CERN Summer Student Report

Interfacing FPGA Track Processor Units with the LHCb Event Builder

Yara Al-Quorashy

Supervised by
Riccardo Fantechi and Niko Neufeld

August 8, 2019

Contents

1 Preamble 2

I CERN Summer Student Project 3

2 Introduction to CERN, LHCb and its Upgrade 4

3 LHCb Data Acquisition for the Upgrade 6

4 Co-processor for EB nodes and project goal 9

5 Work Progress 12

II Future work 18

III Reflection & Conclusion 20

Chapter 1

Preamble

This report is divided into three main sections, with chapters 2 and 3 covering introductory material related to CERN and, more specifically, the LHCb experiment and one of its proposed projects. Relevant information was extracted from [1] to provide the necessary background information related to the aforementioned chapters. Chapters 4 and 5 explain the project goals and the work progress implemented by me, Yara Al-Quorashy, as a summer student at CERN, respectively. The second section - chapter 6, outlines the future plans of the project, and finally, section three which is the conclusion, summarizes the lessons learned and challenges faced during my stay.

Last but not least, I would like to thank Riccardo Fantechi and Niko Neufeld for giving me the incomparable opportunity to work amongst some of the world's brightest minds. I would like to thank my supervisor, Riccardo Fantechi, for being extremely patient and supportive throughout this incredible learning process. He has taught me valuable technical skills as well as critical thinking and problem-solving skills. I would also like to thank my beautiful family and incomparable friends for their endless support throughout this experience, and for always pushing me to shoot for the moon.

Part I

CERN Summer Student Project

Chapter 2

Introduction to CERN, LHCb and its Upgrade

CERN, the European Organisation for Nuclear Research, is a scientific institute at the border of Switzerland and France. It houses the largest particle accelerator complex in the world. Via different stages of acceleration, protons are accelerated up to 6.5 TeV in the Large Hadron Collider (LHC). The LHC is basically a very large microscope to probe physics at the level of elementary particles. This is done by colliding two beams of protons rotating in opposite direction which results in interaction between the (partons of the) protons at up to 13 TeV.

Along the ring of the LHC, 4 large and a couple of smaller experiments can be found. The four large experiments are: ATLAS, CMS, LHCb, and ALICE. ATLAS and CMS are general purpose detectors. LHCb is specially set-up to perform measurements on B-quarks to look for CP-violation, and ALICE looks for quark-gluon-physics with heavy ion collisions. Where ALICE, CMS and ATLAS are symmetrical detectors along the beam pipe, LHCb is asymmetric and can only detect particles leaving within a certain angle and very high precision. Figure 2.1 depicts an overview of the (asymmetric) LHCb experiment.

