Searching for Stable Massive Particles in the ATLAS Transition Radiation Tracker

by

Steven Schramm

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Science (Hons)

in

THE FACULTY OF SCIENCE
(Physics and Astronomy)

The University Of British Columbia
(Vancouver)

April 2011

© Steven Schramm, 2011

Abstract

The search for new particles is an important topic in modern particle physics. New stable massive particles are predicted by various models, including R-parity conserving variants of supersymmetry. The ATLAS Transition Radiation Tracker can be used to look for charged stable massive particles by using the momentum and velocity of particles passing through the detector to calculate its mass. A program named TRTCHAMP has been written to perform this analysis.

Recently, ATLAS changed their data retention policies to phase out the use of a particular format of output file created during reconstruction. TRTCHAMP relies on this file type, and therefore this change prevents the algorithm from working in its original state. However, this problem can be resolved by integrating TRTCHAMP into the official reconstruction, which has now been done.

This paper documents the changes involved in integrating TRTCHAMP with the existing InDetLowBetaFinder reconstruction package. Topics discussed include the structure of reconstruction algorithms in ATLAS, the retrieval of calibration constants from the detector conditions database, and the parsing of necessary parameters from multiple reconstruction data containers. To conclude, the output of TRTCHAMP is shown to accurately estimate the velocity of protons in minimum bias tracks. Additionally, mass plots generated with the TRTCHAMP results are shown to agree with the known mass for both monte carlo and real protons in minimum bias tracks, and agreement is shown between the input and output masses for a monte carlo sample of 300 GeV R-Hadrons.

Table of Contents

Abstract . . . . . **ii**

Table of Contents . . . . . **iii**

List of Figures . . . . . **v**

Glossary . . . . . **vi**

Acknowledgments . . . . . **vii**

1 Introduction **1**

1.1 New Particles 1

1.2 Our Project 3

2 Theory **6**

2.1 Tracking Charged Particles in the TRT 6

2.2 Estimating Relativistic Velocities in the TRT 8

3 Methods **13**

3.1 Reasons for Switching to Reconstruction 13

3.2 Private Algorithm Implementation 15

3.3 Reconstruction Implementation 15

3.3.1 LowBetaAlg 17

3.3.2 Retrieving TRTCHAMP Calibration Priors 18

3.3.3 Compiling TRTCHAMP Input Parameters 19

4 Results **21**

4.1 Comparing the Old and New TRTCHAMP 21
4.2 Velocity Residuals 22
4.3 Monte Carlo Mass Plots 22
4.4 Data Mass Plots 25

5 Discussion 26
5.1 Calibration Database Problems 26
5.2 Refitted Tracks 28

6 Conclusions 30

Bibliography 32

A Code 34
A.1 LowBetaAlg.h (Wrappers Header) 34
A.2 TrtToolsWrapper.cxx (Wrappers Body) 37

List of Figures

Figure 2.1 Cross section diagram of a TRT straw 7
Figure 2.2 A particle track in the TRT 9
Figure 2.3 Track timing diagram for a particle moving with $\beta \approx 1$ 11
Figure 2.4 Track timing diagram for a particle moving with $\beta < 1$ 12

Figure 3.1 Stages in the data acquisition and analysis process 14

Figure 4.1 Relative residuals for the TRTCHAMP velocity estimator . . 23
Figure 4.2 Reconstructed Mass Plot of Monte Carlo Protons 24
Figure 4.3 Reconstructed Mass Plot of Monte Carlo 300 GeV R-Hadrons 24
Figure 4.4 Reconstructed Mass Plot of Real Data Protons 25

Glossary

AOD Analysis Object Data

ATLAS A Toroidal Lhc ApparatuS, one of the detectors at the LHC

CHAMP CHArged Massive Particle

ESD Event Summary Data

LHC Large Hadron Collider, a particle accelerator in Geneva, Switzerland

LSP Lightest Supersymmetric Particle

ROOT A C++ object-oriented framework for high energy physics

SMP Stable Massive Particle, a long-lived massive particle

SUSY SUperSYmmetry

TRT Transition Radiation Tracker, a subdetector in ATLAS

TRTCHAMP The program used to search for CHAMPs in the TRT

WIMP Weakly Interacting Massive Particle

Acknowledgments

I am grateful to the UBC ATLAS group for all of the support I have received over the past year, and for fostering my knowledge of particle physics. In particular, I would like to thank my supervisor, Prof. Colin Gay, for giving me the opportunity to pursue my academic dreams.

I also wish to give my most sincere thanks to Bill Mills, of the UBC ATLAS group. His continual assistance and encouragement, thoughout the course of my thesis, was greatly appreciated.

Chapter 1

Introduction

The Large Hadron Collider (LHC), based in Geneva, Switzerland, is currently the highest energy particle accelerator in the world. Now that the LHC is operational and colliding beams of protons as intended, the search for new physics has begun. With a current collision energy of up to 3.5 TeV per beam, there is the potential for the discovery of new particles, which would be monumental. The Standard Model of particle physics predicts the existence of an as of yet unobserved particle, known as the Higgs boson. The search for the Higgs boson has been highly publicized, and the discovery of the Higgs boson would be a major accomplishment that is theorized to be within the LHC energy range[11].

While the Higgs boson would certainly be a notable discovery, it is not the only new particle that may be discovered. There may be many other exotic particles just waiting to be found by the LHC.

1.1 New Particles

With higher collision energies, more massive particles can be discovered. This is due to the relation between energy and matter, as laid out by the relativistic energy-momentum relationship:

$$E = \sqrt{(mc^2)^2 + (pc)^2} \quad \Longrightarrow \quad m = \frac{1}{c^2}\sqrt{E^2 - (pc)^2}$$

New particles will have to be extremely massive to have not been discovered

by previous particle collider experiments, such as those conducted at Fermilab. Of these massive particles, some of them are expected to be stable, earning them the name of Stable Massive Particles (SMPS). In the recent past, charged SMPs were also known as long-lived CHArged Massive Particles (CHAMPS), but more recently the designation of SMPs has been used in academic papers. Many such SMPs are theoretically observable by the LHC detectors, such as the A Toroidal Lhc ApparatuS (ATLAS) Experiment's Hadronic Calorimeter[2]. In fact, such particles would stand out in collisions, due to their large mass resulting in less available space for kinetic energy. As all known particles will be moving at approximately the speed of light in a 3.5 TeV per beam collision, any particle with a large mass that is detected to be moving significantly slower is an exotic particle candidate.

The search for SMPs has a strong theoretical backing, including the support of SUperSYmmetry (SUSY), one of the prominent theoretical models of particle physics beyond the Standard Model. There are a large number of supersymmetric models, many of which predict the existence of some form of SMP. In SUSY, each particle in the Standard Model has a supersymmetric partner particle. Additionally, supersymmetry introduces a new parity, called R-Parity, which has an associated value of +1 for every particle in the Standard Model and -1 for each superparticle. Some SUSY models enforce a conservation of R-Parity, which then requires the Lightest Supersymmetric Particle (LSP) to be stable, as there is nothing that it can decay into. Additionally, the decay of the second lightest supersymmetric particle will be supressed due to only having a single possible decay mode, the LSP. Therefore, this creates a set of possible SMPs to investigate.

The search for SMPs has numerous applications to further research, and could even provide answers to some of the mysteries of high energy physics. For example, if a supersymmetric SMP is discovered, it could provide the first experimental validation of supersymmetry, which in turn could supply solutions to two of the most important puzzles in modern particle physics. First off, SUSY presents a solution to the hierarchy problem related to the Higgs boson by the existence of a superparticle partner for each normal particle, which leads to a cancellation of the quantum corrections to the Higgs mass[9]. Secondly, the LSP is generally expected to be a Weakly Interacting Massive Particle (WIMP), and this WIMP is thought to be a primary candidate for the source of dark matter in the universe[6]. As such,

validating or constraining the model parameters of SUSY by searching for SMPs is a definite possibility at the LHC[5], and would be a major accomplishment.

