# SystemC simulation of the future SAMPA ASIC for use in the ALICE Experiment in Run 3

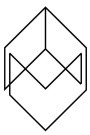
Håvard Rustad Olsen

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2015




# Abstract

The ALICE experiment at CERN is making upgrades to most of its equipment. One of its sub-detectors the TPC, requires new readout electronics because of an increase in data volume gathered from the detector. This means that new custom electronic chips needs to be developed. It can be costly to create many different prototypes when testing different specifications for the chips. This motivates finding a different and cheaper way to test the electronics.

One way this can be achieved is by creating a computer model of the electronic system, and do computer simulations on them. This thesis will evaluate the possibility of creating a model which is accurate enough to give realistic results, and by extension test different parts of the electronic system. The SAMPA ASIC is one of the new chips being developed for the readout electronics. This chip is the focus of this thesis, the goal is to identify the necessary size for its FIFO buffers. The SAMPA will receive a huge amount of data from the TPC detector, which means that it needs a compression scheme in order to deal with it. This thesis compares the performance and give some insight into two different compression schemes: Zero Suppression and Huffman encoding.

There are many tools to create a computer simulation, one being the SystemC framework. SystemC is a C++ library, which can be used to design a computer model and run simulations on it. Using this framework, a model of the readout electronics was created, and different types of data was passed through it. The results gathered from the simulations were studied and evaluated, determining their validity, and discussing what their impact on the development of the readout electronics.

The thesis shows that it is possible to create an accurate representation of the electronic system that gives realistic results. The results found regarding the size of the SAMPA chips FIFO buffers indicate that initial numbers would

be too small, and should be increased. Regarding the comparison between the two compression schemes, it was found that their results highly depended on the amount and shape of the input data. Huffman encoding works better with higher amounts of data than Zero Suppression, but relied more on the shape, making it more unpredictable.

# Acknowledgements

There are many who in their own way has helped me finish my work. First of I would like to thank my supervisors Håvard Helstrup and Johan Alme for their endless support and guidance during the last two years. They managed to help me acquire the domain knowledge I needed to accomplish my work, as well as motivating me during this time. I would also like to thank Dieter Röhrich, Arild Velure and Christian Lippmann for taking the time to help me with any technical questions or problems. For his part in designing the simulation model, I would like to thank Damian K. Wejnerrowski. In addition, I would extend my appreciation to Marius Fjeld Wold for helping me making the final touches to my thesis. For anyone who feels left out, just know that I am grateful to all who have shown their support.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

## 1.1 Motivation

The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is the world's largest particle accelerator, hosting multiple ongoing experiments. After a run period of more than three years, the LHC is shut down from 2018 until 2021[1]. The purpose of this shutdown is to perform maintenance on various equipment in the LHC, as well as significant upgrades to the different detectors, one of which is the detector used in A Large Ion Collider Experiment (ALICE). The ALICE detector consists of multiple sub-detectors, which combined collect an enormous amount of data. This amount is expected to increase after the shutdown period as the interaction rate of the LHC will increase. Due to the increase in data output, the ALICE collaboration is seeking to upgrade and enhance the detector capabilities[2]. This includes a partial redesign of the readout electronics, upgrades to multiple sub-detectors and additional hardware upgrades.

The Time Projection Chamber (TPC) is the ALICE detector's main subdetector for tracking and identifying particles. A starting design for the new TPC readout electronics has been made, and the different components are currently being developed. As this is still being worked on, many questions about the different components have yet to be answered. Are the current specifications sufficient to handle the expected increase in output from the detector? Do they have the necessary bandwidth to be able to send the data with minimal sample loss. Is the buffer memory sufficient to handle the traffic. Is it possible to optimize the current solution in any way?

The previous paragraph provides motivation for us to find a reliable way of determining a sufficient design for the readout electronics, that is both time and cost efficient. One strategy for solving this problem, which is explored further in this thesis, is creating a simulation of the system. Performing a simulation requires designing an accurate representation of the readout electronics, and creating a test-bench where it is possible to configure and run multiple tests.

## 1.2 Research Question and Thesis Goal

Given the motivation and introduction given in Section 1.1 the research question for this thesis becomes:

Is it possible to design and implement a simulation which directly represent the readout electronics, and in doing so will it have an optimizing effect?

Further explained, the main tasks of this thesis is to create a computer model of the main components of the readout electronics, and run multiple simulations on it. Experimenting with different configurations in order to find bottlenecks, faulty design or areas of improvement. The experiments are logged, and the results are be presented in an organized fashion.

## 1.3 Report Structure

Chapter 2 gives the reader the background information needed to be able to understand the different academic and scientific terms used, as well as some information about the context of the report. This includes information about CERN, the ALICE experiment and the physics most relevant to the thesis. It discusses the current readout electronics as well as the proposed upgrade. Chapter 3 goes into the topic of computer simulations, and the tools used in this thesis. In chapter 4 the problem areas of this thesis are further discussed, and the readout electronics are brought into the context of making a computer simulation. Chapter 5 will talk about the implementation of the simulation, what problems occurred along the way, and the chosen solution. The chapter discusses the design, as well as code snippets from the implementation. With the information given in chapter 4 and 5, chapter 6 discusses the results of the different simulation runs, and evaluate the solution. Chapter 7 concludes the thesis with some closing words, and work that can be done in the future.

## 1.4 Audience

The reader of this thesis is expected to have general scientific knowledge, and a good understanding of computer science. Basic concepts are assumed to be known by the reader and as such, they will not be given much elaboration. However, more complicated terms will be given a brief explanation if it is considered important to understand the thesis. Some physics terms are also used, but the reader is not expected to be familiar with these, and they will be explained when necessary.

# Chapter 2

# Background

## 2.1 CERN

CERN is a European research and scientific organization based out of Geneva near the Franco-Swiss border[3]. CERN is a collaboration between 21 countries with a member staff of over 2500, and more than 12000 associates and apprentices. The organization was founded in 1954 and has since then been the birthplace of many major scientific discoveries. These are not limited to discoveries in the field of physics, but includes the creation of the World Wide Web[4]. Currently the biggest project at CERN is the LHC particle accelerator, which serves as the foundation for multiple experiments in the field of particle physics.

## 2.2 The Large Hadron Collider

Starting up on 10 September 2008, the LHC is the latest construct added to CERN's particle accelerator complex[5]. It consist of a 27 kilometre underground ring of superconducting magnets, which boost the energy of the particles travelling inside the collider. The collider contains two adjacent parallel high-energy particle beams. These beams consist of protons extracted from hydrogen atoms by stripping them of electrons. Along the collider there are four intersection points where collisions occur. Each point corresponds to the location of a particle detector - ATLAS, ALICE, CMS and LHCb. The particle detectors are each built and operated by large collaborations, with thousands of scientists from different institutes around the world. The beams travel at close to the speed of light and are guided by magnetic fields, which are created and maintained by superconducting electromagnets. Superconducting meaning that it is in a state where it can most efficiently conduct

electricity, without resistance or energy loss. Achieving this state requires cooling the magnets to -271.3° C , which is done by the distribution of liquid helium. The layout of the LHC ring including its four collision points can be seen in Figure 2.1.

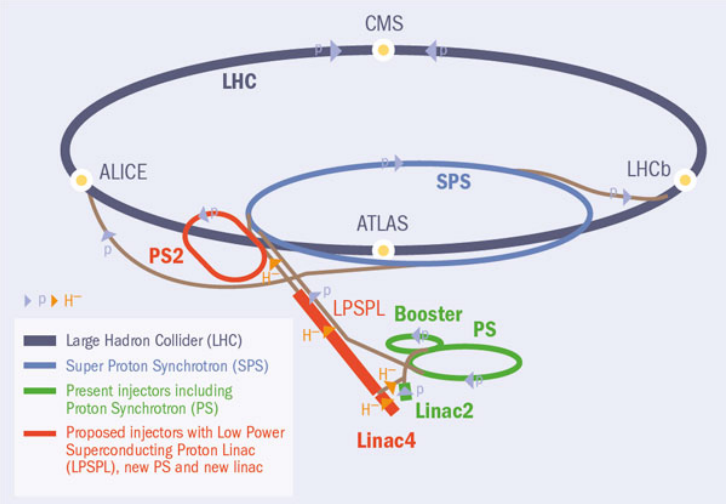



Figure 2.1: The Large Hadron Collider [6].

The beams travelling inside the LHC reach an energy-peak of 7 Tera Electron Volt (TeV), which means that on impact with each other the collision reach an energy of 14 TeV[7]. During a normal run of the collider there will be about 600 million particle collisions per second during a period of 10 hours. This leads to a huge amount of data for each of the detectors to read out. ALICE is the detector which produce the most data per collision, with a design value of about 1.25 GB/s written to permanent storage. The high amount of data per collision is produced primarily by the TPC sub-detector, which records a high number of points per track, and has a low momentum threshold. Detectors like ATLAS and CMS are designed with a higher momentum threshold, but can cope with significantly higher collision rates than ALICE. ALICE is designed for the study of heavy ion reactions, where particle correlations at low momentum is an important measure. The number of tracks correlating with momentum is exponentially declining. This means that a lot of tracks which do not get registered in ATLAS, produce data in ALICE.

## 2.3 ALICE

### 2.3.1 Introduction

ALICE is designed as a heavy-ion detector, which means it studies collisions between heavy nuclei of high energy[8]. The experiments are run with two different particle collision systems, lead-lead(Pb-Pb) and lead-proton(Pb-p). Both systems produce an extreme amount of temperature and density. They produce different, but equally interesting results. Pb-Pb collisions create Hot Nuclear Matter, while Pb-p create Cold Nuclear Matter. The explanations for these types of matter is not within the scope of this thesis and will not be discussed further. The high temperature and density is necessary to produce a phase of matter called quark-gluon plasma.

### 2.3.2 Quark-gluon Plasma

Shortly after the Big Bang, the universe was filled with an extremely hot cluster of all kinds of different particles moving around at near the speed of light[9]. Most of these particles were quarks, fundamental building blocks for matter, and gluons which ties quarks together in order to form heavier particles. Normally quarks and gluons are very strictly tied together, but in the conditions of extreme temperature and density as in the time shortly after the Big Bang, they are allowed to move freely in an extended volume called quark-gluon plasma. The existence of quark-gluon plasma and its properties is one of the key issues in Quantum Chromodynamics (QCD). The ALICE collaboration studies this, observing how it behaves.

### 2.3.3 The Detector Setup

The detector weight is about 10,000 ton, it is 26 m long, 16 m wide, and 16 m high[10]. It consists of 18 sub-detectors, each with its own set of tasks regarding tracking and identifying particles. This large number of sub-detectors is needed in order to obtain a complete picture of the complex system which are being studied(i.e different types of particles and the correlations between them). Most of the detector is embedded in a magnetic field, created by a large solenoid magnet. It makes particles formed in collision bend according to their charge, and behave differently relative to their momentum. High momentum equals near straight lines while low momentum makes the particles move in spiral-like tracks. During lead to lead collisions, the collision rate peaks at 8 kHz(Where Hz is defined as number of events per second). The number of recorded events is smaller in practice because the ALICE detector

uses a triggered readout, which only triggers on head-on(central) collisions. The maximum readout rate of the current ALICE detector is 500 Hz, which is sufficient to track central collisions. Figure 2.2 shows a cross section of the detector as it is today with the red solenoid magnet, and all sub-detectors labelled.

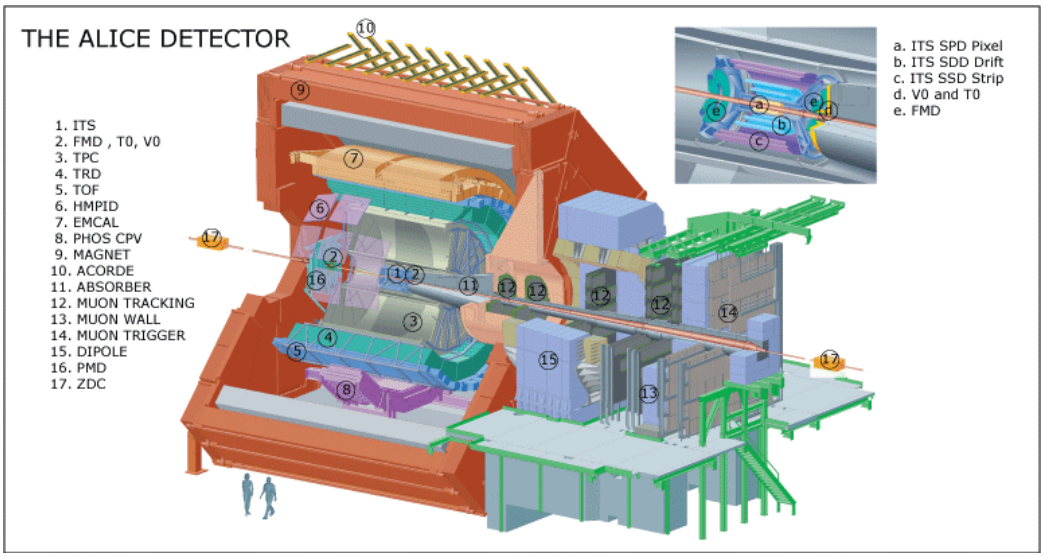



Figure 2.2: The ALICE detector [11].

## 2.4 The TPC Detector

### 2.4.1 Introduction

One of the most important sub-detectors, and the one that is relevant for this thesis, is the TPC detector. Located at the center of the ALICE detector, it is among the first entry points when gathering data from a particle collision. It is a $88m^3$ cylinder filled with gas. The gas works as a detection medium, which means that charged particles from a collision crossing will ionize the gas atoms, freeing electrons that move towards the end plates of the detector. The readout is done using specially designed readout chambers, which are capable of handling the high amount of data produced in heavy-ion collisions.

### 2.4.2 Readout Electronics

Signals from the readout chambers are passed along to the front-end readout electronics, which today consist of 4356 ALTRO Application Specific

Integrated Circuits (ASIC) chips[12]. ASIC is the term used for specially customized chips, rather than chips with a more general-purpose use[13]. The ALTRO chip is made up of 16 asynchronous channels that digitize, process and compress the analogue signals from the readout chambers. It operates on a so called triggered readout mode. In short when ALTRO receives the first trigger, it stores the following data stream into memory, holding on to it until it is ready to pass on the data. The front-end electronics are able to readout data at a speed of up to 300 MB/s.

The Front-End Card (FEC) sends the digitized signals further down the readout chain to the Readout Control Unit (RCU), where it is further processed and shipped to and stored in the online systems. The schematics are shown in Figure 2.3.

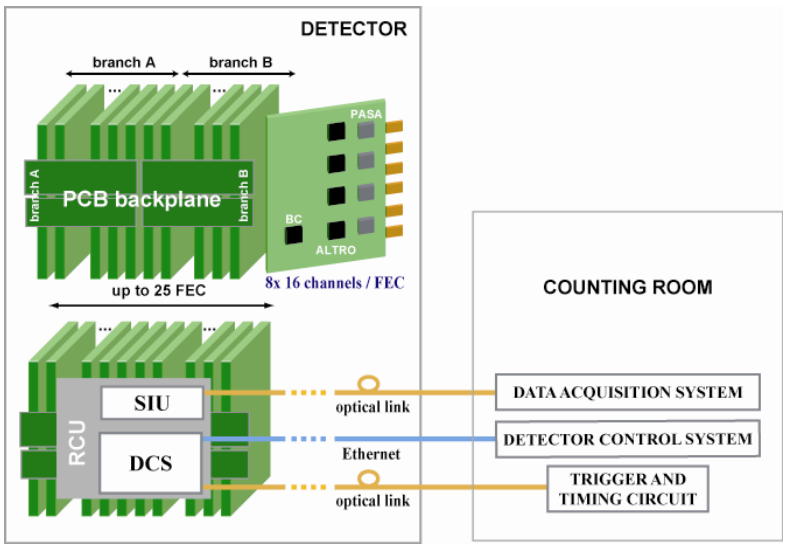



Figure 2.3: Readout schematics for the current TPC detector [14].

## 2.5 Long Shutdown 2

As mentioned in Section 1.1 the LHC ring will be shut down for approximately three years, starting in 2018. During that time, the ALICE detector will undergo an extensive upgrade. The upgrade strategy for ALICE is based on the expected increase in collision rate to 50 kHz, and will now track every collision. Essentially this comes down to an increase by a factor of 100, compared to what is achievable today.

To be able to handle the increase in collision rate the TPC will receive upgrades to both its readout chambers, and front-end readout electronics. The current Multi Wire Proportional Chamber (MWPC) based read-out chambers will be replaced by Gas Electron Multiplier (GEM) detectors, which has a much higher readout rate capability. Signals will be passed from the new readout chambers to the FEC via a readout pad structure similar to the one presently used. There are multiple pad structures depending on its location on the detector, but the difference in structure is not relevant for this thesis. What is relevant however is that more data is expected from low pad numbers, an example of a pad structure is shown in Figure 2.4.

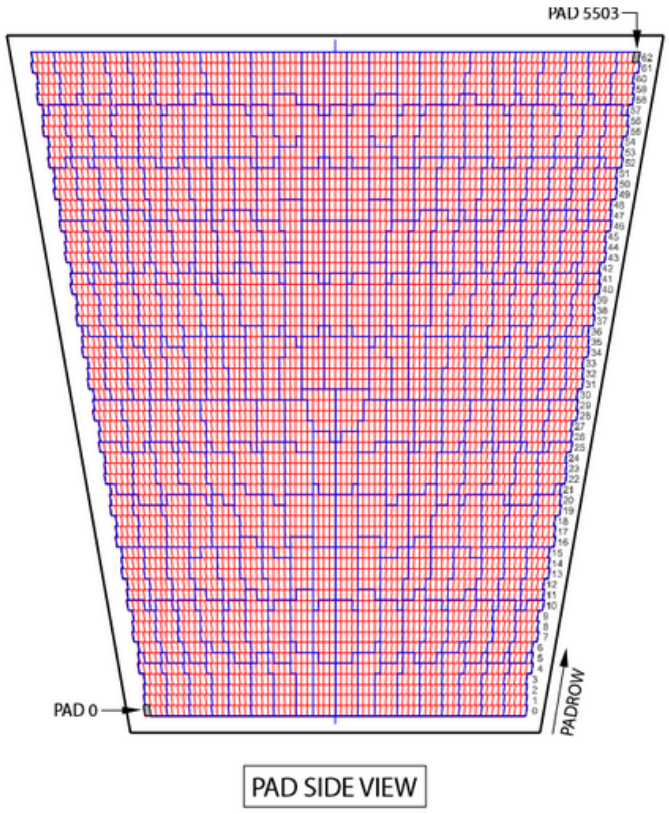



Figure 2.4: Pad structure of an Inner Readout Chamber(IROC)[15].

The entry point in the FEC is the new custom-made ASIC, the SAMPA, which will replace the ALTRO chip[16]. The SAMPA chip is capable of processing signals asynchronously in 32 individual channels, each channel is directly connected to a single pad. Signals are further on digitized and concurrently transferred to the GBTx, which enhances the signal strength

and transmits them via multiple optical fibre links to the Common Readout Unit (CRU). The CRU can be thought of as the new RCU and serves as an interface to the online systems. The data flow from the detector, and a working schematics can be seen in Figure 2.5. Chapter 4 goes into more detail about the readout electronics in the context of our simulation.

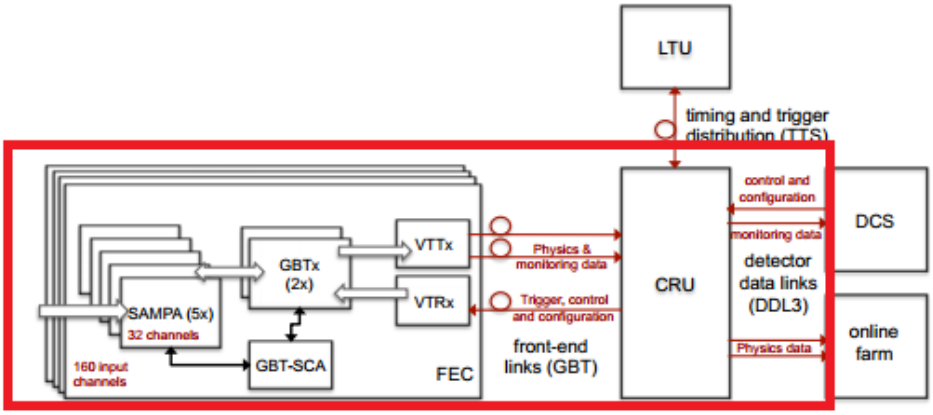



Figure 2.5: Schematics of the readout electronics [14].

# Chapter 3

# Simulations

## 3.1 Simulation Theory

A simulation can be seen as the imitation of a real-world system and its operations over time. This requires a model representation of the system which is accurate enough to conduct experiments on, which produce real-like results. The model should include key characteristics, specifications and functions of the selected system, but in a simplified fashion. A simulation model can take many forms as it can be used in different contexts ranging from physical object such as electrical circuits, bridges, and even entire cities to abstract systems like a mathematical equation or a scientific experiment[17].

