ethernet mode or 56G Infiniband FDR mode.

The first benchmark scenario consists of point-to-point communication between the two systems using NetIO high-throughput sockets and a single connection. The sending side uses the NetIO test tool `netio_throughput` to send messages to the receiving node, which uses the program `netio_recv` to receive the messages. The throughput achieved for various message sizes is shown in Figure 5.5.

NetIO on Ethernet and ZeroMQ on Ethernet have a very similar peak performance of around 30 Gb/s. NetIO however reaches higher throughput

	System 1	System 2
CPU Type	Intel Xeon E5-2630 v3	Intel Xeon E5-2660 v3
CPU Clock Speed	2.40 GHz	2.60 GHz
Nr of cores		
real	8 per CPU	10 per CPU
hardware threading	16 per CPU	20 per CPU
Nr of CPUs	2	2
Memory	64 GB	64 GB

Table 5.3: Systems used for NetIO benchmarks.

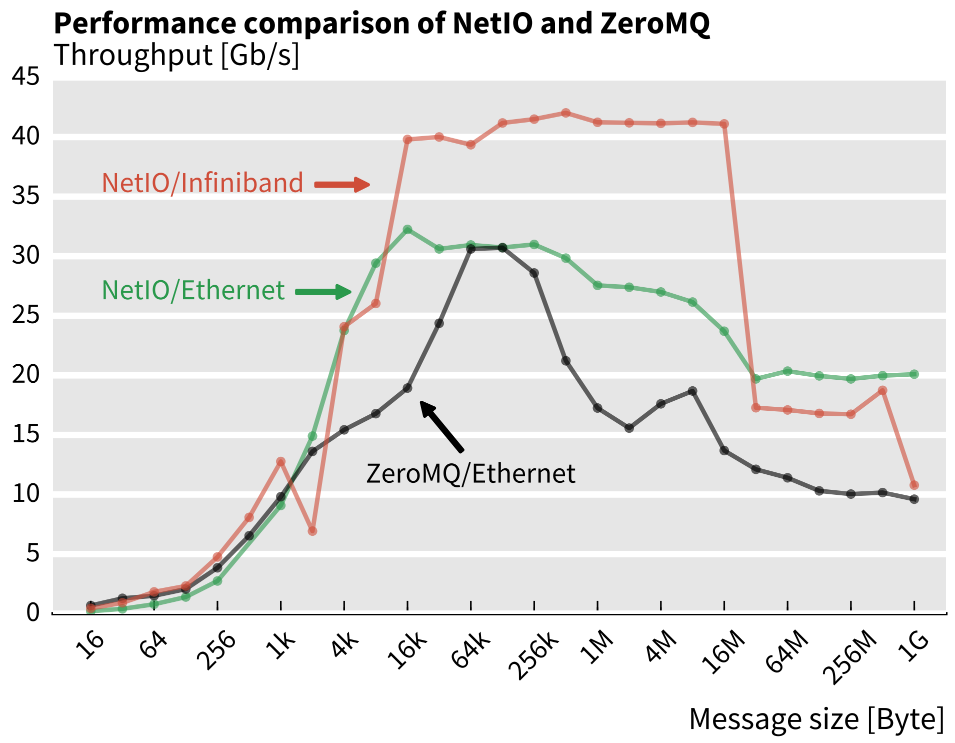


Figure 5.5: Throughput measured with NetIO and ZeroMQ on 40G Ethernet and NetIO on Infiniband FDR for various message sizes. Note that only a single connection is used between the two systems, hence the link is not fully utilized.

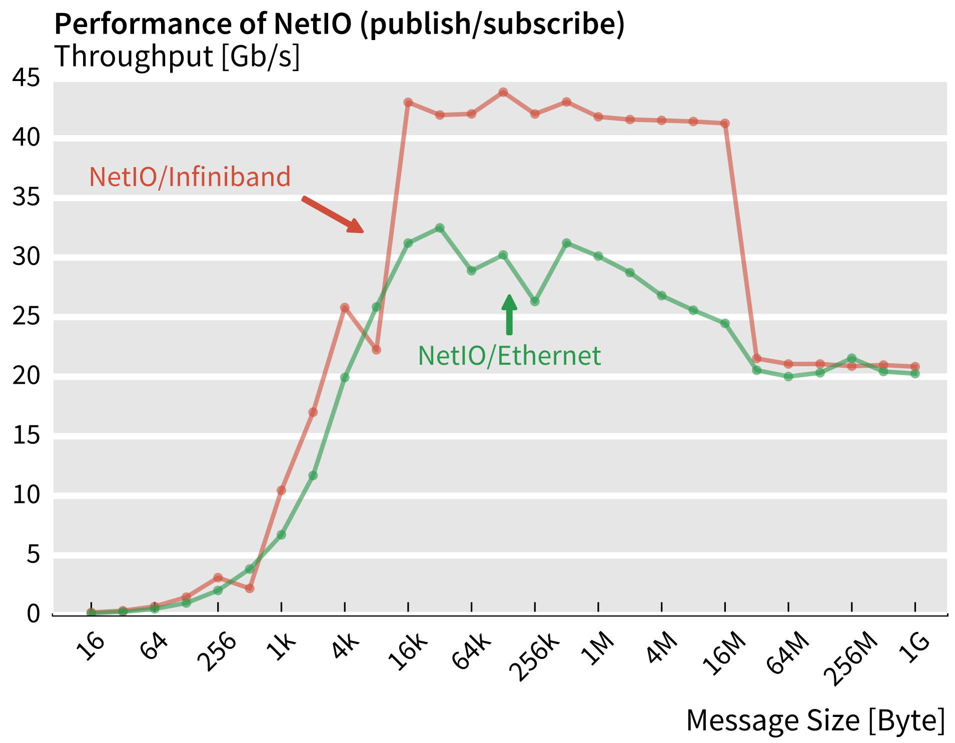


Figure 5.6: Throughput performance of NetIO publish/subscribe sockets on 40G Ethernet and Infiniband FDR. The peak performance of NetIO with the Infiniband back-end is more than 30% faster than NetIO with the Ethernet back-end.

values for small and large message sizes. For message sizes less than 1 kB NetIO an up to two-fold better throughput is measured than with ZeroMQ. NetIO on Infiniband outperforms both NetIO and ZeroMQ on Ethernet. The peak performance is around 40 Gb/s.

An experiment with NetIO high-throughput publish/subscribe sockets is shown in Figure 5.6. Similar to the previous benchmark NetIO on Infiniband outperforms NetIO on Ethernet. The achieved peak performance in each case is comparable to the point-to-point benchmarks.

A third benchmark analyzes the performance of NetIO low-latency sockets. We measure the round-trip time (RTT) between two systems in Table 5.3. In both cases, Ethernet and Infiniband, there is one switch in the middle. The results of the measurements can be seen in Figure 5.7.

NetIO on Ethernet, NetIO on Infiniband, and ZeroMQ show all very similar RTT values. The average RTT is in the case of Ethernet around $40\mu s$, for Infiniband it is just slightly higher. The difference for Infiniband indicates that the NetIO Infiniband back-end is slightly less efficient.

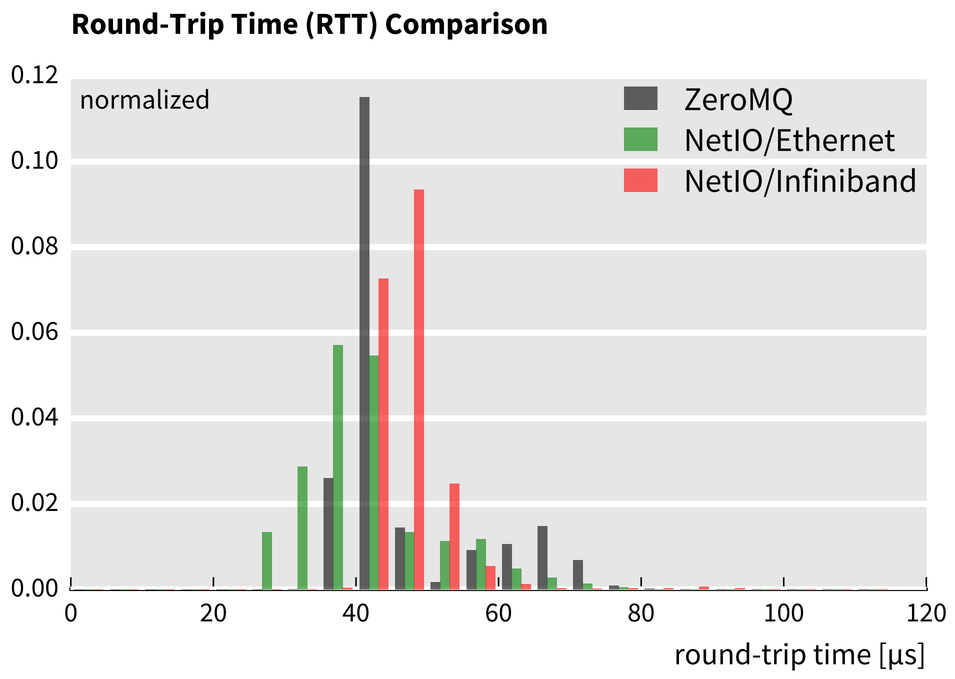


Figure 5.7: Round-trip time comparison between NetIO on Ethernet, NetIO on Infiniband, and ZeroMQ. All three implementations have a similar average value. NetIO on Infiniband has a slightly higher round-trip time than NetIO on Ethernet which can likely be explained by the different NetIO back-end implementation.

