Development and Integration of on-line Data Analysis for the ALICE Experiment

Matthias Richter

Dissertation for the degree philosophiae doctor (PhD)
at the University of Bergen

February 06, 2009

Abstract

The ALICE detector setup is a dedicated experiment in Heavy Ion Physics, located at the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN)/ Geneva. Its various sub-detectors are electronically read out by a few millions of channels and are expected to provide a huge sample of data for the investigation of strongly interacting matter. A data rate of up to 25 $GByte/s$ imposes a challenge to both storage facilities and subsequent data analysis. On-line data processing is applied in order to reduce the data volume and selection of interesting events is suggested to increase the valuable information in the recorded data.

In general, trigger systems cover the task of event selection. As the first signal needs to be available immediately after the collision, often a multi-level triggering scheme is applied. Fast detectors are deployed to generate different levels of hardware triggers. Subsequent software triggers accomplish event selection on the basis of on-line analysis.

A High-Level Trigger (HLT) system has been implemented in ALICE which provides sufficient computing resources and infrastructure. It allows on-line event reconstruction at the full data rate and generates trigger information based on reconstructed events. HLT combines efficiently the communities of computing and physics. It thus requires a modular design for optimized workflows and well defined interfaces for exact communication as an integral part of the project. The ability of selection and rejection of events has direct impact on the data analysis performance and physics results. On-line algorithms have therefore to be evaluated and compared to the results of the general analysis.

This thesis has contributed to the overall design of on-line data processing with ALICE HLT. Data flows have been developed and implemented, and a modular concept based on a common HLT analysis framework and abstract interfaces has been established. HLT on-line analysis has been integrated into the ALICE computing model. The work describes in particular the important aspects which have been considered during the design of modularity, the HLT analysis framework and related software, as well as the application of all new developments for one specific sub-detector.

Acknowledgments

Now the time has come to hand in my thesis. I am happy to arrive at this point and I am grateful to many people supporting me and helping to get here. I joined the ALICE collaboration and the experimental nuclear physics group in Bergen in 2004 in a very important period of development and detector commissioning. I was lucky to participate in the exciting phase of commissioning of the today's most powerful particle accelerator, the Large Hadron Collider, and one of the experiments build in conjunction with its development.

First of all, I want to express my gratitude to my supervisors at the University of Bergen, Prof. Dieter Röhrich and Prof. Kjetil Ullaland. Dieter after all is one of the reasons which led me to Bergen, I still remember his reply on my formal application in 2003. I came into an environment which gave me best chances for development in all directions. Thank you for the establishment of a unique group covering research reaching from physics to electronics, detector development, and software engineering. The nice discussions, social and professional input have been an irreplaceable guidance. I appreciate the excellent financial backup which allows and encourages to present results at conferences.

Kjetil has been a good adviser at all times, and last but not least a perfect teacher in Norwegian by consequently challenging me in talking Bergensk. He provided me with interesting tasks of software development and gave me the opportunity to learn more about hardware and electronics. I like his straight forward get-to-the-point philosophy which helped me a lot also in the final phase of this thesis. Last but not least we share a similar understanding of humor.

I am grateful to all my colleagues at the Department of Physics and Technology, especially Hongyan Yang, Kalliopi Kanaki, Håvard Helstrup, Johan Alme, Ketil Røed, Dominik Fehlker, Dag Larsen, Sebastian Bablok, Kenneth Aamodt, Gaute Øvrebekk, Boris Wagner, Agnes Nyiri, and Sven Zschocke. Thank you all for a nice time in Bergen and at places around the world.

I also would like to thank our close collaborators at the Kirchhoff Institute for Physics at the University of Heidelberg, especially Prof. Volker Lindenstruth for support in the startup phase of my work in Bergen. Timm Steinbeck, Jochen Thäder and Torsten Alt have been excellent partners to work with. I appreciate all the nice discussions and constructive development. Thanks for keeping the computing cluster running at CERN.

This work would not have been possible without the support from my family I am in love with. My wife Maria had to carry a huge portion of the load and deserves my deepest gratitude. You perfectly managed to tackle daily life with two small kids and my physical and mental absence. Thank you for always keeping up the atmosphere in our family and believing in me, and so I do. Both my parents and my parents-in-law have been helpful all the time and I want to thank them for stepping in whenever a babysitter or any help was urgently needed. I thank my kids teaching me the relevance of things. There is certainly a world beyond physics, although Hanna, now at the age of three, will soon ask more about what I am doing. Clemens, not older than 7 months now, always motivated me with his smile and incredibly rapid development.

I am in debt to Jochen Thäder and Dr. Kurt Richter for reading my thesis carefully and helping to get rid of that number of little misprints and inconsistencies one gets blind about over time.

I want to thank the University of Bergen for financial support and the four-year stipend allowing me to accomplish the presented work.

Certain moments in life are unpredictable and have a remarkable impact despite of seeming to be incidentals. So I finally want to thank my good friends and tour mates Halvor and Karsten for various lessons in the norwegian mountains and telemark backcountry skiing. Sitting on top of a mountain in the Bernina Alps in 2003, Karsten eagerly encouraged me to apply for a position in Norway. A short discussion of big consequences, and with a good outcome.

Matthias

Bergen, February 2009

Contents

List of Figures

List of Tables

1. Introduction

High energy nuclear physics studies matter in extreme conditions. The main idea is the exploration of the phase diagram of strongly interacting matter. Phenomena in this field are usually studied by investigation of collisions of accelerated particles. The decay products of the generated intermediate state of matter are detected and allow us to gain knowledge about the properties of the medium. Many particle accelerators of different scale and performance have been built in the past decades and new ambitious projects are currently underway. The Large Hadron Collider at CERN/Geneva will provide several experiments with proton and heavy ion beams for particle collisions at energies of $\sqrt{s} = 14\ TeV$ and $5.5\ TeV$ respectively per nucleon-nucleon pair. The presented work has been accomplished in conjunction with the development of the ALICE experiment, a unique and in many aspects innovative experimental setup dedicated to heavy ion physics.

Due to the scale of the experiment and its high readout granularity, the produced data volume imposes a challenge to computing systems. For our experiment, data rates can peak at $25\ GByte/s$, producing a data volume both difficult to store and to analyze if not pre-selected and treated directly during the process of data taking. Fast on-line data analysis at the full input rate allows to reconstruct all events and to reduce the data rate by using e.g. data compression techniques.

The necessity of on-line analysis is furthermore motivated by the nature of the collision process itself. Rare processes are the main target of investigation of new phenomena but are often embedded into a huge number of background events. As one possible solution for this problem, on-line selection of interesting events allows the efficient collection of data. It helps to sharpen the results of the experiment, and thus increasing the benefit-cost ratio which is a relevant question in fundamental research. For this purpose, the ALICE High-Level Trigger (HLT) provides a powerful computing resource and dedicated infrastructure for the task of on-line analysis.

The past five years have been spent on the final commissioning of the detector. It was in particular a phase of final implementation of detector electronics and control systems, as well as design of the data reconstruction scheme. Naturally, such a phase is not a distinct and straight road, but more a path to be followed in a slightly known and sometimes completely new terrain. Based on experience made so far during the development, the decision where to continue had to be evaluated on a short timescale, sometimes on a monthly or even daily bases. However, all development pointed into one direction, the final commissioning of the accelerator and the experiments.

Consequently, also the scope of the presented work was not a fixed and rigid task. Algorithms and software prototypes have been developed before and in parallel to this work. Critical evaluation of the project made it evident to the group that emerging solutions and implemented prototypes needed a solid foundation. As a result of this, the focus of this work has been set to be a project study for integrated software solutions and abstraction layers for data readout and processing of detectors in ALICE. The design and development is motivated by the integration of on-line analysis for one particular sub-detector. Understanding the full scale of the ALICE HLT system was the requirement for implementation of analysis components. The new design of the HLT software modules is a direct outcome of the presented work and makes it possible for the user to integrate the desired physics analysis and algorithms with minimal effort. A dedicated interface hides all complex processes and inter-process communication.

Other important aspects, quality assurance and reliability of the trigger, require the HLT analysis to be part of the off-line data processing environment. The ALICE off-line project provides a complete framework for the simulation and reconstruction of events, as well as the subsequent physics analysis. In order to evaluate the selectivity and performance of the High-Level Trigger, algorithms must be compared with the standard analysis and with respect to simulated data. This requirement motivates the complete separation of the actual HLT analysis from the data transport functionality.

One major challenge of the project turned out to be the efficient combination of different communities and sub-systems. Success is achieved by efficiently combining skills of scientists and developers from different communities, which requires optimized work-flows for cross-subject working groups. A major part of the work is related to system design and interface implementations, enabling each side to work as efficient as possible. This paradigm also takes the value and cost of human resources into account.

Due to the nature of the project, this thesis covers in a large part computational aspects. The reader is expected to be familiar with the principles of object oriented programming or a basic understanding of object oriented approaches. Though, the concept of base classes and overloading of virtual function will be introduced shortly. An appropriate software design and tools for the purpose of on-line data analysis with the ALICE High-Level Trigger are the major outcome of this work and will be described in detail in chapters 4 and 5. All development has been carried out under the aspect of the real application of the framework and chapter 7 is dedicated to concrete realization and integration of appropriate on-line analysis for one particular sub-detector, the Time Projection Chamber (TPC).

Although finally this work could not use any real physics data due to the delay in the LHC startup phase, the operation of ALICE in 2008 provided a huge sample of data to study detector properties and system integration. The detector system was running continuously for about 7 months, delivering various data sets like e.g. cosmic ray and noise measurements. Detailed tests have been accomplished in order to commission not only the different sub-systems stand-alone, but also the interplay and communication in between. After a long phase of development and final commissioning, the ALICE High-Level Trigger is ready for on-line data analysis and is awaiting first data from LHC operation.

2. ALICE and the Large Hadron Collider

The field of sub-atomic physics studies the fundamental interactions between elementary particles and the properties of strongly interacting matter. Though many particles of elementary nature have been discovered and studied in the past decades, a lot of effects and aspects remain uncertain. Among the developed theories, Quantum Field Theories have been very successful in describing a large number of experimental observations.

Most of the experiments in this field entail very large experimental setups. The most powerful tools are provided by particle accelerators which have been continuously developed to allow acceleration of particles to higher and higher energies including a broad variety of particles, e.g. electrons, protons and heavy ions. Particle accelerators of many different scales have been developed and operated so far. The currently largest machine of such type is the **L**arge **H**adron **C**ollider at CERN/Geneva ([1], Figure 2.1).

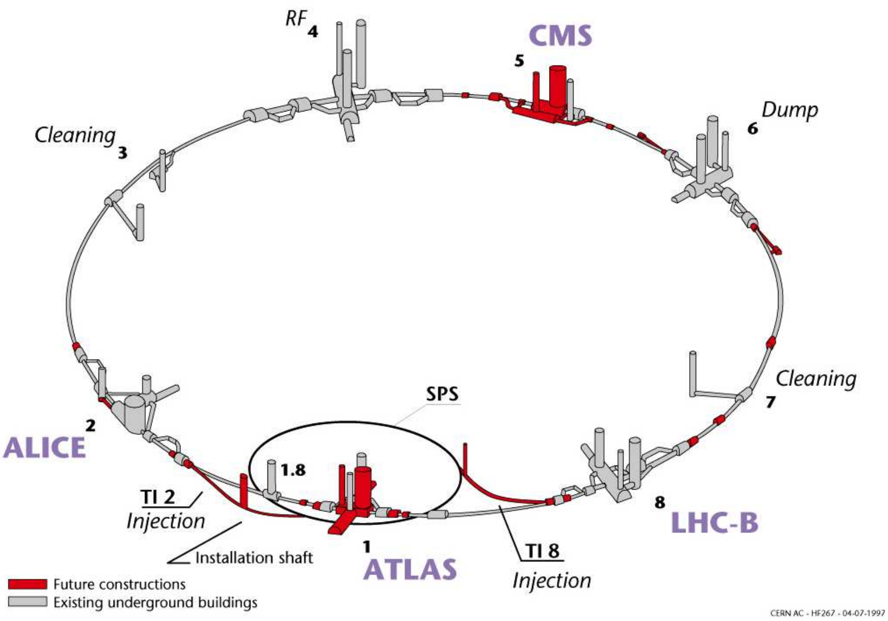


Figure 2.1: Outline of the Large Hadron Collider at CERN/Geneva [2]. The ALICE experiment is located at Point 2, lower left.

The accelerator finished its final commissioning in September 2008 and was inaugurated in autumn 2008. A tunnel with circumference of 26.7 km hosts the apparatus intended to accelerate protons to the energy of 7 TeV. In two adjacent beam lines particles circulate in opposite direction. The two beam lines intercept at four experimental areas and allow to collide the particle beams. At the four interactions points, the main experiments are located.

Primarily built for the acceleration of protons, LHC also has a Heavy Ion research program which includes experiments with lead (Pb) ion beams. Collisions of heavy ions allow a different kind of physics with focus on the properties of a very dense medium.

ALICE - **A** **L**arge **I**on **C**ollider **E**xperiment is located at Point 2 of the LHC ring. The experiment is especially designed for the investigation of collisions of heavy ions creating a hot and dense matter. In particular, the properties of a special phase of matter will be studied. The so called *Quark Gluon Plasma (QGP)* will be introduced in Sec. 2.2.

2.1 Physics Motivation

Colliding beam or fixed target experiments have been built for various energy regions. The construction of bigger accelerators was always motivated by the wish of increased energy liberated in a collision.

Continuous theoretical and experimental investigation allow to establish models which describe the observed processes sufficiently within a certain energy scale. The so far most successful model describing sub-atomic interactions and processes is the so called *Standard Model*. It is a gauge theory describing the 16 known elementary particles, their anti-particles and interactions unifying three of the four fundamental interactions, *Electromagnetic*, *Weak*, and *Strong* interaction. Though, not free of contradictions, Standard Model describes many of the observed processes. The model has been probed in the current experimental energy scale and predictions are made for an extended energy scale. Still there are many aspects to be proven and physicists expect new phenomena at many frontiers.

One of the fundamental questions not answered by the Standard Model is the existence of massive gauge bosons. Gauge bosons are the quanta of the gauge field of a quantum field theory and are understood to mediate the interaction between elementary particles. One of the important aspects of a gauge theory and the understanding of nature is *Gauge Invariance* of the theory, meaning the theory is invariant under a certain symmetry transformation group, which is a property of the theory. For technical reasons, the gauge bosons are massless in this description and this is in contradiction with the observation of the massive exchange bosons of Weak interaction, the $\mathbf{Z}$ and $\mathbf{W}^{\pm}$ bosons.

In theory, masses of those two elementary particles are introduced ad hoc by adding a *Higgs field* with the property of *Spontaneous Symmetry Breaking*. This phenomenon occurs in gauge symmetric system with a non-symmetric vacuum state. In the mathematical description, the bosons get mass through their coupling to the *Higgs field* which also exhibits another fundamental particle, the *Higgs boson*. Whether this mathematical formalism can sufficiently describe nature beyond the so far investigated energy scale needs to be proven. Search for the Higgs boson is number one priority of today's particle physics.

Also, the nature of mass is unclear. A striking observation was the discovery of constituents of hadrons, the quarks, and an appearing lack of mass. Masses of individual quarks have been measured, and the effect has been observed that hadrons are often much heavier than the sum of their constituents. Quarks and Gluons, the gauge boson of the Strong Force mediating the interaction, are subject to strong interaction and its theory, *Quantum Chromo Dynamics (QCD)*, which has been incorporated into the Standard Model. Under normal condition, quarks are confined in hadrons and cannot be observed freely in nature. Recently a new state of matter has moved into focus, a dense and energetic state where quarks seem to exist freely, the so called Quark Gluon Plasma produced in ultra-relativistic heavy ion collisions. Quark Gluon Plasma is expected to have existed in a very early phase of the evolution of the universe. Its investigation is of special interest because it can illuminate regions of QCD, difficult to handle numerically, and helps to understand the fundamental principle of strong interaction.

2.2 Quark Gluon Plasma

The main goal of heavy ion physics is investigation of strongly interacting matter. The collision system under investigation possesses a very high energy density which is reached by colliding heavy nuclei at ultra-relativistic energies. Quarks and Gluons as the elementary particles of nuclear matter are confined within hadrons and cannot be observed independently. Theories in the 1980s predicted the possible deconfinement [3] and have triggered the search for observables appropriate for the study of this effect. Especially the phase transition between the bound hadronic phase of quarks and the deconfined phase has become of interest since the region of the unstable phase is very sensitive to new physics.

In the 1990s first signs of a new phase have been found at the CERN Super Proton Synchrotron (SPS) in which quarks and gluons can exist freely within a dense and continuous medium and can undergo direct interactions because of their deconfinement. The idea of the Quark Gluon Plasma was discussed in various publications, e.g. [4]. Research has focused on the verification of this discovery and the study of the properties of the created medium. The current understanding of the phase diagram of strongly interacting matter is outlined in Figure 2.2.

Though the existence of a Quark Gluon Plasma was still in question in the beginning of this decade, recent experimental data from RHIC[1] show strong evidence for the existence of the QGP and the community meanwhile is convinced of its existence. The investigation of its properties and in-medium effects have moved into focus. Formerly considered to be a state like a gas where particles are loosely bound, QGP turned out to have more the character of a fluid.

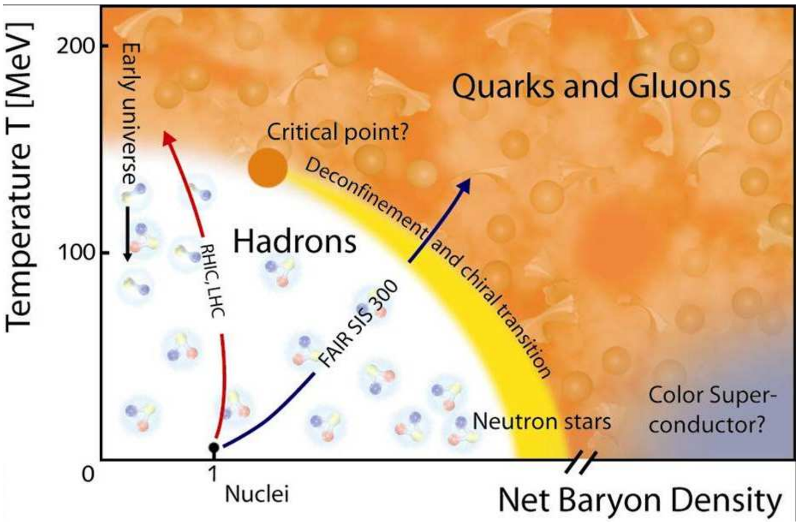


Figure 2.2: A schematic phase diagram of strongly interacting matter. The direction of exploration is depicted for facilities like RHIC, LHC, and upcoming FAIR[3]. From [5].

Several theories of the behavior of hot and dense matter have been developed and established after the discovery of QCD. The energy density and the strong coupling makes perturbative approaches of QCD inappropriate for theoretical calculations. The most used method for theoretical calculations and predictions is *Lattice Gauge Theory* where space-time has been discretized onto a lattice. Also, hydrodynamic models have been developed based on the assumption of liquid character of the QGP.

Quark Gluon Plasma is nowadays considered to be strongly coupled, motivating hydrodynamic models [6]. Analysis of data from RHIC is well advanced and ongoing. It has given answers and raised new questions. The community is now awaiting data from an extended energy scale at LHC.

Experimental methods to probe the QGP include preferably particles and processes originating from a very early stage of the collision. As those particles go through all stages of the evolution, they can provide information on the interaction of particles with the dense medium. In particular the investigation

[1] Relativistic Heavy Ion Collider at Brookhaven National Lab
[3] Facility for Antiproton and Ion Research at GSI Darmstadt/Germany

of Elliptic Flow, jets from hard collisions, and the response of heavy quarks to the medium is of special interest.

Elliptic Flow

Collisions at medium impact parameters create a spatial anisotropy which leads to a pressure gradient in the medium in an early stage before the evolution of the created state takes place. In hydrodynamic models the collective behavior of the medium is described by *Flow* components of a Fourier decomposition. The second component v_2 represents the spatial anisotropy. It is called *Elliptic Flow* because of the shape of the overlapping region of two colliding nuclei. It can be determined by measuring the azimuthal particle distribution with respect to the reaction plane.

Recent experimental data show a strong Elliptic Flow in heavy ion collisions which underlines the fluid-like behavior of the medium under investigation [7]. Figure 2.3 shows recent measurements of Elliptic Flow in Au-Au collisions at RHIC. The raise of the azimuthal anisotropy of heavy quarks emphasizes the hydrodynamic model. The medium has properties of an almost 'perfect fluid'.

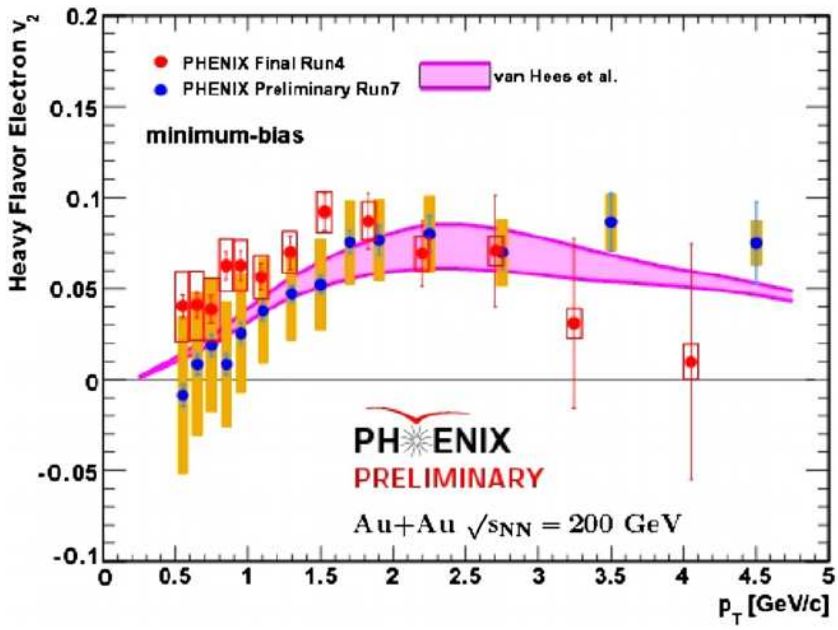


Figure 2.3: Azimuthal anisotropy parameter v_2 of heavy flavor electrons in minimum bias Au-Au collisions (Phenix). Data from Run 4 denoted as boxed dots. From [8].

Hard Collision and Jets

A collision at small impact parameter is referred to be a hard collision. The liberated energy reaches its maximum allowing the creation of a parton pair of high transverse momentum. As the two partons move away from each other, a fragmentation process leads to the generation of a bunch of particles directed into the primary direction, which is called a *jet*. While traversing the medium, partons are subject to modification due to interaction with the medium.

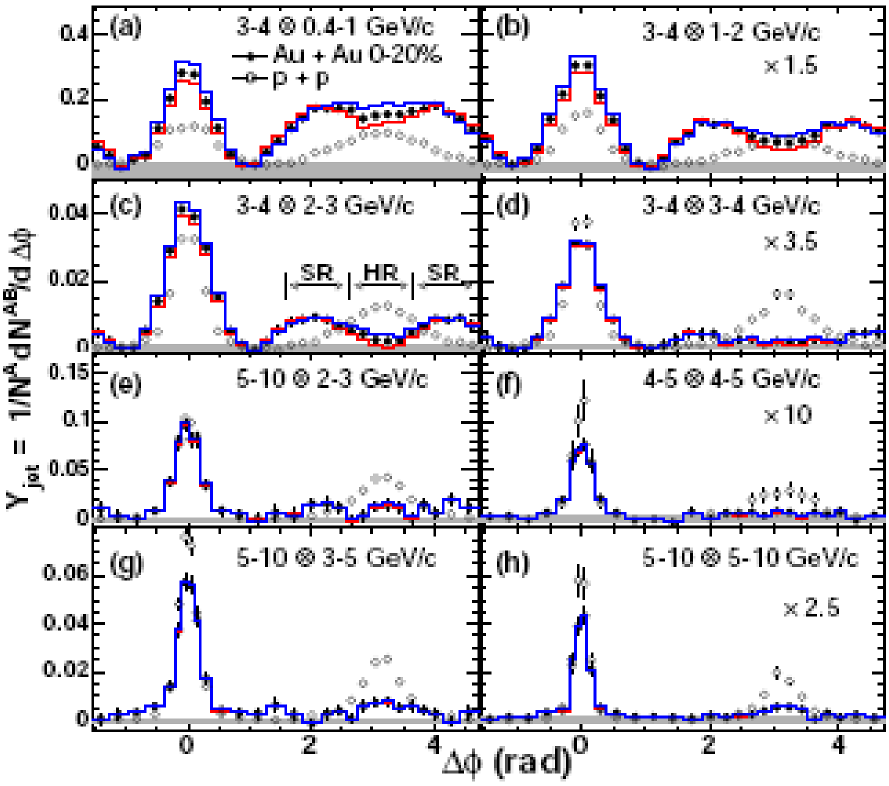


Figure 2.4: Jet correlation functions for Au Au collisions at $\sqrt{s_{NN}} = 200\ GeV$ (Phenix). From [9].

The effect is studied by measuring both jets escaping from the medium in opposites directions. Correlation functions as shown in Figure 2.4 indicate an increasing suppression of the jet traversing the medium with increasing centrality of the collision.

2.3 ALICE - A Large Ion Collider Experiment

ALICE [10, 11] is a multi-purpose experiment involving many different aspects and interests in Heavy Ion Physics. It consists of a variety of different sub-detector systems measuring and identifying hadrons, leptons and photons produced in the interaction.

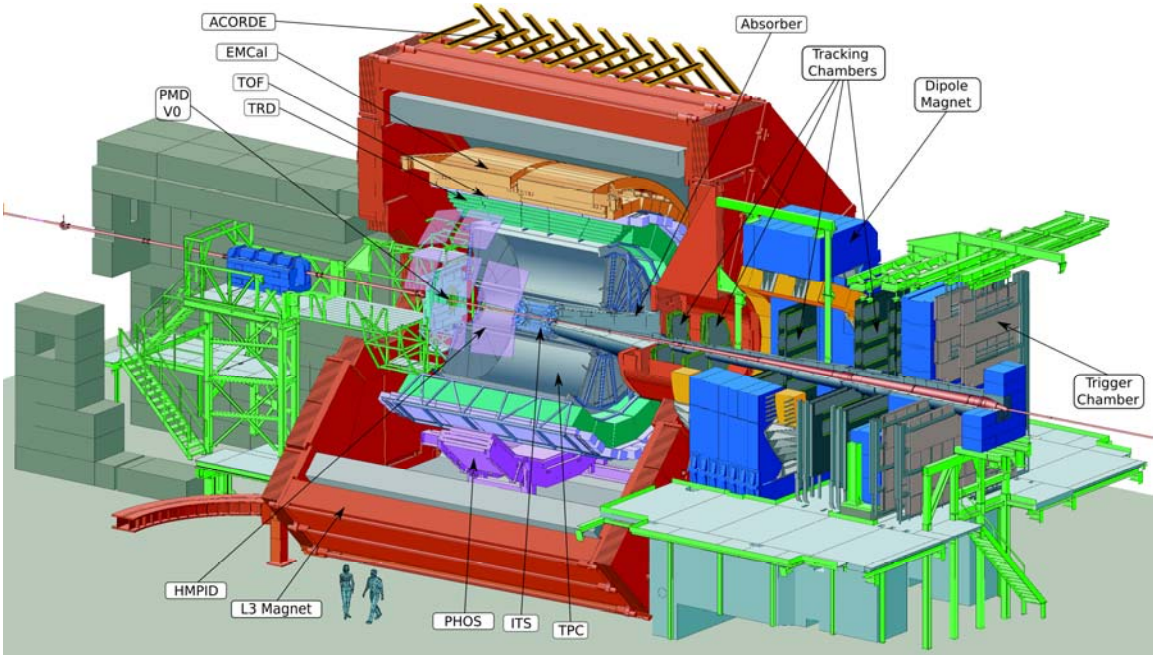


Figure 2.5: Sketch of the ALICE detector.

The apparatus follows at most the common design of particle detectors, a multi-layer setup of detectors of different type as outlined in Figure 2.5. The innermost layers are formed by position and tracking detectors of low material budget in order to influence particles as little as possible. In the outer layers, calorimeters finally stop the particles and measure energy.

The barrel section in the central rapidity region ($-0.9 \leq \eta \leq 0.9$) is implemented inside a solenoid magnet. The magnet of the former *L3* experiment at LEP Point 2 was used for the ALICE experiment with small modifications and is able to create a moderate magnetic field of up to $0.5\,T$. A silicon tracker of high resolution and relatively small dimensions - ITS - embeds the interaction point. A TPC of large dimensions embeds the ITS and is the main tracking detector in the central region. In the next layer, a TRD can discriminate electrons and positrons from other charged particles. TRD also implements local tracklet reconstruction in the hardware level and can contribute to the trigger. Particle identification is provided by detectors such as the TOF system and the HMPID. A crystal Photon spectrometer (PHOS) is located in the outer barrel section with limited coverage in rapidity and azimuthal angle. It completes together with the recently added electromagnetic calorimeter EMCAL the central section.

Outside of the solenoid magnet on one side of the experiment, a muon spectrometer covers a large rapidity range ($-4.0 \leq \eta \leq -2.4$). It is designed for the detection of muons originating from decay of the J/ψ and Υ resonances. For the purpose of event characterization and interaction trigger, a number of smaller detectors cover an acceptance region of $-3.4 \leq \eta \leq 5.1$, such as FMD and timing and veto detectors (T0/V0).

2.4 Event Reconstruction Paradigms

The ALICE experiment will operate at an event rate of 200 Hz and delivers a data rate up to 25 $GByte/s$. Since the event reconstruction and physics analysis cannot be carried out at this high rate with the desired accuracy, data is recorded to mass storage from where it is processed after a run has been accomplished. Since data processing is disconnected from the process of data taking it is referred to be performed *off-line*. Off-line computing makes furthermore use of the availability of the complete data set.

In contrast to that paradigm, *on-line* data processing is performed attached to the data taking itself. The overall processing time is reduced by optimization of algorithms for processing speed rather than providing the highest possible accuracy. On-line processing is carried out in a serial fashion and does not require the entire set of data.

The ALICE off-line project implements a comprehensive solution for detector simulation, event reconstruction, physics analysis and event visualization under the hood of the software package *AliRoot*. Design and implementation of AliRoot follow the concept of *Object Oriented Programming (OOP)*. It is based on the analysis framework *ROOT*[4] [12] and provides a development platform for the ALICE community. The computing model is described in detail in [13].

In order to meet the high demands to computing resources imposed by the recorded data volume within one run period, AliRoot builds on the distribution of data processing by utilization of the so called *GRID*. The term *GRID computing* has its origin in the application of a *computing grid* to a problem in order to gain more computing resources [14]. The philosophy of GRID computing entails the abstraction of computing resources and appropriate infrastructure. Both data and tasks are distributed transparently for the user.

2.5 Operation of the ALICE experiment

The experiment is divided into four subsystems under the hood of the ECS as outlined in Figure 2.6. Each of the systems controls a distinct task in the operation and data flow and implements FSMs on the main and sub-levels.

[4] http://root.cern.ch

The structure of the on-line systems is described in detail in [15]. Here it is introduced briefly for better understanding of subsequent sections.

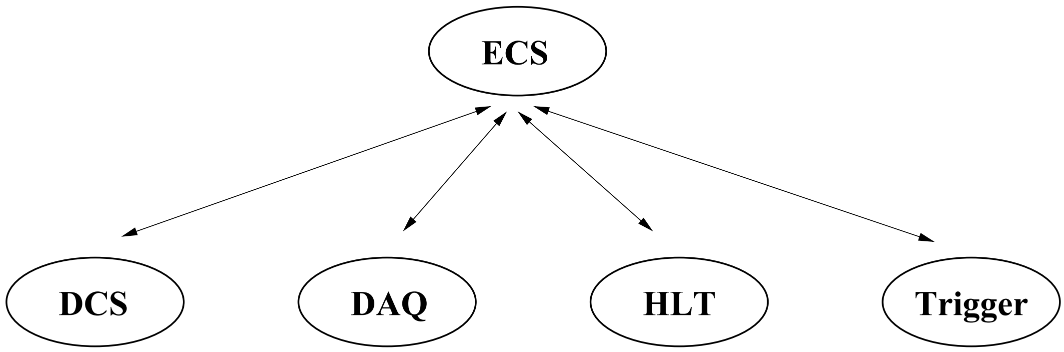


Figure 2.6: On-line systems of ALICE. ECS controls all operation and interplay, while communication between sub-systems is restricted.

Experiment Control System (ECS) [16] forms the main control layer. It contains the operator interface and allows the operation of the experiment from the control room. The ECS steers all sub-systems, communication is carried out as transitions of the implemented state machines.

Trigger deploys a 3 level triggering scheme. A Level 0 (L0) signal reaches detectors at 1.2 μs. L0 provides a very fast trigger signal. A second one, Level 1 (L1) is issued after 6.5 μs. The third level allows past-future protection and has been introduced in order to meet the requirements of detectors of slow readout. The Level 2 (L2) signal is issued after 88 μs and can be either an L2 *accept* or L2 *reject*. Past-future protection has been added for the sake of overlapping central Pb-Pb collisions which cannot be reconstructed due to the high particle multiplicity. The common *Trigger, Timing and Control (TTC)* project at LHC[5] defines a protocol for the transmission of timing and trigger signals [17] for the LHC and its experiments.

Data Acquisition's (DAQ) role in the system is the collection of data from all detectors, building events with respect to trigger classes, and transfer them to permanent storage.

Although ALICE consists of 18 different sub-detector systems with different design constraints and physics requirements, all use a common data transport solution. The *Detector Data Link (DDL)* is a hardware and protocol interface between the front-end electronics and the data acquisition. DAQ system deploys a 2-layer structure. The front-end machines, so called *Local Data*

[5] http://ttc.web.cern.ch/TTC/intro.html

Concentrators (LDC), host hardware devices receiving data. LDCs perform sub-event building on the DDL level. An event building network connects to the other layer, formed by *Global Data Collectors (GDC)* carrying out the final event building according to trigger classes. The GDC layer is connected to permanent storage. The complete system is described in [15].

According to the ALICE Technical Proposal [10], DAQ was designed to provide storage bandwidth of 1.25 $GByte/s$. This value has been chosen considering constraints imposed by technology, cost, storage capacity, and computing resources.

High-Level Trigger (HLT) is the focus of this work. In ALICE, HLT consists of a separated computing system providing DAQ with the necessary event selection information. HLT is designed to operate at an input data rate of 25 $GByte/s$. The layout of HLT will be described extensively in section 3.

The DAQ - HLT interplay implements three running modes:

(A) DAQ is running without HLT,

(B) HLT is fully included into the system, but its decision is not considered, and

(C) DAQ performs event and sub-event selection based on the trigger information from HLT.

In modes B and C, HLT is also treated like any other detector and generated data are stored as part of the event building.

Detector Control System (DCS) covers the tasks of controlling all technical and supporting systems of the detectors, such as the cooling system and the ventilation system. It also carries out the configuration and monitoring of the *Front-End Electronics (FEE)*.

The storage of on-line conditions during data taking has an important role in the subsequent data analysis. DCS provides on-line measurements of detector conditions which are important for the event reconstruction. HLT implements an interface to DCS ([18]) in order to fetch current values of data points and to provide these data to reconstruction algorithms. For off-line reconstruction, DCS values are stored in the *Offline Conditions Data Base (OCDB)*. The computing model and requirements regarding the availability of detector conditions are explained in [13].

3. High-Level Trigger

This section introduces the concept of the ALICE High-Level Trigger and its sub-systems. The overall structural design and layout including inter-communication within the HLT system are a major result of the presented work.

3.1 Conceptual Design

The ALICE HLT is designed to operate at a data input rate of 25 $GByte/s$. In order to meet the high computing demands, HLT entails a large PC farm of up to 1000 multi-processor computers and several software sub-systems. On the software side, a versatile on-line **Data Transport** framework based on the publish/subscribe paradigm, called PubSub framework (section 3.4.1), builds the core of the HLT. It interfaces to a complex **Data Analysis** (section 4) implementing the actual event reconstruction and triggering calculations. Figure 3.1 sketches the most important sub-systems and the communication flow in between.

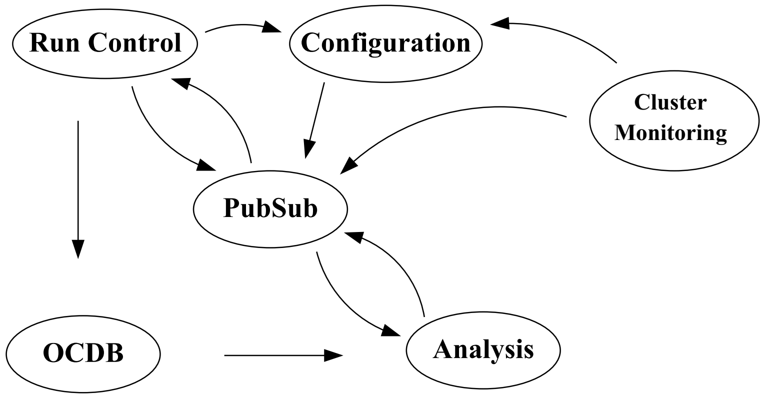


Figure 3.1: Sub-systems of ALICE HLT, the on-line Data Transport Framework PubSub takes a central role in the system.

A **Run Control** system interfaces to the ALICE ECS and provides the operator interface for daily shifts. A stable operation of a computing cluster of that scale also requires a fail-safe **Cluster Monitoring and Management**. A separate project has been launched in the course of ALICE HLT development and the application *System Management for Networked Embedded Systems and Clusters (SysMes)* suited for system management [19] has been created. HLT on-line processing running during data taking in ALICE is described by a configuration which is decoupled from the data transport and analysis.

A separate **HLT Configuration** software package defines the format and notation of configurations, data base, and the transcription into steering scripts for the on-line system. The OCDB is not an HLT system. It is in general required for analysis algorithms and allows in addition the transfer of information from the RunControl to the algorithm.

Each of the software systems forms a complex system on its own. This thesis has its focus on the data analysis framework, other sub-systems will be briefly introduced.

The challenge of ALICE HLT is not only imposed by the requirements in computing performance but also by the amount of data to be processed. This makes single high-performance computers inappropriate as those have limitations on the maximum data throughput. ALICE HLT has chosen the approach of diversified processing levels distributed over many computing nodes. The system implements a tree-like computing structure which allows to combine reduced data volume with increasing complexity and causality throughout the data processing as outlined in section 3.3.

The conceptual design of HLT is influenced by requirements imposed by the HLT physics program. The primary intention is the full reconstruction of events at the full data rate. Based on the reconstructed events, advanced analysis allows event selection by physics criteria. The HLT physics program includes search for open charm, jet analysis, and triggering on di-muon candidates, see also [20, 21, 22].

3.2 Processing Methodology

In order to achieve the desired data throughput, massive parallel computing is required for ALICE HLT. Parallelism can be applied on various levels in data processing which implies different architectural computing solutions and processing performance. In this section, different approaches in parallel computing are introduced in order to motivate the chosen solution for ALICE HLT.

Sequential Event Processing

Sequential event processing allows a straight forward solution. It is characterized by the one-to-one relationship between events and processes: one event is handled by one process. As a consequence, the smallest entity for parallel processing is one event and all data of a single event must be available on the same machine. In order to achieve a high data rate, whole events are distributed among many machines of a computing cluster or the GRID. The approach is sketched in Figure 3.2 and comes at the cost of high data transfer. The ALICE off-line event reconstruction makes use of sequential event processing.

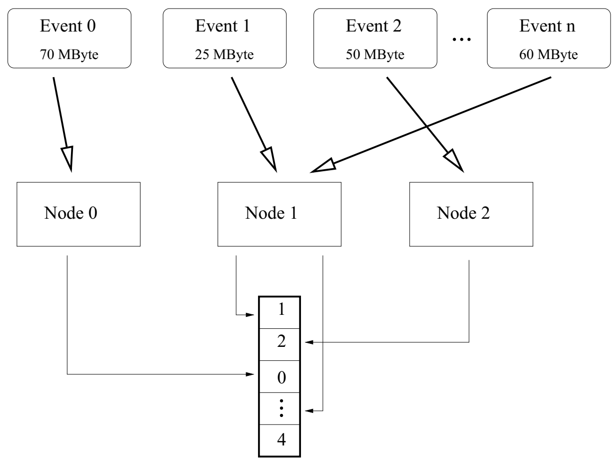


Figure 3.2: Sequential event reconstruction. Full events are shipped to many computing nodes. Depending on the processing time, the order of the events is changed.

A typical Pb-Pb event in ALICE has a size of roughly 50 to 100 $MByte$. Taking the DAQ bandwidth of 1.25 $GByte/s$ gives a data volume of roughly 100 $TByte/d$ and 20 $PByte/running\ period$. For efficient use of computing resources of a computing GRID data need to be shipped, which imposes obvious limitations. Even on a local high-performance computing cluster, HLT's target bandwidth of 25 $GByte/s$ cannot be reached by a normal network.

Parallel Event Fragment Processing

An important feature of the HLT on-line system is the implementation of parallelism on the level of reconstruction steps or event fragments. The event reconstruction is divided into sub-tasks working locally on a sub-set of data. E.g. cluster finding algorithms processing raw data and searching for space points in the detector can work locally on the level of sectors and readout partitions. Space points can be connected to tracks in the next stage on a sector level, tracks can be merged on the level of the whole sub-detector.

This approach is motivated by the fact that data are received by different nodes anyhow. The diversification of data transport from the detector front-end to Data Acquisition and on-line HLT is required by the peak data volume produced by the detector and the target event rate of the experiment. HLT's *Front-End processors (FEP)* form the receiving nodes and first processing layer at the same time. The distributed event reconstruction on sub-event level is illustrated in Figure 3.3.

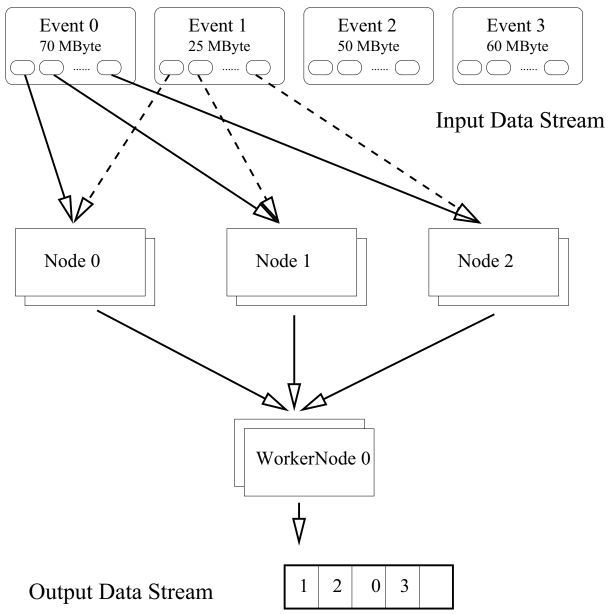


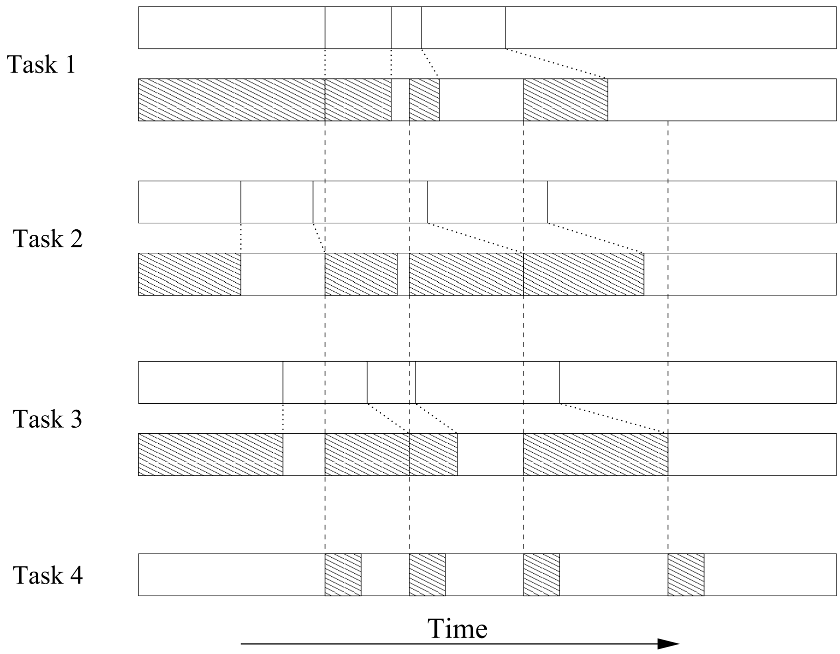
Figure 3.3: Parallel event fragment reconstruction. Data processing of parts of the event is distributed over many computing nodes. Data is received by many nodes and the *task* distribution follows the natural *data* distribution.