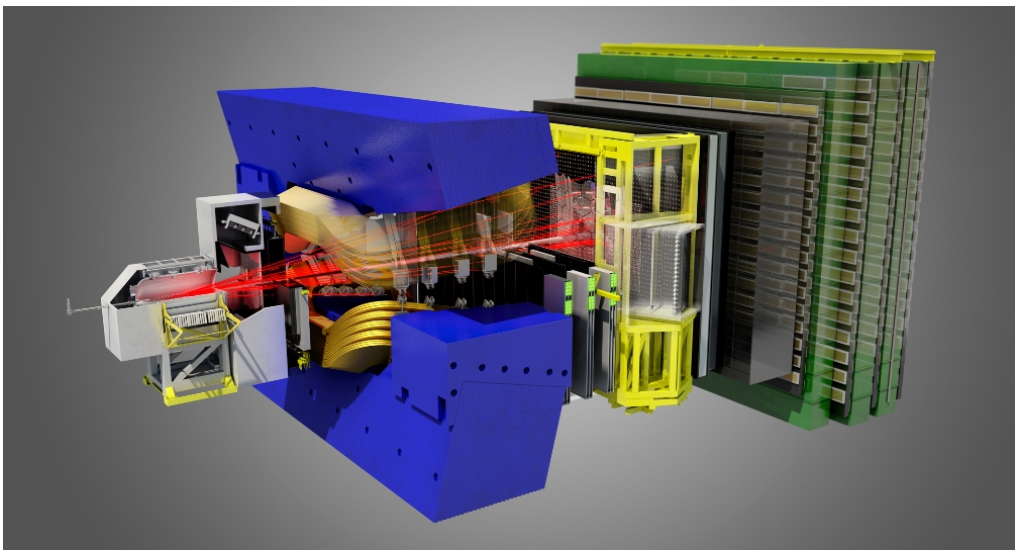

Figure 2.1: Overview Design of the LHCb Experiment

At the end of 2018, the LHC was shut down and the new Run of data acquisition (Run 3) will start in 2021. During this stop of about two years, both the detector and the Data Acquisition system of the LHCb experiment are being upgraded.

Chapter 3

LHCb Data Acquisition for the Upgrade

This section will give an introduction into the proposed Data Acquisition (DAQ) topology for the LHCb upgrade. Figure 3.1 shows a schematic overview of the DAQ system of LHCb. In the LHC, proton-proton interaction/collision (i.e. events) happen at a rate of 40 MHz. The data of the events is sent to a cluster of 500 computers: the Event Builders (EB). Every EB receives a data-stream from a different detector sub-unit. The task of the EB is to package all the data of each event from the detector sub-units into one data package, such that all further calculations (i.e. track reconstruction in the Event Filter Farm) can be performed on per event data instead of per detector data.

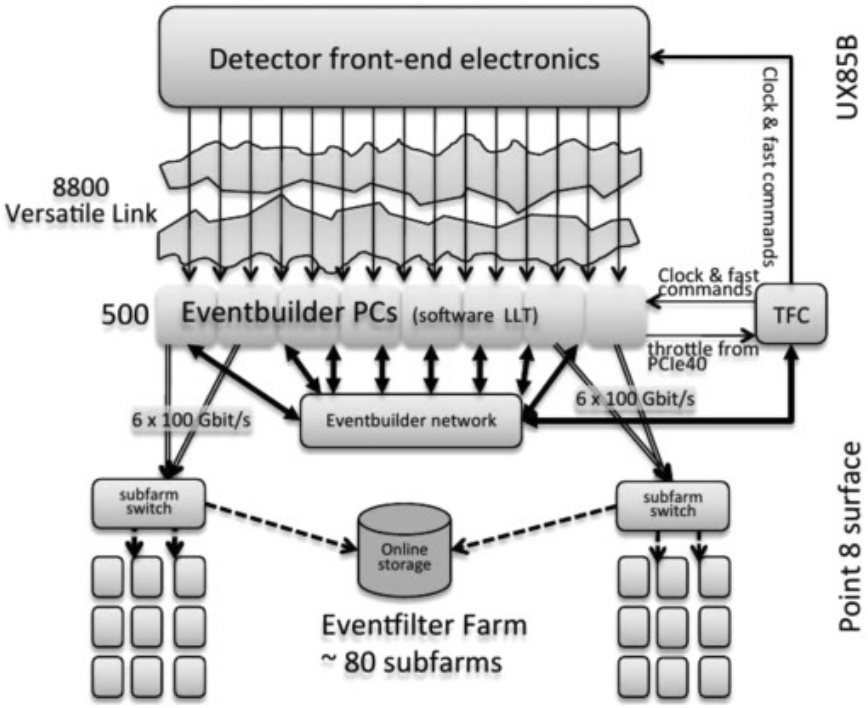

Figure 3.1: Overview of the DAQ System of LHCb

An EB can be schematized into a two processor system with separate memories. Figure 3.2 shows an overview of an EB as well as the data streams within an EB. The data of the different detectors enter an EB via optical interfaces at the PCIe40 card. This PCI-express expansion card will dump the data in the optical interfaces directly into Memory 1 via a DMA transaction (bold blue arrow). Meanwhile, the EB's elect one of the 500 EBs which will be responsible for building a certain set of events. All EB's will send the data they have related to this set to the elected EB via a LAN network (bold green arrow).

