Implementation of a Custom Muon High Level Trigger Monitoring System

University of Cape Town

Thesis presented for the degree of Master of Science in the Department of Physics

Author: Seforo Mohlalisi

Supervised by: Professor Roger Fearick, Dr Zinhle Buthelezi and Dr Zeblon Vilakazi

May 26, 2010

I know the meaning of Plagiarism and declare that all of the work in the document, save for that which is properly acknowledged, is my own.

Signed by candidate Signature Removed

Seforo Stephen Mohlalisi

Abstract

A Large Ion Collider Experiment (ALICE) is one of the 4 major experiments at the Large Hadron Collider (LHC) at CERN. Its main aim is to investigate the physics of strongly interacting matter in proton-proton, nucleus-nucleus and nucleus-proton or proton-nucleus collisions at ultra high energy densities, where the Quark Gluon Plasma (QGP) is expected to form.

The experiment is expected to produce data at very high rates of about 25 Gbytes/s however the bandwidth to permanent storage is limited to about 10% (1.25 Gbyte/s) of the total expected data rates. In order to reduce data to permanent storage a special level of selecting interesting/relevant physics events is required. In ALICE the trigger (selection) of signals is issued based on a series of levels varying from levels 0 (L0) up to the High Level Trigger (HLT). For the ALICE muon spectrometer, the role of the trigger is to select events containing muon tracks, with the transverse momentum (p_t) above a given threshold. Due to the limited spatial resolution of the muon trigger chambers a pt cut above a few GeV with the L0 trigger is not possible. While the L0 signal for the muon spectrometer is issued at about 700 - 800 ns, the HLT is delivered at about 1 ms.

The role of the HLT is to perform online and offline reconstruction of the ALICE muon spectrometer data in order to improve the measured (L0) p_T resolution. In this way a better separation between relevant physics events and unwanted events (background) can be attainable, which could eventually lead to lower trigger rates. The HLT is designed to improve signal-to-background ratio in the raw data transferred to the storage. In order to facilitate online/offline data analysis the HLT monitoring system which will enable the user to graphically view the events during the reconstruction phase was developed in this study. The system will read and decode the reconstructed events from the HLT analysis chain using the HLT Online Monitoring Environment including ROOT (HOMER) and displaying them on the ALICE Event Visualization Environment (ALIEVE). In addition, the

utility, dHLTdumpraw, that inspects, with finer detail, the contents of all muon HLT internal data blocks and the detector data link (DDL) raw data stream is also described.

Acknowledgements

I would like to thank my supervisor Dr Zinhle Buthelezi for her unwavering support during the course of my project. My humble acknowledgment is also directed to Artur Szostak for coming through when the ship was about to sink. I would also like to thank my supervisors Dr Zeblon Vilakazi and Professor Roger Fearick for the support they gave me. Professor Jean Cleymans also deserves a special thanks for making me feel welcomed in the UCT-CERN research group.

I further pass my sincere gratitude to my family, Lekometsa Mokhesi, and Kaibe Mokoma who were always by my side when the going was very tough. My last but not least acknowledgment is to the University of Cape Town which gave me the opportunity and the conducive learning environment to pursue masters degree in physics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Large Hadron Collider (LHC) accelerator at the European Organization for Nuclear Research (CERN) in Geneva is currently the biggest particle accelerator in the world. It is installed 44 meters underground in the 26.7 km ring at the border of Switzerland and France [1]. It is designed to accelerate and collide proton-proton (p-p), Lead-Lead (Pb-Pb) ions, p-Pb and Argon-Argon (Ar-Ar) to recreate conditions that existed 10^{-5} s after the Big Bang [2].

The LHC accelerates beams of nuclei in opposite directions. These beams overlap and hence collide at several interaction points where detectors such as A Toroidal Large Acceptance Spectrometer (ATLAS)[3], Compact Muon Solenoid (CMS)[4], LHC b factory (LHCb)[5] and A Large Ion Collider Experiment (ALICE)[2] are installed. Figure 1.1 shows how these four major LHC detectors are located with respect to each other. While ATLAS, CMS and LHCb search for the existence of signatures predicted by the Standard Model, the ALICE detector is dedicated to study the physics of strongly interacting matter at extreme conditions of high temperature and baryon densities, where a new phase of matter, the Quark Gluon Plasma (QGP), is expected to form. For the purpose of this study we focus our discussion on the ALICE detector and related topics.

ALICE detector is an ensemble of central barrel detectors, forward detectors and the muon spectrometer. It is installed underground in the LHC tunnel at St. Genis-Pouilly on the French side of CERN. The schematic diagram of the ALICE detector is shown in Figure 1.2.

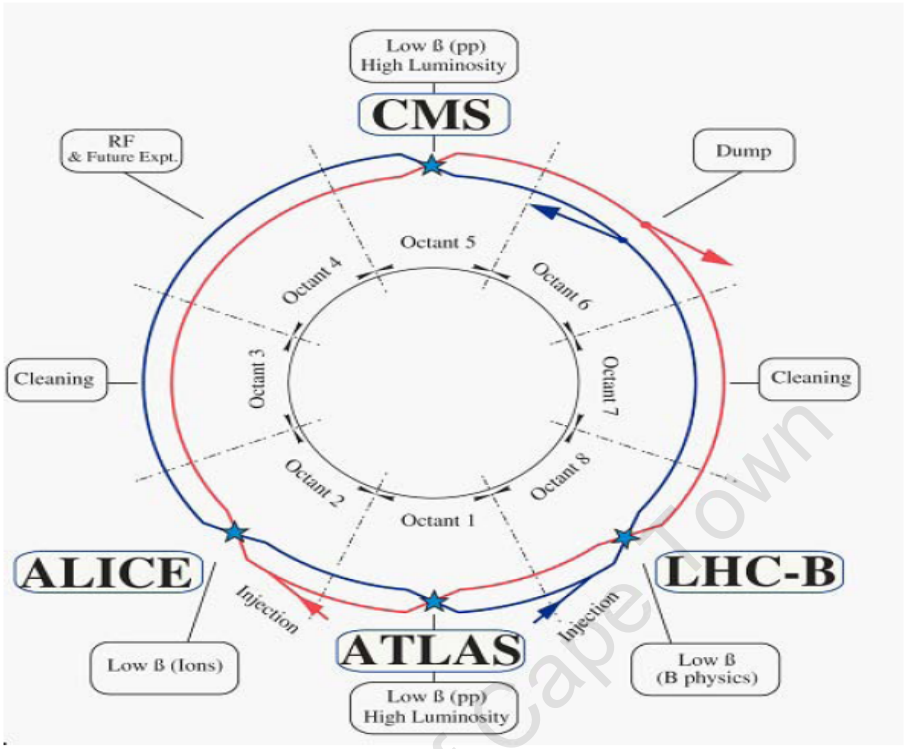

Figure 1.1: Schematic diagram of LHC [1].

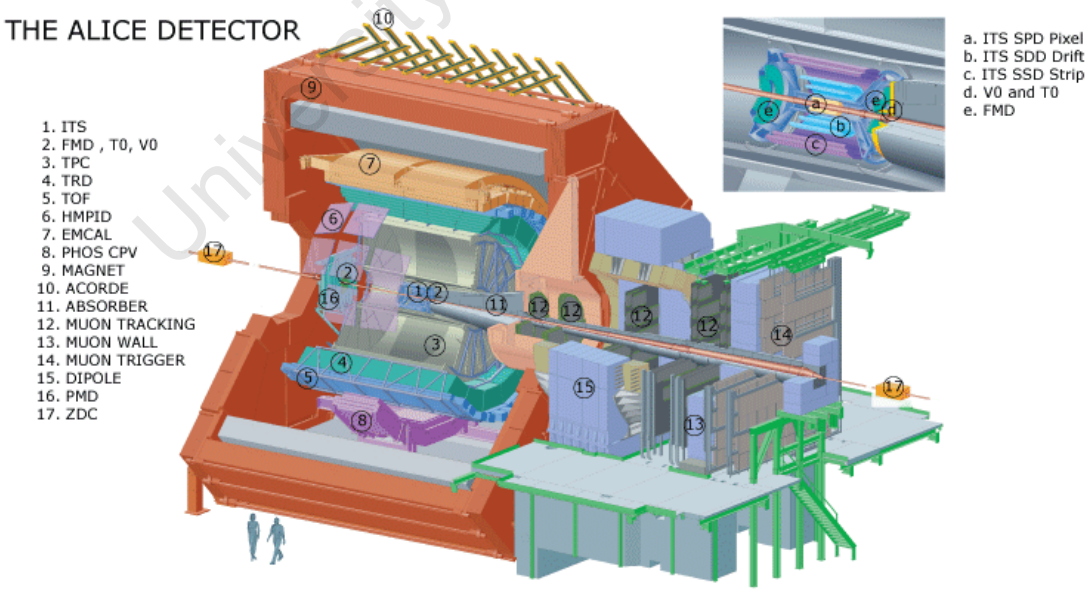

Figure 1.2: ALICE detector layout [6].

1.1 ALICE central barrel and forward detectors

The central barrel detectors are discussed in detail elsewhere [7, 8] only a brief summary is given in this section. These detectors are located around the interaction point and they are housed inside a large solenoid magnet. They consist of the Inner Tracking System (ITS), Time Projection Chamber (TPC), Transition Radiation Detector (TRD), and Time Of Flight (TOF). While ITS determines the primary vertex with resolution better than 100 μm, TPC measures the momentum of charged particles with good two-track separation. TRD provides electron identification for momentum above 1 GeV/c, and TOF is dedicated to charged particle identification in the low transverse momentum (p_T) range of 0.2 GeV/c $< p_T <$ 2.5 GeV/c.

Other detectors that form part of the central barrel are : High-Momentum Particle Identification Detector (HMPID) enhances particle identification capability of ALICE by enabling identification of particles beyond momentum interval attainable to TPC and ITS. The PHOton Spectrometer (PHOS) main physics objectives are to test thermal and dynamic properties of an initial phase of collision and to study jet quenching (interaction of dense matter with energetic partons). The ElectroMagnetic CALorimeter (EMCal) enables ALICE to analyze, in more details, the physics of jet quenching in large kinematic range. The ALICE COsmic Ray DEtector (ACORDE) provides a fast trigger (level 0) signal for calibration and alignment procedures of some of the ALICE tracking detectors and it is used in conjuction with TPC, TRD and TOF to detect single and multi muon events to enable ALICE to study high energy cosmic rays.

The ALICE forward detectors that give information about the collision, that is, the event centrality and multiplicity [7] are they are the Zero Degree Calorimeter (ZDC), the Photon Multiplicity Detector (PMD), the Forward Multiplicity Detector (FMD), the V0 and T0 detectors.

1.2 The ALICE muon spectrometer

The ALICE muon spectrometer [9, 10, 11] is placed outside the central barrel. It consists of the front absorber, the beam shield, the dipole magnet, the tracking stations, the muon filter wall and the trigger stations, as shown in Figure 1.2. The main aim of the muon spectrometer is to detect muons

produced from the decays of Upsilon (Υ) and J/Psi (J/ψ) resonances and their families to an accuracy of 1-2% on their mass resolution. It detects muons in the polar acceptance angle of $171\,^\circ \leq \theta \leq 178\,^\circ$.

1.2.1 The absorbers and the dipole magnet

The muon spectrometer consists of the front absorber, the beam pipe shield and the muon filter wall. The absorbers reduce the hadronic background and low energy muon background. The front absorber also limits multiple scattering and energy loss of muons.

The dipole magnet is made from resistive coils and it has a nominal field strength of 0.7 T and a field integral of $\int |B| dz \sim 3$ Tm along the beam pipe. Its field is in the direction perpendicular to the beam pipe, that is, in the x-direction. The dipole magnet makes it possible to measure p_T of charged particles.

1.2.2 The muon tracking and trigger systems

The tracking system is made up of 10 tracking planes consisting of several hundreds of high resolution cathode pad chambers in the y-z bending plane. The tracking system will make it possible to achieve spatial resolution of about 100 μm and it works very well in high particle multiplicity environment.

The muon trigger system is made up of two trigger planes consisting of Resistive Plate Chambers with moderate space resolution. The RPCs are optimised to achieve space resolution of 1 cm to enable the muon spectrometer to trigger only on high p_T muons produced from the decays of heavy quarkonia ($J/\psi, \Upsilon$). About eight low p_T muons produced from the decays of pions and kaons are expected to reach the trigger detectors in a central Pb-Pb collision [12]. In order to reduce the propability of triggering on these low p_T muons, a cut on p_T has to be imposed by means of a trigger system in the x-z non bending plane and with a very fast response.

1.3 The aim of the study

Because ALICE is a multifaceted system, it is expected to produce data at an average rate of 25 GB/s however, the bandwidth to permanent storage is limited to only 1.25 GB/s. These large data rates will definitely overload the Data Aquisition (DAQ) system. To resolve this problem, the High Level Trigger (HLT) [13, 14] has been developed to select only the relevant physics events amongst background before data is sent to the DAQ. All functions of HLT, .e.g. event selection, data compression, etc are done in real time i.e. as ALICE is running. To meet the demands of an online or real time system, the HLT only performs partial event reconstruction and selection while detailed event selection and analysis are left to offline systems.

The HLT is a complex system that consists of data analysis and data transport components, as such problems are expected to arise during data processing. Hence there is a need to visualize data during its reconstruction (particle tracks, hits, fired pads in the ALICE detector, etc) while the HLT is running. This will help to identify and isolate problems as early as possible and raise appropriate alarms if errors occur during processing.

This thesis describes the monitoring system that has been developed for the HLT component of the ALICE muon spectrometer. The objectives of the muon HLT monitoring system are to give the user the ability to visualize reconstructed events and to inform the user of any errors occuring during the reconstruction process. The monitoring system will read reconstructed event data from the muon HLT publisher-subscriber[15] framework and display it in the ALICE Event Visualization Environment (ALIEVE).

1.4 Thesis outline

Chapter 2 provides the general background on the ALICE trigger system, the fast trigger (so called L0 trigger), and the muon HLT system. The design and implementation of the muon HLT monitoring system are described in Chapter 3, while Chapter 4 gives results obtained from implementation tests. The remarks, comments and conclusions are given in Chapter 5.

Chapter 2

Background

2.1 Summary

Two beams of nuclei traveling in opposite directions will collide at ALICE interaction vertex and as a result of the collision several particles including muons will be produced [7]. The interaction vertex is located at the origin of the ALICE coordinate system and it is that point shown in Figure 2.1 where particles that enter, and hence make tracks (shown as yellow lines in Figure 2.1), in the ALICE detector originate. The particles that enter the muon spectrometer (shown in Figure 2.2) of ALICE will go through the front absorber (discussed under Section 1.2.1) then pass through the first and the second tracking stations of the muon spectrometer and as they pass through they make hits (energy deposits) on the chambers of the tracking stations. The locations of the tracking stations are shown in Figure 2.1 where labels ST1 to ST2 are short forms for station 1 to station 5. Figure 2.3 shows tracking station 2 in more detail and shows its front end electronics.

Particles that emerge from the second station enter the third tracking station which is located inside the dipole magnet. From the third station the particles proceed to the fourth and the fifth tracking stations. Figure 2.4 shows tracking stations 4 and 5. After the fifth tracking station particles proceed to the muon filter wall. Only particles which are muons will be able to pass through the muon filter wall and the rest of the other particles will be trapped. The muons which have been able to pass through the wall will proceed to the first and then to the second trigger stations where they will make hits as they pass through the chambers of the trigger stations. The trigger stations of the muon spectrometer are shown in Figure 2.5.

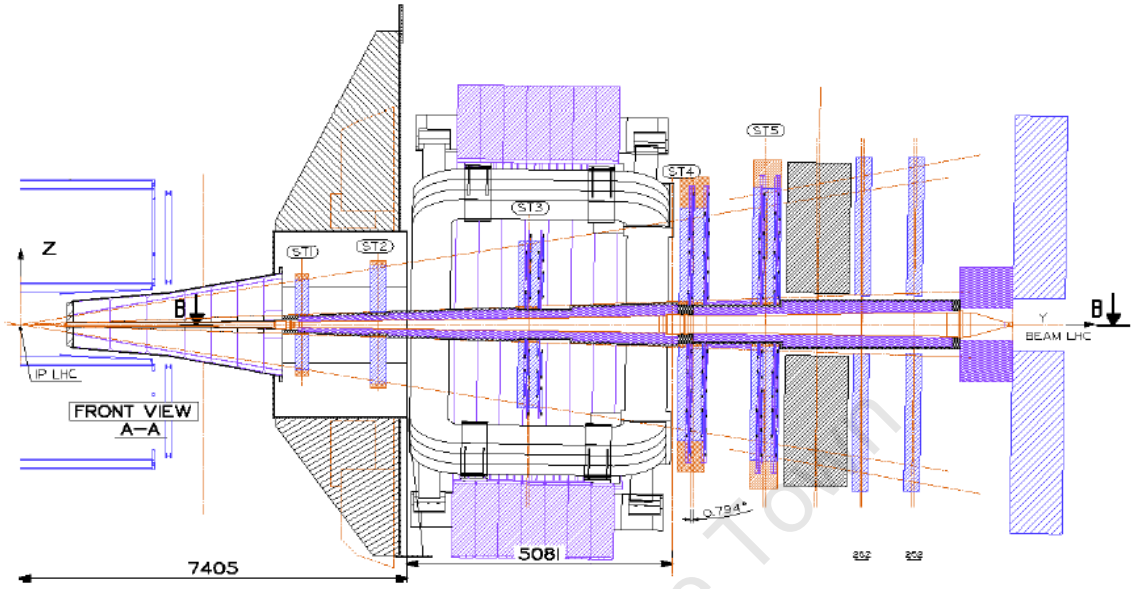

Figure 2.1: Schematic diagram of the muon spectrometer [7].

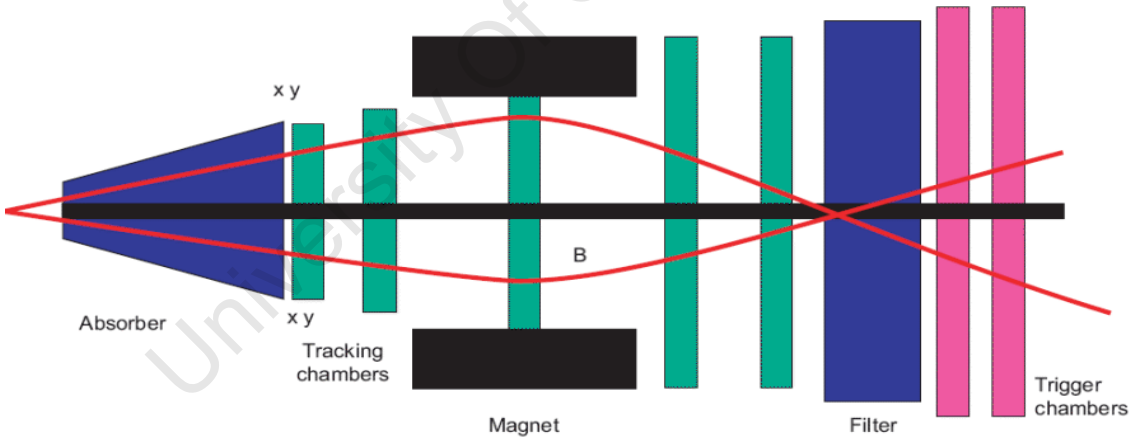

Figure 2.2: Sketch of the muon spectrometer with individual parts labeled [16].

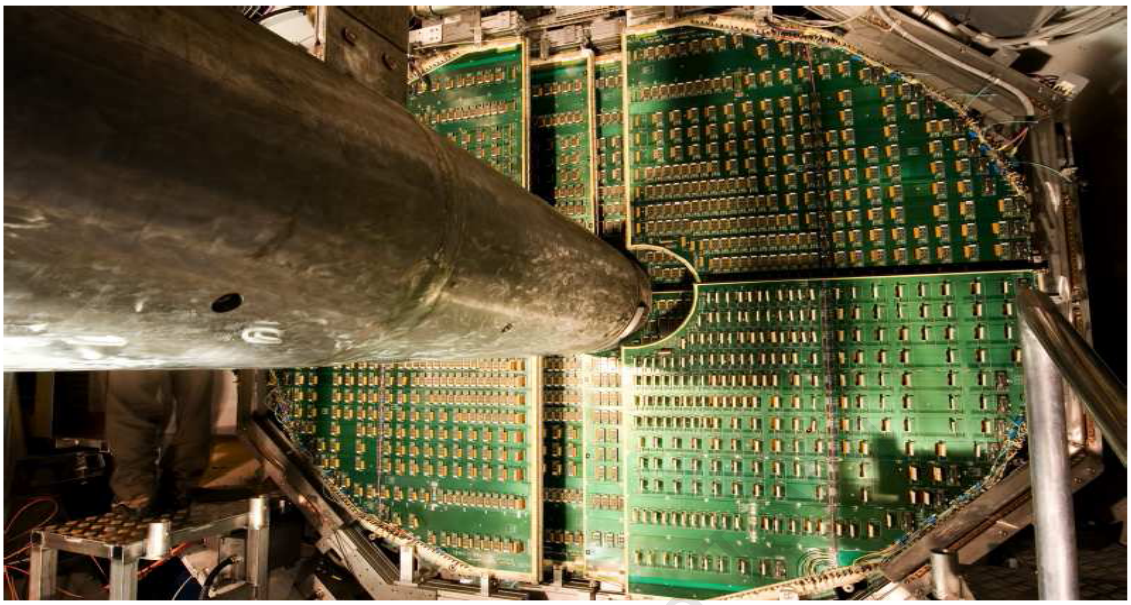

Figure 2.3: Station 2 of the muon spectrometer [7].