Pipelined Data Processing

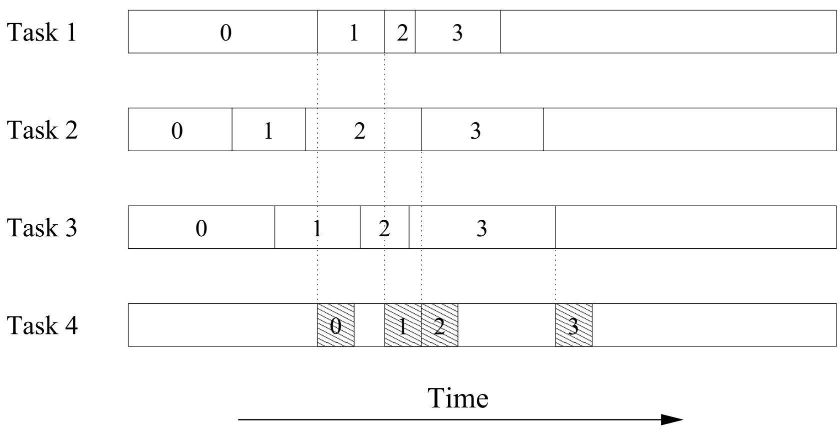
Pipelined data processing is a second paradigm, HLT on-line system takes advantage of. Each process immediately gets the next event after it finished the previous one. The distribution of events to tasks and processes is independent of other tasks. This technique allows a significantly higher data throughput due to the fact that the processing time for a task differs from event to event. In normal processing, all tasks have to wait until the last task has been finished. For the next event, another task needs the highest processing time. The pipeline stores finished events for all tasks of one level and propagates fully finished events to the next stage. On average, the events arrive faster at the next stage of the reconstruction (see Figure 3.4).

Pipelined data processing can easily be implemented by decoupling the actual data from the meta information communicated between processes. The concept of data block descriptors is also the foundation for efficient data exchange and will be introduced in section 3.4.3.

Efficient pipelined processing requires a sufficient number of output buffers for temporary storage. The number of events in the pipeline is only restricted by the available memory and the number of output buffers of each process.



(a) Non-pipelined data processing



(b) Pipelined data processing

Figure 3.4: Reduced processing time in pipelined event processing. Figure (a) shows schematically a processing sequence including 3 tasks with varying processing time for different events. Task 4 waits until all tasks can provide data for a specific event. The next event can be processed if task 4 is finished. The upper bar for tasks 1 to 3 illustrates the summed processing time and the corresponding displacement. Tasks are idle when waiting for each other to be finished. Figure (b) shows the same sequence for pipelined processing. Each task can process events independently of others.

Shared Memory based Data Exchange

The inter-process communication of HLT is designed to work with a minimum of overhead. One of the key paradigms is the optimization of data transport. Copying of data comes at the cost of performance, especially large data volumes need a significant amount of processing time. In order to avoid this, data are exchanged via *Shared Memory*. Normally, two processes do not have access to the same region in physical memory, *Shared Memory* is a specific approach to allow processes to use the same physical memory (section 3.4.2).

3.3 Data Processing Scheme

HLT makes use of parallel computing by implementing a processing hierarchy of independent layers. The first layer of processes receives detector raw data and extracts cluster and hit information. A subsequent layer reconstructs the event independently for each detector. A global reconstruction layer combines all information from the detector-reconstruction and calibration. Based on the result of the global reconstruction and run specific physics selection criteria, trigger decisions are calculated resulting in the selection of events or regions of interest. The general processing scheme is outlined in Figure 3.5

In order to achieve the high data throughput, all nodes within one layer work independently from each other. This working scheme is based on pipelined data processing and the fact of uncorrelated data sets on the level of the processing. E.g. clusters can be calculated from sets of raw data individually and clusters of disjoint sections of the detector can be connected to tracks independently. Each layer in the processing hierarchy reduces the data volume before passing data on to the next layer.

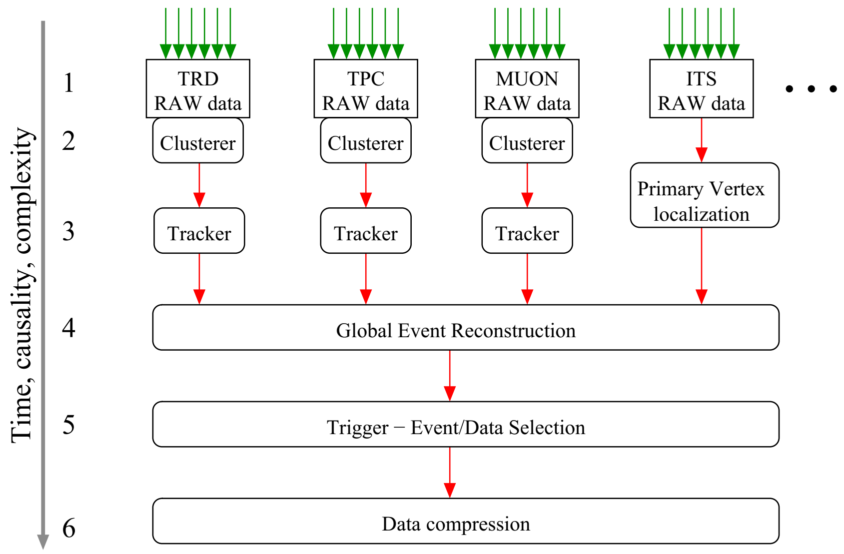


Figure 3.5: The general architecture of HLT entails a multi-stage processing scheme where processes of different levels carry out individual processing tasks independently. From [11].

3.3.1 The Concept of Components

In order to clarify the terms used throughout this thesis, the concept of HLT components is described in this section. In particular, what is meant when the term *HLT component* is used.

Data treatment within the HLT processing hierarchy is carried out by individual processes. These separated applications derive from the same base class of the data transport framework which provides the interface. Following identical working principles, the processes are referred to be *HLT components*. This term is also motivated by the modular concept. Components carry out different tasks in the HLT processing hierarchy, treating input data and producing new data for subsequent components.

Each component is an individual process which is started once at startup of the processing chain and implements a state logic. The state of the component can be changed by commands it receives via network ports.

The data transport framework implements in general three types of components: *data source components* load data from the input device or file into shared memory and create the corresponding data descriptors. *Data processing components (Processors)* subscribe to data of its parents, process the data and publish the result to the next level. Finally, *data sink components* implement the last stage in the chain, doing the appropriate action on the output of the chain (Figure 3.6).

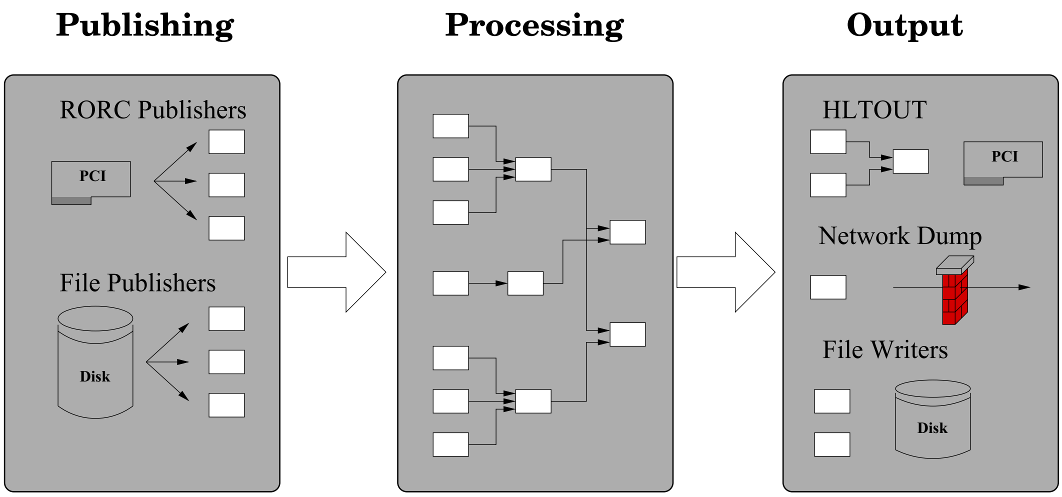


Figure 3.6: The three groups of processes in HLT and variable interconnects. The modular setup motivates the terminology *HLT components*.

A special group of processing components is formed by *HLT analysis components*. Motivated by the necessity of different running environments, an approach has been developed which decouples data analysis from transport. It allows to run the analysis processes in either the off-line environment or in the on-line HLT environment without any change in the code or the need of recompilation. The HLT Analysis Framework is an integral part of the presented work and will be described in detail in section 4. A special component of the HLT data transport framework, the *AliRootWrapperSubscriber* is the link between analysis components and the on-line environment. The subscriber implements a processor and integrates external modules in order to access the analysis algorithm.

3.3.2 Data Input of the HLT on-line System

Figure 3.7 shows the integration of the HLT into the data flow of the ALICE experiment. The raw data are transferred via optical fibers from the detector front-end to the DAQ system. The DDL optical link is used commonly for data readout of all ALICE detectors. The input devices of the DAQ, *DAQ RORCs (D-RORC)*, send an exact copy of the data to HLT before reading data into the Local Data Concentrators.

The data stream is received by the *HLT RORC (H-RORC)*. In total, 454 DDLs are forwarded to HLT, including all relevant detectors. The H-RORC is a Virtex-4 FPGA based PCI-X card ([23]) designed for both (i) receiving and pre-processing of the detector raw data of all ALICE detectors and (ii) transmitting processed events out of the HLT computing farm to the DAQ. The H-RORC therefore implements the interface between the HLT system and

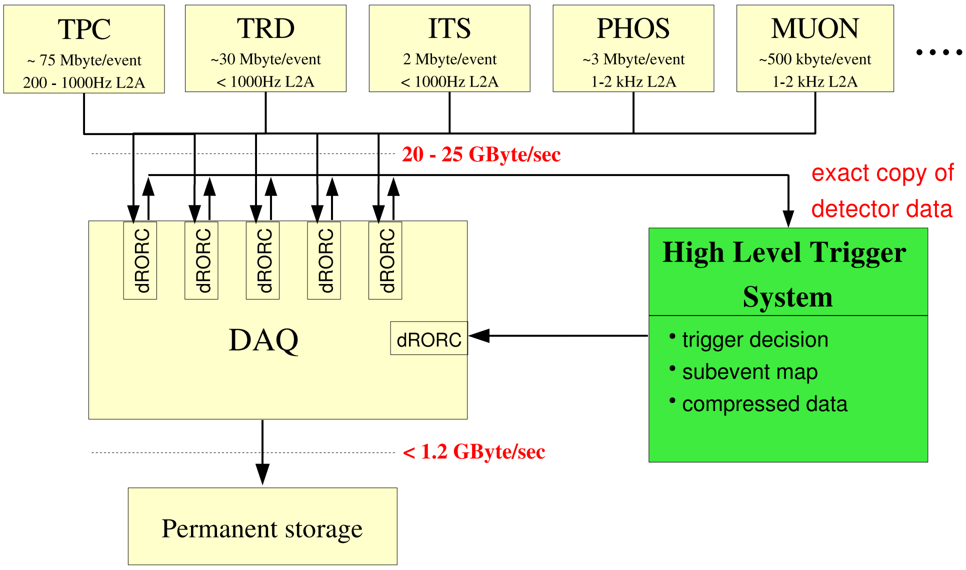


Figure 3.7: High-Level Trigger system in the ALICE data stream. The HLT receives a copy of the detector data and is treated by DAQ as an additional detector. The specified numbers are upper limits for the event size delivered by a sub-detector.

the ALICE data transport links. It is interfaced to the FEP nodes through the internal PCI-X bus.

The trigger decision, reconstructed events, and compressed data are transferred back to the DAQ via the ALICE standard DDL.

3.4 Data Transport

The concept of individual processes allows a high flexibility in the configuration of a processing chain, failure handling, as well as in development. Because of the high overall processing rate, data transport plays an important role in the HLT system. A dedicated data transport framework, the so called PubSub framework, carries out all data transport and communication [24].

3.4.1 Data Transport Framework

In addition to parallelism on event by event basis, the ALICE HLT's approach and its data transport framework allow to split and distribute single events over the cluster nodes. Splitting of the processing reduces the amount of data to be copied dramatically as the first step of the reconstruction can be performed already on the Front-End Processors.

Usually, in this first step of the analysis, clusters and/or space points are reconstructed from the raw data. The resulting data volume is already significantly smaller than the raw data.

The entire communication mechanism is designed for a low processing overhead. Figure 3.8 shows the working principle of the on-line framework. On one node, the data are exchanged via shared memory, a publisher can write the data directly to memory and makes it available for subscribers without any intermediate copying. All processes communicate via *named pipes*[1] and exchange small data description information. The data transport framework takes care of data transport among the nodes transparently. Solutions for shared memory between nodes have been investigated but are not used. More details about the on-line framework can be found in [25] and [26].

Distribution of the event processing is most effective on the FEP nodes. Between HLT computing nodes, copying of data is unavoidable. Here, the Pub-Sub system ensures a high degree of parallelism. As data are treated in a tree-like hierarchy, processing along the individual branches does not interfere with each other and events are collected at a very late stage of the processing when the data volume is already small.

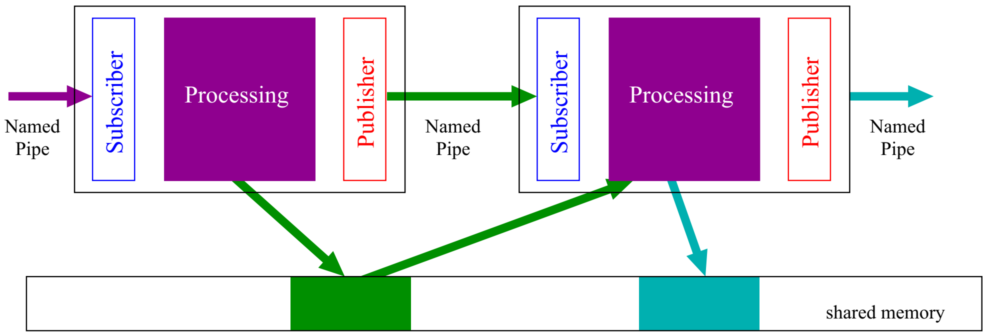


Figure 3.8: Working principle of the HLT data transport framework.

For load balancing, the data stream can be split into several data streams each carrying a smaller data volume, e.g. via a round robin mechanism. Furthermore, the distribution of the analysis leads to a short processing time per event and node. If one node fails, the data loss is much smaller than in a conventional event building and filtering approach where a complete event or at least the processing time is lost. An intelligent scheduler can restart processes on another node and resume event processing in case of a severe failure of a computing node. This subject is not covered by this paper.

Whenever talking about parallel computing, process synchronization plays an important role. E.g. a consumer must not access data until the producer

[1]Named pipes implement a method of inter-process communication on Unix/Unix-like systems

has announced readiness. The data transport framework carries out all process synchronization. This exempts the HLT analysis from any additional synchronization and liberates resources for the main task, which is efficient analysis.

3.4.2 Memory Management

For various reasons, an abstraction layer between physical memory and the application has been introduced in modern computing architectures and operating systems (OS). Individual processes need clearly separated domains in order to prevent one process from read or alter memory of a second process. This abstraction is an obvious requirement both with respect to security and stability. In practice, each process allocates *virtual* memory and the OS handles the mapping between virtual memory pages and physical memory.

Consequently, channels have to be created in order to allow data exchange between HLT processes. As already mentioned, data exchange over shared memory is the implemented approach and allows a minimum of communication overhead as the consumer of data has access to the same physical memory as the producer.

Operating systems provide different solutions for shared memory. In the 1980s, Unix System V first introduced an API[2] for inter-process communication. This has become a standard in Unix-type operating systems and is often referred to be *sysv* shared memory. A process can allocate a shared memory resource which is identified by a *shared memory key*. Any process knowing the key can request access to the shared memory region. The *bigphys* kernel extension [27] implements another approach to provide *big* segments of *phys*ical memory. It allows to reserve at bootup time a certain part of the physical memory for shared memory applications. Both approaches are used in the HLT on-line system. The *bigphys* shared memory extension provides the manner to transport large data blocks as it does not impose any limitation on the size of the shared memory segment, except the size of the physical memory and system requirements ([24]).

The advantage of a common memory segment comes at the cost of an open system. There is no rigid boundary between data segments. Especially the *bigphys* memory approach allows basically any process to access and modify memory in the specified region. The responsibility of memory management has been transferred from the operating system to the application. In HLT, the data transport framework implements the corresponding functionality. However, care has to be taken in the implementation of memory access by

[2] Application Programming Interface

processes. Techniques have been implemented in both data transport and analysis framework to detect potential memory access conflicts and memory access violations.

3.4.3 Data Abstraction

Instead of sending data directly from a producer to a consumer, HLT makes use of data abstraction by pointers. The major bulk of data to be exchanged are written to shared memory by the publisher. The corresponding meta information is stored in a *block descriptor*. The block descriptor holds all relevant information like location, size, data type and specification. Once data sets are in the system, block descriptors are propagated to subscribers in the next level. *RORCPublishers* are the data sources for each analysis chain. This special type of data source components interfaces the hardware input device on the FEP nodes and retrieve information about data blocks from the H-RORC devices. The actual memory transfer though is carried out by the H-RORC itself using DMA[3] transfer, RORCPublishers solely provide the meta information to subscribers.

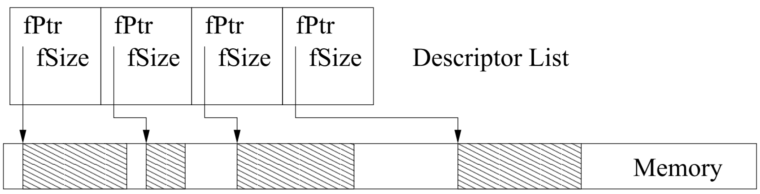


Figure 3.9: Block descriptor references to memory objects.

The format of the exchanged data is sketched in Figure 3.9. The sequence of block descriptors determines the input data, each descriptor points to a certain region in shared memory. Components only exchange the list of block descriptors.

Data Identification

The nature of each HLT data block is described by its data type which is in HLT defined as a combination of *data origin* and *type id*. Both properties are represented by 4- and 8-character arrays respectively in a data structure `AliHLTComponentDataType` as shown in Listing 3.1.

[3]Direct Memory Access

Listing 3.1: HLT data type consists of the *origin* and *type id* members.

```
/** length of the origin member of AliHLTComponentDataType */
const int kAliHLTComponentDataTypefOriginSize=4;

/** length of the type id member of AliHLTComponentDataType */
const int kAliHLTComponentDataTypefIDsize=8;

struct AliHLTComponentDataType
{
  AliHLTUInt32_t fStructSize;
  char fID[kAliHLTComponentDataTypefIDsize];
  char fOrigin[kAliHLTComponentDataTypefOriginSize];
};
```

Data origins follow when ever possible the notation of the sub-detectors in ALICE and are defined by the framework. HLT Data Types can be defined as part of a component library. Since the definitions are used only internally, detector modules are free to define any key describing proprietary detector data. A list of common definitions can be found in appendix B.2.

Data blocks can thus be specified by combinations of the pre-defined or module-defined keys like e.g. {`DDL_RAW :TPC`} and {`ESD_TREE:ANY`}.

Data Specification Scheme

The data block descriptor allows an additional characterization of data blocks by *data specification*. It allows to classify data blocks of identical data type by one 32bit word. The specification can be freely chosen throughout the processing. However for the initial data publishing common rules apply: Each DDL is identified by a unique id, the so called *equipment id*. For all sub-detectors having less than 32 DDL links, the DDL number within the sub-detector is encoded into a bit pattern. The number of the position of the bit which is set corresponds to the number of the DDL, e.g. *0000000000001000* specifies DDL no 3 (started counting from 0). Only TPC and TOF exceed the range of 32 DDLs. Data specification for the TPC is used to specify ranges of DDLs as explained in Sec. 7.3. By means of the specification, components of the same type can determine their place in the analysis and load the correct calibration data sets.

3.4.4 Intrinsic Data Properties

All modern computer architectures organize memory in quantities of bytes. One byte consists of 8 bits, the smallest entity of information. Data variables and data structure in software are of arbitrary size and resolution. Consequently, the representation of data in memory needs to be defined, especially the sequence of bytes referred to be *endianness* and the alignment of structure members in memory. Both represent important properties of the computer architecture the software is compiled for and the compiler used for the generation of the software.

Data alignment applies specifically to data structures combining several variables of varying data types. The compiler places the beginning of a structure member only at certain multiples of memory addresses, so there might be gaps between the members. Consequently, structures can occupy different sizes in different compilation environments. Transferring a piece of memory to a system with different alignment can lead to mismatches.

Endianness is the second important property of data and determines the sequence of bytes how data are stored or sent. There is consensus on the bit-ordering within a byte on almost all modern computer architectures. A single byte value is read exactly the same on all architectures. For values requiring more than one byte, different architectures need to agree on the sequence of bytes stored in memory or file, or sent over network. Endianness becomes an important attribute of data.

In particular three definitions are of importance in modern computing architectures:

(i) Little endian:
The *Least Significant Byte (LSB)* value is at the lowest address and other bytes follow in increasing order of significance.

(ii) Big endian:
This format specification starts with the *Most Significant Byte (MSB)* at the lowest address and other bytes follow in decreasing order of significance.

(iii) Network byte order:
Endianness has in particular importance for all network transmission of data. The big endian format is used when sending data over networks, in this special case it is referred to be network byte order and the MSB is sent first.

3.5 HLT Configurations

The HLT configuration forms an individual group within ALICE HLT and describes setup and interconnections of components independently of the implementation. A configuration describes basic properties of the HLT processing layers

- Type of the component
  The configuration specifies the application to run.
- Command line arguments of the component
  In order to adapt to different needs, components understand certain command line parameters. The exact format of parameters depends on the implementation of the specific component.
- Interconnections of components
  The configuration specifies all parent processes the component needs to subscribe to in order to receive data.
- Reserved output buffer size and number of available buffers
  Output data buffers have to be allocated before the processing loop of the component is invoked. The component has a fixed size of memory available in order to write its output. The number of available buffers has an important impact to pipelined processing.
- Process multiplicity
  For load distribution, multiple instances of the same component can run in parallel. The process multiplicity is part of the configuration.
- Fan-in/Fan-out
  The configuration defines the topology of the chain which includes merging of multiple publishers to one consumer (Fan-in) and distribution of one publisher to multiple consumers (Fan-out)

In the on-line HLT, a configuration is described in XML notation, which stands for *Extensible Markup Language*. In general, a *Markup Language* is an artificial language adding information on how to deal with a certain type of text and content. Being used traditionally in publishing by adding printing and typesetting instructions to the document, such languages have become very important in computing in recent decades. Markup languages are suited to categorize properties in an hierarchical way and are easy to parse in order to build the structure defined by such a document.

The HLT configuration defines the execution sequence of the *TaskManager* applications, which run on each node of the computing cluster and supervise all processes on one node [28]. Since the HLT chain can easily describe several hundreds of components on hundreds of nodes, abstractions of configurations have been introduced and an automated generation of the final configuration is available [29].

3.6 Integration into the ALICE experiment

HLT is integrated into the ALICE experiment by a couple of interfaces. For normal operation, HLT requires not only data exchange with the DAQ but also the availability of calibration data and permanently monitored values from the detector survey. Thus it has to communicate with all other ALICE systems, such as the ECS, DCS, and the off-line system. All interfaces are described in detail in [18].

The ECS has an outstanding role as it controls the experiment. All interactions with the other systems are governed by states and state transitions issued from the ECS. HLT implements a finite state machine as outlined in [30] and Figure 3.10.

Each state and transition correspond to certain actions and run conditions of the different parts of HLT.

(i) The transition from state **INITIALIZED** to **CONFIGURED** entails the PubSub framework compiling all configuration files. At this stage, no components are yet running, only *TaskManagers* on the different nodes.

(ii) The transition from state **CONFIGURED** to **READY** by the *ENGAGE* command implies start and setup of all components. After going to state **READY**, HLT can be started by a short *START* command and processes events when in state **RUNNING**.

3.7 Development Methodology

A software project like data analysis for a physics experiment of the scale as ALICE naturally evolves quickly during data taking as new phenomena are observed and experience with the detector setup is gained. The project has to provide the flexibility for enhancement and correction of functionality at the same time as it has to assure the quality of the analysis. Also small modifications in a distinct part of the project can cause malfunctions in other parts and can have a major impact on the overall performance and operation.

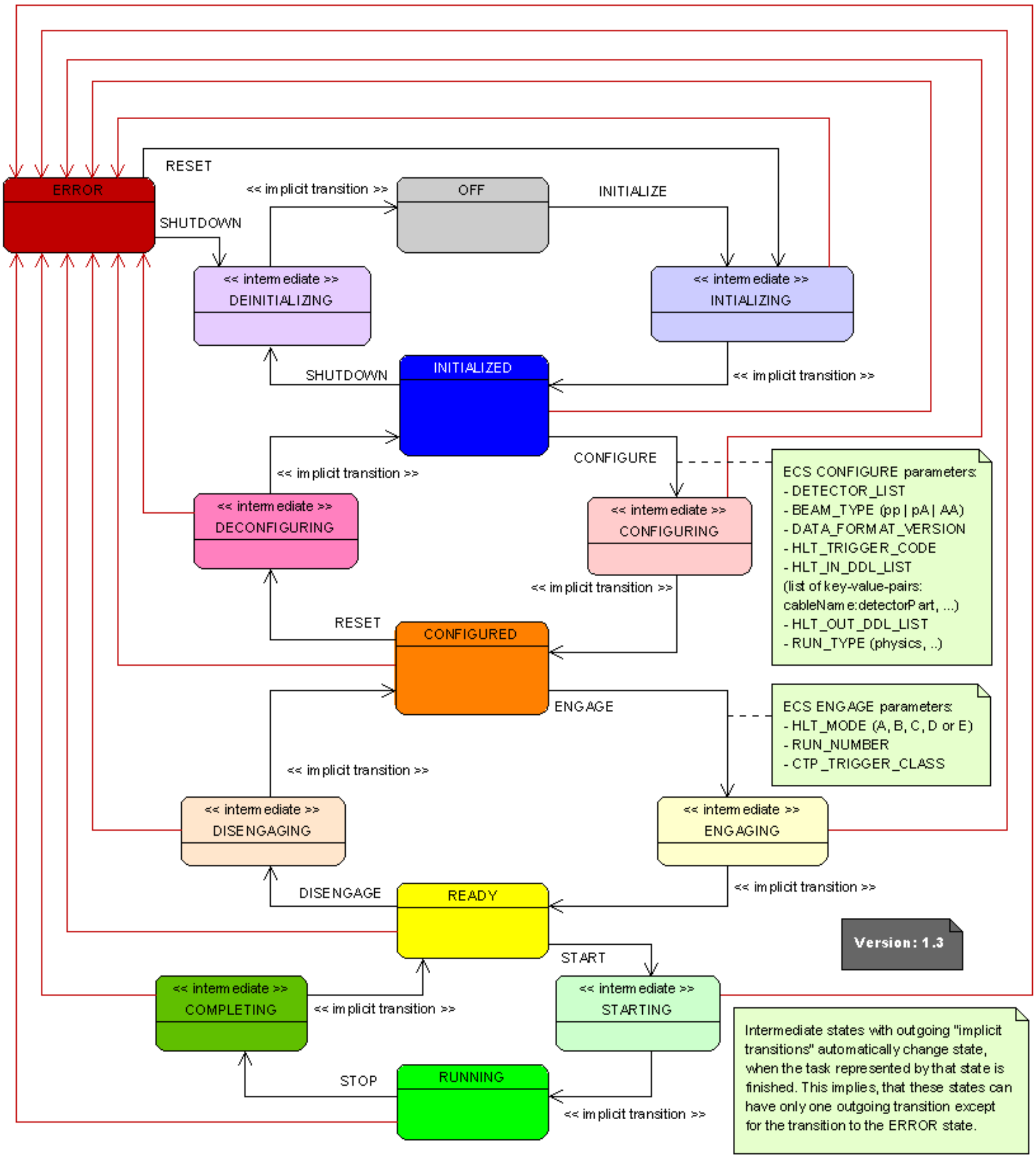


Figure 3.10: State diagram of the HLT system. ECS controls HLT by means of commands and states. From [30].

A software project needs development policies in order to establish

- Development synchronization
- Backward compatibility
- Quality assurance.

Modularization is an appropriate technique which supports those three criteria and is extensively used at different levels of ALICE HLT software. The separation of Data Transport and Data Analysis has already been introduced, more on modularity can be found in section 4.2.1.

As a general paradigm, HLT software builds upon common Unix applications for software development. This applies to all aspects like e.g. the overall build system, source code data base, and source code documentation.

Another paradigm applied for the verification of HLT software is separation of development and verification. A developer can often only account for a biased test since conceptual design, thinking and test are not independent of each other.

This section introduces in particular development methodology designed and applied for the HLT analysis software project.

3.7.1 Software Compatibility

As the software projects evolves, new formats and functionality are needed while other parts appear to be deprecated. Compatibility is referred to be the ability of treating data independently of software version and is a relationship between software modules. As long as the project is depending solely on its own, compatibility issues are likely to be ignored. However, this case is rather theoretical as most of all software projects **(i)** store data in some format, and **(ii)** depend upon third-party modules. The former case directly motivates the wish to read old data with a newer version and treat data of a new format with an old version, though likely limited support for new features. The issue of data formats also applies to the second aspect as data formats define the interoperability.

Especially backward compatibility, the ability of treating data of an older format, is an important aspect for rapidly evolving projects. Since malfunctions can occur at any time in the development process and can be induced indirectly, software needs to be cross-checked with older versions in order to trace down the reason of malfunction. Design paradigms of HLT analysis include preservation of backward compatibility.

Forward compatibility is more difficult to preserve as implementation must take account for changes not known at time of development. HLT aims forward compatibility wherever possible by appropriate design of data formats.

The very strict compatibility requirements imposed on the HLT software include furthermore *binary compatibility*. It allows compatibility without the need of recompilation in contrast to *source compatibility* where a new build of the software package is required but without changes. *Binary compatibility* is a very important outcome of the presented work and will be introduced in detail in section 4.

3.7.2 Development Environment

As an important aspect of overall system stability the development environment must be considered as an integral part. It has to enable the developer of efficiently add and update functionality of the application. At the same time it must ensure:

- Internal consistency:
  as the source code is divided into many files, the build system must ensure that the application is up-to-date and uses only most recent versions of files.
- Portability:
  Software packages need to be compiled and run on various platforms.
- External Dependencies:
  Most of modern software applications rely on third-party modules. This allows efficient re-use of previously developed functionality.
- Code documentation
- Unit and system tests
- Versioning

The make Utility

As early as in the 1970s, software projects already became so large establishing the need for automatic control of internal consistency. Since than the utility *make* is *the* solution on Unix-like operating systems. *make* relies on source file and dependency definitions, so called *Makefile*s and controls the different states of compilation. However, a *Makefile* is often system dependent. Cross-platform support can be implemented, but is a tedious process. Since system dependency is common for all applications, another layer was motivated.

The GNU build system

Unified build systems take account for the requirements of cross-platform development and provide an integrated development environment. The GNU project[4] provides a feature-based build system which allows to automatically generate system-adapted *Makefiles*. The software package is configured prior to compilation. During configuration, the system is probed for available features like compilers, system functionality, and third-party software and the *Makefile* is generated according to the result.

A major feature of the build system is automation of most of the verification and distribution process. As part of this thesis, the GNU build system has been applied with benefit to the HLT analysis source code.

The most recent development in terms of build systems is *cmake*[5] which incorporates many ideas of the GNU build system and supports all operating systems. *cmake* has the potential to replace the GNU build system in the future.

Version Control

HLT entails a large number of individual processes and applications. The clear separation of data transport and data analysis allows simple versioning. All analysis components are developed within the off-line framework. For production runs, only tagged releases are allowed. Those tags follow the releases of the off-line code or can be private HLT tags in the common source code repository.

A network file system on the HLT cluster distributes the current version of the compiled analysis components and makes sure that the same version is used on all nodes. AFS[6] has been chosen for that purpose. Properties of the analysis components like version and running conditions are stored in a data base at the beginning of each run.

[4] The GNU Project is a mass-collaboration project founded in 1983 devoted to the creation of a computer operating system and a collection of applications consisting of entirely free software. http://www.gnu.org

[5] https://www.cmake.org

[6] Andrew File System - specific standard for distributed network file systems

Source Code Database

HLT analysis software is maintained within the ALICE off-line software. Source code is organized in a central database using the tool *Subversion*[7] for version control.

Source Code Documentation

HLT analysis software makes intensive use of C++ classes. The classes provide natural functional modules with a defined interface to be used by callers. Consequently, documentation can be divided into external and internal documentation, the former for the user, the latter only intended for the developer. Class documentation is part of the class definition. As a general concept, well documented source code is the best manual and this concept is also applied to HLT analysis software as a general rule.

Furthermore, common tools are available in Unix/Linux distribution which parse source code and create a structured documentation directly from the code. The tool *doxygen* is widely common and also used for the HLT analysis software. Generation of documentation is embedded into the build process and also publicly available on the web[8].

3.7.3 Unit tests and Functional Verification

The most successful strategy to build a stable and complex system is extensive verification of all parts of the system. Furthermore, an even bigger detective power is achieved by adding automatized verification to the manual one.

Unit tests are small test programs for the verification of a sub-set of the functionality or a software module. The tests are implemented according to the interface specification of a module and run independently of each other and separated from the main application. Unit tests have a remarkable potential to detect accidentally induced malfunctions during the development process and are suited for automatized verification.

Unit tests have been applied to many modules of the HLT analysis framework.

[7] Subversion is an Open Source project designed for version control of software projects, http://subversion.tigris.org

[8] http://web.ift.uib.no/~kjeks/doc/alice-hlt

3.7.4 Component Development Cycle

HLT components undergo a restricted development scheme in order to assure quality and stability of HLT operation. The status of a component in the course of development can be defined as

- *EXPERIMENTAL*
  development and testing is ongoing,

- *TESTED*
  development has been finished; ideally, unit tests for functional groups are added and executed on a regular basis; at least one test case exists to run the component in the off-line environment, or

- *COMMISSIONED*
  the component is running successfully and stable in the on-line environment; a stand-alone test configuration is existing as well as a system test including the component.

The development cycle is illustrated in Figure 3.11. The implementation up to status *TESTED* can be done using the ALICE off-line environment. This approach has been chosen to allow the flexible development of HLT analysis. Only the last step requires knowledge of the on-line system and can be done by *users* of the analysis component instead of the *developer*.

For final production runs, only components of status *COMMISSIONED* are used.

3.7.5 Automated Verification

Verification of HLT analysis has been automated in order to ensure system stability and early detection of malfunctions. Every night a test cycle is executed with the most recent version of the software package. In addition, every change in the source code data base is cross-checked to be compilable. The test cycle includes **(i)** check-out and compilation of the ALICE off-line package, **(ii)** compilation of the HLT analysis package, **(iii)** unit tests of the HLT analysis package, **(iv)**, execution of test macros, and **(v)** distribution of HLT analysis package and documentation.

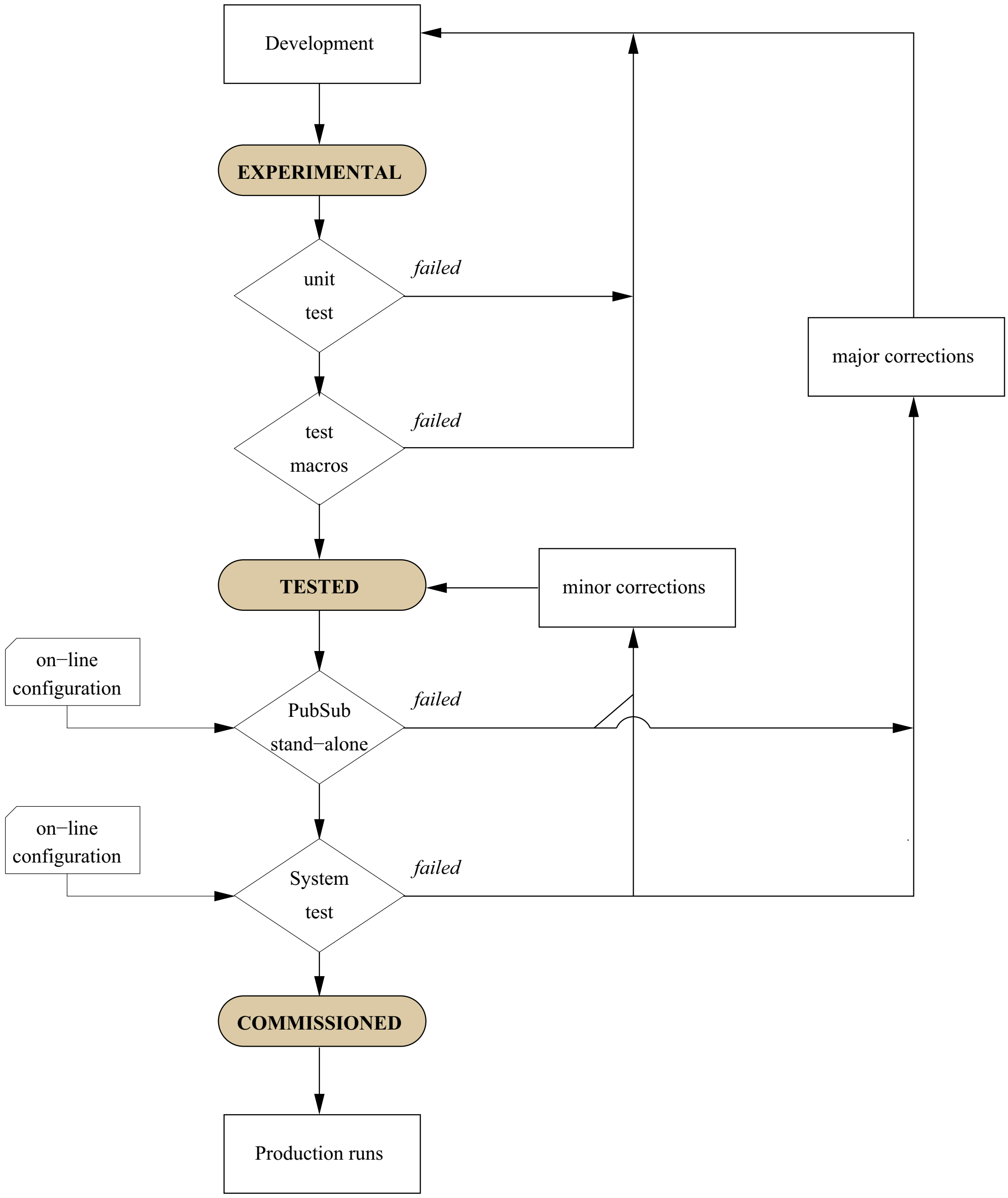


Figure 3.11: Development cycle of HLT components.

3.8 Output of the High-Level Trigger

3.8.1 Trigger

The primary intention of HLT is naturally the generation of trigger information. Triggering implies the analysis of the event with respect to trigger criteria such as particle multiplicities, particle energies, certain decay processes, and distinct particle concentrations in limited regions of the detector setup.

Triggering is mostly a yes/no decision, the event is either considered valuable and accepted or not. Still the result of the on-line reconstruction needs to be stored together with the trigger information in order to determine the efficiency and selectivity of the trigger in later off-line stages of the analysis. In addition, data volume is reduced by selective readout and storing only the relevant part of data. This technique is called **R**egion **O**f **I**nterest readout. ROI criteria can be applied in the first place on the level of DDL inputs and HLT simply provides a list of DDLs to be included into final event building. The format is defined in [31] and section 5.3 elaborates more on that topic.

An analysis component has to generate the DDL readout list if it is designed to contribute to the trigger information. The analysis framework provides common functionality for triggering components.

All readout lists from the running trigger components are treated by the processes on the HLTOUT nodes and formatted together with the HLTOUT payload into the defined DDL format. The HLT-to-DAQ interface ([31] defines a format on top of the DDL protocol [32].

3.8.2 HLT Output Payload

Beside trigger information, HLTOUT format foresees also payload data which DAQ just forwards to storage. As a matter of fact, any output of HLT components can be directed to the HLTOUT and is included in the payload.

Since HLT is an on-line system, the payload contains information about fully reconstructed events as part of the normal raw data stream. Beside this information which is essential for the evaluation of the trigger, data blocks of varying nature are stored depending on the configuration of the analysis chain. Strategies on data treatment and extraction have been implemented in the HLT Analysis framework and the AliRoot binding functionality and will be described in section 5.

Compressed Raw Data

A special use case of HLT is the reduction of the data volume by applying appropriate compression techniques. The original data are encoded according to a compression algorithm and stored as part of the HLTOUT payload. Data compression has been extensively studied for the Time Projection Chamber as the biggest contributor to the data volume in ALICE [33, 34].

In general there are two data compression paradigms. Loss-less compression techniques allow to restore the original data from the compressed data set. Higher compression ratios can be achieved by applying models especially suited for the nature of data and omitting irrelevant information. The original data cannot be restored from the compressed data.

In both cases the off-line reconstruction will treat the compressed instead of the original data and data flow schemes need to be designed. It is appropriate to completely separate reconstruction algorithm and data input in order to handle different types of data. Section 5.6 introduces the developed approach which allows to transparently run reconstruction algorithms on either the original or compressed data set.

3.8.3 Calibration Data and Off-line Storage

Besides the normal data output, HLT implements also an interface to the OCDB which is the central storage for all run related information and conditions needed later for reconstruction. A dedicated synchronization mechanism is implemented by the ALICE off-line group in order to migrate data after the end of data taking to the data base ([35, 36]). E.g. calibration data are stored in the OCDB as well as the magnetic field and temperature measurements.

The OCDB is not a data base in the literal sense of the word but a format definition and a file catalog structure. The OCDB access framework also provides a storage abstraction. Data can be stored in local file catalog or a GRID folder. The latter is the final target as event reconstruction will be carried out on the GRID making availability of the calibration data and run related information a crucial prerequisite.

The HLT interface to the OCDB implements both retrieval of important calibration data and run conditions as well as storage of on-line generated calibration data. Furthermore, run conditions of the HLT are stored. The communication between HLT system and OCDB is presented in [18].

3.9 Detector Monitoring

The ALICE High-Level Trigger provides a powerful data processing facility working at the full input rate. This real-time capability makes it an excellent tool for on-line monitoring.

3.9.1 Monitoring strategies

The main purpose of detector monitoring is the survey of data produced during a run. It provides a fast estimation about the informational content, data quality, and detector operation. Different running modes are possible which require different monitoring strategies, e.g. normal data taking will last for a couple of hours. A quick response is desired to evaluate the quality of the run and in order to encounter problems and react. In this case, detector monitoring must not disturb the normal data taking and runs with a lower priority. On the other hand, detector development requires special runs with focus on detector properties and calibration. Those runs are a stand-alone monitoring applications.

Monitoring is accomplished outside the HLT processing chain on an event by event basis. In that case, data are shipped from the HLT to a monitoring application and processed and displayed there. The second approach allows accumulative data processing inside the HLT chain. Histograms are filled and results for many events are recorded internally. The results are provided on demand or at a reduced rate. The second approach allows on-line monitoring of all events.