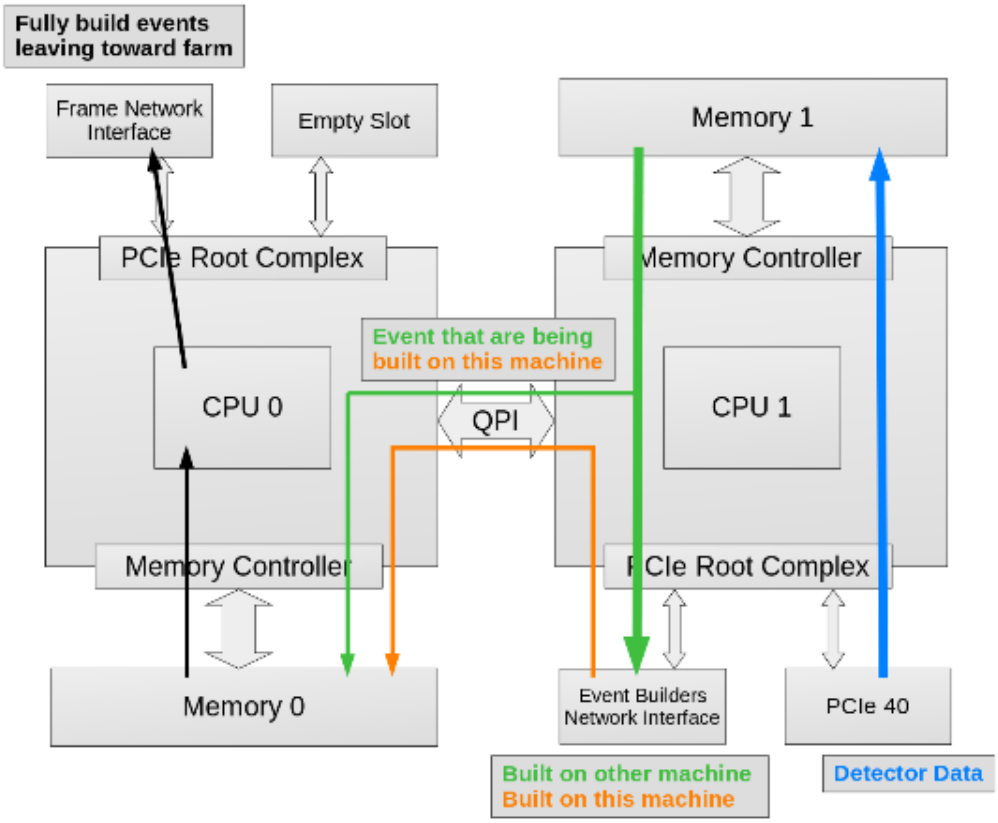

Figure 3.2: Overview of an EB and its data streams.

The elected EB will receive this data via its EB Network Interface. It will then store this data in Memory 0 (thin orange arrow) and will move the data it already acknowledged from Memory 1 to Memory 0 (thin green arrow). If all the data is available, it will be combined (i.e. the event is built) and the data package will be forwarded to the Event Filter Farm via a LAN (thin black arrows).

The main challenge of this process is the large amounts of data being transferred. For example, the incoming detector data (bold blue arrow) can be up to 100 Gb/s such that the memory load in CPU1 is quite heavy while the computational workload of both CPU's is moderate.

Chapter 4

Co-processor for EB nodes and project goal

The LHCb collaboration is investigating the possibility to perform partial data processing in the EB nodes. This section will elaborate on this idea and explain how it is related to the CERN Summer Student project.