Supersymmetric particles are by no means the only exotic particle candidates that could be observed at the LHC. However, they are examples from one prominent theory that have been shown to have theoretical masses within the range of the LHC[5], either now at 3.5 TeV per beam or later after the 2012 upgrade to 7 TeV per beam.

1.2 Our Project

Our project focuses on using the Transition Radiation Tracker (TRT), which is part of the inner detector of the ATLAS experiment, to search for SMPs. As such, this paper will focus on the methods used in the ATLAS experiment from now on, unless otherwise specified. The TRT can detect charged SMPs by measuring the momentum and relativistic velocity, β, of particles passing through the detector. If the resulting β value is small compared to the speed of light, then the observed TRT event may involve a new massive particle, which can then be investigated in more detail. This is all of the information needed to determine the mass of a particle, as per the following relativistic momentum equation:

$$\vec{p} = \gamma m \vec{v} \quad \Longrightarrow \quad m = \frac{|\vec{p}|\sqrt{1-\beta^2}}{\beta c}$$

As the LHC continues to operate, our understanding of the collider improves, allowing us to increase its luminosity. Luminosity, a measure of the number of particles per unit area, per unit time, is related to the number of collisions that occur in a given amount of time. Therefore, as the luminosity of the LHC is increased, larger and larger quantities of raw data will be available. Increasing the amount of raw data leads to a data storage problem, as the data acquisition hardware can only process a set amount of information in a given time period. This creates the need for hardware triggers to differentiate between interesting and uninteresting events (individual collisions and their products), and to then only store the raw data for interesting events.

Events that have passed the hardware triggers must then be converted from raw

electrical signals into physical quantities, such as 4-vectors. The process by which this is done is known as reconstruction, and is managed by a program called Athena in the ATLAS experiment. Athena combines the observations of all of the subdetectors in order to recreate the paths of particles as they move through the ATLAS detector, in addition to determining measurements of the particles within each individual detector. For our project, the relevant information is that reconstruction calculates the 3-momentum and direction of particles that pass through the TRT. As such, in order to determine the mass of particles in the TRT, we are left with the requirement of measuring their velocity.

Before this project, the program used to calculate particle velocities in the TRT was executed during data analysis, which is in a separate processing pass multiple steps beyond reconstruction. As a result, this program relied upon a certain set of parameters and data to be generated by the reconstruction stage and retained until the analysis stage. This was adequate at first, but with the luminosity upgrades, the data throughput was becoming too large to handle. To remedy this problem, ATLAS controllers decided to discard some of the data past the reconstruction stage, including some of the parameters the old analysis program depended upon for calculating the velocity of particles. Due to this change, it became very important to migrate the velocity calculation routines from the analysis stage into Athena.

In addition to allowing the search for SMPs in the TRT to continue and decreasing data throughput by around six orders of magnitude[4], this change also provides a key new feature in the reconstruction. In the past, the velocity of particles in the TRT was calculated by and available to only our research group, as it was done with a private algorithm. With the algorithm integrated into the official reconstruction process, the particle velocity estimate is available for anyone to use. This change provides new functionality to Athena, which in turn gives everyone in the ATLAS experiment access to additional information about events that they may wish to analyze.

As the LHC luminosity continues to increase, this project becomes more and more important. Our changes aim to drastically decrease the data throughput required for the search for SMPs in the TRT by integrating a post-processing analysis script into Athena, the official canonical reconstruction for the ATLAS experiment. However, even if there were no data limitations, our project still would be benefi-

cial because it adds additional information, in the form of the relativistic velocity of particles in the TRT, to reconstruction for use by others.

Chapter 2

Theory

In order to fully understand the computational requirements of this project, the means by which the relativistic velocity $\beta = v/c$ is obtained must be discussed. This first requires an explanation of how the TRT conducts its primary purpose, which is to track particles as they move through the detector.

2.1 Tracking Charged Particles in the TRT

The TRT is a combined transition radiation detector and straw tracker, which contains approximately 300,000 straws. Each straw is comprised of a central anode, an outer wall held at a highly negative voltage with respect to the anode, and a gas mixture of 70% xenon, 26% carbon dioxide, and 3% oxygen[13], a cross-section of which is shown in figure 2.1. As a charged particle passes through the straw, it ionizes the gas, which is referred to as a "hit". These electrons then move to the anode due to the influence of the electric field generated by the straw wall to anode potential difference. The anode records the arrival of electrons as electrical signals. These electrical signals can then be used to recreate the path of the charged particle, as the amount of time it takes an ejected electron to reach the anode depends on its distance from the anode. An electron ionized at the straw wall will take 45ns[10] to reach the anode, while an electron that is ejected arbitrarily close to the center of the straw takes essentially zero time to reach the anode. The time at which an electron would reach the anode if the charged particle were to pass arbitrarily close

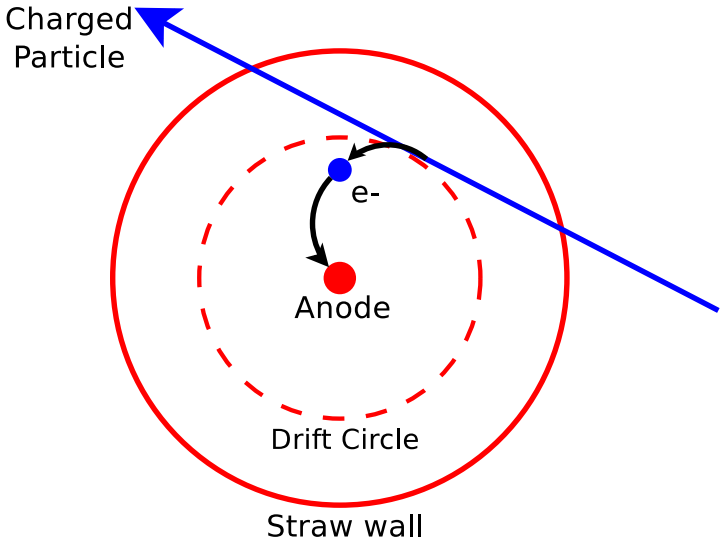

Figure 2.1: The cross section of a straw in the TRT. The red circle in the middle is the anode, while the outer red circle is the straw wall, held at -1500V with respect to the anode. The space in between is filled with a gas comprised of 70%Xe, 27% CO_2, and 3% O_2. As charged particles pass through the straw, they ionize the gas. These electrons move to the anode, due to the potential difference between the anode and straw wall. The anode records the arrival of electrons as electrical signals, and can use the difference between the start time, t_0, and the arrival of the first electron to determine the point of closest approach, which is represented by the drift circle. A series of these drift circles can be used to reconstruct the path of the charged particle.

to the anode is called the t_0 value, and is the start time for TRT calibrations.

On average, a charged particle will generate seven such hits per straw, meaning that the anode will usually record an electron from approximately the straw wall, as well as an electron from approximately the point of closest approach. Additionally, particles are assumed to be moving at nearly the speed of light ($\beta \approx 1$), and therefore each of those hits will occur at approximately the same time relative to how long it takes for the electron to reach the anode. As the time at which the first electron would be observed if the charged particle passed arbitrarily close to the anode is known, and has value t_0, we can determine the point of closest ap-

proach. Specifically, the difference between the t_0 value and the actual arrival time of the first electron can then be used to reconstruct the radius of closest approach for the charged particle, which is known as the drift circle radius, and which has a corresponding drift circle as shown in figure 2.1.