Figure 2.4: Stations 4 and 5 of the muon spectrometer [7].

In order to determine whether the particles that reached the trigger stations are muons and if they are muons whether their momenta are above

threshold, data will be collected from both the tracking stations and trigger stations and be analyzed to find the exact locations on the tracking and trigger chambers where the particle hitted the muon spectrometer. After finding the hits on the chambers, tracking of the particle occur. This involves correlating hits found on different tracking chambers with the aim of finding hits that originate from the same particle. Hits that come from the same particle are said to make track segment. This track segment is matched with another track segment, called trigger record, formed by hits found on the trigger chambers and if the two track segments match they are joined together to form a muon track. The tracking process is done by both the hardware trigger system discussed under Sections 2.2 and 2.3 and by the software trigger system discussed under Section 2.4.

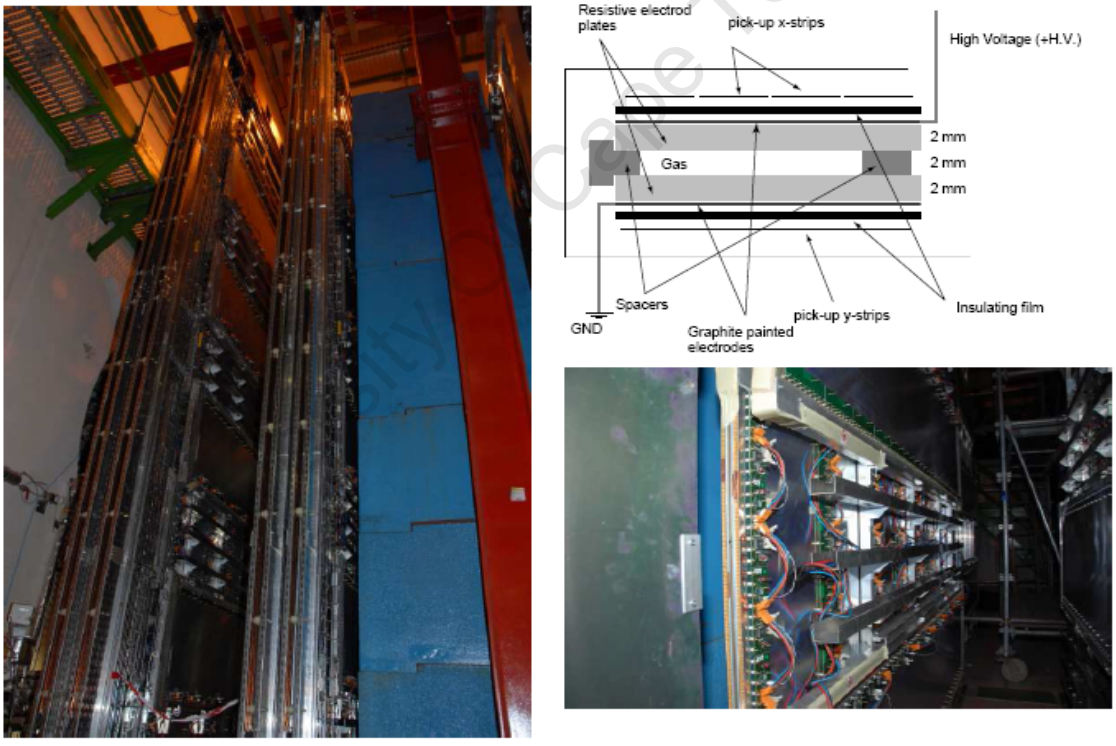

Figure 2.5: Trigger stations of the muon spectrometer [7].

2.2 ALICE trigger system

The primary goal of the ALICE trigger system [13] is to select interesting physics events[1]. It is designed to operate in different running modes such as proton-proton, nucleus-nucleus, proton-nucleus or nucleus-proton. The trigger system is made up of the Central Trigger Processor (CTP) and 24 Local Trigger Units (LTU) for each sub detector. The CTP is made up of seven different types of 6U VME boards.

The CTP takes inputs from the trigger detectors, process them and outputs the trigger decisions, one of which could be the command to read data from the Front End Electronics (FEE). The three output trigger levels are Level 0 (L0) which is the fastest trigger delivered in 1.2 μs, Level 1 (L1) which is delivered in 6.5 μs and Level 2 (L2) which is delivered at 88 μs.

The ALICE hardware trigger system selects events by means of applying a boolean logic of trigger signals. A more detailed and advanced selection of events is done by means of the high level trigger software, which is discussed in section 2.4. The output signals from the CTP are delivered to the front-end electronics of the detectors through the LTU. If the processed event is accepted by CTP, then data that is sent to the detectors specifies the event identifier, the trigger type and sets the detector to be read out by the Data Acquisition System (DAQ) [13]. In this study we focus our attention only to those discussions pertaining to the L0 trigger, the HLT and related topics.

2.3 The L0 trigger algorithm

The ALICE muon spectrometer trigger system participates at the level 0 (L0) of the ALICE trigger hierarchy or chain. The L0 muon trigger algorithm is implemented in hardware by the front end electronics of the trigger stations (MT1 and MTs).

The L0 algorithm, using information on the four RPC chambers of the trigger system, searches for tracks which point back to the interaction vertex. The algorithm is performed in two parts; L0-X is performed in the bending plane while L0-Y is performed in the non-bending plane [17]. Both L0-X and L0-Y require that a track hits or fires at least three out of four RPC detector

[1]Collision of two nuclei or data coming from this collision is called event

planes and this is called 3/4 coincidence level [18].

The non-bending plane algorithm, L0-Y, reduces low p_T muon background, i.e. muons with p_T less than 1 GeV/c. The bending plane L0-X algorithm estimates the bending of the track in the magnetic field. The bending is estimated with respect to the line of the track with infinite momentum pointing towards the interaction vertex and the deviation between the two tracks, δY_2 (see Figure 2.6), is measured [19]. This deviation is measured between trigger stations MT1 and MT2 as shown in Figure 2.6.

Two different cuts are performed on the deviation to reject low p_T muons which, obviously, have large deviations [20]. The cuts performed are the low p_T (l-p_T) cut, whose threshold is 1 GeV/c and the high p_T (h-p_t) cut whose threshold is 2 GeV/c. The low p_T cut is used to identify muons coming from the decay of J/ψ while the high p_T cut is used to identify muons coming from the decay of Υ. After the L0 algorithm has been performed one of the following trigger signals are issued to the CTP at approximately 800 ns after the interaction:

- At least a single muon above low (or high) p_T cut has been determined.
- At least two muons with different signs (unlike signs), each with p_T above low (or high) p_T cut have been found.
- At least two muons with the same signs (like signs), each having p_T above low (or high) p_T cut have been found.

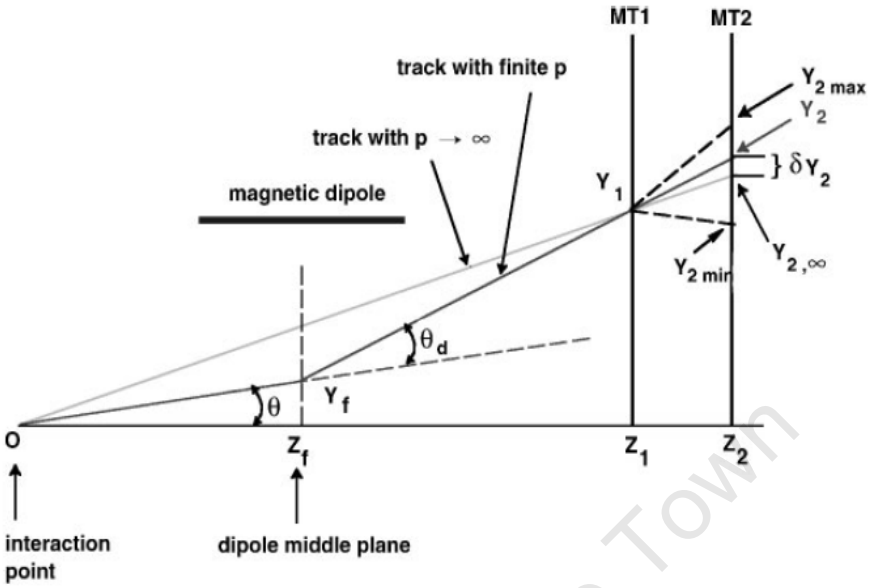

Figure 2.6: Illustration of the L0-X algorithm [19].

2.4 The ALICE high level trigger

As mentioned in the introduction, the ALICE detector is expected to produce data at the rate (25 GBytes/s) which is far above the rate allowable for storage (1.25 GBytes/s). This requires a level of triggering which is much faster and can reduce data rates without compromising the physics. This is the role of the High Level Trigger (HLT) [21]. Its main functions are to

- Select interesting events based on detailed on-line analysis.
- Select a region of interest in the event to compress data.
- Compress data by lossless data compression [22].

The HLT cluster is a farm of about 400 SMP off-the-shelf computers, connected with a high bandwidth and low overhead network [23]. Its functionality is illustrated in Figure 2.7.

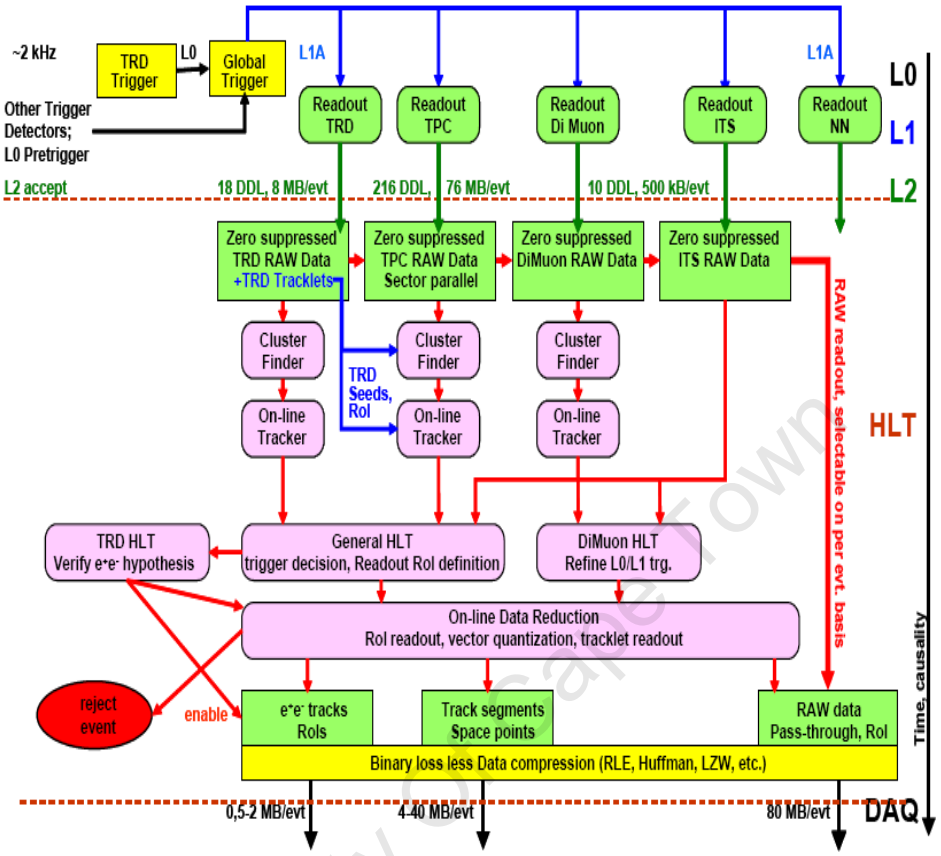

Figure 2.7: High Level Trigger Architecture [13].

The Raw data from the FEE of the detectors is send to the DAQ via Detector Data Links (DDL) and it is copied from the DAQ to the HLT. The HLT processes raw data in real time and gives a trigger decision before sending the output data, with reduced data rates, back to the DAQ. The raw data received by the HLT does not contain events rejected at or before the L2 trigger level [13]. The sequence of steps in the analysis of data inside the HLT is shown in Figure 2.8.

The HLT analysis process works in the following way: first the detector specific hit and track reconstruction processes take place followed by event reconstruction which takes input data from all sub-detectors and reconstruct the event. After the event reconstruction, a trigger decision is issued. The trigger decision could be a command to instruct DAQ to read reconstructed event data from the HLT or it could be an instruction to send reconstructed

data to be further compressed by lossless data compression.

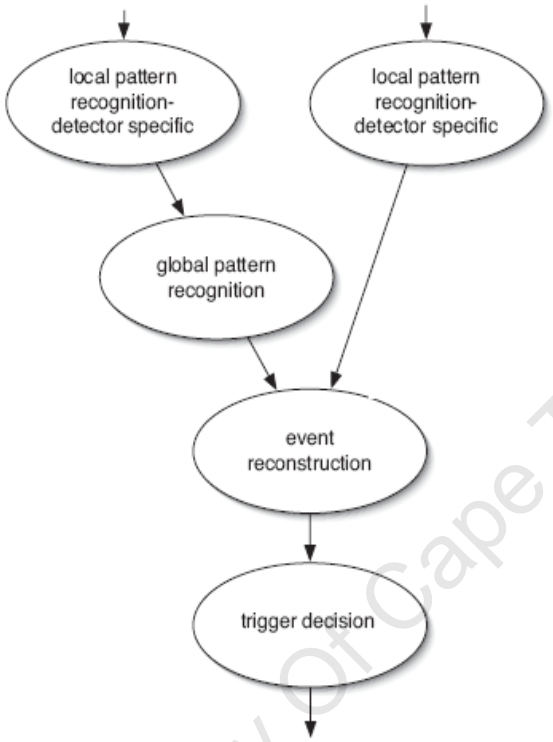

Figure 2.8: HLT functionality hierarchy [13].

As stated above, the HLT processes data from all major detectors, in essence each detector of ALICE has a subset of HLT designed specifically for it. This report discusses the HLT component for the muon spectrometer and its architecture is shown in Figure 2.9. The main components of the muon HLT are the communication framework[2] , muon HLT hit reconstruction, tracking, as well as the monitoring components. The detailed description of these components can be found in [24, 25], here we only give a brief summary.

[2]The communication framework is not only for muon HLT but is it an ALICE-wide HLT standard

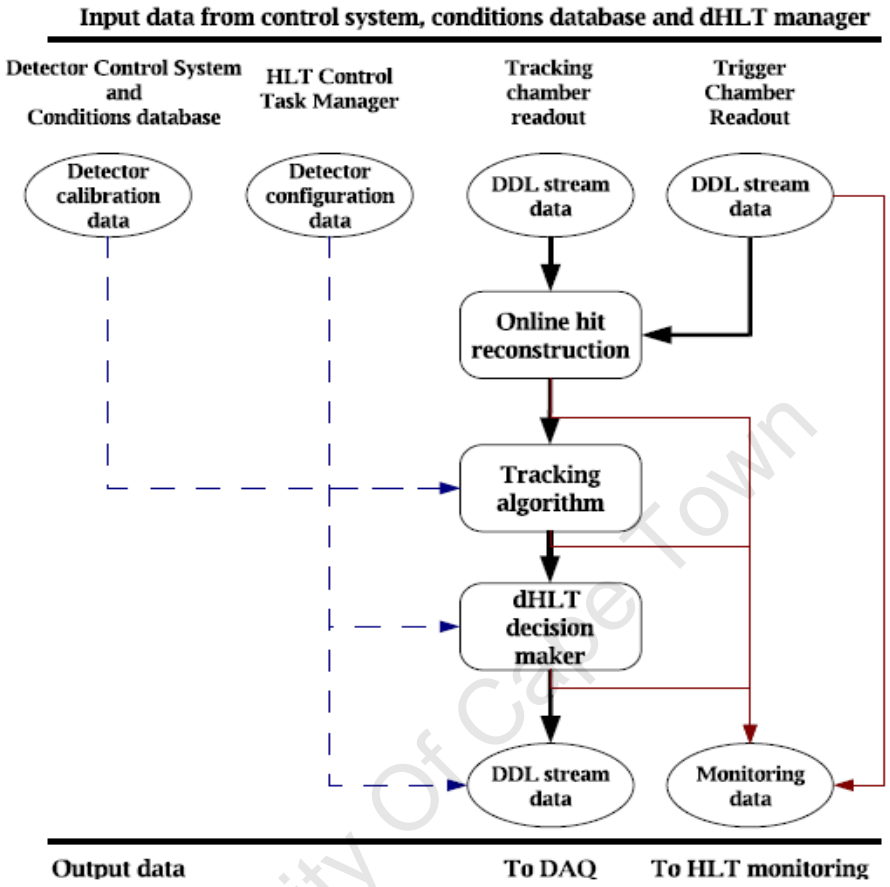

Figure 2.9: Muon high level trigger architecture [24].

As shown in Figure 2.9, the muon HLT receives calibration and configuration data from the Detector Control System (DCS) and the HLT Control Task Manager, respectively. After muon HLT data analysis, the processed data is sent to the DAQ. To facilitate communication between muon HLT and other on-line detector integrated systems (DCS, DAQ, etc), various interfaces have been defined [26].

2.4.1 The muon HLT communication framework

The communication framework, known as the Publisher Subscriber (PubSub), used for data transportation in the HLT analysis chain has been implemented based on publisher-subscriber principle [15, 27]. The framework has several components that can be connected in different setups according

to the specified configuration at the start of the analysis chain and the configuration is changeable at the run time.

The framework consists of the publishers (data producers) and the subscribers (data consumers). The subscriber notifies the publisher if it is interested in the data and in response the publisher will always announce new events to its subscribers. To avoid communication overhead, the data is not moved between processes within the same computer node but it is stored in the shared memory segment where all processes can access it. In this case it is only the data descriptors that are moved. However when the publisher and the subscriber are on different computer nodes data is moved from one node to another with the help of the bridge component described below. The major components of the publisher subscriber framework are briefly discussed in the sections below.

2.4.1.1 Persistent and transient subscribers

There are two kinds of consumers or subscribers supported by the framework: **Persistent or blocking subscribers**: These type of consumer keep data until they have finished processing it and then notify the publisher. A publisher timeout can be specified for persistent subscribers so that they do not hold data indefinitely. However care is taken to ensure that the timeout is large enough not to out-run the consumer in the normal operation.

Monitoring or transient subscribers: For this type of consumer, the publisher can force an early release of data even though the consumer has not finished processing it. This is to make sure that monitoring subscribers do not interfere with the normal operation of the analysis chain. However the timeout for monitoring subscriber should be enough to finish large part of the event processing.

2.4.1.2 Analysis components

Data inserted in the analysis chains is processed in different steps implemented in separate analysis components. Analysis components, depicted in Figure 2.10, are located centrally in the analysis chain and each component has the subscriber to receive data and the publisher to insert data back into the analysis chain after being processed. As stated above, data is stored in the shared memory segment hence the subscriber and the publisher receive

and publish, respectively, only data descriptors.

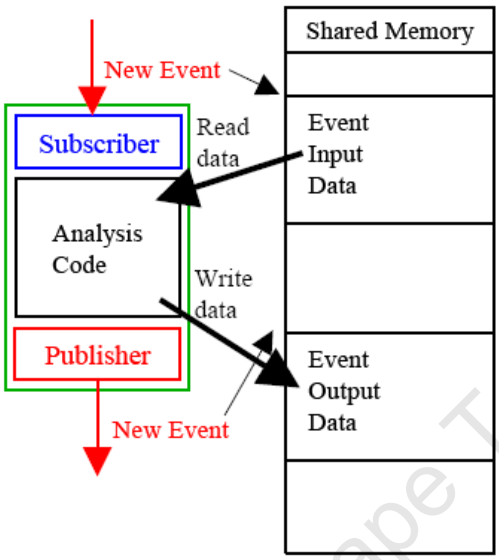

Figure 2.10: Analysis Component [13].