4.1 Co-processor for EB nodes

One option currently under investigation would be to add a Tracking Processor Unit (TPU) to the empty PCIe40 slot of CPU0. For a specific detector, there will be a number of TPU's collecting data from the different sectors of it. All the TPU's will be interconnected using optical links and will implement the Artificial Retina algorithm to find tracks inside the detector. According to [2], the Artificial Retina algorithm can locate tracks sampled by a multi-layer detector and extract their parameters. This accelerator should get the detector data out of Memory 1 and perform its calculations on it. The set of TPU's will compute track parameters which should be added to the detector data as a separate "Pseudo Detector" block. The results are therefore injected back into the outgoing data-streams (probably by injecting the results into Memory 1). For performance reasons, both data transfers from and to memory should be implemented by DMA transactions. A schematic overview of the new layout of the event builders is shown in Figure 4.1. In this figure, the TPU is shown in pink, and the related data-streams (due to TPU's DMA transactions) are also shown in pink.

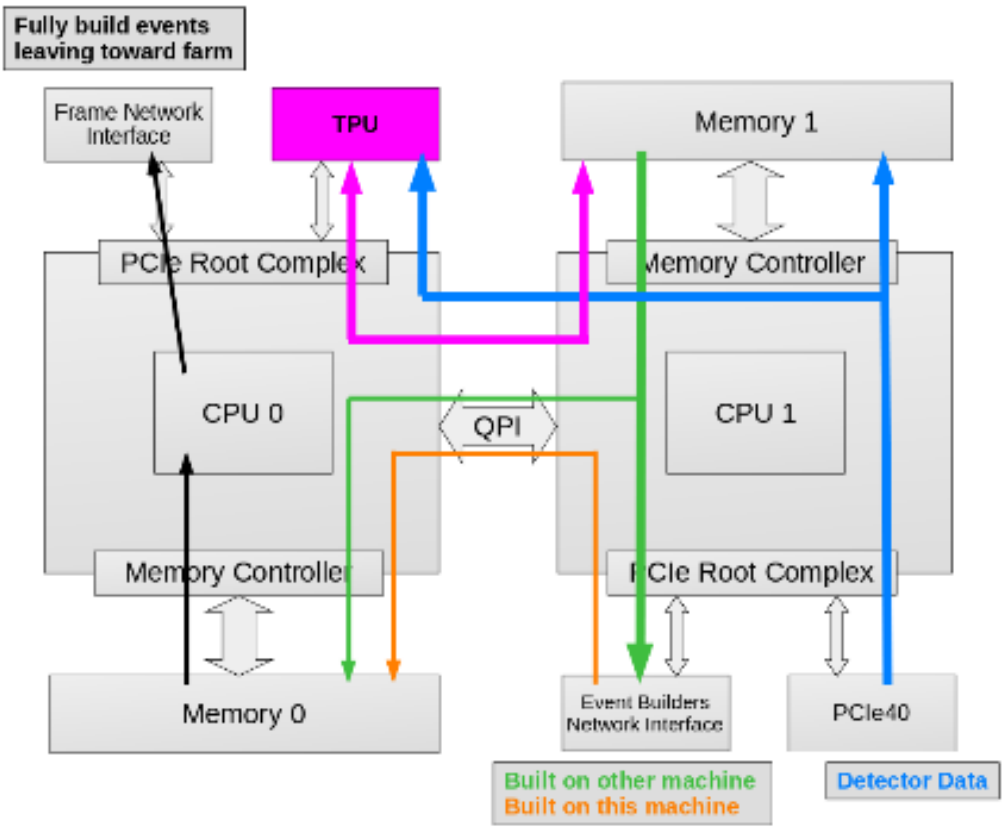

Figure 4.1: New Layout of the Event Builders.

Project Goals: The need of feeding the data into the TPU set as well as sending it to the EB encouraged us to find a suitable algorithm, given the basic constraint which is to avoid modifying the EB software. As a result, we were inspired to work on the device driver on the low level, properly synchronizing the data transfer among the EB and the TPU. By the end of my 8 week stay, I was expected to accomplish the following goals:

1. To simulate the communication and data synchronization between the different hardware peripherals (i.e. PCIe40 daq, EB, and the FPGA Accelerator).

2. To modify the PCIe40 Emulator Driver to implement and debug concurrent data transfer from the PCIe40 data generator to the Event Builder and the FPGA RETINA system.