A single drift circle does not give the trajectory of the charged particle, as we now have a circle of points at which the particle may have passed. However, a charged particle also does not pass through just a single straw. We can combine a series of drift circles, corresponding to a series of consecutive straws traversed by the same particle, to reconstruct the path taken. This is possible because the average charged particle in the TRT passes through thirty straws, to generate what is called a track, an example of which is shown in figure 2.2. Using all of the straws in a path allows us to recreate the trajectory of the particle by fitting a curve to be best-tangent to all of the corresponding drift circles, which is also shown in the same figure.

Now that we have a means of recreating the track a particle took through the TRT, we must turn to velocities. Unfortunately, the TRT was not designed to calculate the velocity of charged particles as they pass through the detector. However, we can exploit the assumption that particles are moving with $\beta \approx 1$ to obtain an estimate of their actual velocity.

2.2 Estimating Relativistic Velocities in the TRT

The electrons generated by the traversal of a straw by a charged particle can be used to construct a timing diagram. We know the time at which the first electron can possibly reach the anode, as this corresponds to the particle passing arbitrarily close to the anode. Therefore, this is the earliest possible rising edge time, and if the anode receives an electron before this point, we know that a problem has occurred. Otherwise, the rising edge of the timing diagram is simply the time at which the first electron reaches the anode, which corresponds to the approximate point of closest approach of the charged particle. The falling edge is then the final electron to reach the anode, which is from the straw wall, and therefore should be exactly 45ns after the earliest possible rising edge time. As part of the core functionality of the TRT implementation, it is calibrated under the assumption that

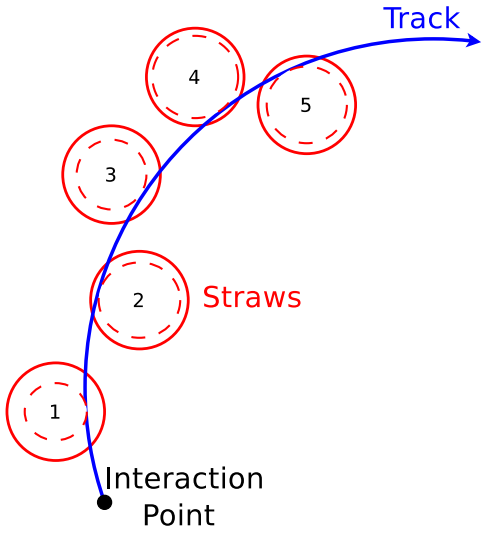

Figure 2.2: A charged particle path passing through five straws to create a track. The solid red circles are straw walls, dashed red circles are drift circles, and the blue path is the reconstructed track of the particle, which follows the drift circles. The interaction point is where the actual event occurred, and the straws are numbered to show the order in which the charged particle passed through them.

charged particles are moving at approximately the speed of light, or with $\beta \approx 1$. This results in all of the falling edges of all of the straws in a track being aligned in a timing diagram, with the straw distance from the interaction point as the vertical axis. If the particle is actually moving near the speed of light, as per the assumption made by the calibration, then the timing diagram works out as shown in figure 2.3.

However, we are interested in particles moving slower than the speed of light. In this case, the ionizing radiation will not arrive at the nominal times corresponding to a track from a $\beta \approx 1$ particle, but will rather be delayed by an amount determined by its actual velocity. This will cause the timing diagrams to have delayed falling edges, as the t_0 value is off. That is, the particle will actually enter a particular straw a time t after the calibration assumed that it would, and therefore the arrival of the first electron in the straw will occur a time t after it was expected to. The value of t will increase with the distance from the interaction vertex, as the

charged particle falls further and further behind the assumption, as shown in figure 2.4. The rate at which the particle falls behind can be related to the delay between the falling edges of sequential straws, which gives us an estimate on the relativistic velocity of the charged particle as it moves through the TRT.

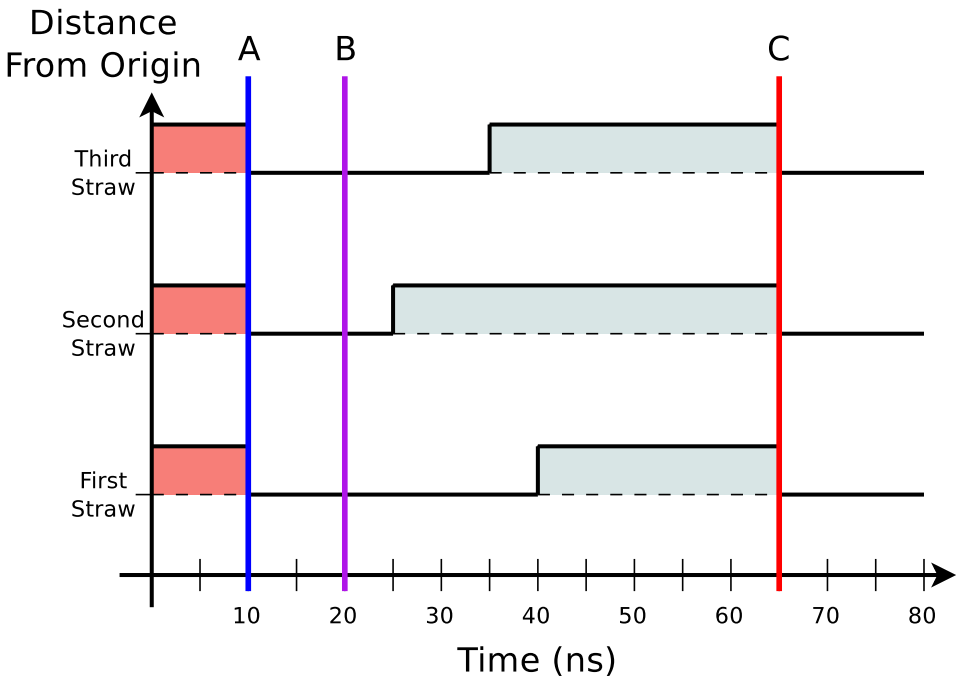

Figure 2.3: Timing diagram for the track of a particle moving with $\beta \approx 1$. The vertical axis contains straws in the track. When the signal for a straw is high, the straw is receiving electrons from the ionized gas. The rising edge is when the first electron reaches the anode (from the closest approach), while the falling edge is the arrival of the final electron (from near the straw wall). Line **A** marks the end of the last event, and **B** marks where the rising edge of the related set of hits could begin if a particle moving with $\beta \approx 1$ passed arbitrarily close to the anode wire. The TRT is calibrated so that, for a particle moving with $\beta \approx 1$, the falling edge of all straws in a track occur at the same time. Line **C** shows this relation with all the falling edges occuring 45ns after line **B**. In this diagram, the particle passes closest to the center of the second straw, characterised by the earliest rising edge, and furthest from the center of the first straw, as seen by the latest rising edge.