As the model represent the system itself, the simulation represents its operations over a set period of time. The simulation is normally conducted in a controlled environment that makes it possible to observe, monitor and log results. To achieve efficient experiments using a simulation, it should be easy to change its parameters with respect to what is being tested.

There are several benefits of simulating a system instead of creating and testing the real thing. A simulation will in most cases be very time efficient, you can conduct the same types of experiments on the system in a much shorter time compared to the real thing. This means that more information about the systems behaviour and its limitations can be gathered in less time, which in turn can result in a better final product. Creating the real-world system can often be very expensive, which may limit the amount of prototypes or test-products that are possible to create. Therefore, using the results of a simulation to fine tune the specifications before starting to produce prototypes will cut unnecessary development costs by a significant

margin.

Taking the upgrade of the readout electronics for the ALICE detector as an example to further address this point one can see the usefulness of not having to create multiple custom hardware components, all with different specifications. In regards to the readout electronics, another important point is that the proposed designs might already function properly, but there is always room for improvement. Finding out that the design does not need as much memory, or less optic fibre cables can impact the overall production costs. One way to efficiently and accurately simulate hardware components is by creating a virtual computer simulation.

### 3.1.1 Computer Simulations

Using computers to perform simulations becomes more and more useful because of their incredible computational power, and ability to quickly produce results. This is important as simulations often become quite complex, both in regards to computational complexity and level of difficulty to understand and further work with. Therefore, it can be wise to use existing tools to help make the process easier. There is an array of different tools that can be used for various kinds of simulations. They vary from complete frameworks, with graphical user interfaces to tools which help programmers write their own simulation programs. The later requires the most work, but will most often end with the better results as you can tailor your simulation on a lower level than with a complete framework. A programming tool designed for creating simulations is the SystemC library, which is discussed in the following section.

## 3.2 SystemC

### 3.2.1 Background

SystemC is a system design library based on C++[18]. It provides an interface to easily create a software model that represents a hardware architecture, and together with standard C++ development tools it is possible to quickly build a full scale simulation. Following the standards of C++, SystemC is built to be easy to understand for both software and hardware developers, resulting in clearer cooperation between them while developing the hardware design. The SystemC library provides an object-oriented approach to model

design, where a single C++ `class` represents a model. This makes it easy to separate concerns between the different models in your simulation.

When simulating a hardware system there are a couple of key points to be aware of, firstly you need to be able to handle hardware timing, clock cycles, and synchronisation. One of the benefits of SystemC is that it takes care of all of this, again taking advantage of the object-oriented nature of C++ to extend its capabilities through `classes`. Here are some of the other features SystemC provides, with emphasis on the ones needed to understand code snippets shown in this thesis.

**Modules**
Container `class` representing a hardware model.

**Processes**
In short, processes are methods inside a module, which describe the module functionality.

**Ports**
Ports represent the input and output points of a module, they can be connected to other modules through Channels. When you declare a port in a simulation, it is required to specify if the port is an input, output or bidirectional port. This is done by specifying a channel interface for the port. Example of a port using a input First-In-First-Out (FIFO) interface:

```
sc_port<sc_fifo_in_if>
```

.

**Channels**
Channels are the wires connecting two Ports. SystemC comes with three predefined channels: FIFO, mutex, and semaphore. It is possible to configure custom channels, but in most cases it is not necessary.

**Signals**
Signals represent data sent between modules via ports. They can be arbitrary data types like `bool` or `int`, but also user defined types.

**Rich set of data types**
SystemC supports all data types defined in C++ as well as multiple custom types.

**Clocks**
SystemC comes with clocks, which can be seen as timekeepers of the system during a simulation.

### 3.2.2 Small Example

To get a basic understanding of how a SystemC simulation looks like, it is useful to see it in action. The following Figure 3.1 and Listings 3.1-3.3 define a trivial example with only two modules: a `Producer` and a `Consumer`. The `Producer` will increase a counter every clock cycle, and send a `bool` whose value depends on whether the count is an even number or not, and send this value to the `Consumer`, which registers how many times the `Producer` counted an even number. The example uses a FIFO channel, connected between an output port on the `Producer`, and an input port on the `Consumer`.

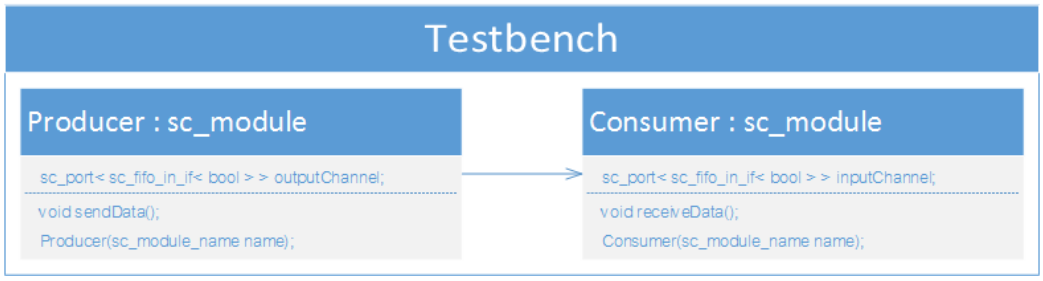



Figure 3.1: Basic SystemC example.

```
1  SC_HAS_PROCESS(Producer); //macro to indicate that the module
       has process
2
3  //Constructor with name of module as parameter
4  Producer::Producer(sc_module_name name) : sc_module(name){
5    SC_THREAD(sendData); //Registrer the sendData thread
6  }
7
8  //Clock frequency: 100 Mhz; 1 / 10 ^ 7 = 10 nanoseconds
9  void Producer::sendData(){
10   bool signal = false; //signal value
11   int count = 0; // count variable
12
13   while(true){ //infinite loop
14
15     if(!(count % 2)){ // if count is even, signal = true
16       signal = true;
17     }
18     outputChannel->nb_write(signal); //write signal to output
           channel
19
20     signal = false; //reset signal
21     count++; //increase count
22     wait(10, SC_NS); //End of a clock cycle, wait 10 nanoseconds
23   }
24 }
```

**Listing 3.1: Producer module.**

```
SC_HAS_PROCESS(Consumer); //macro to indicate that the Module
    has 1 or more processes

//Constructor with name of module as parameter
Consumer::Consumer(sc_module_name name) : sc_module(name){
  SC_THREAD(receiveData); //Registrer the receiveData thread

}
//Thread which runs until the simulation is over.
//Clock frequency: 100 Mhz; 1 / 10 ^ 7 = 10 nanoseconds.
void Consumer::receiveData(){

  int numberOfEvens = 0; // counts number of evens
  bool receivedSignal = false; //received signal variable

  while(numberOfEvens < 10){ //stop lopp when received 10 evens

    if(inputChannel->nb_read(receivedSignal)){ //receiving
        signal; nb_read returns true if signal is read.
      if(receivedSignal){
        numberOfEvens++; // if signal is true, count was even.
      }
    }
    wait(10, SC_NS); //End of a clock cycle, wait 10 nanoseconds
  }
  sc_stop(); //Force stop simulation.
}
```

**Listing 3.2: Consumer module.**

```
int sc_main(int argc, char* argv[]) {

  Producer producer("Producer");
  Consumer consumer("Consumer");

  sc_fifo<bool> channel(20); //(First-In-First-Out) channel with
      depth of 20.

  //Connecting Producer-Consumer channel.
  producer.outputChannel = channel;
  consumer.inputChannel = channel;

  sc_start(); //Alternative: sc_start(30, SC_NS) - Specified
     simulation lenght.

  return 0;
}
```

**Listing 3.3: Simulation test-bench.**

SystemC can be used to create very low level hardware descriptions and models, and can interface directly with hardware description languages like VHDL and Verilog. This is one way to create a simulation, and the models will be accurately represented by doing so. The other way is to have a high level of abstraction, leaving out the unimportant details and focus solely on the expected problem areas. There are benefits and drawbacks with both ways, but sticking to a high abstraction level can in complex cases make it easier to work with the model design, allowing you to focus on the important parts.

# Chapter 4

# Problem Description

The previous chapters briefly introduced the problems of this thesis, discussed relevant background information and looked at tools and the method of solving them. Essentially it boils down to creating a model based on the schematic of the TPC readout electronics, run multiple simulations, testing different parameters for the involved components. Until now there has only been an introduction level description of the different components that are included in the simulation model. This chapter goes deeper into them, giving detailed information about their design parameters, and how the ALICE experiment data is handled by them. Not going too far into the task of implementing this in a SystemC environment, but focus on the different problem areas, what is required in order to solve them and what goals to achieve.

## 4.1 Model Design

The hardware design which is being simulated has already been shown in Figure 2.5. The proposed schematic shown consists of 12 FEC cards for every CRU. Each FEC consists of 5 SAMPA and 2 GBTx ASICs, with the CRU being connected to them via 24 optical links. Out of the 3 main chips, the SAMPA and the CRU are the most interesting as they are still being developed, testing them can provide a lot of valuable feedback. The GBTx is a completed component, so even though it is part of readout electronic being simulated, it will only be a very shallow abstraction of it. This means that it will remain as an empty module whose objective will be to just pass along received data to the correct output links. One important note about the GBTx input and output links. Each GBTx has 10 input e-links, each with a transfer rate of 320 Mbit/s, giving an effective input speed of 3.2 Gbit/s per GBTx. The output is 1 optical fibre link with a speed of 3.2 Gbit/s, giving

the GBTx the same input and output speed. This is the reason that letting data flow directly through the GBTx in the simulation is possible. The next sections goes into details about the more important components.

### 4.1.1 SAMPA

The SAMPA ASIC is based on the work from its predecessor, the ALTRO. Just like the ALTRO it will be the first step for signals being tracked in the TPC detector. The signals will be processed, compressed, digitized, and temporarily stored in the SAMPAs memory before they are passed along. The SAMPA has 32 integrated channels, which separately and asynchronously process the analog signals coming from the detector[14]. Each channel has a readout speed of 10 bit on a 10 MHz clock, which combined results in 3.2 Gbit/s. The channels also have their own FIFO buffer memory where signals coming in are stored as they wait to be sent along. The most efficient size for these buffers is one of the things the simulations will hopefully provide. The output links for the SAMPA chip consists of 4 e-links connecting them to the GBTx. Each e-link has as mentioned in the previous section, a speed of 320 Mbit/s, which sums up to 1.28 Gbit/s[16]. The e-links are connected to 4 readout buffers on the SAMPA that read from the channel buffers and transports the data to the e-links. The readout buffers read from 8 channels each. Since each SAMPA and GBTx has a specific number of output and input links, only certain setups are desirable. This is why the proposed schematic uses 5 SAMPA and 2 GBTx chips for each FEC. This setup gives exactly 20 output links from the SAMPA chips, and 20 input links on the GBTx chips.

As the ALTRO, the SAMPA can be run in triggered readout mode, but in addition it can be run continuously. Being able to read out continuously is a necessary upgrade required to handle the increased data load coming from the detector. During continuous mode the data acquisition is uninterruptible, meaning that there is no pause between reading two consecutive events from the detector. The difference compared to triggered mode can be seen in Figure 4.1. Every event, from now on referred to as a time frame, is 1023 clock cycles long, and all 32 channels of the SAMPA use the same time frame. This means that every 1023 clock cycle a 1023 long time frame is initiated for all 32 channels, meaning they can readout 10 bit data samples 1023 times during this window. A synchronization input allows multiple SAMPA ASICs to align their time frames with respect to each other[16].

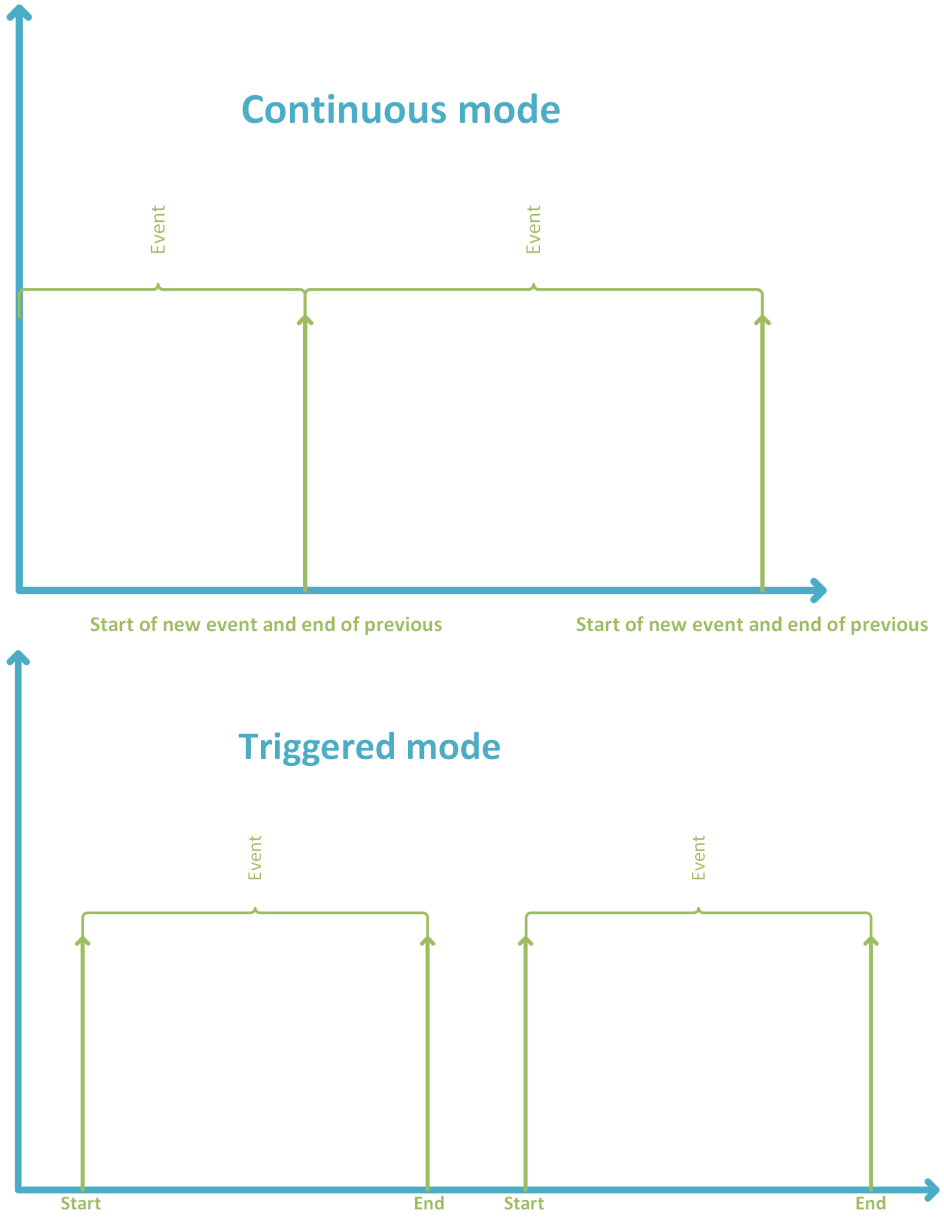



Figure 4.1: Continuous vs Triggered mode.

The SAMPA creates data packets from the data assembled from each time frame. Consisting of a header of fixed size 50 bit, followed by a list of 10 bit samples, created from a single time frame. Even though a time frame consists of 1023 clock cycles, in practice a maximum of 1021 samples are received each time. This is due to the fact that 2 * 10 bit words are required to represent cluster size (size of consecutive samples) and a time stamp. The headers are

stored in their own FIFO buffers, separate for each channel, much like the sample buffers. Figure 4.2 shows the structure and format of the packets.

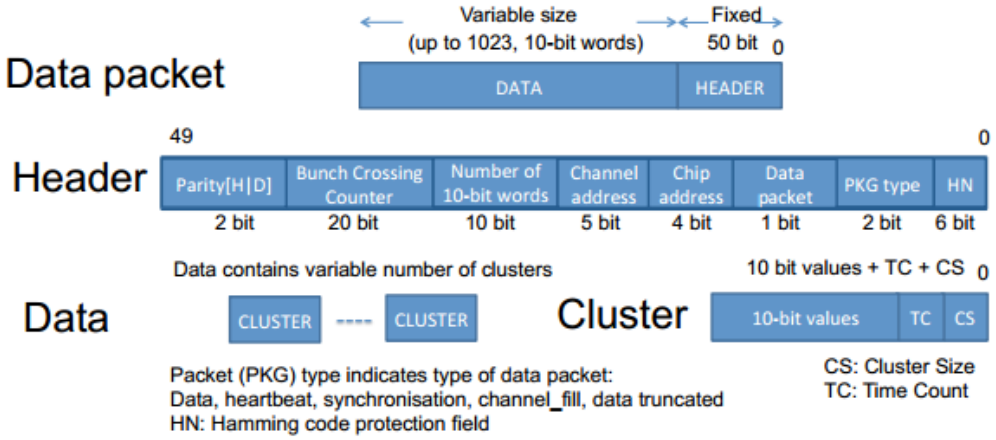



Figure 4.2: Data packet format [16].

The header consists of information regarding the data, such as address of the channel and chip, number of data words in the time frame and packet type. The packet type is used as a marker to see if anything out of the ordinary has happened to the data. For example if there are no samples in the time frame, causing the packet type to just become a channel fill packet. It can indicate if the stream of data was cut short because the FIFO buffer was full, causing buffer overflow. In case of buffer overflow all data for the particular time frame is discarded and the empty packet is sent with type overflow. Overflow can cause a lot of data to get discarded if the SAMPA can not empty the buffers fast enough, this can happen if the buffers don't have enough space. As the input rate is 3.2 Gbit/s and the readout speed is 1.28 Gbit/s, the SAMPA can receive up to 2.5 times more data per second than it can pass along. This is why the FIFO buffers are necessary, and finding a size which is sufficient, without resulting in overflow is crucial.

There have been done some calculations on how much data is likely to be received from the detector at any given time. It is estimated that on average over all channels for every SAMPA there is roughly 30% occupancy. This means that on a global average there is 30% data in every given time frame. Some channels may be full while others are empty, and some may have 40%, but on average there is 30%, which means 306 samples out of 1021 for every time frame. Taking this into account when calculating the input speed of the

SAMPA gives 960 Mbit/s which the design should be able to handle without any buffer overflow. Even though there is an estimated average occupancy there can still be some channels which time frame after time frame gets a lot more than that, so how much can the design handle? These are some of the questions the simulation will provide answers to.

### 4.1.2 CRU

The CRU serves as an interface between electronics directly on the detector and the online computing systems. It is based on high performance Field-Programmable Gate Array (FPGA) processors, with optical fiber used as input and output [16]. The CRU is somewhat out of the scope of the thesis, and will be regarded in the same fashion as the GBTx. How the CRU is implemented in our design model has no effect on the tests that are performed on the SAMPA and its channels. It is discussed in the thesis work of Damian K Wejnerowski, who is simulating the CRU and inspecting it in more detail.

## 4.2 Signal Processing in the SAMPA

The SAMPA chips will receive and process a huge amount of data, both relevant signals and background noise. Section 4.1.1 talked about occupancy and amount of samples in each time frame. The estimated amount of 30% refers to relevant samples, removing or compressing the background noise. Seeing as it will always be some interference in the background, there will always come samples with data, and gathering all will be a waste of time and space that could be used on the actual collision data in the detector. Figure 4.3 shows two actual events collected from two separate ALTRO channels, the events will look similar after the upgrade and we can use this as a starting point. The x-axis expresses the current time bin within a time frame from 0 to 1021. Here one can see that every sample in the time frame has a value. Most of which is between 48-52, but peaking here and there. Those peaks or pulses are what is interesting, everything else is considered noise and should be removed. In order for any compression scheme or method of reducing noise to be valid, it needs to have a compression factor of at least 2.5 for the average amount of data being processed. The compression factor is the number of bits in a time frame before compressing compared to after. `factor = (bits before compression / bits after)`. There are a number of ways to reduce the amount of noise, and/or compress the data to a manageable size. What has been used with the current setup and is also discussed to use in the upgraded setup is Zero Suppression.