5.10 Related Work

This chapter was originally published as part of the conference proceedings for the International Conference on Computing in High-Energy Physics (CHEP) [Schumacher et al., 2016]. The benchmarks presented there were based on older versions of NetIO, network software stack, and network interface firmware.

The development teams at the other LHC experiments ALICE, CMS and LHCb are preparing upgrades for their respective DAQ systems as well. An import topic is the choice of DAQ network technologies. An overview of different 100 Gbps interconnect techologies with a focus on future DAQ applications is given in [39], which compares 100G Ethernet, Intel OmniPath and EDR Infiniband.

LHCb also investigates the potential use of Infiniband as network technology for their event builder network [40,41].

On the network side ALICE has been using a mixture of Ethernet and Infiniband in the past, and is investigating future network technologies [42].

The CMS experiment currently uses a mixture of Ethernet and Infiniband networks for their DAQ system [25, 43].

ZeroMQ [38] is a message queue implementation that has gained increasing popularity in high energy physics applications. The CERN Middleware Project is building a common middleware framework based on ZeroMQ [44, 45] that replaces old CORBA-based middleware. ALICE also considers to use ZeroMQ as a networking software framework.

A project similar to ZeroMQ is nanomsg [46]. A project in the ALICE experiment is evaluating the use of nanomsg and developing a OFI (OpenFabrics Interface) transport for nanomsg that allows to run nanomsg on Infiniband, OmniPath and other high performance interconnects [47].

Chapter 6

System Evaluation of a COTS-based Read-Out

6.1 Methodology

In the previous chapters I discussed and developed different aspects of a COTS-based read-out of high-energy physics experiments. This chapter provides an analysis of the results of the previous chapters and answers the question, "Is the COTS-based approach viable in realistic applications?" Evaluation is based on case studies of the FELIX system that were used as an example throughout the thesis. In the ATLAS Phase 1 upgrade in 2018 a first installation of FELIX systems will be deployed to be used for data-taking in LHC run 3. The case studies evaluate FELIX with respect to the requirements of this installation.

In run 3 FELIX PCs will connect to detector front-ends electronics of the New Small Wheel (NSW) muon detector [48], the LAr detector [49], Level-1 Calorimeter Trigger (L1Calo) systems, and the Tile detector. The NSW front-ends are divided into two distinct categories: small-strip Thin Gap Chambers (sTGCs) and Micromegas detectors (MMs) detectors. Both NSW front-end types use the GBT protocol in normal mode with forward error correction. All other systems use full-mode detector links (see Section 3.2 for an overview of the different detector link operation modes). Table 6.1 shows an overview of the anticipated installation size of 2019 including expected data volumes. The L1 trigger rate of ATLAS will be 100 kHz and thus the front-end electronics will transmit data packets at this rate to the FELIX systems. Additionally, a few FELIX systems connecting to special Liquid Argon trigger electronics will be deployed. Since these FELIX systems will be exclusively used for TTC and DCS traffic and do not carry event data streams they are not relevant for this evaluation.

The traffic patterns of the NSW detector and the full mode detectors are very different. The NSW data packets are typically less than 50 byte and very

	No. of Links	Type of Link	Data E-Links per Link	Framerate [kHz]	Avg. Framesize [Byte]	Avg. Throughput [MB/s]
NSW sTGC	512	GBT (normal)	3	100	38	11.4
NSW MM	512	GBT (normal)	8	100	22	18
LAr	31	Full Mode	1	100	3900	390
L1Calo	40	Full Mode	1	100	4800	480
Tile	4	Full Mode	1	100	2500	250

Table 6.1: The different sub-systems connecting to FELIX PCs as expected for the ATLAS Phase 1 upgrade in 2018.

small compared to full mode packets with multiple kilobyte. On the other hand, the NSW uses the E-link feature of the GBT protocol and thus has more virtual input channels per FLX card. A full mode FELIX with 24 input links receives data packets at the 2.4 MHz rate. In the NSW MM case a FELIX with the same number of input links and eight E-links per physical link would receive data packets at rate of 19.2 MHz. Large packet rates are CPU intensive as more PCIe packets have to be decoded (Chapter 4, also see Section 6.5). Furthermore, network links operate less efficiently with smaller packets. Although, the bandwidth requirements are relatively low. For the full mode the situation is reversed, i.e., because of the lower packet rate there is less pressure on the CPU, but since the packet sizes are much larger the network link bandwidth becomes a bottleneck.

6.2 Case Study: The New Small Wheel

The NSW is a new detector to be used in LHC run 3 and onwards. It replaces the current Small Wheel (SW), a muon detector located on the end-caps of the ATLAS experiment. The NSW is designed for the more demanding requirements of LHC Phase 2 and is able to cope with higher background radiation providing track reconstruction with higher resolution.

The NSW front-ends use the GBT protocol in normal mode to communicate with FELIX devices. Per GBT link eight (MM) or three (sTGC) E-links are dedicated to carry event data traffic. More E-links are used for control and monitoring traffic with chips on the NSW electronics.

To test the NSW use case, I used a FELIX prototype server with a FLX-710 GBT interface card. The internal data generator of the FLX firmware was used to generate test data according to the NSW specifications. At this point the FLX-710 firmware only had support for up to four GBT links. However, by increasing the number of E-links per GBT link accordingly I was able to compensate and emulate input from 24 GBT links. This compensation is transparent to the FELIX software. Both cases, sTGC and MM were tested. The generated data was read out by the FELIX application and sent over a network to a second server. The second server ran a benchmark application that receives data and measures the achieved throughput. Both servers are connected via a 40G Ethernet network with a single switch. The benchmark system specifications are defined in Chapter 5 in Table 5.3 (the FELIX application is running on system 2).

The FELIX application used 10 worker threads to match the number of cores per CPU. The application was limited to only run on the first of two available CPUs. Both the FLX card as well as the network interface of the test system are connected to the PCIe controller in this CPU. When allowing the scheduler of the operating system to use both CPUs, the NUMA architecture of the system would lead to a severe performance degradation because memory would be transferred back and forth between the memory nodes. Simply pinning the worker threads to CPU cores to prevent movement of threads among NUMA nodes would not help. Memory still would need to be transferred as the FLX card and the network interface card are physically attached to one of the NUMA nodes. In a production system a second FLX card and network interface could be installed and connected to the PCIe controller of the second CPU. In this case a second FELIX application could be run on the second CPU and thus double the overall link density of the FELIX server.

For technical reasons the data emulator on the FLX card operates at a fixed bandwidth. Thus, the rate of generated chunks (Chapter 4) depends on their size. The desired target rate of 100 kHz is reached with a chunksize of approximately 72 byte and smaller chunksizes will yield a higher rate. This needs to be considered when interpreting the benchmark results. In the case of sTGC this effect can be ignored as the benchmark systems are able to handle the additional load. However, for the more demanding MM use case the rate per simulated E-link with 22 byte chunk size is about 250 MHz and too high for the FELIX application to handle. The benchmark needs to be slightly adjusted to compenstate for the higher rate.