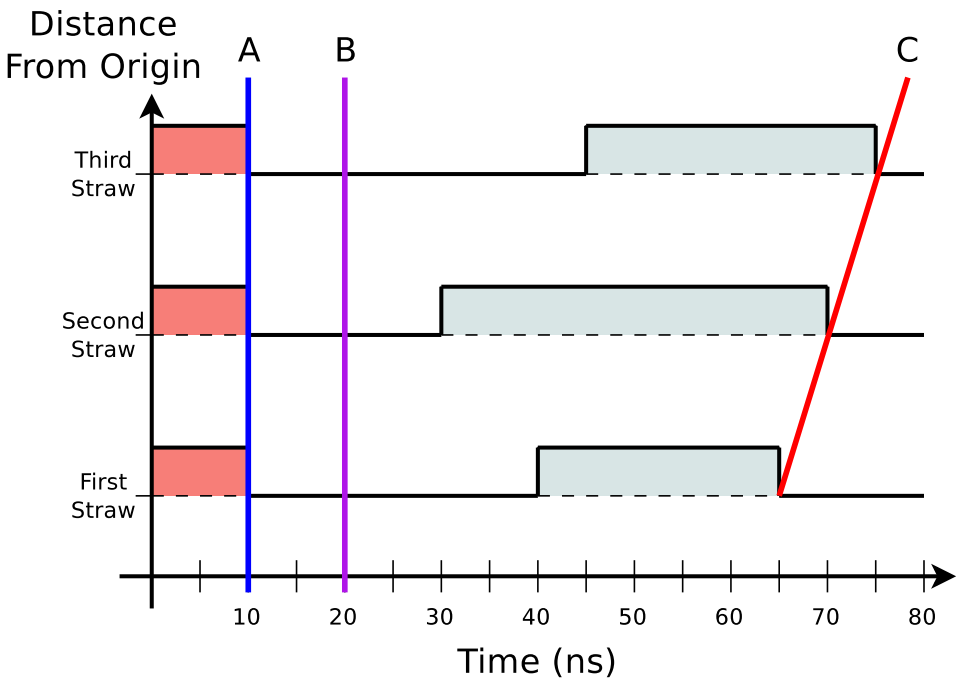

Figure 2.4: Timing diagram for the track of a particle moving with $\beta < 1$. This figure is a continuation of Figure 2.3, where now the particle is moving slower than the speed of light. Lines **A** and **B** still have the same meaning as before. As the TRT is calibrated to line up falling edges for particles moving with $\beta \approx 1$, a particle moving with $\beta < 1$ will fall further and further behind by comparison. In this example, the first straw still observes the final electron at 65ns, while the second track has been delayed by 5ns and the third track by 10ns (relative to a fast particle). Line **C** shows this relation. The delay between sequential falling edges can be used to determine β.

Chapter 3

Methods

Before discussing the actual changes that were made, it is useful to have an understanding of the general process by which data is taken and analyzed in the ATLAS experiment. Figure 3.1 shows the flow of information and sequence of stages of programs when acquiring and analyzing data. The first stage is to select "interesting" events to record, managed by hardware triggers, so that we don't overwhelm the computer systems. The events that are retained are then stored as raw data, which is a digitized electrical signal. In the next step, the electrical signals are turned into physical quantities, such as 4-vectors, through the reconstruction process. This stage is also where selection cuts can be made, meaning software filters can be applied to decide which information is relevant to a particular search or research group. The output of this stage is known as an Event Summary Data (ESD) file. From there, more selection cuts can be made, which results in Analysis Object Data (AOD) files. These AOD files can then be used as input to various analysis scripts to obtain actual results.

3.1 Reasons for Switching to Reconstruction

Before this project, the velocity estimator in the TRT took place in the private analysis script stage, right at the end of the analysis chain, as shown in figure 3.1. The program ran on inputs of ESD files, which created an implicit dependence on particular outputs from the reconstruction stage. This was not a problem before, but

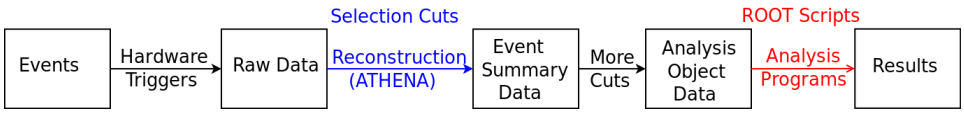

Figure 3.1: The computer programs and intermediary stages involved in acquiring and analyzing data in the ATLAS experiment, including calcuating β in the TRT. Interesting events are selected, digitized, and read out from the TRT via hardware drivers, which results in raw data. Reconstruction converts these digitized electrical signals into physical quantities, and then applies selection cuts, generating an ESD file. More cuts are applied to create AOD files, which are then used as input to user analysis programs. TRTCHAMP used ESD files, although it was run after AOD files were generated. The original program's location in this chain is shown in red, while the velocity estimator's new position as a part of reconstruction is shown in blue.

has become an issue due to a recent change in the ATLAS data retention policy. Due to a significant increase in the luminosity of the LHC, the ATLAS group has been forced to find ways to decrease data throughput, which is the amount of data being written to and read from disk at any given time. To do this, they have elected to phase out ESDs, and have begun by deleting them six weeks after creation. This is important because only ESDs contain hit-level information, which is information that deals with the individual ionization events, as is needed for the timing diagrams used by our β estimator. As such, if anything goes wrong and has to be redone at a later point, this will no longer be possible. This change in the data retention policy therefore requires our research group to integrate the β estimator routine into reconstruction stage if we wish to continue the search for SMPs in the TRT, as there will no longer be a guaranteed way to obtain all of our required inputs as a private analysis algorithm.

Beyond this strict requirement, there are several benefits to our program being part of the official reconstruction. Once added to Athena, our algorithm will be run by default whenever reconstruction is run. Being part of reconstruction means that we will no longer have to run a separate processing pass afterwards to obtain our results, which in turn decreases the amount of time required for our research. Another reason for our program to be a part of reconstruction is that we are calculating

a value of physical significance, which could be used by other research groups in the future. Before our group's project, there was an estimate of the speed of particles as they passed through the TRT added by a different research group. However, our estimate is independent and competitive with theirs, so the two values can be compared to confirm both individual estimates. As such, by adding our program to Athena, other groups could benefit from our work and could use the β value for their own purposes.

3.2 Private Algorithm Implementation

As mentioned, the program was originally run during the user analysis stage, which is the domain of private algorithms. TRTCHAMP was initially written and run using an object oriented framework specifically designed for high energy physics (ROOT). ROOT is an extension of C++, with a set of new libraries, and which is interpreted instead of compiled. This difference was not too important, because the C++ code can still be compiled like a normal program, so long as you properly find and include all of the relevant root libraries. It was a small but crucial step of the project to find the necessary libraries and then compile TRTCHAMP, in order to ensure that no required ROOT functions were missing.

The TRTCHAMP program was written before I started this project, and carries out the actual β estimation given a prescribed set of inputs. The core functionality of the program did not need to be changed at any point, although a few methods for providing alternate forms of inputs were added. The program that calls TRTCHAMP, on the other hand, is what necessitated a major overhaul. As a private algorithm, TRTCHAMP was called by a program that read input parameters from ESD or AOD files. By integrating TRTCHAMP into reconstruction, this was no longer an option.

3.3 Reconstruction Implementation

Reconstruction is a large and complicated task that involves calculating local physical values within each individual sub-detector, and then combining the results from the full ATLAS detector to determine the global value of other physical quantities. This enormous task is managed by a program named Athena, which coordinates

the execution of many different algorithms. Each algorithm run by Athena must conform to a three-stage process of initialization, execution, and finalization. The initialization and finalization stages are intended to be computationally inexpensive, while execution is generally a very intensive process.

Athena itself is also ordered into a set of individual stages, which can be characterised as:

1. General reconstruction setup routines
2. Algorithm initialization stages
3. Algorithm execution stages, where initialization was successful
4. Algorithm termination stages
5. Data output and general reconstruction termination routines

If the initialization stage of a particular algorithm fails, then its respective execution stage is not called. This allows for the initialization stage to test the parameters with which the reconstruction is being run, and to abort if it has detected that the algorithm will not be successful under the given conditions. For example, if the algorithm relies on having access to a particular set of calibration constants and the constants are missing for whatever reason, then there's no point in running the execution stage of the program. In short, the initialization stage generally retrieves data containers necessary for the execution of the algorithm. The finalization stage, on the other hand, generally releases these containers and frees up memory that the algorithm used.