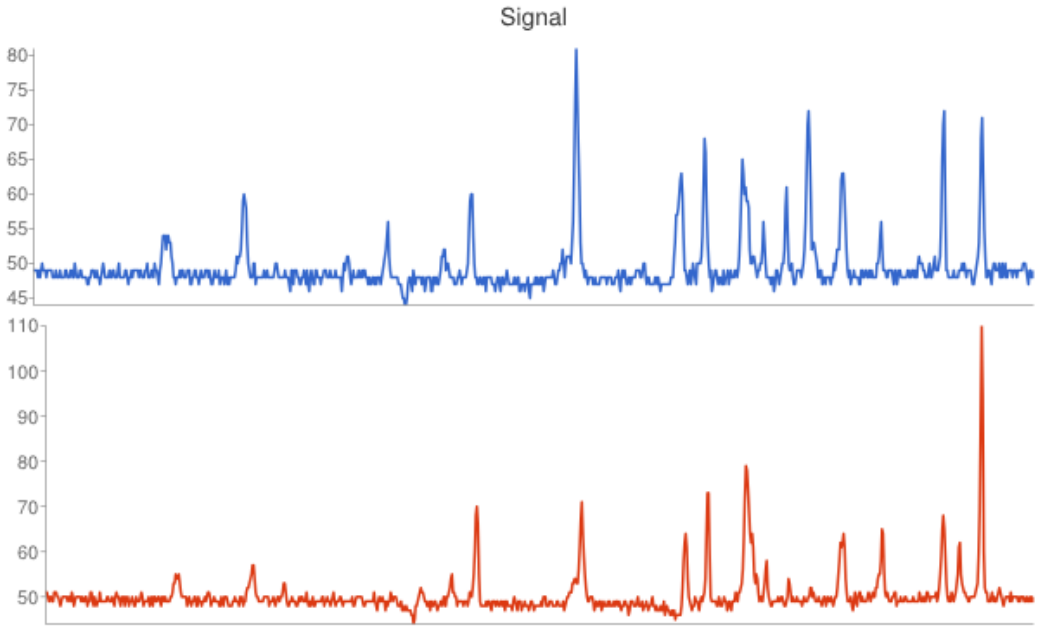



Figure 4.3: Two signals from RUN 1.

### 4.2.1 Zero Suppression

Zero Suppression is the process of removing insignificant values below a given threshold or baseline[19]. In order to apply this to remove the background noise without discarding any important samples, a baseline for the Zero Suppression must be established. The challenge is that the baseline may shift, in the case of our two example time frames from Figure 4.3, the first one has a visibly lower baseline by 1 or 2. In the upgrade plans described in [16], it is specified how the signal processing will take place. It works by looking at consecutive signals with value over the set threshold, confirming that the peak is indeed a real pulse. The term real pulse refers to a sequence of signals over the threshold with more than one signal, standalone values over the threshold are discarded. The difference is displayed in Figure 4.4.

Because of the fact that Zero Suppression removes signals from various places in a time frame, the data looses its temporal positioning. Therefore, every real pulse must be tagged with a time stamp and a word representing the number of words in the pulse. Since for every pulse we add two words, if two consecutive pulses are closer than three words they are merged and counted as one (Figure 4.5).

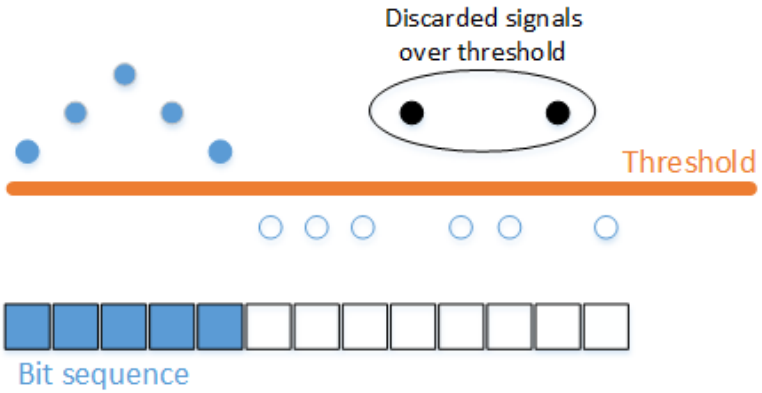



Figure 4.4: Difference between a valid and invalid signal sequence.

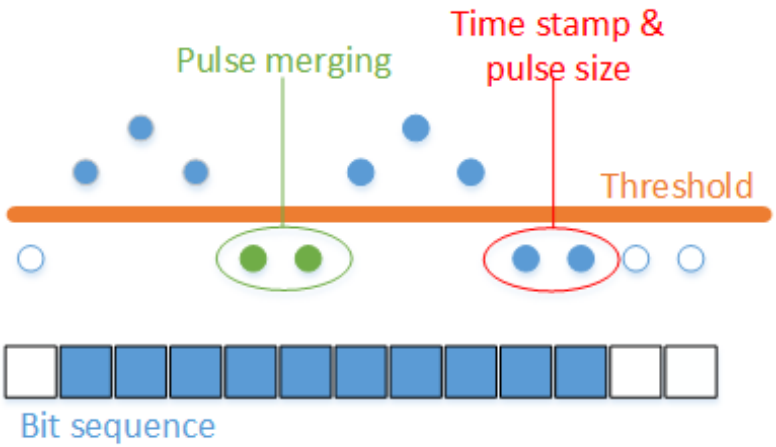



Figure 4.5: Merging of two pulses and the storing of extra pulse information.

In some later discussions regarding the upgrade there have been questions if the described method is insufficient. The theory behind the discussion is that the baseline will shift too much to be able to do efficient Zero Suppression without loosing important samples in the process. Another argument against Zero Suppression is that with time frames with larger occupancies (40%++) the compression factor is drastically reduced and will not be good enough. This is because time frames with higher occupancy will have more signal pulses, and pulses are closer together, meaning that more pulses are merged rather than discarded. This encouraged finding another way of processing the signals. One proposed method was to use Huffman encoding on the signal values.

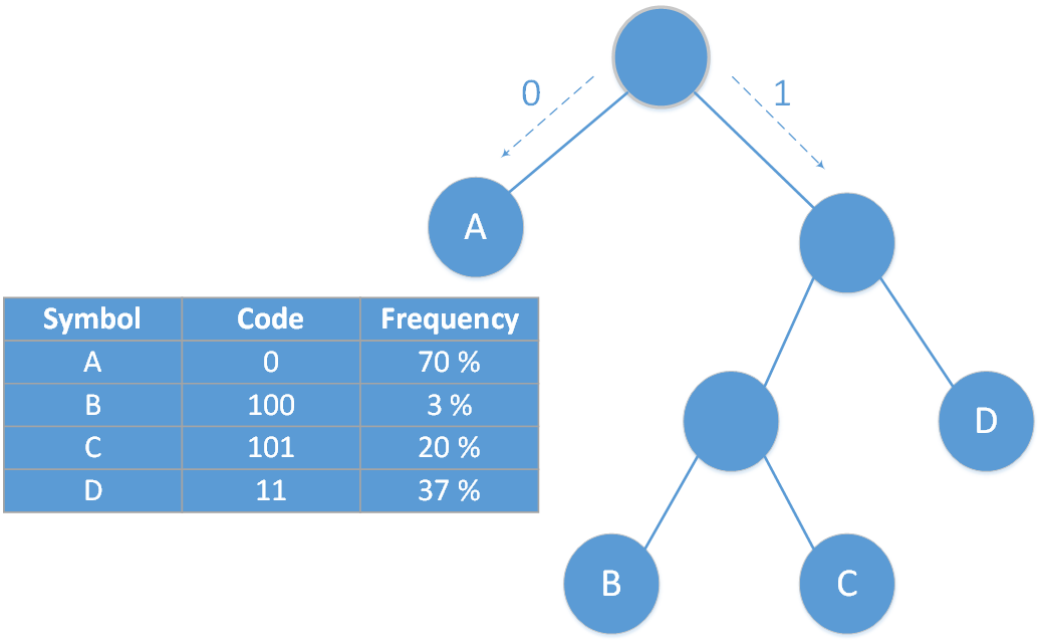
### 4.2.2 Huffman Encoding

Huffman encoding is a method used to achieve data compression[20]. It works by assigning binary codes to symbols in order to reduce the number of bits used to encode the symbol. By looking at the frequency of appearance for every symbol used one can produce a frequency table sorted by most frequent. One thing to note is that since the binary codes are of variable length, they may not all be uniquely decipherable. For instance, if the following codewords are used: {0,01,11,001}, the code 0 is a prefix to 001. This is solved by using the right data structure to store the codes, the one most used is a *full* Binary Tree (BT). A *full* BT is a tree where every node either has zero or two child nodes. The symbols are then generated by the path from the root to a leaf node, where left and right indicates 0 or 1. Figure 4.6 shows an example of a Huffman tree using made up frequencies for the letters A to D. Here you can see the advantage of sorting by frequency, since the most frequent symbol A only needs one bit to store. Creating the Huffman tree can be implemented using the following pseudo-code algorithm sourced form [21]:

```
1  //Input: An array f[1..n] of frequencies
2  //Output: An encoding tree with n leaves
3  //let H be a Priority Queue of integers, ordered by f
4  function Huffman(f) {
5    for(int i = 1; i <= n; i++){
6      H.insert(i);
7    }
8    for(int k = n+1; k <= 2n - 1; k++){
9      i = H.deletemin();
10     j = H.deletemin();
11     //Create a node numbered k with children i,j
12     f[k] = f[i] + f[j];
13     H.insert(k);
14   }
15
16 }
```

**Listing 4.1: Huffman algorithm.**

In the context of compressing data coming from the detector there is one particular foreseen complication. First of all, generating the Huffman tree needs values from the detector, so how does one create a tree with high compression factor without having these? One answer to this is to generate a tree based on existing data from previous experiments, then update the tree when receiving new data. This gives us an uncertain compression factor




| Symbol | Code | Frequency |
|---|---|---|
| A | 0 | 70 % |
| B | 100 | 3 % |
| C | 101 | 20 % |
| D | 11 | 37 % |

Figure 4.6: Huffman tree with four symbols.

in the beginning, but it will become better over time. Because of a shifting baseline encoding the signal values directly may lead to a large Huffman tree, and the best tree for one channel may not be the same for another. It is inefficient to create a separate tree for each channel, as there will be 160 channels for every FEC. A possible solution to this is to encode the derivative of each signal in a time frame compared to the previous value. In other words, for every signal *n*, you store the value: *signal(n) - signal(n - 1)*. Doing so resolves the problem caused by the shift in the baseline as it only stores the difference between two signals. This method requires that the first value of every time frame is stored somewhere(maybe the header of a SAMPA packet) in order to decode it later.

The way the FIFO buffers for each SAMPA channel works is that it stores up to 10 bits in parallel for each slot. This means that compressing 10 bit samples into smaller sizes will still take up 10 bit of space in the buffers. However reading the data from the buffer is faster as there is less data to read.

## 4.3 Designing the Simulation Model

With all of the information regarding the different components already specified, creating a simulation model should be more than feasible. There will be a total of 3 main modules in the simulation: the SAMPA, GBTx and CRU, but the thesis focuses on the SAMPA, leaving the GBTx and CRU mostly empty as what happens after data leaves the SAMPA is outside the scope. In addition to the different modules there is need for a module, which can be tasked with producing and/or distributing sample data to the simulation. This module contains all the methods for sending samples to the different SAMPA channels and in doing so start the entire simulation process. The tasks, objectives and goals are summarized in the list below.

- **Tasks**
  - Designing a model that is accurate, simple and customizable.
  - Creating a data generator module that can send data to the simulation, both synthetic and real.
  - Create a simulation test bench that allows for quick changes in order to run multiple simulations.
  - Run different stress tests on the system, find out where it breaks and why.
  - Run focused simulations on the SAMPA channel buffers.
  - Run simulations that compare Zero Suppression and Huffman encoding.
  - Gather, and compile the simulation data into a readable and understandable format.
  - Verify that the simulation results are comparable to what is expected, and calculated beforehand.
- **Goals**
  - With a verified simulation model, we have a created a strong argument that the results are valid.
  - Find out how much SAMPA buffer space is needed.
  - Conclude the compression factor of both Zero Suppression and Huffman encoding.
  - Verify the overall design of the SAMPA chip, and use the results to come with a recommendation on possible changes.

## 4.4 Workflow

Approaching this project, one must assume that there will be many uncertainties along the way. Trying to simulate behaviour of an electronic system based solely on its early schematics, while others are working on the design in different areas will undoubtedly lead to many changes in the simulation model. Another characteristic concerning this project is that it requires a lot of work before one can start to see any results, but after completing a satisfying model the results should be easy to obtain without many changes to the simulation program. Splitting the work into different phases, first a longer period of only working on the model, implementing the aspects that are known, and making the model ready to run simulations on. When the base model is complete, an iterative process can start. Simulate for a specific scenario, gather results from the simulation, compile it into a readable format, verify the correctness of the results, in the case they are not legitimate, make adjustments before running new simulations in the same scenario. Customize the simulation parameters and tweak the model for different scenarios, and do the same as before. This way any changes in requirements, or changes to the model can be handled in a separate iteration. Working like this will result in a large period with no speakable results, but this will towards the end be very beneficial.

# Chapter 5

# Solution Implementation

## 5.1 Implementing the Model in SystemC

With the design given in the last chapter, the simulation model can be implemented in SystemC. The implementation was a product of an iterative process, and as such it evolved over time. This chapter will give some insight into the early implementation, but focus mostly on the final product. The thought process behind the implementation choices will be discussed and there will be shown snippets of code when it is deemed beneficial in order to understand the implementation.

### 5.1.1 The SAMPA Module

As the focus of study in this project, the implementation of the SAMPA module is the most important piece to the simulation. The overall structure of the SAMPA consists of 32 channels, with an input port for each channel, and in total 4 serial outputs which reads data from the channel buffers. There are a couple of things to think about when translating this design into code.

**What SystemC channel should the input and output ports use:**
The requirements for the SAMPA I/O ports is that everything comes in the correct order, and on a specific clock cycle. SystemC comes with the channel type `sc_fifo`, it contains both read and write methods, depending on what channel interface is implemented. So for our one directional design this should work perfectly. The clock cycle is not tied to the ports specifically and will be handled separately.

**What data structure to use for the channel buffers:**
When choosing a data structure one needs to think about what the

purpose of it is, what operations are being done on it, and so forth. The essential attributes the structure must have are: *Insert* items to the back, *Read/Remove* items from the front, dynamical storage space, and the structure should be a linear one-dimensional sequential storage. At first glance using a FIFO like structure appears to be the best way to go. However in addition to the essential attributes it may be needed to be able to remove and read from the back of the buffer. This is important, as it allows the system to grab statistical data from the buffer, and read operations won't have any impact on the simulation result, but instead can make the buffers more versatile. C++ has many different data structures to choose from. In Table 5.1 three different C++ data structures are evaluated: `vector`, `list` and `queue`. From this table and the requirements of what is needed from the buffer structure, it becomes clear that the `list` container has all the attributes needed, as well as performing equally or better than the rest in the different operations.

**Handling the clock frequency:**
SystemC will handle the clock frequency for us, the only thing to note is that SystemC uses pauses in the threads as a way to simulate the clock cycles. In other words, one performs the actions for one clock cycle, then executes the wait statement, and repeat. This means that the frequencies need to be converted to a time delay. An example of such a conversion is shown in Listing 3.2.

The setup for the SAMPA uses a single thread for receiving signals, and four individual threads which represents the serial outs. It also stores the channel header and data buffers in two separate arrays, declaring input ports, and the output e-links. Let's look at what the implementation for receiving signals might look like. In Listing 5.1 the thread logic is implemented, and already some weaknesses with the SAMPA module becomes clear. Having to iterate over every channel every time bin can become costly and hard to maintain when the code becomes more complex. In the code shown there is already a flaw, if one of the channel buffers has `overflow`, none of the other buffers will receive data. The `overflow` variable needs to be stored as an array to be able to know what channel it represents. The same problem will occur for every other variable that is unique for each channel. A principle of Object-Oriented Programming (OOP) is single-responsibility, meaning that every `class`/object should be responsible for one piece of functionality[25]. One solution to the previous problem following this principle can be to create

a `Channel` sub-module inside of the SAMPA module. The `Channel` module will contain logic for receiving signals, as this happens for every channel, while the SAMPA will contain the serial outs, which are accessing data from the buffers.

| | Time | | | |
|---|---|---|---|---|
| **Operation** | **Vector** | **List** | **Queue** | **Remarks** |
| Add back | O(1) | O(1) | O(1) | Constant time for all containers. |
| Add front | O(n) | O(1) | X | Vector does not have a direct method for adding to front. Queue can not do that at all. |
| Access back | O(1) | O(1) | O(1) | Constant time for all containers. |
| Access front | O(1) | O(1) | O(1) | Constant time for all containers. |
| Remove front | O(n+m) | O(1) | O(1) | Vector erase is linear to number of deleted elements + number of elements after last deleted item (moving). |
| Remove back | O(1) | O(1) | X | Queue does not have a method for doing this. |
| Size of container | O(1) | O(1) | O(1) | Constant time for all containers. |

Table 5.1: Data structure comparison[22], [23], [24].

```
1  void SAMPA::receiveSamples(){
2
3    while(true){
4      for(int i = 0; i < NUMBER_OF_CHANNELS; i++){
5        if(inputPorts[i].nb_read(sample)){
6
7          if(dataBuffers[i].size < MAX_BUFFER_SIZE && !overflow){
8            dataBuffers[i].push_back(sample);
9            currentTimebin++;
10         } else {
11           overflow = true;
12         }
13       }
14       //End of timeframe
15       if(currentTimebin == MAX_NUMBER_OF_TIMEBINS){
16         Packet packet(timeFrame, i, dataBuffers.size(), overflow
               );
17         headerBuffers[i].push_back(packet);
18       }
19     }
20     wait(waitTime, SC_NS);
21   }
22 }
```

**Listing 5.1: Sampa receive thread.**

The corrected overall structure for the SAMPA, and the new `Channel` module is shown in Appendix A. As seen there the SAMPA now has an array of `Channel` modules, a new method called `initChannel()` that initiate the channels, and connects them to the correct input port. The `Channel` module now has the `receiveData()` thread, and its own data/header buffer. Having it structured like this makes it possible to add `Channel` specific variables or methods, without disturbing the SAMPA as a whole. The implementation of the receive thread is now more simple in terms of complexity, and is exclusive for each `Channel`. Listing 5.2 shows the basic thread structure, excluding any data compression or processing. It continuously receives samples and adds them to the buffer, unless the buffer reaches its maximum size.

```
while(true){
    if(port_DG_to_CHANNEL->nb_read(sample)){
      numberOfClockCycles++; //timebin

      //Check if max buffer size is reached.
      if(dataBuffer.size() + sample.size >
          CHANNEL_DATA_BUFFER_SIZE){
        overflow = true;
      }

      //Add sample to buffer if there is no overflow.
      if(!overflow){
        addSampleToBuffer(sample, numberOfClockCycles);
      }
    }
    //End of timeframe
    if(numberOfClockCycles ==
       NUMBER_OF_SAMPLES_IN_EACH_TIME_FRAME){

      //Remove samples added earlier in timeframe if overflow
      if(overflow){
        for (int i = 0; i < numberOfSamples; ++i){
          dataBuffer.pop_back();
        }
      }
      //Create header packet, and add to header buffer.
      headerBuffer.push(header);
    }
    wait(constants::SAMPA_INPUT_WAIT_TIME, SC_NS);
}
```

**Listing 5.2: New version of the SAMPA receive thread now implemented in the Channel sub-module.**

The implementation of reading from the buffer is somewhat more complicated than receiving data. This is because the real reading procedure is more complex in itself. In Listing 5.3 the most important parts of the code implementation is shown. The procedure loops through the eight channels for the given serial out, getting the correct channel, and reading out the entire time frame. This procedure is run for all four serial out threads at the same time. One thing to notice here is that the actual data from the buffers is not passed along further. This is because we only care about the time it takes to transfer it, which is being calculated in the `wait` statement. The header packet is sent as it contains important information which can be useful later on.

```
1  void SAMPA::processData(int serialOut){
2
3    //Go through all channels for specific serialout
4    for(int i = 0; i < CHANNELS_PER_E_LINK; i++){
5      float waitTime = 0.0;
6      //Find channel
7      int channelId = i + (serialOut*CHANNELS_PER_E_LINK);
8      Channel *channel = channels[channelId];
9
10     //find header
11     if(!channel->headerBuffer.empty()){
12       Packet header = channel->headerBuffer.front();
13       channel->headerBuffer.pop();
14
15       //Read from databuffer, but check for overflow.
16       if(!header.overflow || header.numberOfSamples > 0){
17         for(int j = 0; j < header.numberOfSamples; j++){
18             channel->dataBuffer.pop_front();
19         }
20       }
21       //Simulate number of clock cycles it took to read the
             timeframe.
22       waitTime = (5 + header.numberOfSamples); //50bit header +
             10 bit samples
23       porter_SAMPA_to_GBT[serialOut]->nb_write(header);
24       wait((SAMPA_OUTPUT_WAIT_TIME * waitTime), SC_NS);
25     }
26   }
27 }
```

**Listing 5.3: Reading data from the SAMPA buffers.**

### Implementing Zero Suppression

Now that the input channels, and the serial outs are in place, the data processing can be implemented. The Zero Suppression occurs before the samples are added to the channel buffers, so the correct place to put it in the simulation will be in the `Channel` module's `receiveData()` method. Since this is a piece of code which needs to be turned off and on, the implementation is done in its own method called `zeroSuppress()`. The method needs the current sample, the last received sample, and some meta-data about the current behaviour of the time frame. The rules for the Zero Suppression is described in detail in Section 4.2.1, therefore this section only looks at the implementation.