2.4.1.3 Data transport components

The HLT analysis chain is complex. Data comes from multiple producers and sometimes it has to be merged into one single stream. For the purpose of load balancing, data is scattered so that it can be processed by multiple central processing units (CPU). The discussion below concentrates on the components which are used to achieve this goals.

Event scatterer: Has one subscriber to receive incoming stream of sub-events and a number of publishers on which the event is distributed. The event scatterer is used to distribute events to many subscribers for load sharing. The data received by the subscriber of the event scatterer is published through one of the publishers depending, on the algorithm implemented between the subscriber and the publishers. The event scatterer is shown in Figure 2.11.

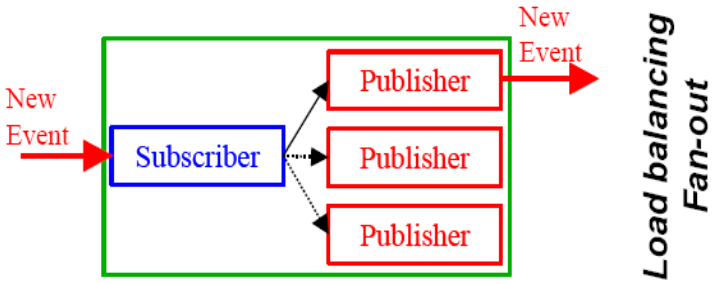

Figure 2.11: Event Scatterer [13].

Event gatherer: Shown in Figure 2.12 is the inverse of the event scatterer in the sense that it has multiple subscribers and a single publisher. It receives the event streams previously scattered by the the event scatter via its subscriber components and it re-publishes them through its one publisher. In this way the streams are united into a single stream again.

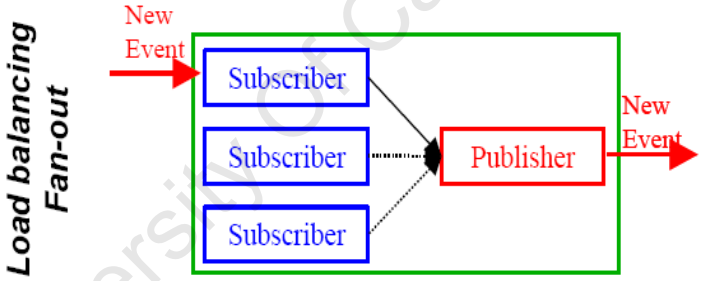

Figure 2.12: Event Gatherer [13].

Event merger: This merger is shown in Figure 2.13. Like the event gatherer, it has multiple subscribers and one publisher, however, unlike the event gatherer, the event merger builds and publishes a single event descriptor out of the descriptors it receives from the subscribers. The application of the event merger is in the track finding stage where reconstructed hits as well as the coordinates of the interaction vertex all coming from different preceding processing stages in the HLT chain are required [13].

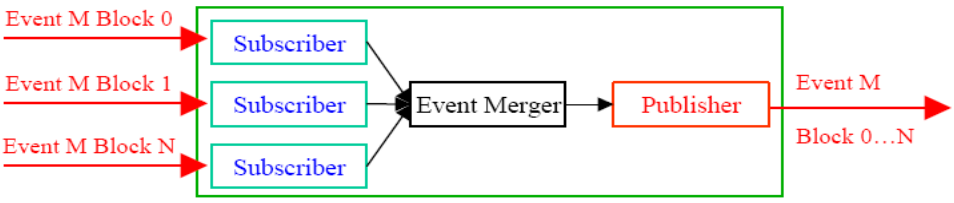

Figure 2.13: Event Merger [13].

2.4.1.4 Bridge components

The publisher-subscriber components described above do not copy or move data but they only move data descriptors. This is because these components run on the same computer. However when data is to be processed by publisher-subscriber components running on the different computer then data has to be copied to the other computer. The bridge component allows the publisher to send data to the subscriber on the peer computer node without the publisher and the subscriber noticing whether they are connected directly or through the bridge.

The bridge component has SubscriberBridgeHead and PublisherBridgeHead components. The SubscriberBridgeHead is used to connect to the publisher whose data is to be copied. The SubscriberBridgeHead connects and send data to the PublisherBridgeHead component on the peer computer node which uses its publisher to publish data to the subscribers. This is shown in Figure 2.14. The communication between the bridge components is carried out by the communication class library developed for the publisher subscriber framework.

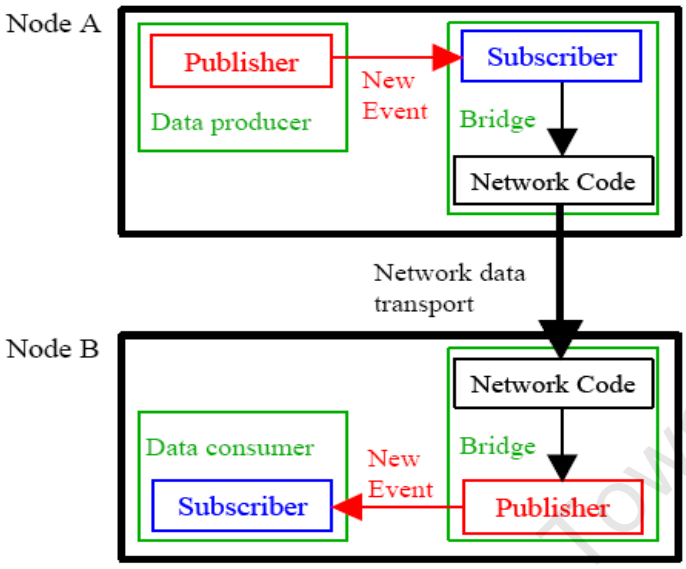

Figure 2.14: Bridging Component [13].

2.4.2 Hit reconstruction component

The hit reconstruction module of the muon HLT is composed of the subscriber, the hit reconstructor and the publisher components. Hit reconstruction can be done either on the HLT Read Out Receiver Card (H-RORC) or on the physical memory of the Front End Processors (FEP) of the HLT cluster [28].

The hit reconstruction algorithms first unpack raw data with the help of the Look Up Tables (LUT), from the DDLs into the corresponding pad positions of stations 4 and 5 which are the only stations used by the Manso tracking algorithm described below. The full details of the format of raw data from the DDLs is explained in [29].

The algorithms search for neighboring pads with non zero charge and form a cluster. The cluster is made up of at least three fired pads because a muon track is only considered to be valid and therefore generated if it fired at least three neighboring pads. The cluster has a unique central charge which is the charge of the pad (within the cluster) with the greatest amount of charge. The hit is reconstructed from the cluster by calculating the center of mass of the pad charges in the cluster and the value obtained is considered as the coordinates of the hit position [28, 30]. For example, the hit position

for a cluster consisting of three pads is calculated as follows:

$$\vec{R} = \frac{q_i\vec{r}_i + q_{i+1}\vec{r}_{i+1} + q_{i+2}\vec{r}_{i+2}}{q_i + q_{i+1} + q_{i+2}} \tag{2.1}$$

$\vec{R}$ is the reconstructed hit position, q_i and r_i are the charge and the position of the i'th pad, respectively. The reconstructed hit information (coordinates x, y, x and charge) is then published to the tracking component of the muon HLT for track reconstruction (section 2.4.3). A detailed description of the algorithms is given in [28].

2.4.3 Tracking component

The tracking component reconstructs muon tracks from the hit coordinates passed to it by the hit reconstructor component. Thus, the tracking component has a subscriber to receive published hits from the hit reconstructor component and it also has a publisher to publish its analysis results. There are two algorithms proposed for the implementation of the tracking component. The Manso algorithm is the one that has been fully developed and used in the tracking component while the hierarchical algorithm has been proposed as the alternative. The two algorithms are described in the sections below.

2.4.3.1 Manso algorithm

The trigger stations are not able to perform sharp p_T cuts because of the limited segmentation of the Resistive Plate Chambers (RPC)[31]. Therefore the Manso algorithm is tasked to correlate the hits between tracking stations and the trigger stations by performing partial tracking, through the muon filter wall, from the trigger stations to the two last tracking stations of the muon spectrometer. In this way the Manso algorithm improves the p_T resolution and hence the p_T cut. This section provides the summary of the Manso algorithm which is explained in details elsewhere [32].

The muon tracks found in the trigger stations are reconstructed and converted into trigger records (muon tracklets in trigger stations) with estimated information on position, deviation in the y direction and the transverse momentum. These trigger records serve as seeds to find valid track candidates on the tracking stations that penetrated the muon filter wall and hence formed

those trigger records. From each trigger record a vector is projected to the tracking chamber 10 thereby making an intersection point R_p with the tracking chamber, see Figure 2.15. A circular Region of Interest (RoI), associated with the trigger record, with the radius $R_s = a*R_p+b$ is formed on the tracking chamber. Only RoI with fired pads are kept and the rest are rejected. The constants a and b have been determined with AliRoot simulation of hits produced by muons from the decay of J/ψ resonance as shown in [32] and they are chosen to optimize signal identification and to reject background. Table 2.1 shows the values of a and b for stations 4 and 5 of the muon spectrometer.

Station 5	$a_5 = 0.020$	$b_5 = 3.0$
Station 4	$a_4 = 0.016$	$b_4 = 2.0$

Table 2.1: Optimized parameters used to define the RoI for the muon HLT tracking algorithm

For each hit found inside the RoI on chamber 10 a vector is projected to chamber 9 and the RoIs on chamber 9 are determined. The same sequence of steps is used to find RoIs on chambers 8 and 7. In short, the hit is found by projecting a line from chamber n to chamber n-1 to form RoI in chamber n-1. If the hits are not found, the line is extended to chamber n-2 until the RoI is found.

The muon tracks are reconstructed from the found hits. Only tracks with at least 2 out of 4 expected hits on the tracking stations are kept. Two sets of p_T cuts are then applied to these tracks; low p_T cut (threshold of 1 GeV/c) to identify tracks from decay of J/ψ and high p_T cut (threshold of 2 GeV/c) to identify muon tracks from decay of Υ resonance. Only tracks with transverse momentum greater than the cuts' thresholds are kept and other tracks are rejected as background. If at least two muon tracks with opposite signs and momenta above the threshold are found by the algorithm, then the muon HLT is validated.

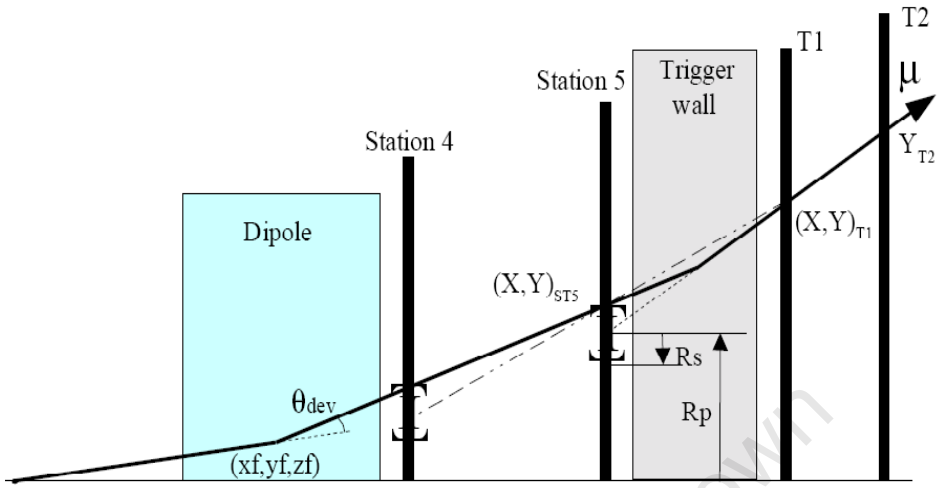

Figure 2.15: Schematic depiction of the muon spectrometer and the Manso algorithm [32].

2.4.3.2 Hierarchical algorithm

Like the Manso algorithm, the hierarchical algorithm uses regions of interests to reconstruct tracks, however unlike the Manso algorithm it does not use the trigger records as seeds to reconstruct tracks but it uses only hits found on the tracking stations. Furthermore, to improve momentum resolution, the hierarchical algorithm extends the tracking process to tracking station number one [24].

In this algorithm, tracking is done in independent stages and this makes it possible to parallelize the code that implements the tracking algorithm. Background tracks are rejected as early as possible by applying different cuts at early stages but care is also taken not to mistake a valid track for a background track. Hence a track is only discarded after failing various cuts imposed on it. Cuts requirements performed in this algorithm are:

1. **Polar angle cut**: Two hits, A and B, from consecutive chambers of the station are joined by a vector $\vec{AB}$. The angle between the vector $\vec{AB}$ and the z axis is called the polar angle.

2. **Deflection angle cut of the trajectory**: For three hits A, B and C from three consecutive chambers the vectors $\vec{AB}$ and $\vec{BC}$ are formed. The deflection angle is the angle between the two vectors $\vec{AB}$ and $\vec{BC}$.

3. **Vertex distance cut**: A vector joining the two hits A and B from two consecutive chambers of station 1 is extrapolated towards the interaction vertex. Then, a cut on the distance of the closest approach $|\vec{d}|$ to the primary vertex can be used to eliminate many background tracks which do not originate from primary vertex.

4. $5/10^{th}$ **cut**: For a muon track to be accepted as valid, at least one hit from every tracking station must be found.

In the hierarchical algorithm, tracking process is done in several stages and cuts are imposed on the reconstructed track fragments at each stage. These tracking stages are described below.

1. **Segment finding stage**: On each and every tracking station a line segment between hit h_i on the first chamber and hit h_j on the second chamber is reconstructed. Then the polar angle cut is applied and the segments which do not pass the cut are discarded. However the hits which are not matched are not discarded but are carried forward to a next tracking stage. At this point the algorithm tries to match these unmatched hits to those of the next station. This is done in order to account for the case where the track made only five hits instead of ten on all the tracking stations (the spectrometer has 10 tracking chambers so a muon track is expected, by default, to make at least one hit in each chamber).

2. **Tracklet finding stage**: A tracklet is defined as hits from two consecutive stations that have been matched to form a section of a track candidate. The segment, s_i, from the first tracking station is joined to the first hit of the segment, s_j, on the second tracking station to form a tracklet. Similarly, the segment, s_j, on the second tracking station is matched with the last hit of the segment s_i on the first station. The deflection angle cut is then done in order to eliminate fake tracklets. Furthermore all hits, h_i, that were not matched on the segment finding stage are matched with the segment s_j to form tracklets and the deflection angle cut is done. Tracklets which do not pass the cut are discarded. Similarly, the unmatched hit h_j on the second station is joined with the track segment s_i from the first station to form tracklets and the tracklets which do not pass the deflection angle cut are discarded.

The hits on the first station which are still not matched are joined with unmatched hits on the second stations to form tracklets. The polar angle cut is applied and the tracklets which do not pass the cut are discarded. The hits which are still unmatched are then discarded.

3. **Vertex matching**: The tracklets found on the station 1 and 2 are extrapolated toward the interaction vertex and the cut on the distance of closest approach is done. The tracklets which do not pass this cut are rejected and others are passed on to the first merging stage.

4. **Merging stage 1**: There are four sets of tracklets available at this stage. Tracklets $L_i^{12}, L_i^{23}, L_j^{34}$, and L_j^{45} reconstructed on stations 1 and 2, stations 2 and 3, stations 3 and 4 and stations 4 and 5, respectively. Then track fragment F_i^{123} is formed by merging tracklets L_i^{12} and L_i^{23} if they have the same hit coordinated on station 2. Track fragment F_j^{345} is formed by merging tracklets L_j^{34} and L_j^{45} if they have the same hit coordinates on station 4. All tracklets that are merged are discarded.

5. **Merging stage 2**: This stages involves merging track fragments into complete tracks T_i i.e track fragments F_i^{123} and F_j^{345} are merged into complete tracks T_i if they have the same hit coordinate on tracking station 3. All unmarged track fragments are discarded.

6. **Extended deflection angle cut**: The deflection angle cut is done on three consecutive hits of track T_i to further eliminate tracks whose deflections between hits do not conform to real muon tracks. If any three consecutive hits on the track do not pass the cut, the track is discarded.

7. **Track overlap resolving**: In this stage, tracks that share a hit coordinate are fitted with a straight line in the x-y plane and a cubic function in the y-z plane. The hit coordinate is kept only in the tracks that gives lowest χ^2 from the fit. The other tracks then have the shared hit coordinate removed from them and they are tested against the $5/10^{th}$ cut. These tracks get discarded if they fail the cut.

When the tracking process is complete, p_T and the signs of the reconstructed tracks are determined and then trigger decision is issued. If there

are at least two tracks with opposite signs and have momenta above the threshold (1 GeV/c for J/ψ and 2 GeV/c for Υ), a valid trigger signal is then issued. The summary of the hierarchical algorithm is given by Figure 2.16.

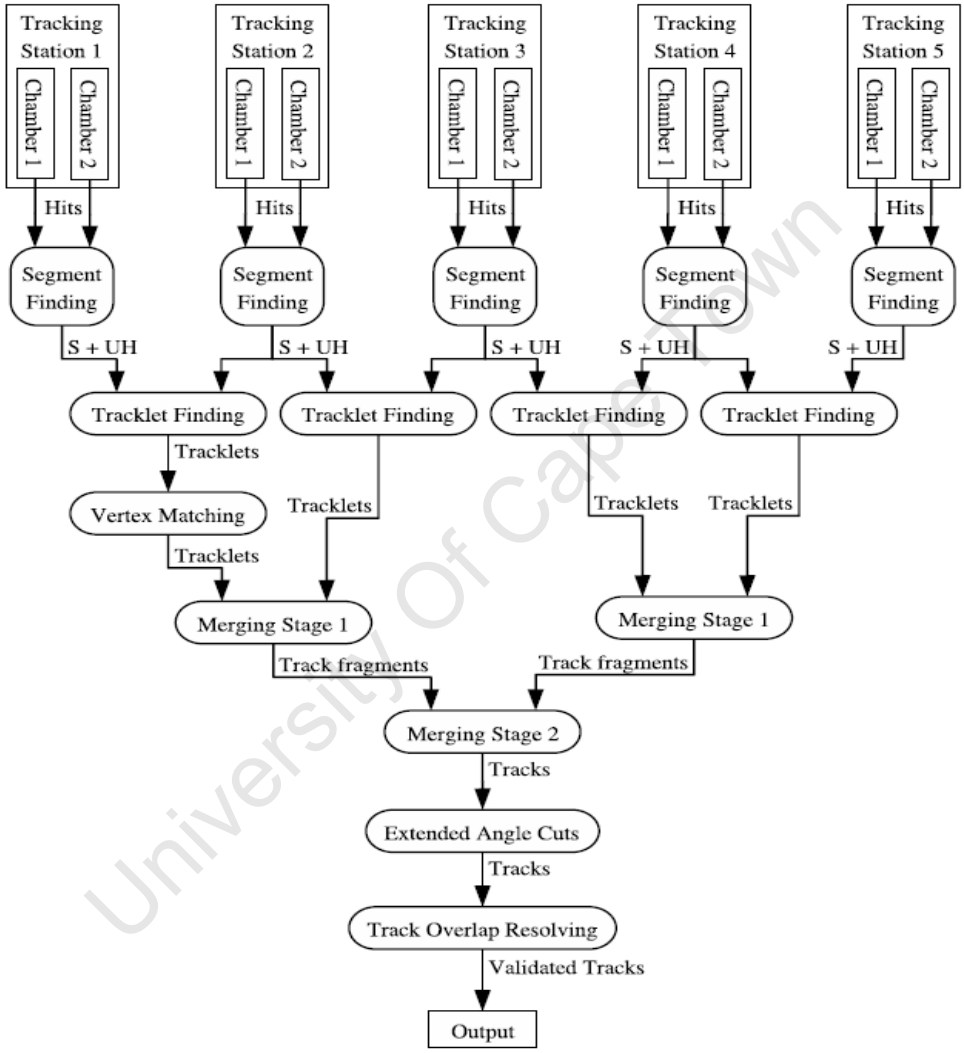

Figure 2.16: A summary of the hierarchical algorithm, where S denotes the line segment and UH means unmatched hits [24].

2.4.4 HLT physics performance

The muon HLT is designed to improve the L0 p_T cuts, specifically the low p_T cut (1 GeV/c) that is used to select muons from the decay of J/ψ and the

high p_T cut (2 GeV/c) that selects muons from the decay of Υ. In [33] the results showed that the implemented muon HLT is indeed able to improve the L0 p_T cuts as shown in Figure 2.17 where efficiency is plotted against p_T. This figure illustrates that the muon HLT cut at 1 GeV/c (red line) is sharper than the L0 cut implying that all the background below 1 GeV/c is significantly reduced, without compromising the physics. Similarly the muon HLT cut at 2 GeV/c (green line) reduces, significantly, the background of p_T below 2 GeV/c and clearly the HLT improves the efficiency of L0, too.