These algorithms are sorted into packages, where each package has a specific, distinct purpose, and a central algorithm. Through coordination with the rest of the research group, and with the help of the package maintainer[8], we obtained space for the results of our β estimation routine in the InDetLowBetaFinder package (Inner Detector Low Beta Finder). The InDetLowBetaFinder package already existed as an official reconstruction algorithm searching for slow moving particles in the Inner Detector (of which the TRT is a part), so the addition of our program to this package simply adds additional information. To be specific, we were given two floats per particle that passes the selection cuts, so the value of our β estimate in

the TRT and the error in that estimate are the two physical quantities that we have added to the reconstruction.

3.3.1 LowBetaAlg

The central algorithm in the InDetLowBetaFinder package is called LowBetaAlg, and it implements the three stage framework just discussed. In order to integrate TRTCHAMP with LowBetaAlg, a set of wrapper functions was created. These wrapper functions serve the purpose of providing an intermediary layer between the two programs, and ensuring that changes can still be made to LowBetaAlg in the future without breaking TRTCHAMP, and vice versa. This is possible because the two are relatively isolated entities that only interact through three very small code snippets, with one corresponding to each stage (initialization, execution, and finalization). In essence, these wrappers create a clear division between LowBetaAlg and TRTCHAMP, while at the same time providing a rigid framework for connecting the two, thereby enforcing a clearly defined interaction.

The initialization and finalization stages of LowBetaAlg call the respective TRTCHAMP wrapper method directly, meaning that so long as the LowBetaAlg method is called, the associated TRTCHAMP method will also be called. However, the execution stage is different. During the execution of LowBetaAlg, selection cuts are applied to determine which particles are considered to be SMP candidates, and therefore which particles we need to calculate the velocity of. To ensure that we only calculate the β of relevant particles, the TRTCHAMP execution wrapper is called after the success of the first round of selection cuts. This is the one possible point of future conflict, as the execution wrapper is not entirely self-contained. It requires a particular data container as input, but LowBetaAlg relies on it extensively as well, so removing the container would require a complete rewrite of the execute stage of LowBetaAlg. As such, so long as LowBetaAlg is not entirely re-written, the TRTCHAMP functionality should not be interfered with.

The wrapper programs for TRTCHAMP fulfill two main purposes. The initialization wrapper ensures that TRTCHAMP has access to the correct calibration constants, while the execution wrapper compiles the necessary data for the input parameters of TRTCHAMP. The finalization wrapper only performs trivial tasks,

and therefore will not be discussed in detail. The code for the wrapper methods is in appendix A. Note that the code for the TRTCHAMP and LowBetaAlg routines are not included, as my project focussed on integrating the two with only minimal changes to the core programs, and therefore they can be treated as black boxes for the most part.

3.3.2 Retrieving TRTCHAMP Calibration Priors

The TRTCHAMP algorithm makes use of a set of calibration values, known as priors, during the β calculation. The set of priors is a measure of the probabilility distribution representing the likelihood of finding the falling edge in each possible time bin, for particles moving with $\beta \approx 1$. That is, there is one prior per time bin. This probability distribution and the associated priors are determined experimentally by recording the falling edges of a large sample set of minimum bias tracks, with momenta of greater than 5 GeV[10]. These tracks will be of particles moving with $\beta \approx 1$, as the heaviest minimum bias track is the proton, and thus a proton moving with 5 GeV will be moving at approximately the speed of light. Therefore, we have a sample set of particles moving with $\beta \approx 1$, and we can record in which timing bins the falling edges for these particles occur, giving us our prior values.

These prior values must be generated on a regular basis in order to stay up to date with various detector conditions and calibrations. As this is a core part of TRTCHAMP, we had to find a way to dynamically retrieve the correct set of priors for a given data set, during the execution of Athena. We were informed that the TRT conditions database supported such functionality[7], although the documentation on how to use the database was either sparse or outdated. The documentation that did exist noted that the database supports tagging[12], meaning that we could tag a particular set of priors as having been active up to a specified time and date, therefore providing a way to ensure that the correct priors for a data set undergoing reconstruction are accessed. This seemed like an elegant solution to the problem, and so we decided to take this approach.

Unfortunately, the TRT conditions database is challenging to work with, as there are lots of little quirks that are easy to circumvent once you know what to do, but hard to figure out in the first place. Again, the lack of recent documentation was

a problem, but thankfully I was in contact with two individuals who had some familiarity with the process. First, I was told to use a local database and link it to the conditions database for testing, and even given a sample local database generation script[7]. Then, I was informed of some little tweaks that were needed to get the local database to be recognized by the conditions database during reconstruction[1]. In the end, I managed to set up a local database with a set of priors, and to link it to the conditions database. From that point, I was able to start on the code that retrieves information from the conditions database, as now there was a database to read priors from.

The code to retrieve data from the conditions database was much simpler, and is contained in the initialization wrapper and an "update" method. The initialization wrapper simply loads a bunch of default priors, specifies the folder of the conditions database to connect with, namely our priors folder, and registers a callback update function with the Athena detector information storage service (detStore). This means that we want to update our default priors to the most recent set of priors in the database. Note that this is not what we want in the future, as we want to work with tags representing timestamps. However, this implementation was intended to lay out the core framework, so that others can add tag support at a later date.

The update function is called after all of the athena algorithm initializations, but before executions. This ensures that TRTCHAMP will not be left using the default priors during execution. In order to replace these default priors, we retrieve a collection of containers from the (currently local) priors folder linked in the conditions database, parse the containers into a format similar to how TRTCHAMP stores the priors it needs to use, and then replaces the default priors being stored in TRTCHAMP with the versions from the database. At that point, the calibration constants we needed have been read from the conditions database, and are ready for use by TRTCHAMP once it is asked to calculate β in the execute stage.

3.3.3 Compiling TRTCHAMP Input Parameters

The execution wrapper, which parses data from two containers into the format expected by TRTCHAMP, is much simpler. The inputs that TRTCHAMP requires are

the transverse momentum and pseudorapidity values associated with a given particle track, and a bunch of information for every straw the particle passed through in the TRT. Vectors are used to store all of the information needed on a per-hit basis. This per-hit information is comprised of a word that describes whether the measurement can be trusted, the global cartesian coordinates of the hit, channel indices, the drift circle radius, the track radius, and the timing calibration. All of this information, except the track radii, can be obtained from the Track container. The Track container, as the name implies, holds all of the general data about a specific particle track. This container is widely used, meaning that it is relatively simple to loop over all of the hits in a track and add the required information to a vector.

The track radii, however, are not used by most groups. As such, it is not included in the Track container, making it much more challenging to obtain this information. These track radii quantify the shortest distance from the anode to the track for each hit. If the track fits to the drift circle tangents perfectly, then the drift circle radius and track radius will be equal. However, we have no reason to assume that we have a perfect fit, and so we need both values for our β estimates.

With TRTCHAMP as a private algorithm using ESDs, we solved the problem by refitting the track of a particle to every ionization event (hit), and then recaclculating the track radii. However, as the program is now a part of reconstruction, we have access to the CombinedIndetTracks container. This container is an unslimmed version of the Tracks container, meaning that it still has all of the track radii. We can access this data by retrieving the container from the Athena event information storage service (evtStore). Once we have done this, we can immediately retrieve the track radii without the need to recalculate them from the ionization events. We had to do this when TRTCHAMP was a private algorithm, so our new implementation has the added bonus of saving some cpu time. More importantly, this solves the primary problem that this project set out to address. As the track radii and all of the other required data are obtained during reconstruction and processed immediately, TRTCHAMP no longer relies on ESDs, and therefore we no longer have to worry about them being phased out.