The results of the benchmarks are shown in Figure 6.1 and Table 6.2. For the sTGC case in total 72 E-links which carry 38 byte chunks are emulated. The average chunk rate is about 12 MHz, which translates to an average chunk rate of about 166 kHz per E-link, which is more than the required 100 kHz. This confirms that the FELIX application can handle the specified

	sTGC	**MM**
Nominal number of E-links	72	192
Nominal chunk rate per E-link [kHz]	100	100
Nominal total chunk rate [MHz]	7.2	19.2
Emulated number of E-links	72	48
Emulated chunk rate per E-link [kHz]	166	250
Emulated total chunk rate [MHz]	12	16
Equivalent number of 100 kHz E-links	120	160

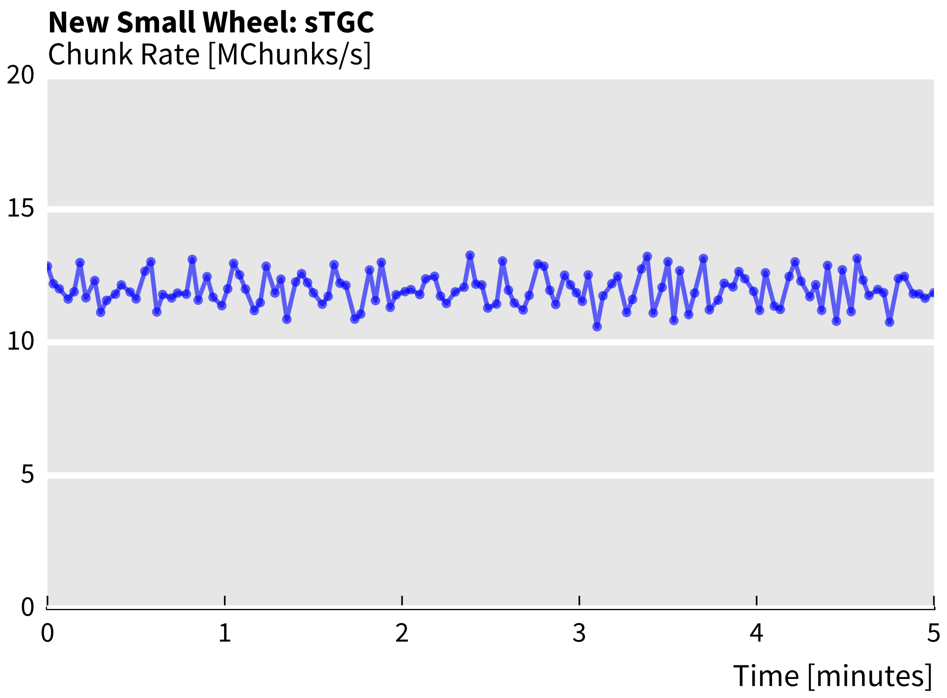
Table 6.2: Nominal operational parameters and emulated equivalents with compensation for the high emulated chunk rate per E-link. In the sTGC case 120 E-linkes can be emulated and processed, exceeding the nominal number of 72 E-links. In the MM case only 160 E-links can be emulated and processed, slightly lower than the nominal number of 192.

24 input GBT links on a single CPU of the benchmark system.

In the MM scenario I lowered the number of emulated E-links to 48 (each carrying 22 byte chunks) in order not to overload the FELIX application. The total chunk rate that the system is processing is 16 MHz, approximately the maximum that the benchmark system can handle on a single CPU. This translates to 250 MHz chunk rate per emulated E-Link. This number can be projected on to the equivalent scenario with 100 kHz E-links: the measured rate of 16 MHz is equivalent to 160 of 100 kHz E-links or 20 GBT links with the NSW MM workload (8 E-links per GBT link). This implies that the benchmark system in use performs sligthly below the specification. It is acceptable to operate the NSW FELIX systems with only 20 GBT links per card instead of 24. However, the CPUs of the benchmark PC are of the Intel Haswell generation from 2014. More recent and powerful CPUs are expected to be used for the FELIX deployment in 2018 and it is likely that the full specification of 24 links is supported on this hardware. The effect of the CPU on FELIX performance is further discussed in Section 6.5.

6.3 Case Study: Full Mode

The front-ends of LAr, L1Calo, and the Tile calorimeter all use full-mode detector links. Since full-mode links do not support E-links, naturally each detector link transports only one channel. The expected data frame sizes for event data traffic range between 2500 byte (Tile) and 4800 byte (L1Calo). Full-mode FELIX systems cannot be operated with the theoretical maximum of 24 input detector links. For example, in the case of L1Calo, the bandwidth re-



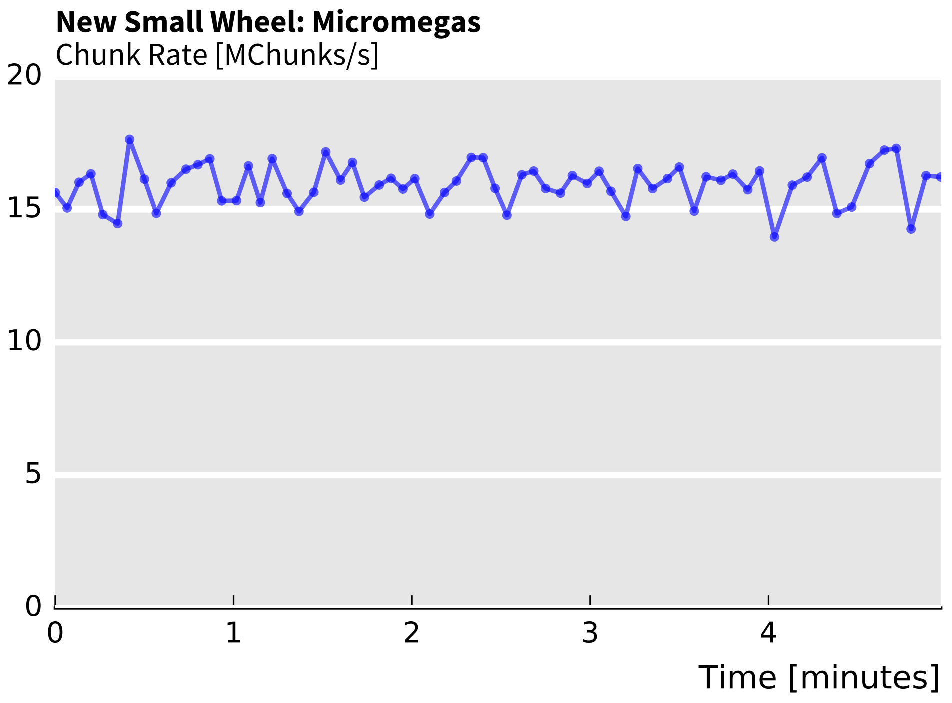


Figure 6.1: Chunk rates for a FELIX system with emulated New Small Wheel data.

quirements would sum up to more than 90 Gb/s. This exceeds the available bandwidth of a PCIe Gen-3 x8 device (around 60 Gb/s), and also both 40G Ethernet and FDR Infiniband (56 Gb/s). In theory, a full-mode FELIX system with 24 links would be possible with PCIe Gen-4 x8 or PCIe Gen-3 x16 and 100G Ethernet or EDR Infiniband (100 Gb/s). Although for cost and availability reasons it is more practical to reduce the link density and operate the full-mode FELIX PCs with less than 24 detector links. This may be subject to change due to evolution of PC hardware in the near future.

At the time of this writing no firmware for the FLX-709 or FLX-710 was available that included an emulator for full-mode links and the FLX-711 was not yet manufactured. To study the full-mode use case I instead opted to use input from pre-generated files in the FELIX application. The hardware setup of the two test systems is the same as in the NSW use case. The FLX-711 card uses 16 PCIe Gen-3 lanes, but instead of exposing a single x16 interface to the operating system, the FLX-711 exposes two PCIe Gen-3 x8 interfaces to the operating system. This allows the software to read out the card in parallel in two threads. In the benchmarks this is emulated by starting two separate FELIX applications, each receiving half of the input. As in the NSW case study the FELIX applications are both pinned to only one of the two CPUs. Each of the two applications starts three worker threads apart from the card reader thread and the NetIO event loop threads such that the 20 hardware threads available on one CPU is not exceeded.

The data throughput on the network interface is shown Figure 6.2. In each of the three scenarios a peak network throughput of around 32 Gb/s is measured which corresponds to a 75 % link utilization. Based on this value, the maximum amount of full-mode links that can be read out on a single CPU is 16 (Tile), 10 (LAr), and 8 (L1Calo).

In the FELIX application only a single thread is reading out data from the card by copying the blocks and assigning them to worker threads. In the full mode scenarios the chunk sizes are much bigger and thus more data and consequently more blocks are transferred over the PCIe bus. As a result, the single thread handling input from the card becomes the bottleneck of the application. This is not an issue for the NSW use cases because much less blocks are transferred over the PCIe bus due to the much smaller chunk sizes. By exposing two PCIe interfaces this bottleneck is partially addressed by the FLX-711. Introducing more parallelism can further increase the throughput of the FELIX application. This however requires support from the FLX firmware which is currently not implemented. A faster CPU in the final FELIX system is also expected to increase the throughput further.