3. To conduct tests on the loaded driver to debug the various modifications implemented.

I have worked on the standard LHCB PCIe40 driver and its utilities, using a separate GIT branch to maintain the modifications and to share

the code with my colleagues. It is worth noting that the work was carried out under (`/home/git/temp_emu_new/lhcb-daq40 -software`) folder and under a private file, (`mysync.c`).

Chapter 5

Work Progress

5.1 PCIe40 Emulator Simulation

In order to familiarize myself with the general communication flow behind the proposed project, I was assigned a task to simulate the data read and write synchronization mechanism between the emulated PCIe40 card, FPGA accelerator, Buffer, and the EB using a C program. The order would be such that the PCIe40 card would send data continuously to the FPGA accelerator through a Buffer, and the EB would read the same data through the Buffer only when the FPGA is done reading.

To achieve this, the flow has to be done concurrently, where the reading and writing is done continuously and in parallel. Hence, three threads were created, representing the aforementioned hardware peripherals, along with a circular array representing the Buffer. Every thread had two properties; a thread number and a circular array. The thread number is useful when checking which thread is allowed to read at any given moment (i.e. ensuring that if the EB tries to access the Buffer array, it would be prevented until the FPGA is done reading). The circular array represents the hardware peripherals' data storage which could accept data continuously, thus, being circular. A circular array refers to an array which can restart the index to zero once the last element is reached and there is more data to read, which would overwrite the previously stored data.

To keep track of the slots being written to and read from, two pointers were created; *read* and *write*. The *read* pointer of the Buffer is only updated once the data is read by the EB and not the FPGA. This is done to avoid any discrepancies in the data being read by the FPGA and the EB (i.e. the FPGA updating the read pointer and causing data loss for the EB).

Two methods were written to implement the action of writing by the PCIe40 card and reading by the FPGA and EB. In order to synchronize the data being written with what is being read, mutex locks were introduced in the code. The mutex lock is responsible for locking any shared resource

to avoid reading from or writing to the resource when it is not supposed to. In addition, a number of flags were introduced in the code to help in the synchronization process, where a flag was added to continuously block the EB from reading or updating the read pointer while the FPGA is reading.

5.2 PCIe40 Emulator Driver Modifications

In order to modify and test different modifications on the driver, an emulator exists such that any implementations can be tested on it and not the real card. According to [1], the Emulator can generate fake data, starting a kernel thread which fills the available slots in the circular buffer. The available slot is retrieved depending on the current positions of the read and write pointers. The write pointer is then updated which makes the emulator's functionality identical to that of the hardware card.

The emulator driver is very similar to the simulation of section 5.1, where the flags added the simulation have an equivalent here. The implementation of the EB and FPGA accelerator flags (i.e. ebflag and isaccel) helps in distinguishing between the PCIe40 card and the FPGA interfaces which ensures synchronized data transfer. In the emulator driver, linking streams of different interfaces is crucial when it comes to copying data from one stream to the other. In addition, saving the read/write offsets temporarily helps in ensuring that data transferred to the EB is the same data that was copied to the FPGA accelerator, but only when the FPGA is done with reading the data. Within these implementations, certain exceptions start to appear, which called for the following modifications.

- EB and Accelerator Flag Retrieval: In the emulated peripherals, different data streams belonging to certain components have to be synchronized whenever they begin reading or writing data. The device belonging to the PCIe40 has to be linked to the FPGA accelerator to which data is to be copied, and for this to happen accurately, certain flags are added to the implementation. The purpose of these flags is discussed in section 5.1, which is to prevent the EB from accessing the read pointer of the PCIe40 Buffer and updating it before the FPGA accelerator can finish copying data. For debugging purposes, the EB and Accelerator flags were retrieved to ensure that they are being set to the correct values at the right time.

  Files modified:

  - PCIe40 Emulator Driver:
    `pcie40_ioctl.h`: Defining two new parameters in the I/O control file (ioctl) using the offset and type of ioctl operations (IOR)

to control the ebflag and isaccel flags was the first crucial step in retrieving the flag values correctly.