Chapter 4

Results

4.1 Comparing the Old and New TRTCHAMP

As the goal of this project was to integrate the existing TRTCHAMP implementation into the official reconstruction, the most logical way to test that everything is working is to compare the output of the two programs when run on the same data set. If the program has been integrated properly, then the two versions should give the same β estimates. After I completed the addition of TRTCHAMP and the associated wrappers to the InDetLowBetaFinder package, I sent the code to Bill Mills[10], the author of TRTCHAMP. He then performed various tests, and confirmed that the two programs were giving the same results, to within our precision. The small differences beyond our precision are due to the use of slightly different track radius fitting techniques. The technique used in Athena can always be changed, but there is no indication that any one of the options for fitters will outperform the others in an outright majority of cases, and so we decided to stick with the default fit that had already been conducted, thereby saving computational time. Regardless of this minor difference, the integration can be said to be successful, as the differences are small enough that the results of the two different TRTCHAMP implementations are identical to within our precision.

4.2 Velocity Residuals

One way to confirm that the β estimator is working correctly is to generate a monte carlo data set of particles with known velocities. If this data is then used for reconstruction and TRTCHAMP, then you know what the actual velocity of each particle is, and can therefore determine the error in the estimate. If β_e is our estimate from TRTCHAMP and β_t is the true velocity used to generate the data set, then the relative residuals are defined as:

$$\frac{\beta_e - \beta_t}{\beta_t}$$

If we plot the relative residual value on the x-axis and the number of occurrences of this residual on the y-axis, we get figure 4.1. We can then fit a curve to the peaks to determine how accurate our estimator is, which shows us that they are very nearly centered around an error of zero, and have a small standard deviation from zero. This tells us that our estimator is performing reasonably well. The secondary peaks in this figure correspond to fake $\beta \approx 1$ particles, which occur so often in minimum bias tracks that they were not completely supressed.

4.3 Monte Carlo Mass Plots

Now that we have confirmed the velocity estimator is working as intended, we can confirm that our mass estimates are accurate too. To do this, we created a set of monte carlo minimum bias tracks. As protons are the heaviest stable standard model particle, they will be the primary constituent of any minimum bias track in the TRT. Therefore, if we apply TRTCHAMP to this data set, we should get the mass of the proton, which is 938.272 MeV. It turns out, as shown in figure 4.2, that we do get this result, to within our uncertainty. The barrel of the TRT gives a mass of 938 ± 0.3 MeV, with a standard deviation of 84 ± 0.3 MeV, while the endcap gives a mass of 923 ± 0.5 MeV and a standard deviation of 85 ± 0.5 MeV. As such, the barrel gave exactly the result we wanted, while the endcap gave the expected result to within one standard deviation.

In order to try something slightly more exotic, consider the R-Hadron. The R-Hadron is a theorized supersymmetric particle, comprised of a supersymmetric particle and at least one Standard Model quark. There are many predicted R-

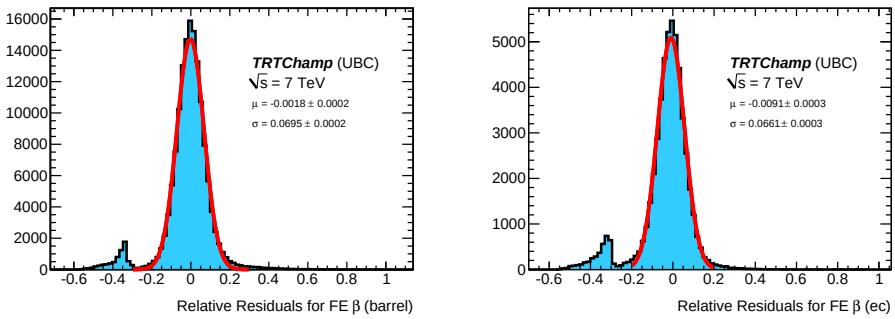

Figure 4.1: A pair of plots for the relative residuals of the velocities of protons selected from minimum bias tracks, as estimated by TRTCHAMP, created by Bill Mills[10]. The x-axis is the value of the relative residual, as defined by ([estimated β] - [true β])/[true β], while the vertical axis is the numer of occurrences of that residual value in the sample. The left figure is the results from the barrel of the TRT, and the right figure is the results from the endcap of the TRT. As can be seen, both sets of residuals are centered very close to zero, and have a small standard deviation, so the estimator is performing well. The secondary peaks are due to fake $\beta \approx 1$ particles, which are so numerous in minimum bias tracks that they can't all be perfectly supressed.

Hardrons, but it is generally expected that, if they exist, they will have a mass of greater than 100 GeV. This limit is due to constraints from past experiments, such as those conducted with the Tevatron. Similarly to the previous case, let us create a monte carlo sample of 300 GeV R-Hadrons, and if our TRTCHAMP returns a mass of 300 GeV, then that's more proof that our program is working properly. Figure 4.3 shows the results, where the barrel estimates are a mass of 298 ± 1.1 GeV with a standard deviation of 41 ± 0.9 GeV, and an endcap mass estimate of 286 ± 1.2 GeV with a standard deviation of 36 ± 1.1 GeV. Therefore, the estimates of the particle mass are relatively close to the actual (300 GeV) mass. This continues to show that our algorithm is properly estimating the mass of SMPs being observed in the TRT.

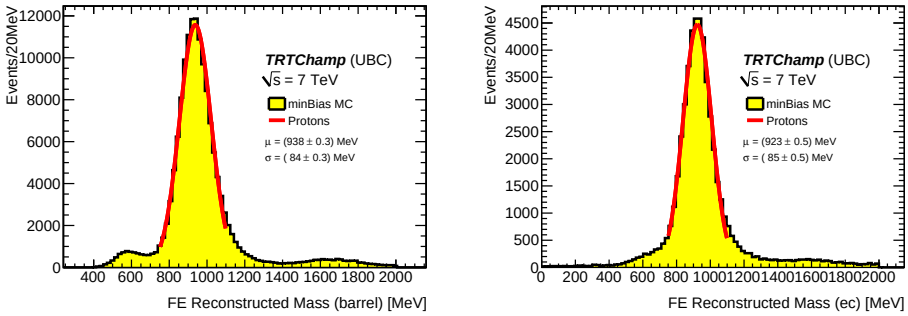

Figure 4.2: Mass plot of a monte carlo proton data set, created by Bill[10]. The x-axis is the mass calculated for a given particle, in MeV, with bins of size 20 MeV. The vertical axis is the number of times a particle with the specified mass was observed. The left plot is for the barrel of the TRT, and the right plot is for the endcap. Fitting a curve to the peak gives the estimated mass and standard deviation in the mass, which are easily within one standard deviation of the actual proton mass of 938.272 MeV.

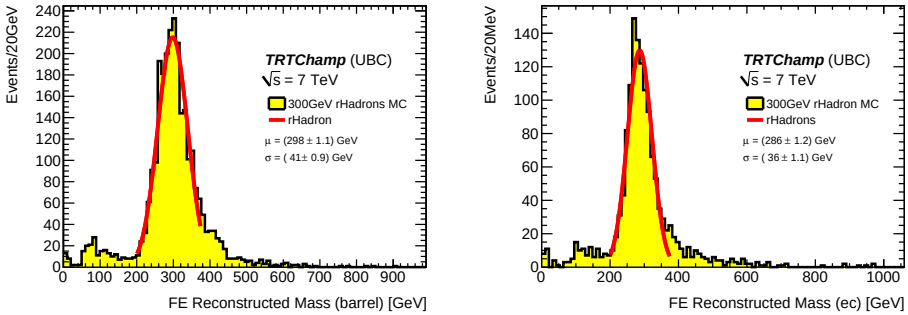

Figure 4.3: Mass plot of a monte carlo 300 GeV R-Hadron data set, created by Bill Mills[10]. The x-axis is the mass calculated for a given particle, in GeV, with bins of size 20 GeV. The vertical axis is the nuber of times a particle with the specified mass was observed. The left plot is for the barrel of the TRT, and the right plot is for the endcap. Fitting a curve to the peak gives the estimated mass and standard deviation in the mass, which are easily within one standard deviation of the input R-Hadron mass of 300 GeV.