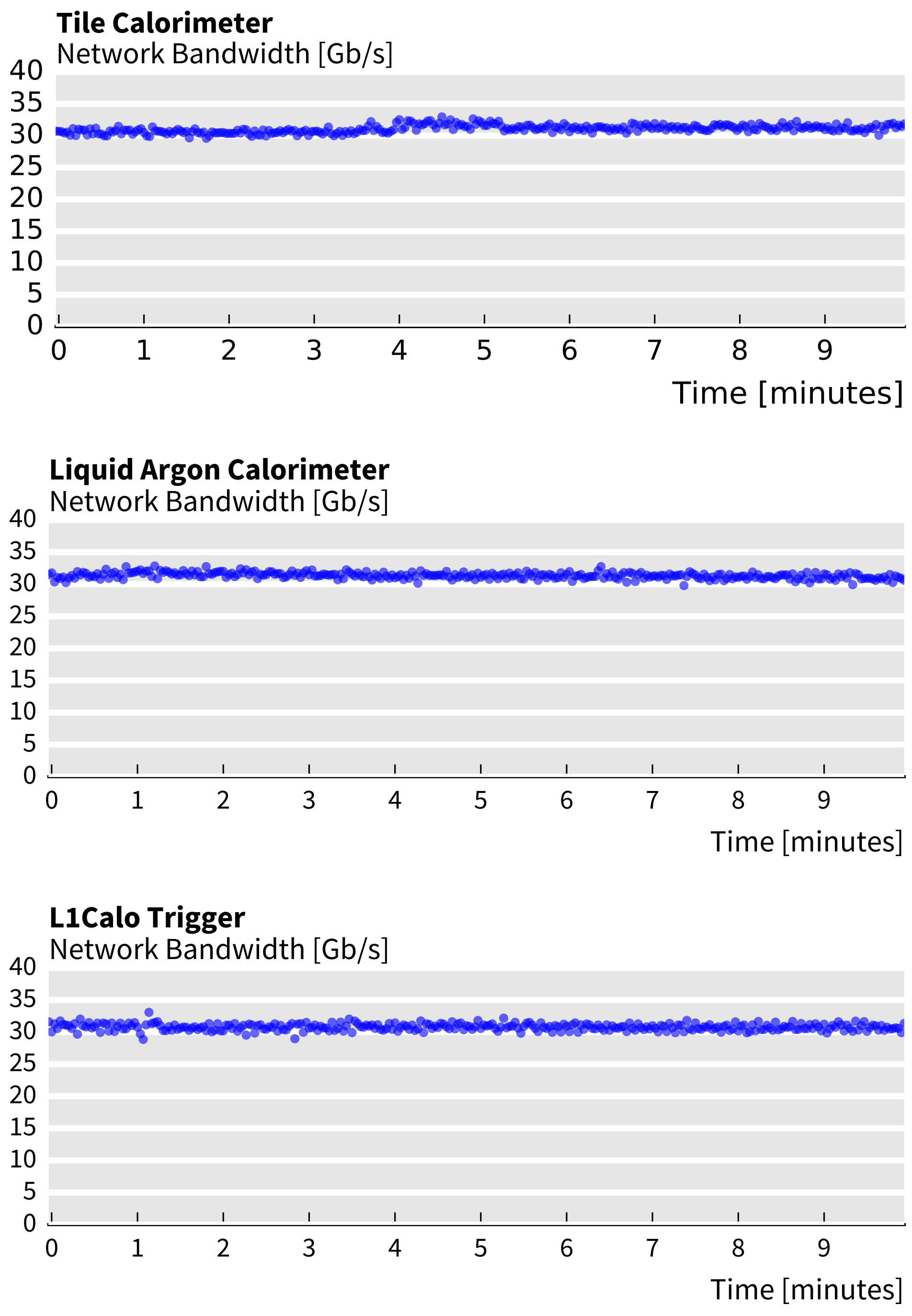


Figure 6.2: Bandwidth measurements of the three different full-mode applications. In each case the maximum network bandwidth that can be utilized is about 32 Gb/s.

6.4 Scalability

The experiments mentioned above are all exclusively performed in a lab environment with two PC servers. It is important to ensure the scalability of a COTS-based readout approach like FELIX given the large amounts of data produced in high-energy physics experiments.

Therefore, I performed scalability experiments on a 16-node Infiniband cluster.[1] Some of the nodes simulated FELIX servers and generated data and some of the nodes simulated readout servers that received data from the FELIX systems. I considered two scenarios: (i) a one-to-one mapping between FELIX servers and readout servers where each readout server receives data from exactly one FELIX system and (ii) a two-to-one mapping where each readout server receives data from exactly two FELIX systems. Consequently eight of the 16 nodes were used to simulate FELIX systems and either four or eight server were used to simulate the readout servers (Figure 6.3). The cluster nodes were connected via an Infiniband QDR network with a peak bandwidth of 32 Gb/s and a leaf-spine topology. There were two spine switches and thus there were two redundant paths between each pair of nodes (excluding node pairs connected to the same leaf switch). Table 6.3 shows the hardware configuration of the cluster nodes.

The nodes simulating FELIX servers generated traffic on multiple connections. Over each connection constant-size messages were transmitted at the rate of 100 kHz. NetIO high-throughput sockets were used for communication. The receiving application processed the incoming data and measured

[1]I would like to thank the University of Castilla-La Mancha in Albacete, Spain, for granting me access to their cluster to perform these measurements, and especially Pedro, Jesús and Raúl for the productive collaboration.

Infiniband Cluster Node Configuration	
CPU Type	Intel Xeon E5-2630L v3
Number of CPUs	1
CPU Frequency	1.80 GHz
CPU Cores (real)	8
CPU Cores (hardware threads)	16
Memory	32 GB
Network Interface	Mellanox ConnectX-3
Network Type	QDR Infiniband

Table 6.3: Hardware configuration of the 16-node Infiniband cluster that was used for the scalability tests.

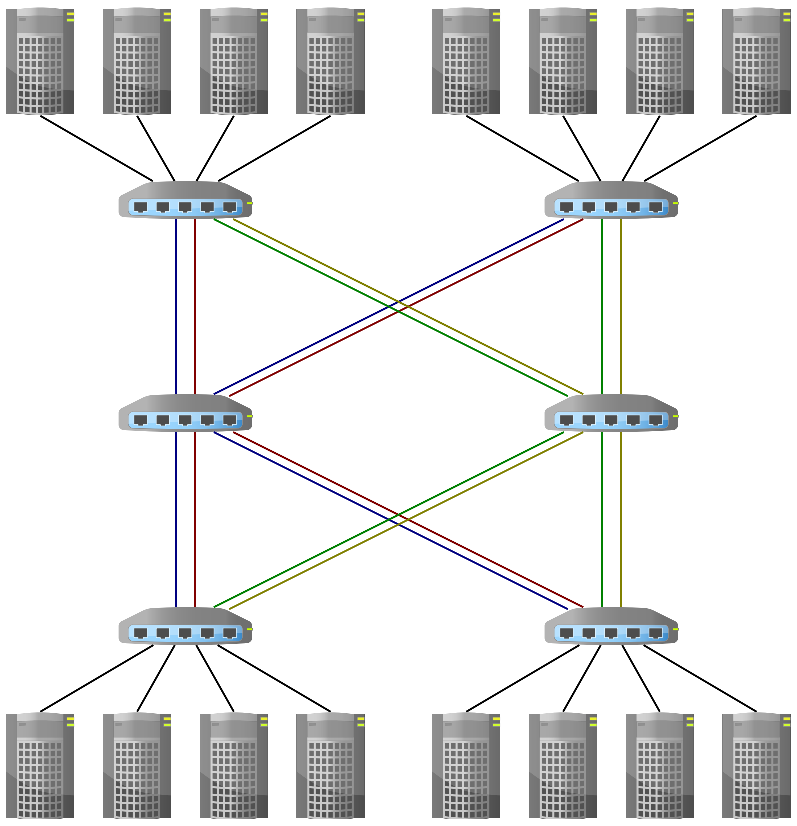
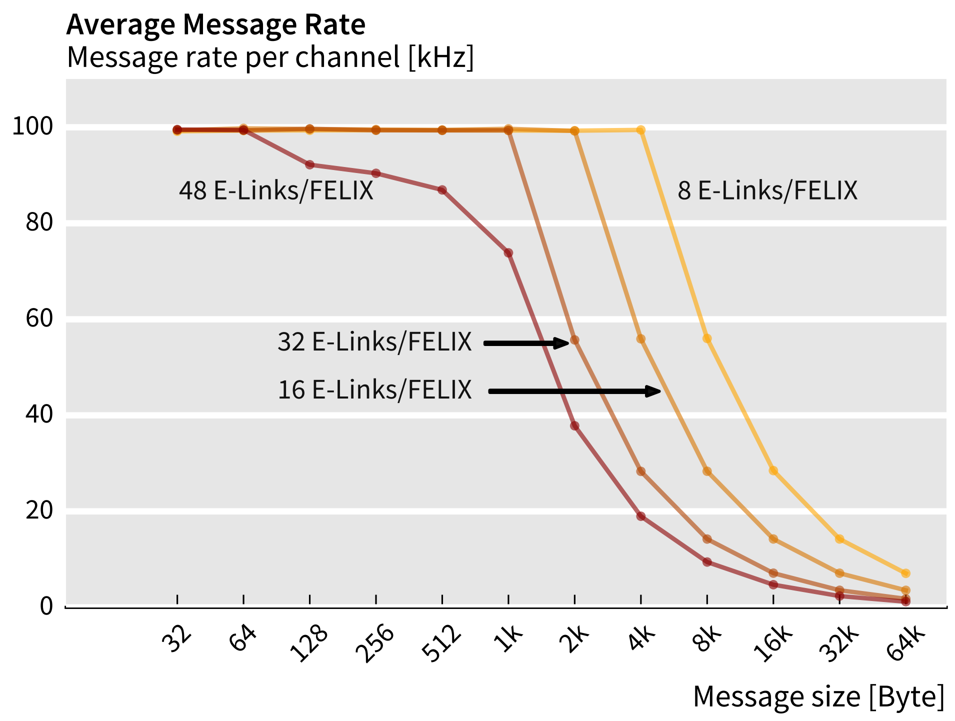