In either approach it is necessary to access data at an arbitrary point in the HLT processing chain.

3.9.2 HLT On-line Monitoring Environment

For the purpose of transparent and system independent transport of block descriptions, *HLT On-line Monitoring Environment including ROOT (HOMER)* framework has been developed [37]. It provides a simple means to ship out data from the HLT system to a subscriber application. The framework consists of a core library providing the data formatting functionality and satellite applications handling the data to be shipped.

The HOMER format consists of **(i)** a descriptor section containing sizes, types and meta information in a system independent format, and **(ii)** a payload section. Both are encoded into one single contiguous data buffer or stream (Figure 3.12). Each entry contains also an offset to the actual payload described by the corresponding block descriptor. The monitoring framework handles system architecture dependence only for data block descriptors. The actual payload is not known to the monitoring framework and such dependencies must be handled by the producer and consumer application of a certain data block. The framework however provides system information, i.e. endianness and data structure alignment of the producer system.

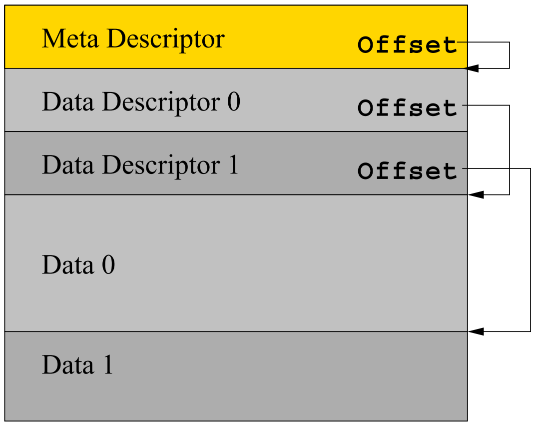


Figure 3.12: The HLT monitoring data format HOMER consists of a meta descriptor, followed by data descriptors. The actual data blocks are appended at the end, and offset specifiers in the data descriptors point to the start offset.

Data Access in the On-line HLT Chain

The so called *HOMERWriter* encodes all block descriptors into the system independent HOMER format. Two flavors of the monitoring framework have been implemented; a sink component providing the data via shared memory, and another one working with a TCP/IP port over network. The exchange method is transparent for the user. Figure 3.13 illustrates the data flow of the HOMER framework for the TCP approach. A normal PubSub sink component can subscribe to the output blocks of arbitrary components.

The TCP approach has been used extensively. As this approach allows access via the network from another machine, it disentangles the monitoring backend from the actual data processing. A reader library is provided by the analysis framework to access data blocks. It can be used directly from the interactive ROOT prompt.

Subscription to Data

The counterpart of the HOMERWriter is the *HOMERReader*. It can be used directly from the ROOT prompt to fetch data from an HLT on-line chain.

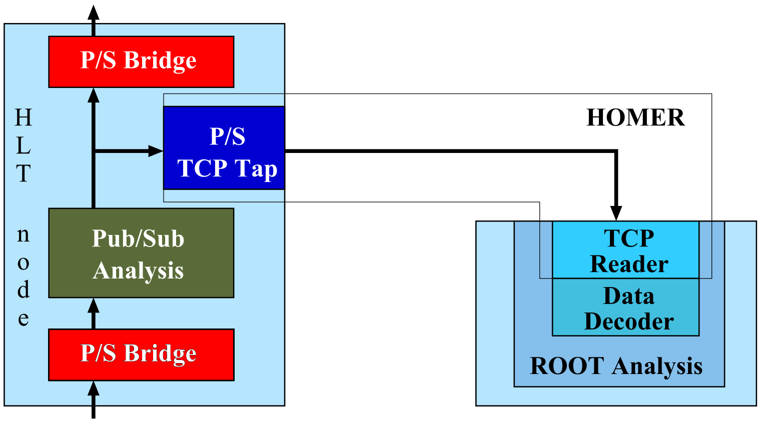


Figure 3.13: Data flow of the HLT monitoring interface. The HLT cluster node to the left, a separated monitoring node to the right. The publish/-subscribe sink component (P/S TCP Tap) subscribes to arbitrary data in the stream and provides them over network. The depicted P/S Bridge components are part of the data transport framework and responsible for data exchange between computing nodes.

A HOMERReader can handle different types of input. The class provides different constructors for the purpose of initialization. Once the reader is created, access to data blocks is transparent and independent of type of input. Data transport is done on event basis. As soon as the event buffer of the *HOMER* interface is empty, the next event is prepared for transmission. On the receiver side, the event is fetched by means of the `ReadNextEvent` function.

The HLT block data type and origin as introduced in chapter 3.4.3 are only encoded into 64 and 32 bit words respectively. The generic design of the *HOMER* interface makes it stand-alone and without any library dependency. The 8 and 4 byte character arrays of data type and origin are stored in reverse order in the HOMER block descriptor.

3.9.3 Monitoring Back-end

For the sake of simplicity it was decided to use the off-line monitoring framework *AliEve* as a back-end also for on-line monitoring applications. This reduces redundant development and allows users to work with a coherent set of tools. AliEve is based on ROOT *Eve* and part of AliRoot [38, 39]. It combines an event display including 3D visualization with tools for investigation and browsing of ROOT data structures and histograms.

The actual treatment of data is responsibility of the module library. Macros are used to bind the processing functions into AliEve.

4. HLT Analysis Framework

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.
> *From R. Gauthier, Designing Systems Programs [40]*

This very descriptive statement from the early years of software development in 1970 has not suffered any loss of its philosophical depth concerning software projects. It describes very shortly all issues which are also addressed by the design of the HLT project. The previous section has outlined the complex structure of the ALICE HLT. The HLT analysis framework is an integral element of the ALICE High-Level Trigger and has been accomplished as part of this thesis. Before introducing it in detail, general design considerations are presented which have been the motivation for its definition and design. The general requirements for a system can be summarized as follows:

- Efficient use of development resources.
  The system naturally needs to address different problems both in technical and physics aspects. Reuse of project internal source code is suggested as well as the utilization of third-party implementations.

- Efficient implementation of the application.
  HLT aims to provide a high-speed on-line computing cluster. The imposed performance requirements make optimization a fundamental part of the design.

- Well documented source code.
  Functional units and the interplay of units must be documented consistently.

- Long term maintainability.
  The life time of ALICE makes maintainability an important aspect

- Extensibility of the system.
  As new functionality is desired and new requirements are imposed by the evolution of the experiment, the system must be easy extensible without potentially harming other parts of the system. Also, developers of various skills must be enabled to add new modules in an easy and intuitive way.

- Quality assurance.
  The correct and smooth operation must be assured for the whole life cycle of the system. Test cycles must be applied on a regular basis.

Some more requirements are specific to HLT and already motivated by the issues above:

- Separation of Data Transport Framework and HLT Analysis.

- Uniform HLT Analysis for different environments.

- Low threshold for contributions from all users of the ALICE community.

The tangling problem is the obvious dependency and even contradiction of those requirements. An easy understandable implementation is often inefficient. In contrast to that, an efficiency oriented implementation of a complex application naturally obscures the functional flow and maintainability.

In the evolution of programming techniques there have been different solutions addressing this issue. The traditional *Procedural Programming* straight forwardly encapsulates functional units into procedures and organizes them in a flow graph. Approaching more complex applications, Object Oriented Programming (OOP) has been considered to be the better approach because the object abstraction fits the underlying problem much better. The OOP approach works well as long as the issues to be implemented do not cross-cut the systems basic object decomposition. Optimized memory access and the efficient utilization of temporary data often imply constraints on the appropriateness of the object oriented decomposition.

However, Object Oriented Programming has many advantages in terms of modularization of development and maintainability. Especially inexperienced developers can benefit from OOP concepts which hide the actual implementation from the user and assign functional units to objects.

4.1 Interface Methodology

In any case decomposition of a system into smaller functional units - software modules - is an important part of the solution. Decomposition of complex software systems has been investigated already in the early 1970s. The basic principles apply today even more as the complexity of systems has increased remarkably.

A system is decomposed by design decisions whereas the decisions have to be taken before the work on individual modules can start. It is of great importance that the decisions include all system levels. As a result of the decomposition, self-consistent software modules hide information by encapsulating it into a unity or other construct. This technique is in software engineering often referred to be *encapsulation* or *information hiding*. Interfaces between modules allow well defined communication.

An interface is usually formed by a set of functions accessible from the outside of the module. Following the concept of information hiding, a module possesses internal private and external public data and function members. Object oriented languages like C++ support this classification by defining members *public* or *private/protected*.

Depending on the application, modules can be linked together at compile/link time. Execution of such a program requires availability of all modules. Dynamic loading of modules at run time allows an even more abstract implementation of modularization. The application does not need the specific module for start up, instead the module is loaded on demand. All HLT analysis components are based on this concept.

Finally, the domain of the interface is of great interest as this allows to separate application and module system wise. ALICE HLT implements an abstract C interface between data transport framework and analysis framework in order to allow the maximum of flexibility. The abstract interface is introduced in section 4.4.3.

The benefits and properties of a modularization can be summarized as follows:

- Encapsulation hides the implementation details of a class from the external world. In this way the implementation can easily be modified without affecting any other modules.

- Encapsulation can avoid mistakes by consequently hiding internal data members, referred to be *data hiding*. Access to data members is encapsulated in a method and validation checks can be added.

- Encapsulation reduces debugging time in large applications because the outer world can only access a well defined interface. Hence the state of variables can only be altered internally and better controlled in debugging.
- Encapsulation leads to better documentation and long term maintainability.
- Encapsulation allows one module to be written with little knowledge of the code in another module
- Encapsulation allows modules to be reassembled and replaced without reassembly of the whole system.

4.2 HLT Modules and Libraries

4.2.1 General Concepts of Modularity

An application which is designed without modularity is referred to be monolithic. This is often a fast approach for a small application of distinct purpose. However, the program grows as new aspects come into play and new features are required. It is often difficult to determine the turning point when a monolithic application is no longer an appropriate solution.

Modularity is desirable, as it supports reuse of parts of the application logic and also facilitates maintenance by allowing repair or upgrade of parts of the application without requiring complete replacement. Another important aspect is testing and quality assurance. Modular applications are easier to test as they allow dedicated checks of the functionality of parts of the project. The third aspect addresses the development in a multi-user environment. Interference between the various developers involved in the project can widely be excluded by an appropriate modular concept.

Different approaches are used to various extents in order to achieve modularity. The main aspect of reuse and repair of parts of the project is addressed by code based modularity. However, development tools are required to perform these maintenance functions. The application may need to be recompiled and the developer needs all the development tools and the complete source code. Object based modularity often allows a smarter solution as it provides the application as a collection of separate executable files. In this approach, the various parts can be independently maintained and replaced without redeploying the entire application. This approach is widely used in most of the computer architectures like e.g. *shared object* files on Sun/UNIX or *dll* files on other architectures. Some object messaging capabilities allow object based applications to be distributed across multiple computers. Service oriented architectures use specific communication standard/protocols to communicate between modules.

Traditional software development has focused on decomposing systems into units of primary functionality, while recognizing that there are other issues of concern that do not fit well into the primary decomposition. The traditional development process leaves it to the programmers to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate. Programmers need to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time. These concerns span across the primary functional units within the application, and often results in serious problems faced during the application development and maintenance. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve - a system maintainer must find and correctly update a variety of situations.

Aspect Oriented Programming (AOP) as presented in [41] is in contrast to traditional Procedural Oriented Programming a different approach addressing the mentioned issues. However, aspect oriented programming languages are still less common to be used than OOP languages. The community of nuclear and particle physics is furthermore influenced by the ROOT analysis platform, which entails object oriented strategies at all levels.

Aspects in the Design of HLT

Aspects in software engineering are defined as cross-cutting concerns, i.e. concerns which are not reflected by the primary separation according to modularization decisions.

Several studies have been carried out in order to develop a modular concept for the HLT analysis framework. The deployed concept of components and separate processes ensures a high modularity. The development of the various components is to first extent independent from each other. However, all analysis components are based on the same processing strategy and suggest a common interface and module. Components can benefit from common functionality if an object based modularity is deployed.

One major characteristic of AOP is to relieve the developer from the responsibility of implementing all issues of concern. This idea has been considered into the HLT analysis framework, though primarily being of object oriented nature.

4.2.2 Framework Organization

Modularization of the HLT analysis framework is deployed at different levels. Following the concept of object based modularity, functional units and the analysis component implementations are realized in UNIX shared object files, often referred to as libraries.

An important aspect in library design is imposed by library dependencies. Since the functionality is implemented in separate executable files, there are cross-dependencies between libraries. If `liba` implements an application calling a function implemented in `libb`, the latter library needs to be loaded in order to resolve this dependency. Usually one is confronted with the following dependency problems:

- long chains of dependencies
  An application depends on `lib1`, which depends on `lib2`, ..., which depends on `libn`. In order to avoid manual resolving of dependencies, the library `lib1` should be linked at compile time to all others it depends on and so on. Package managers can help to install the necessary libraries from external repositories.
- conflicting dependencies
  It is important to allow simultaneous installation of different library versions. Otherwise two applications depending on different versions of the same library cannot run at the same time. A rigid library version scheme is deployed on most UNIX/Linux systems and also for ALICE HLT.
- circular dependencies
  If a library `lib1` depends on `lib2` depending on `lib3` containing a dependency back to `lib1` one speaks of a circular dependency. Circular dependencies are often a sign for bad code design.

The HLT analysis framework aims to follow a clear dependency tree in order to avoid the quoted problems. The organization is outlined in Figure 4.1. The libraries fall in 3 categories, (i) Framework Layer, (ii) Service Layer, and (iii) Module Layer.

In the **Framework Layer**, a base library, `libHLTbase`, provides all functionality needed for HLT analysis. Apart from system libraries, it only depends on ROOT. This is a clear design goal which reduces dependencies of `libHLTbase` to a minimum and allows to implement common interfaces and functionality independent of AliRoot. The base library defines all interfaces of processing plug-ins of the different stages. It contains furthermore a simple stand-alone

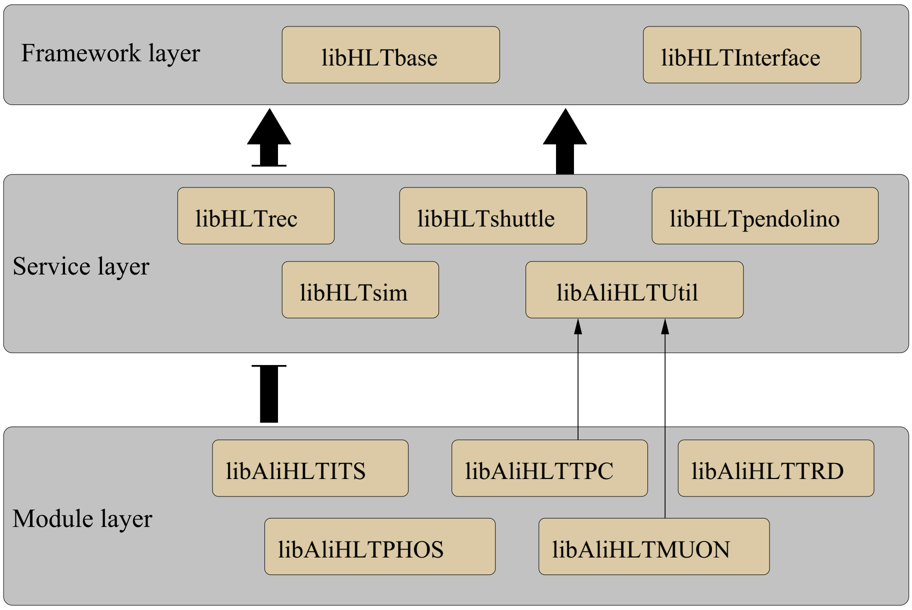


Figure 4.1: Modular organization of the HLT analysis framework. External dependencies to system libraries, ROOT, and AliRoot have been omitted for the sake of simplicity.

running environment for HLT chains, `AliHLTSystem`, which is introduced in section 4.5.2. The second library of the Framework Layer, `libHLTinterface`, implements an abstract interface intended to allow utilization of HLT components in external applications.Details can be found in section 4.4.3.

The **Service Layer** contains a couple of libraries with AliRoot binding functionality, `libHLTsim` and `libHLTrec` for simulation and reconstruction in the off-line environment respectively. Other supporting functionality is related to the interface to OCDB and DCS. A utility toolbox, `libAliHLTUtil`, implements both supporting classes and utility components. Libraries of the Service Layer depend upon AliRoot and need the corresponding libraries.

In the category of the **Module Layer** fall all libraries implementing HLT analysis. All libraries depend on the Framework Layer and occasionally on `libAliHLTUtil`. As the libraries of the module layer implement HLT analysis components, they are often referred to be component libraries.

4.2.3 Features and Functional Units of an HLT Module

HLT module libraries for detectors or distinct purposes can be loaded dynamically into the framework. Each module implements all functionality related to the same issue, e.g. components to be run on-line on the HLT cluster, and the treatment of the produced data in other stages of the processing chain.

For the time being, there are five groups of different plug-ins realized by a module:

- Analysis components to be run in the on-line environment,
- *PredictionProcessors* allow components to predict the value of on-line data points provided by DCS,
- *PreProcessors* prepare certain data output blocks of the component for storage in OCDB,
- HLTOUT handlers supposed to treat data of the components during off-line processing of HLT output, and
- monitoring plug-ins preparing data for visualization.

Following the basic requirements for the HLT system as presented in the beginning of this chapter, the user must be allowed to implement a new unit without the need of adding new functionality into other units. The plug-in concept is realized by interface classes defined in the `libHLTbase` and concrete implementations in the module. The resulting encapsulated functionality can be integrated dynamically into the system by simply loading the library.

4.2.4 Module Agents

Still, the new functionality needs to be announced to the system. For the purpose of fully dynamic registration of module functionality at run time, each module library implements a so called *agent*. The agent defines the properties of the library which are explained in detail below.

The automatic registration of module agents in the system is the fundamental feature which ensures the on-demand modularity of the HLT Analysis framework.

Module agents inherit from the base class `AliHLTModuleAgent` and are intended to be singletons in the running environment, meaning there is only one instance of each module agent at any time. In order to make the agent available to the system, one global object of the particular agent implementation has to be specified in the source code. Since all global variables are allocated when the library is loaded, this trick ensures the automatic creation of the object. Functionality of the base class carries out registration and integration into the HLT system automatically. Once the agent is available in the list, the system will query its features on demand. New units can be added without intervention in other functional units.

Component Registration

Components are the worker plug-ins for HLT analysis and are created dynamically according to the configuration. As a prerequisite, components must be known to the system. The analysis framework implements a registration scheme which is the backbone for the realization of modularity. Details will be presented in section 4.4. The agent needs to implement the function

```
int RegisterComponents(AliHLTComponentHandler* pHandler) const;
```

which is invoked by the system after the library has been loaded.

HLT Configurations

Following the described processing methodology, the HLT analysis hierarchy is described by configurations. In the on-line system, configurations are fully separated from the actual implementation of both data transport and analysis.

The off-line environment as the counterpart also requires mechanisms to describe an HLT chain and run components in an analysis hierarchy, both for the sake of simulation and utilization of AliRoot as a development environment. The AliRoot binding functionality of the HLT analysis framework provides simple notations to specify hierarchies of component processing (see section 4.5.2). The module agent is allowed to specify HLT configurations supposed to run as default configurations in the different steps of AliRoot by means of the functions

```
int CreateConfigurations(AliHLTConfigurationHandler* handler,
                         AliRawReader* rawReader,
                         AliRunLoader* runloader) const;
const char* GetReconstructionChains(AliRawReader* rawReader,
                                    AliRunLoader* runloader) const;
```

The full integration of HLT into the ALICE software framework is described in section 4.5.

PreProcessor

The propagation of data to OCDB, implies the existence of so called *PreProcessors* which are executed in the course of data and statistics collection after the end of a run. An asynchronous link between data producing facilities in the detector and the OCDB is provided by the so called *Shuttle* [36].

The Shuttle is a stand-alone application started immediately after the run has been stopped. It fetches data designated for storage in OCDB. Each detector implements one PreProcessor which treats corresponding data blocks and prepares them for storage. As HLT adds another layer of modularity, an additional abstraction is required. The modularization of HLT PreProcessors for different types of data must follow the diversification of analysis components as producers of data. That is, a component library containing a producer component must also contain the corresponding PreProcessor. The module agent provides worker plug-ins for the overall *HLTPreProcessor* which is included into the off-line Shuttle framework.

PredictionProcessor

The HLT on-line calibration framework [18] also implements the interface to DCS on-line data points. E.g. the temperature measurements of the TPC are relevant for on-line TPC event reconstruction as it affects the drift velocity. For on-line analysis, measurements are always in the past and the current value must be estimated which is the task of the PredictionProcessor. A similar scheme applies for off-line reconstruction where all measurements are available but values must be interpolated. The HLT PredictionProcessor framework is described in [42], the module agent provides appropriate binding functionality.

Handling of HLT output

The output of the HLT is processed as part of the normal AliRoot event reconstruction. The analysis framework has the capability to handle common data blocks in the HLT output payload (HLTOUT payload). On the contrary, proprietary data produced by a component/module during the on-line analysis is often understood only by the module itself. A modular plug-in mechanism allows to organize the treatment of HLT payload in a flexible way. In order to bind a module into the standard treatment of HLTOUT payload, its module agent needs to specify the type of data it can process and methods to actually do the processing. The treatment of HLTOUT payload is described in detail in section 5.3

4.3 HLT Analysis Component Interface

This sections introduces the `AliHLTComponent` base class, which provides the common interface for HLT analysis components. Main objective of the interface is a unified utilization from the on-line and off-line systems. Both have different requirements and constitute two completely different running environments for the HLT event reconstruction and physics analysis. In order to study the impact of high-level trigger information on the off-line reconstruction performance, HLT algorithms must be part of the off-line simulation process, as well as evaluated and compared to the off-line algorithms. Furthermore, a coherent development environment is highly desirable in order to simplify the work for developers.

As an important outcome of the presented work an approach has been developed which allows to run the analysis components in either the off-line environment AliRoot or in the on-line HLT environment without any change in the source code. The interface ensures also binary compatibility, meaning that the same compiled library can be used from both frameworks. The benefits can be summarized as

- A high degree of flexibility.
  Development of analysis components is independent from the data transport framework and thus easier to handle.
- Quality of the analysis.
  Analysis components can be run within the off-line analysis without changes. The results are directly comparable.
- Clear development framework and policies.
  Developers of physics analysis components can use the AliRoot environment they are familiar with.
- Reproducibility of trigger decisions
  The analysis components are literally part of the off-line framework, only tagged releases will be used in on-line production runs.

The component interface makes use of function overloading supported by C++ classes. A couple of member functions of the base class are defined *virtual*, indicating the compiler that those functions can be overloaded by child classes. The analysis framework always uses references to objects of type `AliHLTComponent` for each instantiated object of component implementations. If an overloaded function is present in the child class, it is invoked instead of the base class method. The ability to treat derived class members just like their parent class' members in software engineering is referred to be

Polymorphism. The concept of class inheritance and function overloading in OOP is fundamental for the understanding of the component interface and the reader is directed to literature.

The actual analysis code is implemented in one or more distinct class(es). The component implements the interface and calls the analysis code for data processing. An important aspect of this approach is the availability of any part of the off-line code for on-line components. The other way around a step of the off-line reconstruction can be run on-line by simply adding a wrapper component which implements the interface.

An HLT analysis component can be in one of five processing states depicted in Figure 4.2. The bare component which has been created using the C++ constructor must be initialized according to some configuration. Different schemes apply for this stage depending on the availability of component arguments. The event processing loop as the most important state for the analysis is steered from an external application on an event by event bases. Three entry points are available provided by interface functions of the base class:

- `DoEvent` function: called to process one event.

- `Reconfigure` function: called when re-configuration has been triggered by the framework during the run.

- `ReadPreprocessorValues` function: indicates the availability of updated data from DCS.

4.3.1 Characterization of HLT Analysis Components

Each component is uniquely identified by the *ComponentId*, a character string of arbitrary length and content. It is used by the *ComponentHandler* later introduced in section 4.4 to register the component and to create instances on demand. For practical reasons, a descriptive name is used, like e.g. *TPCClusterFinder*. The data processing carried out by the component is characterized by the data types of input and output blocks and the amount of data going to be produced with respect to the input data volume. A component is furthermore classified by the amount of memory it uses for data processing and the overall processing time.

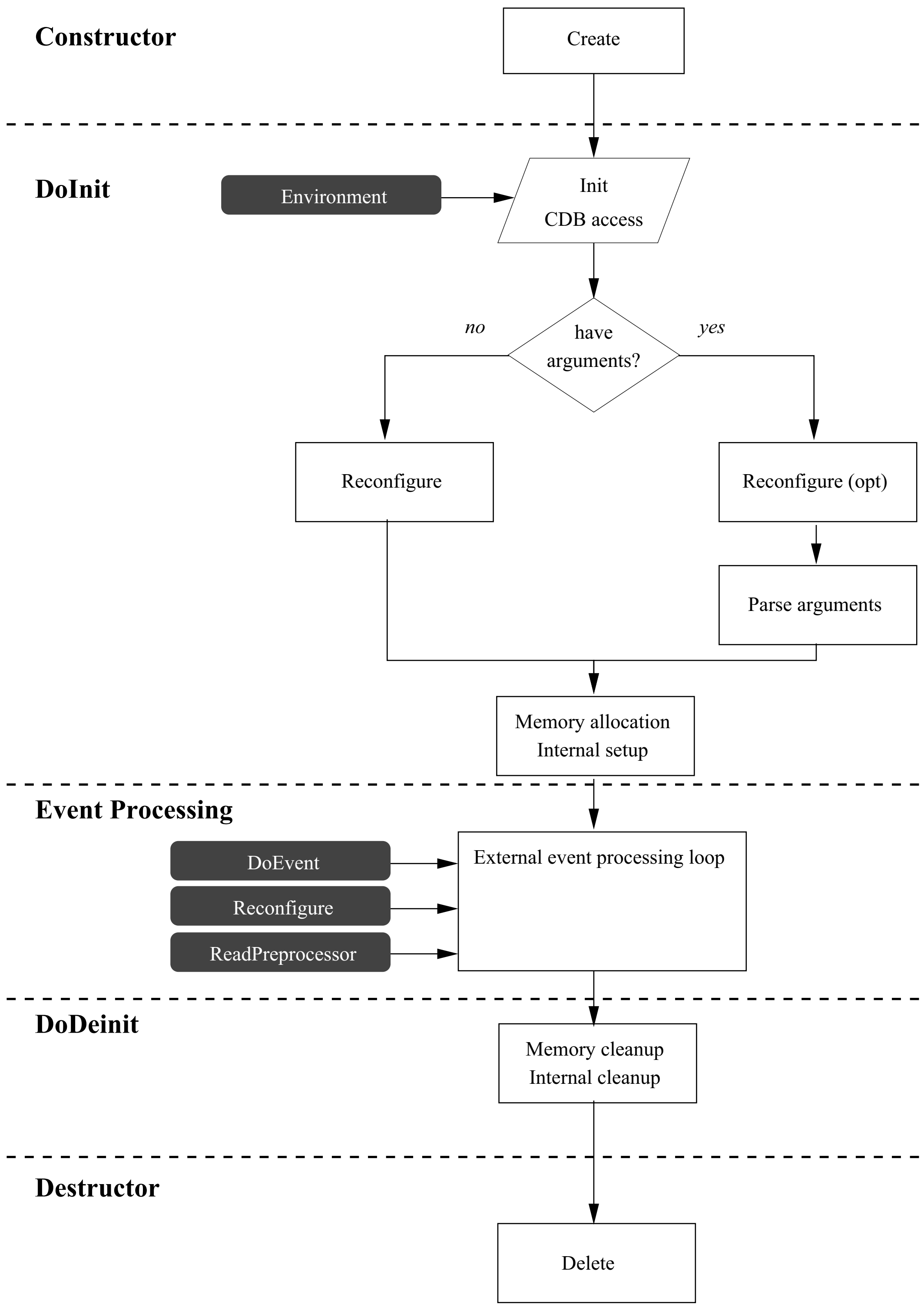


Figure 4.2: The HLT component workflow includes five stages. The processing loop has a central role and is preceded by appropriate initialization.

Component ID The function `GetComponentID` is mandatory and required to retrieve the components identification.

```
virtual const char* GetComponentID();
```

Component Data Types HLT data types consisting of type id and origin (see 3.4.3) are used to describe the exchanged data blocks in both HLT data transport framework and analysis. By means of HLT data types the component can identify data blocks it is able to process. It has to announce the data types it understands and produces to the frameworks by implementing the mandatory interface methods `GetInputDataTypes` and `GetOutputDataType`. The optional method `GetOutputDataTypes` can be used in case of multiple output data types.

```
virtual void GetInputDataTypes(AliHLTComponentDataTypeList&);
virtual AliHLTComponentDataType GetOutputDataType();
virtual int GetOutputDataTypes(AliHLTComponentDataTypeList&);
```

Data Size The shared memory approach used for data exchange in HLT requires providing the output memory prior to data processing. Thus, the framework roughly needs to know how much memory it has to provide for the component output data. An upper estimation must be provided by the `GetOutputDataSize` function. A constant offset and a multiplier per block can be set.

```
virtual void GetOutputDataSize(unsigned long& constOffset,
                               double& inputMultiplier );
```

Component Creation The function `Spawn` needs to be implemented in order to allow the *ComponentHandler* to create a new instance. As seen from the function definition, the return type of the function is a pointer of the general type `AliHLTComponent` even though the created object inside the function is an instance of a specific child. The child class inherits all functionality of the base class and only these methods are available through the interface.

```
virtual AliHLTComponent* Spawn();
```

4.3.2 Running Environment

In order to adapt to different environments, i.e. on-line or off-line, components get an environment structure with function pointers as shown in listing 4.1. The reason for a classification of the running environment is imposed by the possibly different execution domains of application and component.

Some communication between the component and the application is realized by callback functions. For instance, the treatment of logging messages is very important for the operation of a system and each application might implement its own handling. This is in particular important for the on-line setup of the HLT. A multi-process system on a multi-machine architecture requires a central collection of logging messages. Every component sends logging messages through a defined logging mechanism rather than just printing to the standard output channels on screen. AliRoot also provides its own logging and filtering functionality.

Listing 4.1: Definition of the running environment for HLT components

```
struct AliHLTAnalysisEnvironment
{
  /** size of the structure */
  AliHLTUInt32_t fStructSize;

  /** the component parameter given by the framework on creation */
  void* fParam;
  /** allocated memory */
  void* (*fAllocMemoryFunc)( void* param, unsigned long size );
  /** allocate an EventDoneData structure. */
  int (*fGetEventDoneDataFunc)( void* param, AliHLTEventID_t eventID,
                                unsigned long size,
                                AliHLTComponentEventDoneData** edd );
  /** logging callback */
  int (*AliHLTfctLogging)( void* param,
                           AliHLTComponentLogSeverity severity,
                           const char* origin,
                           const char* keyword,
                           const char* message);
};
```

The `AliHLTComponent` base class provides member functions for those environment dependent functions. The member functions are used directly by the component implementation and are re-mapped to the corresponding functions. For the component, the whole process of adaption to the running environment is carried out transparently. Figure 4.3 illustrates the utilization of the environment definition and a call-back function.

4.3.3 Initialization and Cleanup

The handlers defined by listing 4.2 can be implemented for the purpose of component initialization and cleanup. During invocation of `DoInit`, the component arguments can be scanned and the components behavior initialized according to the provided component arguments. Furthermore, all internal structures have to be created in `DoInit`. This takes account of the fact that

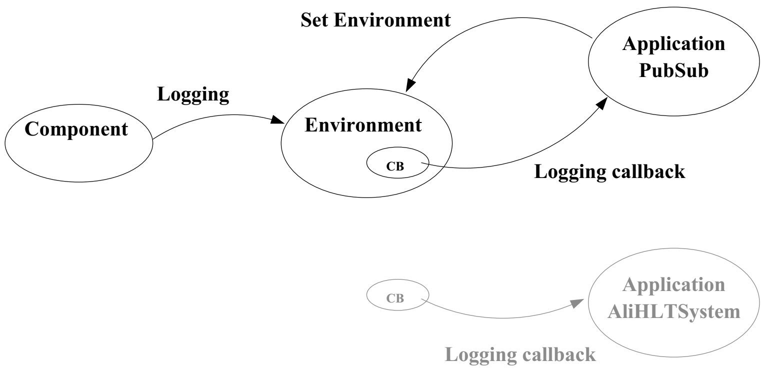


Figure 4.3: Redirection of Logging messages via the running environment. The host application initializes the call-back function (CB) as part of the environment. Calls to the logging method are redirected by the call-back to the application. A second application initializes its own environment and uses a different redirection.

the initialization of the component might take longer and should not be part of the event processing. In the on-line HLT controlled by ECS, component creation and initialization takes place during the ENGAGE sequence in the HLT state diagram Figure 3.10. A relatively short START sequence then starts the run and data processing.

Listing 4.2: Optional initialization and cleanup functions

```
virtual int DoInit( int argc, const char** argv );
virtual int DoDeinit();
```

Command line arguments have been introduced to component initialization in order to adapt at run time to different conditions. The arguments are implemented by the component and specified in the chain configuration. Scanning of component arguments is responsibility of the component. Arguments are provided as an array of strings. By choosing a similar approach as for the `main()` function of every application, the argument list is simple to parse and easily extensible. E.g. the *FileWriter* component as introduced later understands the arguments *"-datafile myfile -directory /a_directory"*, where arguments start with the '-' character followed by parameters for the specific argument.

During initialization, the component also needs to load calibration data from the OCDB. Initialization, either from OCDB entries or command line arguments, or both is inherently component specific and falls into responsibility of the developer.

A component has to implement a standard behavior if no arguments have been provided. Ideally this also represents the most probable use case.

4.3.4 Data Processing

Once the component has been set up it is ready for data processing. This section describes the two types of data processing interfaces and the three types of components which are known to the system.

Component type

Components can be of type

- `kSource` components which only produce data
- `kProcessor` components which consume and produce data
- `kSink` components which only consume data

where data production and consumption refer to the analysis data stream. For each type a standard implementation is available as outlined in Figure 4.4.

- `AliHLTDataSource` for components of type `kSource`
  All types of data sources can inherit from `AliHLTDataSource` and must implement the `AliHLTDataSource::GetEvent` method. The class also implements a standard method for `GetInputDataTypes` as data sources do not expect any input data.
- `AliHLTProcessor` for components of type `kProcessor`
  All types of data processors can inherit from `AliHLTProcessor` and must implement the `AliHLTProcessor::DoEvent` method.
- `AliHLTDataSink` for components of type `kSink`
  All types of data processors can inherit from `AliHLTDataSink` and must implement the `AliHLTDataSink::DumpEvent` method. As sink components do not produce output, the class also implements standard methods for `GetOutputDataType` and `GetOutputDataSize`.

Out of those, only the `AliHLTProcessor` is relevant for on-line processing. The two others are mainly intended to be used within the AliRoot framework. The on-line system always expects the component to be a processor. Literally, `kSink` components can also be considered processors, just without output. There are several utility components which can be used on-line in order to collect data. `kSource` components cannot be used in the on-line environment as initial data sources because the data transport framework does not provide a corresponding *DataSource* wrapper component. A data source in the on-line system must always be an appropriate component out of the PubSub package.

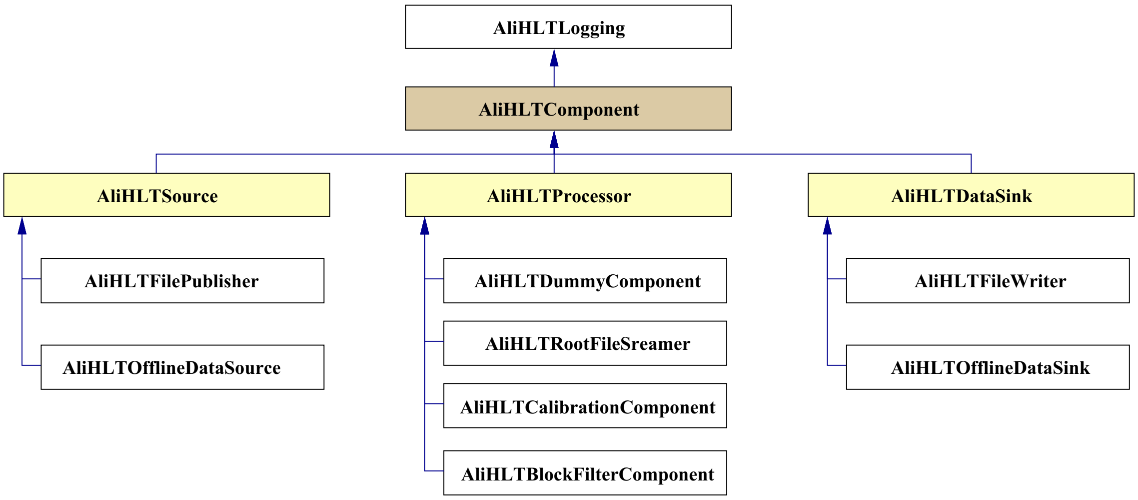


Figure 4.4: Inheritance diagram for base class `AliHLTComponent` and derived standard implementations for data sources, sinks and processors, and examples components for each of the three types. The base class inherits common logging functionality from the class `AliHLTLogging`.

Data exchange

Fundamental concepts of data exchange in the HLT have been presented in section 3.4. Descriptors contain all relevant information like data type, data specification, data buffer and data size. The component gets a list with all available descriptors for one event and can access data from memory by means of the descriptor members as illustrated in Figure 3.9 of section 3.4.3. The corresponding data structure is defined in listing 4.3

Listing 4.3: Definition of the data block descriptor. The size of the structure is used as version identifier

```
struct AliHLTComponentBlockData
{
  /* size and version of the struct */
  AliHLTUInt32_t fStructSize;
  /* shared memory key, ignored by processing components */
  AliHLTComponentShmData fShmKey;
  /* offset of output data relative to the output buffer */
  AliHLTUInt32_t fOffset;
  /* start of the data for input data blocks, fOffset to be ignored*/
  void* fPtr;
  /* size of the data block */
  AliHLTUInt32_t fSize;
  /* data type of the data block */
  AliHLTComponentDataType fDataType;
  /* data specification of the data block */
  AliHLTUInt32_t fSpecification;
};
```

One descriptor object of this definition occupies minimum 50 Bytes in memory, depending on alignment and bit width of the architecture it might be a few bytes more. In contrast to that, data blocks can reach arbitrary sizes. This approach ensures minimization of computing resources spent on information exchange.

Each HLT module can define its own data exchange structures and corresponding data type ids. Usually, the first steps of the analysis hierarchy are self-consistent for each detector and the components just need to recognize the private data type ids of the HLT module. For merging of data blocks and track matching among sub-detectors, common data types and data structures have to be used.

A data format for exchange between components can be a simple C structure. The exchanged data consist of an array of such C structures in memory and a variable containing the number of such structures in the beginning. This approach provides the most effective method to transfer data, it is reduced to the minimal necessary information. Since system architecture properties are not stored within the data structure, it can only be used between components running on machines of the same architecture. As soon as the system architecture is changed, data exchange must be carried out with respect to data properties as outlined in section 3.4.4. This is in particular important for the output of HLT, no matter if it is data for monitoring, designated to be stored in the OCDB or reconstructed events to be stored in the data stream.

The ROOT framework provides functionality to carry out data exchange between different computer architectures transparently for the user. A ROOT object is derived from the base class `TObject` and can be serialized and optionally compressed into a buffer. Increased flexibility comes at the cost of increased overhead in terms of data size and/or processing time due to serialization and compression. This topic is investigated in detail in section 4.6.

Data Processing Interface

A processing function is the heart of the data treatment and must be implemented by each component. It is called once per event. Based on the data types, the component selects the input data blocks it can treat from the input stream. Calculated output data are stored in the provided buffer and the corresponding block descriptors are inserted into the output stream.

Each of the component base classes `AliHLTProcessor`, `AliHLTDataSource` and `AliHLTDataSink` provides its own processing method, which splits into a high and a low-level method respectively. For the low-level interface, all parameters are shipped as function arguments, the component has access to the output buffer and handles all block descriptors. This interface allows full flexibility and is used for performance critical algorithms.

For the sake of simplicity and usability, the high-level interface has been introduced. It is the standard processing method and will be used whenever the low-level method is not overloaded. The high-level interface simplifies the implementation of the processing function by moving common functionality often repeatedly implemented by components to the base class. It is also suited for the data exchange of complex ROOT objects like histograms, object arrays and `TTree` objects. The data access paradigms are illustrated in Figure 4.5.

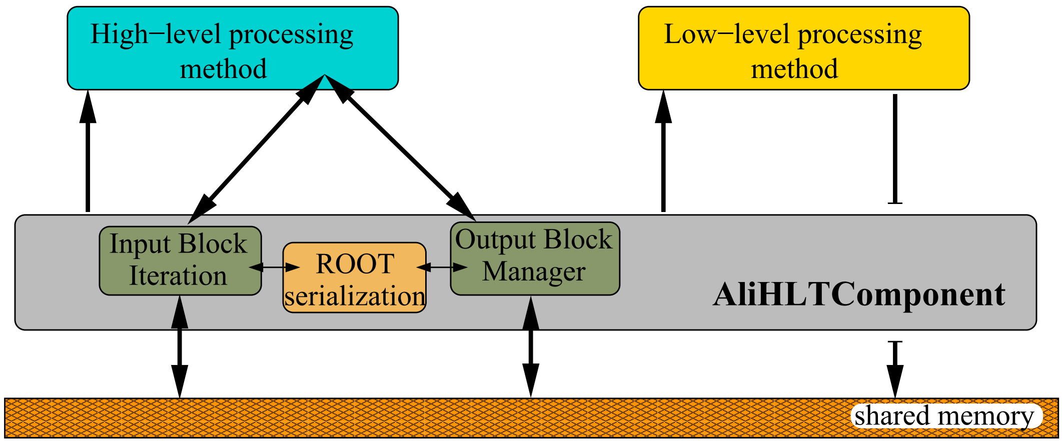


Figure 4.5: The high- and low-level processing interfaces allow implementation of processing based on different paradigms. The high-level interface inserts an abstraction layer into the data access. The component base class controls access and provides common functionality for simplified handling of e.g. ROOT objects. The low-level processing function allows direct access to data.

In both cases it is necessary to calculate/estimate the size of the output buffer before the processing. Output buffers can never be allocated inside the component because of the push-architecture of the HLT. For that reason the `GetOutputDataSize` function has to return a rough estimation of the data to be produced by the component. Based on the estimation, the framework provides an output buffer big enough to receive the complete output. The component writes output data to the buffer and creates new block descriptors.

Low-level Interface

The low-level component interface consists of the specific data processing methods for `AliHLTProcessor`, `AliHLTDataSource`, and `AliHLTDataSink`, an example is shown in listing 4.4 for the processor.

Listing 4.4: Definition of the low-level version of the `DoEvent` processing method for the `AliHLTProcessor` class.

```
int AliHLTProcessor::DoEvent(const AliHLTComponentEventData& evtData,
                             const AliHLTComponentBlockData* blocks,
                             AliHLTComponentTriggerData& trigData,
                             AliHLTUInt8_t* outputPtr,
                             AliHLTUInt32_t& size,
                             vector<AliHLTComponentBlockData>& outputBlocks)
```

The base class passes all relevant parameters for data access directly on to the component. Input blocks can be accessed by means of the array `blocks`. Output data written directly to shared memory provided by the pointer `outputPtr`. Output block descriptors are inserted directly to the list `outputBlocks`. Appendix B.1.3 sketches an example implementation of the low-level processing method. `AliHLTDataSource` and `AliHLTDataSink` components have similar methods, however parameters concerning input or output data access are omitted respectively.

High-level Interface

The processing methods are simplified by cutting out most of the parameters. For processors, the function prototype looks like outlined in listing 4.5.