4.4 Data Mass Plots

The final set of results is of a proton mass plot with actual data. As before, we are expecting a mass of 938.272 MeV/c^2. Looking at figure 4.4, we do get this expected result. Specifically, the data gives us a mass estimate of 946 ± 0.2 MeV with a standard deviation of 102 ± 0.2 MeV in the barrel of the TRT, and a mass of 914 ± 0.4 MeV with a standard deviation of 115 ± 0.5 MeV in the endcap. The actual proton mass is within significantly less than one standard deviation of the mass estimates given by both the barrel and endcap measurements. This means that we get the expected results with actual data, which strongly implies that the TRTCHAMP program is working as intended.

As a brief note, figure 4.4 is less accurate than the previous figures due to having more than one source of error. For the monte carlo simulations, the exact momentum is known, as it was one of the input parameters. Therefore the monte carlo uncertainties are due to solely the β estimate. On the other hand, the real data set involved using both the measured momentum and velocity, and both of these quantities have associated uncertainties.

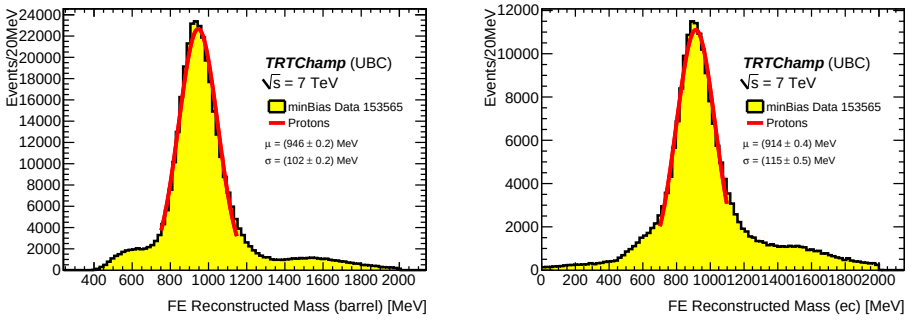

Figure 4.4: Mass plot for a proton from an actual data set, created by Bill Mills[10]. The x-axis is the mass calculated for a given particle, in MeV, with bins of size 20 MeV. The vertical axis is the number of times a particle with the specified mass was observed. The left plot is for the barrel of the TRT, and the right plot is for the endcap. Fitting a curve to the peak gives the estimated mass and standard deviation in the mass, which are easily within one standard deviation of the actual proton mass of 938.272 MeV.

Chapter 5

Discussion

Throughout this project, I encountered numerous problems stemming from a lack of organization in the ATLAS experiment's computing framework. This absence of organization was displayed in many ways including, but not limited to, a sparse documentation, a lack of high-level support algorithms to perform common tasks, obfuscated job control options, and large log files. These all came up while trying to set up a local database to link with the calibration database, as discussed in section 3.3.2, and so I shall discuss this example in more detail below.

5.1 Calibration Database Problems

While attempting to implement a connection to the conditions database, I went to the ATLAS TWiki[12], which is the general source of documentation, and searched for "conditions database". The first link gives a summary that looks like it might explain the database, but when you click on the link, it says that the page no longer exists. Then, looking at the other results turned up by the search, there is none that is immediately apparent as dealing with the conditions database. The second page actually does have a useful link, but you won't know this if you don't already know that the conditions database is implemented in the COOL language, and therefore is listed as "AthenaCool". It was only through talking with others[7] and reading several related TWiki pages that I realized that the conditions database and AthenaCool database are the same thing. With this knowledge, you would

then expect the AthenaCool documentation to provide the answer of how to link the a local database into AthenaCool. It does give an explanation, but it turns out that the syntax is incorrect, because it's from a different revision of the ATLAS software. To find the correct syntax, you can look through other people's code, thanks to LXR[3]. LXR is quite useful, but you have to already know what you're looking for. Now, you have the right syntax for how to connect a local database to the conditions database.

From there, the next step is to add a command to the job options so that the database is accessible at runtime. However, with the current job transform routines, this process is obfuscated. The job transforms are intended to handle all requests in a user-friendly way, and they do if you want to stick to a simple tasks. However, as soon as you have to change the job options for an individual stage of the reconstruction, it becomes quite challenging. By reading through the job transform routine, you can find that another routine is called, which in turn calls "skeleton.RAWtoESD.py". This file contains the job options for the raw data to ESD stage of reconstruction, and is where you have to put the command specifying to link the local database into the conditions database. This effort to hide the actual location of the job options from the user may not be intentional, but either way, it was hard to find.

At this point, I must mention that all of this could be written as a high-level support algorithm. The steps that are taken would be very similar for any other addition to the conditions database, and so it would be immensely useful for a method to be written that performs all of the necessary actions for you, given a couple of input parameters. There is no need for every group that needs to work with the conditions database to have to go through the arduous process of setting it up step by step.

Now that everything is theoretically working, you have to test your code, which is done by running reconstruction. However, reconstruction takes 15+ minutes on a data set containing a single event, so this is no quick task. Then, a 75,000+ line log file is created detailing the reconstruction process. You have to know what types of errors to look for and where in the file they might appear if you want to have a chance of debugging any problems that may have occurred. It is possible to only view warnings and errors in the log file, but often you need the information around

the problems to determine possible causes, and therefore it's generally impractical to work only with warnings and errors.

5.2 Refitted Tracks

The calibration database was a nice example because it showed all of the problems at once, but it was by no means the only time I encountered each of those problems. Before we found out about the CombinedInDetTracks container, we thought that it was necessary to refit the tracks to obtain the track radii during reconstruction, like we did when TRTCHAMP was a private algorithm (discussed in section 3.3.3). To do that, we first had to find a flag somewhere in the inner detector job options file, and switch it to be on, thereby providing track refitting capabilities. I started out by checking the TWiki[12], to no avail. Next, I used LXR[3], and managed to find the name of the flag that I wanted. However, I once again had to make use of the well hidden "skeleton.RAWtoESD.py" file to actually enable this option.

Once enabled, reconstruction started failing. After searching through the enormous log file, I managed to find a line that told me the refitting procedure was being called more than once, and therefore there were conflicts preventing reconstruction from executing properly. After going back to the TWiki to read more about job transforms, I had an idea of what the problem might be. I knew that I had to set the flag in a conditional statement, but I didn't know the name of the property that the condition depended upon. From there, I had to use LXR to go through the source even more, and I found the actual job control script that decides when the refit flag is on and when it's off. That allowed me to determine which flag to watch for before enabling the refit, and to make the appropriate changes. With that change, the reconstruction was successful.

Later on, we were encountering more problems from the refit scheme, and asked the track reconstruction experts for help. Their response was to ask why we were refitting in reconstruction instead of just using the CombinedInDetTracks container, as the refit was never meant to be run during reconstruction[14]. We now use the CombinedInDetTracks container, as it's easier and fixed all the problems we had. Even now that I know the name of the container, I can search for it on the TWiki and get nothing, as the page for the container has not been created.

A brief glance shows that other pages make reference to the container, but only because they are using it, not because they are explaining what it is. This is a perfect example of a large problem that could have been entirely avoided with better documentation.

Chapter 6

Conclusions

The discovery of a new particle would be a major breakthrough, and could provide the first conclusive evidence of physics beyond the Standard Model. The existence of SMPs is predicted by multiple theories, notably including R-parity conserving variants of SUSY. Such SMPs should be observable in the ATLAS TRT, and if observed, the mass of such particles can be calculated from the momentum and velocity of the particle. The momentum of the particle is already measured, and so TRTCHAMP was written to estimate the velocity of charged particles as they pass through the TRT.