Figure 6.3: Network topology of the 16-node Infiniband QDR cluster. Eight of the nodes are used to simulate FELIX traffic, and four or eight nodes are used as corresponding readout servers.

the throughput and rate of arriving messages. In this benchmark I assumed only event data traffic as the low-throughput control and monitoring traffic is negligible.

For the one-to-one test scenario the results are shown in Figure 6.4 and the respective results for the two-to-one scenario are shown in Figure 6.5. The results show that there are two main performance ceilings in the system respectively for small and large fragments. Upon further analysis, it became clear that for small message sizes up to 1 kB the maximum amount of E-links that can be processed is limited by the CPU of the receiving side, the readout server. The network bandwidth only becomes a bottleneck for larger message sizes above 1–4 kB. In the one-to-one mapping scenario up to 32 E-links can be simulated per FELIX server to be able to process the data at a 100 kHz rate for most message sizes less than 1 kB. In the two-to-one scenario only 16 E-links can be simulated per FELIX server. This 2:1 ratio is expected since data from twice as many FELIX servers are received on each readout server.



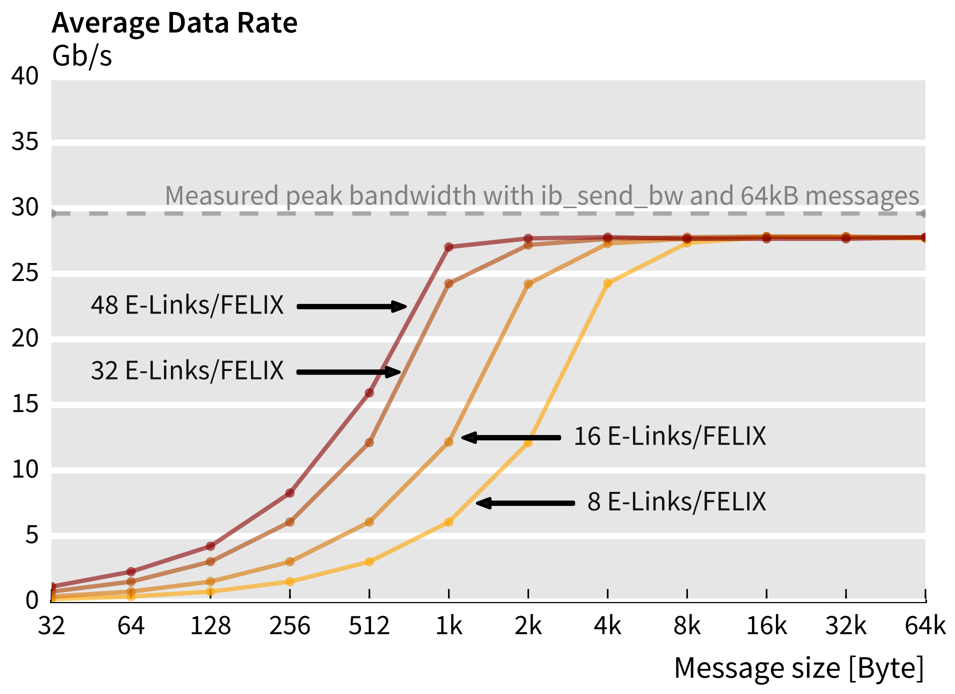
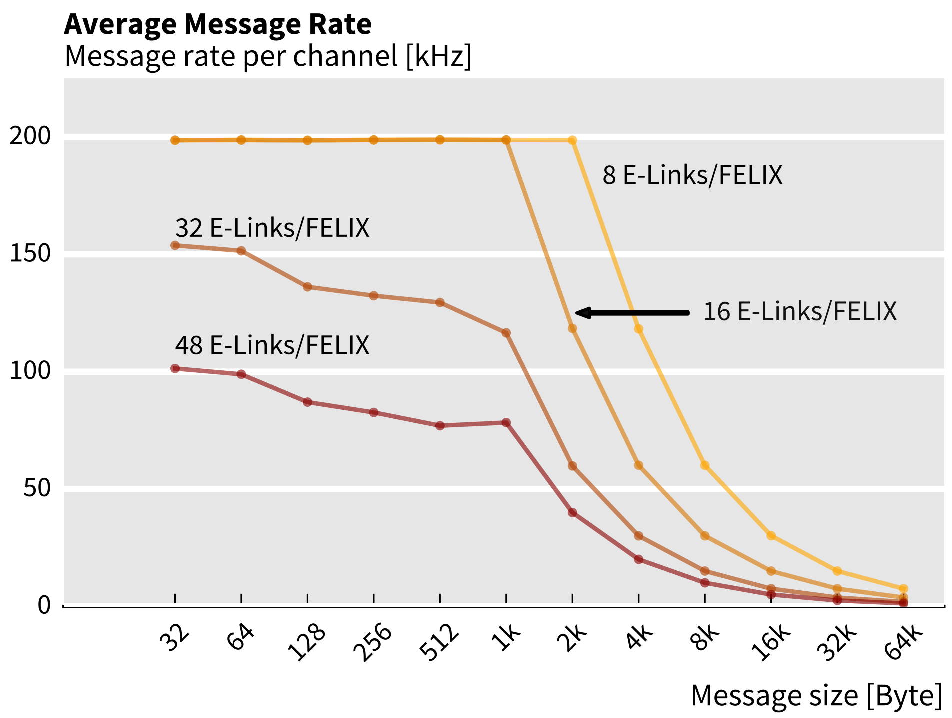


Figure 6.4: Measured data and message rates on the cluster with a one-to-one mapping between the FELIX and readout servers.