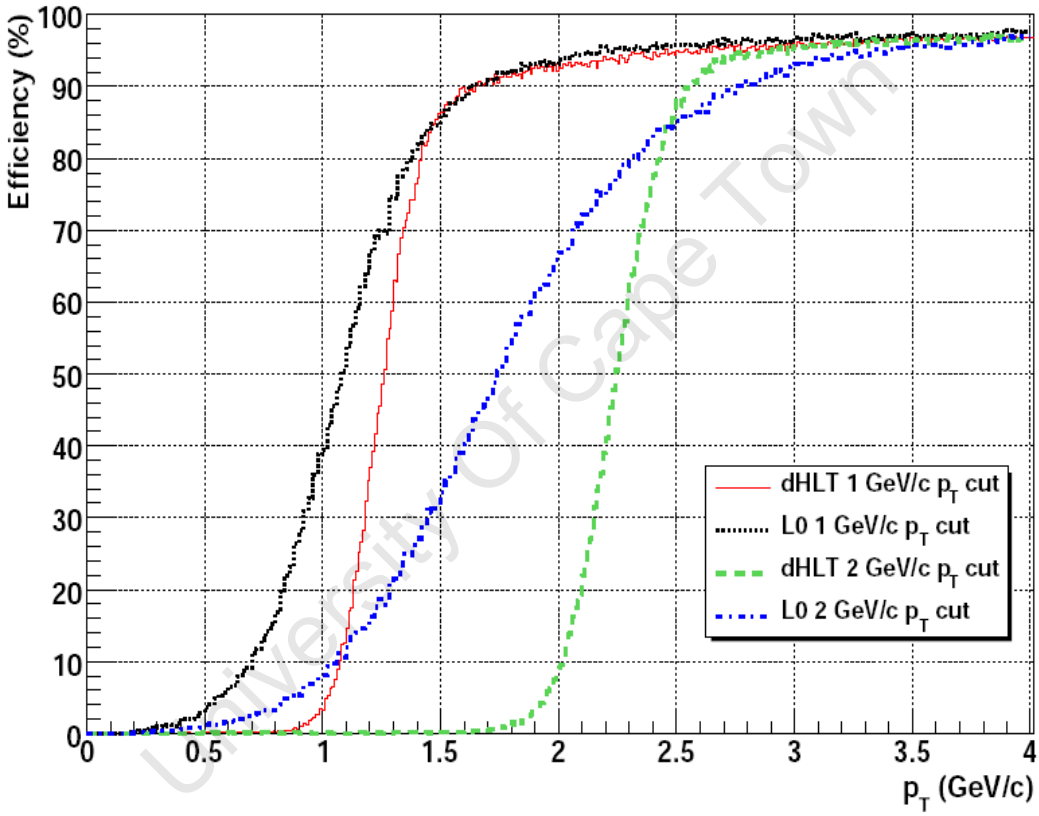

Figure 2.17: The L0 and HLT efficiency distribution functions for J/ψ and Υ signals [33].

2.5 HLT monitoring system

As muon HLT is running the user would want to visualize the events that are being reconstructed and would want to know, as early as possible, if some errors have occurred [34]. Furthermore, one would want to make plots or

histograms with data acquired from the HLT runs but, most importantly, the user would want to know, at any point in time, the connection status of the HLT process. This is the function/role of the HLT monitoring system. The implementation of the muon HLT monitoring tool is described in the next chapter.

Chapter 3

Implementation of the ALICE muon HLT monitoring system

The muon HLT monitoring system collects data to be monitored at every processing point of the analysis chain using the HLT Online Monitoring Environment including ROOT (HOMER) system. HOMER collects data either from a TCP[1] port of the processing computer node or from the shared memory segment [35]. In principle HOMER works like an oscilloscope in that it is plugged into the system and collects data to be monitored. The data collected is then displayed in ALICE Event Visualization Environment (ALIEVE).

The Unified Modeling Language (UML) diagram illustrating the relationship between the muon HLT monitoring components is shown in Figure 3.1. The monitoring system reads data from multiple data sources (data channels) which are the Pub/Sub file publishers, ROOT [36] files (Simulated data), and HOMER. The data channels are managed by the channel manager which also manages the communication between the data channels, event queue and ALIEVE (the Graphic User Interface (GUI)). When data is present in one of the channels, the channel notifies the manager which, in turn, reads the data and stores it in the event queue. The messages in the event queue are read and displayed by ALIEVE . ALIEVE or GUI has the buffer where the displayed data is stored temporarily so that it can be redisplayed upon request.

[1]TCP is a protocol used for data transportation over ethernet network

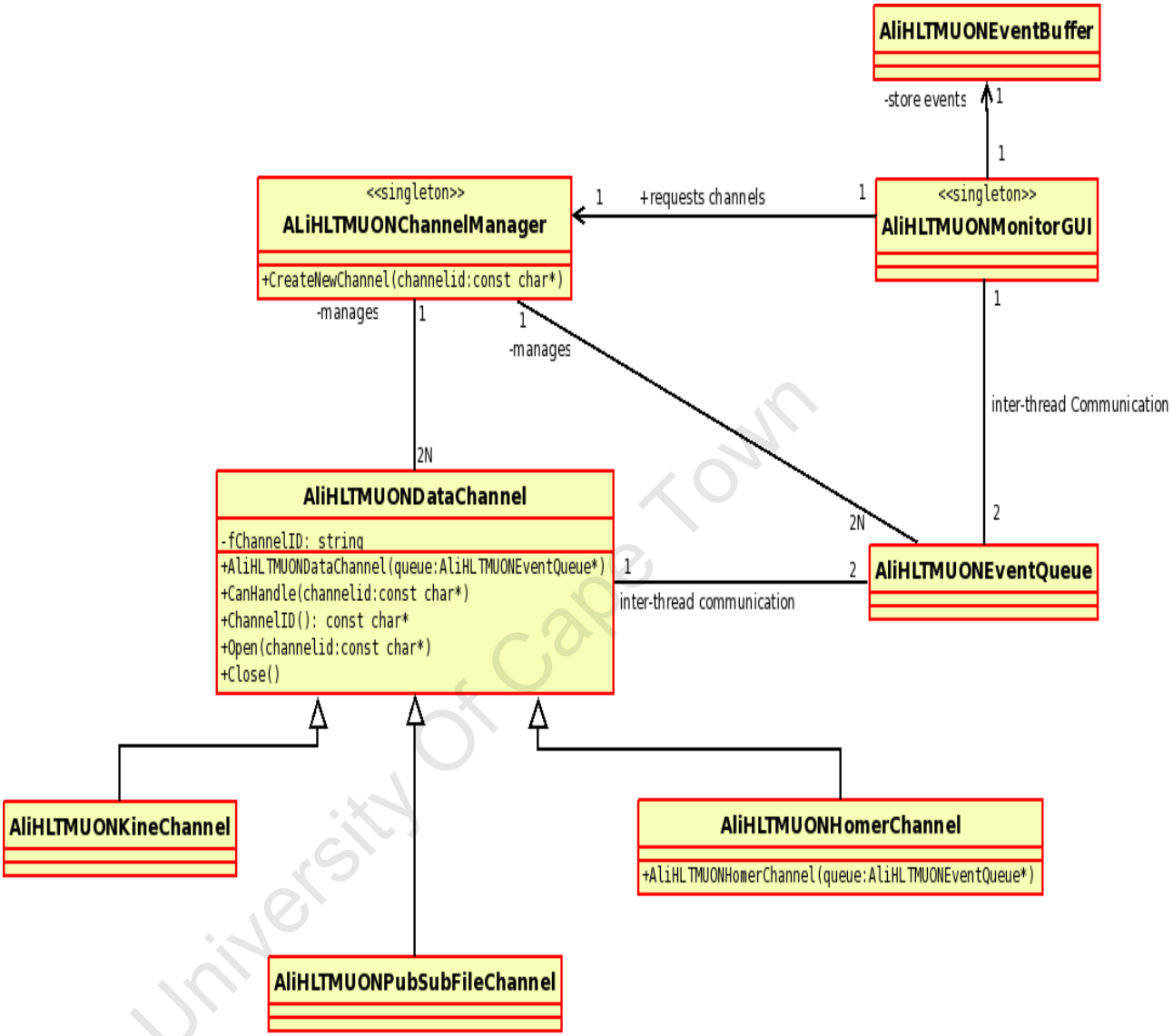

Figure 3.1: UML diagram illustrating the relationship between the components of muon HLT monitoring system

The prototype of Early Version of Muon HLT Monitoring System (EVMHMS) can be found in [37]. The main aim of this project is to improve some of the componets (Figure 3.1) of the EVMHMS in order to enhance the performance of the tool. In this project the message queue AliHLTMUONEventQueue was completely re-designed and the details of the implementation are discussed later in this chapter (section 3.5).

The GUI component of the EVMHMS was also modified since ALIEVE could not handle muon HLT data (tracks, hits, trigger records, etc) therefore one of the objectives of this project was to extend ALIEVE so that it can display these data. This part is described in section 3.4. In addition, the implementation of the utility (dHLTdumpraw [38]) that inspects with finer detail, the contents of all muon HLT internal data blocks and DDL raw data stream is described in section 3.3.

3.1 ALICE offline data processing framework

The data used for testing muon HLT monitoring system was generated using AliRoot[39] framework and the reconstruction was done using both the muon HLT analysis chain and AliRoot to get the muon tracks information.

The schematic summary of the offline data processing framework is shown in Figure 3.2. During processing, events are simulated via the simulation chain on the left hand side and the simulated data is reconstructed on the right hand side. The analysis chain is represented by the arrow labeled **comparison**. The simulation and reconstruction processes are explained in details in [39] and [40]. Only a summary of these processes will be given in this section.

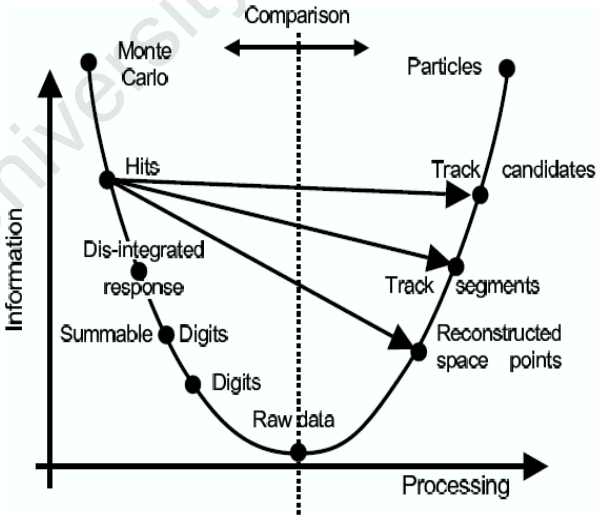

Figure 3.2: ALICE data processing framework [40].

The simulation of primary interactions is done using Monte Carlo event

generators and the transportation of particles through the detectors is done with the Geant3[2] [41] transport parkage. Particles will deposit energy (hits[3]) on the detectors as they are being transported. The hits will contain information about the particles that created them. These hits are then transformed into digits by taking detector response into account. There are two types of digits: summable digits and digits. Summable digits are predecessor of digits and they are high-resolution and zero-threshold digits and they can be summed when different simulated events are superimposed. In contrast, when forming digits real thresholds are used and background simulation is activated. The digits are then transformed into the raw data format, which basically is the output of the front end electronics of detectors.

After the simulation, the reconstruction and analysis chains are activated to evaluate the detector and software performances. Event reconstruction either takes raw data or digits as inputs and reconstruct tracks from them. The input data can either be real or simulated data. An example of a typical C++ macro used for offline reconstruction of events is given in section 3.1.2 while the code that was used for simulation in this work is given in section 3.1.1.

3.1.1 Data simulation

Simulation of data is done with the AliSimulation class which provides the simple user interface to the simulation framework. Below is an example of a C++ macro that implements the simulation process. This macro is available in AliRoot framework and users configure it differently to achieve simulation results they want. For all the work done in this project only p-p events were simulated.

```
001 void Simulation(int seed = 1234567, int nevents = 5, const
char* config = "$ALICE_ROOT/MUON/Config.C")
002 {
003   AliSimulation* MuonSim = new AliSimulation(config);
004   MuonSim->SetSeed(seed);
005   MuonSim->SetWriteRawData("MUON");
```

[2]Although Geant4 is also available as a transport package, Geant3 was used because it is a default transport package for ALICE and it has a lot of documentation and it is easy to install

[3]These hits are different from reconstructed hits of muon HLT. In muon HLT, hits are what is shown as reconstructed space points in Figure 3.2

```
006   MuonSim->SetMakeDigits("MUON");
007   MuonSim->SetMakeSDigits("MUON");
008   MuonSim->SetMakeDigitsFromHits("");
009   MuonSim->Run(nevents);
010 }
```

A pointer to the simulation object is declared and initialized and the name of the detector configuration file[4] is passed to the simulation object. The seed number that will be used by the generators in creating the particles is then set. After this, the simulation object is configured to produce raw data and to create summable digits and digits and it is also set not to make digits directly from hits. Several output files are produced from the simulation, including `galice.root`[5] and `raw.root` which are used during the reconstruction phase.

3.1.2 Event reconstruction

The reconstruction of tracks is done with the AliReconstruction class which provides a simple user interface to the reconstruction framework. The output from the reconstruction process is the Event Summary Data (ESD) containig information about the reconstructed muons. ESD is stored in the file called `AliESDs.root` and the larger the number of muons reconstructed the bigger will be `AliESDs.root` file. An example of a typical reconstruction C++ macro is given below.

```
001 void Reconstruction(const char* input = "./")
002 {
003   AliReconstruction* MuonRec = new AliReconstruction("galice.root");
004   MuonRec->SetInput(input);
005   MuonRec->SetRunVertexFinder(kFALSE);
006   MuonRec->SetRunReconstruction("MUON");
007   MuonRec->SetNumberOfEventsPerFile(1000);
008   MuonRec->Run();
009 }
```

Like in the simulation process, a pointer to the AliReconstruction object is declared and initialized. Line 4 specifies the directory where the DDL raw

[4]Configuration file is a C++ macro that creates and configures the monte carlo object, the generator object and detector modules and the magnetic field maps. It is run before simulation

[5]galice.root file stores configuration and management objects and header data (Information describing the event and state of detector)

data files to be used in reconstruction can be found. Then the reconstruction of primary vertex is switched off by line 5. On line 6 the AliReconstruction object is set to do local reconstruction, tracking and creation of ESD tracks and filling of additional ESD information. Finally the number of events per digits/clusters/tracks file is set to 1000 and the reconstruction process begins.

3.2 Running the muon HLT chain

The muon HLT analysis chain is executed after the simulated raw data is generated. This chain is used to reconstruct muon HLT hits (reconstructed space points) and tracks using the algorithms described in sections 2.4.2 and 2.4.3.

The chain requires a configuration file (in XML format [42]) which contains information about data sources, processor components and data sink. Data source components are the DDL raw data file publishers while processor components are the hit reconstruction component and tracking components. The data sink is the component where data processed by the muon HLT chain can be obtained by the monitoring systems. This component can either be ShmDumpSubscriber (uses shared memory for data output) or TCPDumpSubscriber (uses TCP port for data output). For this project, TCPDumpSubscriber was used because it allows the user to access data from the computer running HLT chain over the network. In this case monitoring of data can be done from a computer which is separate from the one processing data.

A Python [43] program called MakeTaskManagerConfig.py uses the XML configuration file as an input to generate other configuration files for the TaskManager (TM) [44] control software. The TM is responsible for the control of all muon HLT chain components.

A running chain is controlled by the TM GUI program called TMGUI.py implemented with Python. Like MakeTaskManagerConfig.py, TMGUI.py is also available on the HLT software. A snap shot of TM GUI depicting a running muon HLT chain is shown in Figure 3.3. In this figure the states of the children are **busy** and **running**. This means that the muon HLT chain is running very well and output data can be obtained from the sink component.

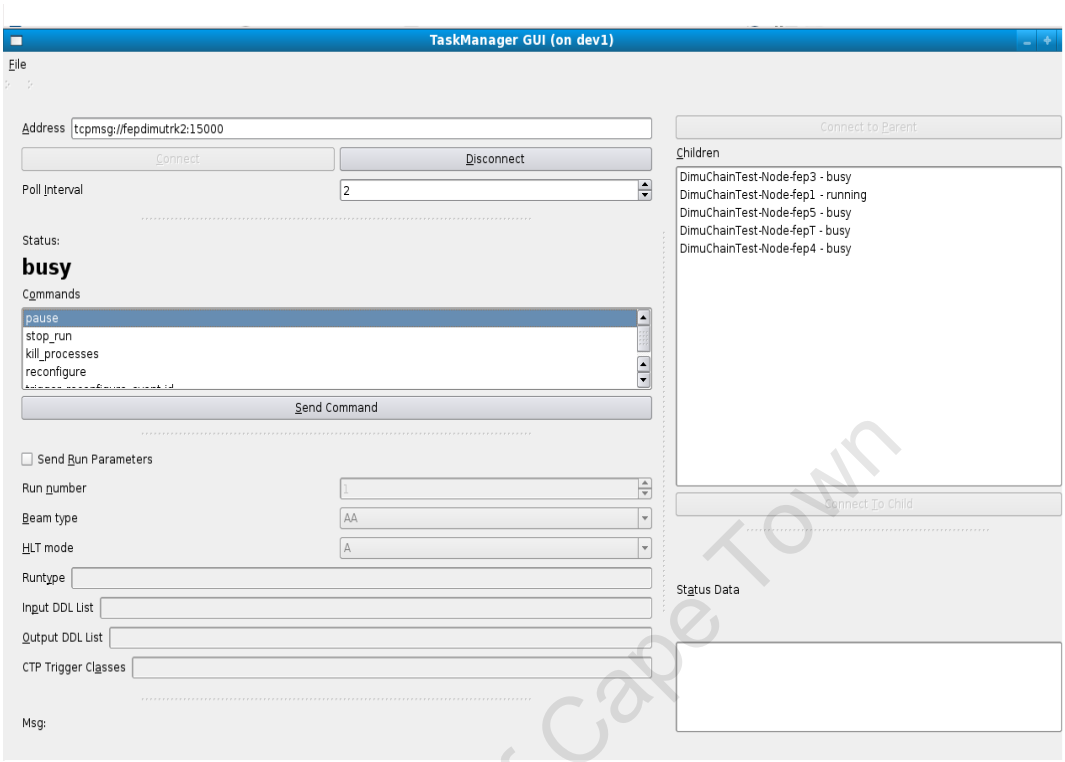

Figure 3.3: Snapshot of a running muon HLT chain

3.2.1 Reading data from the muon HLT chain

The data to be monitored is read from a running muon HLT chain using HOMER. HOMER connects and reads data from the TCPDumpSubscriber data sink using the TCP protocol. An example of a C++ code to connect HOMER to the TCPDumpSubscriber and then read the next available event is given in Appendix A.

3.2.2 Reconstructing events using the RunChain.C macro

Another way of reconstructing events is by using the C++ macro RunChain.C [45]. In this method HOMER is not needed to read the reconstructed events. RunChain.C takes raw data as input, runs the muon HLT chain and gives the output in either binary format or root format. In this study all the data used to test the muon HLT monitoring system were generated using RunChain.C.

3.3 Implementation of utility that prints the muon HLT output to the screen

The utility, dHLTdumpraw.cxx, that prints the DDL raw data as well as the reconstructed event data to standard output (screen) was developed for monitoring purpose. This utility uses the binary output files generated by the RunChain.C macro. These binary files contain the muon HLT data blocks which, in turn, contain reconstructed hits, tracks, trigger records, clusters and the trigger decisions.