Listing 4.5: Definition of the high-level version of the `DoEvent` processing method for the `AliHLTProcessor` class.

```
virtual int DoEvent(const AliHLTComponentEventData& evtData,
                    AliHLTComponentTriggerData& trigData);
```

In contrast to the low-level interface, a couple of functions can be used to access data blocks of the input stream and send data blocks or ROOT `TObject` instances to the output stream. In particular the following functions are available:

- `GetNumberOfInputBlocks`
  return the number of data blocks in the input stream
- `GetFirstInputObject/GetNextInputObject`
  Iteration over ROOT objects of a specific data type. If the data block describes a serialized ROOT object it is extracted and made available.
- `GetFirstInputBlock/GetNextInputBlock`
  Iteration over data blocks of a specific data type

- `PushBack`
  Insert an object or data buffer into the output. The framework carries out all necessary operations to serialize the ROOT object and/or writes the data to the output buffer. Block descriptors are inserted to the output stream.

An example implementation of a high-level processing function is given in appendix B.1.4.

Exchange of ROOT Objects

Based ROOT's `TMessage` class, the high-level component interface provides functionality to exchange ROOT structures between components. The transport of ROOT object is the foundation for re-use of part of the off-line algorithms in on-line applications and the simplified realization of monitoring and calibration algorithms based on histograms. It is a direct achievement of the component interface implementation which has been accomplished as part of the presented work. Data exchange will be described in detail in section 4.6.

4.4 Component Handler and External Interface

The *ComponentHandler* is an important part the modular concept of the HLT analysis framework. It provides an abstraction layer between applications going to instantiate and use components, and the actual implementation of components. On the side of the application, the chain configuration determines the types of components to run and the interconnections between those. In the HLT framework, the described task is carried out by the class `AliHLTComponentHandler`.

4.4.1 Overview

By making a clear separation between 'users' of components on the application side and 'developers' of components, the whole system can be implemented with a minimum of cross-dependencies and without the need of adapting source code and functionality of the base system whenever a new module is added. Figure 4.6 sketches the role of the ComponentHandler in the HLT analysis framework.

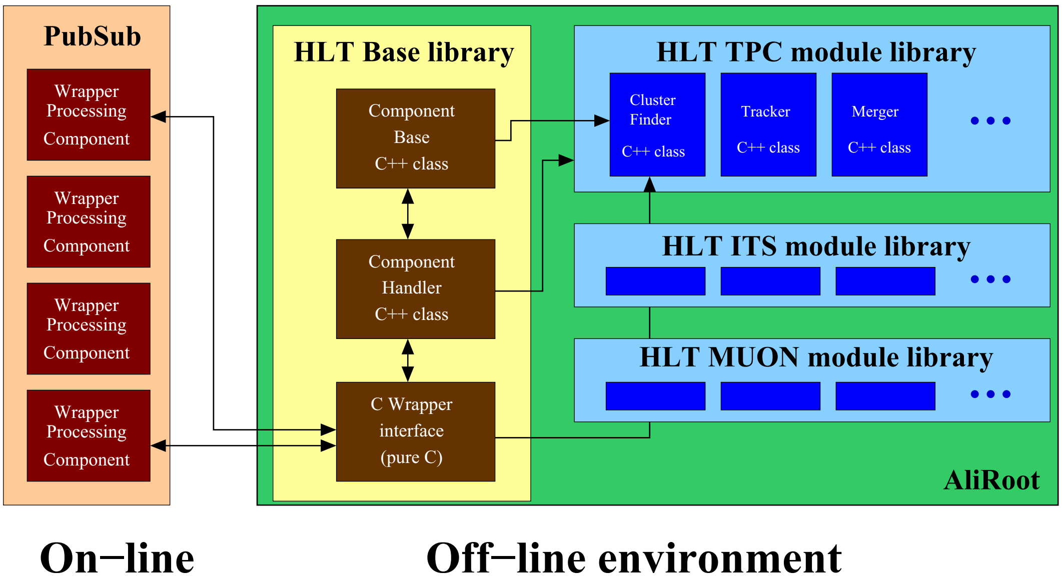


Figure 4.6: The ComponentHandler provides the interface between an application and the component implementation in a module library. Figure depicts the utilization of analysis components in the on-line HLT.

An application queries the ComponentHandler for the existence of a certain component by means of the ComponentId. The handler checks whether the component is available and creates an instance of the component. Availability of components is based on registration.

Components are registered automatically at load-time of the component shared library under the following suppositions:

- the module library has to be loaded from the `AliHLTComponentHandler` using the `AliHLTComponentHandler::LoadLibrary` method,
- the component implementation defines one global object of each component class, or
- the module library implements a module agent (`AliHLTModuleAgent`) which supports registration of components of the module.

4.4.2 Component Registration

An HLT analysis component can be used in the system after registration, which is responsibility of the module library. Two methods are available.

Registration from the Module Agent

The more advanced approach makes use of an `AliHLTModuleAgent` implementation in the module library by overloading the `RegisterComponents()` base class function. This function is invoked for each library loaded by the component and allows the agent to register all components implemented by the module. All necessary source code is in the scope of the module, listing 4.6 shows an example. The source code has been taken from the `libAliHLTUtil` which implements a toolbox of common components.

Listing 4.6: Sample implementation of the `RegisterComponents` method.

```
/** @file    AliHLTAgentUtil.cxx
    @author  Matthias Richter
    @brief   Agent of the libAliHLTUtil library
*/

// .....

// header files of library components
#include "AliHLTDataGenerator.h"
#include "AliHLTRawReaderPublisherComponent.h"
#include "AliHLTLoaderPublisherComponent.h"

// .....

int AliHLTAgentUtil::RegisterComponents(AliHLTComponentHandler* pHandler) const
{
  // ...
  pHandler->AddComponent(new AliHLTDataGenerator);
```

Listing 4.7: Implementation of a class using component registration via a global object.

```
/** @file     AliHLTSampleOfflineSinkComponent.cxx
    @author   Matthias Richter
    @brief    A sample offline sink component.
*/

#include "AliHLTSampleOfflineSinkComponent.h"
// ...

/** global instance for agent registration */
AliHLTSampleOfflineSinkComponent gAliHLTSampleOfflineSinkComponent;

/** ROOT macro for the implementation of ROOT specific class methods */
ClassImp(AliHLTSampleOfflineSinkComponent)

AliHLTSampleOfflineSinkComponent::AliHLTSampleOfflineSinkComponent()
{
  // ...
}
```

```
  pHandler->AddComponent(new AliHLTRawReaderPublisherComponent);
  pHandler->AddComponent(new AliHLTLoaderPublisherComponent);
  // ...
  return 0;
}
```

Registration by Global Objects

The component registration via global static objects follows the same idea as the registration of module agents as described in section 4.2.4. This simple and quick approach makes use of the fact that all global static objects of a library are instantiated when the library is loaded for the first time. Consequently, the library must not be loaded before. This can accidentally be the case when the module library is explicitly linked to the application or loaded before dynamically, e.g. in the course of the *rootlogon*[1] procedure. Both cases must be avoided for a correctly working registration. Listing 4.7 shows an example of a class implementation with global object registration.

The utilization of global objects is often considered bad programming style. In the presented case it does not cause any problems since the object itself

[1] *rootlogon* denotes the customization script for ROOT startup. If existent, the macro *rootlogon.C* is automatically executed. The macro is used to set up the environment for a ROOT session

is not used for data processing at all. However, for a smooth operation the component should not allocate memory in the constructor.

4.4.3 Utilization in the On-line System

The interface between the on-line data transport framework and the HLT analysis components is provided by a C-interface. In dynamic libraries, functions in the source code correspond to entry pointers in a compiled library. During program execution, all parameters are stored on the *stack*[2] and the program jumps to the address in the library. A C-interface makes use of that execution sequence by providing the entry pointers to the external application. Figure 4.7 sketches the entry points of the interface used by the on-line HLT system.

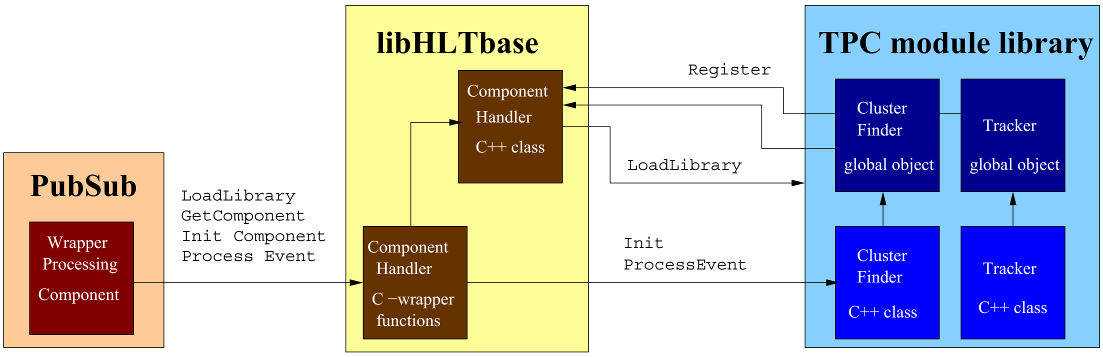


Figure 4.7: Utilization of the C wrapper interface by the on-line HLT. The Wrapper Processing Component of the data transport framework has access to the analysis component through methods of the C interface.

During initialization of the interface, the internal ComponentHandler and the execution environment is set up. Subsequently, component libraries can be loaded, components be created and events processed.

Care has to be taken with all memory allocation and data processing as the two frameworks run in different domains. E.g. a memory buffer allocated inside the HLT analysis cannot be freed in the PubSub system without introducing a potential source for memory corruption. Depending on the compilation and running environment, memory management might be different in the two domains. As a consequence, the interface contains functions allocating memory on the side of the external application.

The interface consists of a couple of functions and data structure definitions provided by the HLT analysis framework and known to the external appli-

[2] In computing architectures, a stack is a special kind of data storage based on the Last-In-First-Out paradigm

cation. The definitions contain three parts, which will be described in the following.

- Common data structures for data exchange
- Definition of interface functions to be called by external applications (listing 4.8)
- Definition of call-back functions implemented by the external application and called by the interface (already introduced in listing 4.1)

The HLT analysis framework implements the corresponding functions in the `libHLTinterface` library. The defined entries correspond to addresses in the compiled library. Common functionality of the operating system can be used to retrieve the address for a function entry.

Data Structures

Several data structures are defined to organize data exchange between the two participants of the interface. Although data processing entirely takes place on the HLT analysis framework's side and the on-line system manages data exchange between processes without touching data payload, a minimum of common data structures is required for the interplay. In particular the data block descriptor as introduced in section 4.3.4 needs to be known to both sides of the interface. The on-line system retrieves type and size of the data block from the block descriptor in order to handle payload in the shared memory correctly. The `AliHLTComponentBlockData` structure has been presented in listing 4.3.

Interface Functions

The names of the functions are self-explanatory and some of the functions can be recognized as counterparts of the presented `AliHLTComponent` interface.

Environment Functions

The external interface utilizes the environment definition as presented in listing 4.1. The role of the running environment has been pointed out in section 4.3.2.

Listing 4.8: Definition of interface functions of the HLT external interface

```
int (*AliHLTExtFctInitSystem)(unsigned long version,
                              AliHLTAnalysisEnvironment* externalEnv,
                              unsigned long runNo,
                              const char* runType );
int (*AliHLTExtFctDeinitSystem)();

int (*AliHLTExtFctLoadLibrary)( const char* );

int (*AliHLTExtFctUnloadLibrary)( const char* );

int (*AliHLTExtFctCreateComponent)( const char*,
                                    void*,
                                    int,
                                    const char**,
                                    AliHLTComponentHandle*,
                                    const char* description );
int (*AliHLTExtFctDestroyComponent)( AliHLTComponentHandle );

int (*AliHLTExtFctProcessEvent)( AliHLTComponentHandle,
                                 const AliHLTComponentEventData*,
                                 const AliHLTComponentBlockData*,
                                 AliHLTComponentTriggerData*,
                                 AliHLTUInt8_t*,
                                 AliHLTUInt32_t*,
                                 AliHLTUInt32_t*,
                                 AliHLTComponentBlockData**,
                                 AliHLTComponentEventDoneData** );
int (*AliHLTExtFctGetOutputDataType)( AliHLTComponentHandle,
                                      AliHLTComponentDataType* );
int (*AliHLTExtFctGetOutputSize)( AliHLTComponentHandle,
                                  unsigned long*,
                                  double* );
```

Using the interface definition, the wrapper processing component of the data transport framework initializes the analysis framework and a component as summarized in the flow diagram of Figure 4.8.

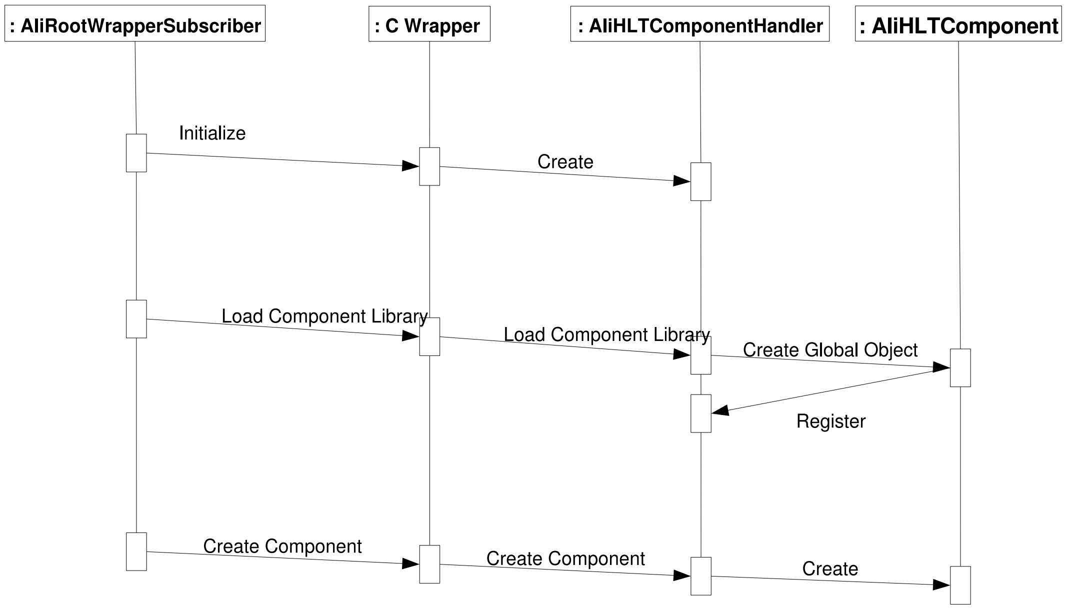


Figure 4.8: Component initialization and registration from the on-line wrapper processing component. Component is registered by using global objects.

Interface Implementation

The interface information is defined in the header file `AliHLTDataTypes`[3]. This file contains the common denominator for HLT analysis and external application and is maintained in the AliRoot repository. The file defines the interface version, data types, common data origins, data structures and interface functions together with callback functions. A copy of the file in the PubSub project allows to compile the latter independently of AliRoot. The versioning scheme allows to check whether the definition of the interface on both sides are compatible. All changes of the interface must be backward compatible in order to allow older versions of either analysis or data transport framework to work with newer versions of the other.

In the evolution of software projects, new functionality needs to be added, other functionality becomes deprecated. Also the communication between interface provider and user needs to be extended and adapted to emerging features. Usually, an interface client knows a list of function names and can query the function entry points by means of `dlopen/dlsym`[4]. One has to keep

[3] http://alisoft.cern.ch/viewvc/trunk/HLT/BASE/AliHLTDataTypes.h?root=AliRoot&view=log

[4] available on all Unix-type operating systems, similar methods exist for other OS

in mind that the returned pointer does not have any attributes. The interface approach only works under two assumptions: (i) the name of a function must be unique and known to the client. (ii) the number and types of parameters must be known to the client. In such an interface approach, there is no semantic check possible.

Regarding function names, compilers impose another restriction caused by *Name Mangling*. This technique provides a way of encoding additional information about the function in the functions name. This is of importance for programming languages like C++ which support overloading of functions and coexistence of functions of identical name but different parameters. In such a case, the final unique function name encoded in the library cannot be directly derived from the function name in source code. Moreover, different compilers use different mangling techniques and formats. It is of great importance to define function names in a universal way which allow clients to retrieve the entry. This is the reason why a C-interface has been chosen. The C programming language does not support overloading of functions, thus no name mangling is necessary. The encoded name in the library is in plain text and can directly be derived from the name in source code. Functions must be defined using the attribute `extern "C"` in order to indicate a C-style function when using a C++ compiler.

In order to overcome the mentioned issues and to support the desired flexibility in the interface design, a technique called *Function Signature Query* has been developed for ALICE HLT. The interface implements its own scheme to retrieve library entry addresses. A general function definition as given in listing 4.9 is supposed to be constant throughout the whole live cycle of the project. It can be utilized by external applications to get the corresponding function pointer. The existence of a function is queried by passing a unique signature consisting of a string with the following attributes:

- the name of the function,
- the return type of the function, and
- the number and types of parameters.

Listing 4.9: The general interface call allows to query function entries by means of the function signature.

```
extern "C" void* AliHLTAnalysisGetInterfaceCall(const char* function);
```

The interface implementation on the HLT framework side checks function signatures for existence and returns the pointer to the function if the signature is known. This scheme allows any extension in the future by defining new functions and signatures. At the same time it solves the problem of preserving backward compatibility when new function parameters are added. Backward compatibility of the interface implementation can easily be preserved by just keeping older function signatures, calls to the old functions are redirected to the new version using default values for the newly added parameters. More details can be found in the daily updated ALICE HLT analysis framework on-line documentation[5].

4.5 Integration into the ALICE Software Framework

In the previous sections, HLT analysis has been described for the on-line system. The modular concept based on library plug-ins and components allows to build a distributed analysis chain with a high degree of flexibility. New modules can be implemented and added to the analysis framework by utilization of module agents and component interface introduced earlier in this chapter. For the full integration into the ALICE software framework, an appropriate HLT running environment is required.

A broad community of physicists develops analysis tools and algorithms for the ALICE experiment and is familiar with the off-line data processing framework. In order to gain from that knowledge, this community must be enabled to develop HLT algorithms within the off-line framework. Furthermore, assurance of trigger selectivity and performance requires comparison of HLT components to the off-line reconstruction and analysis. Since the off-line algorithms are tested in extensive simulation cycles, the obvious approach for HLT analysis is the integration into the already existing procedure. Here it is important to run on-line analysis components without any change in the source code. Customizations of software to the needs of a certain framework is an erroneous and tedious process, which must be avoided. The objective is to run HLT analysis chains in AliRoot in the same way as in the on-line framework, i.e. the full components are run also from the off-line framework rather than just the algorithm hooked on by a special interface class.

Encapsulation of HLT analysis has been considered an important capability already from the beginning of the development. We think of the HLT as a 'black box' with certain data input and output. In addition there is access to calibration data from a data base. Data processing inside components is restricted to input data. As the different detector algorithms/components will run in separate processes and even on different machines, no data exchange is possible via global data structures and variables. This section describes the binding AliRoot functionality of the ALICE HLT which has been developed in the course of the presented work. After a brief introduction into the overall

[5] http://web.ift.uib.no/~kjeks/doc/alice-hlt-current/group__alihlt__wrapper__interface.html

data flow, the `AliHLTSystem` steering class will be described, followed by concrete examples how to run HLT analysis during AliRoot simulation and reconstruction.

4.5.1 AliRoot Data Processing Flow

Data processing in AliRoot is mainly divided into three major parts: simulation, reconstruction and analysis ([13]). There is also satellite functionality like e.g. event visualization. The overall data flow is sketched in Figure 4.9. Within that scheme, the reconstruction step takes its input either from simulation of the detector response or real data. Event reconstruction is a centrally managed layer which usually hides all detector raw data, calibration and alignment processes from the end-user doing the subsequent analysis. It produces a detailed summary of the event, the *Event Summary Data (ESD)*, containing all information on particles found in the event. The last step, the actual physics analysis is naturally distributed and diversified as it depends on many different interests. Based on the ESD, any physics analysis can be run independently of both simulation and reconstruction.

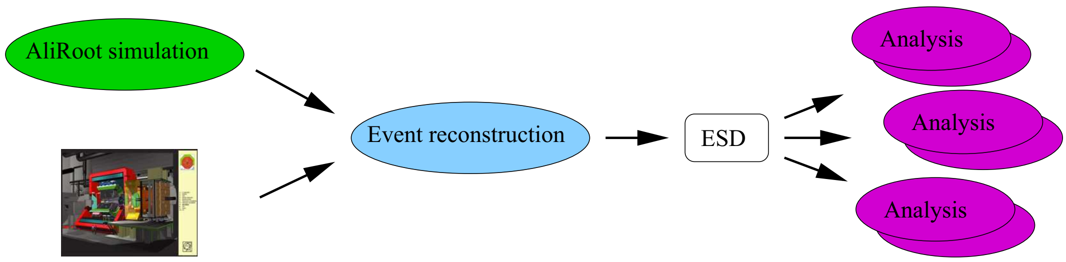


Figure 4.9: Overall AliRoot processing sequence. Event reconstruction is performed on either real raw data or simulated data and produces *Event Summary Data (ESD)* which is the input to the subsequent analysis.

In a complex data analysis system it is important to study the behavior of the different modules and the system response with respect to known input separately in order to assure the result of the entire analysis. This is in particular important in an experiment like ALICE where the physics going to be studied can only be 'seen' through secondary processes and require a complex data processing. The impact of the measuring apparatus and the data processing on the measured quantity needs to be studied carefully. Usually in electronics engineering, the input to a certain signal processing unit is known and can be compared directly to the output. This is not the case for the considered type of experiment.

Analysis of simulated events is thus required to study the behavior and performance of the detector reconstruction. Only after evaluation of the algorithms with high statistics it makes sense to process real data. For simulated events, the result of the reconstruction and analysis can be compared to the original simulated events. This approach is similar to signal response studies used in electronics engineering.

HLT has a special position within the described AliRoot processing chain. As a matter of fact, HLT always runs *event reconstruction* since it is not a sub-detector delivering data. Consequently, it needs to run at the end of the AliRoot simulation when data for all sub-detectors has been generated. Embedded into AliRoot simulation, this reconstruction is performed on simulated data. It is often referred to be *HLT simulation*. Figure 4.10 sketches the data flow for HLT simulation.

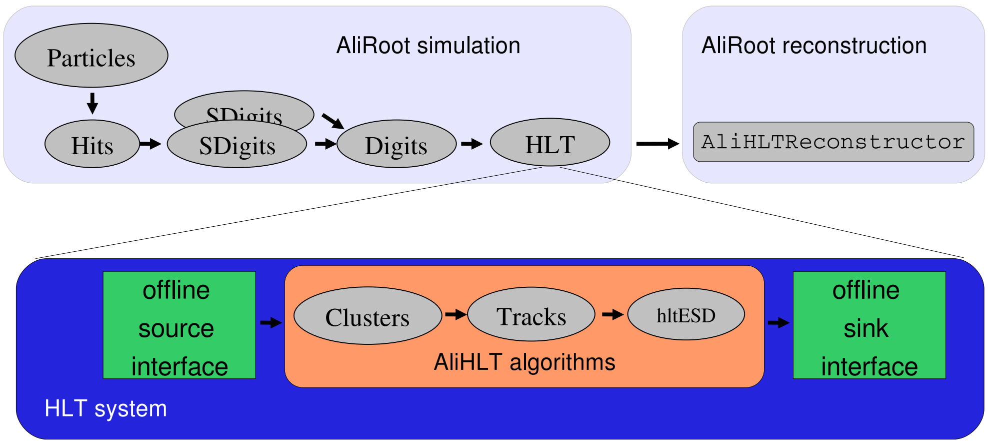


Figure 4.10: HLT simulation in AliRoot.

The lower block denoted HLT system in Figure 4.10 implements an encapsulated HLT analysis with well defined input and output. It is important to notice that the placement of HLT analysis is flexible and just depending on the availability of customized *input data sources* and *output data sinks*. This concept follows the High-Level Trigger design as outlined in Sec. 3.3.1.

Since HLT reconstruction has already been executed either as part of simulation or on-line data processing, the result just needs to be extracted during AliRoot reconstruction and stored appropriately. The treatment of HLT output is described in chapter 5.

Abstract Data Access

The AliRoot software package is organized into steering libraries, providing common functionality, and detector module libraries. Two abstraction layers form the foundation of the modular AliRoot data processing architecture:

- The detector modules follow a plug-in concept in order to provide the detector specific simulation and reconstruction.
- AliRoot provides abstract interfaces for data access which disentangle data processing and all necessary steps of data storage and transportation.

The exact format of data in the different steps of the processing chain will be explained later in the corresponding sections 4.5.3 and 4.5.4. However, the two interfaces for data access in AliRoot will be introduced already here for better understanding.

The RunLoader Interface

In the course of AliRoot simulation, first particles are "generated" and the response of the detector to those virtual events is simulated. Data of intermediate stages are stored in a tree structure in proprietary ROOT files. The common denominator for all detector modules is a naming scheme for trees, branches and files. The `AliRunLoader` provides the interface to the data sets of the different stages.

The *RunLoader* concept is not of major importance for the ALICE HLT and is mentioned here for completeness.

The RawReader Interface

Each detector defines its own raw data format. Because of that, data transportation and storage treats detector raw data as payload. Following the hardware solution of DDLs as described in section 3.3.2, the data flow is grouped in entities of DDL blocks. Each data block has a *Common Data Header (CDH)*, which contains all relevant properties of the block like, e.g. size and trigger information. The format of the CDH is described in [32] and sketched in figure 4.11. Out of the eight 32bit words, word 0 containing the block length and the *Error and Status bits* in word 4 are of interest for the HLT. Furthermore the event id and trigger information is used by the data transport framework for data flow control.

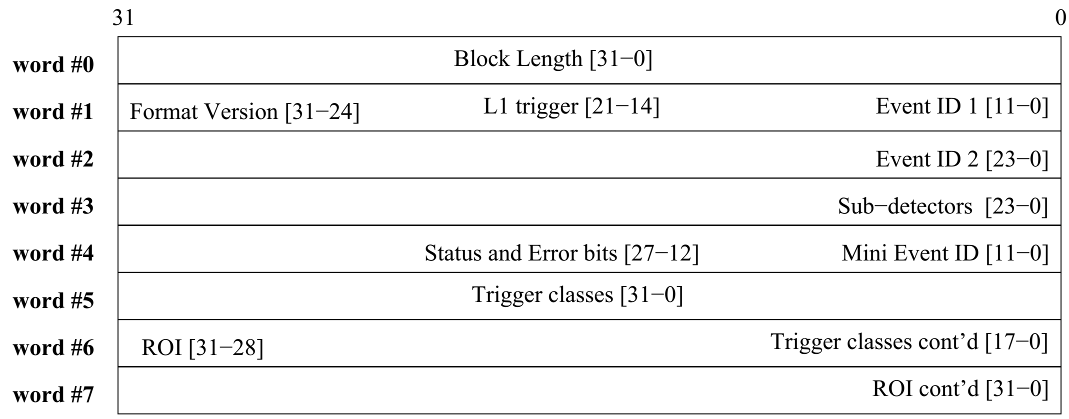


Figure 4.11: The Common Data Header (CDH) consists of eight 32bit words.

Access to raw data is generalized by the AliRoot class `AliRawReader` Raw data are formatted in various representations throughout the processing. Figure 4.12 illustrates the implementations for different formats.

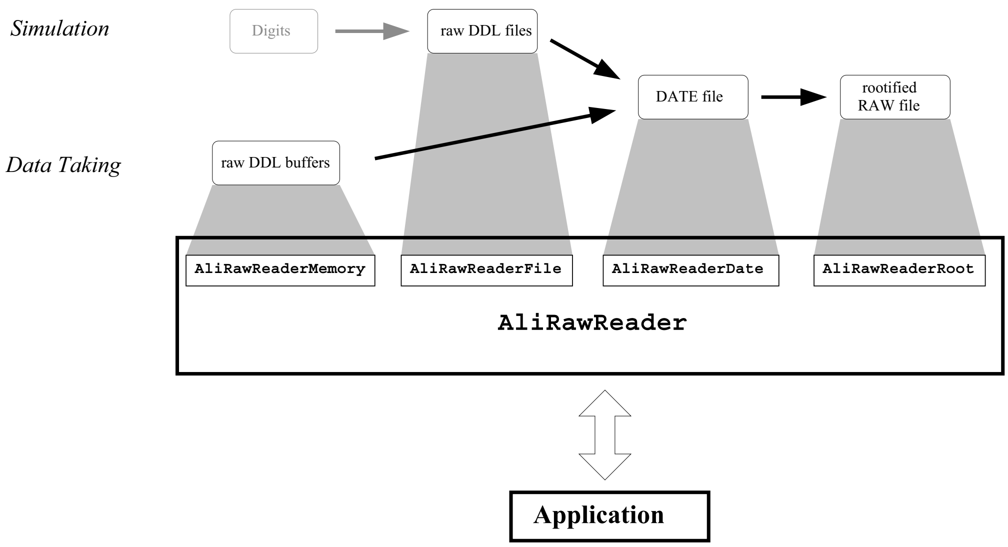


Figure 4.12: The `AliRawReader` interface. Different implementations of readers fit different raw data representations.

The ALICE DAQ formats DDL data blocks as received from the Detector Data Links in a proprietary data format, a so called *Date file*. For final storage, those files are converted into the ROOT file format. In the particular case of ALICE, payload of each DDL is stored in branches of a tree[6]. On the way to the *Date file*, raw data first exists in the form of *raw data blocks*. In case of simulation, these raw data blocks are written to disk in separate files. For all those formats, the class `AliRawReader` provides an abstract access layer. Applications just use the interface, the actual access details are hidden

[6]The `TTree` concept is a fundamental feature of the ROOT storage

(Figure 4.12). For HLT, the `AliRawReaderMemory` has been implemented in the course of the presented work. It allows to feed a number of data blocks into the RawReader and make the payload accessible for the application. It is the prerequisite for both **(i)** utilization of raw data decoding classes from the off-line project and **(ii)** running off-line detector algorithms embedded in an analysis component.

4.5.2 The Off-line HLT System

In order to run analysis components in a chain-like fashion, the behavior of the on-line data transport framework is modeled by the HLT AliRoot integration. The `AliHLTSystem` class together with a few satellite classes steers HLT analysis chains running embedded into AliRoot.

Seen from outside, `AliHLTSystem` is like a black box processing some input data with a certain response and follows strictly the principle of information hiding. The processing is determined by a configuration describing different components connected to each other. Even though `AliHLTSystem` is running in the same process as the `aliroot` executable, accessibility to data is strictly limited. As outlined in Figure 4.13, apart from data input and output data processing is exactly identical in both on- and off-line environment. Together with binary compatibility of the HLT libraries, this is an important achievement of the presented work.

`AliHLTSystem` is able to parse a configuration, build a task list from the entries in the configuration and connects them forming an HLT analysis chain. It also carries out memory allocation and organizes data transport between the components of the chain. The entity of `AliHLTSystem` can be plugged into the AliRoot processing stream at different places. Some exemplary case like e.g. `AliSimulation`, `AliReconstruction`, or the treatment of HLT output payload are described in the next sections.

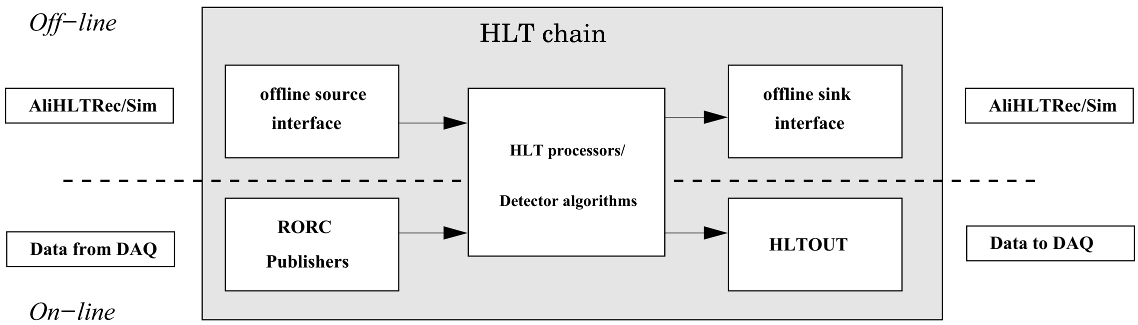


Figure 4.13: HLT analysis chains in the on-line and off-line system. `AliHLTSystem` provides the integration into the off-line data processing flow.

`AliHLTSystem`	Steering class of the whole environment
`AliHLTTask`	Task describing a step in the off-line HLT analysis chain. Holds information about the type and arguments of the HLT component and its connections to other components. Organizes data exchange with other components.
`AliHLTConfiguration`	Helper class for the description of one task in an off-line analysis chain. It contains the meta information which allows `AliHLTSystem` to create the list of tasks
`AliHLTConfigurationHandler`	Registry of available configurations.
`AliHLTOfflineInterface`	Component interface to internal AliRoot data.

Table 4.1: The main classes of the AliRoot HLT environment.

The implementation of the HLT bindings into AliRoot is separated by functionality into a couple of classes. Table 4.1 shows an overview of the most important ones. This whole group is referred to be the `AliHLTSystem` classes.

Usage of AliHLTSystem

The class aims to provide an easy-to-use interface. As a matter of fact, the system can be set up and run with only a few commands (see listing 4.10).

Listing 4.10: Execution of an off-line HLT chain with the `AliHLTSystem`.

```
{
  // ... instance of AliRunLoader runLoader present
  AliHLTSystem gHLT;
  gHLT.ScanOptions("libAliHLTUtil.so loglevel=0x7c"
                   "config=<config-file> chains=<mychain>");
  gHLT.Configure();
  gHLT.Reconstruct(1, runLoader);
}
```

This little programming sequence is just exemplary and does not implement any handling of error conditions. As already mentioned, this module is usually run embedded into other processes and the user does not need to worry about data input and data access via `AliRunLoader` or `AliRawReader` as described in Sec. 4.5.1. For most of the applications, `AliHLTSystem` is hidden from the end user.

The important line here is the `ScanOptions` function, `AliHLTSystem` can be configured by the following optional arguments:

- Detector libraries
  a sequence of detector libraries to be loaded in the course of configuration can be specified. Libraries are recognized by the *.so* suffix. If no library is specified, a list of default libraries will be loaded automatically.
- `config=<config-file>`
  allows to specify a file from which the configuration will be read.
- `chains=<mychain,yourchain>`
  a list of comma separated chain identifiers to be run during the reconstruction.
- `loglevel=0x7c`
  logging level for the filter of log messages. In the particular case, only messages of level *Info* and higher will be propagated.

This options will also be recognized later in the following sections. Arguments of this form directly point to utilization of `AliHLTSystem` behind the scene. Chains to be run are specified by the identifier of the topmost configuration. The format of the configuration will be explained in the next section.

HLT Configurations in AliRoot

In the on-line HLT, a configuration is described in XML notation (section 3.5). Though, writing a configuration in XML notation is straight forward it requires experience and some effort. A simplified solution has been implemented for the definition of off-line HLT configurations, based on the fact that the ROOT physics analysis framework uses C++ as a scripting language and suited for the target community of the AliRoot HLT functionality. An `AliHLTTask` is described by the following parameters:

1. Configuration id, a unique identifier of the entry within the analysis chain,
2. Component id, describing the type of the component to run,
3. Parents, describing the data sources of the task,
4. Component arguments, specifying the optional initialization of the component, and
5. Size of output data buffer, which is an optional property.

Those properties form a minimum description of a component and have been incorporated into a class, `AliHLTConfiguration`, which is sufficient to fully describe the task.

```
AliHLTConfiguration(const char* id, const char* component,
                    const char* sources, const char* arguments,
                    const char* bufsize=NULL);
```

Using this class, configuration entries are simply added by creating an object of this class, specifying all properties within the constructor. An example defining a chain consisting of one publisher, one processing and one data sink component can simply look like illustrated in listing 4.11.

Listing 4.11: A sample configuration describing a chain of two publishers, one processor component and a sink component.

```
{
  AliHLTConfiguration publisher1("fp1", "FilePublisher", NULL,
                                 "-datatype 'DUMMYDAT' 'SMPL'"
                                 "-datafile some-data.dat");
  AliHLTConfiguration publisher2("fp2", "FilePublisher", NULL,
                                 "-datatype 'DUMMYDAT' 'SMPL'"
                                 "-datafile some-data.dat");
  AliHLTConfiguration copy("cp", "Dummy", "fp1 fp2",
                                 "output_percentage 80");
  AliHLTConfiguration sink("sink", "FileWriter", "cp", NULL);
}
```

The corresponding analysis chain is depicted schematically in Figure 4.14. All configuration entries are automatically registered with the *configuration manager*, `AliHLTConfigurationHandler`. The manager makes the entries available to the system for later parsing and building of analysis chains. Complex configurations can include loops in order to build hierarchies. More examples can be found in the on-line ALICE HLT analysis framework tutorial [43].

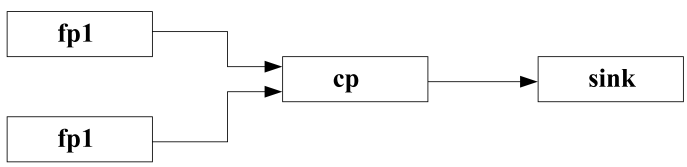


Figure 4.14: Simple example of an off-line HLT chain described by Listing 4.11.

Data Input and Output

As shown in figure Figure 4.13, data source and sink components implement the binding to either on-line or off-line system. The task of a data source is

`FilePublisher`	Simple publisher for files
`RootFilePublisher`	Dedicated publisher for ROOT files and objects from those
`AliRawReaderPublisher`	Publisher for DDL data from an AliRawReader, used both in reconstruction and simulation
`AliHLTOUTPublisher`	Publisher for data blocks from HLT output payload
`AliLoaderPublisher`	Publisher for digit tree objects from the simulation.

Table 4.2: Common HLT data sources. The table shows the ComponentId which can differ from the actual class name.

to read data going to be published from file are another source into memory and to create the corresponding block descriptor.

In the on-line HLT, the task of data sources and sinks is carried out by specific components of the data transport framework as described in section 3.4.3. In the off-line environment, data sources and sinks depend on the host application of `AliHLTSystem`. The framework provides some common publishers to be used in different situations (table 4.2). The simplest case is a stand-alone program or macro where `AliHLTSystem` is used directly without any embedding functionality. In this case, file publishers are often the appropriate method. Depending on the type of the file, the *FilePublisher* or *RootFilePublisher* components can be used.

Running HLT analysis embedded into AliRoot requires special publishers to get hold on internal data of the processing loop. There are different use cases, like simulation, reconstruction and AliRoot analysis framework, but all have in common the separation of the event loop and steering from the actual detector code. Access to data and incrementation of the event position is provided by steering classes calling the detector processing for each event.

In addition, each detector can define and implement its own data sources in order to provide a better suited translation of data into formats understood by the components of the analysis chain.

In all cases, source and sink components need access to the internal AliRoot data handlers, the *RunLoader* and *RawReader* as described in Sec. 4.5.1. The HLT analysis framework implements a base class for that kind of data access, the `AliHLTOfflineInterface` class, Figure 4.15 shows its dependency tree.

Two child classes are especially designed for the purpose of data sources and sinks. In order to combine functionality of a data source with access to internal

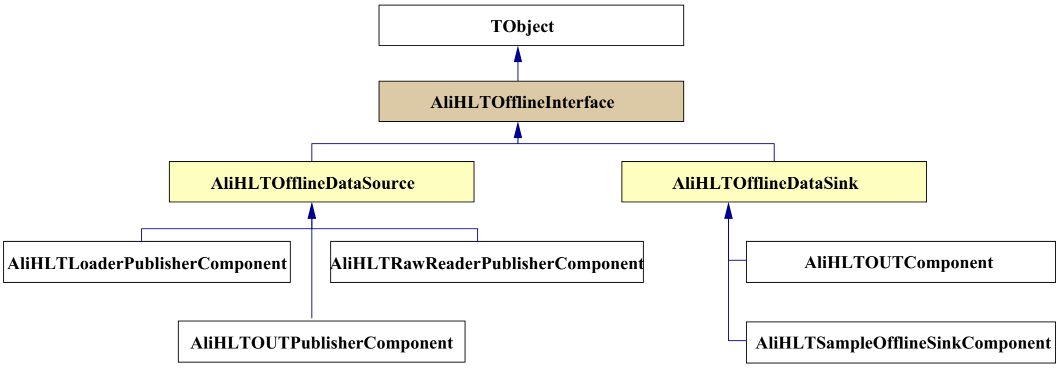


Figure 4.15: Inheritance Diagram for the `AliHLTOfflineInterface` class and the common data sources and sinks provided by the HLT analysis framework.

AliRoot data, the `AliHLTOfflineDataSource` and `AliHLTOfflineDataSink` classes inherit from two base classes as demonstrated in Figure 4.16 for the former. In the event handler methods of the off-line data source and sink implementations, AliRoot data handlers are accessible through methods of the `AliHLTOfflineInterface`:

```
AliRunLoader* GetRunLoader() const;
AliRawReader* GetRawReader() const;
AliESDEvent* GetESD() const;
```

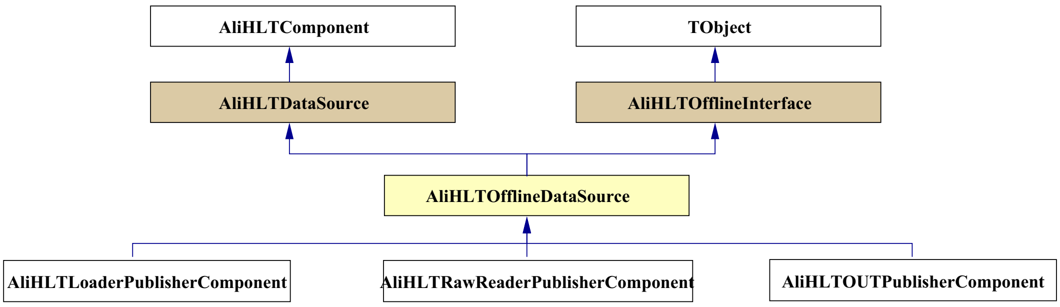


Figure 4.16: The `AliHLTOfflineDataSource` class inherits functionality from both the `AliHLTDataSource` and `AliHLTOfflineDataSource` classes in order to combine functionality of a data source with access to internal AliRoot data.

4.5.3 AliRoot HLT Simulation

In order to simulate the behavior of HLT analysis chains, HLT reconstruction can be embedded into AliRoot simulation.

AliRoot simulation is steered by the `AliSimulation` class. The normal procedure includes the simulation of events by various event generators, simulation of the detector response and the resulting signals, and the generation

Listing 4.12: Manual intervention regarding HLT setup in AliRoot simulation.

```
AliSimulation sim;
...
sim.SetRunHLT("libAliHLTSample.so loglevel=0x7c");
```

of data in raw format. A complex configuration stored in a macro *Config.C* determines the exact process and the parameters like e.g. magnetic field, the generator and its settings to be used, the included sub-detectors, and decay processes.

Since HLT requires all other detector data to be simulated, HLT simulation is run as the last step of `AliSimulation`. Depending on the configuration, simulation can stop at the level of generated *Digits* which are the format of simulated detector response signals. Optionally, those signals can be converted into simulated raw data of exactly the format delivered by the detector front-end electronics. Event reconstruction can be run either from digit data or raw data. This separation is necessary as many of the data productions on the GRID skip the raw data generation at the end of the simulation. Consequently, input to HLT chains must be provided for both alternatives and depending on the availability of simulated raw data or just digit data, the input to the HLT chains is chosen. The actual analysis is designed to be independent of the type of input.

In the default configuration of `AliSimulation` the HLT analysis chain to be run depends on the available plug-ins as described in Sec. 4.5.2. HLT simulation is controlled by the class `AliHLTSimulation` which loads default libraries and runs chains according to the available agents. Figure 4.17 illustrates the whole process.

For the development process and debugging, manual intervention is possible and the options for the HLT simulation can be set by means of the `AliSimulation::SetRunHLT` function (listing 4.12). It allows to specify single module libraries as well as options like the logging level. The options are propagated to `AliHLTSystem` and have already been introduced in chapter 4.5.2. An example macro for AliRoot simulation is shown in listing 4.13.