By looking at the two last samples the `Channel` received, the Zero Suppression algorithm can do one of four things depending on if the samples are over or under the Zero Suppression threshold. In code this translates to four different conditional statements, in this case the implementation uses `if-else` statements, as seen in Listing 5.4.

```
1  if(sample.data > 0 && lastSample.data > 0){
2    //Sample is part of a cluster, add it to the fifo
3  } else if(sample.data > 0 && lastSample.data <= 0){
4    //Start of new cluster. Add sample, but remove if next sample
       is invalid.
5  } else if(sample.data <= 0 && lastSample.data <= 0){
6    //If sample is part of cluster meta-data add to fifo.
7  } else if(sample.data <= 0 && lastSample.data > 0){
8    //If last sample was part of valid cluster, add current sample
       .
9    //If not, last sample should be removed from the buffer.
10 }
```

**Listing 5.4: If-else structure for the Zero Suppression algorithm.**

When both of the last samples are over the threshold, the algorithm knows for sure that a valid cluster is found, and marks it as such. If the current sample is valid, but the last one is not, then it is not clear if the sample is part of a valid cluster, so it is added to the buffer, but marked as invalid for now. If the last two samples were both invalid, it means that either they are just noise, they can be a part of the last valid cluster, or if the next sample is part of a cluster, they can be merged together, forming a single cluster. Most of the time the last condition will be very straight forward, but there exists one edge case which makes it more complicated to implement. The edge case occurs when the last couple of signals has a particular shape. The problematic sequence is displayed in Figure 5.1.

Now the last sample, which was added to the buffer turned out to be an invalid cluster, and needs to be discarded. The algorithm must then remove it, and replace it with an empty sample as part of the previous clusters meta-data. The final implementation of the Zero Suppression algorithm is shown in Listing A.3. Since the code was implemented in its own method, the only change needed in the `receiveData()` thread is to call the `zeroSuppress()` method instead of adding samples directly to the buffer.

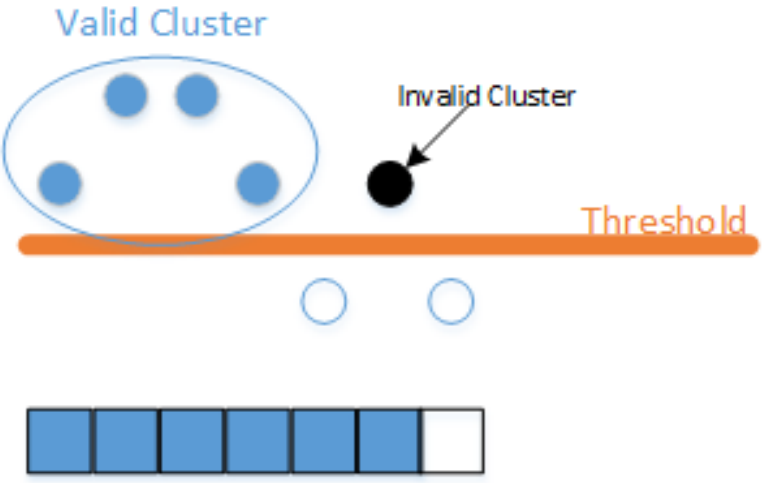



Figure 5.1: An edge case that the Zero suppression needs to handle.

### Implementing Huffman Encoding

Huffman is a well known, and widely used algorithm. Because of this, there are multiple providers which has made the implementation available online for people to use. The Huffman code used in the simulation is based on code provided by Rosetta Code, licensed under GNU Free Documentation License 1.2[26][27]. The code is modified slightly to suit the requirements of the simulation program. In addition methods for writing a full Huffman tree to a file, as well as reading a tree from a source file is implemented.

The simulation will only be able to use the Huffman encoding when it uses realistic data, i.e where the samples have realistic values. Only when there is a wide spectre of different values available can the simulation determine an accurate compression factor. There are two different sets of realistic data available for use, and they are described in more detail in Section 5.1.2. The Huffman tree is created directly from the dataset that is used, and saved to a file so that it can be used to decode the samples later on. Creating the tree in such a manner means that it gets the optimal compression possible. In the real world a perfect compression is not plausible as signals coming from the detector will continuously change, and it does not contain a steady pattern. Even though using the optimal compression in the simulation does not give us the worst case, it can still establish a base estimate that is reasonable.

### 5.1.2 The DataGenerator Module

The simulation model is scoped to the readout electronics, which does not handle the creation of signals. This means that the simulation needs a module which can create and distribute data, both simulated and real to the SAMPA modules. The module needs to be able to continuously send samples based on what is desired for a specific simulation scenario. It is important that all the different methods for sending samples, do so in the same fundamental way. The rest of the simulation model should not need to change for each different scenario, but instead the data generator should make sure that it sends samples in the correct format, regardless of the simulation type.

With the expected behaviour given in the previous paragraph, the bounds and requirements of the data generator can be established. The module will continually be updated, and new functionality will be added as new simulation scenarios surface. In order for this to be efficient, the module needs to be easily extended, without causing any disturbance to the rest of the module. Sending samples should follow the specifications for the actual hardware, emulating the connection between the readout chambers and the SAMPA ASIC. That includes the clock frequency and making sure every channel receives data asynchronously.

#### Basic Data Generating Functions

Achieving the same format for every simulation sample is done by using a custom `class` which forms the link between the data generator and what the `Channels` are expecting as input. This means that every time the data generator sends a signal it uses this `class`. The format of the `Sample class` is discussed and shown in Section 5.1.3. Creating a single function that has the core functionality of data generator, which all simulation scenarios uses as base could have been a good way to remove some of the boilerplate code needed, but without knowing all types of simulations from the start this could lead to unwanted restrictions later on. Because of this, for every different way of creating/distributing data, there is a completely different function. The benefits of implementing it like this is that the functions do not depend on each other and can be separately updated or improved, which allows for easier development as the module becomes larger and more complex.

To be able to quickly switch between different functions when running different simulations, the module contains only a single SystemC thread, where it is possible to choose the correct function based on the simulation

type. The `sink_tread()` function can be seen in Listing 5.5. In Listing 5.5 the variable `DG_SIMULATION_TYPE` is used. This variable is stored in what is referred to as the simulation test-bench. Throughout the following code listings many variables from the test-bench are used. In short the test-bench is a `class` which defines a global `namespace` where every shared variable or property for all modules that are part of the simulation is stored. The test-bench makes it very easy to run multiple different simulations in quick succession. In most cases this will be the only file which is modified in between simulations, only changing the properties the simulation uses.

```
1  void DataGenerator::sink_thread(void){
2    if(DG_SIMULTION_TYPE == 1){
3      standardSink();
4    } else if(DG_SIMULTION_TYPE == 2){
5      incrementingOccupancySink();
6    } else if(DG_SIMULTION_TYPE == 3){
7      alternatingOccupancySink();
8    } else if(DG_SIMULTION_TYPE == 4){
9      sendBlackEvents();
10   } else {
11     sendGaussianDistribution();
12   }
13 }
```

**Listing 5.5: Data generator SystemC thread.**

They were implemented one by one in different stages of the development phase. The first three: *standardSink()*, *incrementingOccupancySink()*, and *alternatingOccupancySink()* are functions created in the early stages, before there was any need to implement compression schemes in the simulation model. They all assume that the data being generated is already compressed, and can be sent through the simulation without any data processing. The sheer amount of samples is the important factor with these functions, relying on a flat occupancy value to determine if a sample is sent. The chosen occupancy(specified as percent(0-100)) is compared against a number generated by a Random Number Generator. If the number is lower than the occupancy the sample is sent, otherwise it sends an empty sample instead. In other words, a sample is sent a fixed percent of the time, specified by the occupancy. The *standardSink()* implementation is shown in Listing 5.6. The double loop seen here is similar for all five functions, where the outer one counts the number of time frames and the inner goes through all SAMPA channels present. The wait statement is called after the inner loop, which is

equivalent to sending samples to all channels in the same clock cycle. This function is a good starting point in order to test the rest of the simulation model, and to verify that the model can be trusted. Since it relies on a flat occupancy it is expected that most channels will get similar data, and there will be little to no fluctuations. Real experiment data will not be as flat, and the occupancy will differ from time frame to time frame.

```
1  void DataGenerator::standardSink(){
2    //While we still have timeFrames to send
3    while(currentTimeFrame <= NUMBER_TIME_FRAMES_TO_SIMULATE)
4    {
5      //Loop each channel
6      for(int i = 0; i < NUMBER_OF_SAMPA_CHIPS *
           SAMPA_NUMBER_INPUT_PORTS; i++)
7      {
8        if(randomGenerator.generate(0, 100) <= DG_OCCUPANCY){
9          //Send real sample
10       } else {
11         //Send empty sample
12       }
13     }
14     currentSample++;
15     //Increments timeWindow
16     if(currentSample == NUMBER_OF_SAMPLES_IN_EACH_TIME_FRAME )//
           1021 samples
17     {
18       currentTimeFrame++;
19       currentSample = 0;
20     }
21     wait((constants::DG_WAIT_TIME), SC_NS);
22   }
23 }
```

**Listing 5.6: Data generator SystemC thread.**

The limitations of the *standardSink()* function was the motivation behind the *incrementingOccupancySink()* and the *alternatingOccupancySink()*. As their names suggest they implement more diversity into the simulation with the ability to increase, or alternate the data occupancy over time. This creates more possibilities to test the limits of the model, especially the SAMPA buffers. The code for these functions is similar to the *standardSink()*, and the differences are shown in Listing 5.7 and Listing 5.8.

```
1  //Standard sink
2  if(currentSample == NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW )
3  {
4    currentTimeWindow++;
5    currentSample = 0;
6  }
7
8  //Increasing occupancy sink
9  if(currentSample == NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW ){
10   //How often do we increase the occupancy?
11   if(currentTimeWindow % (NUMBER_TIME_WINDOWS_TO_SIMULATE /
         TIME_WINDOW_OCCUPANCY_SPLIT) == 0){
12     //Increase occupancy.
13     occupancy = occupancy + TIME_WINDOW_OCCUPANCY_SPLIT;
14   }
15   currentTimeWindow++;
16   currentSample = 0;
17 }
```

**Listing 5.7: Difference between *standardSink* and *incrementingOccupancySink* functions.**

```
1  //Standard sink
2  if(randomGenerator.generate(0, 100) <= DG_OCCUPANCY){
3    //Send real sample
4  } else {
5    //Send empty sample
6  }
7
8  //Alternating occupancy sink
9  if(randomGenerator.generate(0, 100) <= occupancyPoints[
       currentTimeWindow - 1]){
10   //Send real sample
11 } else {
12   //Send empty sample
13 }
```

**Listing 5.8: Difference between *standardSink* and *alternatingOccupancySink* functions.**

The *incrementingOccupancySink()* will provide initial results on how much occupancy the system is able to handle, while the *alternatingOccupancySink()* tests its ability to handle various amount of data distributed over time. This is an improvement from the *standardSink()*, but the functions do still not take into account how the data is shaped or distributed inside a time frame. To get more accurate results from the simulation there is a need for data

which more closely resembles actual experiment data. The implementation so far does not worry about the value of the samples, but the values are required in order to test Zero Suppression and Huffman encoding.

### Creating Normally Distributed Samples

Measured occupancies gathered from experimental events recorded in 2010 provides a graph of expected average occupancies within a time frame. This is shown in Figure 5.2, where the x axis is the position of a pad row on the actual detector. The closer to origin, the closer the pad is to the center of the detector. The pads closer to the center generally have higher occupancy than the outer ones, which makes them the worst case, and the most interesting for use in the simulation. Figure 5.2 shows that the average occupancy for the inner pads is 28%, but it can reach 44%, or with a high multiplicity as high as 74%. The chance of reaching a multiplicity equal to 44% occupancy is 10%, while there is only 1% chance to reach 75%. Based on this a normal distribution of occupancies can be created for a more realistic simulation. A normal distribution is a statistical distribution of values which depend on a central value (the mean value) and a stretch factor (the standard deviation)[28]. The formula for a normal distribution is as followed:

$$f(x, \mu, \sigma) = \frac{1}{\sigma}\phi(\frac{x-\mu}{\sigma}) \tag{5.1}$$

Where $\mu$ is the mean value and $\sigma$ the standard deviation. An example of a normal distribution is shown in Figure 5.3

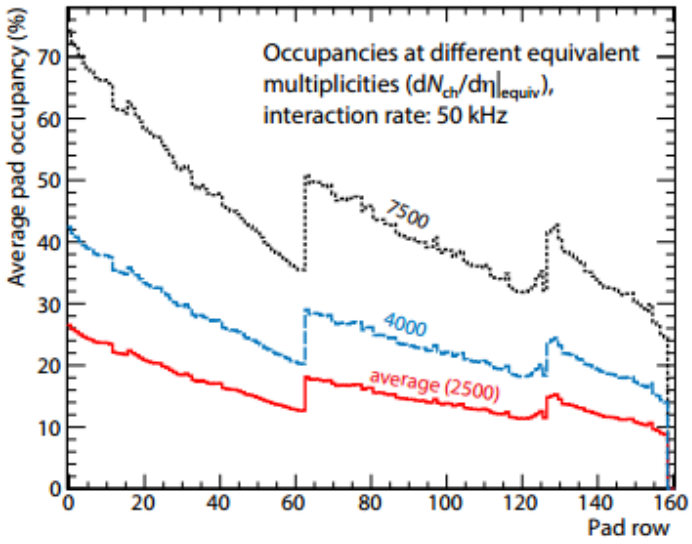



Figure 5.2: Expected average occupancies within a given time frame [14].

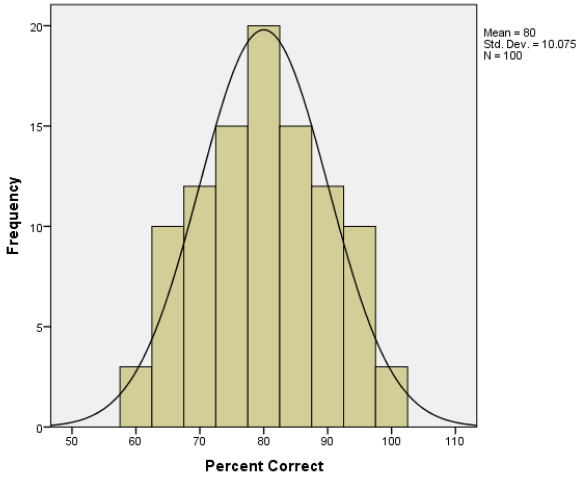



Figure 5.3: Example of a normal distribution [29].

Using a normally distributed Random Number Generator, a set of occupancy values can be created. For every new time frame in the simulation, a random value from the set is picked as the current occupancy. The size of the set is determined by the number of time frames in the simulation. The mean value for the distribution is already known is 28%, but the standard

deviation is not obvious and needs to be calculated. Using the statistics given in Figure 5.2 the deviation can be calculated. In a set $x$ of 100 values, 1 value will be 74%, 10 will be 44% and the rest will on average be 28. First of the *variance* must be calculated by the following formula:

$$\frac{\sum_{i=1}^{100}(x_i - mean)^2}{100} \tag{5.2}$$

Take the square root of the *variance* and the result is the standard deviation. How this is implemented programmatically is shown in Listing 5.9 along with creating the final set of occupancies.

```
1  std::vector<int> DataGenerator::getOccupancy(){
2
3    array[0] = 74;
4    for(int i = 1; i <= 10; i++){
5      array[i] = 44;
6    }
7    //+-10 for a wider distribution.
8    for(int i = 11; i < 100; i++){
9      array[i] = generator.generate(mean-10, mean+10);
10   }
11   for(int i = 0; i < 100; i++)
12     sum += pow(array[i] - mean, 2.0);
13
14   double deviation = sqrt(sum/100);
15   std::default_random_engine gen(seed);
16   std::normal_distribution<double> dist(mean, deviation);
17
18   //create a vector of occupancies based on the normal
        distribution.
19   for(int i = 0; i < NUMBER_TIME_FRAMES_TO_SIMULATE; i++){
20       result.push_back((int)dist(gen));
21   }
22   return result;
23 }
```

**Listing 5.9: Data generator SystemC thread.**

After testing the distribution, the conclusion was that the distribution became very narrow towards the mean. Using values from $mean - 10$ to $mean + 10$ gave a bigger deviation, and as a result a wider distribution. A wider distribution will most likely create more interesting results. The difference by increasing the deviation is shown in Figure 5.4.

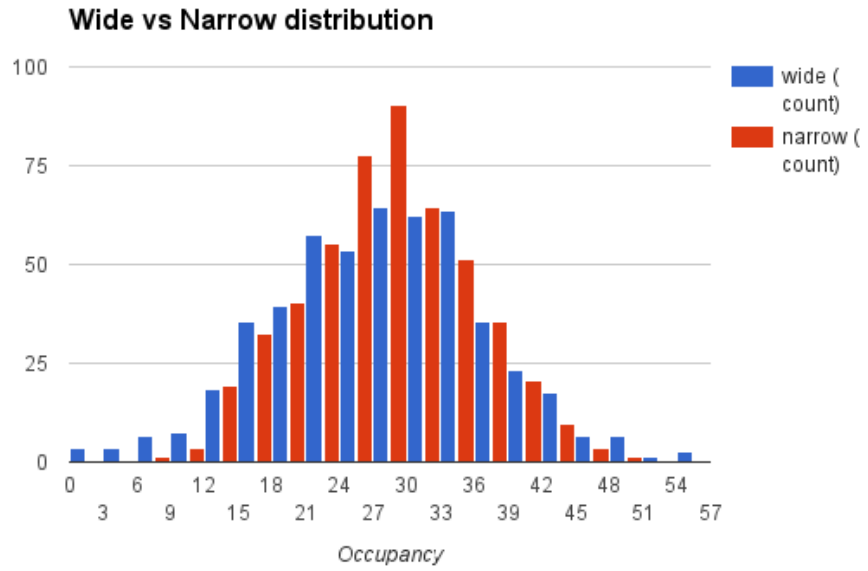



Figure 5.4: Difference in the normal distribution.

To be able to test the Zero Suppression the samples need to have a shape which allows for this. The Zero Suppression works best with data where the peaks are wider, but further apart from each other. So to test it in its worst case, the samples need to have as many peaks as possible. A peak will be of width 3, and the occupancy for the time frame determines the space between two peaks. An example using 28% occupancy is displayed in Figure 5.5. Even though the data still does not look like the real thing, in the case of testing the Zero Suppression it should suffice since it only cares if the sample value is zero or above.

A process for determining the space between each peak is explained next. The space can be calculated by first finding the number of samples with value 0, and divide it by the number of peaks in the time frame. Since both the occupancy and the number of samples in a time frame is known, the math becomes very straight forward. The solution used is shown in Listing 5.10.

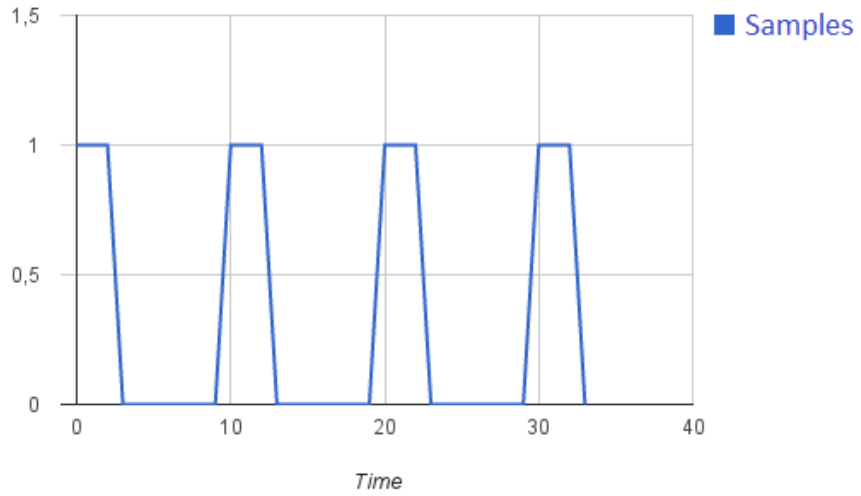



Figure 5.5: Shape of the normally distributed data.

```
1 double DataGenerator::calcSpace(int occ){
2   double numberOfSamples = (NUMBER_OF_SAMPLES_IN_EACH_TIME_FRAME
        * occ) / 100.0;
3   double numberOfPeaks = numberOfSamples / 3.0;
4   double numberOfEmptyTimebins =
       NUMBER_OF_SAMPLES_IN_EACH_TIME_FRAME - numberOfSamples;
5
6   return (numberOfEmptyTimebins / numberOfPeaks);
7 }
```

**Listing 5.10: Calculating the space between two peaks in a time frame.**

With a completed set of occupancies and the necessary functions to help shape the data the only thing left is to implement the sink function. It uses the same loop structure as the three previous sink functions, but in addition before every new time frame it picks the next occupancy from the set. Then using the current occupancy to calculate the number of empty samples between the peaks, it starts sending the samples in the correct order.

### Realistic Datasets

The last type of simulation uses realistic datasets as input. Two types of datasets have been available for use: Black event data collected from RUN 1, and simulated data created by researchers which matches the expected data from RUN 3. Black events is the term for when there is data in all channels at the same time, which means overall higher occupancy than normal events.