- PCIe40 daq:
  `daq.h`: Since this is treated as one of the include files in the `main.c` file, the abstract methods were only defined in this file.

  `daq.c`: The two methods, `get_ebflag()` and `get_isaccel()` were implemented in this file, where both methods retrieve the ioctl parameters defined and issue an error and return -1 in case of an error in the ioctl call.

  `main.c`: This is where the get methods are called when displaying the stream status on the terminal upon using the "-t" option, namely in the `do_stream_status()` method.

- Implementation of the copy between different streams on the same or different interface(s):
  The purpose of this modification was to transform the DMA stream RETINA reading mechanism into a more generic implementation, which allows the method to copy data from one stream to the other without necessarily restricting it to any specific streams.

Files modified:

- PCIe40 Emulator Driver:
  `daq_emu.c`:
  `dma_stream_emu_rtna_read`: The natural structure of the buffer streams is scattered around the memory available, hence, the data being stored is mapped to certain buffers after retrieving the number of bytes to be read from the source stream, which is returned by the function. However, in order to synchronize the read and write pointers of the source and destination streams, a spinlock was utilized to allow only one thread to access the pointers at any given time, which prevented a race condition from occurring.

  `dma_stream_emu_rtna_write`: In this method, the implementation was altered owing to how certain exceptions of the read and write pointers began to arise. A first implementation of the copy function did not handle correctly the fact that in the emulator (as it is in the firmware), the buffer is built up of many segments, scattered in the memory. A good copy operation should handle this, switching from one segment to another, accordingly to the current read and write offsets. This was achieved by incrementing the segment number using the variables, *map_idx_in* and *map_idx_out* for the input and out streams, respectively. It

is worth mentioning that once the segment number reaches the maximum possible size, it rotates back to the first position to fetch new data and copy it into the output stream. The write function has then been made to handle this properly and it has been found to be a generic one; able to copy data to/from any pair of streams.

The original code did not take into account how the read and write pointers in the dma stream and copy thread may not be always aligned, which resulted in some incorrect data. Hence, the copy mechanism was modified depending on three scenarios; the read pointer of the input stream, `buff_off_in`, being greater than, equal, or less than the output stream, `buff_off_out`. To clarify, if the `buff_off_in` is greater than the `buff_off_out`, the data copied into the output stream would be the difference between the stream map size and the `buff_off_in` pointer in the input stream. The `buff_off_in` and `buff_off_out` pointers would then be updated to the new positions. If both buffer pointers are equal, the `buff_off_out` would be reset and the buffer would be incremented. If the `buff_off_in` is less than `buff_off_out's` position, the data copied into the output stream would be the difference between the stream map size and the position of `buff_off_out`. In this case, the `buff_off_out` is reset to 0 and the buffer is incremented.

- Adding the "-Z" option:
  The older version of the emulated PCIe40 daq had the capability to read from one device at a time given the device number and the source stream. However, to increase the system's flexibility, the "-Z" option was added which allows two streams to be read at the same time; the main and the RETINA streams. This provided the possibility to have a thread reading data from either the main or the RETINA stream of a different interface. This gives the ability to write the data in two separate files from the two streams and do a comparison to check for the accuracy of the transfer.

Files modified:

  - PCIe40 daq:
    `main.c`:
    Read rtna -Z option: This option was added to validate how accurate the data transfer between streams is being carried out. Its output is then compared to that of `thread_proc_out`. An independent device number, `g_dev_num2`, was added to give more

flexibility in terms of testing the transfer between the RETINA and main stream. Upon executing the "-Z" option, the chosen stream is enabled, and the `do_read()` is called, passing the stream and interface number as its parameters. The `do_read()` method is responsible for retrieving the stream chosen from the command on the terminal using a simple modulus function (i.e. RETINA stream would be denoted by a result of 0, whereas the main stream is denoted by a 1). Following that, the corresponding stream is added and the appropriate thread is created. The methods passed as parameters to either threads are either `thread_proc_rtna1` or `thread_proc_out`, which are both responsible for writing to RETINA or main stream, respectively.