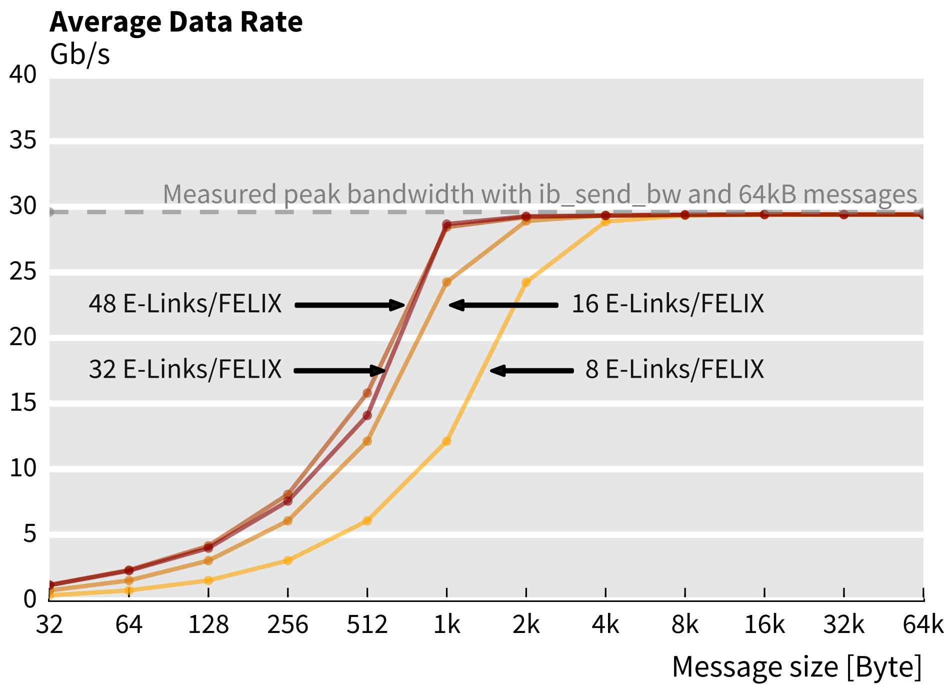


Figure 6.5: Measured data and message rates on the cluster with a two-to-one mapping between the FELIX and readout servers.

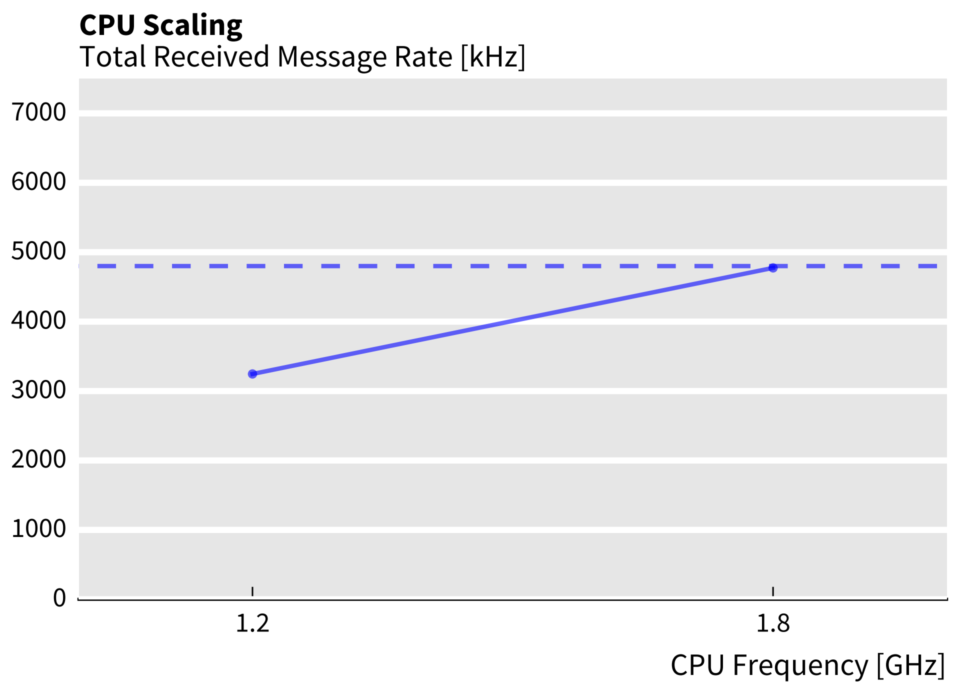


Figure 6.6: Scaling the CPU frequency of the cluster nodes. Shown is the total message rate that is processed at a single readout server when simulating 48 E-links. The dashed line represents the nominal processing rate. Only at 1.8 GHz the readout server is able to process enough messages to handle the input traffic.

The plots in Figures 6.4 and 6.5 show results of one of the readout servers, but the measured data is representative. The full bandwidth of the links can be used and the overall network performance does not degrade when scaling to multiple FELIX servers. The chosen leaf-spine topology supports the anticipated FELIX workload. As assumed earlier in this chapter, the results suggest that the performance of full mode FELIX is limited by the available network bandwidth while GBT-mode FELIX PCs are limited by the CPU.

6.5 CPU Scaling

To explore the effect of the CPU on the overall performance I experimented with scaling of the CPU frequency in the cluster (Figure 6.6). In the measurement I assume 32 Byte message size and the one-to-one mapping scenario. The received total message rate on a single readout server with the default CPU clock frequency of 1.8 GHz is 4.8 MHz when simulating 48 E-links per FELIX. In other words, at 1.8 GHz the readout server is able to sustain the

100 kHz processing rate per E-link for 48 E-links. When scaling the CPU frequency down to 1.2 GHz the rate at the readout server drops and the system is not able to process the full input from 48 E-links anymore. The experiment highlights that there is a clear dependence between the number of supported E-links and the CPU frequency, although with only two data points it is not possible to model the dependence.

In conclusion, a COTS-based detector readout approach with a system like FELIX is able to scale to an installation with multiple FELIX servers and data handler servers. Leaf-spine is a viable network topology. The choice of CPU directly affects the link density of a COTS-based readout system.

Chapter 7

Conclusion

7.1 Summary

In this thesis I explored data-aquisition systems that are primarily based on commercial-off-the-shelf components for large high-energy physics experiments. DAQ systems traditionally used many custom designed electronic components due to environmental factors like radiation or performance requirements like high data rates and real-time constraints. While I specifically analyzed the LHC ATLAS experiment at CERN, the results can be applied to other HEP experiments as well.

In the first part of the thesis I described the current ATLAS DAQ system and analyzed the load distribution on the readout system. Subsequently, I introduced a new COTS-based DAQ architecture for ATLAS. The key component in this architecture is FELIX, a new PC-based device that connects ATLAS detector front-end electronics with the readout system, detector control system and detector-specific control and calibration systems. I discussed the processing of PCIe traffic packets which was optimized to meet the performance requirements of FELIX and the network interface of FELIX which is provided by the software library NetIO. Furthermore, I performed an evaluation of the FELIX system and showed that the approach is viable to cope with the requirements of the first deployment of FELIX systems in LHC run 3.

7.2 COTS-based Readout for HEP experiments

The main research question in this dissertation is whether a COTS-based readout of HEP experiments is viable and if yes, how can it be implemented. This question was answered. With the example of FELIX it was shown that at a readout of the ATLAS detector as planned for LHC run 3 is possible. The interest for FELIX in other experiments (see Section 7.4) and the fact that other experiments implement similar techniques (see Section 3.5) shows that the COTS approach has a broader applicability than just ATLAS.

Of course, the ATLAS experiment will still include many custom electronic components in run 3 outside of the DAQ system (e.g. the detector front-ends and the level-1 trigger system). For these systems the cost and development effort would not justify the advantages of a COTS-based approach. Additionally, the environmental factors like radiation are a problem for the front-end electronics. Since these systems handle the raw 40 MHz input of the detectors the data processing volume would be too high. Furthermore, these systems have strong real-time constraints that is easier to achieve with custom electronics.

Also, the FLX card in the FELIX system itself will be a custom design. Commercial alternatives have been tested, but no card fullfilled all requirements. Specifically no commercial card was found that would allow implementing the TTC interface. With a mezzanine card one could implement a TTC interface externally as it has been done for internal development with the FLX-709 and FLX-710 (Section 3.3). Such a design would exceed the physical dimensions specified by the PCIe standard, thus the FELIX team opted for the custom FLX-711 design instead.

Not all subsystems of the ATLAS experiment will initially use FELIX. Only NSW, LAr, L1Calo and Tile will be used. The reason for this is pragmatic. Since no major upgrades are done on the other subsystems there is no reason to switch to a new readout model. The requirements of the other detectors are not much different from the detectors for which the FELIX readout was tested in Chapter 6.

7.3 FELIX

Several design decisions concerning FELIX have been proven to be effective. The publish/subscribe system as interface to the DAQ network meets the heterogeneous requirements of different data handlers and leads to a clean separation of the different architecture layers. Also, differentiating between different traffic patterns (high-throughput and low-latency) in a single, consistent API is helpful to integrate different classes of systems. The support for multiple network technologies allows delaying a technology decision to later phases of the FELIX development. The decision can be based on factors like cost, maintenance effort, performance, availability of suitable hardware etc., without a technology or vendor lock-in. This is a clear advantage of the COTS approach.