In addition to this, we want to use the data from RUN 1 to create pileup data. Pileup occurs when a channel gets overlapping data from multiple neighbouring channels in the same time frame. In the worst case the pileup can happen five times, i.e samples from five neighbouring channels are causing interference. By overlapping five time frames from the RUN 1 data, a third set for use in the simulation is made available. For the sake of clarity the datasets will be noted as: black-events, simulated RUN 3 and pileup data from now on. The format and an example for the black-events and the pileup datasets are as follows:

```
1  //Format
2  ddl <ddl number>
3  hw  <hardware addr>
4  <start time> <time frame length>
5  <time bin> <signal>
6  <time bin> <signal>
7  ....
8  hw <hardware addr>
9
10 //Example
11 ddl 12
12 hw 201
13 1021 1007
14 1021 49
15 1020 50
16 ....
17 hw 123
```

**Listing 5.11: Format for the black-event and pileup dataset.**

The format shown in Listing 5.11 displays several variables. The ddl number, the hardware address, the start time for the time frame, the length of the time frame, and the sample data identified by the time bin and signal. The hardware address is a decimal number that represents the channel, the ALTRO chip, the front-end card and branch address. Even though it is displayed as a decimal, it can be translated into four different values using bitwise operators. The bitwise operations are as follows: $BranchNr << 11|FecNr << 7|AltroNr << 4|ChannelNr$. In Figure 5.6 an example address is shown, displayed in binary and the bits representing each of the four values are highlighted. In this context the ddl number is not important, but ddls with low numbers usually have a higher than average occupancy among their channels.

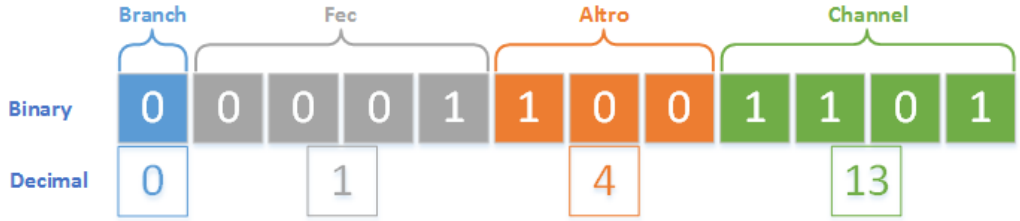



Figure 5.6: Example hardware address(205) in binary, with translated addresses below.

To get the different values from the original hardware address, the opposite bit operation than the one used to store it is applied. Taking the *BranchNr* as an example. Its value is stored by taking the value and left shifting it 11 bits, so in order to reverse it all that is needed is to right shift the hardware address 11 bits.

```
1 int BranchNr = (hw >> 11);
2 int FecNr = (hw >> 7) & 15;
3 int AltroNr = (hw >> 4) & 7;
4 int ChannelNr = hw & 15;
```

**Listing 5.12: Bitwise operation to retrieve values from the hardware address.**

The formatting used for the simulated RUN 3 data is more straight forward than the one for black-events. It starts with one row with the title of values in its column, before continuously storing rows of data, following the format specified in the first row. The main difference between the two formats is that the simulated RUN 3 dataset stores the exact pad and pad row for every time bin, instead of storing it before the start of every time frame, as in the black-events.

```
1 #sector #padRow #padNr #time bin #signal #meta-data
```

**Listing 5.13: Format for the simulated RUN 3 dataset.**

The next step is creating a shared data container which can store samples from any dataset, this way the simulation only needs one sink function. The data container needs to store samples for every channel, and for every time frame that is being simulated. This can be made by defining a simple `template` using existing containers.

```
1 typedef std::map< int, std::list<Sample> > DataEntry;
2 typedef std::vector< DataEntry > Datamap;
```

**Listing 5.14: Data container.**

The `DataEntry` template stores one entire time frame worth of samples for every channel in the simulation. The `Datamap` is just a `vector` storing `DataEntry` objects. One weakness of storing the samples in this manner is that the sink function will now expect that it contains full time frames for all channels. In other words, it expects that the datasets contain 1021 signals for every channel, which is not the case. The black-event data was based on the ALTRO chip, meaning there are only 8 chips per front-end card, and 16 channels per ALTRO. At the same time the time frame seems to be shorter, so there is always less than a full time frame worth of data. The lack of samples can easily be fixed by placing empty samples at the front and/or back of each time frame. Doing so will create full time frames no matter the actual number available. Regarding the channel mismatch in the black-event data, it can be remedied by creating a mapping table, using neighbouring channels to fill in the gap. Even though there are enough channels to work with, the dataset still only contains three time frames worth of data for each channel used in RUN 1. Three time frames are not enough to create valid results, which means that data from other channels needs to be used as well to create longer simulations. Since the current solution still proposes to use a fake mapping between ALTRO and SAMPA channel, why not read in data without caring about which channel it belongs to. Doing so will provide more than enough time frames, and since these are black events, the occupancy should be on average the same.

The simulated RUN 3 data has a mapping which is usable for the SAMPA channels, but it only contains 1000 signals per time frame. As with the black-event data, the lack of samples is solved by placing empty ones where it is needed. There is more than enough simulated RUN 3 data to use, but since it stores time frames for every single pad in the readout chamber reading out a meaningful amount of time frames for the same channels will be very time costly. The only alternative is to disregard the mapping, and focus on reading in data instead.

With three different data sinks that use a flat occupancy, a more advanced data model using normal distribution, and three different sets with realistic data, the simulation is sure to provide some interesting results. The different simulation types and sink function being used are all displayed in Figure 5.7.

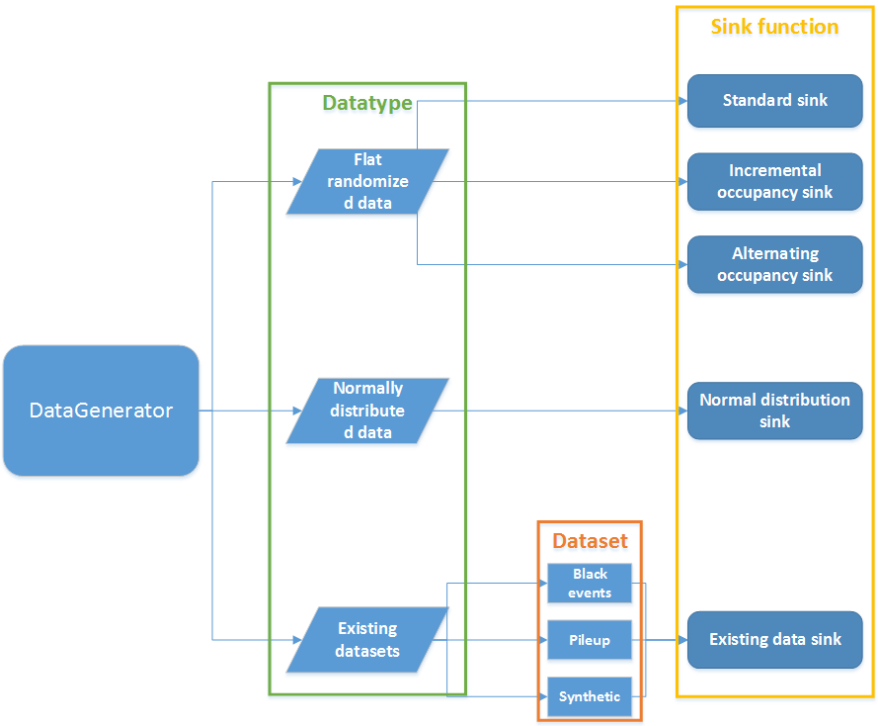



Figure 5.7: Overview of the different data sources and sink functions in the DataGenerator.

### 5.1.3 Signal Classes

SystemC allows the creation of custom `classes` to be used as data types when transferring data between modules. There are some requirements when doing so that needs to be fulfilled for it to work. SystemC requires you to define several methods which is vital for the read/write methods of a SystemC channel. The read/write methods involve copying the custom data type. Because of this it requires the definition of the assignment operator(`operator=()`). In addition the output streaming(`ostream& operator<<()`) method is required.

The simulation uses two different data types, these have been briefly shown in the previous code listings. The `Sample` and the `Packet` classes. `Sample` represents a single 10-bit signal, storing information about what time frame the sample belongs to, the signal value itself, and other statistical variables. Representing the SAMPA header is the `Packet class`, it stores the relevant values selected from its documentation. This include the time frame, channel id, sampa id, number of samples and whether there was overflow in its time

frame. Source code for the `Packet class` is shown in Listing 5.15. The `Sample class` is implemented in a similar fashion and is not shown in the report.

```
1  class Packet
2  {
3  public:
4    //Class variables
5
6    Packet(int _timeFrame, int _channelId, int _numberOfSamples,
         bool _overflow, int _sampleId, int _occupancy);
7    Packet();
8
9    inline friend std::ostream& operator << ( std::ostream &os,
         Packet const &packet )
10   {
11     os << "Packet: time frame: " << packet.timeFrame << ",
           sampaId: " << packet.sampaChipId << ", channelId: " <<
           packet.channelId << ", number of samples: " << packet.
           numberOfSamples;
12
13     return os;
14   };
15   Packet& operator = (const Packet& _packet);
16 };
```

**Listing 5.15: Custom data type - The SAMPA header.**

### 5.1.4 Connecting the Modules Together

As seen in Chapter 3, connecting the modules together is done in the `sc_main` method. The implementation becomes a little more complex when dealing with such a large amount of different modules, as well as sub-modules. In order for the simulation to function correctly the modules must be connected the right way. The connection occurs in three stages:

**Instantiating the modules:**
This is done by simply creating arrays of module objects, iterate through that array and instantiate new objects. The number of modules is specified in the test-bench, which makes it possible to easily edit the number of SAMPA chips, or channels per SAMPA.

**Initialize the FIFO channels:**
An array of FIFO channels must be created for every pair of modules that are connected together. An overview of the number of channels between each module is shown in Figure 5.8.

**Connecting the modules together:**

In this case the DataGenerator needs to have 32 different channels per SAMPA module, the SAMPA needs four channels to the GBTx and from the GBTx to the CRU one is needed. So by having three arrays of FIFO channels, they can be set as channels for the three pairs of modules that needs to be connected. In Listing 5.16 the connection between the SAMPA and the GBTx is used as an example.

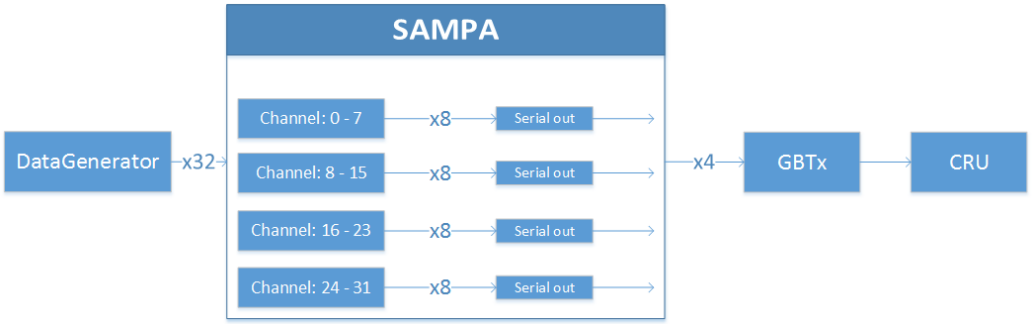



Figure 5.8: Overview of the number of channels between every module.

```
1  for (int i = 0; i < NUMBER_OF_SAMPA_CHIPS *
       NUMBER_OUTPUT_PORTS_TO_GBT; i++)  //8
2  {
3    if (i != 0 && i % GBT_NUMBER_INPUT_PORTS == 0)
4    {
5      gbt_number++;
6      gbt_port = 0;
7    }
8    if (i != 0 && i % NUMBER_OUTPUT_PORTS_TO_GBT == 0)
9    {
10     sampa_number++;
11     sampa_port = 0;
12   }
13   sampas[sampa_number]->porter_SAMPA_to_GBT[sampa_port++](*
         fifo_SAMPA_GBT[i]);
14   gbts[gbt_number]->porter_SAMPA_to_GBT[gbt_port++](*
         fifo_SAMPA_GBT[i]);
15 }
```

**Listing 5.16: Connecting the SAMPA modules with the GBTx.**

## 5.2 Data Gathering

Taking advantage of the fact that the main method stores every module in memory, they can be accessed after the simulation is done, and simulation data stored in them can be gathered. Most of the data gathering happens on a per *channel* basis, so this section will focus on the data gathering from the *channel* module, but the same principle is used for all modules. Every module has `struct` objects or arrays, which store statistical data that has been recorded throughout the simulation time. For the *channel*, the information stored is the current buffer usage for that channel, and some extra meta-data about the time frame, and channel. To avoid large data files, filled with unnecessary information it only records the buffer usage at the end of every time frame. In order to record how many samples make it through the Zero Suppression algorithm, an array is filled every time a sample is added to the buffer when the Zero Suppression is used. When the Huffman encoding is active, the total size for every time frame is monitored and stored as an array.

In the main method after the simulation is done, the different data objects are accessed and iterated, writing their results to a semicolon separated file(.cvs). This file can be processed by any excel like program, and the data can be arranged/structured in a more readable way. In most cases this involves creating graphs and diagrams which visually display the results gathered in the simulation.

# Chapter 6

# Evaluation and Results

## 6.1 Results

### 6.1.1 Verify the Simulation Model

#### Preface

To begin with, a number of different simulations is run, all with one goal in mind: verifying the simulation model. Before using realistic data, the model must be tested to see if it behaves as expected. In order to trust the results provided by the simulations, the simulation model must be dependable, and accurate enough. Using static input data which only depends on occupancy, will give a controlled environment where the results can easily be predicted beforehand and compared to the actual results. Doing some test simulations will also flush out any flaws or weaknesses in the way data is gathered, or if any parameters must be changed.

#### 1. Static Simulations

A good place to begin is to run multiple simulations with different static occupancy level. This will create a wide range of results to use in verifying the simulation model. To begin with, three different occupancy levels were used, 30, 60 and 90 percent. Starting small with only eight time frames per simulation, using the entire FEC, giving a total of 160 channels. The maximum buffer size is set to 4k * 10 bit for the data buffer and 256 * 10bit for the header buffer. The purpose of this simulation is to see how the buffers hold up with the expected data occupancy, compared to higher amounts.

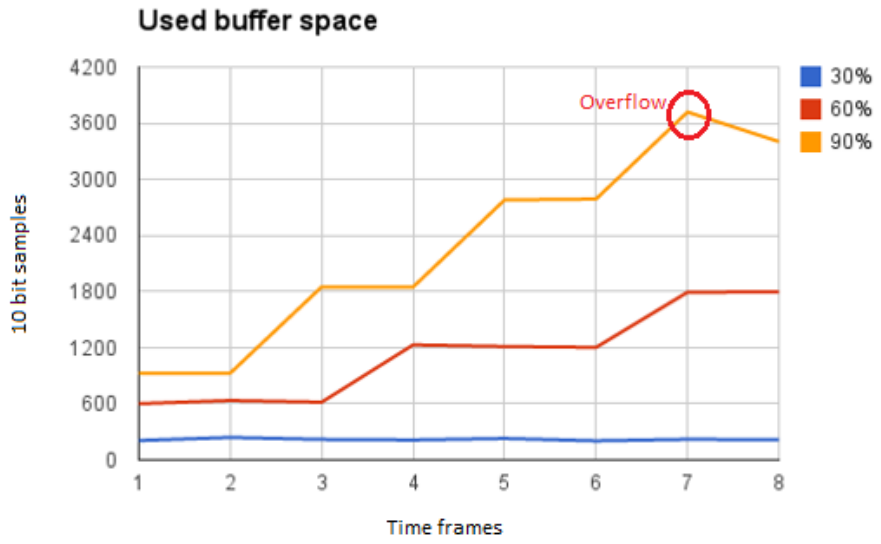



Figure 6.1: Comparing buffer usage for three different levels of occupancy.

Results from the first simulation were very much as expected. Given 30 percent occupancy the buffer usage remained stable at around 250-300 samples over all time frames, while 60 and 90 percent used more space as time went on. After seven time frames the simulation using 90 percent reached the maximum buffer usage, causing it to overflow. From the results one can predict that using 60 percent will also cause overflow, given enough time. To confirm this prediction, a slightly longer simulation using 50 and 70 percent occupancy was performed. This time the simulation used 30 time frames instead of just 8, and the maximum buffer size was set to 8k * 10 bit.

The results in Figure 6.2 confirm the prediction, and it seems that with occupancies higher than 30 percent the buffer usage keeps growing as time goes on, and the serial outs will never be able to read the data fast enough. Looking at the header buffer during these simulations, it became clear that it will most likely never become full, using only one to two header packets at the most. This is because it does not depend on the data occupancy, every header packet is always 50 bit, no matter how many samples there are in a time frame. So far the model seemed to be working as expected, but the simulation had used very controlled, and static data. Some more tests needed to be done in order to confirm that the simulation model was accurate enough to produce valid results with more realistic input data.

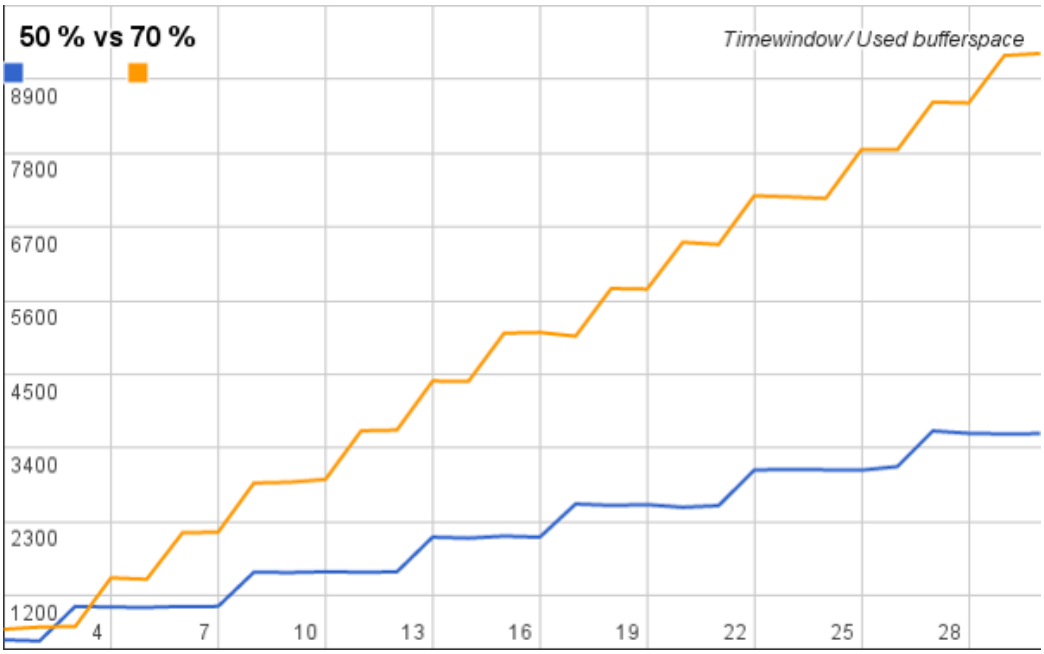



Figure 6.2: Comparing buffer usage between 50 and 70% in a longer simulation.

### 2. Alternating Occupancy

The occupancy will in most cases vary from time frame to time frame, and sometimes there won't be any data at all. Using the *alternatingOccupancySink()* function, a pattern of different levels of occupancies can be used as input. This way there will be more fluctuations in the amount of input data, which will give more fluctuating results. The pattern chosen for this simulation is displayed in Figure 6.3. It starts off with the expected average occupancy, before increasing in a steep fashion to 100 percent, then more slowly decreases down to zero. With this pattern, the model was tested to see how it handled increasing occupancy over time, and how fast it could stabilize after the amount of data decreases.

Results from Figure 6.4 shows that after 10-11 time frames the buffers are already full, but even though the occupancy decreased the serial outs still needed more time in order to stabilize. It is seen that the resulting buffer usage follows the occupancy pattern, giving more confidence in the model.