Default HLT Chains in AliSimulation

Currently, default HLT simulation includes TPC reconstruction and MUON reconstruction. As soon as other detector modules are ready for productions, chains can be defined by means of the `AliHLTModuleAgent` as described in chapter 4.2.4.

Listing 4.13: Example macro for AliRoot-embedded HLT simulation, building on already simulated detector data. Example taken from AliRoot distribution (HLT/exa/sim-hlt-rawddl.C)

```
void sim_hlt_rawddl() {
  AliSimulation sim;

  // switch of simulation and data generation
  sim.SetRunGeneration(kFALSE);
  sim.SetMakeDigits("");
  sim.SetMakeSDigits("");
  sim.SetMakeDigitsFromHits("");
  sim.SetMakeTrigger("");

  // write HLT raw data since we want to replace the original
  // detector data from the HLTOUT
  sim.SetWriteRawData("HLT");

  // set the options for the HLT simulation:
  // libAliHLTUtil.so libAliHLTSample.so
  //  loads the specified libraries since the HLT chain will use components
  //  from those two
  // loglevel=
  //  the internal logging level in the HLT, use 0x7c for higher verbosity
  // config=<file>
  //  the configuration to be run
  // chains=<chain>
  //  run the specified chains, defined in the configuration macro
  sim.SetRunHLT("libAliHLTUtil.so libAliHLTSample.so loglevel=0x7c "
                "config=\$ALICE_ROOT/HLT/exa/conf-hlt-rawddl.C "
                "chains=publisher");
  sim.Run();
}

void conf_hlt_rawddl() {
  ///////////////////////////////////////////////////////////////////////////
  ///////////////////////////////////////////////////////////////////////////
  //
  // the configuration
  TString arg;

  // publisher configuration
  // see AliHLTRawReaderPublisherComponent for details
  arg.Form("-detector ITSSDD -skipempty -datatype 'DDL_RAW ' 'SMPL' -verbose");
  AliHLTConfiguration pubconf("publisher", "AliRawReaderPublisher", NULL ,
                              arg.Data());

  // currently, no more components in the chain, the original data is just
  // forwarded to the HLTOUT
}
```

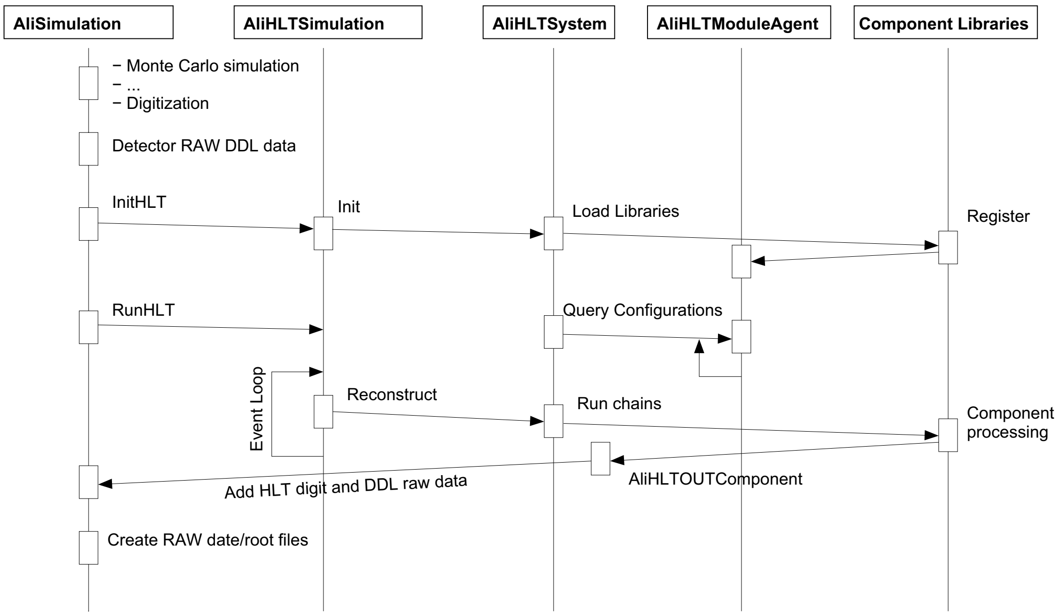


Figure 4.17: Sequence of HLT reconstruction embedded into AliRoot simulation (`AliSimulation`). After the simulation process for the sub-detectors has been finished, HLT simulation runs over all events. Depending on the defined output, the *HLT digit* file and raw data are generated.

Output of HLT simulation

The High-Level Trigger system establishes 10 DDL connections to the Data Acquisition. The HLT decision and payload is transported distributed over the 10 available links. In case of HLT simulation, there is no intermediate *digit* file created. HLT only simulates its DDL raw data links. However, those DDL files are also stored in an *HLT.digit* file. in order to include HLT simulation also in simulations skipping the raw data generation. ROOT's `TTree` class provides a versatile tool for storage of repetitive data sets. In the case of *HLT.digits.root*, the content of the 10 raw DDLs is stored in branches of a `TTree object` named *rawhltout*. Relying on ROOT functionality has the advantage of getting tools for data visualization and investigation, and a graphical user interface. With a minimum effort, handlers for data objects can be implemented.

All output data blocks generated by the components of the last stage are collected by a special component, the `AliHLTOUTComponent`. This component is implemented in `libHLTsim` and is added automatically by the system whenever the defined chains produce some output. The `AliHLTOUTComponent` is a special data sink component which produces the output similar to the HLTOUT of the on-line HLT cluster. The format will be described in Sec. 5.3.

4.5.4 AliRoot HLT Reconstruction

AliRoot reconstruction is steered by the class `AliReconstruction`. Several options are possible in order to deactivate certain detectors, choose the input and to define the output. Also AliRoot reconstruction implements a modular system. The specific detector reconstruction is carried out by detector plug-ins. The interface for the so called *reconstructors* is defined by the `AliReconstructor` class. At the beginning of the reconstruction, the steering class carries out the setup of data input, OCDB, and all necessary reconstructor modules. After the initialization, the event loop is entered which consists of mainly three steps:

1. Local event reconstruction is usually the place for digit/raw data conversion to clusters/space points.
2. Event reconstruction accomplishes track and event reconstruction on the basis of clusters and space points.
3. ESD fill as the last step of reconstruction writes the information to the ESD.

Like all other detector modules, HLT implements the `AliHLTReconstructor` plug-in in order to be hooked up into the event reconstruction loop. However, all analysis is supposed to run on-line on the HLT farm. Thus, only the processing of the HLTOUT data is necessary during the default reconstruction. For a fully reconstructed event, the ESD data block is extracted from the HLTOUT payload and stored into the HLT tree of the final ESD. Thus, `AliReconstructor::FillESD` has turned out to be the natural place to run HLTOUT processing.

Although only HLTOUT processing is needed in the course of normal reconstruction, `AliHLTReconstructor` provides the full flexibility to run also HLT chains embedded into `AliReconstruction`. This is mainly for the purpose of debugging and the development cycle. HLT chains in the course of `AliReconstruction` can only run on raw data. This restriction is imposed by the design of the `AliReconstructor` interface. The data flow is illustrated in Figure 4.18, listing 4.14 shows an example macro.

Any data which are produced by the chain are automatically added to the HLT output collection. In other words, the actual location of an HLT chain in the overall data processing flow is irrelevant. It can be processed as part of the normal on-line processing and the result be added to HLTOUT, it can run embedded into `AliSimulation` and the output added to simulated HLTOUT, or it can run embedded into `AliReconstruction` and the output is added as

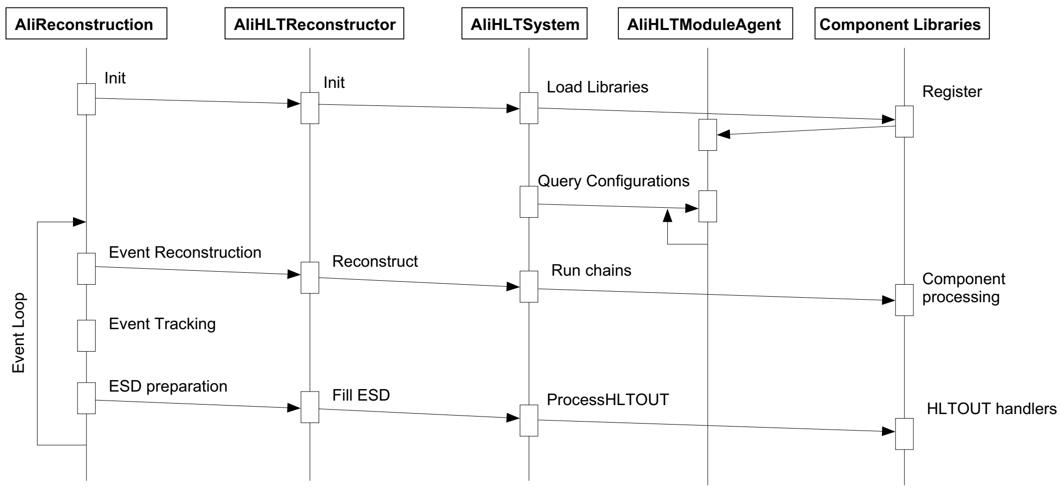


Figure 4.18: Sequence of HLT reconstruction embedded into AliRoot reconstruction (`AliReconstruction`). Processing of HLT output will be discussed in section 5.3.

sub-collection to the existing HLTOUT. The developer has the full flexibility to test and commission a set of components.

Default Reconstruction Chains

All module agents can define default reconstruction chains by means of the function `AliHLTModuleAgent::GetReconstructionChains`. The same function is used for both AliRoot simulation and reconstruction. The two applications differ by the available parameters. In the former case, the *RunLoader* is always available and the *RawReader* might be optionally provided depending on whether raw data are simulated or not. In the latter case, the *RunLoader* is always *NULL* and the *RawReader* is available. Currently, none of the detector modules is using the possibility to run default reconstruction chains in the course of `AliReconstruction`. However, there are several use cases for stand-alone runs and test macros.

Listing 4.14: Example macro for AliRoot HLT reconstruction. DDL data blocks from the RawReader are published, the connected FileWriter component writes the blocks to files. The complete macro is part of the AliRoot distribution (HLT/exa/publish-rawreader-data.C)

```
void publish_rawreader_data(const char* input, int iMinDDLno, int iMaxDDLno)
{
  /////////////////////////////////////////////////////////////////////////
  //
  // some defaults
  const char* baseName="RAW.ddl";

  AliReconstruction rec;
  rec.SetInput(input);
  rec.SetOption("HLT", "libAliHLTUtil.so loglevel=0x7c chains=sink1");

  /////////////////////////////////////////////////////////////////////////
  //
  // setup of the HLT system
  gSystem->Load("libHLTrec");
  AliHLTSystem* pHLT=AliHLTReconstructorBase::GetInstance();
  if (!pHLT) {
    cerr << "fatal error: can not get HLT instance" << endl;
  }

  /////////////////////////////////////////////////////////////////////////
  //
  // the configuration chain
  TString writerInput;
  TString arg;

  arg.Form("-minid %d -maxid %d -skipempty -verbose", iMinDDLno, iMaxDDLno);
  AliHLTConfiguration pubconf("publisher", "AliRawReaderPublisher",
                              NULL , arg.Data());
  if (!writerInput.IsNull()) writerInput+=" ";
  writerInput+="publisher";

  // the writer configuration
  arg.Form("-specfmt=_%%d -subdir=out%%d -blocknofmt= -idfmt= -datafile %s",
           baseName);
  AliHLTConfiguration fwconf("sink1", "FileWriter"    ,
                             writerInput.Data(), arg.Data());

  /////////////////////////////////////////////////////////////////////////
  //
  // the reconstruction loop
  rec.SetRunReconstruction("HLT");
  rec.Run();
}
```

4.5.5 Event Summary Data (ESD)

As the analysis of experiment data is a multi-stage process, in particular data formats for intermediate storage and exchange between stages play an important role. The result of the event reconstruction is stored in the Event Summary Data (ESD). Although the design and development of the ESD concept is not part of the presented work, this section introduces the concept for better understanding of the following sections. The ESD is the input to the physics analysis and contains all relevant information about the event such as reconstructed tracks for various sub-detectors, multiplicity information, clusters in the calorimeters, raw data error log, and vertex information. The ESD structure is provided by the `AliESDEvent` class and is sketched in Figure 4.19.

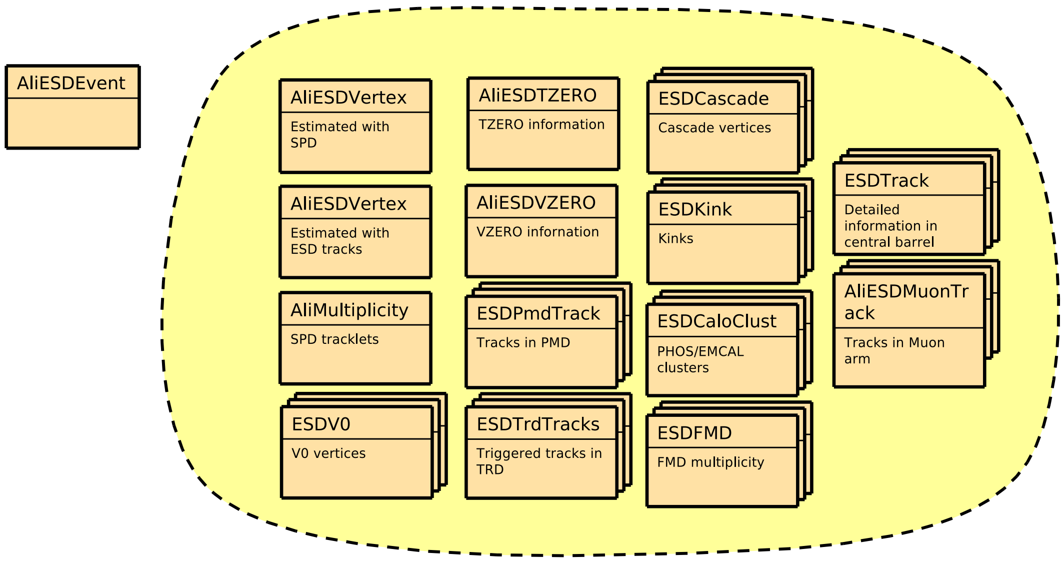


Figure 4.19: Structure of the `AliESDEvent` class. From [44].

The ESD is considered *the* container for all relevant data of a reconstructed event. Different members take account for the various types of information, e.g. vertex, multiplicity, deposited energy, and reconstructed tracks. The following description refers to the members of `AliESDEvent` shown in Figure 4.19.

Reconstructed Tracks

One major contribution to the ESD are arrays of reconstructed tracks. Due to the multi-purpose nature of the ALICE detector there are several types of reconstructed tracks foreseen. Tracks for the barrel part of the detector (ITS, TPC, TRD) are stored in the *Tracks* array, tracks for the MUON spectrometer are stored in the *MUONTracks* member. All ESD members of this type are stored as arrays, namely the `TClonesArray` container provided by ROOT (see Sec. 4.6.2). Different types of tracks are reconstructed by the various modules of the reconstruction also in on-line analysis. Following the reconstruction hierarchy described in Sec. 3.3, each detector reconstruction ends up with the relevant information. Converting it to ESD format allows to easily exchange those information between modules. E.g. tracks found in TPC and TRD must be matched to produce combined barrel tracking information, and ITS reconstruction needs seeds form the TPC.

However, there is a big impact on the performance when using ESD objects for data exchange. The multi-purpose design of the `AliESDEvent` class comes at the cost of introduced overhead in memory consumption and processing time (see section 4.6).

Vertex Information

There are several sub-detectors producing a vertex information which all are stored in members of type `AliESDVertex`. Currently, vertex information produced by TPC and ITS is stored separately (members `fTPCVertex` and `fSPDVertex`) as well as a combined information (members `fPrimaryVertex`).

Multiplicity Information

Multiplicity information is retrieved from FMD, VZERO, and SPD and stored in corresponding objects.

Calorimeter Information

There are two calorimeters in ALICE contributing with energy and trigger information. It is stored separately (members `fPHOSCells` and `fEMCALCells`) and combined (member `fCaloClusters`).

Error Log

Raw data can be corrupted due to various reasons. As a consequence, events might not be reconstructed or only partially. Incomplete and missing events have a significant impact on statistics and analysis. Processing failures must be collected and taken into account when normalizing physics results. The error log is produced at the level of reading raw data, the `AliRawReader` class provides appropriate functionality for efficient archiving of raw data errors.

4.6 HLT Data Exchange

Beside the actual processing of data, the total processing cost in the High-Level Trigger system is also heavily influenced by two more contributions, memory access and data exchange. Memory access within an algorithm is very important for its performance. All modern microprocessors implement a caching mechanism in order to reduce memory access latency. The cache is a small but fast memory which stores a copy of the most frequently used data. The memory is organized in so called pages and the cache holds copies of pages of the main memory. An indexing scheme allows the localization. As long as data are mirrored in the cache, the processor can read and write memory with very short latency. Every access to data not currently stored in cache requires more clock cycles. As a consequence, algorithms reach their maximum performance when memory access is local and contiguous within a few pages possible to be mirrored in cache.

A second important role plays data exchange between different stages of the reconstruction. Regardless the fact of distribution of data processing, data need to be exchanged. That holds for data treatment in one single process as well as for distributed processing as utilized in the ALICE HLT. Intermediate data need to be stored in memory, on hard disk, or have to be transferred via network. In any case, data formats must be defined and understood by producers and consumers.

This section investigates different possibilities for data exchange between processes running on different nodes by means of the example of storing track parameters.

4.6.1 C Data Structures

Proprietary Data Structures in C allow the most efficient data exchange due to a minimized overhead. A C-structure is simply a collection of member variables of varying data types. Listing 4.15 gives an example of a structure describing tracks in a specific parametrization. The structure is motivated by the class `AliExternalTrackParam`, base class for tracks parametrization in *AliRoot*, and describes tracks in a local coordinate system rotated by the angle alpha. The track model has 5 parameters (*Y, Z, $1/P_t$, sine of the azimuthal angle and tangent of polar angle*) and thus 15 members in the covariance matrix. The coordinate of the last point assigned to the track is stored as well as an optional index list all points assigned to the track. The data format consists just of an array of such C-structures in memory and a variable containing the number of such structures in the beginning. This approach provides the most effective method to transfer data, it is reduced to

Listing 4.15: The `AliHLTExternalTrackParam` C-structure as used for exchange of track information between HLT components.

```
struct AliHLTExternalTrackParam
{
  Float_t fAlpha;
  Float_t fX;
  Float_t fY;
  Float_t fZ;
  Float_t fLastX;
  Float_t fLastY;
  Float_t fLastZ;
  Float_t fq1Pt;
  Float_t fSinPsi;
  Float_t fTgl;
  Double_t fC[15];
  UInt_t  fNPoints;
  UInt_t  fPointIDs[0];
};
```

the minimal necessary information.

In order to understand the design of cross-platform data exchange approaches, data storage in memory has been introduced in section 3.4.4. Endianness and structure alignment directly influence the computing scheme of the ALICE High-Level Trigger. C/C++ provides different basic data types of varying sizes, e.g. a floating point variable of single precision is 32 bits long, a short integer occupies 16 bits, and a character variable 8 bits. The concept of data types of different sizes is not restricted to the C/C++ language.

Figure 4.20 sketches utilization of a C-structure for data transport between two processes. In this approach, no data format and system information is shipped together with the data. This does not impose a problem on homogeneous processing architectures where every machine uses the same computing architecture. An image of the memory segment used by one component and holding the relevant data can simply be provided to a consumer component, either via shared memory or copying data over network.

4.6.2 ROOT Objects

Appropriate handling of changes in data alignment, endianness and other system properties is in particular important for the output of the HLT, no matter if it are data for monitoring, designated to be stored in the OCDB or reconstructed events to be stored in the data stream. This section describes the approach provided by the ROOT framework.

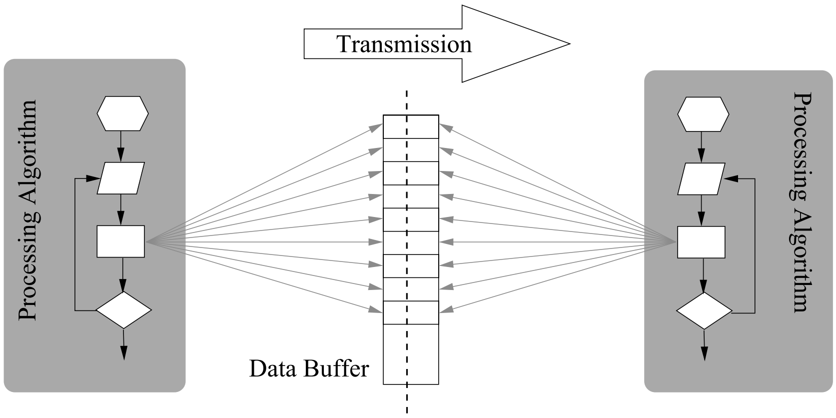


Figure 4.20: C data structures are allocated in memory and can be directly accessed from the algorithm. Shared memory allows data transport to the next algorithm without copying of data.

Whenever a complex object in its representation in memory is going to be saved into a file for later use or sent over network it needs to be translated into a binary format. The translation is carried out using a serialization format. Reading the resulting series of bytes according to the format allows the consumer to restore an exact clone of the original object for its own use.

Serialization of data and transparent storage and transmission of objects is a very effective method of simplifying development. ROOT has an inbuilt serialization mechanism for its objects. The approach is based on the so called *C Interpreter (CINT)* dictionary generator, an automatic code parser and generator of, besides other functionality, object serialization methods. The methods are called *Streamers* and are widely used within the ROOT framework allowing the user cross-platform data storage and transmission. The process is sketched in Figure 4.21. An algorithm cannot directly work on the transmitted data buffer but has internal objects for its calculations. The streamer of the object is called in order to prepare data for transmission. The consumer needs to rebuild a clone object from the serialized information.

However, the versatile and transparent serialization of objects comes at the cost of performance. Data need to be copied several times, bytes swapped, etc, which all consume a share of the available processing time. The transport of ROOT object needs to be evaluated with care for each single application taking into account all constraints in order to find the optimum between performance and uniform data storage.

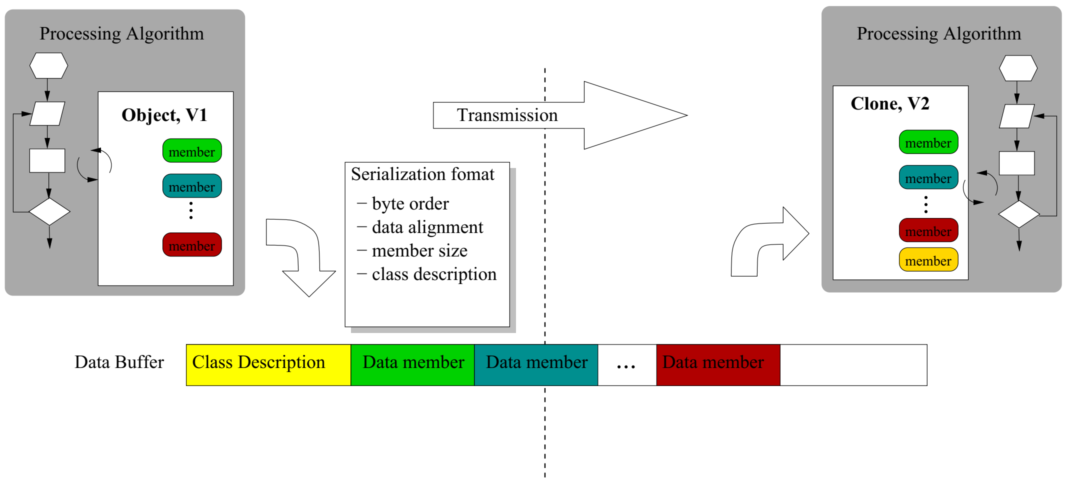


Figure 4.21: Serialization of objects into a buffer. Data members are saved according to a serialization format which determines byte order, alignment and size of data members. If the class description in the consumer application is different, new members are initialized with default values from the class description.

Serialization of Single Objects

As an example, the track parametrization class `AliExternalTrackParam` will be used. This class is commonly used in AliRoot to store reconstructed tracks. It describes tracks in terms of 5 parameters and associated 15 covariance matrix elements (listing 4.16), in total 20 double precision float members. The parametrization is with respect to a starting point and angle.

`AliExternalTrackParam` inherits through `AliVParticle` from `TObject` and gets additional member variables from this base class.

Listing 4.16: The `AliExternalTrackParam` class, used for common track parametrization in AliRoot

```
class AliExternalTrackParam: public AliVParticle {
 public:
  // ....

 private:
  Double32_t    fX;       // X coordinate for the point of parametrization
  Double32_t    fAlpha;   // Local <-->global coor.system rotation angle
  Double32_t    fP[5];    // The track parameters
  Double32_t    fC[15];   // The track parameter covariance matrix
};
```

A double precision float member is represented by 8 bytes in memory. The equivalent C-structure needs $8 * 22 = 176$ bytes to represent this characterization in memory. The 53 bit mantissa allows a precision of approximately 16 decimal digits. Since most of the realistic scenarios are not limited by the nu-

	Memory	Serialization
C-structure double precision	176	176
C-structure single precision	88	88
ROOT object	188	148

Table 4.3: Data sizes of the serialized `AliExternalTrackParam` object.

merical precision but the detector resolution, single precision float values with 4 bytes are sufficient. ROOT uses double precision for variables in memory and automatically reduces to single precision when the object is serialized. Consequently, a reference C-structure would need 88 bytes.

A serialized `AliHLTExternalTrackParam` objects totals 148 bytes as shown in Table 4.3. The overhead of two thirds comes from additional data members in the base class and class information stored as part of the serialization format. In addition, endianness of all members is eventually changed to network byte order, where the most significant byte comes first.

Serialization of Arrays of Objects

ROOT provides several types of lists which all can be used to store and serialize an array of objects. Some of those lists like `TList` and `TObjArray` can be used for objects of different type. This case will not be considered in the course of this section. `TObjArray` describes an array of objects with random access by the array index. The collection basically manages a list of pointers to objects which are generated at varying locations in memory. The memory is not contiguous.

The second possibility for an array of identically objects is provided by the class `TClonesArray`. The collection class is designed to overcome the problem of memory fragmentation by allocating and reserving a certain amount of memory for a limited number of objects. If the number of objects are within the original limit, memory does not need to be allocated on and on, objects are just placed at the locations in the reserved memory.

Serialization of arrays of identical objects benefits from a few optimizations. Class information needs only to be stored once and is valid for all objects. The resulting buffer can be significantly reduced by applying data compression algorithms. The compression is more efficient as the number of objects increase.

ROOT implements a compression technique on the working principle of the *deflate*[7] algorithm. It is based on the search for repetitive patterns. The length of the pattern is determined by the compression level. Higher levels allow better compression ratios, however processing time and memory consumption increase as the length of the pattern increases. Figure 4.22 illustrates the achievable ratios for `TClonesArray` structures of 100 to 10000 `AliExternalTrackParam` elements. The data is compressed to about 5 to 10 % depending on compression level and number of elements in the array. Measurements on the processing performance are presented in section 6.4.

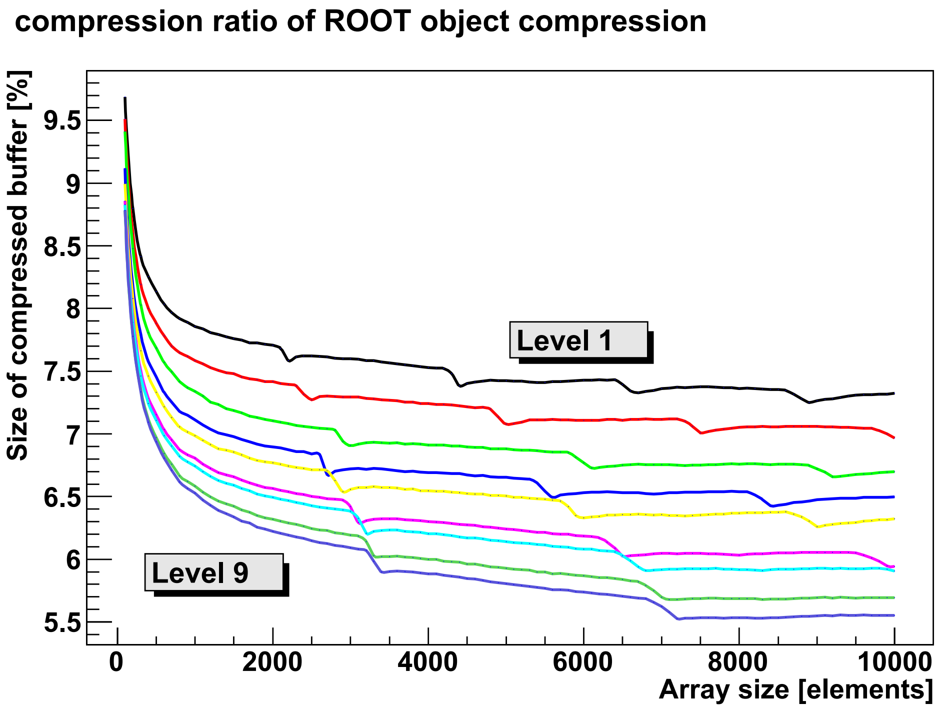


Figure 4.22: Compression ratios achieved with the *deflate* algorithm applied to a `TClonesArray` of `AliExternalTrackParam` objects.

[7]Deflate is a lossless data compression algorithm, a description can be found in e.g. [45]

5. Global HLT data flow and processing scheme

This chapter introduces the work flow of the ALICE HLT during operation of the experiment. Besides the generation of trigger information, HLT also produces data like any other detector. The content of this payload depends on the HLT configuration. Focus will be put on the on-line generation of HLTOUT payload and the subsequent processing during off-line event reconstruction.

5.1 Data Flow during Operation of ALICE

HLT entails reconstructed events at the full data rate. The reconstructed events are stored in ESD format and the basis for on-line physics analysis producing trigger information. However, due to the optimization for high data throughput, content of the ESD differs from the off-line reconstructed ESD. The access to calibration information might be limited for HLT, algorithms are optimized for high speed processing, and event processing is distributed on the level of event fragments.

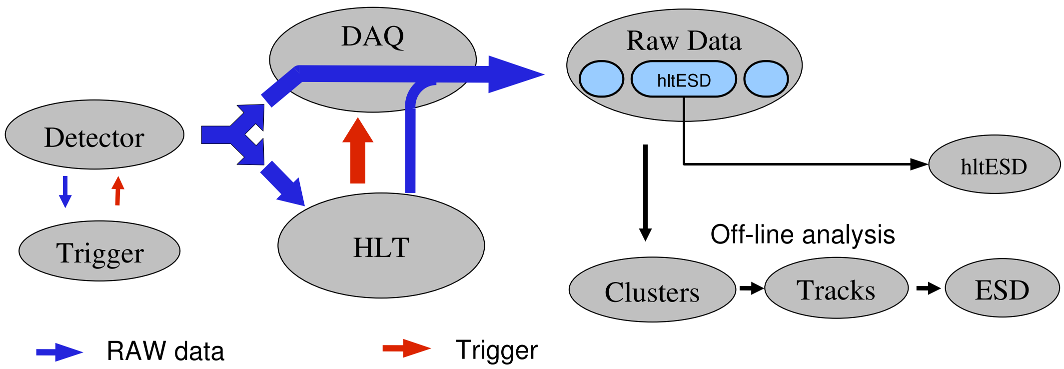


Figure 5.1: Raw data flow during data taking. Beside event selection, HLT stores fully reconstructed events and proprietary data from the reconstruction in the raw data stream.

This makes output of the HLT sufficient for trigger algorithms, but not for sophisticated and accurate off-line analysis. Still, the on-line information is very valuable since statistics is higher by a factor 10 to 20. This makes information from the on-line ESD an excellent tool for rough studies and quick scan through data in order to find interesting aspects in the physics analysis.

Furthermore, the result of the on-line reconstruction and triggering algorithms needs to be stored in order to verify the operation of HLT. Detector algorithms are also allowed to store proprietary data, which are treated during the off-line reconstruction. In either case, the design of the HLT data flow and HLT analysis framework allows for the required flexibility.

The general data flow during operation of the experiment and data taking is described in the corresponding chapter of [46]. The specific data flow for HLT is outlined in Figure 5.1. The data path is split and data are provided to DAQ and HLT in parallel. HLT adds its payload to the raw data stream according to the chain configuration. In any case, a variety of preprocessed data are included in the HLT payload. Preprocessed data like the ESD data set coexists with raw data from detectors in the ALICE raw data stream.

During the off-line event reconstruction, data blocks just need to be extracted from the HLT payload. Some data blocks might be of proprietary nature and need special treatment. HLT analysis framework implements the corresponding plug-in infrastructure. For the end user, the overall data flow appears as a transparent transport layer. This allows to focus on implementation of HLT components and subsequent off-line treatment of their output.

5.2 Generating the Event Summary Data

Event Summary Data is the storage format of the reconstructed event and has been introduced in section 4.5.5. The final ESD information is stored in a ROOT file `AliESDs.root`. The file contains two `TTree` objects, one for the off-line ESD and one for the HLT ESD respectively.

```
root [1] .ls
TFile**          AliESDs.root
 TFile*          AliESDs.root
  KEY: TTree     esdTree;1         Tree with ESD objects
  KEY: TTree     HLTesdTree;1      Tree with HLT ESD objects
```

Figure 5.2 shows the possible reconstruction paths leading to HLT ESD. The path of standard operation is (c), where detector data are processed by the on-line farm and the result is part of the HLT raw data. Options (a) and (b) implement both HLT response for the AliRoot simulation. The post-processing of detector raw data by HLT algorithms and HLT chains is mostly a development feature (d).

It is important to notice that the format of the two data sets `esdTree` and `HLTesdTree` is identical. The content of objects of type `AliESDEvent` is stored in both cases. This uniform treatment of the two data sets allows detector algorithms to transparently fill the relevant field of the ESD. The algorithm does not notice whether it works on an HLT ESD or off-line ESD. This uniformity is especially important for running off-line analysis algorithms on the

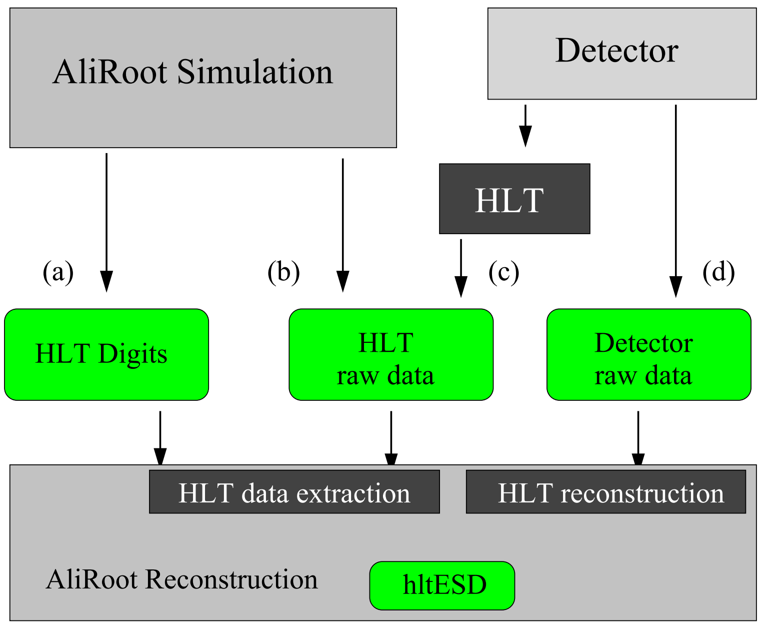


Figure 5.2: Work flow for the generation of Event Summary Data by the HLT.

HLT farm. As mentioned, HLT analysis framework allows not only to run HLT components in either on-line or off-line environment, but also to embed original off-line algorithms into on-line components.

Only at the stage of AliRoot reconstruction on-line and off-line ESD become separated data sets. One more peculiarity effects the handling of HLT ESDs. In contrary to off-line reconstruction where the events of the `esdTree` are filled one after the other, the on-line ESD contains only one event at a time. Also, different sub-chains might fill different members of the ESD. At time of off-line reconstruction, those individual ESDs must be merged and added to the `HLTesdTree`, see section 5.5.

In the AliRoot analysis subsequent to reconstruction, input can be easily switched from off-line to HLT ESD. The same analysis can be applied without any change in source code.

5.3 High-Level Trigger output - HLTOUT

The output of the ALICE High-Level Trigger consists of a trigger decision and generated data. Both data blocks are sent over the HLT output DDL to the DAQs LDCs. HLTOUT data follow the general DDL format. Each DDL block consists of the Common Data Header (CDH) and the DDL payload [32]. Normally, DAQ does not touch the data except for the CDH which is an identifier of the data block within the system and important for redirection of data and steering of the event building. In the case of HLT however, trigger decision needs to be processed by DAQ and makes the definition of an appropriate data format necessary. The HLT output format is defined in [31] and contains the following blocks:

- The Common Data Header is mandatory for all DDL data. For HLT, it also indicates the existence of an optional HLT data block by setting corresponding bits in the CDH status and error bits. The indication of the HLT Decision block is important for the DAQ system to treat trigger decisions.
- A mandatory HLT Event Header contains all HLT-related information concerning the event fragment, mainly a version number and an extended event identification.
- The optional HLT Decision block describes the decision taken by the HLT farm for the given event. The decision consists of a DDL readout list. In mode C, DAQ reads only the DDLs which have been flagged by HLT to be read out.
- HLT optional Payload block contains all data produced by the HLT farm for the given event.

5.3.1 Generation of HLT Output

Dedicated components of the PubSub framework generate the HLT output. Those components are running on the so called HLTOUT nodes of the HLT farm. HLTOUT nodes have H-RORC devices installed which are equipped with the DAQ SIU. In total, 10 DDL links defined for HLT to be read by DAQ. The `HLTOUTFormatter` component at the end of the HLT chain subscribes to all data blocks to be published including the decision data block, which is generated by dedicated `HLTDecision` components. The formatter component prepares the output to be sent and transfers it via shared memory and the PCI-X interface to the H-RORC cards.

5.3.2 Decision List

When running in mode C, DAQ includes original DDLs according to the HLT Decision into the event building. The readout list determines which detector DDLs are stored. This can reach from full rejection of the detector data links and just storing a relatively small event summary for the HLT to full readout of the detectors because of an interesting event.

The HLT Decision List consists of a bit field as described in [31]. The current version defines 30 words of 32-bits for the Decision List. In order to account for extensions, the size of the pattern list is stored in the first word and allows together with the overall format version identifier the preservation of compatibility.

5.3.3 HLT Output Payload

Depending on the running HLT chain configuration, data blocks are produced by HLT components in the last stage of the HLT analysis chain. Examples are ESD data blocks describing a short summary of the reconstructed event, compressed DDL raw data, and proprietary data from the event reconstruction. HLTOUT payload contains a collection of all data blocks in the HOMER format. As described in sec. 3.9.2, HOMER interface aims to provide a system independent interface for shipping data blocks out of the HLT analysis chain. Data blocks are identified by HOMER descriptors specifying data type and specification, system endianness and alignment and other relevant information. It is important to notice that the HOMER interface cannot handle the system dependence of the data blocks itself because the nature of the data is unknown.

For that reason, care has to be taken concerning the format of data blocks, in particular whenever the system architecture of the producer and consumer can be different. This is the case for HLTOUT data blocks which are produced on the HLT farm and analyzed later on another farm or even the Grid. The producer and the consumer have to entirely handle the data and necessary conversions. Methods are provided for the indication of changes in endianness and alignment.

The origin and arrangement of HLT data blocks in HOMER format is illustrated schematically in Figure 5.3.

5.4 HLTOUT processing

5.4.1 Overview

As outlined in section 4.5.4, AliRoot event reconstruction loop consists of three major steps: **(i)** Local event reconstruction - clusterization of the detector raw data, e.g. reconstruction of space points, **(ii)** Tracking, and **(iii)** Filling of ESD.

Some minor steps have been ignored here. Each of the steps is realized as a method of the `AliReconstructor` interface. Detector implementations using this plug-in interface implement the actual processing of one step for a detector.

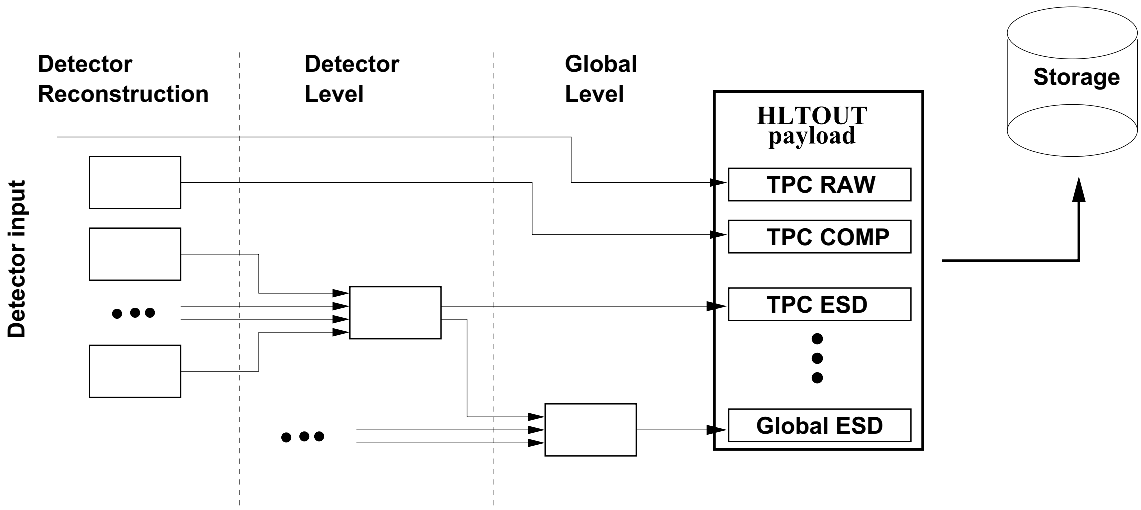


Figure 5.3: Constituents of HLT output payload. Different components connected to HLTOUT contribute to the HLTOUT collection. For simplicity, components are sketched in an example hierarchy. Exemplary content is denoted (i) `TPC RAW` - TPC raw data, (ii) `TPC COMP` - TPC compressed raw data, (iii) `TPC ESD` - ESD of TPC on-line reconstruction, and (iv) `Global ESD` - ESD from global reconstruction.

The modularity of the reconstruction is achieved by common processing of global event properties and the number of plug-ins defined. In this scheme, HLTOUT processing takes formally place in the function `FillESD` of the `AliHLTReconstructor`. As already pointed out, there is no HLT reconstruction to be executed in the normal reconstruction cycle as all HLT reconstruction has been either **(i)** carried out on-line on the HLT farm during acquisition of data, or **(ii)** embedded into AliRoot simulation. In either case, the result is already present and stored as part of the HLTOUT payload.

During AliRoot reconstruction, HLT is the first module to be called. This ensures that HLT data can be provided to other modules during reconstruction.

Data blocks of the HLTOUT payload are of various origins and the processing depends on the knowledge of data format. Thus, for most of the data blocks, specific implementations are required and the developer of a component is engaged to implement also the corresponding data treatment. An effort has been made to collect the "thematically" adjacent source code in one place. A similar modular framework like the component libraries has been developed for HLTOUT data. For each type of data a handler must be available in the system. *HLTOUT handlers* are implemented by the different modules. The system queries module agents whether a handler can be provided for a certain data type by its module. In the processing loop, the `AliHLTModuleAgent` of the module creates the appropriate handler for a data block. Figure 5.4 illustrates the dependencies of a module library and the transparent data flow.