Some implementation aspects of FELIX can be improved in the future. For example, FELIX does not perform any data coalescing. That means that in some cases like the New Small Wheel data is sent out in very small data packets of less than 100 Byte. Processing such small packets at a high rate is much less efficient than processing bigger packets at a lower rate (see Chapter 6). The FELIX system could be optimized by coalescing packets in the

FLX card, thus reducing the output rate of FELIX while increasing the average message size. The coalescing could be implemented by merging packets from multiple E-links that are associated with the same collision event. This type of coalescing is performed anyway later in the event building stage, but doing so earlier in the DAQ chain would free up CPU resources in the rest of the DAQ system.

Furthermore, Chapter 6 has shown that reading out the FLX card in only a single thread per PCIe interface is a performance bottleneck. By adjusting the FLX firmware to allow readout on multiple DMA channels in parallel this bottleneck could be removed.

7.4 Outlook

Next steps in the FELIX project are the finalization of the hardware platform, including the procurement of server PCs, the manufacturing of the FLX cards, the choice of network technology, and the completion of the software and firmware development. In the Long Shutdown 2 (2019-2020) the FELIX system will be deployed for part of the ATLAS subsystems. The Large Hadron Collider (LHC) is scheduled to restart for run 3 in 2021, the first run including FELIX systems.

The LHC Phase 2 is scheduled to start in 2026 with run 4 and has significantly increased requirements on the experiments. The FELIX approach will be used for all ATLAS subsystems. The FELIX platform will be upgraded to cope with the new requirements: more input links, higher trigger rates, and larger data volumes.

Furthermore, an effort has started to use FELIX outside of ATLAS. The Deep Underground Neutrino Experiment (DUNE) is a large-scale neutrino experiment at Fermilab in the USA. Development is currently in progress and first beams are expected in 2026. A small-scale prototype of one of the DUNE detectors is developed at CERN. The prototype is intended to be used for research and development. Developing dedicated electronic components for the readout of the prototype would be costly and is therefore not practicable for the prototype. Instead, it is planned to base the readout system on FELIX devices. This demonstrates the versatility of a COTS-based readout approach: the use of standardized readout elements reduces significantly the cost and time of building a DAQ system.

In Chapter 5 I discussed the usage in DAQ systems of network technologies like Infiniband that are traditionally only found in HPC applications. It could be interesting to study the use of HPC technology in datacenter applications. The library NetIO that was presented in this dissertation could be expanded and tailored to more applications than DAQ systems. NetIO is planned to be released under an open source license later in 2017. The differentiation of traffic types that was introduced in NetIO could also be interesting for other applications. One could imagine the backend of a video streaming website, where user requests are answered from a database with a low-latency connection, while video files are retrieved from a storage server via a high-throughput connection.

7.5 Final Words

HEP has proven to be an interesting field of study from a computer science point of view. DAQ systems are considerably different from other large-scale computing systems like HPC clusters or commercial datacenters and have unique requirements and challenges. HEP experiments on the other hand can benefit from applying computer science principles not only to DAQ systems, but also to other computing tasks like simulation or offline data processing. Collaboration in this area is likely to be profitable for both sides.

Publications of the Author

[Borga et al., 2016] Borga, A., Anderson, J., Boterenbrood, H., Chen, H., Chen, K., Drake, G., Dönszelmann, M., Francis, D., Gorini, B., Lanni, F., Miotto, G. L., Levinson, L., Narevicius, J., Roich, a., Ryu, S., Schreuder, F., Schumacher, J., Vandelli, W., Vermeulen, J., Wu, W., and Zhang, J. (2016). A new approach to front-end electronics interfacing in the ATLAS experiment. *Journal of Instrumentation*, 11(01):C01055–C01055.

[Borga et al., 2014] Borga, A., Crone, G. J., Green, B., Kugel, A., Joos, M., Panduro Vazquez, J. G., Schumacher, J., Teixeira-Dias, P., Tremblet, L., Vandelli, W., Vermeulen, J. C., Werner, P., and Wickens, F. J. (2014). Evolution of the ReadOut System of the ATLAS experiment. *Technology and Instrumentation in Particle Physics*.

[Chen et al., 2016] Chen, K., Anderson, J., Bauer, K., Borga, A., Boterenbrood, H., Chen, H., Drake, G., Dönszelmann, M., Francis, D., Guest, D., Gorini, B., Joos, M., Lanni, F., Miotto, G. L., Levinson, L., Narevicius, J., Vazquez, W. P., Roich, A., Ryu, S., Schreuder, F., Schumacher, J., Vandelli, W., Vermeulen, J., Whiteson, D., Wu, W., and Zhang, J. (2016). Felix: a pcie based high-throughput approach for interfacing front-end and trigger electronics in the atlas upgrade framework. *Journal of Instrumentation*, 11(12):C12023.

[Narevicius et al., 2016] Narevicius, J., Anderson, J., Borga, A., Boterenbrood, H., Chen, H., Chen, K., Drake, G., Donszelmann, M., Francis, D., Gorini, B., Guest, D., Lanni, F., Miotto, G. L., Levinson, L., Roich, A., Ryu, S., Schreuder, F., Schumacher, J., Vandelli, W., Vermeulen, J., Wu, W., and Zhang, J. (2016). Felix: The new approach for interfacing to front-end electronics for the atlas experiment. In *2016 IEEE-NPSS Real Time Conference (RT)*, pages 1–2.

[Schumacher et al., 2015a] Schumacher, J., Anderson, J. T., Borga, A., Boterenbrood, H., Chen, H., Chen, K., Drake, G., Francis, D., Gorini, B., Lanni, F., Lehmann Miotto, G., Levinson, L., Narevicius, J., Plessl, C., Roich, A., Ryu, S., Schreuder, F. P., Vandelli, W., Vermeulen, J., and Zhang, J. (2015a). FELIX: a High-Throughput Network Approach for Interfacing to Front End Electronics for ATLAS Upgrades. *Journal of Physics: Conference Series*, 664(8):082050.

[Schumacher et al., 2015b] Schumacher, J., Anderson, J. T., Borga, A., Boterenbrood, H., Chen, K., Chen, H., Drake, G., Francis, D., Gorini, B., Lanni, F., Lehmann Miotto, G., Levinson, L., Narevicius, J., Roich, A., Ryu, S., Schreuder, F. P., Vandelli, W., Vermeulen, J., and Zhang, J. (2015b). Improving packet processing performance in the ATLAS FELIX project. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems - DEBS '15*, pages 174–180.

[Schumacher et al., 2016] Schumacher, J., Plessl, C., and Vandelli, W. (2016). High-Throughput Network Communication with NetIO. submitted.

Bibliography

[1] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *Journal of Instrumentation*, vol. 3, no. 1-4, p. S08003, 2008.

[2] L. Evans, "The Large Hadron Collider," *New Journal of Physics*, vol. 9, no. 9, p. 335, 2007.

[3] D. H. Perkins, *Introduction to high energy physics*. Cambridge University Press, 1972.

[4] O. S. Brüning, P. Collier *et al.*, *LHC Design Report*. Geneva: CERN, 2004, vol. 2. [Online]. Available: https://cds.cern.ch/record/815187

[5] T. Kawamoto, S. Vlachos *et al.*, "New Small Wheel Technical Design Report," CERN, Tech. Rep. CERN-LHCC-2013-006, June 2013. [Online]. Available: https://cds.cern.ch/record/1552862

[6] M. Shochet, L. Tompkins *et al.*, "Fast TracKer (FTK) Technical Design Report," CERN, Tech. Rep. CERN-LHCC-2013-007, June 2013. [Online]. Available: https://cds.cern.ch/record/1552953

[7] The ATLAS TDAQ Collaboration, "The ATLAS Data Acquisition and Higher Level Trigger system," *Jinst*, vol. 11, no. to be published, 2016.

[8] W. D. Peterson, *The VMEbus handbook; 4th ed.* Scottsdale, AZ: VITA, 1997.

[9] A. Kugel, "The ATLAS ROBIN – A High-Performance Data-Acquisition Module," Ph.D. dissertation, Mannheim University, Mannheim, 2009, presented on 19 Aug 2009. [Online]. Available: http://cds.cern.ch/record/1209243

[10] H. C. van der Bij, R. A. McLaren *et al.*, "S-LINK, a Data Link Interface Specification for the LHC Era," in *Nuclear Science Symposium, 1996. Conference Record., 1996 IEEE*, 1996, pp. 465–469.

[11] B. Gorini, M. Joos *et al.*, "A RobIn Prototype for a PCI-Bus based Atlas Readout-System," in *Proceedings of the Nineth Workshop on Electronics for LHC Experiments : Amsterdam, Netherlands, 29.09.2003*, vol. 03-006. Geneva: CERN, 2003, pp. 152–156.