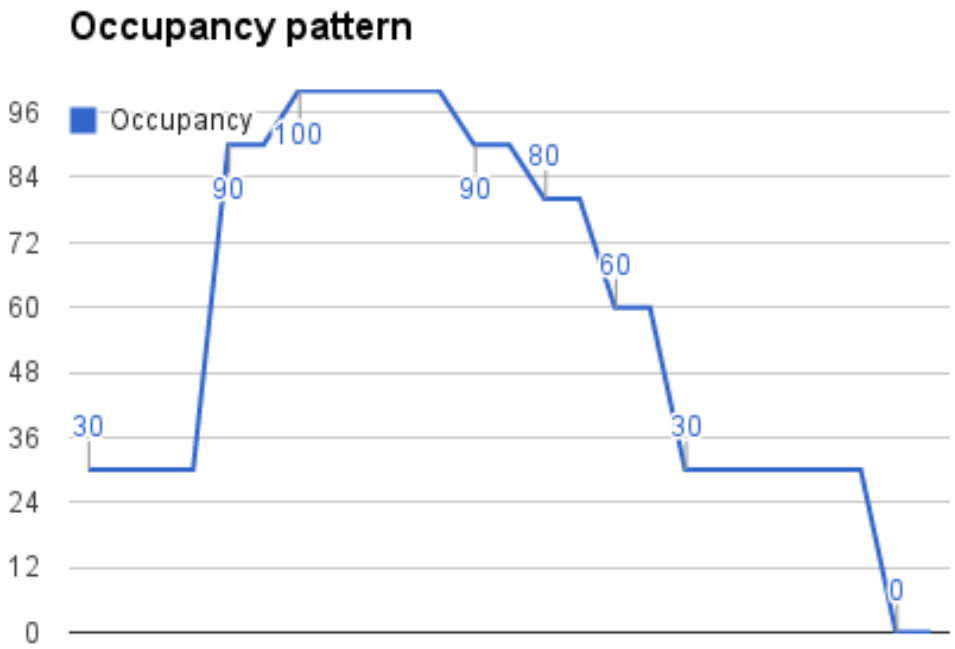



Figure 6.3: Occupancy pattern used for the simulation.

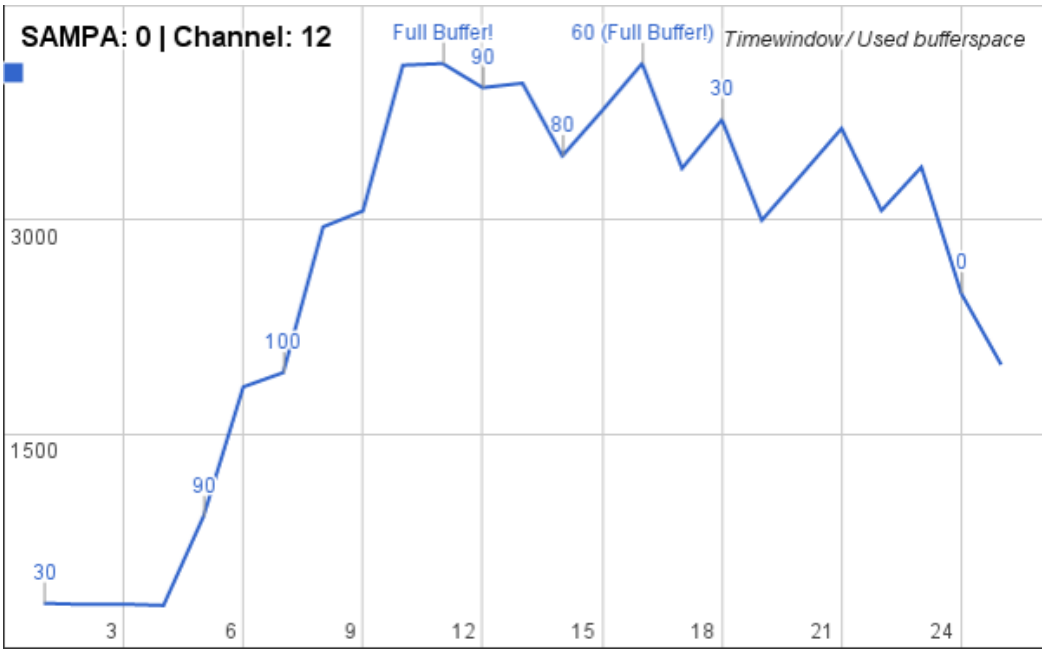



Figure 6.4: Results using a static pattern of occupancies.

### 3. Flat 100% Occupancy

This simulation examines how fast the buffer usage increases when getting 100 percent occupancy, as well as how many time frames it takes to reach the estimated 4k * 10 bit size of the buffers. The expected result was that it would go beyond 4k in the ten time frames being simulated, because of this the maximum buffer size was set to 8k. The readout speed is 2.5 times slower than the input speed, meaning that every serial out will be able to read around three out of eight channel buffers before the next time frame is finished. Let's take the first channel as an example: It will be the first channel the serial out will read, and while being read more data will come pouring in. The channel will have 40 percent of the data from the next time frame already stored when the serial out finishes with the first time frame. Then it needs to wait while the other seven channels are read before coming back. The point here is that the expected behaviour is that the channel buffer usage will rapidly increase, but in small periods it slows down because data is being read out.

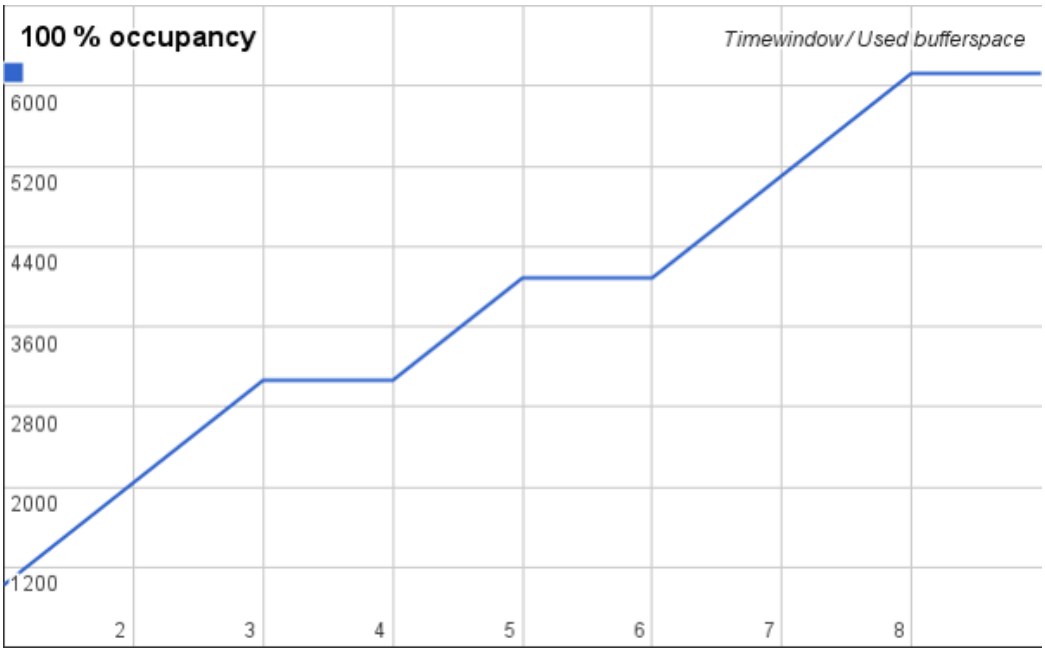



Figure 6.5: Results from a specific channel.(Channel 7, SAMPA 0)

The results in Figure 6.5 show that the buffer usage increased in a linear fashion until the third time frame. This means that after three time frames, the serial out started reading from it, which is consistent with the fact that

channel number 7 is the last channel read by one of the serial outs. After five time frames the buffer usage reached 4k, and at the end of the simulation it was around 6k.

### 4. Increasing Occupancy

Some of the problems with the previous simulations were their length. They have been too short to be able to say anything for certain. Improving on this, the next simulation ran for about 100 time frames, with continuously increasing occupancy. With the high amount of time frames, the resulting graph should provide more detailed information. Starting ten percent below the expected average of 30 percent, and then increased in a steady fashion until reaching 100 percent.

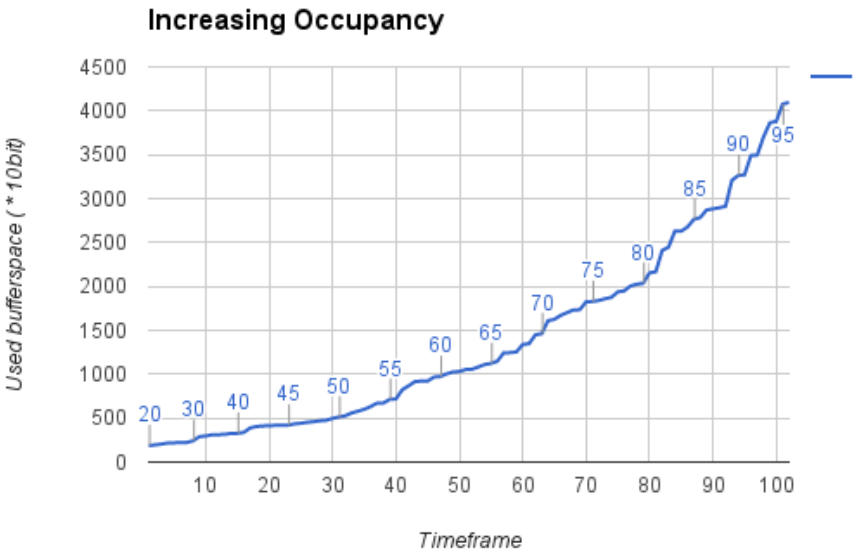



Figure 6.6: Results using a static pattern of occupancies.

No real surprises were found from this simulation, the graph shows a steady increase in buffer usage, which grows faster as the occupancy increases. This confirmed once again, that the simulation model was working as intended, and should produce both valid and interesting results. One thing that was unexpected was the amount of time it took to run the simulation just for one FEC. It took about two hours to complete the 100 time frames, and it

looked as if the higher the occupancy, the longer it took to complete a time frame.

#### Evaluating the Results

The results from the verification simulations were very useful when determining how good the model was, but it did not really give an indication about the performance of the channel buffers. This was not the intent of the simulations and to test the performance itself, more realistic simulation data is necessary. In the results from the alternating occupancy simulation it can be seen that every time it reached the maximum buffer size, overflow occurred and the data from that time frame was removed, causing the graph to decrease slightly. What was learned from this is that in the future it can be a good idea to remove the buffer size restrictions, in order to see how much is actually used. Already from the first simulation it became clear that the size of the header buffers will not be an issue, and in future simulations no information about them is included. Regarding the simulation time for longer simulations, it is clear that it needs improvement. One of the reasons it uses a long time, is that there are 160 channels to simulate. If instead only 32 channels, i.e one SAMPA module was used, it would be five times as fast. Using 32 channels will still give valid results because the simulation does not care about what happens after the data leaves the SAMPA module. This was implemented for the rest of the simulations, and it can be done without much effort because the test-bench stores information about the number of channels.

### 6.1.2 Normal Distribution

#### Preface

Now that it is established that the model is working as intended, more realistic data can be used as input. Using the normal distribution sink explained in Section 5.1.2, a more accurate estimation of buffer usage can be established. Another goal of these simulations is finding the compression factor for the Zero Suppression, see how it changes with different levels of occupancy, and where it reaches the necessary factor of 2.5. The verification simulations showed that simulations of about 100 time frames gave a good amount of data, so the simulations using normal distribution were run at least that long.

### 1. 28% Mean Occupancy

According to Figure 5.2 the mean occupancy that is expected is 28 percent, this becomes a natural starting point for the first simulation. If the results show that the model can not handle 28 percent or it was too little, the occupancy level can be adjusted for the next simulation. What is most likely to happen is that it is too much, because the shape of the data is the worst possible case for the Zero Suppression algorithm. In Figure 6.7 the randomly picked normal distribution used for this simulation is shown. Values ranging from 0 to 50 percent were used.

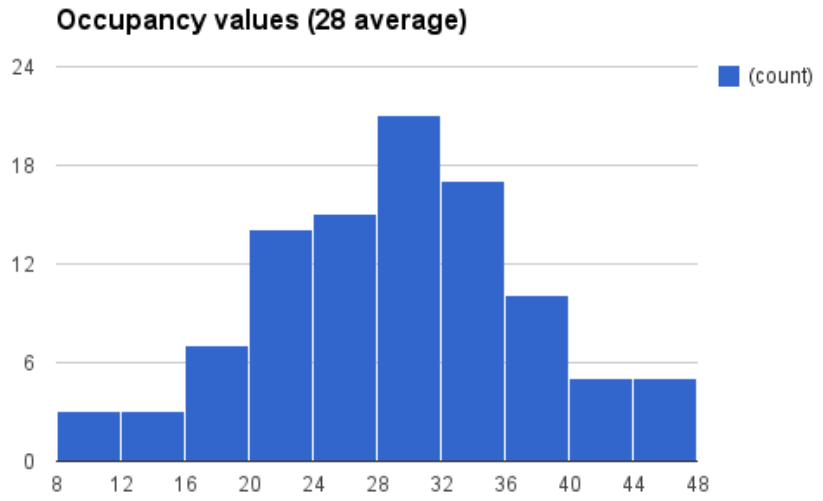



Figure 6.7: The distribution of occupancies used in the simulation from Figure 6.8.

As seen in Figure 6.8 the channel buffers filled up over time, and there was too much data for the serial outs to read. For the first 15-20 time frames it seems somewhat stable, this is most likely because the simulation was "lucky" and picked low numbers of occupancies for the first part. The buffer usage reached 4944 samples, but this number continued to grow if the simulation was longer. The difference when using a more realistic data model, instead of static occupancy becomes very clear in the resulting graph. There was a lot more fluctuation in the buffer usage, and in comparison to the verification simulations, the resulting graph does not just increase in a linear fashion.

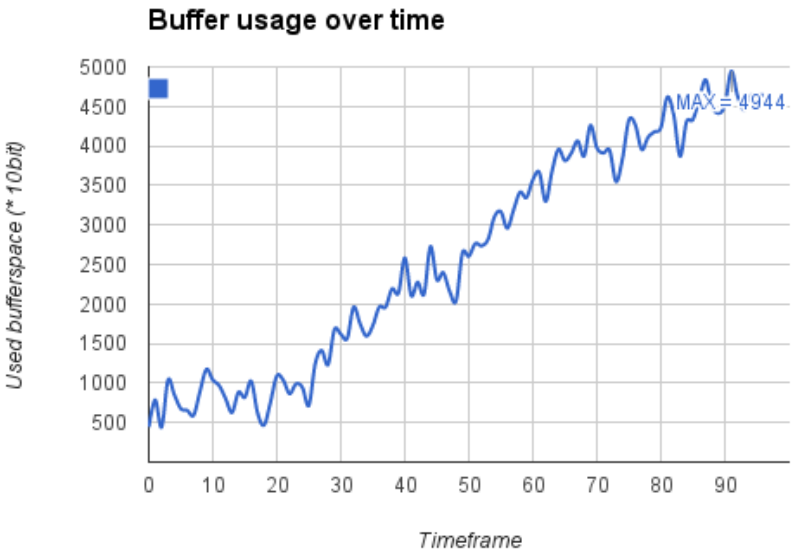



Figure 6.8: Using 28 percent mean occupancy.

This is a good indication that both the behaviour of the simulation model and the normal distribution is working.

### 2. 23% Mean Occupancy

Since 28 percent mean occupancy was too much for the model to handle, the next simulation tried five percent less, i.e 23%. It used the same setup in all other regards so that the results were comparable. 23 percent was chosen because it is far enough away from the first simulation to create a different result, without it being obvious whether the model would be able to handle it or not.

The graph in Figure 6.9 shows shifting buffer usage, varying between 100 and 1500 samples. However the graph shows that the model is able to read out the samples fast enough for the buffers not to overflow, always staying below the lowest of the suggested buffer sizes, 2k. On average the buffers stored 675 samples at any time. This is a number which is more than acceptable, and the simulation could most likely run forever and it would not matter. To confirm this, a simulation using 10 000 time frames was run (Figure 6.10), and the results confirm what the smaller one showed. The results from it is compressed into a histogram over the occurrence of differ-

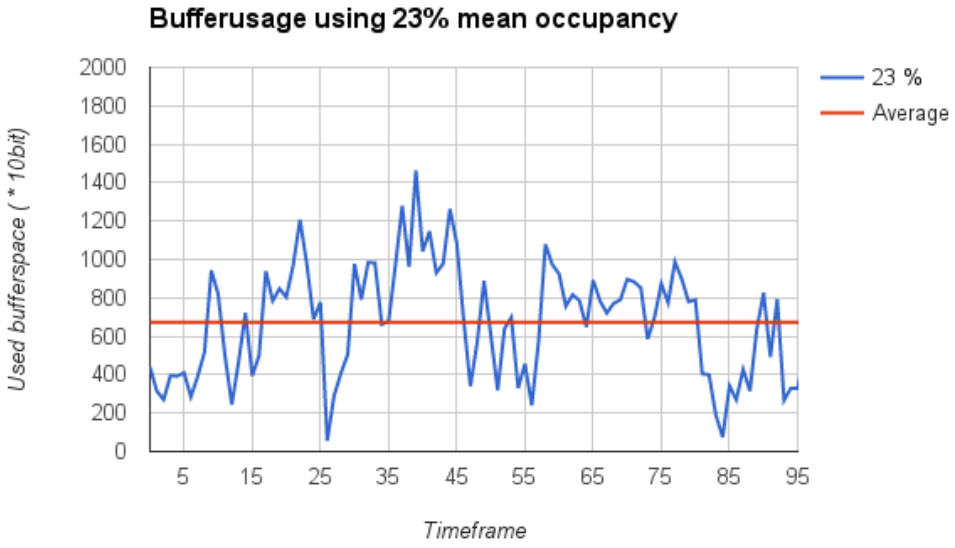



Figure 6.9: Results from simulation using 23% mean occupancy.

ent buffer usage values. It shows that most of the time the buffer usage is around 300-600, with an average of 473. When running longer simulation it is only natural that it will shift more than the smaller ones, reaching as high as 2100 samples. This means that a maximum buffer size of 2k is probably too small, as it is surpassed multiple times during 10 000 time frames.

The results gathered here show two very important things. First of all it is clear that the model is able to level out the buffer usage when there is 23 percent occupancy, and it does so without any big spikes. Second, it shows that when using 100 time frames the results gives an over estimate of the average usage. When it ran for longer, it evened out to a smaller average, but with more and bigger spikes. Now that it is clear that 23 percent is manageable, the next step is to find out where the limit it. Given that 28 percent was too much, the limit is somewhere between 23 and 28 percent occupancy.

### 3. Finding the Limit

Since finding the occupancy limit required simulations using 24 to 27 percent occupancy, a combined graph, comparing the development of the buffer usage over time was compiled. This gives a good overview of what happens both above and below the occupancy limit.

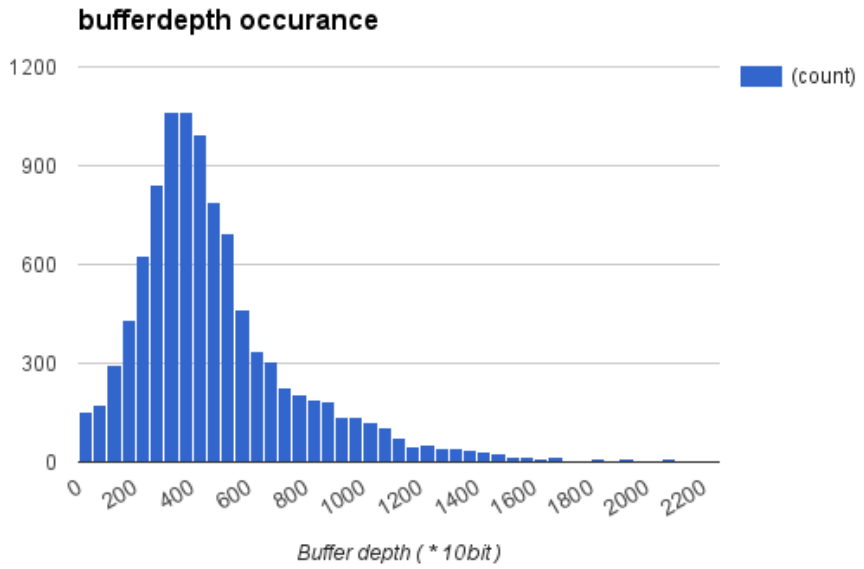



Figure 6.10: Results from running 10 000 time frames using 23% mean occupancy.

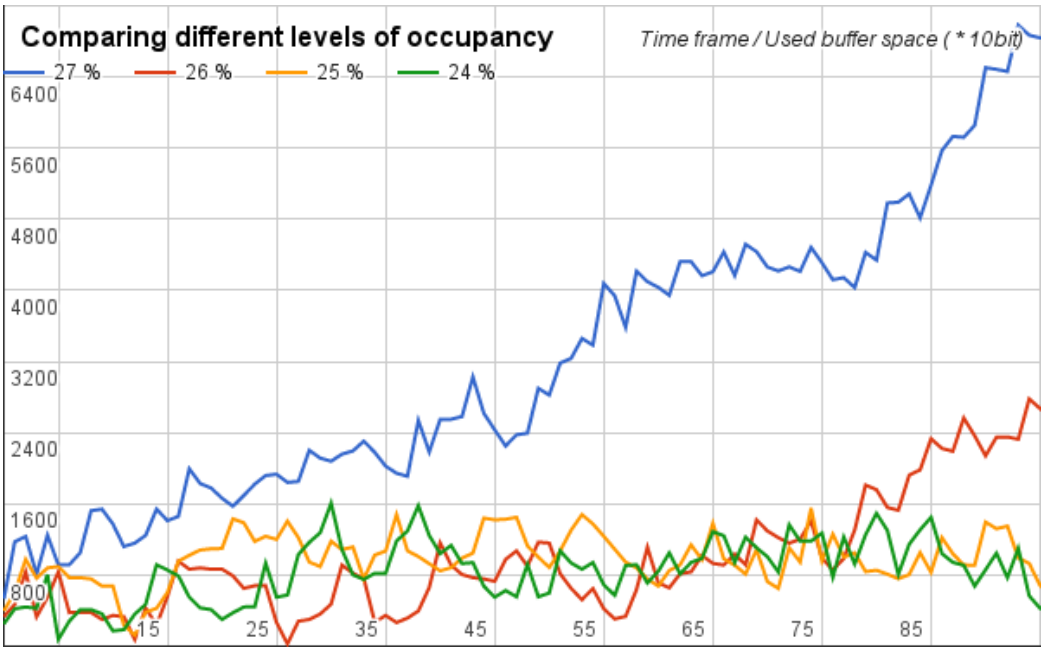



Figure 6.11: Results from using 24-27 percent occupancy.