5.3 PCIe40 Emulator Driver Tests

In order to test the validity of the modifications, a number of tests were conducted. The tests can either check the correctness of the output, measure the system's performance (i.e. throughput), or debug certain sections of the code.

Testing the system was done by running the following commands on the terminal:

- The user needs to log in using the command **ssh -X lhcb-hot** and then they will be prompted to type in their password.
- The user then needs to gain sudo privileges using the commands **sudo su** and **xauth add** `$(xauth -f username/.Xauthority list|tail -1`. This is to enable the graphical interface.
- The directory which includes all the necessary PCIe40 daq and driver files is located at cd/home/git/temp_emu_new/lhcb-daq40-software
- To check if the driver is loaded, one can use the command: lsmod—grep ...
- To remove a currently loaded driver, one can use the command: rmmod lhcb_pcie40 for the real driver or rmmod lhcb_pcie40_emu for the emulator.
- To load a driver, the command: modprobe lhcb_pcie40 mainmibs=16 metamibs=4 is used. In case of testing the emulator, the command would be: modprobe lhcb_pcie40_emu numboards=2. The default values for mainmibs and metamibs are in /etc/modprobe.d/lhcb_pcie40

- To check if the driver and boards are loaded correctly, the command ls /dev/pcie* is used.
- To obtain any Kernel messages which can help with the debugging purposes, dmesg is used.
- To compile any modified files, the command make is used only if the user is in the appropriate directory.
- To test any of the available options (i.e. -Z, -o, -t...etc.), one can use the following command: ./pcie40_daq -vr -i 0 -l 1 -fego /tmp/file concatenated with the option chosen. It is worth mentioning that feg options activate the generator, but these options may not be always needed.

Part II

Future work

The current PCIe40 Emulator Driver is under edits and updates to ensure that the project concept can be accomplished in an efficient way. In addition to the current features offered by the emulated hardware peripherals, the list below summarizes a few of the future project goals:

1. Conduct further tests on the PCIe40 Emulator Driver.
2. Complete the implementation and debugging of the PCIe40 firmware to be able to write from the host to the FPGA accelerator board.
3. Porting the relevant modifications implemented in the emulator driver in order to use the real boards.
4. Test and optimize the current throughput and performance of the PCIe40 emulator driver to achieve a higher data transfer rate to the EB and the FPGA accelerators.

Part III

Reflection & Conclusion

As my eighth week comes to an end, I realize how this experience truly added to my personality and skills. I remember the first week when I met with Riccardo Fantechi and was completely intimidated by the complexity of the project, and how throughout the weeks, I was able to slowly grasp the meanings behind the project concept, the code I attempted to modify, and most importantly, I learned the importance of diving into the task and picking up any useful piece of information that could help me along the way. This included asking a lot of questions, and falling into many mistakes. This was only possible because my supervisors selected me to be among their incredible team, and because of how Riccardo was very keen on helping me understand the different concepts and tasks.

An unfortunate aspect was the fact that the duration of my stay is somewhat short as compared to the amount of work I wished to accomplish here.

In terms of the challenges faced during my stay, the main obstacle was to prioritize what information I should focus on understanding in order to progress through the tasks I was assigned. At first, I mainly focused on reading the documentation available and the papers related to this project, but I found it difficult to keep track of the different variables in terms of the physics background along with the firmware design. However, the simulation task made it a lot easier to understand the general flow of the emulator driver and consequently, this helped me when it was time to modify the driver emulator code.

Bibliography

[1] Wendo Beuker. *Digital Design of Tracking Processor Unit.* CERN, Geneva, Switzerland, 2018.

[2] A. Abba, G. Punzi, F. Spinella, P. Marino, D. Tonelli, S. Stracka, F. Lionetto, D. Ninci, M. Petruzzo, A. Cusimano, et al. *A specialized track processor for the LHCb upgrade.* Technical report, 2014.