[12] The RD-12 Collaboration. The TTC System. [Online]. Available: http://ttc.web.cern.ch/TTC/intro.html

[13] S. Haas, M. Joos, and W. Iwanski, "Design and Performance of a PCI Interface with four 2 Gbit/s Serial Optical Links," CERN, Tech. Rep. CERN-ATL-COM-DAQ-2004-018, 2004.

[14] G. L. Presti, O. Barring *et al.*, "Castor: A distributed storage resource facility for high performance data processing at cern." in *MSST*, vol. 7. Citeseer, 2007, pp. 275–280.

[15] R. Jones, S. Kolos *et al.*, "Applications of CORBA in the ATLAS prototype DAQ," in *Real Time Conference, 1999. Santa Fe 1999. 11th IEEE NPSS*, 1999, pp. 469–474.

[16] W. P. Vazquez, "The ATLAS Data Acquisition System: from Run 1 to Run 2," *Nuclear and Particle Physics Proceedings*, vol. 273–275, pp. 939 – 944, 2016, 37th International Conference on High Energy Physics (ICHEP).

[17] F. Vasey, D. Hall *et al.*, "The Versatile Link common project: feasibility report," *Jinst*, vol. 7, no. 01, p. C01075, 2012.

[18] P. Moreira, A. Marchioro, and K. Kloukinas, "The GBT: A proposed architecture for multi-Gb/s data transmission in high energy physics," *Published in Prague*, 2007.

[19] P. Moreira, R. Ballabriga *et al.*, "The GBT Project," *Topical Workshop on Electronics for Particle Physics*, pp. 342–346, 2009.

[20] S. Kama, "Evolution of the trigger and data acquisition system in the ATLAS experiment," *IEEE Nuclear Science Symposium Conference Record*, vol. 396, no. 1, pp. 1787–1790, 2012.

[21] C. Haeberli, A. dos Anjos *et al.*, "ATLAS TDAQ DataCollection software," *Ieee Transactions on Nuclear Science*, vol. 51, no. 3, pp. 585–590, 2004.

[22] M. Bellato, G. Collazuol *et al.*, "A PCIe Gen3 based readout for the LHCb upgrade," *Journal of Physics: Conference Series*, vol. 513, no. 1, p. 012023, 2014.

[23] F. Réthoré, J. P. Cachemiche *et al.*, "The PCIe-based readout system for the LHCb experiment," in *Topical Workshop on Electronics for Particle Physics*, 2015.

[24] M. B. Marin, A. Boccardi *et al.*, "The Giga Bit Transceiver based Expandable Front-End (GEFE)—a new radiation tolerant acquisition system for beam instrumentation," *Journal of Instrumentation*, vol. 11, no. 02, pp. C02 062–C02 062, 2016.

[25] G. Bauer, T. Bawej *et al.*, "The new CMS DAQ system for LHC operation after 2014 (DAQ2)," *Journal of Physics: Conference Series*, vol. 513, no. TRACK 1, 2014.

[26] T. Bawej, U. Behrens *et al.*, "The new CMS DAQ system for run-2 of the LHC," *IEEE Transactions on Nuclear Science*, vol. 62, no. 3, pp. 1099–1103, 2015.

[27] Xilinx, "Virtex-7 FPGA VC709 Connectivity Kit." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html

[28] HiTech Global, "HTG-710 PCIe Development Board." [Online]. Available: http://www.hitechglobal.com/Boards/PCIE-CXP.htm

[29] A. Borga, F. P. Schreuder, and O. Kharraz. Wupper PCIe DMA Engine. [Online]. Available: http://opencores.org/project,virtex7{_}pcie{_}dma

[30] P. Durante, N. Neufeld *et al.*, "100 Gbps PCI-Express Readout for the LHCb Upgrade," *IEEE Transactions on Nuclear Science*, vol. 62, no. 4, pp. 1752–1757, 2015.

[31] Intel. (2013) Intel VTune Amplifier XE. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe

[32] T. Bingmann. Parallel Memory Bandwidth Benchmark. http://panthema.net/2013/pmbw/.

[33] S. Williams, A. Waterman, and D. Patterson, "Roofline," *Communications of the ACM*, vol. 52, no. 4, p. 65, apr 2009.

[34] *MPI: A Message-Passing Interface Standard*, Message Passing Forum Std., 1994.

[35] T. Stitt, *An introduction to the Partitioned Global Address Space (PGAS) programming model.* Connexions, Rice University, 2009.

[36] OpenFabrics Working Group. Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/

[37] M. S. Birrittella, M. Debbage *et al.*, "Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics," *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 1–9, 2015.

[38] P. Hintjens, M. Sústrik, and Others. ZeroMQ. [Online]. Available: http://zeromq.org/

[39] A. Otto, D. H. C. Pérez *et al.*, "A first look at 100 Gbps LAN technologies, with an emphasis on future DAQ applications." *Journal of Physics: Conference Series*, vol. 664, no. 5, p. 052030, 2015.

[40] E. Bonaccorsi, J. Manuel *et al.*, "Infiniband Event-Builder Architecture Test-beds for Full Rate Data Acquisition in LHCb," in *Energy*, vol. 022008, 2010.

[41] D. Campora Perez, A. Falabella *et al.*, "The 40MHz trigger-less DAQ for the LHCb Upgrade," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 824, pp. 280–283, 2016.

[42] F. Carena, W. Carena *et al.*, "Preparing the ALICE DAQ upgrade," *Journal of Physics: Conference Series*, vol. 396, no. 1, p. 012050, 2012.

[43] T. Bawej, U. Behrens *et al.*, "Boosting Event Building Performance Using Infiniband FDR for CMS Upgrade," in *Proceedings of Technology and Instrumentation in Particle Physics 2014 (TIPP2014)*, Amsterdam, 2014.

[44] A. Dworak, M. Sobczak *et al.*, "Middleware trends and market leaders 2011," in *Conf. Proc.*, vol. 111010, no. CERN-ATS-2011-196, 2011, p. FRBHMULT05.

[45] A. Dworak, F. Ehm *et al.*, "The new CERN Controls Middleware," *Journal of Physics: Conference Series*, vol. 396, no. 1, p. 012017, 2012.

[46] M. Sústrik and Others. nanomsg. [Online]. Available: http://nanomsg.org/

[47] Ioannis Charalampidis. (2016) A libfabric-based Transport for nanomsg. [Online]. Available: https://github.com/wavesoft/nanomsg-transport-ofi

[48] T. Kawamoto, S. Vlachos *et al.*, "New Small Wheel Technical Design Report," CERN, Tech. Rep. CERN-LHCC-2013-006. ATLAS-TDR-020, Jun 2013, aTLAS New Small Wheel Technical Design Report. [Online]. Available: https://cds.cern.ch/record/1552862

[49] M. C. Aleksa, W. P. Cleland *et al.*, “ATLAS Liquid Argon Calorimeter Phase-I Upgrade Technical Design Report,” CERN, Tech. Rep. CERN-LHCC-2013-017. ATLAS-TDR-022, Sep 2013, final version presented to December 2013 LHCC. [Online]. Available: https://cds.cern.ch/record/1602230

List of Figures

List of Tables

Acronyms

COTS commercial off-the-shelf. 1, 8, 25, 29, 41, 71, 85

DAQ data acquisition. 1, 6–8, 10, 11, 14–16, 22, 25–27, 29, 34, 55, 57, 58, 86–88

DCS Detector Control System. 56–58, 71

DUNE Deep Underground Neutrino Experiment. 87

FTK Fast Tracker. 12

GBT Gigabit Transceiver. 30, 39, 71

GEFE GBT-Based Expandable Front-End. 39

HDLC High-Level Data Link Control. 30, 41

HEP high-energy physics. 55, 65, 85, 88

HPC high-performance computing. 55, 87, 88

L1Calo Level-1 Calorimeter Trigger. 71, 74, 76, 86

LAr Liquid Argon. 20, 32, 71, 74, 76, 86

LHC Large Hadron Collider. 87

MM Micromegas detector. 71, 73, 74

NSW New Small Wheel. 71–74, 76, 86

ROBIN Read-Out Buffer Input. 14, 23

ROD Read-Out Driver. 13, 14, 20, 24, 25, 27, 29

ROS Read-Out System. 12–16, 19, 20, 22–24

SPMD single-program-multiple-data. 55

sTGC small-strip Thin Gap Chamber. 71, 73

SW Small Wheel. 72

TDAQ Trigger and Data-Acquisition. 8, 13, 16, 22, 27

TTC Timing, Trigger and Control. 13–15, 27, 71