The graph in Figure 6.11 shows results from all four different simulations. The difference between 24 and 25 percent is not visibly seen, and it looks like they both are pretty stable over the course of 100 time frames. However the difference in average buffer usage is still 150 samples, where 24 has around 850 and 25 has 1000. 1000 samples on average is starting to become high enough to cause issues with more time frames. In this particular simulation it may be stable and without any large spikes, but this can be because the simulation received occupancy levels which had a narrow distribution. Another thing to note about the difference of 24% and 25%, is that 24% has higher peaks than 25%. The reason for this can be seen when looking at their respective occupancy levels over time. Even though the mean is one percent higher for 25, it has smaller range of values, and almost never gets multiple high occupancies in a row. 24 percent has on the other hand a wide distribution, with multiple values over the mean in a row.

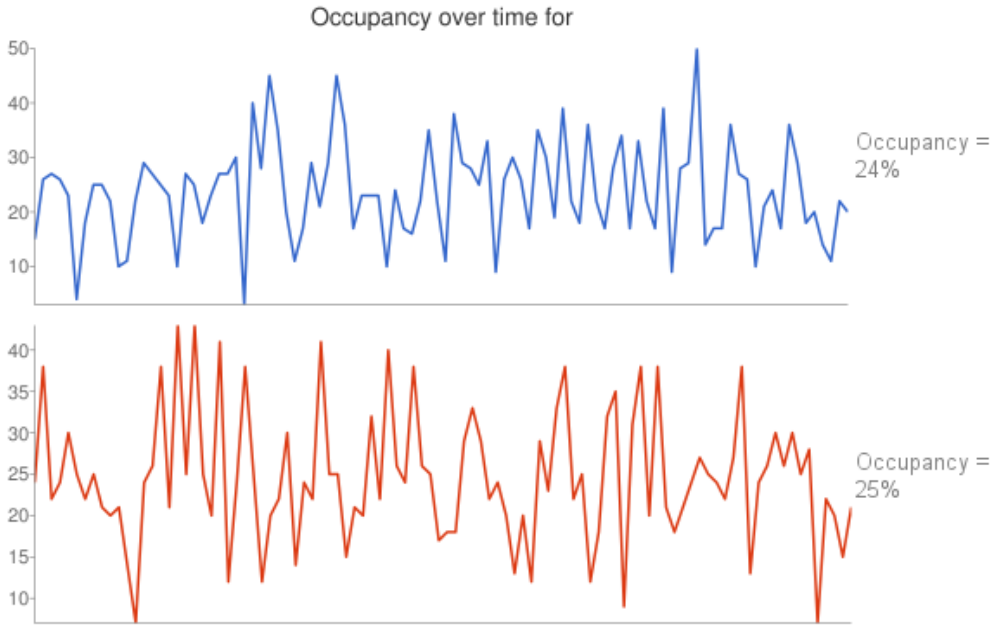



Figure 6.12: Showing occupancy over time using 24 and 25 percent.

When looking at 26 percent mean occupancy it can already be established that this is too much. With a good distribution and lucky picking it can probably stay stable, but after a while the buffer usage will start to grow. Not shockingly, this is definitely the case for 27 percent, increasing in buffer usage almost as fast as 28 percent.

### Evaluating the Results

From all the simulations ran using the normal distribution, a graph showing the Zero Suppression compression factor for every level of occupancy has been compiled. The compression factor is calculated by dividing maximum number of samples in a time frame with how many samples makes it through the Zero Suppression. From prior calculations it is concluded that a compression factor of 2.5 is necessary in order for the serial outs to read fast enough. If the compression factor of 2.5 is consistent with the occupancy limit found in the previous section, the results can be considered valid.

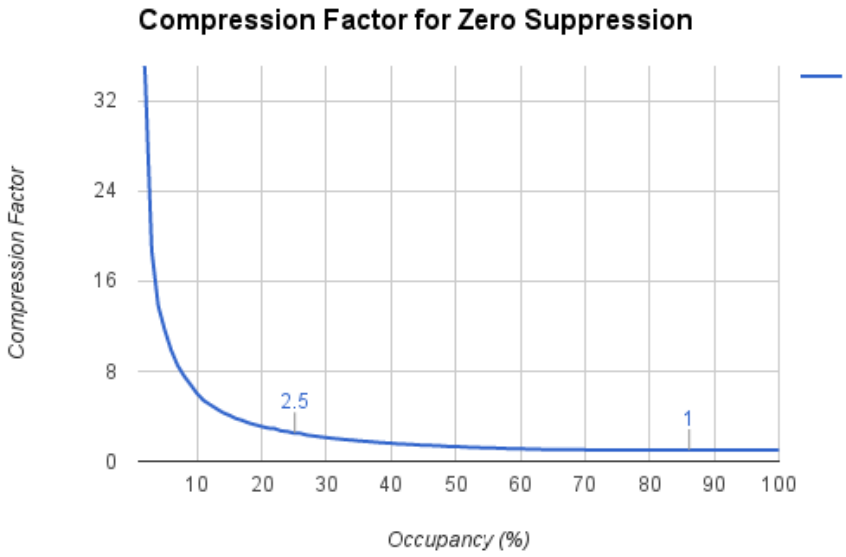



Figure 6.13: The compression factor over level of occupancy.

The compression factor graph shows that is reaches 2.5 at around 25 percent occupancy, decreasing faster and faster, giving an exponential function. 25 percent was exactly where the simulation model started to have trouble reading the data quickly enough, confirming that the results found were indeed valid. A conclusion regarding this is that a 4k maximum buffer size should be enough with average occupancy lower than 25 percent. This is only about three percent lower than the estimated average occupancy, and considering that the shape of the samples represent the worst case for Zero Suppression this is an acceptable difference.

### 6.1.3 Black Events

#### Preface

Using the black event dataset, the Zero Suppression and the Huffman encoding can be compared. A crucial difference between the two different compression schemes is that the Zero Suppression is applied before the samples are stored in the buffers, while the Huffman encoding is applied while reading out from the buffers. In other words, the Zero Suppression decreases the number of samples stored, while the Huffman effectively increases the readout speed by compressing each sample. For the Huffman encoding there are two things that can happen. Either it compresses the data enough that it manages to read the entire time frame before the next one is finished, which means that the buffer usage for every time frame will always be the size of an entire frame, i.e 1021 samples. The other case is that the compression is not good enough, and the buffers will overflow. If the first case happens, then measuring the buffer usage seems pointless, therefore what is measured is the size of the time frame after compression. This will be the closest way of comparing the results from both compression schemes. Another thing that will be discussed is what difference black events with pileup will have in comparison to no pileup.

#### 1. Zero Suppression

The focus when looking at the Zero Suppression on the black events, is the resulting buffer usage and not the compression factor itself. The reasoning for this is that the Zero Suppression worst case has already been discussed. The first results, which are displayed in Figure 6.14 are using black events w/o pileup. Results seems to show a rather low, and stable buffer usage, averaging out at about 330 samples, and reaching 600 samples. Since the data is collected from RUN 1, they have slightly less occupancy than what is expected for RUN 3, but it still has the most realistic shape available.

One of the reasons for creating pileup data is to increase the amount of real samples, which possibly can give more interesting results than using plain black event data. Running the simulation with the pileup data did indeed create interesting results, which can be seen in Figure 6.15. As the previous results it stayed at around the same level during the entire simulation, but one can clearly see the increase in overall occupancy. The pileup results reached about 200 samples higher, and had an average of about 100 samples more. Even though it is still manageable, it is clear that pileup has a real effect on the buffer usage when using Zero Suppression.

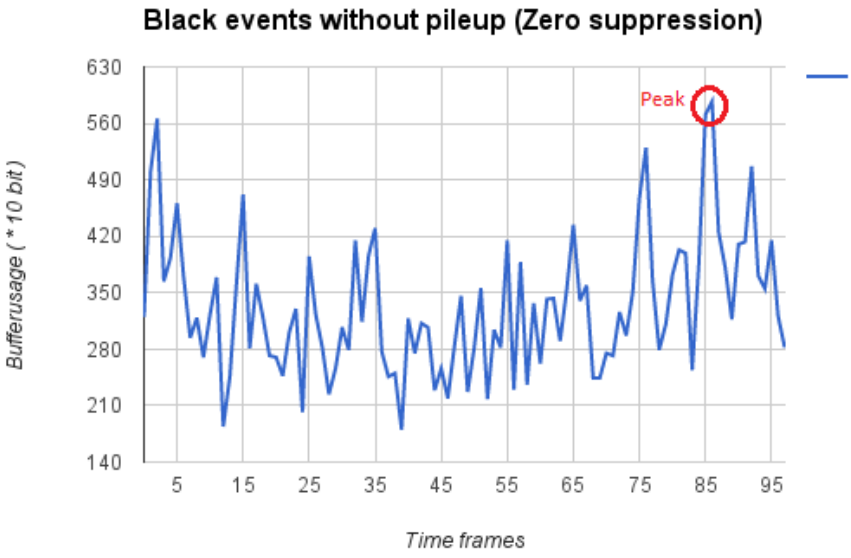



Figure 6.14: Results from Black events w/o pileup using Zero suppression.

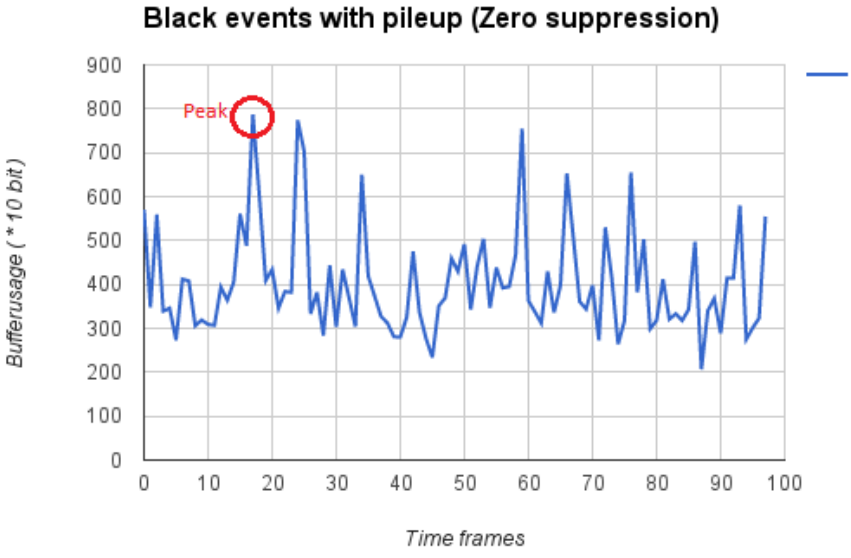



Figure 6.15: Results from Black events with pileup using Zero suppression.

### 2. Huffman Encoding

Now the same two simulations as before can be run using the Huffman encoding instead of Zero Suppression. The expectation for the Huffman

results was that the pileup should have less of an effect on the outcome, than when using Zero Suppression. This is because occupancy does not play as big of a role for Huffman, but instead the distribution of the sample values is what is important.

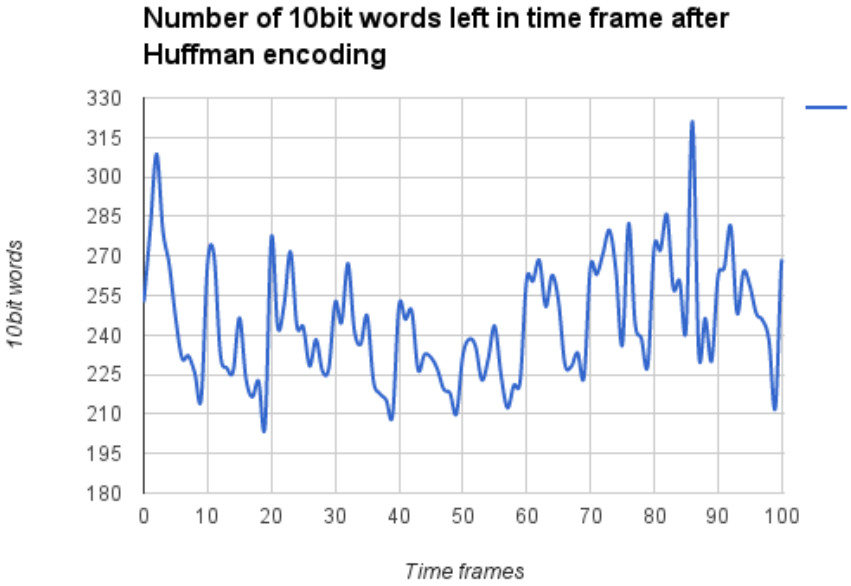



Figure 6.16: Results from Black events w/o pileup, using Huffman encoding.

The normal black event data was compressed enough for the serial outs to be able to read fast enough, therefore the buffer usage never surpassed the size of one full time frame. In Figure 6.16 the compressed size of each time frame can be seen. The graph shows that in most cases the Huffman encoding was able to compress the time frame to about 1/4 the original size, i.e a compression factor of four. A full overview over the compression factors during the simulation is displayed as a histogram in Figure 6.17. The factor went down as low as 3.2, but on the other hand reached as high as 4.95. This is not only more than the 2.5 that was needed, but also higher than what was expected.

Comparing the results from normal black events with the pileup results, there is a clear difference in regards to compression. The reason being that now that samples have been piled on top of each other, the difference between neighbouring time bins is bigger, which results in a worse Huffman table. The

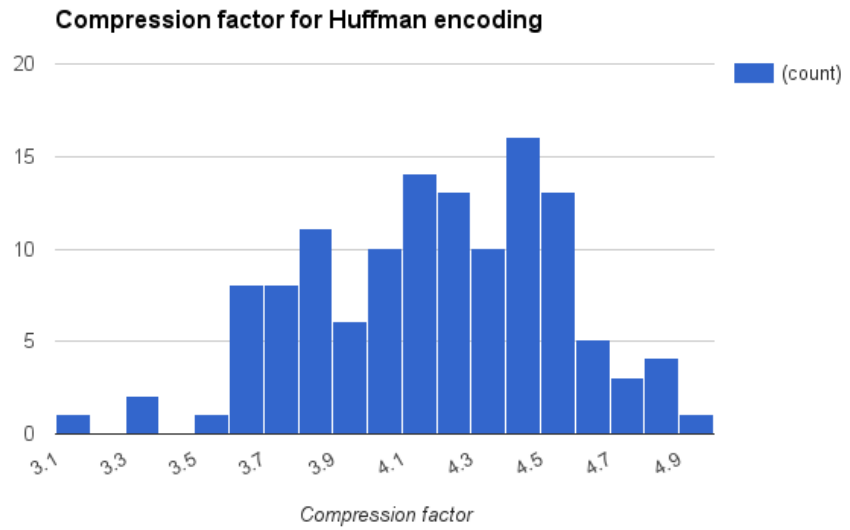



Figure 6.17: Compression factor of Huffman on normal black events.

compressed size of each time frame is now on average 100 words more than the normal black events, varying between 300 and 400 words.

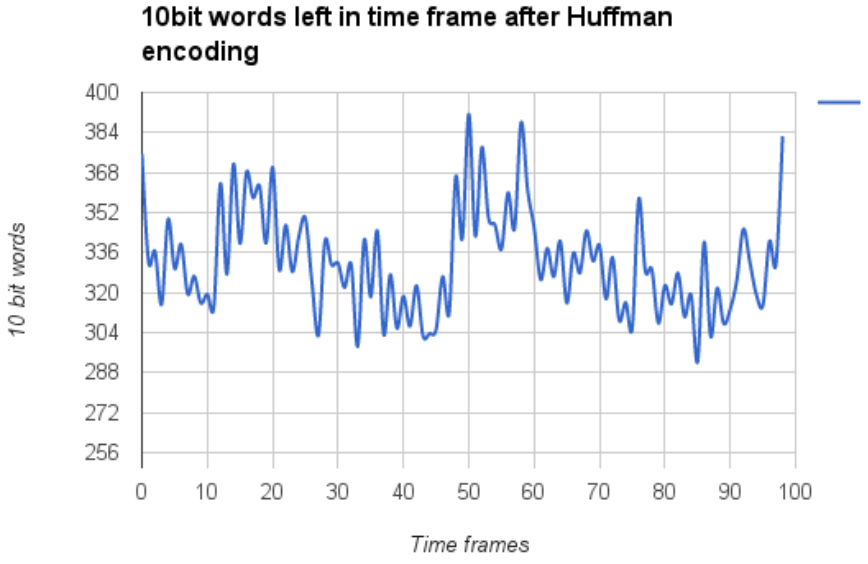



Figure 6.18: Compression factor of Huffman on piled up black events.

The pileup results show a definite decrease in the compression factor, as

can be seen in Figure 6.19. However with an average compression factor of three, the model is still able to read out data fast enough. The factor varied between 2.6 and 3.5, always staying above the 2.5 limit, meaning that the model should never have any trouble.

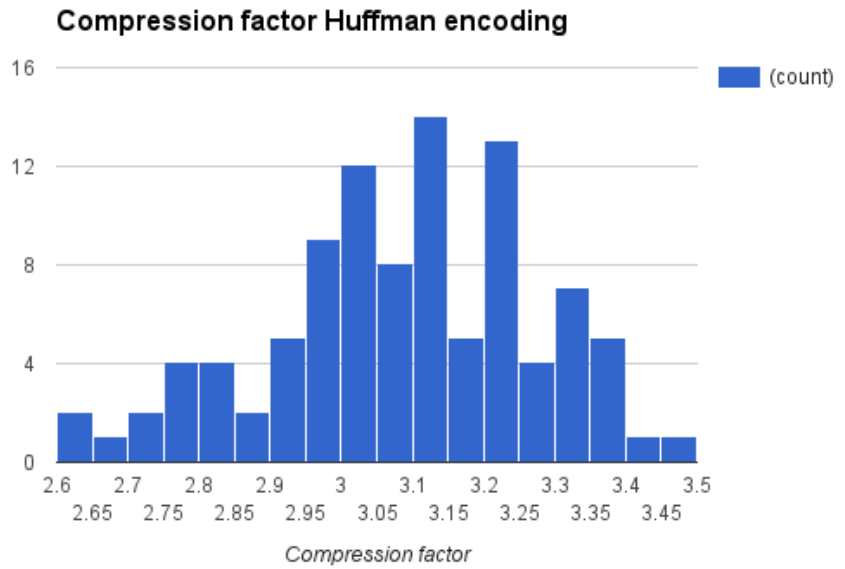



Figure 6.19: Results from black events with pileup using Huffman encoding.

### Evaluating the Results

Comparing the results when using Zero Suppression to using Huffman encoding ended up being harder than expected. This was because of the difference in how they compress the data, one being applied before storing, and one after. However there are multiple things to learn from these results. For all the scenarios, the buffers never came close to being full. It seems that for the black event dataset, the Huffman was able to perform better than Zero Suppression, but only by a small margin. Huffman had a stronger decrease in performance than Zero Suppression when comparing normal black events, and pileup. It can be argued that this is because the Huffman considered a best case scenario, seeing as it used the best possible Huffman table. There is a significant difference between normal and pileup data, but not enough to be an issue.

There was no way to see how the model behaved when Huffman didn't compress well enough, this is hopefully something that will be seen when

trying the simulated RUN 3 data.

### 6.1.4 Simulated Data for RUN 3

#### Preface

Using the simulated RUN 3 dataset, a final set of simulations can be performed. As with the black events, the focus is comparing Zero Suppression and Huffman encoding in terms of buffer usage, as well as examining the compression factor for the Huffman. Hopefully there will be enough spread in the data to really test how the model behaves when the Huffman encoding is not enough. It is impossible to make any predictions here, as the properties of this dataset is unknown, and not previously been tested.

#### 1. Zero Suppression

For the Zero Suppression, instead of taking the overall buffer usage now the channel with the highest average will be picked out and shown. This gives a slightly different look into the model, focusing on the behaviour of a single channel. Seeing as it is still the channel with the highest buffer usage it is still comparable to the overall highest shown in previous sections.