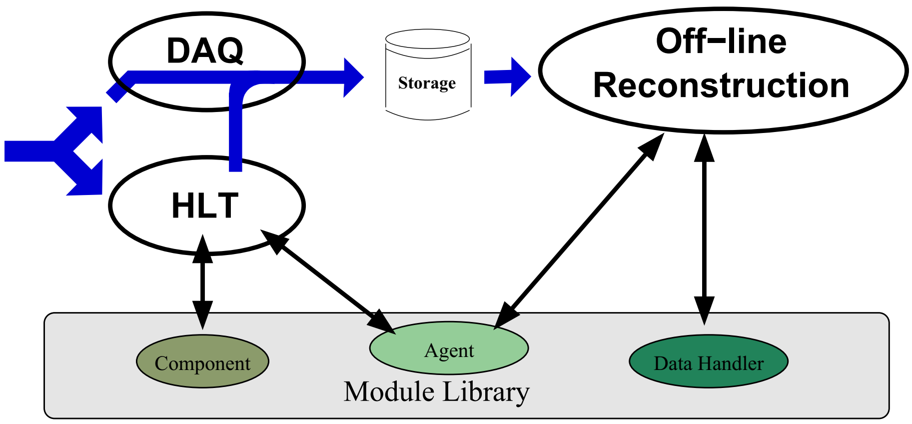


Figure 5.4: The complex data transport is transparent for the developer of an HLT module library.

5.4.2 Classification of HLTOUT data blocks

The data type of the individual blocks is set by the producer components and specifies the nature of the required data processing. `AliHLTOUTHandler` provides the interface for HLTOUT handlers, which consists mainly of one processing function. The output and interpretation of the data depends on the type of the handler. Depending on the data type, common functionality can be classified, e.g. there might be several components producing ESD objects. The hierarchy of default HLTOUT handlers provided by the framework enables the developer to add data treatment for specific data blocks with a minimum of effort. There are 5 groups of HLTOUT handlers, which are described below in more detail.

- `kESD`: Output is in ESD format and can be handled by the framework
- `kRawReader`: Data describes DDL raw format
- `kRawStream`: Pre-analyzed data to be fed into the normal reconstruction
- `kChain`: Data blocks to be processed by an HLT analysis chain
- `kProprietary`: Handler for detector specific data blocks.

The handlers `kRawReader` and `kRawStream` have a special role for data replacement as described in section 5.6. The processing sequence for the other handlers is sketched in Figure 5.5. All `kChain` handlers are processed first. Since those utilize a normal HLT chain, the output is added to the HLTOUT collection. This allows `kChain` handlers to generate an ESD as output and pipe this into the normal ESD processing just following as the next step.

HLTOUT processing is handler-oriented instead of block-oriented. The same handler can be defined for more than one block and is called only once.

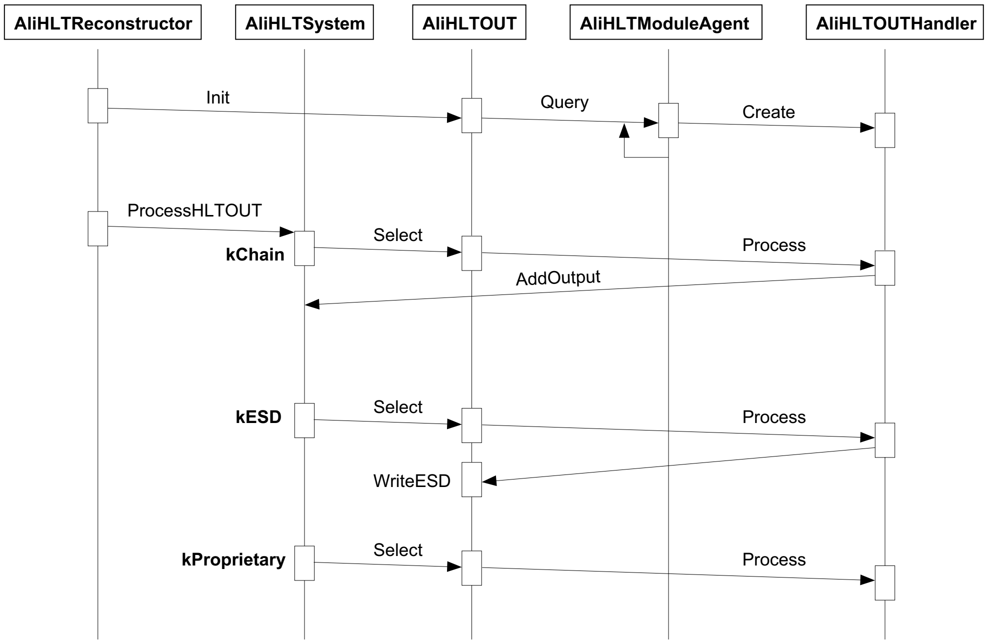


Figure 5.5: Processing sequence HLT output payload. Handlers are created within a modular system and executed in the sequence `kChain` → `kESD` → `kProprietary`. `kChain` handlers are allowed to add new data to the stream and `kESD` handlers can process the data block prior to the writing.

ESD HLTOUT data

The framework provides standard handling of ESD data blocks in order to write entries in the ESD during AliRoot reconstruction. Each ESD block contains data of only one event. This is in contrast to the AliRoot scheme where ESDs of the processed events populate a TTree. The collection of the individual ESDs and merging by a specialized class `AliHLTEsdManager` are the main properties of the `kESD` HLTOUT handler.

It is important to notice that ESDs data blocks do not need an HLTOUT handler. However, an optional handler can be defined for ESD data blocks of a certain data origin and specification. This handler is of type `kESD`. For the generation of ESD data, several use cases apply. Either the producer component in the HLT on-line chain already published the data in ESD format or the handler provides some kind of after burner to pre-processed data and outputs the ESD. In the first case, data blocks have to be published with data type `kAliHLTDataTypeESDTree` {`ESD_TREE:ANY`}. *ANY* denotes any detector origin.

The module agent can provide a handler for multiple ESD data blocks, e.g. for merging within one event prior to the writing. Instead of the individual ESDs, the one provided by the handler is passed to the `AliHLTEsdManager`.

DDL Raw HLTOUT Data

As a common HLT application, a reduced amount of detector raw data can be produced in the original raw DDL format. The difference is the location of data in the DDL stream. Instead of the original detector DDLs, the HLT DDLs now contain detector raw data. Redirection of the input stream to the off-line reconstruction will be described in section 5.6.

Preprocessed Raw HLTOUT data

Handlers of type `kRawStream` are foreseen though at the time of writing the concept is not fixed. Advanced data compression algorithms can produce a raw data format which is not convertible into the raw DDL data, e.g. lossy compression techniques storing clusters parametrized with respect to tracks. A specific RawStream is needed here since the data are detector specific and the first stage of the off-line reconstruction might need some adaptions.

Adaptive compression of TPC data has been investigated ([34]). Off-line reconstruction from such a compressed raw data requires the implementation of a `kRawStream` HLTOUT handler as the original raw data cannot be restored from the compressed data.

HLTOUT Data Processing by a Chain

The motivation for `kChain` HLTOUT handlers is the utilization of normal HLT components for HLTOUT processing. As a matter of fact, an on-line analysis chain can be split at an arbitrary stage, adding all data blocks to HLTOUT. During the off-line reconstruction the remaining part of the analysis chain can be executed within a `kChain` handler. For certain time consuming processes, e.g. the conversion to ESD format, this technique can be an alternative way to meet the on-line performance requirements. Of course this is only possible if the trigger to be produced on-line does not require the data produced within the postponed levels of the analysis chain.

The interface for `kChain` handlers is implemented by an HLT analysis framework class, `AliHLTOUTHandlerChain`, which can be used either as a base class or directly. The behavior is controlled by arguments provided to the constructor. Listing 5.1 illustrates the instantiation of a specific handler as part of the `AliHLTModuleAgent::GetOutputHandler` function. The concept implies definition of the appropriate HLT chain (see section 4.5.2) and the handler must be initialized with the chain id. The `AliHLTOUTHandlerChain` constructor accepts arguments which have been introduced for `AliHLTSystem`.

The chain(s) to be run can be defined with the 'chains=...' argument. In fact, an `AliHLTSystem` instance runs under the hood of a `kChain` handler.

Listing 5.1: Examplary handling of the request for a data handler. A `kChain` handler is set up to process blocks of the histogram data type.

```
// afterburner for some histograms
if (dt==kAliHLTDataTypeHistogram|kAliHLTDataOriginSample) {
  return new AliHLTOUTHandlerChain("chains=SAMPLE-my-histo-converter");
}
```

The configurations can be defined by the module agent as described in Sec. 4.2.4. Alternatively, a class derived from `AliHLTOUTHandlerChain` can overload a virtual function of the base class to specify the configurations.

The HLT chain can either utilize data sink components at the end in order to write all data appropriately or a processor component to publish data blocks at the end of processing. All output blocks produced by the analysis chain are added to the HLTOUT collection and are processed afterwords if handlers are defined. As a consequence, chains run from a `kChain` handler can only produce data blocks handled by `kEsd` and `kProprietary` handlers.

Proprietary HLTOUT data

All data which does not fit into one of the other categories can be processed by handlers of type `kProprietary`. Such handlers are highly detector specific and do not have any standard output to the framework. Data can be processed and stored to files.

5.5 Common Handling of ESD objects

Common handling of HLT ESD data is one of the key-features for the flexible data processing in HLT. As already pointed out, the exact origin of an ESD object is fully transparent for algorithms.

ESD merging is carried out during `AliReconstruction` on the basis of simple merging of disjoint members. The `AliHLTESDManager` merges content of multiple ESDs originating from different HLT chains and stores the result in the `HLTesdTree` as illustrated in Figure 5.6.

Default ESD merging is only possible on the level of ESD members. If two objects have non-overlapping content, e.g. only the *Tracks* member array is filled with entries in one object and only the *MUONTracks* member in the

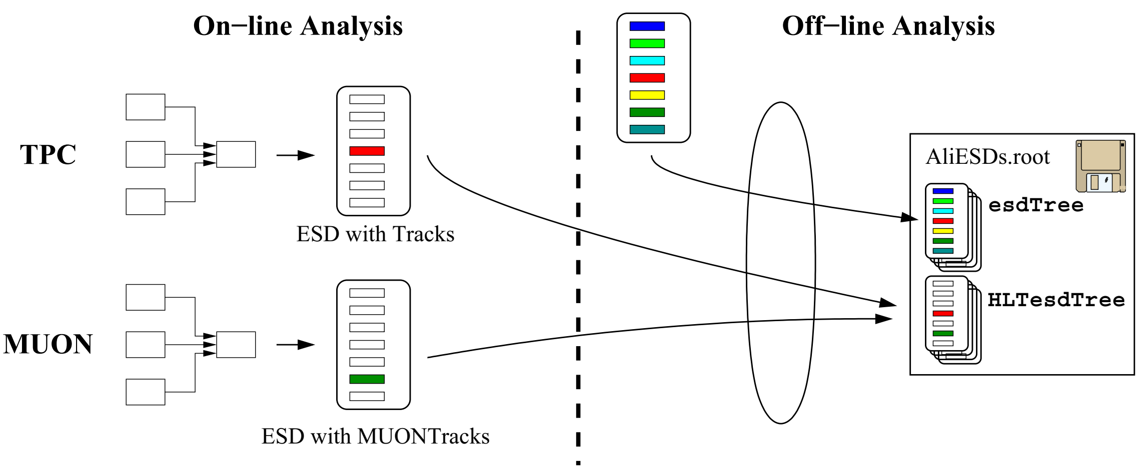


Figure 5.6: Schematic data flow for individual HLT ESD objects.

other object. Content of the same type needs to be *matched*. E.g. reconstructed tracks of the TRD and TPC are stored both in the *Tracks* member. The tracks need to be matched in order to provide a combined tracking information. ESD matching is not part of the standard ESD handling. The task is carried out by a dedicated component and can be executed on-line, or as part of a `kCain` HLTOUT handler.

Dynamic Extension of HLT ESD Content

As outlined in section 4.5.5, the ESD has a fixed content. This is suitable for off-line reconstruction which follows the default work-flow and produces standardized event summary. Output of HLT on-line reconstruction depends on the actual configuration. Furthermore, the result of the on-line analysis has to be available during off-line analysis in order to estimate HLT triggering efficiency. Since the ESD is the only input to off-line analysis, this information needs to be stored in the HLT branch of the ESD.

The required dynamic extension of ESD content has been added to the initialization procedure of `AliReconstruction`. Due to the implementation of the ROOT `TTree` functionality, the layout of the ESD must be created before the first event is stored. The layout of the HLT ESD is determined as part of the on-line analysis and stored into OCDB at end of run. From there it is loaded from during the initialization of reconstruction, the actual data blocks are treated by specific HLTOUT handlers and added to the specific entries of the HLT ESD.

5.6 Data redirection

As an important application of the ALICE High-Level Trigger, data rate can be reduced by replacement of original detector data with pre-processed data from HLT. Example for this technique are selective readout on the level of the DDL detector data and generation of compressed raw data by loss less compression algorithms.

The important fact here is the unchanged off-line reconstruction. Apart from a dedicated input mechanism, the off-line reconstruction can be run without changes from the corresponding data blocks of HLT payload. The input redirection is designed to be completely transparent to the user.

The input to any reconstruction is provided by the `AliRawReader` as introduced in section 4.5.1. The reader class allows transparent access to different data streams. On the side of the reconstruction, a common interface is used to access data. This part is detector independent. The decoding of data on the detector side is done by so called *raw streams* following the stream model suggested by the `AliRawReader`. A *raw stream* implements data decoding for the first step of reconstruction.

Simple Redirection by the RawReader

In the simplest use case, an on-line component decodes data, selects some relevant parts and encodes it back into the original raw data format. The data block looks like it came from the detector, only the information is reduced. Selective readout for the TPC as presented in Sec. 7.4.2 deploys this technique to select only channels contributing to reconstructed clusters or are close by a contributing channel. There is no loss in information and precision since all relevant data are kept. The task of the HLTOUT handler is to retrieve the equipment id from the data type and specification. Eventually, data needs to be decoded in order to provide the original raw data format.

Compression of raw data is a second group of applications making use of data redirection. On-line components compress raw data in order to reduce the data volume. If the compression algorithm is loss-less, original data sets can be restored. To redirect compressed data to the detector reconstruction, the handler must not only calculate the equipment id from data specification but also restore the original data block. The uncompressed data block is redirected as already described.

It is important to notice that all details of the redirection are hidden from the user. The HLT analysis framework makes use of the `AliRawReader` common interface and implements a specific reader to redirect data from HLT out to

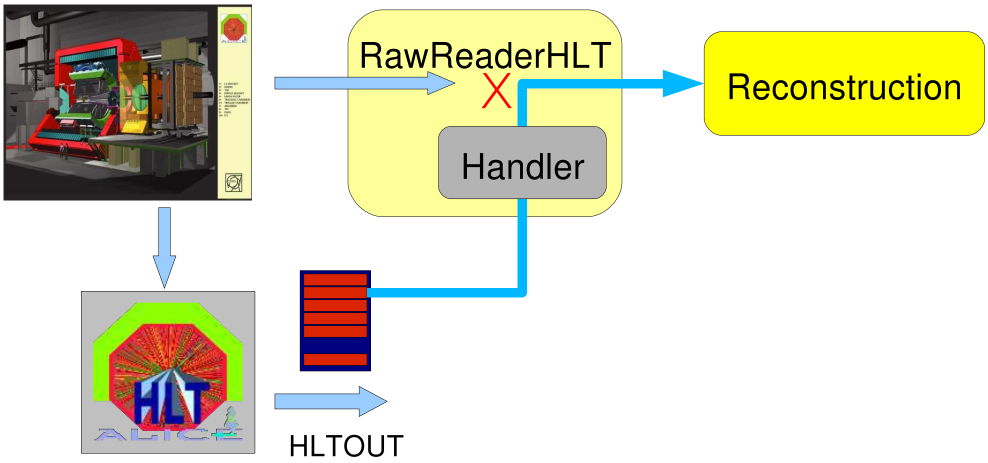


Figure 5.7: HLT data redirection by the `AliRawReaderHLT`. Data of the HLTOUT block is either passed unchanged to the reconstruction or decoded in order to be in the format of detector raw data.

the reconstruction as sketched in Figure 5.7. As an example of an `kRawReader` HLTOUT handler, loss-less Huffman compression has been applied to TPC raw data and implemented for on-line processing ([33]). The original raw data can be restored from the compressed data and no adaption of the off-line reconstruction is necessary.

Implementation of Data Redirection

In order to announce the ability of redirection of a certain data block to the framework, an HLTOUT handler must be implemented and provided to the HLT analysis framework.

- The primary task of HLTOUT handlers of type `kRawReader` is the translation of HLT data specification into the equipment id of the original DDL link.
- An optional task of the handlers is the decoding of the data block and provision of uncompressed data in the original raw format
- The agent of the HLT module must indicate the availability of a handler.

Data blocks in the original raw format and following the general scheme of data identification on HLT (see Sec. 3.4.3) do not need a specific implementation of an HLTOUT handler. The framework provides two classes with the following standard functionality:

`AliHLTOUTHandlerEquId`	Returns data specification of the HLTOUT block as equipment id
`AliHLTOUTHandlerDetectorDDL`	Extracts equipment id from the bit pattern, position of active bit corresponds to id

The first is at the same time the base class for all HLTOUT handlers of type `kRawReader`. All data blocks containing compressed data must implement such a handler in order to provide the uncompressed data to the reconstruction.

The module agent needs to overload two functions of the `AliHLTModuleAgent` base class:

1. `GetHandlerDescription`
   The frame work uses that method to query all agents whether handling of a certain data block is implemented or not.

2. `GetOutputHandler`
   The framework calls the method after the agent announced the availability of an HLTOUT handler. The handler returns an instance of the handler. Multiple data blocks can be served by one handler, also across events.

An implementation of the agent functionality can be seen in the `AliHLTSample` module[1] as presented in listing 5.2.

Data can be accessed by means of `AliRawReaderHLT` if the handler has been implemented. Macro 5.3 illustrates the usage of the reader class although it does not execute any processing. The `AliRawReaderHLT` needs two arguments:

1. `AliRawReaderHLT` does not implement any real data access, only the access to the HLT data links. The original RawReader must be provided in order to access the data.

2. A string identifying the detectors for which data redirection has to be applied.

[1] http://web.ift.uib.no/~kjeks/doc/alice-hlt-current/classAliHLTAgentSample.html

Listing 5.2: HLTOUT functionality of the `AliHLTSample` module

```
int
AliHLTAgentSample::GetHandlerDescription(AliHLTComponentDataType dt,
                                         AliHLTUInt32_t spec,
                                         AliHLTOUTHandlerDesc& desc)
                                         const
{
  // module can handle {DDL_RAW ,SMPL} data blocks
  if (dt==(kAliHLTDataTypeDDLRaw|kAliHLTDataOriginSample)) {
      desc=AliHLTOUTHandlerDesc(kRawReader, dt, GetModuleId());
      return 1;
  }
  return 0;
}

AliHLTOUTHandler*
AliHLTAgentSample::GetOutputHandler(AliHLTComponentDataType dt,
                                    AliHLTUInt32_t /*spec*/)
{
  // create handler for {DDL_RAW ,SMPL} data blocks
  if (dt==(kAliHLTDataTypeDDLRaw|kAliHLTDataOriginSample)) {
    // use the default handler
    return new AliHLTOUTHandlerEquId;
  }
  return NULL;
}
```

Listing 5.3: Example macro using the `AliRawReaderHLT` for access of raw data in the HLTOUT payload. This specific implementation of the `AliRawReader` is hidden by the `AliRawHLTManager::CreateRawReaderHLT`.

```
{
  AliRawReader* orgReader=AliRawReader::Create(...);
  AliRawReader* rawReader=NULL;
  rawReader=AliRawHLTManager::CreateRawReaderHLT(orgReader, "ITSSDD");
  rawReader->Select("ITSSDD");
  int count=0;
  while (rawReader->NextEvent()) {
    cout << "scanning event " << count++ << endl;
    UChar_t* pSrc=NULL;
    while (rawReader->ReadNextData(pSrc)) {
      cout << "  equipment: " << rawReader->GetEquipmentId() << endl;
    }
  }
}
```

For a normal AliRoot reconstruction, all functionality has been implemented into the `AliReconstruction` steering class. Data redirection can be enabled for a single detector or a group of detectors by a simple switch, the function `AliReconstruction::SetUseHLTData` (listing 5.4). It has one single parameter, a string containing a blank-separated list of detector Ids. Detector Id's for use in AliRoot are defined in the class `AliDAQ`.

Listing 5.4: Generalized access to raw data from the HLTOUT payload in the course of `AliReconstruction`. In this example, redirection is enabled for the SDD detector.

```
{
  AliReconstruction rec;
  // ... setup reconstruction

  // set the redirection for ITS SDD
  rec.SetUseHLTData("ITSSDD");

  rec.Run();
}
```

6. Integration of the Analysis Framework

As one achievement of the presented work, the HLT analysis framework has been fully integrated into the on-line HLT system and the AliRoot software environment. Emphasis has been put on the realization of a stable system. In many cases, optimization for high processing rates has been treated with lower priority as also the data rate during the first years of LHC is expected to be reduced. In the course of initial development of the data transportation framework, detailed measurements in particular on the performance of inter-process communication have been accomplished [24]. Further measurements with different chain topologies and the data analysis framework have been started as part of the presented work, though systematic tests have been beyond the scope.

This chapter summarizes the current performance status of the HLT, using benchmark tests of different chain topologies. In particular, the performance of different data transport approaches has been studied to give an advice for the implementation of components.

6.1 Test Suite

In order to test the Data Transport Framework, additional functionality has been added solely for the sake of benchmarking the system. The following requirements have been set up for the test environment:

- Data Load must be simulated. Since real processing components are not appropriate as they introduce further degrees of freedom by carrying out real data processing, dedicated components are required.

- Processing Load must be simulated in a controlled way and as part of the simulation of data load in order to study cross-cut and dependencies.

- Automatic scan within a range of processing and data load.

- Catching of information from the components of the running chain.

- Negligible impact on the overall processing speed.

6.1.1 Data Load Simulation

A dedicated HLT component has been implemented for the sake of performance studies. The component `AliHLTDataGenerator` is an analysis processor and creates block descriptors for fake data blocks of different size without touching or creating physical data. There is no real processing of data, however, the component can introduce processing load within a certain range.

6.1.2 Analysis Component Statistics

The Performance scan relies on statistics of the input and output for each component. The `AliHLTComponent` base class can optionally generate component statistics data blocks. The data block stores information on processing time, input and output size and the level within the hierarchy. The data format is defined by the structure `AliHLTComponentStatistics` as shown in listing 6.1.

Listing 6.1: Format of optional component statistics data blocks

```
struct AliHLTComponentStatistics
{
  AliHLTUInt32_t fStructSize;
  AliHLTUInt32_t fLevel;
  AliHLTUInt32_t fId;
  AliHLTUInt32_t fTime;
  AliHLTUInt32_t fCTime;
  AliHLTUInt32_t fInputBlockCount;
  AliHLTUInt32_t fTotalInputSize;
  AliHLTUInt32_t fOutputBlockCount;
  AliHLTUInt32_t fTotalOutputSize;
};
```

Each component creates a data block of this format and adds it automatically to the output with data type {`COMPSTAT:PRIV`}. If the input contains component statistics blocks those are added to the new block. In order to mark the different levels of the hierarchy, the `fLevel` member of the structure is set to the maximum level of the input blocks plus one. A unique id allows to trace back a certain data block in the final list to a specific component. The id is derived from the component identifier within the analysis chain by application of a CRC algorithm[1]. The algorithm allows to calculate a 32 bit number from a string of arbitrary length with a low probability of identical ids for different strings. Especially if the strings follow a certain naming scheme and differ in just a few letters, the algorithm ensures different ids. The generation of the list of component statistics is illustrated in Figure 6.1.

[1]Cyclic Redundancy Check

During the Start-of-Data event, each component sends its identifier string and component arguments together with the corresponding 32bit id from the CRC calculation in automatically added data blocks of type {`COMPTABL:PRIV`} and format `AliHLTComponentTableEntry`. Furthermore, each component forwards every block of this type from input to output and all blocks are propagated through the chain. Subsequent components can subscribe to the blocks during the SOD event and create a relation between 32 bit id and real component identifier in order to link component statistics data blocks to specific components. The 32bit id has been introduced for performance reasons and for the sake of fixed data length of the component statistics.

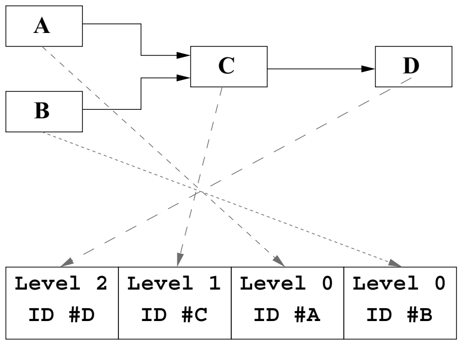


Figure 6.1: Sequence of component statistics data blocks. Each component adds information in front of the information in the input block list.

6.1.3 Collection of Component Statistics

The counterpart of the generation of component statistics information by the `AliHLTComponent` base class is a collector component handling the information at the end of the chain. The `AliHLTCompStatCollector` component subscribes to all data blocks of type {`COMPSTAT:PRIV`} and {`COMPTABL:PRIV`}, and sorts data into appropriate ROOT storage objects. Statistics information from the former blocks is extracted and stored in a `TTree` while the latter data blocks only present during the SOD event are sorted into a `TFolder` structure. The formatted objects are either saved to a ROOT file or are published to the component output. The advantage of the conversion to ROOT objects is the availability of browsing tools within ROOT. E.g. this pre-processed output can be sent from the component via the HOMER interface to a monitoring application. Relations can be easily displayed as illustrated in Figure 6.2. The component is part of the framework utilities in `libAliHLTUtil`.

The only obstacle is imposed by the relatively high computing demands of the pre-processing of component statistics. Filling data into the target tree object in real-time and subsequent object serialization require a high share of computing time. Currently, the `AliHLTCompStatCollector` component lim-

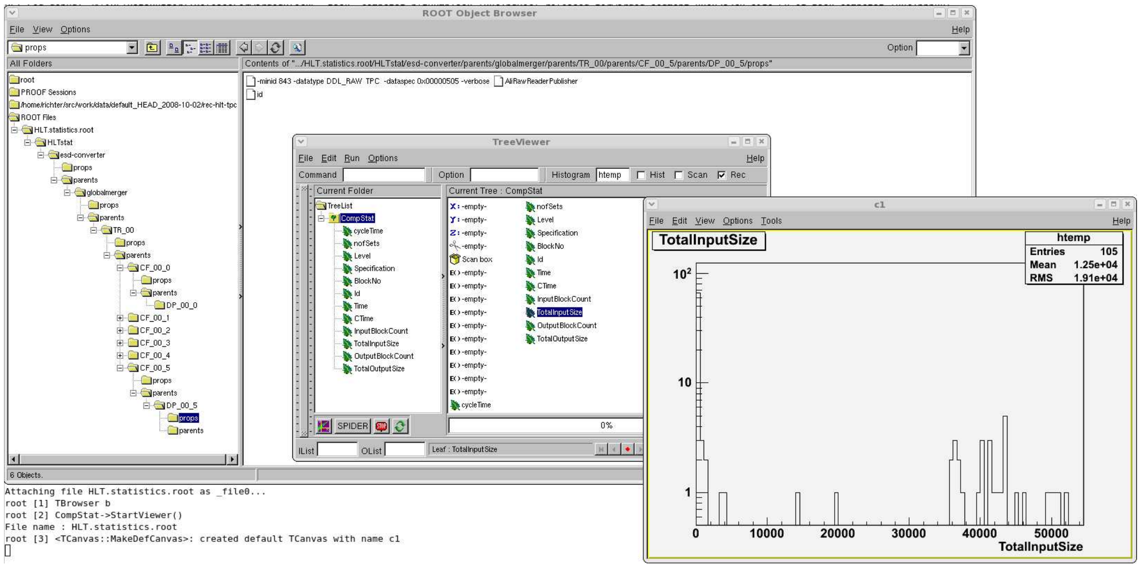


Figure 6.2: The `AliHLTCompStatCollector` component formats component statistics to ROOT objects which can be investigated using the default ROOT graphical user interface.

its processing rate to about 400 Hz when running inside an on-line chain. This makes it inappropriate to run on-line within the normal event reconstruction. However, the statistics data blocks can be written to disk in raw format. Attaching the *FileWriter* component to a chain does not have any significant impact. The raw statistics data can be post-processed by the `AliHLTCompStatCollector` as illustrated in listing section C.1. For the purpose of monitoring, the input rate of the collector can be scaled down.

The next step in the development includes optimization of the ROOT streamer and filling procedure for the particular case of the statistics `TTree` object.

6.1.4 Dummy Data Sources

HLT analysis components are processors in the PubSub framework and, as mentioned in section 3.3.1, cannot serve as data sources in an on-line chain. However, for the presented benchmark tests the data sources are formerly realized by `AliHLTDataGenerator` components in the first level. Consequently, another layer of FilePublishers is necessary to fulfill the PubSub requirement of existence of data sources in each branch. In the test case, the FilePublishers publish one empty DDL just containing the Common Data Header each. The CDH of 32 byte is loaded by the FilePublishers during initialization and this solution does not introduce any performance constraints.

In order to benefit from pipelined data processing it is important to provide a sufficient number of output buffers for all components. The initial dummy

FilePublishers have been configured to operate on 4096 buffers of 64 byte. For the presented tests, the DataGenerator components of the processing levels have been operated on 1024 100 kByte blocks and 4096 25 kByte blocks.

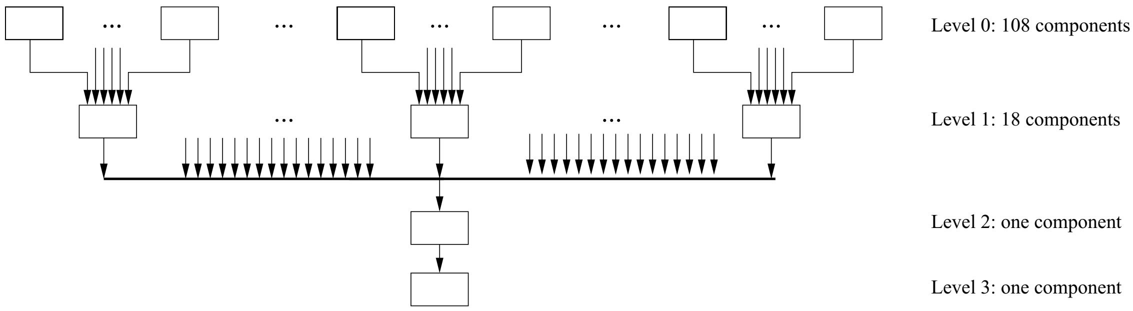
6.2 Data Transport Performance

The overall performance of the on-line HLT has been studied using a 4-level processing hierarchy as outlined in Figure 6.3. In the lowest level, a number of DataGenerator components emulate the generation of output data and publish the corresponding block descriptors. In the next level, 6 publishers from the first level are collected. The output of the level 1 components is collected in one single components in level 2 for topologies (a) to (c). Topology (d) is dedicated for the investigation of the effect of decoupling of branches of the processing hierarchy.

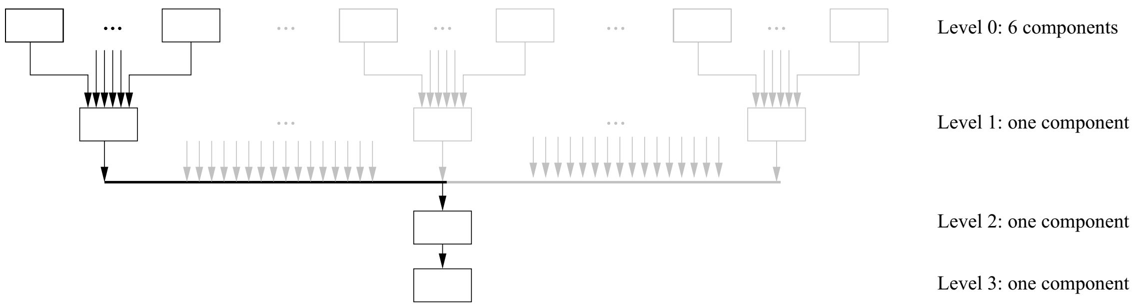
Topology		Data Size [Byte]	Processing rate [Hz]	comments
c)	(1)	10000±500	5285±40	
c)	(1)	5000±500	5580±25	
c)	(1)	1000±500	8835±30	
b)	(6)	10000±500	1840±40	
b)	(6)	5000±500	2310±40	
b)	(6)	1000±500	2385±35	
a)	(18)	10000±500	290±10	
a)	(18)	5000±500	560±15	
a)	(18)	1000±500	1250±40	
d)	(18)	10000±500	340±15	
d)	(18)	5000±500	675±15	
d)	(18)	1000±500	1300±45	

Table 6.1: Performance measurements of HLT hierarchy. Chain topologies are according to Figure 6.4. The *DataGenerator* components publish data blocks of sizes in the specified range.

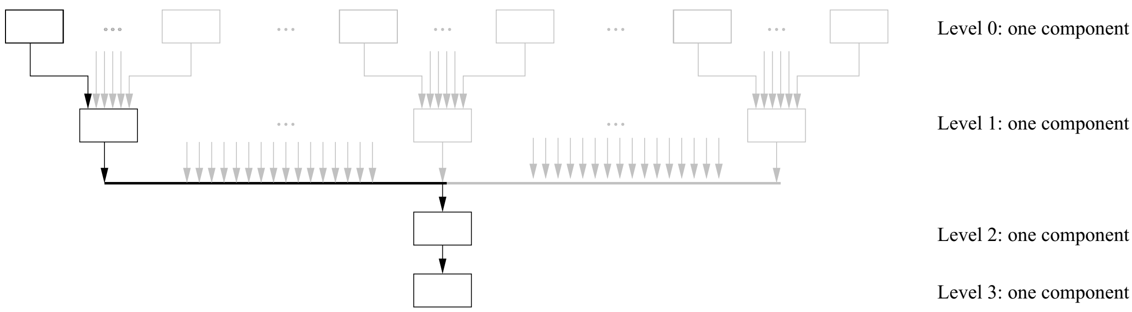
Table 6.1 and Figure 6.3 show the results for 3 different data size ranges. For larger data volumes a drop in the processing rate can be observed especially for setups with 108 publishers in topology (a). The problem occurs especially for topologies with a large data fan-in. Further investigation is needed in order to understand the behavior of the on-line data transport framework and to estimate the potential for optimization. Also the effect of decoupling branches and merging the processing at higher levels is much smaller than expected.



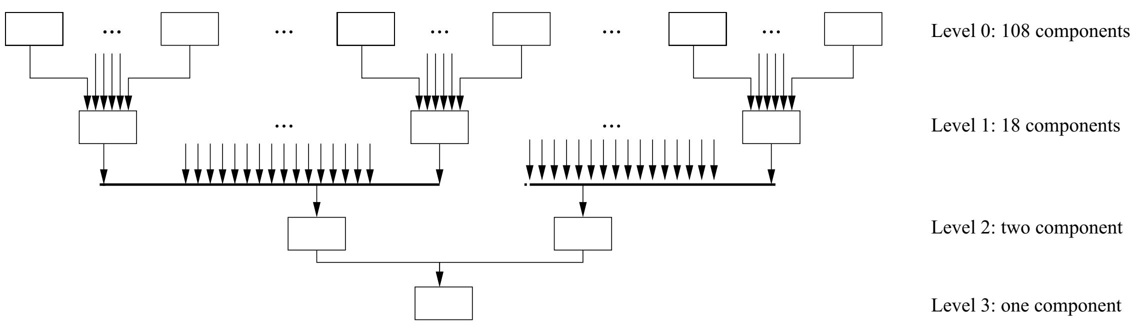
(a) Full chain



(b) One sector



(c) One partition



(d) Full chain, two branches

Figure 6.3: Chain topology of HLT performance test. A 4-level hierarchy motivated by the TPC analysis chain is applied. Topology (a) defines the full setup while (b) and (c) define only a subset. Configuration (d) defines two parallel processing branches. Dummy file publishers as initial data sources are omitted in the figure.

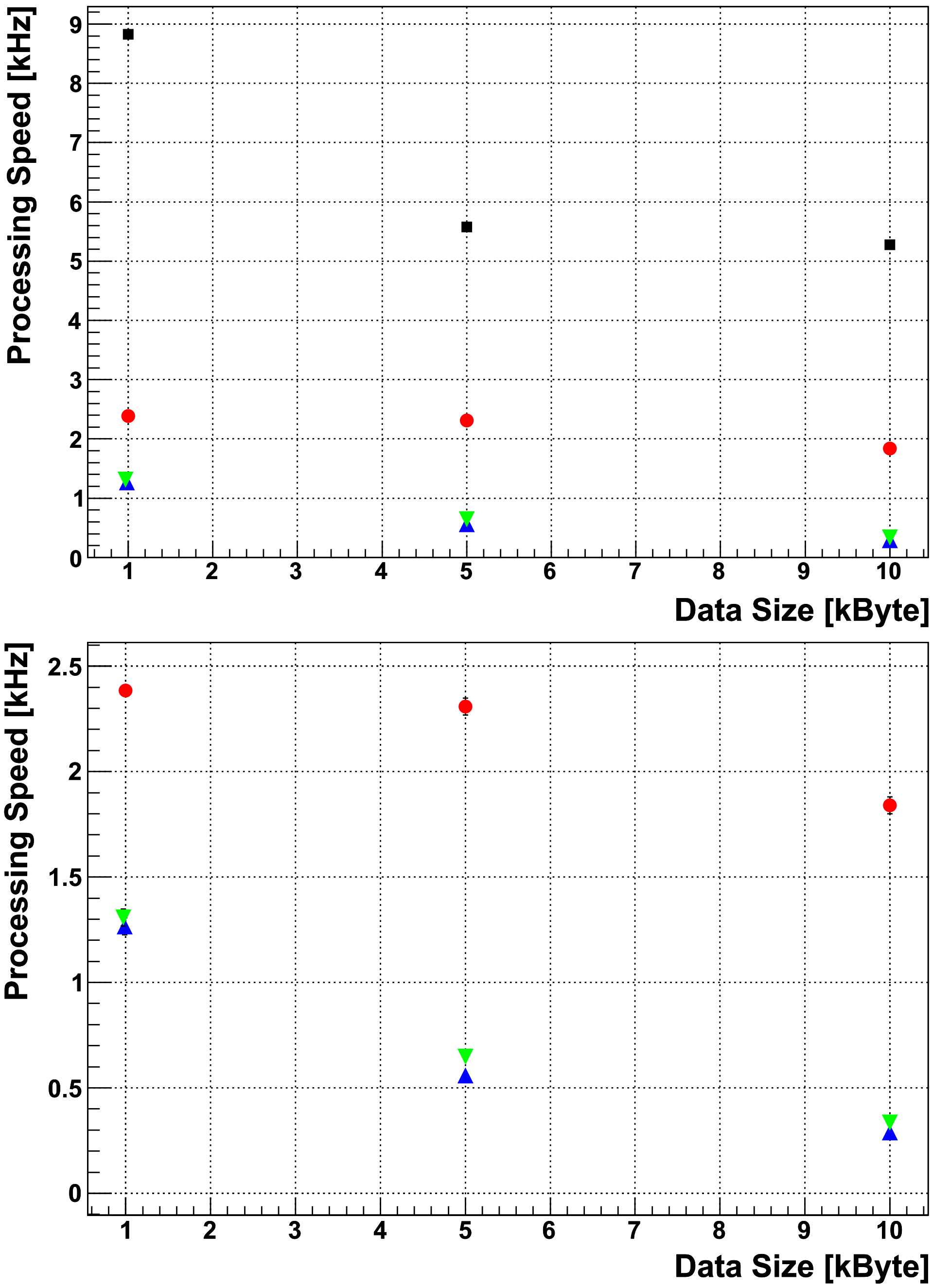


Figure 6.4: Performance of PubSub data transport framework using the HLT hierarchy of Figure 6.3: ▲ a), ● b), ■ c), and ▼ d). .

6.3 Component Fan-In

The HLT hierarchy implies merging of data blocks from multiple parents for the input of another component. This entails also copying of data physically between different nodes. A crucial impact is induced by the merging of input streams for one component. The effect has been studied with a simple topology involving a variable number of *DataGenerator* components on a single node each and one consumer component on a different node. The configuration is shown in the appendix in listing C.2.

The *DataGenerator* components are initialized to publish data of sizes in the range of 10500 ± 500 in order to simulate a moderate data volume which is realistic for many cases. The data transportation system allows processing rates $> 1\ kHz$ for setups with 9 and less publishers as presented in table 6.2. The numbers have been collected using the average processing rates from the HLT status display.

Topology	Data Size [Byte]	Processing rate [Hz]	comments
36 (1) ➡ 1	10500±500	472±12	
27 (1) ➡ 1	10500±500	570±15	
27 (1) ➡ 1	10500±500	572±10	w/o component statistics
18 (1) ➡ 1	10500±500	580±10	
9 (1) ➡ 1	10500±500	1185±20	
4 (1) ➡ 1	10500±500	2636±6	
2 (1) ➡ 1	10500±500	4445±15	
2 (2) ➡ 1	10500±500	3384±40	
1 (1) ➡ 1	10500±500	7040±40	
1 (1) ➡ 1	1000	7580±40	
1 (1) ➡ 1	500	7920±40	

Table 6.2: Performance measurements of component fan-in. Numbers in parentheses denote the number of nodes. The *DataGenerator* components publish data blocks of sizes in the specified range.

In addition, the component statistics collection as described in section 6.1.2 allows investigation of the rate distribution for the individual test cases. The result is shown in Figure 6.5. Plots for 1 and 2 publishers show a broad distribution. An interesting effect can be seen in the measurements for 27 and 36 publishers. Here, a structure with 2 peaks is expressed. While the first peak corresponds to the observed processing rate, the second one which is even more pronounced for 36 publishers is not yet fully understood. As a matter of fact, half of the events arrive at a much higher rate.

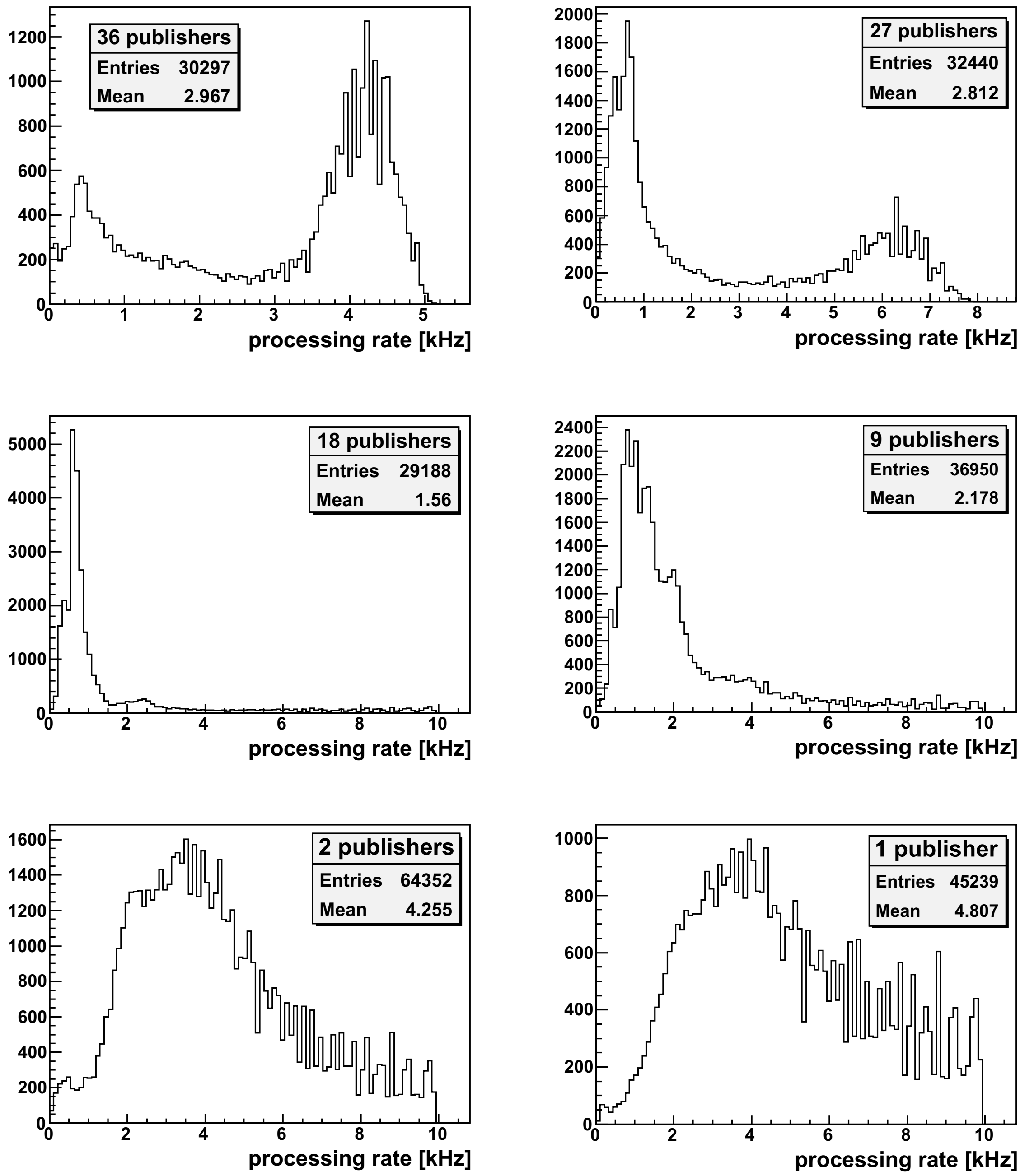


Figure 6.5: Performance impact of component fan-in. The figure shows the distribution of processing rates for topologies with 36, 27, 18, 9, 4, 2, and 1 publisher(s).