The C++ function given below is a snippet of code taken from dHLTdumpraw.cxx while the entire code is in Appendix E. The type of data block to be printed to screen can be specified as an option on command line else, if not specified the dump raw utility checks if the input block is a DDL raw data stream (lines 13 to 27) and if it is not then dump raw utility assumes that it is one of the muon HLT internal data blocks and its header sub-block is printed on the screen by line 41. Depending on the type of the data block (specified by the data block header), the rest of the data block body is printed to the screen by means of lines 76 to 127.

```
001 int ParseBuffer(
002                 const char* buffer, unsigned long bufferSize,
003                 bool continueParse, int& type
004         )
005 {
006 assert( buffer != NULL );
007 int result = EXIT_SUCCESS;
008 int subResult = EXIT_FAILURE;
009
010 // If the -type|-t option was not used in the command line then we need to
011 // figure out what type of data block this is from the data itself.
012 bool ddlStream = false;
013 if (type == kUnknownDataBlock)
014 {
015    // First check if this is a raw DDL stream, if not then assume it is
016    // some kind of internal dHLT raw data block.
017   int streamType = CheckIfDDLStream(buffer, bufferSize);
018   if (streamType == kTrackerDDLRawData or streamType == kTriggerDDLRawData)
019   {
020        type = streamType;
021        ddlStream = true;
022   }
```

```
023 }
024 else if (type == kTrackerDDLRawData or type == kTriggerDDLRawData)
025 {
026      ddlStream = true;
027 }
028
029 if (not ddlStream)
030 {
031     if (bufferSize < sizeof(AliHLTMUONDataBlockHeader))
032     {
033         cerr << "ERROR: The size of the file is too small to contain a"
034                 " valid data block." << endl;
035                 result = PARSE_ERROR;
036          if (not continueParse) return result;
037      }
038     const AliHLTMUONDataBlockHeader* header =
039         reinterpret_cast<const AliHLTMUONDataBlockHeader*>(buffer);
040
041     subResult = DumpCommonHeader(buffer, bufferSize, header, continueParse);
042     if (subResult != EXIT_SUCCESS) return subResult;
043
044
045     // Check if the block type in the header corresponds to the type given
046     // by the '-type' command line parameter. If they do not then print an
047     // error or big fat warning message and force interpretation of the data
048     // block with the type given by '-type'.
049     AliHLTMUONDataBlockType headerType = AliHLTMUONDataBlockType(header->fType);
050
051     if (type == kUnknownDataBlock)
052     {
053       // -type not used in the command line so just use what is given
054       // by the data block header.
055       type = headerType;
056     }
057     else if (type != headerType)
058     {
059          cerr << "WARNING: The data block header indicates a type"
060          " different from what was specified on the command line."
061          " The data could be corrupt."
062           << endl;
063           cerr << "WARNING: The type value in the file is "
064           << showbase << hex << header->fType
065           << " (" << headerType << "), but on the command line it is "
```

```
066           << showbase << hex << int(type) << dec
067           << " (" << type << ")."
068           << endl;
069           cerr << "WARNING: Will force the interpretation of the data block"
070           " with a type of " << type << "." << endl;
071     }
072 }
073
074 // Now we know what type the data block is supposed to be, so we can
075 // dump it to screen with the appropriate dump routine.
076 switch (type)
077 {
078 case kTrackerDDLRawData:
079      subResult = DumpTrackerDDLRawStream(buffer, bufferSize, continueParse);
080      if (subResult != EXIT_SUCCESS) result = subResult;
081      break;
082 case kTriggerDDLRawData:
083      subResult = DumpTriggerDDLRawStream(buffer, bufferSize, continueParse);
084      if (subResult != EXIT_SUCCESS) result = subResult;
085      break;
086 case kTriggerRecordsDataBlock:
087      subResult = DumpTriggerRecordsBlock(buffer, bufferSize, continueParse);
088      if (subResult != EXIT_SUCCESS) result = subResult;
089      break;
090 case kTrigRecsDebugDataBlock:
091      subResult = DumpTrigRecsDebugBlock(buffer, bufferSize, continueParse);
092      if (subResult != EXIT_SUCCESS) result = subResult;
093      break;
094 case kRecHitsDataBlock:
095      subResult = DumpRecHitsBlock(buffer, bufferSize, continueParse);
096      if (subResult != EXIT_SUCCESS) result = subResult;
097      break;
098 case kClustersDataBlock:
099      subResult = DumpClustersBlock(buffer, bufferSize, continueParse);
100      if (subResult != EXIT_SUCCESS) result = subResult;
101      break;
102 case kChannelsDataBlock:
103      return DumpChannelsBlock(buffer, bufferSize, continueParse);
104      if (subResult != EXIT_SUCCESS) result = subResult;
105      break;
106 case kMansoTracksDataBlock:
107      subResult = DumpMansoTracksBlock(buffer, bufferSize, continueParse);
108      if (subResult != EXIT_SUCCESS) result = subResult;
```

```
109      break;
110 case kMansoCandidatesDataBlock:
111      subResult = DumpMansoCandidatesBlock(buffer, bufferSize, continueParse);
112      if (subResult != EXIT_SUCCESS) result = subResult;
113      break;
114 case kSinglesDecisionDataBlock:
115      subResult = DumpSinglesDecisionBlock(buffer, bufferSize, continueParse);
116      if (subResult != EXIT_SUCCESS) result = subResult;
117      break;
118 case kPairsDecisionDataBlock:
119      subResult = DumpPairsDecisionBlock(buffer, bufferSize, continueParse);
120      if (subResult != EXIT_SUCCESS) result = subResult;
121      break;
122 default :
123         cout << "ERROR: Unknown data block type. Found a type number of "
124              << showbase << hex << int(type) << dec
125              << " (" << int(type) << ")." << endl;
126         result = PARSE_ERROR;
127  }
128 return result;
129 }
```

If the data is corrupted a command line option called *continueParse* can be given to force the dump raw utility to print the corrupt data for the user to view. This utility can also be used to check the integrity of the raw data.

3.4 Extending ALIEVE

During the running of the muon HLT a user may want to graphically view the event data that is being reconstructed using ALIEVE. However ALIEVE could only display offline data and there were no means to display online muon HLT reconstructed data. One of the aims of this project is to extend ALIEVE so that it can display the muon HLT data.

C++ class named *AliEveMUONTrack* in ALIEVE was modified to handle an object of type *AliHLTMUONMansoTrack*. This was done as follows:

```
001 class AliHLTMUONMansoTrack;
002
003 class AliEveMUONTrack: public TEveTrack
004 {
```

```
005  public:
006   void  MakeMansoTrack(AliHLTMUONMansoTrack *mtrack);
007   Bool_t IsMansoTrack()const { return fIsMansoTrack;}
008   void PrintMansoTrackInfo();
009  private:
010   AliHLTMUONMansoTrack *fMansoTrack; // Pointer to the Manso Track
011   Bool_t fIsMansoTrack;       // track from MansoBlockStruct
};
```

Two new methods *MakeMansoTrack* and *IsMansoTrack* were added to the existing class *AliEveMUONTrack*. Their respective functions are to assign the Manso track parameters (coordinates, momentum, p_T, etc) to *AliEveMUONTrack* and to return true if *AliEveMUONTrack* is made up of Manso track parameters. In addition another method, *PrintMansoTrackInfo*, to print all parameters of Manso track that were assigned to *AliEveMUONTrack* was introduced to the class *AliEveMUONTrack*. Function definations of *PrintMansoTrackInfo* and *MakeMansoTrack* are given in Appendix C.1.

Part of the modifications to AliEveMuonTrack included adding a pointer *fMansoTrack* and a flag *fIsMansoTrack* as private[6] members. *fMansoTrack* is a pointer to Manso track object while *fIsMansoTrack* indicates whether *AliEveMUONTrack* was assigned *AliHLTMUONMansoTrack* parameters.

Furthermore, in order to display the muon HLT reconstructed hits (*AliHLTMUONRecHit*) in ALIEVE, a C++, *AliEveMUONData*, was modified as follow:

```
class AliHLTMUONRecHit;

class AliEveMUONData : public TObject, public TEveRefCnt
{
 public:
  void LoadHLTHit(AliHLTMUONRecHit* recHit);
};
```

A new function, *LoadHLTHit*, was added to the public[7] section of the class *AliEveMUONData*. *LoadHLTHit* adds reconstructed muon HLT hit coordinates (x,y,z) to ALIEVE so that they can be displayed. Funtion defination of *LoadHLTHit* is shown in appendix C.2.

[6]Private members can only be accessed by member functions of the class

[7]Public members can be accessed by any function defined in the program

ALIEVE gets its muon HLT input data using interthread communication mechanism from the message queue discussed in detail in section 3.5 below.

3.5 Message queue

The message queue was implemented for the monitoring system to allow asynchronous data source reading using threads. It was implemented with the class *AliHLTMessageQueue* using the C++ language. The message queue class provides mechanism for a thread reading data from the data channels to communicate with the thread publishing data in ALIEVE in a controlled and correct manner. The *AliHLTMessageQueue* class is shown below.

```
template<typename MessageType, Int_t kSignalNumber = SIGUSR1>
class AliHLTMessageQueue
{
  public:
  AliHLTMessageQueue(long int bufSize = 1024);
  ~AliHLTMessageQueue();
  Bool_t Empty() const;
  Bool_t Full() const;
  #ifdef DEBUG
  UInt_t Size() const
  {
    Int_t size;
    if(fMutex.Lock() == 0)
    {
       size = fSize;
       fMutex.UnLock();
    }
    return size;  }
  #endif // DEBUG
  UInt_t Push(const MessageType* message, UInt_t count, Bool_t
wakeupReader = kTRUE, Float_t timeout = -1, Bool_t writePartial =
kFALSE);
  UInt_t Pop(MessageType* message, UInt_t maxCount, Bool_t wait =
kFALSE, Float_t timeout = -1);
  void SetReaderThread(Long_t threadId = -1);
  Long_t GetReaderThread() const;
  private:
  Bool_t IsEmpty() const;
```

```
  Bool_t IsFull() const;
  UInt_t fSize;
  MessageType* fFirst;
  MessageType* fLast;
  UInt_t fBufferSize;
  MessageType* fRingBuffer;
  Long_t fReaderThread;
  mutable TMutex fMutex;
};
```

Apart from the constructors, *AliHLTMessageQueue* has nine member functions (also known as class methods) of which 2 are private and 7 are public. It also has 7 data member which are all private. All the data members and the function members of *AliHLTMessageQueue* are explained in details in Appendix E. The sections below only focus on the *Push* and *Pop* function members of *AliHLTMessageQueue*.

3.5.1 The *Push* member function

It is used to insert new messages at the end of the queue starting with the *message[0]* and ending with the *message[count-1]*. If the flag *wakeupReader* is set, *Push* will send a signal to the reader thread after it has completed writing messages into the queue. If the queue becomes full before all messages have been added and at the same time the parameter *timeout* is set to a negative number, then *Push* will waited until a space has been created by the reader thread.

However if *timeout* has been set to a non negative number, *Push* will wait *timeout* number of seconds and if this time elapse before the space is created *Push* will then quit from writing messages even if no message were added to the queue. Furthermore if *writePartial* is set to true and the queue is full *Push* will stop writing messages if at least one new message is written to the queue. *Push* returns the number of messages added to the queue. The function parameters of the method *Push* can be summaried as follow:

message This is a pointer to the buffer which holds the messages that are to be added to the queue

count This is the number of messages to be added to the queue

wakeupReader If this flag is set to *kTRUE* the reader thread will be worken up with the system signal

timeout Non negative value states the number of seconds *Push* should wait for the space to free up. Negative values mean *Push* should not time out while waiting for a free space to write messages.

writePartial If it is set to *kTRUE Push* can return, if the queue is full, after adding at least one new message.

3.5.2 The *Pop* member function

This method pops or reads off as many messages as possible from the front of the queue. As it is popping the messages, it fills them into the array *message*. If the queue is empty and the flag *wait* is set to *kTRUE* and *timeout* is set to a negative number, *Pop* will block until at least one new message is added to the queue. If however *timeout* is set to a positive number and *wait* is still set to *kTRUE*, *Pop* will wait for *timeout* number of seconds for new messages to be added to the queue. This method returns the number of messages popped from the front of the queue. *Pop* is thread safe method. The function parameters of *Pop* can be summarized as follow:

message This is a pointer to the buffer that stores message being popped from the queue

maxCount This is maximum number of messages that can be stored in the array *message*

wait If this flag is set to *kTRUE Pop* will wait until at least one new message is added to the queue

timeout This specifies the number of seconds *Pop* will wait for new messages. If it is negative, *Pop* will not time out.

Chapter 4

Results and Discussion

Some parts of the monitoring system, e.g. the message queue, the dHLT-dumpraw utility and the GUI, were implemented and tested on the clusters at the University of Cape Town (UCT) and on the HLT cluster at CERN. The sections below discuss the results obtained from the tests.

4.1 Message queue

4.1.1 Message queue: testing methodology

The HLT message queue, implemented by the class AliHLTMessageQueue, was tested using different test benches both on single CPU computers (Bambino03 on the sandiego cluster and Bambino04 on CARMEN at UCT) and on multiple CPU (eight CPUs per computer) computers (Fepdimutrg[1] and Dev1[2] nodes on the HLT cluster at CERN). The longest tests took four days while shortest tests took about ten seconds. The primary aim of the tests was to make sure that the message queue can run without crashing or causing race conditions or deadlocks. To prove that the message queue was indeed running, the number of messages that were being popped by the Pop function were printed to the screen (for long tests, results were written to files) as the message queue was running. A snapshot of the segment of the typical output file is shown in Figure 4.1. This figure shows the results for the first 30 seconds in running the message queue.

The message queue program was tested on different platforms, i.e. on single and multiple CPU computers, to make sure that it can run without

[1]Is one of the nodes dedicated for muon HLT tests only

[2]Dev1, like Dev0, is used by all HLT software developers for testing their programs

hindrance on all platforms. This was critical because the message queue was developed using threads and if it was not properly designed it could, for example, run in one platform but not on the others. It was also important to test the message queue on the computers that are used by multiple users (hence the one running multiple processes simultaneously) to observe whether it can co-run with other processes without starving them off resources i.e cpu time and shared memory.

```
Push and Pop parameters:
Push(const MessageType* message = integer array, UInt_t count = 10, Bool_t wakeupReader = kTRUE, Float_t timeout = -1,  Bool_t writePartial = kFALSE)
Pop(MessageType* message = integer array, UInt_t maxCount = 10, Bool_t wait = kTRUE, Float_t timeout = 2)

Poped 921870 messages in 1.00176 seconds. Popping Rate = 920248 Hz. Queue size = 10
Poped 850430 messages in 1.00183 seconds. Popping Rate = 848881 Hz. Queue size = 0
Poped 931420 messages in 1.00182 seconds. Popping Rate = 929723 Hz. Queue size = 10
Poped 926760 messages in 1.00183 seconds. Popping Rate = 925072 Hz. Queue size = 10
Poped 930570 messages in 1.00183 seconds. Popping Rate = 928874 Hz. Queue size = 70
Poped 930420 messages in 1.00182 seconds. Popping Rate = 928725 Hz. Queue size = 100
Poped 931370 messages in 1.00183 seconds. Popping Rate = 929671 Hz. Queue size = 130
Poped 920910 messages in 1.00182 seconds. Popping Rate = 919233 Hz. Queue size = 150
Poped 926740 messages in 1.00182 seconds. Popping Rate = 925052 Hz. Queue size = 160
Poped 932590 messages in 1.00183 seconds. Popping Rate = 930890 Hz. Queue size = 180
Poped 895840 messages in 1.00183 seconds. Popping Rate = 894208 Hz. Queue size = 180
Poped 931980 messages in 1.00183 seconds. Popping Rate = 930281 Hz. Queue size = 200
Poped 860170 messages in 1.00183 seconds. Popping Rate = 858602 Hz. Queue size = 200
Poped 931910 messages in 1.00183 seconds. Popping Rate = 930211 Hz. Queue size = 210
Poped 891060 messages in 1.00182 seconds. Popping Rate = 889438 Hz. Queue size = 240
Poped 895390 messages in 1.00182 seconds. Popping Rate = 893759 Hz. Queue size = 250
Poped 930580 messages in 1.00183 seconds. Popping Rate = 928884 Hz. Queue size = 260
Poped 932330 messages in 1.00182 seconds. Popping Rate = 930633 Hz. Queue size = 270
Poped 932380 messages in 1.00183 seconds. Popping Rate = 930681 Hz. Queue size = 300
Poped 933070 messages in 1.00183 seconds. Popping Rate = 931370 Hz. Queue size = 300
Poped 892570 messages in 1.00183 seconds. Popping Rate = 890942 Hz. Queue size = 320
Poped 920160 messages in 1.00182 seconds. Popping Rate = 918484 Hz. Queue size = 330
Poped 885500 messages in 1.00183 seconds. Popping Rate = 883887 Hz. Queue size = 340
Poped 916680 messages in 1.00183 seconds. Popping Rate = 915009 Hz. Queue size = 340
Poped 932410 messages in 1.00183 seconds. Popping Rate = 930711 Hz. Queue size = 350
Poped 933970 messages in 1.00182 seconds. Popping Rate = 932269 Hz. Queue size = 310
Poped 933600 messages in 1.00183 seconds. Popping Rate = 931899 Hz. Queue size = 350
Poped 933010 messages in 1.00183 seconds. Popping Rate = 931309 Hz. Queue size = 360
Poped 914670 messages in 1.00182 seconds. Popping Rate = 913004 Hz. Queue size = 370
Poped 932110 messages in 1.00183 seconds. Popping Rate = 930411 Hz. Queue size = 380
```


Figure 4.1: Output of test bench for testing message queue

4.1.2 Results and Discussions of Long tests on Message Queue

The Message queue was tested for four consecutive days on multi CPU computers (Fepdimutrg and Dev1) and single CPU computers (Bambino03 on sandiego cluster and Bambino04 on CARMEN) to check whether it can run for extended periods (4 days) without crashing or causing memory leaks. These tests completed without encountering any problems and the results were writen to files. The test bench used for all long run tests is in appendix B.1.

With the results obtained, the average popping rate was plotted against time. The popping rate is the number of messages popped per second while the avarage popping rate is the sum of the popping rates divided by their total number. The results from single CPU tests are plotted in Figures 4.2 and 4.3 while those for mutiple CPU tests are plotted in Figures 4.4 and 4.5.