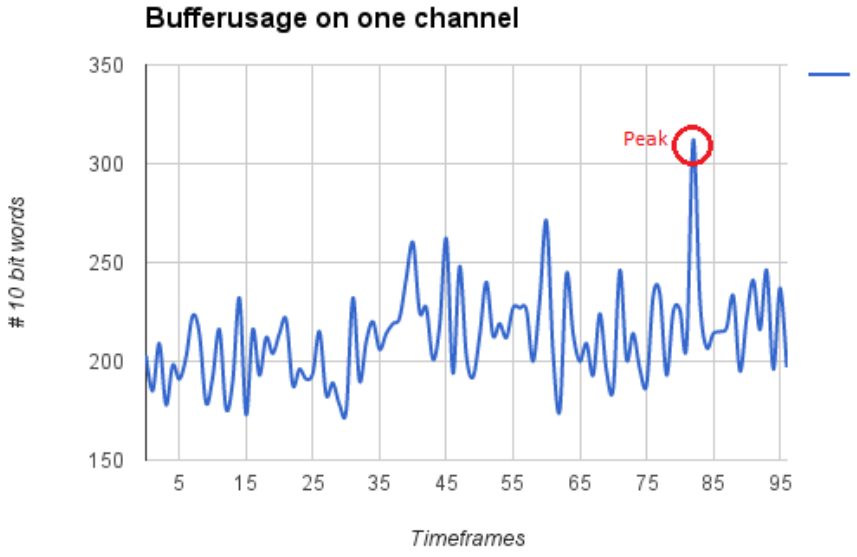



Figure 6.20: Results from using Zero Suppression on simulated RUN 3 data.

Results show an unexpectedly low buffer usage, averaging out on about 220 samples. In addition the difference from time frame to time frame was very low. Comparing this to the results from the black events, it was significantly lower than those. In Figure 6.21 the occupancy of the data is calculated. The highest recorded occupancy found was 24 percent, with most being between 10 and 16 percent. It seems that there is not enough data to get anymore decent results using Zero Suppression, but occupancy does not affect Huffman encoding in the same way.

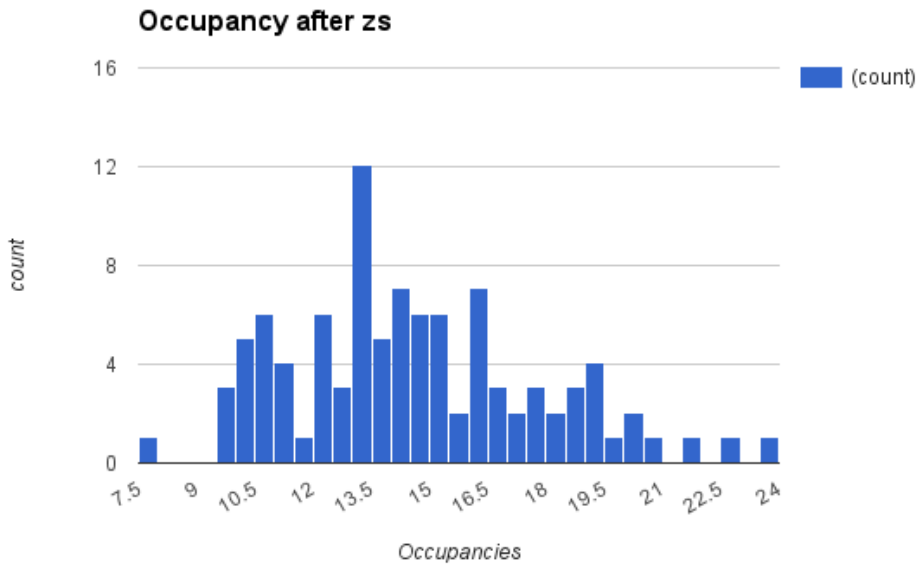



Figure 6.21: Occupancy after Zero Suppression.

### 2. Huffman Encoding

Even though the results from using the Zero Suppression were disappointing, the dataset should give interesting results using Huffman encoding, which can be compared to the black event and pileup results. As with those datasets, both the compressed size of the time frame, and the overall compression factor is what will be monitored, and presented. In Figure 6.22 the graph shows that the average compressed size of a time frame is around 290 words. What was strange, compared to the other datasets was the narrow difference in compressed size, with the largest time frame being 300, while the lowest was 285.

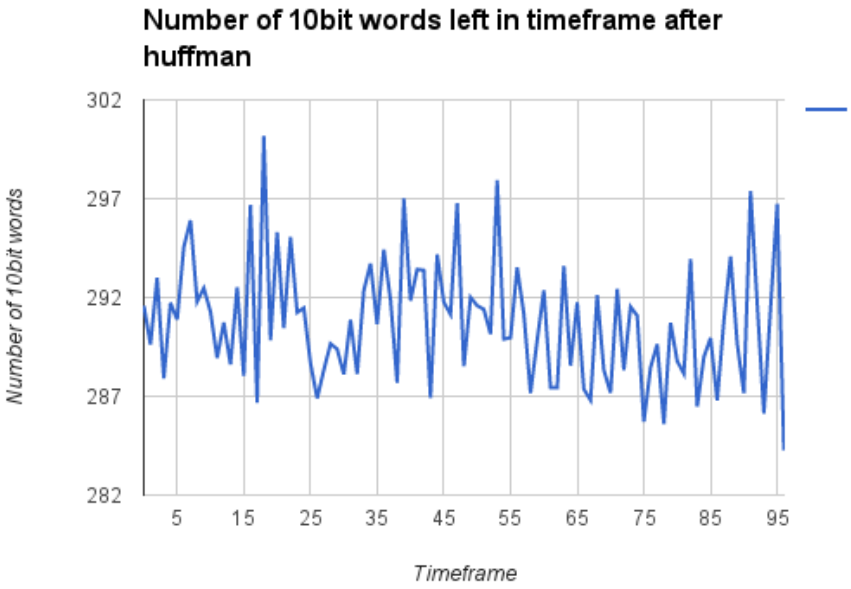



Figure 6.22: Results from using Huffman encoding on simulated RUN 3 data.

The narrow difference in compressed size of a time frame seen in Figure 6.22 means that the dataset had very low, but even difference between consecutive time bins. Since the Huffman encoding stores the difference between two time bins, the same difference has to be present a lot over all the time frames. Looking at the histogram in Figure 6.23 displaying the compression factor, it shows that the difference in compression factor was very small. It varied between 3.4 and 3.6, so to show the difference two decimals had to be used in the figure.

### Evaluating the Results

The results from using Zero Suppression were quite frankly uninteresting and there was not much to be learned from them, except that the dataset has a very low occupancy. The Huffman results on the other hand were much more rewarding, and were comparable to the previous results. Compared to the black events and pileup results, it lies somewhere in between them in terms of compression. Unlike previous results it had a very narrow result window, something not seen in the black event and pileup datasets. One thing that all the Huffman results had in common was that they never reached a compression factor of 5, meaning that between 2.6 and 4.9 is the best compression factor possible. That being said, a factor over four is probably

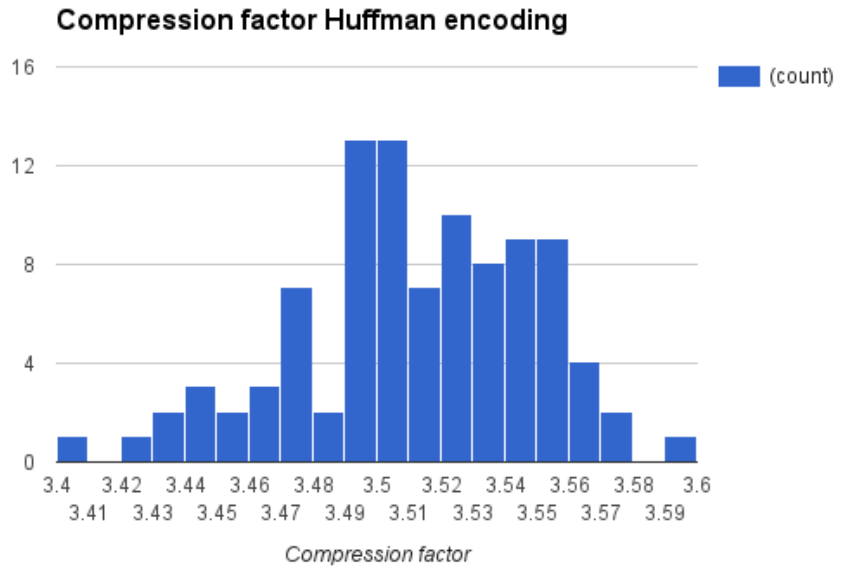



Figure 6.23: Compression factor of using Huffman encoding on simulated RUN 3 data.

not be possible when using Huffman encoding in RUN 3.

Even though the goal of the simulation was to see how the model behaved when Huffman encoding was not sufficient, the results were all in all educational. They showed the best possible compression that is achievable, and that datasets with different levels of occupancy had little to no impact on the Huffman's compression factor. One thing that could have been done differently was to use a worse Huffman table when compressing in order to get the results missing from all of this. That being said, the results would most likely resemble the ones found using Zero Suppression.

# Chapter 7

# Conclusion and Future Work

This thesis, and its surrounding work, tried to create a different way of testing the readout electronics for the TPC detector in RUN 3. By creating a computer model of the electronics and by running this model through a number of different simulations in both naive and realistic environments, an estimation of how the it would behave could be tested. The thesis focused on analysing the buffer usage for the SAMPA buffers, and comparing two different data compression schemes to see which worked best, and in what situation they did so. The previous chapter discussed a lot of the data gathered from the simulations, but does not draw any final conclusions with regards to the actual effects they have on the development of the readout electronics. The results found in this thesis were verified by running multiple tests on the simulation model, and were found to be within an acceptable margin for error. With that in mind the results found can be trusted and have a sufficient level of credibility for them to be taken seriously.

In regards to the results found about buffer usage in this thesis, they confirm that the compression factor needed for the system to work properly is 2.5. Using the normal distribution it was discovered that this limit was reached around 25 percent occupancy, just 2-3 percent lower than the highest calculated average for the inner pads shown in Figure 5.2. Since the simulation used an unrealistic data shape in order to create a worst case scenario, it implies that the electronics should be able to handle an average occupancy of 28 percent. To make an estimate on how big the buffers need to be in order to avoid overflow, one has to look at the results that were able to hold a stable level without increasing over time. It does not matter how big the buffers are if the electronics on average can not handle the amount of incoming data. Figure 6.10 shows that with 23 percent occupancy the buffer usage can be as high as 2k, which means that when the average occupancy is at its

limit it can go higher than that. This excludes 2k as a potential buffer size, and 4k may not be enough for the inner pads. To ensure that overflow does not occur on a regular basis, the buffers need to be 6k or even 8k.

As to the question about what compression scheme works the best, there are a lot of factors to consider. Given the right environment, both of them would work very well. The benefits of using Zero Suppression is that it works perfectly with the right amount of data, and is more dependent on the sheer amount rather than the shape. However since it has a definite limit where it just does not work anymore, it becomes useless if it faces a large enough occupancy. The Huffman encoding on the other hand is not influenced by the occupancy, and is more likely to work with higher levels of occupancy. The problem is to create a good enough Huffman table, and this is hard considering the experiment data is for the most part hard to predict. A solution to this is to have a dynamic table which updates as it receives more data. Another solution is to extend the Huffman algorithm to make it so that all samples that ends up with a negative compression use the actual sample value instead.

## 7.1 Outlook

In the beginning of this thesis, the degree of importance in regards to the development of the readout electronics was very low, but as more results were presented it seemed to become more important. Decisions were being made based on results found using the simulation model, and it seemed as people in charge of development started to rely on them. Because of this, more simulation scenarios will likely be run in the future, and this thesis will be a big part of the finished readout electronics.

As said the simulation model created in this thesis can be very useful later on in the development of the readout electronics. There is still work to be done with testing Huffman encoding, and this is something that the model could be used for in the future. Creating a solution which tests the cooperation between Zero Suppression and Huffman in the simulation could be accomplished by using the current model as a starting point. It has become a powerful tool which could be extended to include other components, or another setup all together. That being said, there is a lot that could be improved. One of things that may need improvement is the data gathering method, and the way it is recorded. Another quality of life change that could

be made is to create a more object-oriented DataGenerator as it quickly became large, and cluttered. Instead of adding new functions for sending data, there should be a base `class` which can be extended, and every implementation contains logic for one way of sending data. A flaw which over time became a big bottleneck was that the simulation depended on a specific timer, instead of forcing it to stop after all signals had been processed. This meant that simulations could run longer than needed, thereby wasting time. The flaw was due to an oversight early in development and it became hard to rectify after it was discovered. Regardless, there are many ways that the simulation model can become useful later on, but how it will be used remains to be decided.

## 7.2 Reflections

To successfully create a simulation model that could produce valid results, and help in the development of the readout electronics is a testament to the endless applications of computer science. Even though the implementation required a lot of work, the hardest part was to acquire the necessary domain knowledge. Starting from scratch with one of the biggest research projects in the field of physics, as a programmer has been challenging to say the least. Working on this project has been rewarding in terms of knowledge and experience gained across multiple fields of science. The experience of working with experts from around the world has been a great opportunity, and one thing I have learned is how important cooperation when working on such a cross domain task is.

# Appendix A

# Code Listings

```
1  class Channel : public sc_module {
2  public:
3
4    //Ports between the DataGenerator and the Channel
5    sc_port< sc_fifo_in_if< Sample > > inputPort;
6
7    //Data and Header buffers
8    list<Sample> dataBuffer;
9    list<Packet> headerBuffer;
10
11   //Main SystemC Thread.
12   void receiveData();
13   //Getter and Setter methods
14     .......
15   //End
16   Channel(sc_module_name name);
17
18 private:
19   //Can specify which pad the channel comes from.
20   int Pad;
21   int PadRow;
22   int Addr;
23   int SampaAddr;
24 };
```

**Listing A.1: Channel header file.**

```
1  class SAMPA : public sc_module {
2  public:
3
4    //I/O Ports.
5    sc_port< sc_fifo_in_if< Sample > >  inputPorts[
         NUMBER_OF_CHANNELS];
6    sc_port< sc_fifo_out_if< Packet > > eLinks[NUMBER_OF_ELINKS];
7
8    //Channels
9    Channel *channels[SAMPA_NUMBER_INPUT_PORTS];
10
11   //Initialize channels
12   void initChannels(void);
13
14   //4 async serial out threads
15   void serialOut0(void);
16   void serialOut1(void);
17   void serialOut2(void);
18   void serialOut3(void);
19
20   //Routing method, serial outs read from correct buffer.
21   int processData(int serialOut);
22   SAMPA(sc_module_name name);
23
24 private:
25   int Addr; //Hardware Address
26 };
```

**Listing A.2: Sampa header file.**

```
1  if(sample.data > 0 && lastSample.data > 0){
2    firstCluster = true; //No valid clusters found until now.
3    validCluster = true; //Found valid cluster.
4    zeroCount = 0; //Number of invalid samples in a row.
5    addSampleToBuffer(sample, numberOfClockCycles);
6  } else if(sample.data > 0 && lastSample.data <= 0){
7    //Count as invalid sample, but add to fifo for now.
8    validCluster = false;
9    zeroCount++;
10   addSampleToBuffer(sample, numberOfClockCycles);
11 } else if(sample.data <= 0 && lastSample.data <= 0){
12   //If part of cluster meta-data, add to fifo.
13   if(zeroCount < 2 && firstCluster){
14     addSampleToBuffer(sample, numberOfClockCycles);
15   }
16   zeroCount++; //Count as invalid so next zero doesn't get added
       .
17 } else if(sample.data <= 0 && lastSample.data > 0){
18   if(validCluster){ //End of validCluster.
19     if(zeroCount < 2 && firstCluster){
20       addSampleToBuffer(sample, numberOfClockCycles);
21     }
22   } else {
23     //Last sample was not valid, and is removed from fifo.
24     removeSampleFromBuffer();
25     //Add if removed sample was part of cluster meta-data.
26     if(zeroCount <= 2){
27       Sample newSample;
28       newSample.timeFrame = sample.timeFrame;
29       addSampleToBuffer(newSample, numberOfClockCycles);
30     }
31   }
32   zeroCount++;
33 }
```

**Listing A.3: Zero suppression algorithm.**

# Terms and Abbreviations

**ALICE** A Large Ion Collider Experiment.

**ALTRO** ALTRO ASIC.

**ASIC** Application Specific Integrated Circuits.

**BT** Binary Tree.

**C++** A object-oriented programming language.

**CERN** European Organization for Nuclear Research.

**CRU** Common Readout Unit.

**FEC** Front-End Card.

**FIFO** First-In-First-Out.

**FPGA** Field-Programmable Gate Array.

**GBTx** Giga Bit Transceiver.

**GEM** Gas Electron Multiplier.

**LHC** Large Hadron Collider.

**MWPC** Multi Wire Proportional Chamber.

**OOP** Object-Oriented Programming.

**Priority Queue** Datastructure which sorts elements based on a priority(numerical value).

**QCD** Quantum Chromodynamics.

**Random Number Generator** Computational device designed to generate numbers that lack a pattern.

**RCU** Readout Control Unit.

**SAMPA** SAMPA ASIC.

**SystemC** A simulation library building on C++.

**TeV** Tera Electron Volt.

**TPC** Time Projection Chamber.

**Verilog** A Hardware description language.

**VHDL** A Hardware description language.

**Zero Suppression** Suppression scheme/algorithm.

# Bibliography


[1] Long Shutdown 2 @ LHC. https://indico.cern.ch/event/315665/session/7/contribution/37/material/paper/1.pdf. Accessed: 2015-01-09.

[2] Werner Riegler. The ALICE Upgrade plans - Article. http://ph-news.web.cern.ch/content/alice-upgrade-plans. Accessed: 2015-01-12.

[3] CERN - Article. http://home.web.cern.ch/about. Accessed: 2015-01-12.

[4] The birth of the web - Article. http://home.web.cern.ch/about. Accessed: 2015-01-12.

[5] The Large Hadron Collider - Article. http://home.web.cern.ch/topics/large-hadron-collider. Accessed: 2014-11-14.

[6] Image of the LHC. http://slhcpp.web.cern.ch/SLHCPP/images/courier_article1.jpg. Accessed: 2015-02-17.

[7] The Large Hadron Collider - Brochure. http://cds.cern.ch/record/1165534/files/CERN-Brochure-2009-003-Eng.pdf. Accessed: 2015-01-16.

[8] The ALICE experiment - Homepage. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2Experiment-en.html. Accessed: 2015-01-17.

[9] Quark-Gluon plasma - Article. http://home.web.cern.ch/about/physics/heavy-ions-and-quark-gluon-plasma. Accessed: 2015-01-18.

[10] The ALICE experiment - Article. http://home.web.cern.ch/about/experiments/alice. Accessed: 2015-01-17.

[11] The ALICE detector. http://alicematters.web.cern.ch/sites/alicematters.web.cern.ch/files/images/ALICE_paper_diag.jpg. Accessed: 2015-02-19.

[12] ALTRO - Article. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2_TPC.html. Accessed: 2015-02-13.

[13] ASIC - Definition. http://www.radio-electronics.com/info/data/semicond/asic/asic.php. Accessed: 2015-02-13.

[14] Upgrade of the ALICE Time Projection Chamber - Techincal Design Report. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2_TPC.html. Accessed: 2015-02-13.

[15] C. Lippmann. Example of a GEM pad structure. private communication.

[16] Upgrade of the Readout & Trigger System - Techincal Design Report. http://cds.cern.ch/record/1603472/files/ALICE-TDR-015.pdf. Accessed: 2015-02-13.

[17] Jerry Banks. *Discrete-event System Simulation.* Upper Saddle River, NJ: Prentice Hall, 2001.

[18] Bunton Black, Donovan and Keist. *SystemC: From the Ground Up.* Springer Science+Business Media, second edition, 2010.

[19] Daintith and Wright. zero suppression. "http://www.oxfordreference.com/10.1093/acref/9780199234004.001.0001/acref-9780199234004-e-5900". Accessed: 2015-02-24.

[20] Ince. Huffman coding. "http://www.oxfordreference.com/10.1093/acref/9780191744150.001.0001/acref-9780191744150-e-1565". Accessed: 2015-02-25.

[21] Dasgupta Papadimitriou and Vazirani. *Algorithms.* Alan R. Apt, 2008.

[22] cplusplus.com. Vector, c++ container - Documentation. http://www.cplusplus.com/reference/vector/vector/. Accessed: 2015-03-17.

[23] cplusplus.com. List, c++ container - Documentation. http://www.cplusplus.com/reference/list/list. Accessed: 2015-03-17.

[24] cplusplus.com. Queue, c++ container - Documentation. http://www.cplusplus.com/reference/queue/queue/. Accessed: 2015-03-17.

[25] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Prentice Hall, 2011.

[26] Rosetta Code. Example of a GEM pad structure. http://rosettacode.org/wiki/Huffman_coding.

[27] Free Software Foundation. GNU Free Documentation License 1.2. http://www.gnu.org/licenses/fdl-1.2.html.

[28] Roger L Casella, George; Berger. *Statistical Inference*. Duxbury, second edition, 2001.

[29] Example of a normal distribution. http://study.com/cimages/multimages/16/normal_distribution2.PNG. Accessed: 2015-04-30.