Bursts of data can be an explanation, though the minor influence to the overall rate is unclear. Also the reason for the occurrence in topologies with many publishers only is not yet understood. More detailed measurements are underway to study the behavior of the system. These data are also necessary to investigate the potential for optimization.

In order to test the impact of the component statistics functionality, processing rates have been checked for disabled component statistics. It is shown exemplary for the topology containing 27 publishers and does not impose any noticeable effect.

6.4 Transportation of ROOT Objects

Transportation of ROOT objects has been introduced in section 4.6.2. It is the basis for platform-independent storage and transmission of data produced in the HLT and facilitates the implementation of on-line detector calibration and monitoring components.

Two major options for the exchange of a complex data object are deployed in the HLT:

- Serialization using the ROOT streamer functionality allows the most flexible approach. The HLT component interface provides the functionality to send and receive ROOT objects be means of the high-level processing interface (see section 4.3.4).
- Proprietary serialization can be applied by copying the relevant data members to a C-structure and restoring the object from that information on the receiver side.

In order to determine the performance of the serialization of ROOT objects, both approaches have been studied for the `AliExternalTrackParam` object. A specific component, the class `AliHLTBenchExternalTrackComponent` with id *BenchmarkAliExternalTrackParam*, has been implemented which publishes arrays of variable size of randomly filled `AliExternalTrackParam` objects. It furthermore can extract objects of that type from the input stream. Examples on usage of the component in off-line and on-line HLT chains can be found in appendix C.

Figure 6.6 shows the processing time vs. number of elements in the array, measured on an Intel© Pentium© M 2.00 GHz processor. The graphs show a linear raise for all compression levels. The relative processing time and compression ratio for the different levels of the *deflate* algorithm is shown in Figure 6.7. Here, both quantities have been normalized to the compression of level 9. While the achieved compression shows only a small increase for higher levels, the processing time is significantly larger for higher compression levels. This observation suggests a medium compression level and the level 4 is used as default by the component interface.

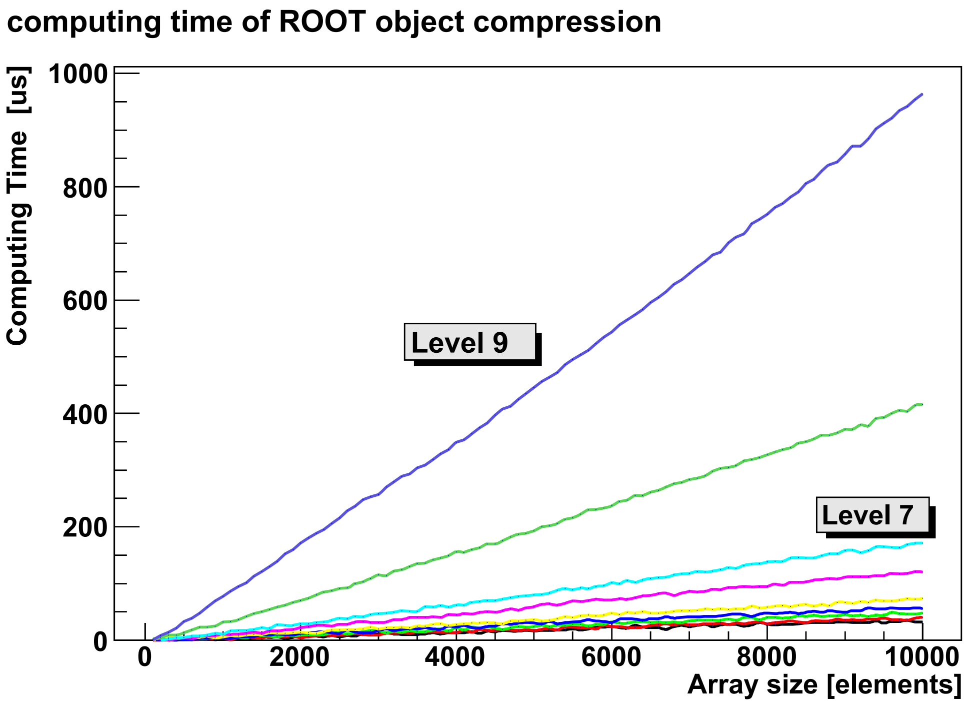


Figure 6.6: Processing time of object serialization for different array sizes and levels of the compression (*deflate*) algorithm.

Finally, benchmark tests have been carried out on the HLT on-line system using one publisher and one subscriber on the same node or two different nodes. The applied configuration is shown in listing C.4 in appendix C. The result is presented in Figure 6.8. The data exchange via a C-structure and copying of members clearly provides the maximum performance.

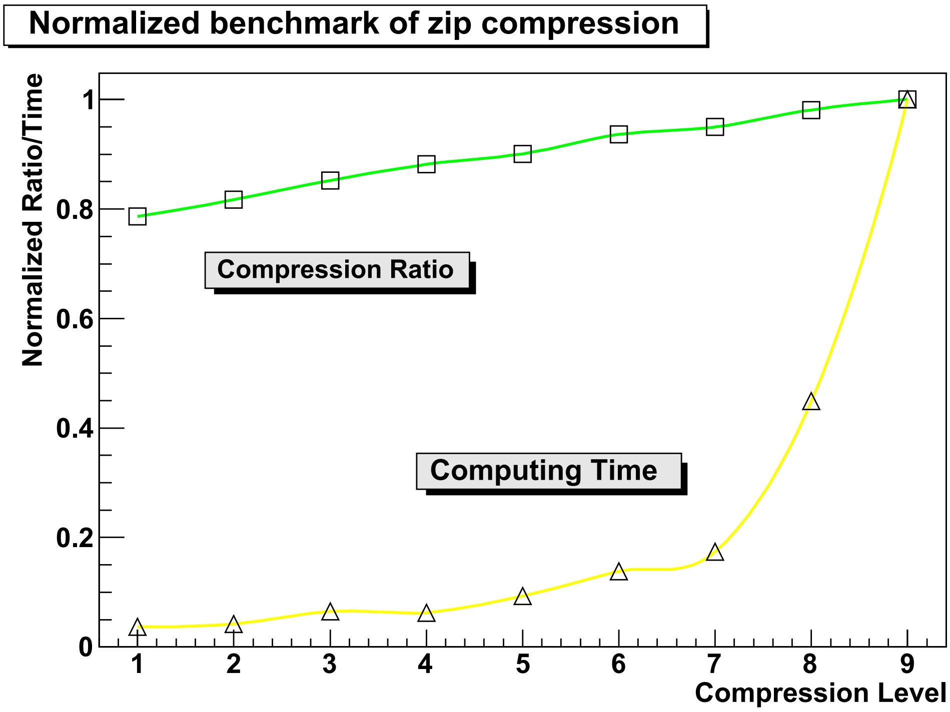


Figure 6.7: Comparison of compression ratio and computing time of object serialization for different levels of the *deflate* compression normalized to compression level 9.

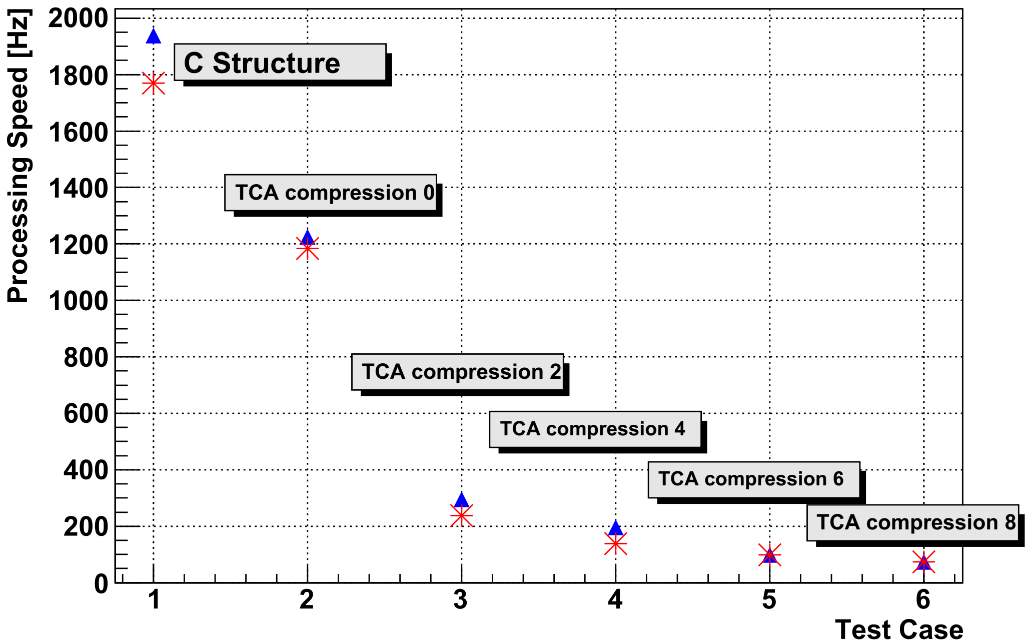


Figure 6.8: Performance of various data exchange approaches. △ exchange between processes on the same node, and $*$ on different nodes.

7. Integration of TPC On-line Analysis

A Time Projection Chamber (TPC) implements the main tracking detector in the central region of the ALICE experiment. Together with ITS and TRD it is designed to provide charged-particle momentum measurements with good two-track separation, particle identification, and vertex determination

The sensitive volume of the TPC is formed by a cylindrical gas volume of 88 m^3 and 5.1 m in length as sketched in Figure 7.1. The inner radius is around 85 cm and the outer radius at around 280 cm. Further details can be found in [47].

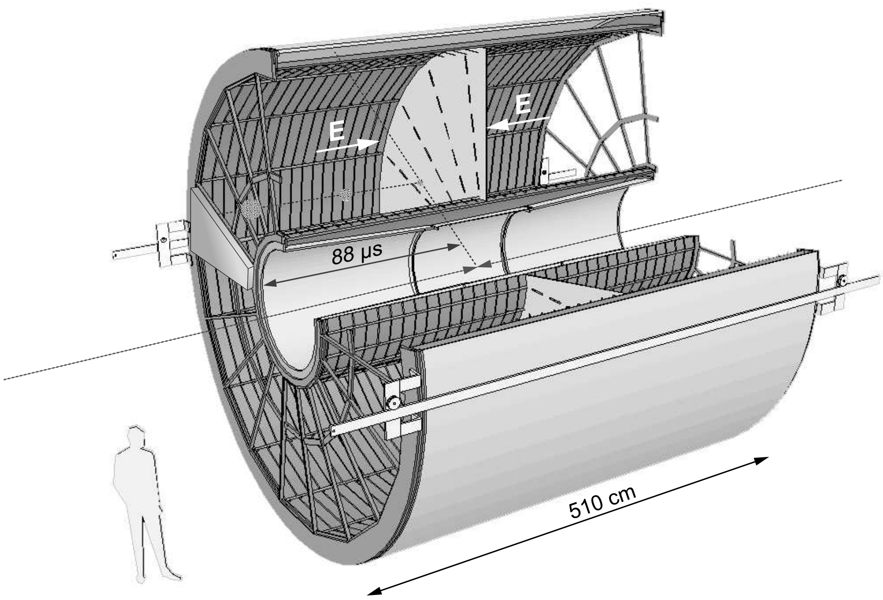


Figure 7.1: The ALICE Time Projection Chamber.

Charged particles originating from the collision traverse the sensitive volume and ionize the gas. The mixture of CO_2, Ne and N_2 is optimized for the required properties. Liberated charges, electrons and ions, drift due to an applied electrical field along the beam line out of the sensitive volume. The two endcaps are equipped with a two-dimensional read-out system provided by Multiwire Proportional Chambers (MWPC) with segmented cathode pads for readout. The charge distribution is sampled over time, and together with the two coordinates this allows 3-dimensional position measurements.

Primary electrons need to be transported over a distance of up to 2.5m in each of the two drift volumes which are divided by the central electrode. The field and gas mixture is adjusted to allow a readout time of 88 μs.

The TPC covers the mid rapidity region with an acceptance of $|\eta| < 0.9$ in pseudo rapidity[1] for tracks with full radial length. In azimuthal direction, TPC acceptance covers the full phase space.

The end plates of the TPC instrument 72 readout chambers. On each side of the drift volume there are 18 Inner and Outer Readout Chambers (IROC/OROC), respectively. The readout is divided into 216 identical subsystems. Each IROC is read out by 2 systems and each OROC by 4 respectively. Those numbers correspond to the composition of the 216 Detector Data Links of the TPC. The TPC Front-End Electronics (FEE) serve in total 557568 readout pads.

The large number of channels together with the sampling frequency of 10 MHz entail a very large data volume of up to 650 MByte per event. In order to cope with the high data rate, several data reduction approaches have been implemented and HLT on-line analysis is an integral part of the solution. The first stage is carried out directly by the TPC FEE. A dedicated readout chip has been developed specifically for the purpose of data readout of the TPC, the ALICE TPC Readout Chip (ALTRO) [48]. Beside the sampling and digitization of the analog signals it incorporates a signal processing unit and has the ability of significant data reduction by applying several stages of filters. The resulting zero suppression reduces the data volume by at least an order of magnitude depending on the nature of the event. For events with very low occupancy data reduction ratio at that level can be several orders of magnitude, for a typical PbPb event the reduction is expected to be approximately 10.

At the stage of HLT, different techniques have been implemented to achieve further reduction. Under normal running conditions, trigger information and compressed data are produced based on the on-line event reconstruction. Furthermore components for zero suppression of ALTRO data in the hLT and for on-line selection of dedicated channels (section 7.4.2) are available.

7.1 TPC on-line Event Reconstruction

As the result of predictions from theory and experimental data from RHIC, central PbPb collisions were expected to produce very high particle multiplicities. The ALICE TPC has been designed for an extreme charge particle multiplicity density of $dNch/d\eta = 8000$, corresponding to about 20000 tracks in the acceptance region of the TPC. In recent years, the expectations have been corrected downwards, still reconstruction of central PbPb events remains a challenge.

[1] $\eta = 0.9$ corresponds to a polar angle of approximately $\theta = 44\circ$.

Initial fast TPC reconstruction algorithms have been adopted and developed in [49]. Prototypes have been implemented and raise expectations for data reduction up to factor 20 by using on-line event reconstruction and subsequent ROI readout or adaptive compression techniques [34, 49]. The recent years have been spent on the full integration of the prototypes.

TPC on-line event reconstruction relies on localization of data processing in order to run processes in parallel. The analysis chain topology follows the granularity of data readout and geometry of the detector as sketched in Figure 7.2. Data are received by 216 DDLs on the HLT FEP nodes, where also the first analysis processes, the *ClusterFinder* components are running. ClusterFinder components reconstruct space point information from raw data, the space points are connected by *Tracker* components individually on the level of TPC slices[2]. Tracks from the different slices are combined on the level of the *Track Merger* and finally converted to a format appropriate for storage.

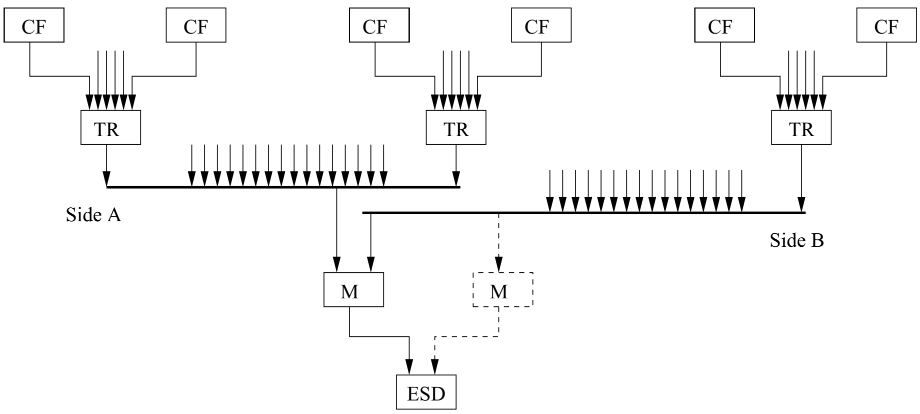


Figure 7.2: Topology of the TPC on-line data analysis chain. The reconstruction includes ClusterFinder components (CF), Tracking components (TR), track merging components (M), and the final conversion to Event Summary Data (ESD).

As the performance of HLT analysis depends on the data volume to be transported between components and computing nodes, data sizes play an important role for the choice of topology. The fan-in of a component collecting data from many parent components has a significant impact on the processing rate as shown in section 6.

The original fast TPC tracking approach is based on conformal mapping and has been discussed in [49]. Recently, development has focused on a Cellular Automaton tracking algorithm as presented in [50]. The concept has been extended and implemented for fast tracking in the ALICE TPC [51]. Both trackers are fully integrated into HLT analysis components. There have been

[2]In TPC notation, both Inner and Outer Readout Chambers are referred to be sectors, the combination of IROC and OROC forms a slice of the same azimuthal position.

also studies on Hough Transform tracking approaches [49, 52]. However, those have not been integrated so far into the on-line HLT.

Due to the relatively large input data volume of the ClusterFinder processes, emphasis has been put on efficient access and decoding of the data stream.

7.2 TPC Raw Data

The raw data format of the TPC is defined by the ALTRO chip. This chip is as already mentioned a dedicated development for the ALICE Time Projection chamber and combines a 10bit ADC converter with additional signal processing units for each channel. The channels are connected to the readout pads of the TPC front-end and sample the collected charge over time. Both for signal and timebin, a bit width of 10 has been chosen. This has influence on the raw data format since all modern computer architectures store data in multiples of bytes. Using a 16 bit word (2 byte) to store a 10 bit word leads to an enormous increase of the data volume as all upper bits are not used for data storage. For the sake of effectiveness, a compressed data format has been chosen where 4 10bit words form a so called 40bit ALTRO word which is stored in 5 bytes. Each TPC readout partition is served by an RCU, a specific hardware device developed for the ALICE TPC ([53]). The RCU assembles the data from up to 200 ALTRO chips and adds a specific data block with additional information regarding the partition at the end of the ALTRO data. The TPC Front-End Electronics skips any sorting of the data and just writes data of all channels sequentially forming a back linked data block as shown in Figure 7.3.

A channel trailer at the end contains the relevant channel meta information. The hardware address uniquely specifies the address of a channel in the set of Front-End-Cards (FEC), ALTRO chips and ALTRO channels. Each ALTRO channel has the ability of significant data size reduction by applying a zero-suppression algorithm. Only the signal passing the suppression is shipped with the data stream. As a consequence, timewise there might be gaps of variable length between the so called *data bunches*. Each bunch is determined by the bunch length, the end time and the signals.

The whole data block is terminated by the RCU trailer of variable length. It contains ALTRO payload specifications and common configuration parameters of the front-end electronics. Due to the back-linked nature of the data block, reading starts from the end.

The ALTRO format has recently been under revision and will probably be changed in order to allow further consistency checks and better handling of corrupted data.

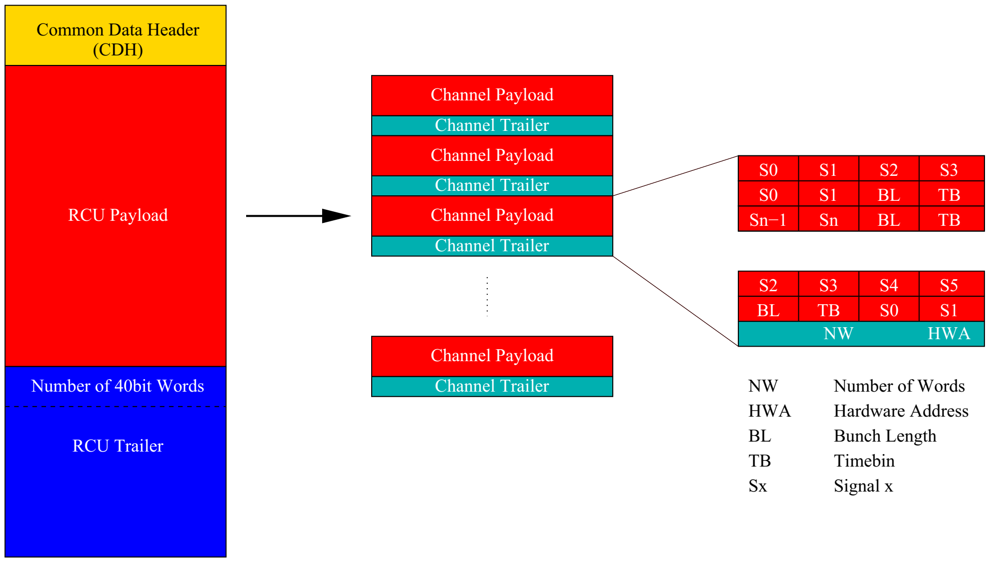


Figure 7.3: The ALTRO format consists of (i) the Common Data Header, (iii) RCU Payload containing the ALTRO channels, and (iii) the RCU trailer.

7.2.1 TPC Data decoding and processing

Because of the 10bit nature of ALTRO data, some pre-processing is needed before the actual processing can start. Bit shift operations are used to convert values into 16/32bit numbers which can be used by the algorithm. Two different techniques have been provided by two different working groups in ALICE. The default off-line reconstruction utilizes a value by value stream model. The approach has been standardized for all detector reconstruction.

Performance constraints have triggered a second development by the PHOS HLT group. The decoder approach applies monolithic treatment of the encoded ALTRO data and provides data in decoded format.

In the course of this work, both techniques have been adopted to the data input of TPC on-line reconstruction.

AliRoot RawReader and TPC RawStream

AliRoot's general interface to raw data are provided by the `AliRawReader` class. Several implementations are available in order to read different sources of raw data through all the same interface (section 4.5.1). The design of the RawReader suggests a stream model for the data decoding. In this model, data processing steps through the data points sequentially. Each data point is characterized by the three coordinates *channel address*, *time bin* and *signal*. Those three coordinates are provided by functions of the RawStream class, a function *Next()* is used to increment the position in the data stream. The standard reader class for ALTRO raw data `AliAltroRawStream` follows that approach as outlined in listing 7.1.

Listing 7.1: Access of TPC raw data though the `AliAltroRawStream`.

```
AliAltroRawStream stream;
// set up stream
// ....

while (stream.Next()) {
  int signal=stream.GetSignal();
  int time=stream.GetTime();
  int channel=stream.GetChannel();
  // data processing
  // ...
}
```

This code design provides a very easy and consistent way of reading data but introduces some overhead. In particular it is necessary to check for each data point whether it still belongs to the same channel and bunch.

Fast ALTRO data decoder

A fast decoder, `AliAltroDecoder`, has been developed by the PHOS HLT group in conjunction with the HLT on-line reconstruction to provide fast access to the ALTRO raw data[3].

This approach benefits from the ability of arbitrary data access within one bunch and the knowledge of bunch and channel length prior to the processing loop. Signal values can be accessed by pointer operations with respect to a buffer rather than function calls. In addition, it fits exactly the channel/bunch structure of the TPC raw data. Listing 7.2 sketches the access sequence which is based on three nested loops on the level of channels, bunches and signals within bunches.

Implementation of Data Access

In order to keep ClusterFinder implementation and any other TPC component working on raw data independent of the input method, an abstract interface has been introduced. The so called *DigitReader* interface hides details of and unifies data access for TPC components and is implemented in the class `AliHLTTPCDigitReader`.

[3]Utilization of the ALTRO chip and the TPC readout system is not limited to TPC and is also used in other ALICE sub-detectors like the Photon Spectrometer PHOS

Listing 7.2: Access of TPC raw data by means of the `AliAltroDecoder`.

```
AliAltroDecoder decoder;
// set up decoder
// ....

AliAltroData channel;
AliAltroBunch bunch;
while (decoder.NextChannel(&channel)) {
  int channel.GetChannel();
  while (channel.NextBunch(&bunch)) {
    int time=bunch.GetEndTimeBin();
    UInt_t* pData=bunch.GetData();
    for (int bin=bunch.GetBunchSize()-1; bin>0;) {
      bin--;
      // data processing
      // signal=pData[bin]
      // ...
    }
  }
}
```

Currently, three maintained implementations are available:

(i) `AliHLTTPCDigitReaderPacked`
implements the access via the off-line stream model,
(ii) `AliHLTTPCDigitReaderDecoder`
utilizes the `AliAltroDecoder` approach, and the
(iii) `AliHLTDigitReaderUnpacked`
is an interface implementation especially used for data transport during AliRoot simulation.

It is important to notice that no change in source code or recompilation is needed in order to run HLT TPC reconstruction algorithms embedded in AliRoot simulation. The abstract interface (`AliHLTTPCDigitReader`) and dedicated implementation (`AliHLTTPCDigitReaderUnpacked`) allow for fully transparent execution of the reconstruction algorithms. The first two versions are both relevant for TPC on-line analysis, data access is sketched in Figure 7.4.

The DigitReader interface defines the two data access paradigms as motivated by the stream and channel/bunch model. The access sequence is outlined in Figure 7.5. The advantage of the latter is the 'predictive' work flow which allows optimized processing loops.

Since both `AliHLTTPCDigitReaderPacked` and `AliHLTTPCDigitReaderDecoder` favor only one of the access methods respectively, support for the other technique has been added (Listing 7.3).

Data access based on the channel/bunch paradigm has also been implemented in the `AliHLTTPCDigitReaderPacked`. It requires anticipated reading and internal buffering of data and comes with some processing overhead.

Listing 7.3: The TPC DigitReader interface methods.

```
// interface methods of the stream model
bool Next();        // Set the reader position to the next value.
int GetRow();       // Get the row number of the current value.
int GetPad();       // Get the pad number of the current value.
int GetSignal();    // Get the current ADC value.
int GetTime();      // Get the time bin of the current value.

// interface methods of the channel/bunch model
bool NextChannel(); // Set stream position to the next Pad
                    // (ALTRO channel).
int NextBunch();    // Set stream to the next ALTRO bunch within
                    // the current pad.
const UInt_t* GetSignals(); // Get pointer to the the current ADC value.
```

Investigation culminated in the establishment of a working group with focus on merging the different data treatment approaches into one officially maintained decoder.

7.3 TPC on-line Data Transport and Specification

TPC on-line reconstruction defines proprietary data structures for data exchange between components. The relevant information is stored in C-structures to achieve the highest possible performance. The final output is supposed to be independent of architecture and a ROOT object is an appropriate choice. The information of the reconstructed event is converted to ESD format or alternatively to another track parametrization supported by AliRoot.

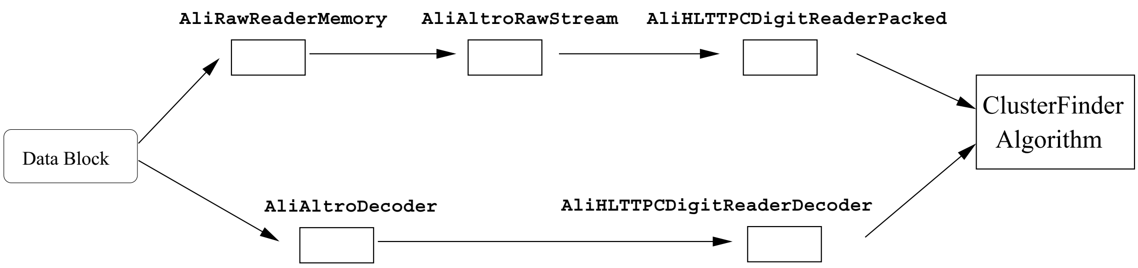
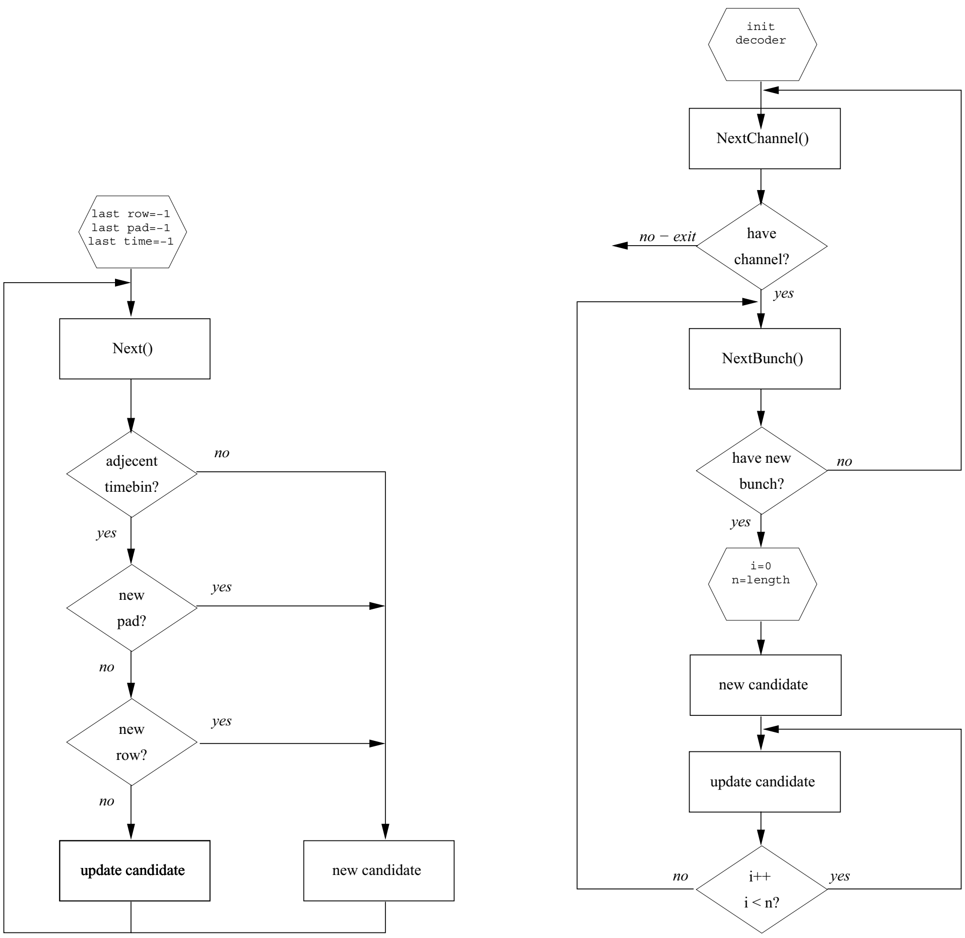


Figure 7.4: Involved classes for data access by means of `AliHLTTPCDigitReaderPacked/Decoder`.



(a) Packed

(b) Decoder

Figure 7.5: Work flow of the two TPC raw data access paradigms. (a) stream model motivated by off-line analysis, (b) channel/bunch access supported by ÂliAltroDecoder.

7.3.1 Data Types

The HLT data types used internally in TPC on-line reconstruction are defined in the class `AliHLTTPCDefinitions` and summarized in Table 7.1.

'CLUSTERS'	fgkClustersDataType	Clusters (Space points)
'TRAKSEGS'	fgkTrackSegmentsDataType	Track segments in local coordinates
'TRACKS '	fgkTracksDataType	Tracks in global coordinates

Table 7.1: HLT data types defined for TPC on-line analysis data exchange. All data types are of origin `kAliHLTDataOriginTPC` ('TPC ') and members of class `AliHLTTPCDefinitions`.

7.3.2 Cluster Data

The data format is defined by the structures `AliHLTTPCSpacePointData` and `AliHLTTPCClusterData`. The latter defines the format of the exchanged data blocks and consists of a *size* member specifying the number of structures and the array of `AliHLTTPCSpacePointData` structures as presented in listing 7.4

Listing 7.4: `AliHLTTPCSpacePointData` and `AliHLTTPCClusterData` - Data structures for exchange of Cluster/Space point data between ClusterFinder and Tracker

```
struct AliHLTTPCSpacePointData {
  Float_t fX;         // X coordinate in local coordinates
  Float_t fY;         // Y coordinate in local coordinates
  Float_t fZ;         // Z coordinate in local coordinates
  UInt_t fID;         // contains slice patch and number
  UChar_t fPadRow;    // Pad row number
  Float_t fSigmaY2;   // error (former width) of the clusters
  Float_t fSigmaZ2;   // error (former width) of the clusters
  UInt_t fCharge;     // total charge of cluster
  UInt_t fQMax;       // QMax of cluster
  Bool_t fUsed;       // only used in AliHLTTPCDisplay
  Int_t fTrackN;      // only used in AliHLTTPCDisplay
};

struct AliHLTTPCClusterData {
  AliHLTUInt32_t fSpacePointCnt;
  AliHLTTPCSpacePointData fSpacePoints[];
};
```

7.3.3 Track Data

Data blocks of type {TRAKSEGS:TPC} and {TRACKS :TPC} use an identical format to exchange track data (listing 7.5). Coordinates are either in local or global coordinate system respectively. The local coordinate system is with respect to the global system rotated by some azimuthal angle in order to fit the center of a TPC slice. Data blocks require therefore the data specification to be correctly set in order to determine the relative rotation.

Listing 7.5: AliHLTTPCTrackSegmentData - Data structure for exchange of track information. The data type determines whether description is in local or global coordinates

```
struct AliHLTTPCTrackSegmentData
{
  Float_t fX;
  Float_t fY;
  Float_t fZ;
  Float_t fLastX;
  Float_t fLastY;
  Float_t fLastZ;
  Double_t fPt;
  Double_t fPsi;
  Double_t fTgl;
  Double_t fY0err;
  Double_t fZ0err;
  Double_t fPterr;
  Double_t fPsierr;
  Double_t fTglerr;
  Int_t fCharge;
  UInt_t  fNPoints;
  UInt_t  fPointIDs[0];
};
```

7.3.4 Data Specification

HLT Data specification is used to identify data blocks of the same type but different position in the analysis chain. E.g. each data input DDL corresponds to a ClusterFinder process and there are consequently 216 data blocks of type {CLUSTERS:TPC }. In contrast to other detectors where a bit pattern can be used to identify DDLs in a 32bit word, for the TPC ranges of links can be specified. The four bytes of the specification correspond in ascending order starting from the least significant byte to (i) minimum partition, (ii) maximum partition, (iii) minimum slice, and (iv) maximum slice. E.g. specification 0x05050303 denotes slice number 5 and partition number 3.

By means of data specification, the ClusterFinder can identify the DDL equipment id in order to load the configuration and apply the correct transformation to space point coordinates. Track data blocks of type {`TRAKSEGS:TPC` } are in local coordinates and must be transformed to global coordinates by merger components. Data specification is also crucial for data compression components. As the data stream needs to be redirected to the off-line reconstruction (see section 5.6), the data specification must enable the corresponding `AliHLTOUTHandler` to derive the original equipment id.

7.4 Further TPC On-line Applications

HLT has been used as a supporting facility for the commissioning of the TPC. Two applications are described below.

7.4.1 Zero Suppression

As already introduced, this very effective technique is implemented in the front-end electronics and reduces the data volume directly at the source of the data stream.

Alternatively, zero suppression can be done on the High-Level Trigger by using a dedicated component. The output of this component is again the ALTRO format and exactly compatible to the original data which allows to run the normal off-line reconstruction algorithm without changes, except the data input. Instead of the original TPC DDL links which even might have been suppressed, data from the HLT DDL links must be used for reconstruction.

Zero suppression can be carried out on the HLT on-line system mainly for the purpose of calibration studies of the detector. In order to calculate filtering parameters for the embedded signal processing of each ALTRO channel, unfiltered data are read out, but processed by HLT in order to filter out the relevant data and save storage space. Also this allows a higher data rate as DAQ is not busy treating the large data volume.

7.4.2 Selective Channel Readout

Zero suppression is not always the appropriate technique, especially for the calibration phase. Since hardware zero suppression is not a loss less process, it cannot be applied before the behavior of the detector has been studied and the parameters have been determined. Here, the overhead is often biggest since just a few tracks are used to study properties like the ion tail. On the other hand exactly those calibration tasks rely on relatively high statistics ($>$ 10000 events [54]).

Out of this situation, an Active Channel Selection procedure has been developed on HLT as outlined in Figure 7.6. From the raw data analysis and cluster reconstruction it is possible to tag the active channels which contribute to clusters. A dedicated HLT component filters the raw data, discards all inactive channels and keeps the active ones unchanged. It is important to notice that the data of the channels is not changed at all, just copied to the new data block. All meta and size information is corrected according to the new channel sequence. This technique is loss less within the scope of interest.

Dedicated runs have been taken in autumn 2008 in order to determine the coefficients for the ALTRO tail cancellation filters. In run 62534, 8200 events have been collected with a total data volume of $\approx 50\ GByte$. Average event size was $\approx 7\ MByte$. The non-zero suppressed TPC raw data would have been $\approx 70\ MByte$/event. The introduced HLT filter represents a possibility for effective data collection in this particular application. Also the subsequent analysis is much faster due to the reduced data volume to be processed. The TPC group will continue using the selective channel readout approach in order to accomplish the calibration tasks.

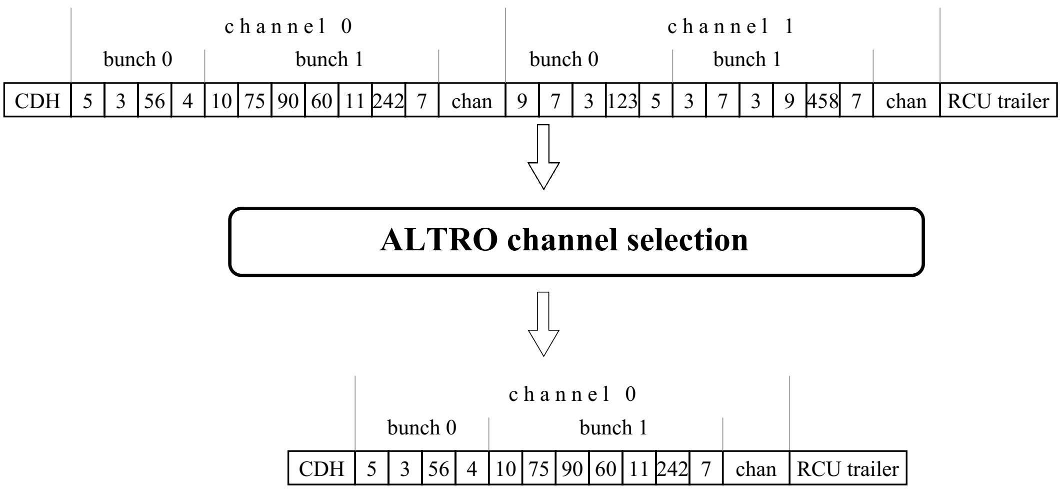


Figure 7.6: Working principle of Selective ALTRO Channel Readout. In this particular example, channels are discarded if the signal is below the threshold of 10 for all bunches. Please note that a bunch has to be read in reverse order and consists of (i) length, (ii) timebin, and (iii) the actual signal values. Channel 0 is kept because of bunch 1 exceeding the threshold. The two bunches of channel 2 both have signals below 10 and the channel is discarded.

8. Conclusion and Outlook

The ALICE High-Level Trigger provides a powerful computing facility for the purpose of on-line data analysis. It allows event reconstruction at the full data rate, the generation of trigger signals with respect to the studied physics, as well as on-line reduction of data rate by application of compression techniques and partial detector readout. HLT takes advantage of the three paradigms **(i)** task distribution within one event, **(ii)** pipelined processing, and **(iii)** optimized data exchange and inter-process communication keeping a low profile in the overall processing performance.

The presented work has studied the necessity of modular software and system design for HLT on-line data analysis. Based on polymorphism in object oriented programming, a framework for HLT analysis components has been developed, implemented and commissioned. The work has directly influenced the overall data processing scheme of the High-Level Trigger system. A complete software solution for HLT on-line analysis based on the existing data transport framework has been developed, including the framework for analysis components, flexible treatment of HLT output payload during the off-line event reconstruction and integration into the ALICE software framework.

Achievements of this work are in particular:

- The HLT analysis framework.
  The framework has been developed as an integral part of HLT. It represents one module along with other sub-systems like e.g. the data transport framework, the run control system, and the computing cluster management. As an outcome of this work, data transport and data analysis have become individual projects. The two projects can now be developed without interference and use separate source code repositories.

- Encapsulation of HLT analysis.
  A major part of the work has been spent on the development of an appropriate modular software design with the result of a self-consistent HLT analysis module. Binary compatibility allows utilization of algorithms in different environments without the necessity of error-prone manual intervention and customization. This is in particular important for the evaluation of HLT analysis.

- Full integration of HLT analysis into the off-line software framework.
  A complete HLT off-line environment has been implemented. HLT analysis is now an integrated module of AliRoot. This includes the simulation and the reconstruction process, as well as the actual implementation of analysis components. The integration allows in particular the utilization of algorithms primarily developed for off-line analysis in HLT and vice versa.

- Data processing of HLT output.
  This work adds the post-processing of HLT output to the general data processing model. HLT output payload is now handled during the AliRoot event reconstruction phase. The foundation for data compression techniques and subsequent off-line analysis from the compressed data has been established and tested.

- Extensibility.
  Following the modular design, new units can now be added to the analysis without the need of changes in other units. The established integration procedure for HLT analysis algorithms enables all users of the ALICE community to contribute and to run specific analysis. The HLT component interface allows the integration of algorithms with minimal effort.

- Integration of TPC on-line event reconstruction algorithms.
  Existing and in parallel to this work developed algorithms for TPC online event reconstruction have been integrated into the HLT and commissioned. The work on TPC on-line analysis has been the motivation and test case for all implemented software solutions. As part of the work, TPC on-line components have been integrated into the global AliRoot simulation process.

Though HLT now represents a stable system in ALICE, there are problems in the overall scalability. The importance of integration of the various parts into the full system has been underestimated in the past. The problems are investigated continuously and the system improved step by step.

As a second obstacle, the overall processing rate is still too low. Optimization has been treated with a lower priority in the past. Bottlenecks are both in the data transport framework as well in the analysis algorithms. In most cases, analysis components can gain in performance if memory access and the handling of intermediate data is properly optimized. However there is no principle problem. Benchmark tests show the potential of the overall HLT architecture. Extensive and systematic benchmark tests which have been beyond the scope of this work will be carried out as the next step. Additional tools have already been provided as part of the HLT analysis framework.

In order to reach the final goal of the target performance and full scalability, the extensive commissioning has to be continued. Extensive system checks including nightly builds, unit tests, and global system test on a regular basis are prerequisites for a stable and maintainable system. As the next step, the test facility will be extended in order to meet the high demands.