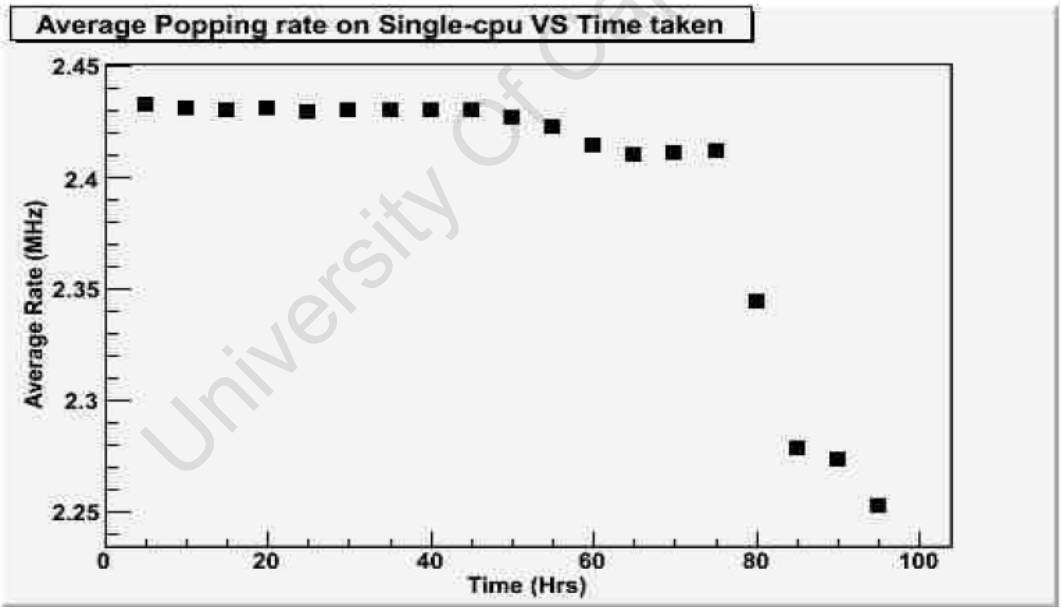

Figure 4.2: Four days test on Bamino03 in sandiego

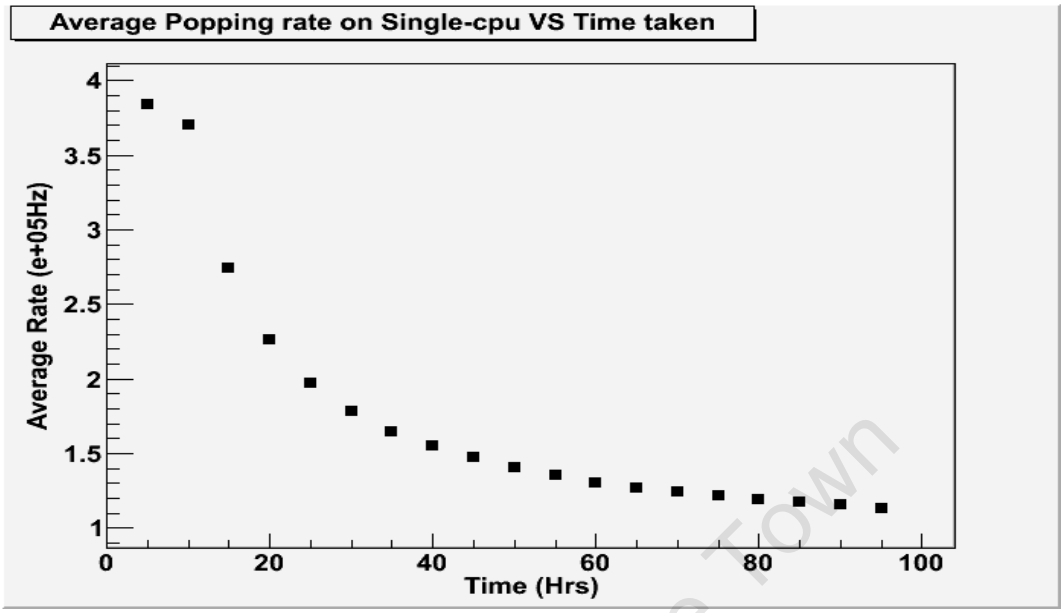

Figure 4.3: Four days test on on Bambino04 on CARMEN

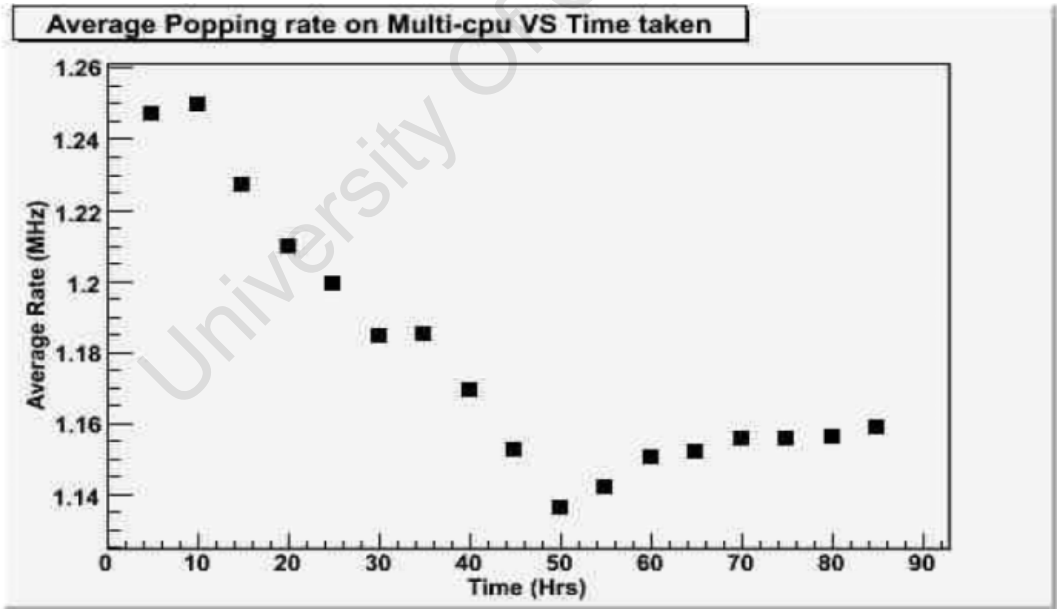

Figure 4.4: Four days test on Fepdimutrg

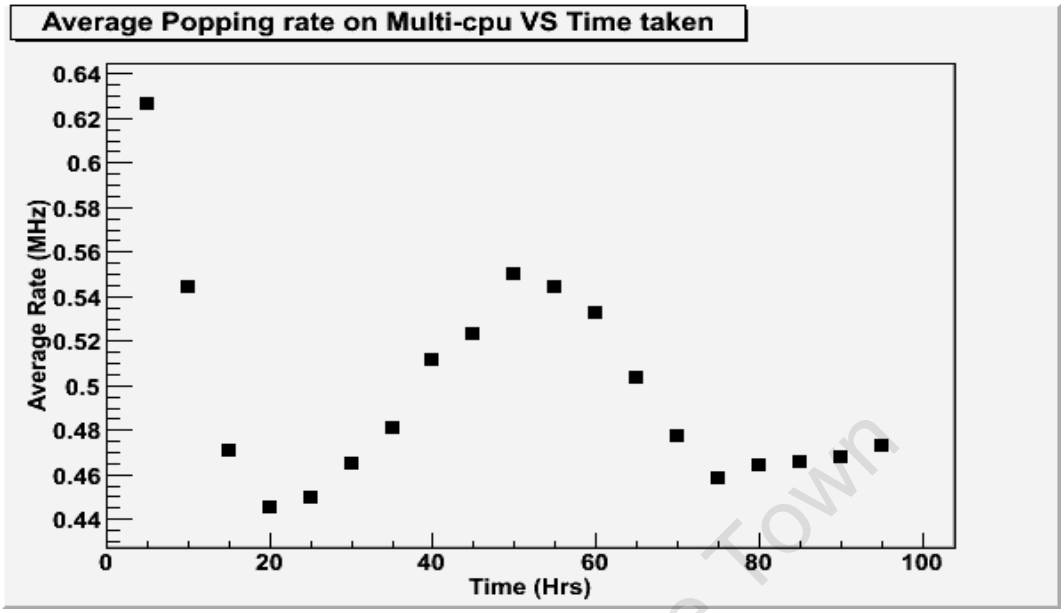

Figure 4.5: Results obtained from four days test on dev1

From Figures 4.4 and 4.5 it can be seen that average popping rates on Dev1 were lower than on Fepdimutrg. This could be due to the fact that many processes run on Dev1 than on Fepdimutrg because Dev1 is used by all HLT users while Fepdimutrg is used only by muon HLT group. So there could have been many processes sharing CPU time with the message queue on Dev1 than on Fepdimutrg hence lower popping rates on Dev1 than on Fepdimutrg.

A continuous decline in the average popping rate (approximately 0.12 MHz) shown in Figure 4.4 could also be attributed to quite a number of processes being scheduled to run at the same time with the message queue. However after the 50^{th} hour the average popping rates began to increase and this shows that more and more of the processes which were sharing CPU time with the message queue finished (or stopped) running and the message queue shared CPU time with few processes and hence it could pop more messages per time.

Figure 4.5 illustrates that the message queue was able to run for four consecutive days without crashing. This is proven by the fact that there has been a continous popping out of messages for that period of four days. Figure 4.5 shows that around the 20^{th} hour the message queue shared cpu time with a lot of processes hence it popped out relatively little number of messages as it did on other hours. The message queue popped most of the

messages around the 50^{th} hour, the reason for this could be that around this hour very few processes shared CPU time with the message queue. Figure 4.5 also shows a drop around the 70^{th} hour in the number of messages being popped. Just like on the 20^{th} hour the message queue shared cpu time with very many processes hence it could only pop out the messages at a very reduced rate. It should however be very clear that the purpose of the tests was to prove that the message queue can run, for any specified amount of time, without crashing. As to how many processes were sharing CPU time with it and the dynamics of the scheduling algorithm are not of interest to this report.

Figures 4.2 and 4.3 show the results of running on single CPU computers Bambino03 on Sandiego cluster and Bambino04 on CARMEN cluster, respectively. It can be seen that the average popping rates on Sandiego are higher than on CARMEN and this could be because Sandiego is a much newer cluster than CARMEN hence Bambino03's CPU works faster than Bambino04's CPU. Similar to Figures 4.4 and 4.5, a decline in average rate with time is observed in Figure 4.2 at about 80 hours of running and this could be due to more processes being scheduled to share CPU time with the message queue. A steady decline in average popping rates is also observed in Figure 4.3 and the only obvious reason for this decline is more and more processes being scheduled to share CPU time with the message queue.

4.1.3 Results and Discussions of Short tests on Message Queue

About 128 short tests were performed on Fepdimutrg, with each test taking 10 seconds. The aim was to test all the five parameters of push function and all four parameters of Pop function and make sure that no combination of message queue parameters can crash the message queue. All the tests ran sucessfully to completion. The values of the *messageType* parameter (in both *Push* and *Pop*) was never changed and it was kept as an integer array for all tests, only the length of the array changed. The length of this array is specified by the parameter *count* incase of Push method and by *maxCount* incase of Pop method. The test bench file used for these short tests is given in appendix B.1.

The first 64 tests of the message queue were performed by setting the value for *count* parameter as 10 then changed to 1 for the last 64 tests as

shown in Table D.1. This was to ensure that the message queue can push both multiple and single messages. Table D.1 further illustrates that for the first 32 tests the message queue was tested with $maxCount = 10$ after which was changed 1 for the next 32 tests. The value was alternated between 10 and 1 for the 32 tests interval until 128 were all done. This was also to ensure that the Pop method can pop both single and multiple messages. Table D.2 shows the values which were assigned to the parameters of Push method (*wakeupReader*, *timeout*, and *writePartial*) and to the parameters of Pop method (*wait*, and *timeout*).

The average popping rates for all the 128 tests are given in Tables D.3 and D.4 while their spread or distribution is illustrated in Figure 4.6. Figure 4.6 shows that from 128 tests about 92 had an average rate ≤ 0.15 MHz and this is more than half the total number of all tests. About 122 tests had an average popping rate less than 1 MHz while only about 6 tests had average popping rate greater than that 1 MHz. Again, this could be attributed to sharing CPU time with many other processes.

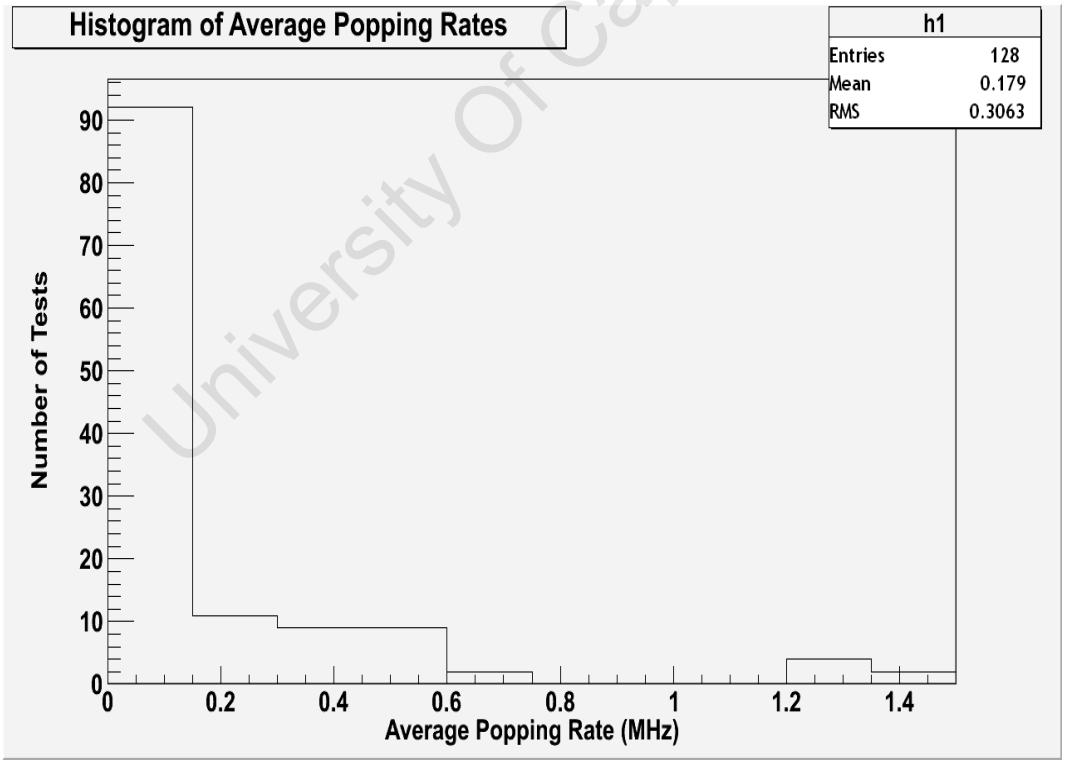

Figure 4.6: Histogram of 128 short tests (10 seconds tests)

4.1.4 Testing for data corruption and memory leaks

The message queue was also tested for data corruption using the testbench shown in Appendix B.2. The test bench pushed the numbers from 0 to 1999 into the queue using the segment of the code shown below.

```
#define COUNT 10
#define MESSAGES 2000
int container[MESSAGES];
unsigned int pushCounter = 0;
void* WriterRoutine(void* param)
{
  int counter = 0;
  cout << "Started writer thread..." << endl;
  while (counter < MESSAGES)
  {
    int message[COUNT];
    for(int i = 0; i < COUNT; i++)
    {
      message[i] = counter;
      counter++;
    }
    pushCounter += queue.Push(message, COUNT,true,-1,false);
  }
  writerFinished = true;

  return param;
}
```

Using the segment of code below, the testbench then popped back the numbers it early pushed into the queue and put them into an array. The numbers contained in the array were then printed to the file. All 2000 messages written into the queue were successfully popped out in the order they were pushed. The first integer pushed into the message queue hence the first integer to be popped out was 0 and the last integer to be pushed in and popped out was 1999. This proves that the message queue does not corrupt data at all. Since the message queue is a c++ template class[3] there was no need to test it with different data types (e.g string, user defined type, etc) because if it is proven to work with one type, which is integer in this case,

[3]A template class is a class which does not have fixed type for its data members in its declaration and the user of that function will assign his/her wanted type to the data members when he/she instantiate the object of that template class

then it can work and handle all other types.

```
int message[COUNT];
unsigned int n;
n = queue.Pop(message, COUNT);
if(n > 0)
{
  for(unsigned int i = 0; i < n; i++)
  {
    container[counter] = message[i];
    counter++;
  }
  popCounter += n;
}

for(int i = 0; i < MESSAGES; i++)
{
  cout << container[i] << " ";
  if(i != 0 and i % 20 == 0)
  {
    cout << endl;
    sleep(1);
  }
}
```

The message queue program does not introduce any memory leaks because of the way it is designed. Its data member *fRingBuffer* is the only variable which is allocated memory dynamically i.e with the use of the keyword *new* inside the class constructor. Hence the memory is allocated dynamically only once when the message queue object is instantiated and this memory is reclaimed when the program finishes executing by deleting *fRingBuffer* with *delete* keyword in the class destructor.

4.2 dHLTdumpraw

The dHLTdumpraw utility that prints, to the screen, the DDL raw data and muon HLT reconstructed data was successfully developed and its code is in Appendix E and in [38]. The input data to dHLTdumpraw is a binary file produced by RunChain.C macro developed in [46].

The binary file contains blocks of muon HLT reconstructed data and each

block is made up of a header and a body. Using a block of reconstructed information as an example, the header contains information about the type of the data block (type is kRecHitsDataBlock for reconstructed hits block), its record width (The number of bytes each hits uses) and the number of the contained reconstructed entities (hits) while the body of the block contains information about reconstructed entities (e.g coordinates of the reconstructed hits).

Seven binary files containing information about reconstructed hits were simultaneously used as inputs to the dHLTdumpraw utility. These files were then processed by retrieving relavant information and printing it to the screen. Figure 4.7 shows the snap shot of the results on the screen. For each block of reconstructed hits, dHLTdumpraw printed its header information, the coordinates of the reconstructed hits and the location in the muon spectrometer where the reconstructed hits were found (i.e chamber number and the detector element ID).

```
########## Start of dump for file: output_0x00000000_0x02_MUON:RECHITS_0x00080000.dat #########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
10        1005        70.6049      192.83       -1436.77
##########   End of dump for file: output_0x00000000_0x02_MUON:RECHITS_0x00080000.dat ##########
########## Start of dump for file: output_0x00000000_0x03_MUON:RECHITS_0x00040000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
10        1015        -59.4643     -93.341      -1448.31
##########   End of dump for file: output_0x00000000_0x03_MUON:RECHITS_0x00040000.dat ##########
########## Start of dump for file: output_0x00000000_0x04_MUON:RECHITS_0x00020000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
9         905         68.75        189.164      -1405.72
##########   End of dump for file: output_0x00000000_0x04_MUON:RECHITS_0x00020000.dat ##########
########## Start of dump for file: output_0x00000000_0x05_MUON:RECHITS_0x00010000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
9         915         -58.3649     -91.8412     -1417.33
##########   End of dump for file: output_0x00000000_0x05_MUON:RECHITS_0x00010000.dat ##########
########## Start of dump for file: output_0x00000000_0x06_MUON:RECHITS_0x00008000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
8         805         64.1344      177.532      -1306.46
##########   End of dump for file: output_0x00000000_0x06_MUON:RECHITS_0x00008000.dat ##########
########## Start of dump for file: output_0x00000000_0x07_MUON:RECHITS_0x00004000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
8         815         -54.6325     -86.6632     -1318.3
##########   End of dump for file: output_0x00000000_0x07_MUON:RECHITS_0x00004000.dat ##########
########## Start of dump for file: output_0x00000000_0x08_MUON:RECHITS_0x00002000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
7         705         62.3214      173.914      -1275.41
##########   End of dump for file: output_0x00000000_0x08_MUON:RECHITS_0x00002000.dat ##########
########## Start of dump for file: output_0x00000000_0x09_MUON:RECHITS_0x00001000.dat ##########
       Block type: kRecHitsDataBlock
     Record width: 16
Number of entries: 1
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
7         715         -53.3652     -85.0645     -1287.32
##########   End of dump for file: output_0x00000000_0x09_MUON:RECHITS_0x00001000.dat #########
```


Figure 4.7: The output of the reconstructed hits blocks printed to the screen by means dHLTdumpraw.

Figure 4.8 shows the printed trigger records block. For this block of data, the dHLTdumpraw printed the information contained in the data block header, the ID of the trigger record, its sign, its momentum and the information about its reconstructed hits (i.e coordinates and location).

```
      Block type: kTriggerRecordsDataBlock
    Record width: 84
Number of entries: 1
============================== Trigger Record number 1 of 1 ==============================
Trigger Record ID: 149
Flags: 0x4000000f [Sign: kSignPlus, Hits set on chambers: 11, 12, 13, 14]
Momentum: (px = -1.97339, py = -3.44456, pz = -48.6469) GeV/c
Hits on chambers:
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
11        1110        -64.6875     -101.316     -1599.9
12        1210        -65.3732     -102.39      -1616.9
13        1310        -68.7214     -106.506     -1699.9
14        1410        -69.4071     -107.568     -1716.9
```

Figure 4.8: Output of dHLTdumpraw : dumped trigger records block file1

The contents of the data block containing the Manso tracks information were printed by dHLTdumpraw as shown in Figure 4.9. This figure shows the block header information and the information of each track contained in the block which is the track ID, its associated trigger record ID, its sign, its momentum, the coordinates of the track hits and the χ^2 obtained from the track fitting stage.


```
      Block type: kMansoTracksDataBlock
    Record width: 92
Number of entries: 2
============================== Manso track number 1 of 2 ==============================
Track ID: 149   Trigger Record ID: 149
Flags: 0x4000000a [Sign: kSignPlus, Hits set on chambers: 8, 10]
Momentum: (px = -1.91768, py = -3.27622, pz = -46.1183) GeV/c   Chi squared fit: 0
Track hits:
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
0         0           0            0            0
8         815         -54.6325     -86.6632     -1318.3
0         0           0            0            0
10        1015        -59.4643     -93.341      -1448.31
============================== Manso track number 2 of 2 ==============================
Track ID: 302   Trigger Record ID: 46
Flags: 0x8000000a [Sign: kSignMinus, Hits set on chambers: 8, 10]
Momentum: (px = 1.78201, py = 5.16102, pz = -35.8879) GeV/c     Chi squared fit: 0
Track hits:
Chamber | DetElemID | X (cm)     | Y (cm)     | Z (cm)
------------------------------------------------------------
0         0           0            0            0
8         805         64.1344      177.532      -1306.46
0         0           0            0            0
10        1005        70.6049      192.83       -1436.77
```

Figure 4.9: Output of dHLTdump : dumped Manso tracks block

4.3 ALIEVE

ALIEVE was successfully enhanced to display muon HLT resconstructed hits and Manso tracks. Data used to test these new modification was produced by running C++ macro RunChain.C. RunChain.C was set to write the output to a ROOT file named output.root. This file, i.e output.root, contains AliHLTMUONEvent[4] objects which in turn contain tracks and hits.

MUON_displaySimu.C that is used to load muon data to be displayed in ALIEVE was modified to retrieve reconstructed hits and tracks from AliHLTMUONEvent and display them in ALIEVE. To retrieve reconstruced hits from AliHLTMUONEvent the following piece of code was added to MUON_displaySimu function in the MUON_displaySimu.C macro and the results of running MUON_displaySimu.C with these changes are shown in Figure 4.10 where reconstructed hits are displayed in muon spectrometer chambers. It should be noted that the graphical user interface of ALIEVE and the muon spectrometer chambers shown in Figure 4.10 were not developed in this project but they have been developed by ALICE collaboration. The purpose of this project was to extend ALIEVE so that muon HLT hits can also be displayed and this has been achieved as shown in Figure 4.10.

```
gROOT->ProcessLine(".L AliEveMUONData.cxx++");

g_muon_data = new AliEveMUONData;
TFile* f = new TFile("output.root");
AliHLTMUONEvent* event;
TObject* object;
AliHLTMUONRecHit* recHit;
f->GetObject("AliHLTMUONEvent;1",event);
TObjArray objects = event->DataObjects();
Int_t nEntries = objects.GetEntries();
for (Int_t i = 0; i < nEntries; i++)
{
  object = objects.UncheckedAt(i);
  const char* name = object->GetName();
  if (strcmp(name, "AliHLTMUONRecHit") == 0)
  {
    recHit = dynamic_cast<AliHLTMUONRecHit*>(object);
    g_muon_data->LoadHLTHit(recHit);
  }
```

[4]AliHLTMUONEvent class's object is a container of all reconstructed muon event data, .i.e tracks, hits etc

}

MUON_displaySimu.C was also modified to retrieve Manso tracks from AliHLTMUONEvent and display them in ALIEVE. The following function was added to MUON_displaySimu.C and was called in MUON_displaySimu funtion where it displays tracks. The results of running MUON_displaySimu.C with this changes is shown in Figure 4.11 where Manso tracks are displayed in muon spectrometer. Figure 4.12 shows both hits and tracks in the muon spectrometer chambers. In this case MUON_displaySimu.C was run with all new implemented modifications.

```
void Manso_Tracks()
{
  gROOT->ProcessLine(".L AliEveMUONTrack.cxx++");
  AliEveMUONTrack* track;

  TFile* f = new TFile("output.root");
  AliHLTMUONEvent* event;
  TObject* object;
  AliHLTMUONMansoTrack* mtrack;
  TEveTrackList* lt = new TEveTrackList("Manso-Tracks");
  lt->SetMainColor(6);

  gEve->AddElement(lt);
  TEveRecTrack rt;
  f->GetObject("AliHLTMUONEvent;1",event);
  TObjArray objects = event->DataObjects();
  Int_t nEntries = objects.GetEntries();
  Int_t trackCount = 1;
  Int_t t = 0;
  for (Int_t i = 0; i < nEntries; i++)
  {
     object = objects.UncheckedAt(i);
     const char* name = object->GetName();
     if (strcmp(name, "AliHLTMUONMansoTrack") == 0)
     {
       mtrack = dynamic_cast<AliHLTMUONMansoTrack*>(object);
       rt.fLabel = t++;
       track = new AliEveMUONTrack(&rt, lt->GetPropagator());
       track->MakeMansoTrack(mtrack);
       gEve->AddElement(track, lt);
     }
  }
```

}

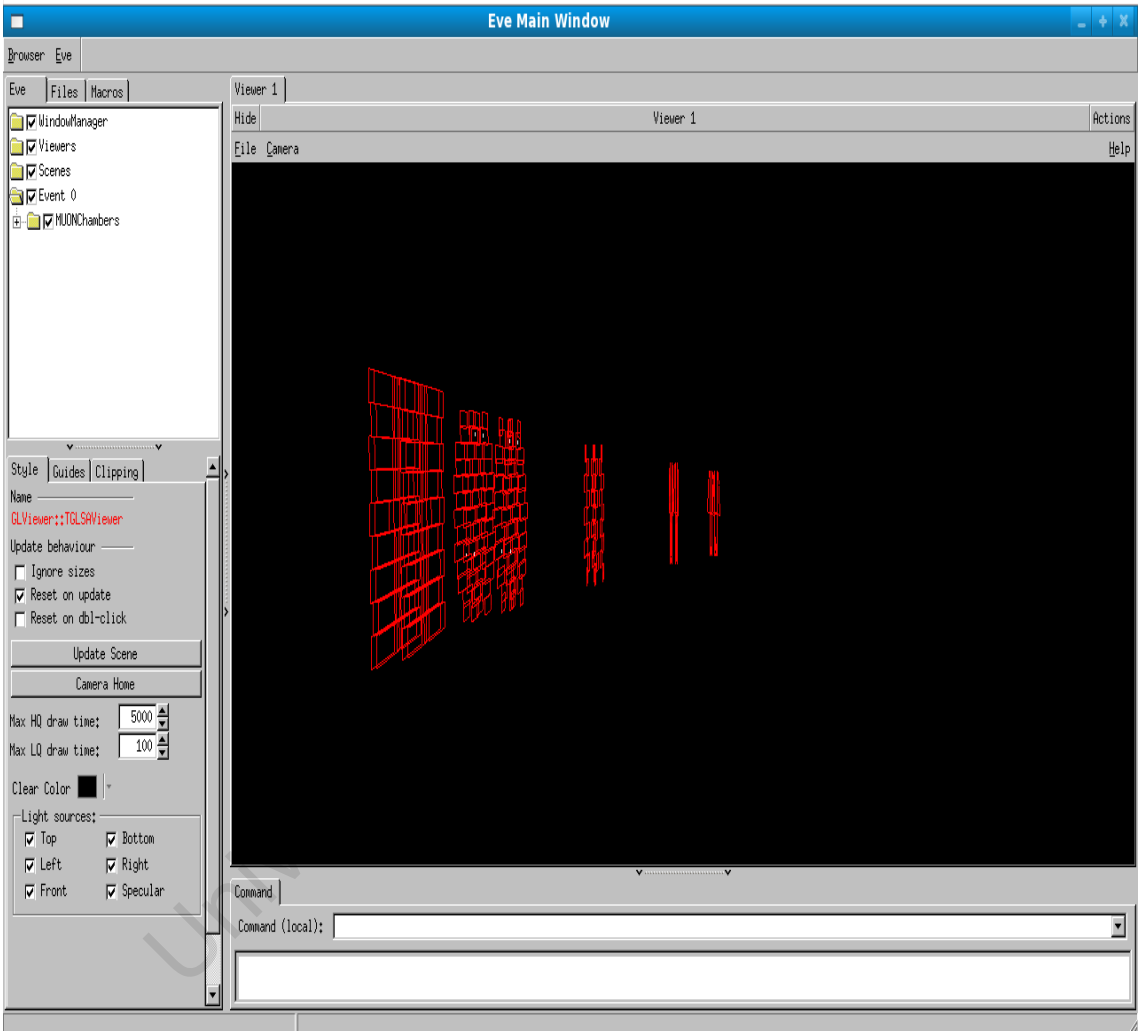

Figure 4.10: Reconstructed hits displayed in ALIEVE

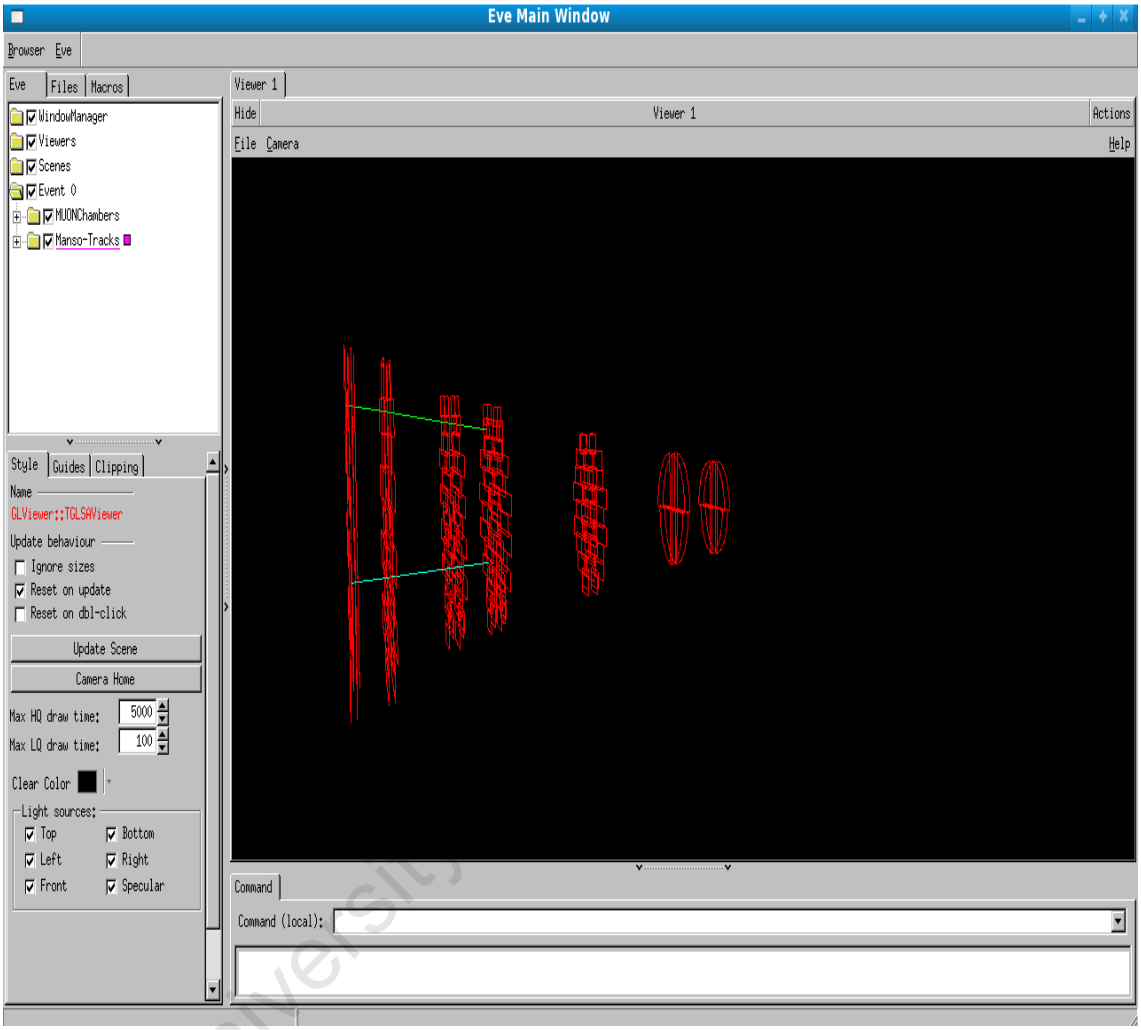

Figure 4.11: Manso Tracks displayed in ALIEVE

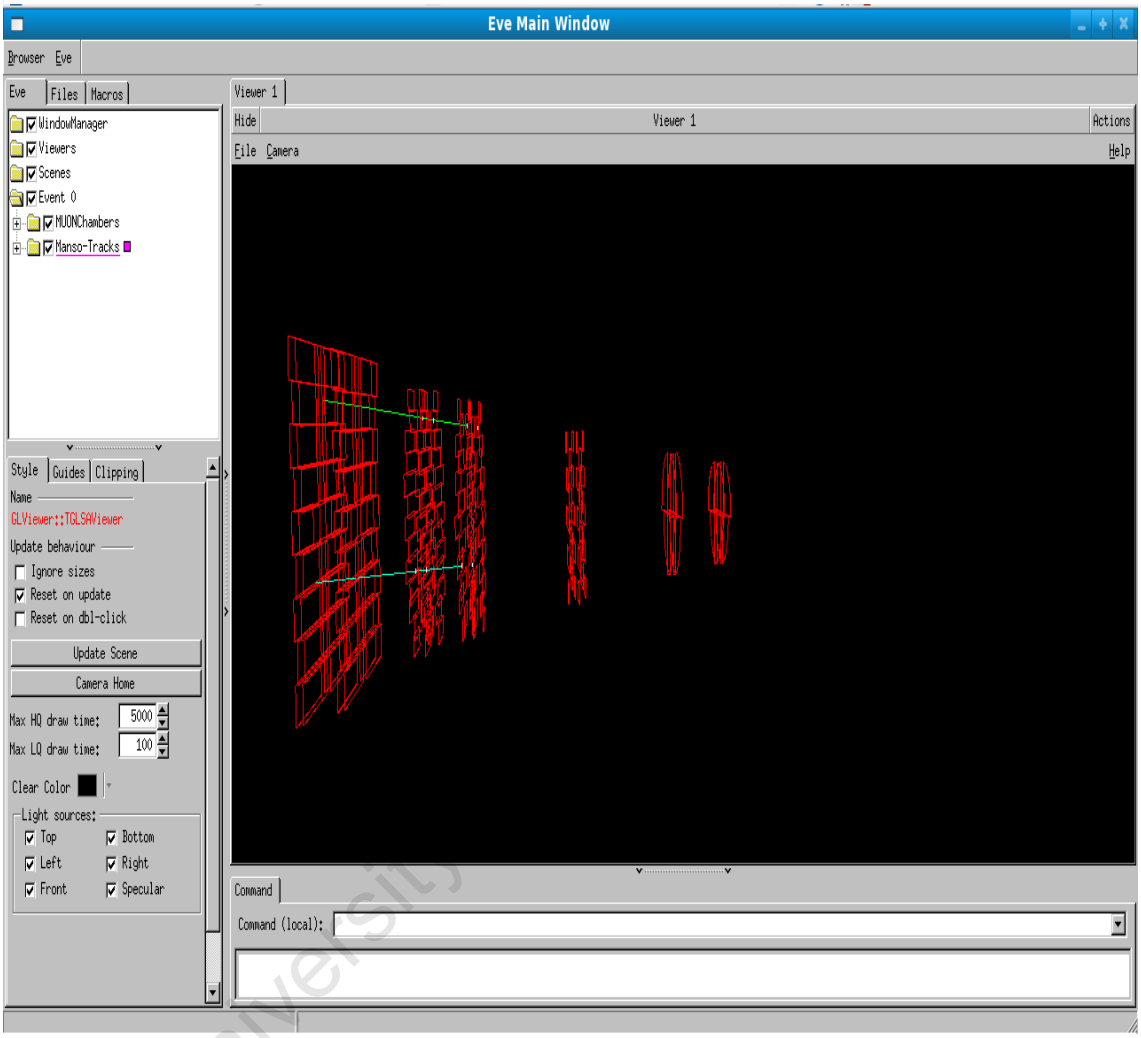

Figure 4.12: Reconstructed hits and Manso tracks in ALIEVE

Chapter 5

Conclusion and outlook

The aim of this study was to modify, extend and implement some components of the monitoring system of the ALICE muon HLT. The monitoring system consists of the channel manager, the data channels, the message queue, ALIEVE (graphic user interface), and event buffer. Of these, only the message queue and ALIEVE were either modified or extended as per requirement by the ALICE experiment.

The message queue was implemented with the C++ class *AliHLTMessageQueue*. Nine parameters of *Push* and *Pop* methods which make up the interface of *AliHLTMessageQueue* were tested successfully to completion without experiencing any crash. The tests were done on multi and single CPU platforms for long periods (4 days) and short periods (10 seconds). In both cases no problems were encountered and this indicates that the code works and the message queue was successfully implemented.

ALIEVE was modified to display muon HLT hits and tracks, and as demonstrated in the previous chapter this was also successfully done. The modified ALIEVE code has been sent to ALICE muon group at CERN to be uploaded into the ALICE software repository.

The other component of the muon HLT monitoring system that was developed is the dHLTdumpraw utility. Although it will not be running in real time with the HLT it will used during offline analysis to print, with more detail, the muon HLT reconstructed event data. The C++ code, dHLTdumpraw.cxx, that implements dHLTdumpraw utility has been uploaded and committed into the ALICE software repository [38].

Although all the modified and implemented components of the muon HLT

were tested separately and the performance was satisfactory, they still have to be integrated into one large monitoring system with other components and then tested together. Furthermore, the data that was displayed in ALIEVE was generated by the RunChain.C macro however during ALICE data taking, data that will be read from the muon HLT chain by using HOMER. So, in future, an effort should be made to simulate a running HLT chain, read data from it using HOMER and display this data in ALIEVE.

An emphasis should be made on the design and development of the component of the muon HLT monitoring system that informs the Task Manager (TM) about the status of the muon HLT. This component is specially useful to monitor errors by notifying the TM which, in turn, will notify the Detector Control System (DCS) through the HLT proxy [26].

Most of the fundamental components of the muon HLT monitoring system are successfully implemented as originally envisaged in this work.

Appendix A

Reading data with HOMER from the muon HLT chain


```
/*
 * @author Artur Szostak <artursz@iafrica.com>,
 *         Seforo Mohlalisi <seforomohlalisi@yahoo.co.uk>
 * @brief  Utility to read data from PubSub TCPDumpSubscriber using
HOMER
 */

#if !defined(__CINT__) || defined(__MAKECINT__)
#define USE_ROOT
#include "AliHLTHOMERReader.h"
#include "AliHLTMessage.h"
#include "Riostream.h"
#include "Rtypes.h"
#include "TH1D.h"
#include "TObject.h"
#include "TClass.h"
#include "TDirectory.h"
#include "TTimeStamp.h"
#include "TSystem.h"
#include "TCanvas.h"
#include <errno.h>
#else
#error You must compile this macro. Try the following command "cp
$ALICE_ROOT/HLT/MUON/macros/rootlogon.C . ; aliroot -l
connectionTest.C++"
#endif
```

```
const char* DataTypeToString(homer_uint64 type)
{
  union
  {
    homer_uint64 val;
    char bytes[8];
  };
  val = type;
  static char str[9];
  for (int i = 0; i < 8; i++)
  {
    str[i] = bytes[7-i];
  }
  str[8] = '\0'; // Null terminate the string.
  return str;
}


const char* OriginToString(homer_uint32 origin)
{
  union
  {
    homer_uint32 val;
    char bytes[4];
  };
  val = origin;
  static char str[5];
  for (int i = 0; i < 4; i++)
  {
    str[i] = bytes[3-i];
  }
  str[4] = '\0'; // Null terminate the string.
  return str;
}


void connectionTest(const char* hostname = "fepdimutrk1", UShort_t
port = 60000)
{
  AliHLTHOMERReader reader(hostname, port);
  int status = reader.GetConnectionStatus();
  if (status != 0)
```

```
  {
    cerr  << "ERROR: Could not connect to HOMER port running on "
<< hostname << ":" << port
      << " (error = " << status << ")" << endl;
    return;
  }
  status = reader.ReadNextEvent(5000000);  // 5 second timeout.
  if (status == ETIMEDOUT)
  {
    cout << "Timedout while trying to read from HOMER port." <<
endl;
    return;
  }
  else if (status != ETIMEDOUT && status != 0)
  {
    cerr  << "ERROR: Could not read from HOMER port running on " <<
hostname << ":" << port
      << " (error = " << status << ")" << endl;
    return;
  }
  else
  {
    cout  << "Received data blocks for event ID = 0x" << hex <<
reader.GetEventID() << dec
      << " and " << reader.GetBlockCnt() << " data blocks." <<
endl;
    cout << "block\ttype    \torigin\tspecification\tsize" << endl;
    for (unsigned long n = 0; n < reader.GetBlockCnt(); n++)
    {
      cout  << n
        << "\t" << DataTypeToString(reader.GetBlockDataType(n))
        << "\t" << OriginToString(reader.GetBlockDataOrigin(n))
        << "\t0x" << setw(8) << setfill('0') << hex <<
reader.GetBlockDataSpec(n) << setfill(' ') << setw(0) << dec
        << "\t" << reader.GetBlockDataLength(n)
        << endl;
    }
  }
}
```

Appendix B

Test Benches

B.1 Test bench used to test queue with debug option

```
#include <iostream>
using namespace std;

#include "TThread.h"

#include <pthread.h>

#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include <cassert>
#include <signal.h>

#include "AliHLTMessageQueue.h"
AliHLTMessageQueue<int> queue;

int EXIT_SUCCESS = 1;
int EXIT_FAILURE = 0;
bool terminateThread = false;
bool writerFinished = false;
bool readerFinished = false;

#define COUNT 10

unsigned int pushCounter = 0;
```

```
void* WriterRoutine(void* param)
{
  cout << "Started writer thread..." << endl;

  int counter = 0;
  while (not terminateThread)
  {
    int m[COUNT];
    for (int i = 0; i < COUNT; i++)
      m[i] = counter++;
    pushCounter += queue.Push(m, COUNT,true,2,true);
  }
  writerFinished = true;
  return param;
}


unsigned int popCounter = 0;


void SigHandler(int)
{
  // Nothing to do. This will interupt the sleeping thread.
}


void* ReaderRoutine(void* param)
{
  struct sigaction act;
  act.sa_handler = &SigHandler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  int result = sigaction(SIGUSR1, &act, NULL);
  if (result != 0)
  {
    cerr << "Could not set sigaction." << endl;
    return NULL;
  }

  cout << "Started reader thread..." << endl;

  popCounter = 0;
  while (true)
```

```
  {
    if(terminateThread and writerFinished and queue.Empty())
      break;
    if (not queue.Empty())
    {
      int m[COUNT];
      unsigned int n;
      while ((n = queue.Pop(m, COUNT)) > 0)
      {
        popCounter += n;
      }
    }
    else
    {
      sleep(1);
    }

  }
  readerFinished = true;

  return param;
}


int main()
{
  cout << "Push(message (integer array), count = 10, wakeupReader =
kTRUE, timeout = 2, writePartial = kTRUE); " << endl;
  cout << "Pop(message (integer array), maxCount = 10, wait =
kFALSE, timeout = -1); " << endl;
  pthread_t readerThread;
  pthread_t writerThread;
  int result = pthread_create(&readerThread, NULL, &ReaderRoutine,
NULL);
        queue.SetReaderThread(readerThread);
  if (result != 0)
  {
    cerr << "Could not create the reader thread." << endl;
    return EXIT_FAILURE;
  }
  sleep(1);
  result = pthread_create(&writerThread, NULL, &WriterRoutine,
NULL);
```

```
  if (result != 0)
  {
    cerr << "Could not create the writer thread." << endl;
    return EXIT_FAILURE;
  }
  bool run = true;
  double start = GetTime();
  while (run)
  {
    sleep(1);
    if(writerFinished and readerFinished)
      run = false;
    double current = GetTime();
    cout << "Pushed " << pushCounter << " and " << " Poped " <<
popCounter << " messages in "
         << current - start << " seconds. Queue size = " <<
queue.Size() << endl;
    if (current - start > 3600)
    {
      terminateThread = true;
    }

  }

  void* returnResult;
  result = pthread_join(writerThread, &returnResult);
  if (result != 0)
  {
    cerr << "Could not join the writer thread." << endl;
    return EXIT_FAILURE;
  }
  result = pthread_join(readerThread, &returnResult);
  if (result != 0)
  {
    cerr << "Could not join the reader thread." << endl;
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

B.2 Test bench used to test message queue for data corruption