The major result of the presented work is a complete and in large parts finalized software solution for on-line analysis with the ALICE High-Level Trigger. Extensive documentation is available and facilitates the maintainability. In the near future, the work on real physics trigger algorithms and components will be continued with high priority. The system is ready for on-line data analysis and is awaiting data from LHC operation.

A. List of Publications

A.1 Related Publications Significantly Contributed To

M. Richter et al., High level trigger applications for the ALICE experiment, IEEE Trans. Nucl. Sci. 55 (2008), doi: 10.1109/TNS.2007.913469

The ALICE Collaboration, K. Aamodt et al., The ALICE Experiment at the CERN LHC, 2008 JINST 3 S08002, doi: 10.1088/1748-0221/3/08/S08002, co-editor of HLT section,

M. Richter and D. Röhrich et al., 3D Reconstruction of 10000 Particle Trajectories in Real-time, in *Proc. 5th World Congress on Industrial Process Tomography* Bergen, Norway, Sep 3-6 2007

M. Richter et al., *The ALICE High Level Trigger*, in *Proc. Computing in High Energy Physics Conf. 2004 (CHEP04)*, Interlaken, Switzerland (2004)

A.2 Related Publications Contributed To

S.R. Bablok et al., High level trigger online calibration framework in ALICE, J. Phys.: Conf. Ser. **119** 022007 (2008), doi: 10.1088/1742-6596/119/2/022007

S.R. Bablok et al., ALICE HLT interfaces and data organisation, in *Proc. Computing in High Energy Physics Conf. 2006 (CHEP06)*, Mumbai, India (2006), [Online] http://cdsweb.cern.ch/record/1066629

D. Röhrich et al., Benchmarks and implementation of the ALICE high level trigger, IEEE Trans. Nucl. Sci. **53** pp 854-858 (2006),
doi: 10.1109/TNS.2006.873770

B. Becker et al., Real Time Global Tests of the ALICE High Level Trigger Data Transport Framework , IEEETrans. Nucl. Sci. **55** pp. 703-709 (2008) arXiv:0801.1252 [physics.ins-det]

A.3 Further Publications

I'am the author of the following pulications not releated to the presented topic but accomplished within my PhD stipend at the University of Bergen:

M. Richter et al., The control system for the front-end electronics of the ALICE time projection chamber, IEEE Trans. Nucl. Sci. **53** (2006) 980. doi: 10.1109/TNS.2006.874726

Alme, J., Richter, M. et al, A distributed, heterogeneous control system for the ALICE TPC electronics, International Conference Workshops on Parallel Processing 2005. (2005) pp. 265 Â 272, doi: 10.1109/ICPPW.2005.7

M. Richter et al., Communication software for the ALICE TPC front-end electronics, *Proc. 11th Workshop on Electronics for LHC and Future Experiments (LECC 2005), Heidelberg, Germany, 12-16 September 2005*, [Online] http://cdsweb.cern.ch/record/921200

Further publications contributed to:

K. Roed et al., Irradiation tests of the complete ALICE TPC front-end electronics chain, *Proc. of 11th Workshop on Electronics for LHC and Future Experiments (LECC 2005), Heidelberg, Germany, 12-16 September 2005* [Online] http://cdsweb.cern.ch/record/920423

J. Alme et al., Radiation-tolerant SRAM-FPGA Based Trigger and Readout Electronics for the ALICE Experiment, IEEE Trans. Nucl. Sci. **55** (2008) 76, doi: 10.1109/TNS.2007.910677

Fehlker, D. et al., Software environment for controlling and re-configuration of Xilinx Virtex FPGAs, *Proc Topical Workshop for Particle Physics 2007, Prague (Czech Republic), September 2007*, URL: http://www.particle.cz/conferences/twepp0

C. Gonzáles Gutiérres et al., "The ALICE TPC readout control unit", IEEE Nucl. Sci. Symp. Conf. Rec. 1, "575-579", "2005", "doi: 10.1109/NSSMIC.2005.1596317"

L Musa et al. (ALICE TPC Collaboration), J. Phys. G: Nucl. Part. Phys. **34** S705-S708, "doi: 10.1088/0954-3899/34/8/S78"

B. Software Appendix

B.1 The AliHLTComponent interface

The `AliHLTComponent` base class provides the interface for all HLT analysis components. This appendix is intended to present a short overview, detailed information and a reference class/interface description can be found in the ALICE HLT analysis framework on-line documentation[1].

The interface is divided in public external methods intended to be used from an application using the component, and internal methods which need to be implemented by the developer of an analysis component.

B.1.1 Public external methods

The external methods are provided to an application using the component. The public interface methods define the binding procedure into e.g. the HLT on-line system PubSub or the AliRoot off-line system. The methods are of little relevance for the developer of a component. Some of them are obligatory to be implemented by the child class in order to determine the properties.

```
/** Init function to prepare data processing.*/
int Init(const AliHLTAnalysisEnvironment* environ,
         void* environParam, int argc, const char** argv );

/** Clean-up function to terminate data processing.*/
int Deinit();

/** Processing of one event. */
int ProcessEvent(const AliHLTComponentEventData& evtData,
                 const AliHLTComponentBlockData* blocks,
                 AliHLTComponentTriggerData& trigData,
                 AliHLTUInt8_t* outputPtr,
                 AliHLTUInt32_t& size,
                 AliHLTUInt32_t& outputBlockCnt,
                 AliHLTComponentBlockData*& outputBlocks,
                 AliHLTComponentEventDoneData*& edd );
```

[1] http://web.ift.uib.no/~kjeks/doc/alice-hlt-current/classAliHLTComponent.html

Component Properties

Each component implementation must implement the component property functions. This set of methods is defined pure virtual.

Listing B.1: Component property functions to be implemented by the child component.

```
virtual const char* GetComponentID() = 0;
virtual void GetInputDataTypes( AliHLTComponentDataTypeList& ) = 0;
virtual AliHLTComponentDataType GetOutputDataType() = 0;
virtual int GetOutputDataTypes(AliHLTComponentDataTypeList& tgtList);
virtual void GetOutputDataSize( unsigned long& constBase,
                                double& inputMultiplier ) = 0;
virtual AliHLTComponent* Spawn() = 0;
```

The type of the component implementation is already defined in the default base classes `AliHLTDataSource`, `AliHLTProcessor`, and `AliHLTDataSink`.

Listing B.2: Property function to get the type of component.

```
virtual TComponentType GetComponentType() = 0; // Source, sink, or processor
```

B.1.2 Private internal methods

This section handles the component interface to a child class and is of high relevance for the developer of a component.

Component Setup

```
/** Child class method for the internal initialization.*/
virtual int DoInit( int argc, const char** argv );

/** Child class method for the internal clean-up.*/
virtual int DoDeinit();

/** Reconfigure the component. */
virtual int Reconfigure(const char* cdbEntry, const char* chainId);

/** Read the Preprocessor values.*/
virtual int ReadPreprocessorValues(const char* modules);

/** Custom handler for the SOR event.*/
virtual int StartOfRun();

/** Custom handler for the EOR event.*/
virtual int EndOfRun();
```

Data Processing

A general internal processing method is called from the `ProcessEvent` function when all internal variables are set and event processing has been prepared. The method is pure virtual and must be implemented by the child class. The method is already implemented in the default base classes `AliHLTDataSource`, `AliHLTProcessor`, and `AliHLTDataSink`.

```
/** Internal processing of one event.
 */
virtual int DoProcessing(const AliHLTComponentEventData& evtData,
                         const AliHLTComponentBlockData* blocks,
                         AliHLTComponentTriggerData& trigData,
                         AliHLTUInt8_t* outputPtr,
                         AliHLTUInt32_t& size,
                         AliHLTComponentBlockDataList& outputBlocks,
                         AliHLTComponentEventDoneData*& edd ) = 0;
```

For the sake of convenience, three component base classes specialized for the task of data processing, data publishing and data dumping are provided by the HLT analysis framework. Each class implements its own flavor of the processing method.

Data Processing in AliHLTProcessor

The class `AliHLTProcessor` is the base class for all processing components which process input data and produce output data. The normal development requires to implement a child class of `AliHLTPprocessor`. This base class implements the `AliHLTComponent::DoProcessing` method and provides a low level and a high level processing method for the child class.

```
int AliHLTProcessor::DoEvent(const AliHLTComponentEventData& evtData,
                const AliHLTComponentBlockData* /*blocks*/,
                AliHLTComponentTriggerData& trigData,
                AliHLTUInt8_t* /*outputPtr*/,
                AliHLTUInt32_t& size,
                vector<AliHLTComponentBlockData>& /*outputBlocks*/);
{
  // we just forward to the high level method, all other parameters
  // already have been stored internally
  size=0;
  return DoEvent(evtData, trigData);
}
int AliHLTProcessor::DoEvent(const AliHLTComponentEventData& /*evtData*/,
                             AliHLTComponentTriggerData& /*trigData*/)
{
  HLTFatal("no processing method implemented"); return -ENOSYS;
}
```

As described in section 4.3, the High Level interface method is especially designed for components which do not need to access the input block list and the output buffer directly. Implementation is also somewhat easier for the developer of moderate experience. Access to the input data is possible by means of the `GetFirstInputObject/GetNextInputObject` functions, respectively the `GetFirstInputBlock/GetNextInputBlock` functions.

The `AliHLTDataSource` and `AliHLTDataSink` components have similar implementations.

Access to input data blocks and output buffer

The data access methods provided by the `AliHLTComponent` base class are an important prerequisite for the usability of the high-level interface. The class provides methods to iterate trough all input blocks or objects. If objects are requested, the base class restores the ROOT objects from the input data and provides it to the child implementation. Furthermore there are methods to publish a newly generated object, and to forward an input block/object to the output. An overview of the most important methods can be found in listing B.3.

A special use case is the so called *memory file* which allows to open a ROOT file in memory instead of disk. The functionality provides normal file I/O operations. The file structure is sent over the normal component output at the end of the processing function.

Listing B.3: High-level data access methods of the component base class.

```
/** Get the first object of a specific data type from the input data. */
const TObject* GetFirstInputObject(const AliHLTComponentDataType& dt,
                                   const char* classname,
                                   int bForce);

/** Get the first object of a specific data type from the input data. */
const TObject* GetFirstInputObject(const char* dtID,
                                   const char* dtOrigin,
                                   const char* classname,
                                   int bForce);

/** Get the next object of a specific data type from the input data. */
const TObject* GetNextInputObject(int bForce);

/** Get data type of an input block. */
AliHLTComponentDataType GetDataType(const TObject* pObject);

/** Get data specification of an input block. */
AliHLTUInt32_t GetSpecification(const TObject* pObject);

```

```
/** Get the first block of a specific data type from the input data. */
const AliHLTComponentBlockData* GetFirstInputBlock(const char* dtID,
                                                   const char* dtOrigin);

/** Get the next block of a specific data type from the input data. */
const AliHLTComponentBlockData* GetNextInputBlock();

/**Get data specification of an input block. */
AliHLTUInt32_t GetSpecification(const AliHLTComponentBlockData* pBlock=NULL);

/** Forward an input object to the output. */
int Forward(const TObject* pObject);

/** Forward an input block to the output. */
int Forward(const AliHLTComponentBlockData* pBlock=NULL);

/** Insert an object into the output. */
int PushBack(TObject* pObject, const AliHLTComponentDataType& dt,
             AliHLTUInt32_t spec=kAliHLTVoidDataSpec,
             void* pHeader=NULL, int headerSize=0);

/** Insert buffer into the output. */
int PushBack(const void* pBuffer, int iSize,
             const AliHLTComponentDataType& dt,
             AliHLTUInt32_t spec=kAliHLTVoidDataSpec,
             const void* pHeader=NULL, int headerSize=0);


/** Create a memory file in the output stream. */
AliHLTMemoryFile* CreateMemoryFile(int capacity,
                                   const AliHLTComponentDataType& dt,
                                   AliHLTUInt32_t spec);

/** Write an object to memory file in the output stream.
int Write(AliHLTMemoryFile* pFile, const TObject* pObject,
          const char* key=NULL, int option=TObject::kOverwrite);

/** Close object memory file. */
int CloseMemoryFile(AliHLTMemoryFile* pFile);
```

B.1.3 Example Implementation of Low-Level Processing

The Low-level processing interface gives full flexibility as it provides access to all parameters of the component processing function.

The function gets an array of block descriptors and must iterate over all blocks in order to find the relevant input data. Output data are directly written to the provided data buffer and corresponding block descriptors are inserted into the list of output blocks. Listing B.4 shows a sketch from the

`AliHLTTPCClusterFinderComponent`.

Listing B.4: Example implementation of the low-level processing method. From the `AliHLTTPCClusterFinderComponent`

```
int AliHLTTPCClusterFinderComponent
    ::DoEvent(const AliHLTComponentEventData& evtData,
              const AliHLTComponentBlockData* blocks,
              AliHLTComponentTriggerData& /*trigData*/,
              AliHLTUInt8_t* outputPtr,
              AliHLTUInt32_t& size,
              vector<AliHLTComponentBlockData>& outputBlocks )
{
  // .....

  // == init iter (pointer to datablock)
  const AliHLTComponentBlockData* iter = NULL;
  unsigned long ndx;

  for ( ndx = 0; ndx < evtData.fBlockCnt; ndx++ )
    {
      iter = blocks+ndx;

      // ... Processing

      // Fill block descriptor
      AliHLTComponentBlockData bd;
      FillBlockData( bd );
      bd.fOffset = offset;
      bd.fSize = mysize;
      bd.fSpecification = iter->fSpecification;
      bd.fDataType = AliHLTTPCDefinitions::fgkClustersDataType;
      outputBlocks.push_back( bd );

      tSize += mysize;
      outBPtr += mysize;
      outPtr = (AliHLTTPCClusterData*)outBPtr;
    }

  size = tSize;

  // ...
}
```

B.1.4 Example Implementation of High-Level Processing

The High-level processing interface has been introduced for the sake of simplification of the processing function and the treatment of ROOT objects. Instead of passing all parameters to the method, the base class providing access methods as presented above. The methods include the serialization of

ROOT objects. Produced data blocks are inserted to the output stream by using specific PushBack methods of the component base class. In the example of listing B.5, the component iterates though all input objects by means of `GetFirstInputObject/GetNextInputObject` and writes the restored objects to a memory file.

Listing B.5: Example implementation of the high-level processing method.

```
int AliHLTRootFileStreamerComponent
    ::DoEvent(const AliHLTComponentEventData& /*evtData*/,
              AliHLTComponentTriggerData& /*trigData*/ )
{
  // see header file for class documentation
  int iResult=0;
  AliHLTMemoryFile* pFile=CreateMemoryFile(fDataType,fSpecification);
  if (pFile) {
    int count=0;
    for (const TObject* pObj=GetFirstInputObject();
         pObj && iResult>=0;
         pObj=GetNextInputObject()) {
      iResult=Write(pFile, pObj);
      if (iResult) {
        count++;
      }
    }
    HLTInfo("wrote %d object(s) from %d input blocks to file",
            count, GetNumberOfInputBlocks());
    iResult=CloseMemoryFile(pFile);
  } else {
    iResult=-ENOMEM;
  }
  return iResult;
}
```

B.1.5 Return and Error Code Scheme

For return codes, the following scheme applies:

- The data processing methods have to indicate error conditions by a negative error/return code, like e.g. system error code `-EINVAL`.
- If no suitable input block could be found (e.g. no clusters for the TPC cluster finder) set size to 0, block list is empty, return 0.
- If no usable or significant signal could be found in the input blocks return an empty output block, set size accordingly, and return 0.
- If the output buffer is not big enough return empty block list and `-ENOSPC`.

B.2 Common HLT Data Types

The framework provides data origins and data types as shown in tables B.1 and B.2. Data origins follow when ever possible the notation of the sub-detectors in ALICE. Additional origin keys have been defined for HLT specific data blocks, and some detectors for a better distinction of the exact origin. Table shows the current definition. All common data types are of origin *ANY*.

'TPC '	kAliHLTDataOriginTPC	Data Origin TPC
'PHOS'	kAliHLTDataOriginPHOS	Data origin PHOS
'FMD '	kAliHLTDataOriginFMD	Data origin FMD
'MUON'	kAliHLTDataOriginMUON	Data origin MUON
'TRD '	kAliHLTDataOriginTRD	Data origin TRD
'ITS '	kAliHLTDataOriginITS	Data origin ITS
'ISPD'	kAliHLTDataOriginITSSPD	Data origin ITS SPD
'ISDD'	kAliHLTDataOriginITSSDD	Data origin ITS SDD
'ISSD'	kAliHLTDataOriginITSSSD	Data origin ITS SSD
'EMCL'	kAliHLTDataOriginEMCAL	Data origin EMCAL
'OUT '	kAliHLTDataOriginOut	Data origin HLT out
'OFFL'	kAliHLTDataOriginOffline	Data origin Offline
'PRIV'	kAliHLTDataOriginPrivate	HLT/PubSub private internal
'SMPL'	kAliHLTDataOriginSample	Data origin for examples
"/0/0/0"	kAliHLTDataOriginVoid	invalid data origin
"***"	kAliHLTDataOriginAny	wildcard data type origin

Table B.1: Common data origins defined for HLT data exchange. Please note the blank at the end of three-character definitions.

'DDL_RAW '	kAliHLTDataTypeDDLRaw	DDL raw data
'ALIESDV0'	kAliHLTDataTypeESDObject	ESD object
'ROOTTREE'	kAliHLTDataTypeTTree	A ROOT TTree object
'ROOTHIST'	kAliHLTDataTypeHistogram	A ROOT histogram
"/0/0/0/0/0/0/0"	kAliHLTVoidDataTypeID	invalid data type id
"*******"	kAliHLTAnyDataTypeID	wildcard data type id

Table B.2: Common data types defined for HLT data exchange. All data types are of origin *ANY*.

C. Benchmark Environment

For the purpose of benchmark studies, dedicated on-line configurations have been added to the HLT RunControl. The configurations allow the periodic test of the system performance. An overview of HLT on-line configurations used for the benchmark tests throughout this work and macros describing useful off-line HLT chains is presented here.

Listing C.1: Post-processing of HLT component statistics raw data by an off-line HLT chain using the `AliHLTCompStatCollector` component. The macro is part of the AliRoot distribution.

```
/**
 * Helper macro to format a block of AliHLTComponentStatistics entries.
 * The block is usually created by attaching a file writer to a chain,
 * writing only COMPSTAT:PRIV data blocks.
 *
 * The macro translates the block into HLTruns a stand-alone chain
 * Usage:
 * <pre>
 *   aliroot -b -q format-statistics.C | tee format-statistics.log
 * </pre>
 *
 *
 * @ingroup alihlt_benchmark
 * @author Matthias.Richter@ift.uib.no
 */
void format_statistics(const char* infile,
                       const char* outfile="HLT.statistics.root")
{
  AliHLTSystem gHLT;
  gHLT.SetGlobalLoggingLevel(0x7c);
  gHLT.LoadComponentLibraries("libAliHLTUtil.so");
  TString arg;
  arg.Form("-datatype 'COMPSTAT' 'PRIV' -datafile %s", infile);
  AliHLTConfiguration publisher("publisher", "FilePublisher", NULL,
                                arg.Data());

  arg.Form("-file %s -publish 0 -arraysize 200000", outfile);
  AliHLTConfiguration sink1("sink1", "StatisticsCollector",
                            "publisher", arg.Data());

  gHLT.BuildTaskList("sink1");
  gHLT.Run();
}
```

Listing C.2: XML configuration describing the benchmark test for component fan-in. The storage of the component statistics by a *FileWriter* component has been omitted.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SimpleChainConfig2 ID="TPC" verbosity="0x78">
  <infoblock>
     <author>Matthias</author>
     <date>20/01/09</date>
     <description>
     Component  fan-in benchmark using DataGenerator components
     </description>
  </infoblock>

<ALICE>
  <Sources type="file">
<!-- one publisher on 1 node -->
     <DDL>768</DDL>

<!-- 4 publishers on 4 nodes -->
<!-- <DDL>768,772,776,780</DDL> -->
     <Directory>/opt/HLT-data/global/empty-Events/raw0</Directory>
  </Sources>

  <FilePublisherOptions>-buffermanager ring</FilePublisherOptions>
  <RORCShm blocksize="64" blockcount="2048"/>

  <TPC>
    <Slice> <Partition>

        <Component ID="pub">
          <ComponentID>DataGenerator</ComponentID>
          <Options>-size 10000 -range 1000</Options>
          <Parent>DDL</Parent>
          <Shm blocksize="100k" blockcount="2048"/>
          <Multiplicity>1</Multiplicity>
          <Library>libAliHLTUtil.so</Library>
          <ForceFEP/>
        </Component>
    </Partition> </Slice>

    <Component ID="fan-in">
      <ComponentID>DataGenerator</ComponentID>
      <Options>-multiplier 0.0</Options>
      <Parent>pub</Parent>
      <Shm blocksize="100k" blockcount="2048"/>
      <Multiplicity>1</Multiplicity>
      <Library>libAliHLTUtil.so</Library>
      <Node>cntpcc040</Node>
    </Component>

</TPC> </ALICE>

</SimpleChainConfig2>
```

Listing C.3: Example macro describing the benchmark test for the exchange of ROOT objects utilizing the `AliHLTBenchExternalTrackComponent`.

```
void bench_externaltrackparam(int events=100, int compressionLevel=4)
{
  ///////////////////////////////////////////////////////////////////
  // init the HLT system in order to define the analysis chain below
  //
  gSystem->Load("libHLTrec.so");
  AliHLTSystem* gHLT=AliHLTReconstructorBase::GetInstance();
  gHLT->SetGlobalLoggingLevel(0x7c);

  ///////////////////////////////////////////////////////////////////
  // define the analysis chain to be run
  //
  int iNofPublishers=1;
  TString dumpInput;
  for (int pub=0; pub<1; pub++) {
    TString publisher;
    TString arg;
    // publishers of AliExternalTrackParam arrays
    publisher.Form("PUB_\%02d", pub);
    arg="-minsize 100 -maxsize 10000";
    arg+=" -tclonesarray";
    arg+=" -object-compression="; arg+=compressionLevel;
    arg+=" -verbosity 1";
    AliHLTConfiguration publisherconf(publisher.Data(),
                                      "BenchmarkAliExternalTrackParam",
                                      NULL, arg.Data());
    if (dumpInput.Length()>0) dumpInput+=" ";
    dumpInput+=publisher;
  }

  AliHLTConfiguration dumpconf("sink1", "BenchmarkAliExternalTrackParam",
                               dumpInput.Data(), "-verbosity 1");

  AliHLTConfiguration statconf("stat", "StatisticsCollector", "sink1", "");

  AliHLTConfiguration writer("statwriter", "ROOTFileWriter", "stat",
                             "-datafile HLT.statistics.root "
                             "-concatenate-events -overwrite");

  ///////////////////////////////////////////////////////////////////
  // Init and run the chain
  //
  gHLT->LoadComponentLibraries("libAliHLTBenchmark.so libAliHLTUtil.so");
  gHLT->BuildTaskList("statwriter");
  gHLT->Run(events);
}
```

Listing C.4: XML configuration describing the benchmark test for the exchange of ROOT objects.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SimpleChainConfig2 ID="TPC" verbosity="0x78">
  <infoblock>
     <author>Matthias</author>
     <date>29/11/08</date>
     <description>
     Data transport benchmark for arrays of AliExternalTrackParam
     </description>
  </infoblock>

<ALICE>
  <Sources type="file">
    <DDL>768</DDL>
     <Directory>/opt/HLT-data/global/empty-Events/raw0</Directory>
  </Sources>
  <RORCShm blocksize="40" blockcount="100"/>

  <TPC>
    <Slice> <Partition>
        <Component ID="publisher">
          <ComponentID>BenchmarkAliExternalTrackParam</ComponentID>
          <Options>
           -minsize 100 -maxsize 1000 -nocheck
<!--       case 1: the C-structure -->
           -carray
<!--       case 2: TClonesArray compression 0 -->
<!--       -tclonesarray -object-compression=0 -->
          </Options>
          <Parent>DDL</Parent>
          <Shm blocksize="1M" blockcount="10"/>
          <Multiplicity>1</Multiplicity>
          <Library>libAliHLTBenchmark.so</Library>
          <ForceFEP/>
        </Component>
    </Partition> </Slice>

    <Component ID="sink">
      <ComponentID>BenchmarkAliExternalTrackParam</ComponentID>
      <Options></Options>
      <Parent>publisher</Parent>
      <Shm blocksize="1k" blockcount="10"/>
      <Multiplicity>1</Multiplicity>
      <Library>libAliHLTBenchmark.so</Library>
<!--  run eiter on the same node (ForceFEP) or not -->
<!--  <ForceFEP/> -->
    </Component>
</TPC> </ALICE>
</SimpleChainConfig2>
```

Glossary

ADC	Analog-to-Digital Converter
AFS	Andrew File System
ALICE	A Large Ion Collider Experiment
AliEve	Alice Event Monitoring Display
AliRoot	Alice ROOT
ALTRO	ALICE TPC Readout chip
AOD	Analysis Object Data
AOP	Aspect Oriented Programming
API	Application Programming Interface
C	C programming language
C++	C++ object oriented programming language
CDB	Configuration DataBase
CDH	Common Data Header
CERN	European Organization for Nuclear Research
CRC	Cyclic Redundancy Check
D-RORC	DAQ RORC
DA	Detector Algorithms
DAQ	Data Aquisition
DATE	Data Acquisition and Test Environment
DCS	Detector Control System
DDL	Detector Data Link
DIU	DDL Destination Interface Unit
DMA	Direct Memory Access
ECS	Experiment Control System
EMCAL	Electromagnetic Calorimeter
EOD	End-of-data event
ESD	Event Summary Data
Eve	ROOT Event Monitoring Display
FAIR	Facility for Antiproton and Ion Research at GSI Darmstadt/Germany
FEC	Front-End Card
FEE	Front-End Electronics
FEP	Front-End Processor
FMD	Forward Multiplicity Detector

FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GByte	Gigabyte
GDC	Global Data Collector
H-RORC	HLT RORC
HEP	High Energy Physics
HLT	High-Level Trigger
HMPID	High Momentum Particle Identification Detector
HOMER	HLT On-line Monitoring Environment including ROOT
IP	Interaction Point
IR	Infrared
IROC	TPC Inner Readout Chamber
ITS	Inner Tracking System
L0	Level-0 trigger
L1	Level-1 trigger
L2	Level-2 trigger
LDC	Local Data Concentrator
LEP	Large Electron-Positron Collider
LHC	Large Hadron Collider
LSB	Least Significant Byte
MSB	Most Significant Byte
MWPC	Multi-Wire Proportional Chamber
OCDB	Offline Conditions Data Base
OOP	Object Oriented Programming
OROC	TPC Outer Readout Chamber
OS	Operating System
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCI-X	Peripheral Component Interconnect eXtended
PHOS	Photon Spectrometer
PID	Particle Identification
PubSub	Publisher- Subscriber Framework
QCD	Quantum Chromo Dynamics
QGP	Quark Gluon Plasma
RCU	Readout Control Unit

RHIC	Relativistic Heavy Ion Collider, Brookhaven Nat. Lab./USA
ROI	Region Of Interest Readout
ROOT	An Object Oriented Analysis Framework
RORC	Read-Out Received Card
SDD	Silicon Drift Detector
SIU	DDL Source Interface Unit
SMP	Symmetrical Multi Processing
SOD	Start-of-data event
SPD	Silicon Pixel Detector
SPS	Super Proton Synchrotron
SSD	Silicon Strip Detector
STAR	Solenoidal Tracker At RHIC
SysMes	System Management for Networked Embedded Systems and Clusters
T0	Time 0 detector
TCP/IP	Transmission Control Protocol/Internet Protocol
TOF	Time-Of-Flight
TPC	Time Projection Chamber
TRD	Transition Radiation Detector
TTC	Timing, Trigger and Control
UI	User Interface
UNIX	(UNICS) UNiplexed Information and Computing System
V0	Vertex 0 detector
WN	Worker Node
XML	Extensible Markup Language

Bibliography

[1] L. Evans and P. Bryant et al. The Cern Large Hadron Collider. *JINST 3 S08002*, 2008. doi: 10.1088/1748-0221/3/08/S08001.

[2] CERN AC - HF 267 http://cdsweb.cern.ch/record/841560.

[3] H. Satz. The Transition from Hadron Matter to Quark-Gluon Plasma. *Ann. Rev. Nucl. Part. Sci.*, 35:245–270, 1985. doi: 10.1146/annurev.ns.35.120185.001333.

[4] J. Stachel P. Braun-Munzinger. Probing the Phase Boundary between Hadronic Matter and the Quark-Gluon-Plasma in Relativistic Heavy Ion Collisions. *Nucl.Phys. A606*, pages 320–328, 1996. doi: 10.1016/j.nima.2006.05.036.

[5] HEP Phase Diagram. Introduction to the CBM experiment, 2007. [Online] http://www.gsi.de/fair/experiments/CBM/1intro.html.

[6] E. Shuryak. Physics of Strongly coupled Quark-Gluon Plasma. *Prog. Part. Nucl. Phys.*, 62, 2009. doi: 10.1016/j.ppnp.2008.09.001.

[7] A. Franz (for the PHENIX Collaboration). Highlights from PHENIX: I (Quark Matter 2008). *J. Phys. G: Nucl. Part. Phys.*, 35(104007), 2008. doi: 10.1088/0954-3899/35/10/104002.

[8] T.C. Awes (for the PHENIX Collaboration). Highlights from PHENIX: II (Quark Matter 2008). *J. Phys. G: Nucl. Part. Phys.*, 35(104007), 2008. doi: 10.1088/0954-3899/35/10/104007.

[9] A. Adare et al. (Phenix Collaboration). Transverse momentum and centrality dependence of dihadron correlations in Au+Au collisions at $\sqrt{s_{NN}} = 200\ GeV$: Jet quenching and the response of partonic matter. *Phys. Rev. C*, 77(011901(R)), 2008. doi: 10.1103/PhysRevC.77.011901.

[10] ALICE collaboration. ALICE - Technical Proposal for A Large Ion Collider Experiment at the CERN LHC. *CERN/LHCC 1995-71*, 1995.

[11] K. Aamodt et al. The ALICE Collaboration. The ALICE Experiment at the CERN LHC. *JINST 3 S08002*, 2008. doi: 10.1088/1748-0221/3/08/S08002.

[12] R. Brun et al. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997. [Online] http://www.ifh.de/CHEP97/paper/358.ps.

[13] K. Aamodt et al. The ALICE Collaboration. The ALICE Experiment at the CERN LHC - Offline Computing. *JINST 3 S08002*, pages 162–182, 2008. doi: 10.1088/1748-0221/3/08/S08002.

[14] I. Foster and C. Kesselmann. *The Grid 2: Blueprint for a new Comuting Infrastructure*. Morgan Kaufmann Publishers, U.S.A., 2004.

[15] ALICE Collaboration. Technical Design Report: Trigger, DAQ, HLT, DCS. *CERN-LHCC-2003-062*, 2004.

[16] ALICE Collaboration. Technical Design Report: Trigger, DAQ, HLT, DCS. *CERN-LHCC-2003-062*, pages 413–422, 2004. [Online] http://cdsweb.cern.ch/record/684651.

[17] B. G. Taylor. Timing distribution at the LHC. *Proceedings of the 8th Workshop on Electronics for LHC Experiments, Colmar France*, 2002. [Online] CERN-2002-003, http://cdsweb.cern.ch/record/592719.

[18] S. Bablok et al. High level trigger online calibration framework in ALICE. *J. Phys.: Conf. Ser.*, 119 022007, 2008. doi: 10.1088/1742-6596/119/2/022007.

[19] *System Management for Networked Embedded Systems and Clusters*, 2008. [Online] http://wiki.kip.uni-heidelberg.de/ti/SysMES/index.php/Main_Page.

[20] D. Röhrich et al. Benchmarks and Implementation of the ALICE High Level Trigger. *IEEE Trans. Nucl. Sci.*, 53:854–858, 2006. doi: 10.1109/TNS.2006.873770.

[21] C. Loizides. *Jet physics in ALICE*. PhD thesis, University of Frankfurt, Inst. Kernphysik, Germany, 2005. [Online] arXiv:0501017 [nucl-ex].

[22] B. Becker et al. ALICE dimuon high-level trigger: project review. Technical Report ALICE-INT-2007-022, CERN, 2007. [Online] https://edms.cern.ch/document/878756.

[23] T. Alt and V. Lindenstruth. The ALICE HLT Read-Out Receiver Card. *GSI Scientific Report 2005*, pages 286, 2005.

[24] T. M. Steinbeck. *A modular and fault-tolerant data transport framework*. PhD thesis, Ruprecht-Karls-University Heidelberg, Germany, 2004. [Online] arXiv:0404014 [cs].

[25] T. M. Steinbeck et al. An Object-Oriented Network-Transparent Data Transportation Framework. *IEEE Trans. Nucl. Sci.*, 49:455–459, 2002. doi: 10.1109/TNS.2002.1003773.

[26] T. M. Steinbeck et al. New experiences with the ALICE High Level Trigger Data Transport Framework. *Proc. Computing in High Energy Physics Conf. (CHEP04)*, 2004. [Online] http://chep2004.web.cern.ch/chep2004.

[27] J. P. R. Middlelink. Bigphysarea kernel patch, 2003. [Online] http://www.polyware.nl/ middelink/En/hob-v4l.html#bigphysarea.

[28] T. M. Steinbeck et al. A Control Software for the ALICE High Level Trigger. *Proc. Computing in High Energy Physics Conf. (CHEP04)*, 2004. [Online] http://chep2004.web.cern.ch/chep2004.

[29] T. M. Steinbeck. *SimpleChainConfig*. HLT project internal information, University of Heidelberg, 2007.

[30] S.R. Bablok et al. ALICE HLT interfaces and data organisation. *Computing in High Energy Physics Conf. 2006 (CHEP06), Mumbai, India*, 2006. [Online] http://cdsweb.cern.ch/record/1066629.

[31] R. Divià and T.M. Steinbeck. Data format and specifications for the HLT-to-DAQ interface. Technical Report ALICE-INT-2007-015, ver. 3, CERN, 2008. [Online] https://edms.cern.ch/document/871995.

[32] P. Vande Vyvre R. Divià, P. Jovanovic. Data Format over the ALICE DDL. Technical Report ALICE-INT-2002-010, ver. 11, CERN, 2007. [Online] https://edms.cern.ch/document/340186.

[33] J. Wagner et al. Lossless Data Compression for ALICE HLT. Technical Report ALICE-INT-2008-020, ver. 1, CERN, 2008. [Online] https://edms.cern.ch/document/948159.

[34] D. Röhrich and A. Vestbø. Efficient TPC data compression by track and cluster modeling. *Nucl. Instrum. Meth.*, A566:668, 2006. doi: 10.1016/j.nima.2006.06.056.

[35] F. Carminati et al. The ALICE Offline Environment. *Computing in High Energy Physics Conf. 2007 (CHEP07), Victoria, Canada*, 2007.

[36] J.F.Grosse-Oetringhaus A.Colla. The Shuttle Framework - A system for automatic readout and processing of conditions data. Technical Report ALICE-INT-2008-011, ver. 01, CERN, 2008. [Online] https://edms.cern.ch/document/924807.

[37] T. M. Steinbeck. HLT Online Monitoring Environment including ROOT. presented during ALICE week March 2007, 2007. [Online] http://indico.cern.ch/conferenceDisplay.py?confId=13370.

[38] M. Tadel. Raw-data display and visual reconstruction validation in ALICE. *J. Phys.: Conf. Ser.*, 119 032036, 2008. doi: 10.1088/1742-6596/119/3/032036.

[39] M. Tadel. The new generation of OpenGL support in ROOT. *J. Phys.: Conf. Ser.*, 119 042028, 2008. doi: 10.1088/1742-6596/119/4/042028.

[40] S. Pont R. Gauthier. *Designing Systems Programs*. Prentice-Hall, Englewood Cliffs, N.J., 1970.

[41] G. Kiczales et al. Aspect-Oriented Programming. *Lecture Notes in Computer Science (LNCS)*, 1241, 1997. doi: 10.1007/BFb0053371.

[42] S. R. Bablok. *Heterogeneous Distributed Calibration Framework for the High Level Trigger in ALICE*. PhD thesis, University of Bergen, Norway, 2008.

[43] ALICE HLT analysis framework on-line documentation - Tutorial. http://web.ift.uib.no/~kjeks/doc/alice-hlt-current/group__alihlt__tutorial.html.

[44] AliRoot tutorial. AliRoot tutorial session Oct 2008, 2008. [Online] http://aliceinfo.cern.ch/export/download/OfflineDownload/tutorial.ppt.

[45] D Bryant D Salomon, G Motta. *Data Compression: The Complete Reference*. Springer Verlag, London, GB, 4 edition, 2007.

[46] K. Aamodt et al. The ALICE Collaboration. The ALICE Experiment at the CERN LHC. *JINST 3 S08002*, page 175, 2008. doi: 10.1088/1748-0221/3/08/S08002.

[47] ALICE Collaboration. ALICE time projection chamber : Technical Design Report. *CERN-LHCC-2000-001*, 2000. [Online] http://cdsweb.cern.ch/record/451098.

[48] R. Esteve Bosch et al. The ALTRO chip: a 16-channel A/D converter and digital processor for gas detectors. *IEEE Trans. Nucl. Sci.*, pages 2460 – 2469, 2003. doi: 10.1119/TNS.2003.820629.

[49] A. Vestbø. *Pattern Recognition and Data Compression for the ALICE High Level Trigger*. PhD thesis, University of Bergen, Norway, 2004. [Online] arXiv:0406003 [physics].

[50] I. Abt, D. Emeliyanov, I. Gorbounov, and I. Kisel. Cellular automaton and Kalman filter based track search in the HERA-B pattern tracker. *Nucl. Instrum. Meth.*, A490(3):546 – 558, 2002. doi: 10.1016/S0168-9002(02)01097-5.

[51] S. Gorbunov. TPC online reconstruction - CA tracker. presented during ALICE offline week April 2008, 2008. [Online] http://indico.cern.ch/conferenceDisplay.py?confId=27911#7.

[52] C. Cheshkov. Fast Hough-transform track reconstruction for the ALICE TPC. *Nucl. Instrum. Meth.*, A566(1):35–39, 2006. doi: http://dx.doi.org/10.1016/j.nima.2006.05.036.

[53] C. Gonzáles Gutiérres et al. The ALICE TPC readout control unit. *IEEE Nucl. Sci. Symp. Conf. Rec. 1*, pages 575–579, 2005. doi: 10.1109/NSSMIC.2005.1596317.

[54] private communication S. Rosseger.

Index

Errata

page I ”Abstract” 2nd line: deleted ”(HEP)”
page 3 ”1. Introduction” 1st paragraph line 2: ” one particluar sub-detector” → ”particular”
page 6 ”2. ALICE and the Large Hadron Collider” 1st paragraph line 2: ”ions beams” → ”ion beams”
page 6 ”2.1 Physics Motivation” 2nd paragraph line 3: ”describing sub-atomic interaction” → ”.. interactions ...”
page 7 ”2.1 Physics Motivation” 1st paragraph line 2: ”need to be” → ”needs to be”
page 7 ”2.2 Quark Gluon Plasma” 1st paragraph line 5: ”1980's” → ”1980s”
page 9 ”2.2 Quark Gluon Plasma” 1st paragraph line 5: ”Jets” → ”jets”
page 10 ”Hard Collision and Jets” 2nd paragraph: 2x ”Jet” → ”jet”
page 13 Fig. 2.6.: ”TTC” needs to be ”Trigger” as TTC describes a common LHC project and the ALICE subsystem is named ”Trigger”
page 16 ”3.1 Conceptual Design” last paragraph line 1: ”HLT is in influenced” → ”HLT is influenced”
page 17 ”3.2 Processing Methodology - Sequential Event Processing” 2nd paragraph, line 2: ”bandwith” → ”bandwidth”; inserted space in ”running period”; skipped ”the” before HLTs.
page 17 Figure 3.2: left node has to be “Node 2”; numbering of the output events corrected “4” → “n”
page 18 ”Pipelined Data Processing” 2nd par, last line: ”later throughout this section” → ”in section 3.4.3”
page 20 ”Shared Memory based Data Exchange”, added reference to section 3.4.2
page 21 ”3.3.1 The Concept of Components”, line 2: ”HLT component” → ”HLT components”
page 22 ”3.3.2 Data Input of the HLT on-line System”, line 3: ”A DDL” → ”The DDL ...”
page 28 ”3.4.4 Intrinsic Data Properties” line 1 and last item line 1: ”endianess” → ”endianness”
page 33 ”3.7.2 Development Environment - The make Utility” 1st paragraph line 1: ” 1970'ies” → ”1970s”
page 41 ”3.9.2 HLT On-line Monitoring Environment” 1st paragraph line 5: ”Endianess” → ”Endianness”
page 45 ”4.1 Interface Methodology” 1st paragraph line 3: ”1970's” → ”1970”
page 47 ”Aspects in the Design of HLT” 2nd paragraph line 4: ”to first extend” → ”extent”
page 57 ”4.3.2 Running Environment” 1st par, line 6: ”Every component send ...” → ”Every component sends ...”
page 57 ”4.3.3 Initialization and Cleanup” last line: ”takes account for ..” → ”takes account of ..”
page 61 ”4.3.4 Data Processing” 3rd paragraph line 9: deleted ')' behind reference 3.4.4
page 80 ”4.5.2 The Off-line HLT System - HLT Configurations in AliRoot”: change in layout in order to avoid enumeration to be broken up. Last paragraph before ”HLT Configurations in AliRoot” has been shortened by replacing ”characterized” with ”specified” and ”configuration file” with ”configuration”
page 81 ”4.5.2 The Off-line HLT System - HLT Configurations in AliRoot” 2nd paragraph line 4: ”can simply look like: ” → ” can simply look like illustrated in listing 4.11” Listing 4.11 placed in the middle of the page
page 81 Listing 4.11: linebreak added in ”fp1 fp2”, ”output_percentage 80” in order to stay within margin
page 86 ”4.5.3 AliRoot HLT Simulation - Output of HLT simulation” line 1: ”10 DDL connection” → ”connections”
page 87 ”4.5.4 AliRoot HLT Reconstruction” line 5: ”pug-ins” → ”plug-ins”
page 92 ”4.6 HLT Data Exchange”, 1st paragraph, line 7: ”the caches holds ” → ”the cache holds ”
page 93 ”4.6 HLT Data Exchange”, 1st paragraph line 2: ”Endianess” → ”Endianness”
page 96 ”4.6.2 ROOT Objects - Serialization of Single Objects” 2nd paragraph line 4: ”Endianess” → ”Endianness”
page 107 ”Preprocessed Raw HLTOUT data” 1st paragraph line 4: ”regarding to tracks” → ”with respect to tracks”
page 130 ”7.2 TPC Raw Data” 1st paragraph line 6: ”bit with” → ”bit width”
page 131 Fig 7.3.: The description of the fields in the right block (Altro channel) is missing and has been added. Adjustment of colors to fit b/w printing.
page 142 last paragraph: ”the overall processing rate still too low” missing ”is” inserted
page 143 2nd paragraph line 4: ”is a prerequisite” → ”are prerequisites”
page 161 “Glossary”: added “XML”
General replacement of ”can not” → ”cannot” (occurrence 15 times throughout the thesis)
General replacement of ”it's” → ”its” (occurrence 18 times throughout the thesis)
Further references added to some index terms

Hyphenation:
page 5 ”2. ALICE and the Large Hadron Collider” 1st paragraph, line 4: ”devel-oped” → ”de-veloped”
page 57 ”4.3.2 Running Environment” 2nd par, line 4: ”envi-ronment” → ”envir-onment”
page 61 ”4.3.4 Data Processing” 2nd paragraph, line 1: ”corre-sponding” → ”cores-ponding”
page 74 ”4.5.1 AliRoot Data Processing Flow” last line: ”perfor-mance” → ”perform-ance”
page 83 ”4.5.3 AliRoot HLT Simulation” last paragraph line 1: ”proce-dure” → ”pro-cedure”
page 84 ”4.5.3 AliRoot HLT Simulation” last paragraph line 1: ”possi-ble” → ”pos-sible”