```
#include <iostream>
using namespace std;

#include "TThread.h"
#include <pthread.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include <cassert>
#include <signal.h>

#include "AliHLTMessageQueue.h"
AliHLTMessageQueue<int> queue;

bool writerFinished = false;
bool readerFinished = false;

#define COUNT 10
#define EXIT_SUCCESS 1
#define EXIT_FAILURE 0
#define MESSAGES 2000
int container[MESSAGES];
unsigned int pushCounter = 0;
void* WriterRoutine(void* param)
{
  int counter = 0;
  cout << "Started writer thread..." << endl;
  while (counter < MESSAGES)
  {
    int message[COUNT];
    for(int i = 0; i < COUNT; i++)
    {
      message[i] = counter;
      counter++;
    }
    pushCounter += queue.Push(message, COUNT,true,-1,false);
  }
  writerFinished = true;

  return param;
```

```
}


unsigned int popCounter = 0;


void SigHandler(int)
{
  // Nothing to do. This will interupt the sleeping thread.
}


void* ReaderRoutine(void* param)
{
  struct sigaction act;
  act.sa_handler = &SigHandler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  int result = sigaction(SIGUSR1, &act, NULL);
  if (result != 0)
  {
    cerr << "Could not set sigaction." << endl;
    return NULL;
  }
  cout << "Started reader thread..." << endl;
  popCounter = 0;
  int counter = 0;
  while(readerFinished == false)
  {
    if(queue.Empty() == false)
    {
      int message[COUNT];
      unsigned int n;
      n = queue.Pop(message, COUNT);
      if(n > 0)
      {
        for(unsigned int i = 0; i < n; i++)
        {
          container[counter] = message[i];
          counter++;
        }
        popCounter += n;
      }
```

```
    }
    if(writerFinished and queue.Empty())
      readerFinished = true;

  }
  return param;
}


int main()
{
  pthread_t readerThread;
  pthread_t writerThread;
  int result = pthread_create(&readerThread, NULL, &ReaderRoutine,
NULL);
        queue.SetReaderThread(readerThread);
  if (result != 0)
  {
    cerr << "Could not create the reader thread." << endl;
    return EXIT_FAILURE;
  }
  sleep(1);
  result = pthread_create(&writerThread, NULL, &WriterRoutine,
NULL);
  if (result != 0)
  {
    cerr << "Could not create the writer thread." << endl;
    return EXIT_FAILURE;
  }
  void* returnResult;
  result = pthread_join(writerThread, &returnResult);
  if (result != 0)
  {
    cerr << "Could not join the writer thread." << endl;
    return EXIT_FAILURE;
  }
  result = pthread_join(readerThread, &returnResult);
  if (result != 0)
  {
    cerr << "Could not join the reader thread." << endl;
    return EXIT_FAILURE;
  }
  for(int i = 0; i < MESSAGES; i++)
```

```
  {
    cout << container[i] << " ";
    if(i != 0 and i % 20 == 0)
    {
      cout << endl;
      sleep(1);
    }
  }
  cout << endl;

  return EXIT_SUCCESS;
}
```

Appendix C

Modification done to ALIEVE

Two ALIEVE c++ classes, AliEveMUONTrack and AliEveMUONData, were enhanced by adding to them functions which display muon HLT hits and tracks. These new functions are shown below.

C.1 New functions added to class AliEveMUONTrack

```
void AliEveMUONTrack::PrintMansoTrackInfo()
{
 if (!fMansoTrack)
  {
    cout << "No manso track found" << endl;
    return;
  }
  fMansoTrack->Print("all");
}

void  AliEveMUONTrack::MakeMansoTrack(AliHLTMUONMansoTrack *track)
{
  fIsMansoTrack = kTRUE;
  fMansoTrack = new AliHLTMUONMansoTrack(track->Id(),track->Sign(),

track->Px(),track->Py(),track->Pz(),track->Chi2(),track->TriggerRecord(),
  track->Hit(7),track->Hit(8),
  track->Hit(9),track->Hit(10));

  char form[1000];
  sprintf(form,"MansoTrack %2d ", fLabel);
```

```
SetName(form);
SetLineStyle(2);
SetLineColor(0);

Float_t xr[28], yr[28], zr[28];
Int_t chr[28];

Float_t pt    = 0.0;
Float_t pv[3] = {0., 0., 0.};

for (Int_t i = 0; i < 28; i++)
{
 xr[i]=yr[i]=zr[i]=0.0;
 chr[i]=-1;
}

pt = fMansoTrack->Pt();
printf("Set line color = %d \n",ColorIndex(pt));
SetLineColor(ColorIndex(pt));

pv[0] = fMansoTrack->Px();
pv[1] = fMansoTrack->Py();
pv[2] = fMansoTrack->Pz();
fP.Set(pv);

Int_t nHit = 0;

for (Int_t chamber = 7; chamber <= 10; chamber++)
{
  const AliHLTMUONRecHit* hit = fMansoTrack->Hit(chamber);
  if(hit != NULL)
  {
     xr[nHit] = hit->X();
     yr[nHit] = hit->Y();
     zr[nHit] = hit->Z();
     chr[nHit] = hit->Chamber();
     SetPoint(fCount,xr[nHit],yr[nHit],zr[nHit]);
     fCount++;
     nHit++;
  }

}
```

```
  const AliHLTMUONTriggerRecord* triggerRecord =
fMansoTrack->TriggerRecord();
  if (triggerRecord != NULL)
  {
    Float_t x, y, z;
    for (Int_t i = 11; i <= 14; i++)
    {
       x = triggerRecord->X(i);
       y = triggerRecord->Y(i);
       z = triggerRecord->Z(i);

       SetPoint(fCount,x,y,z);
       fCount++;
    }
  }
}
```

C.2 New function added to class AliEveMUONData

```
void AliEveMUONData::LoadHLTHit(AliHLTMUONRecHit* recHit)
{
  Int_t cha = recHit->Chamber();
  if (cha > 1)
  {
    fChambers[cha-1]->RegisterHit(recHit->DetElemId(),
       recHit->X(),recHit->Y(),recHit->Z());
  }
}
```

Appendix D

Values used to test message queue parameters

Test Number	Push Parameters		Pop Parameters	
	MessageType	count	MessageType	maxCount
1 - 32	Integer Array	10	Integer Array	10
33 - 64	Integer Array	10	Integer Array	1
65 - 96	Integer Array	1	Integer Array	10
97 - 128	Integer Array	1	Integer Array	1

Table D.1: This table shows the values of the *MessageType* and *count* of the *Push* method and the values of *MessageType* and *maxCount* of the *Pop* method of the message queue class

Test Number	Push Parameters			Pop Parameters	
	wakeupReader	timeout	writePartial	wait	timeout
1,33,65,97	kTRUE	-1	kFALSE	kFALSE	-1
2,34,66,98	kTRUE	-1	kFALSE	kFALSE	2
3,35,67,99	kTRUE	-1	kFALSE	kTRUE	-1
4,36,68,100	kTRUE	-1	kFALSE	kTRUE	2
5,37,69,101	kTRUE	-1	kTRUE	kFALSE	-1
6,38,70,102	kTRUE	-1	kTRUE	kFALSE	2
7,39,71,103	kTRUE	-1	kTRUE	kTRUE	-1
8,40,72,104	kTRUE	-1	kTRUE	kTRUE	2
9,41,73,105	kTRUE	2	kFALSE	kFALSE	-1
10,42,74,106	kTRUE	2	kFALSE	kFALSE	2
11,43,75,107	kTRUE	2	kFALSE	kTRUE	-1
12,44,76,108	kTRUE	2	kFALSE	kTRUE	2
13,45,77,109	kTRUE	2	kTRUE	kFALSE	-1
14,46,78,110	kTRUE	2	kTRUE	kFALSE	2
15,47,79,111	kTRUE	2	kTRUE	kTRUE	-1
16,48,80,112	kTRUE	2	kTRUE	kTRUE	2
17,49,81,113	kFALSE	-1	kFALSE	kFALSE	-1
18,50,82,114	kFALSE	-1	kFALSE	kFALSE	2
19,51,83,115	kFALSE	-1	kFALSE	kTRUE	-1
20,52,84,116	kFALSE	-1	kFALSE	kTRUE	2
21,53,85,117	kFALSE	-1	kTRUE	kFALSE	-1
22,54,86,118	kFALSE	-1	kTRUE	kFALSE	2
23,55,87,119	kFALSE	-1	kTRUE	kTRUE	-1
24,56,88,120	kFALSE	-1	kTRUE	kTRUE	2
25,57,89,121	kFALSE	2	kFALSE	kFALSE	-1
26,58,90,122	kFALSE	2	kFALSE	kFALSE	2
27,59,91,123	kFALSE	2	kFALSE	kTRUE	-1
28,60,92,124	kFALSE	2	kFALSE	kTRUE	2
29,61,93,125	kFALSE	2	kTRUE	kFALSE	-1
30,62,94,126	kFALSE	2	kTRUE	kFALSE	2
31,63,95,127	kFALSE	2	kTRUE	kTRUE	-1
32,64,96,128	kFALSE	2	kTRUE	kTRUE	2

Table D.2: This table shows the value given to *Push* (*wakeupReader*, *timeout*, *writePartial*), and *Pop* parameters (*wait*, *timeout*)

Tests 1 - 28		Tests 29 - 56		Tests 57 - 84	
Test	AvPoppingRate(MHz)	Test	AvPoppingRate(MHz)	Test	AvPoppingRate(MHz)
1	1.38083	29	0.00767683	57	0.00147888
2	1.29708	30	0.00147614	58	0.00143994
3	1.33384	31	0.00151418	59	0.00135963
4	0.654980	32	0.335343	60	0.00575267
5	0.558922	33	0.140243	61	0.00152468
6	0.523599	34	0.141327	62	0.00154295
7	0.166026	35	0.156260	63	0.00138607
8	0.457161	36	0.0875554	64	0.00225255
9	1.43149	37	0.0893466	65	0.364969
10	1.28639	38	0.139284	66	0.364268
11	1.34758	39	0.132368	67	0.339686
12	0.553673	40	0.0153827	68	0.136708
13	0.196812	41	0.147772	69	0.364873
14	0.646916	42	0.150484	70	0.289469
15	0.329672	43	0.144185	71	0.314902
16	0.445420	44	0.0215761	72	0.111070
17	0.00390986	45	0.127354	73	0.279120
18	0.00127525	46	0.109349	74	0.488400
19	0.0131019	47	0.0356449	75	0.468840
20	0.552639	48	0.0742369	76	0.0139098
21	0.00142092	49	0.00171653	77	0.288785
22	0.00140462	50	0.00154387	78	0.358909
23	0.0015359	51	0.00156098	79	0.298760
24	0.522225	52	0.0506563	80	0.0619606
25	0.0053726	53	0.00103024	81	0.00115083
26	0.00425173	54	0.00213492	82	0.0009321
27	0.0032042	55	0.00145767	83	0.00100879
28	0.0484355	56	0.0633168	84	0.0264132

Table D.3: Average popping rates for the first 84 tests

Tests 85 - 106		Tests 107 - 128	
Test	AvPoppingRate (MHz)	Test	AvPoppingRate (MHz)
85	0.000953997	107	0.150503
86	0.000974427	108	0.0285577
87	0.00108303	109	0.0798826
88	0.0538113	110	0.141801
89	0.00109217	111	0.141659
90	0.00102146	112	0.0304901
91	0.00101304	113	0.00148005
92	0.00915707	114	0.00134197
93	0.00170494	115	0.00128502
94	0.00095718	116	0.0399955
95	0.000956532	117	0.0289018
96	0.0168174	118	0.00111108
97	0.153216	119	0.00130358
98	0.141715	120	0.00555094
99	0.157131	121	0.000971308
100	0.0348096	122	0.00191149
101	0.117394	123	0.00123649
102	0.139779	124	0.0942617
103	0.0717772	125	0.00287181
104	0.0400444	126	0.00252134
105	0.134125	127	0.00100372
106	0.143290	128	0.0217365

Table D.4: Average popping rates for the last 44 tests

Appendix E

Message queue and dHLTdumpraw C++ codes

The C++ codes for the message queue and the dHLTdumpraw utility are included in the attached Compact Disk (CD).

Bibliography

[1] L. Evans and P. Brynat. The CERN large hardron collider: Accelators and experiments. *Jinst*, August 2008.

[2] ALICE Collaboration. ALICE Technical Proposal for A Large Ion Collider Experiment at the CERN LHC. Technical report, CERN/LHCC/95-97 LHCC/P3, December 1995.

[3] The ATLAS Collaboration. ATLAS - Technical Proposal for a General-Purpose pp Experiment at the Large Hadron Collider at CERN. Technical report, CERN/LHCC/94-43 LHCC/P2, 1994.

[4] The CMS Collaboration. CMS - The Compact Muon Solenoid Technical Proposal. Technical report, CERN/LHCC/94-38 LHCC/P1, 1994.

[5] The LHCb Collaboration. LHCb Technical Proposal - A Large Hadron Collider Beauty Experiment for Precision Measurements of CP Violation and Rare Decays. Technical report, CERN/LHCC/98-4 LHCC/P4, 1998.

[6] http://aliceinfo.cern.ch/public/en/chapter2/chap2experiment-en.html.

[7] K. Aamodt *et al.* The ALICE experiment at the CERN LHC. *JINST*, 2008.

[8] ALICE Collaboration. ALICE: Physics Performance Report, Volume I. *JOURNAL OF PHYSICS G: NUCLEAR AND PARTICLE PHYSICS*, 2004.

[9] ALICE Collaboration. ALICE Technical Design Report of the Dimuon Forward Spectrometer. Technical report, CERN / LHCC 99-22 ALICE TDR 5, 1999.

[10] ALICE Collaboration. ALICE Addendum to the Technical Design Report of the Dimuon Forward Spectrometer. Technical report, CERN/ LHCC 2000-046 Addendum 1 to ALICE TDR 5, 2000.

[11] F. Christian. The muon spectrometer of the alice. In *5th International Conference on Physics and Astrophysics of Quark Gluon Plasma*.

[12] P. Dupieux. The ALICE Muon Spectrometer and related Physics. In *XLV International Winter Meeting on Nuclear Physics*, BORMIO, Italy, January 14-21 2007.

[13] ALICE Collaboration. Technical design report of the trigger, Data Acquisition, High-Level Trigger and Control System. Technical report, CERN, 2004.

[14] R. Bramm *et al.* High-level trigger system for the LHC ALICE experiment. *Nuclear Instruments and Methods in Physics Research*, 2003.

[15] T.M. Steinbeck. *A Modular and Fault-Tolerant Data Transport Framework*. PhD thesis, http://www.ub.uni-heidelberg.de/archiv/4575/, 2004.

[16] http://aliceinfo.cern.ch/public/en/chapter2/chap2_dim_spec.html.

[17] G. Blanchard, Ph. Crochet, and P. Dupieux. The local trigger electronics of the ALICE dimuon trigger. *ALICE-EN-2003-010*, 2003.

[18] R. Arnaldi *et al.* Performances of a Prototype for the ALICE Muon Trigger at LHC. *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, 51(3), 2004.

[19] R. Arnaldi *et al.* The ALICE dimuon trigger: overview and electronics prototypes. *Nuclear Instruments and Methods in Physics Research A*, 456:126–131, 2000.

[20] F. Guerin, F. Yermia, P. Dupieux, P. Rosnet, and E. Vercellin. ALICE muon trigger performance. *ALICE-INT-2006-0002*, 2006.

[21] T. Alt *et al.* The ALICE High Level Trigger. *JOURNAL OF PHYSICS G: NUCLEAR AND PARTICLE PHYSICS*, 2004.

[22] J. Wagner, V. Lindenstruth, M. Richter, P. Steinbeck, and J. Thader. Lossless data compression for ALICE HLT. *ALICE-INT-2008-020 version 1.0*, 2008.

[23] T. Alt *et al.* Benchmarks and Implementation of the ALICE High Level Trigger. *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, 53(3), June 2006.

[24] The ALICE Dimuon Spectrometer High-Level Trigger Collaboration. ALICE Dimuon High-Level Trigger: Project Review. *ALICE-INT-2007-022*, October 2007.

[25] B. Becker. *Development of a High-Level Trigger for the Dimuon Spectrometer of the ALICE Experiment at the Large Hadron Collider*. PhD thesis, University of Cape Town, 2006.

[26] S. Bablok *et al.* ALICE High Level Trigger Interfaces and Data Organisation. In *Proc. Computing in High Energy and Nuclear Physics Conf. 2006 (CHEP 2006 - Mumbai (India))*, 2006.

[27] T.M. Steinbeck, V. Lindenstruth, and M.W. Schulz. An Object-Oriented Network-Transparent Data Transportation Framework. *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, 49(2), April 2002.

[28] B. Becker *et al.* Implementation of a hit reconstruction algorithm for the ALICE Dimuon High-Level Trigger. *ALICE-INT-2008-013 version 1.0*, 2007.

[29] ALICE MUON spectrometer collaboration. Raw data format for the Muon spectrometer. *ALICE-INT-2005-012*, April 2005.

[30] G. Grastveit *et al.* FPGA Co-processor for the ALICE High Level Trigger. *Computing in High Energy and Nuclear Physics*, pages 24–28, March 2003.

[31] Z.Z. Vilakazi. The dimuon High Level Trigger. In *5th International Conference on Physics and Astrophysics of Quark Gluon Plasma*, volume 50 of *Journal of Physics: Conference Series*, pages 381–384. Institute of Physics Publishing, 2006.

[32] F. Manso *et al.* A first algorithm for a dimuon High Level Trigger. *ALICE-INT-2002-04*, February 2002.

[33] B. Becker *et al.* The ALICE dimuon spectrometer high level trigger. In *IEEE NSS Conference*, Dresden, German, October 2008.

[34] R. Fearick. Private communication.

[35] M. Richter *et al.* High Level Trigger Applications for the ALICE Experiment. *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, 55, February 2008.

[36] R. Brun *et al. ROOT Users Guide*. http://root.cern.ch.

[37] http://hep.phy.uct.ac.za/viewcvs/viewcvs.cgi/dhlt/src/monitor/.

[38] http://alisoft.cern.ch/viewvc/trunk/hlt/muon/utils/dhltdumpraw.cxx?revision=27748&root=aliroot&view=markup.

[39] The ALICE Offline Bible. http://aliceinfo.cern.ch/Offline/.

[40] ALICE Collaboration. ALICE Technical Design Report of the Computing. Technical report, CERN-LHCC-2005-018, 2005.

[41] R. Brun, F. Bruyant, M. Maire, A.C. McPherson, and P. Zanarini. *GEANT3 User Guide*. CERN Data Handling Division DD/EE/84-1, 1985.

[42] http://www.w3.org/xml.

[43] http://www.python.org.

[44] T.M. Steinbeck, V. Lindenstruth, and H. Tilsner. A Control Software for the ALICE High Level Trigger. In *Proceedings of the Computing in High Energy Physics 2004 (CHEP04)*, 2004.

[45] http://alisoft.cern.ch/viewvc/trunk/hlt/muon/macros/runchain.c?revision=30814&root=aliroot&view=markup.

[46] A. Szostak. PhD thesis, (To be published).