As a private algorithm, TRTCHAMP occurred after the raw data was converted into physical quantities, through reconstruction, and stored as ESD files. However, to deal with increasing LHC luminosities and therefore increasing amounts of data, ATLAS has recently changed its data retention policy towards phasing out ESD files. Unfortunately, the new output formats don't contain all of the information TRTCHAMP requires. To resolve this problem, TRTCHAMP needs to be integrated into the reconstruction stage, so that it has access to all of the data that it needs.

In order to do this, the InDetLowBetaFinder package was modified to include two additional floats per particle that passes a set of SMP selection cuts, where the two floats represent the β estimate and the standard deviation in that estimate. The primary algorithm in InDetLowBetaFinder, LowBetaAlg, was then linked to TRTCHAMP through a wrapper interface. This interface was designed to minimize the chance of changes to either LowBetaAlg or TRTCHAMP impacting the

other algorithm, and performed three key duties. First off, the wrappers retrieved a set of specialized calibration constants from the TRT conditions database, which TRTCHAMP needs to function. Next, the wrapper obtained the inputs necessary for the execution of TRTCHAMP through reconstruction data containers, and reformed them in the way that TRTCHAMP expects. Finally, the wrappers call the TRTCHAMP algorithm, and store the results in the two aforementioned floats.

Through this change, the wrappers ensure that the TRTCHAMP algorithm functions as it used to, although it is now a part of reconstruction instead of being an independent analysis algorithm. These modifications allow the TRTCHAMP algorithm to continue to run, despite the changes to the ATLAS data retention policy. Additionally, as the β estimator is now a part of reconstruction, the value is now available for others to use in their own research, and therefore this provides benefits to the ATLAS collaboration, rather than just our research group.

In section 4, we have shown that the algorithm accurately calculates the velocity of charged particles in the TRT. We have also shown how this can then be used to determine the mass of the charged particle, for both monte carlo data sets and using real data. Therefore, the TRTCHAMP algorithm remains a valuable part of the TRT search for SMPs, and may help in the discovery or further analysis of new particle(s).

In the future, the interaction with the calibrations database could be improved. As mentioned in section 3.3.2, the wrapper retrieves the most recent copy of the calibration priors from the conditions database. This works for now, but shoud be changed in the future to include tagging. That is, sets of priors should be tagged with a range of when they were valid, and the initialization wrapper should then check the time stamp of the data set that reconstruction is being run upon, and from that determine which set of priors to read from the calibrations database. Alternatively, there may be other ways to obtain the priors. A lot of work has been required to do anything with the calibration database, and there may be some other system of maintaining priors that is easier to manage. Others in the lab are currently looking into such alternate approaches[10].

Bibliography

[1] A. Alonso. Personal communication, 2011.
→ page(s) 19

[2] Shoji Asai et al. Measuring lifetimes of long-lived charged massive particles stopped in lhc detectors. *Phys. Rev. Lett.*, 103(14):141803, Oct 2009.
→ page(s) 2

[3] Brookhaven National Laboratory. LXR, 2011. URL http://alxr.usatlas.bnl.gov/.
→ page(s) 27, 28

[4] C. Gay. Personal communication, 2010-2011.
→ page(s) 4

[5] I. Hinchliffe et al. Precision susy measurements at cern lhc. *Phys. Rev. D*, 55 (9):5520–5540, May 1997.
→ page(s) 3

[6] Gerard Jungman et al. Supersymmetric dark matter. *Physics Reports*, 267 (5-6):195 – 373, 1996. ISSN 0370-1573.
→ page(s) 2

[7] E. Klinkby. Personal communication, 2011.
→ page(s) 18, 19, 26

[8] C. Marino. Personal communication, 2011.
→ page(s) 16

[9] S. P. Martin. A supersymmetry primer. *arXiv e-print*, 1997. URL http://arxiv.org/abs/hep-ph/9709356v5.
→ page(s) 2

[10] B. Mills. Personal communication, 2010-2011.
→ page(s) 6, 18, 21, 23, 24, 25, 31

[11] M. Spira et al. Higgs boson production at the lhc. *Nuclear Physics B*, 453 (1-2):17 – 82, 1995. ISSN 0550-3213.
→ page(s) 1

[12] The ATLAS Collaboration. The ATLAS TWiki, 2011. URL https://twiki.cern.ch/twiki/bin/view/Atlas/WebHome.
→ page(s) 18, 26, 28

[13] The ATLAS TRT collaboration et al. The atlas trt barrel detector. *Journal of Instrumentation*, 3(02):P02014, 2008.
→ page(s) 6

[14] A. Wildauer. Personal communication, 2011.
→ page(s) 28

Appendix A

Code

A.1 LowBetaAlg.h (Wrappers Header)

This is the header file that contains all of the methods used in the main LowBetaAlg routines, as well as all of the wrapper methods. The wrapper methods are all at the bottom of the file, under "private".

```
//////////////////////////////////////////////////////////////////
// LowBetaAlg.h, (c) ATLAS Detector software
//////////////////////////////////////////////////////////////////

/* class LowBetaAlg
LowBetaAlg is an algoirthm for the identification of Charged Stable
Massive Particles based on tracking information mainly from the TRT.
Timing information and energy deposition are used to indentify
candidate tracks and make measurement of beta and excess ionization
comparing to relativistic particles.

author Christopher.Marino <Christopher.Marino@cern.ch>
*/

#ifndef LOWBETAALG_H
#define LOWBETAALG_H

#include "AthenaBaseComps/AthAlgorithm.h"
#include "TRT_ConditionsServices/ITRT_CalDbSvc.h"

//////////////////////////////////////////////////////////////////////////
```

```
class AtlasDetectorID;
class Identifier;
class TRT_ID;
class IMagFieldAthenaSvc;
class TrtToolUBC;



// Predeclare histogram classes that you use.


namespace InDetDD{ class TRT_DetectorManager; }



namespace Trk {class Track;}

namespace Rec { class TrackParticleContainer;
                class TrackParticle; }

namespace InDet
{


  class LowBetaAlg:public AthAlgorithm {
  public:
    LowBetaAlg (const std::string& name, ISvcLocator* pSvcLocator);
    StatusCode initialize();
    StatusCode execute();
    StatusCode finalize();

    std::vector<float> ChargedSMPindicators(const Trk::Track& track);



  protected:

    const TRT_ID*                 m_trtId;                // TRT ID helper

    const InDetDD::TRT_DetectorManager* m_TRTdetMgr;      // TRT detector manager (to get ID helper)

    unsigned int                  m_minTRThits;           // Minimum number of TRT hits to give PID.
    float                         m_RcorrZero;            // Inputs for R Correction
    float                         m_RcorrOne;             // Inputs for R Correction
    float                         m_RcorrTwo;             // Inputs for R Correction
    float                         m_TimingOffset;         // timing offset for trailing bit time
```

```
  //std::string m_tracksName;                          //!< Name of track container in StoreGate
  std::string m_trackParticleCollection;               //!< Name of track container in StoreGate
  std::string m_InDetLowBetaOutputName;              //!< Name of output container to store results

  /** trying to get ahold of the TRT calib DB: */
  ServiceHandle<ITRT_CalDbSvc> m_trtconddbsvc;

  ServiceHandle<IMagFieldAthenaSvc> p_MagFieldAthenaSvc;

private: // Functions/variables for using TrtToolsUBC, see TrtToolsWrapper.cxx

  ////////
  //////// Member variables needed for TrtToolUBC to function properly
  ////////

  // The actual TrtToolUBC class instance (all the rest here is the wrapper)
  TrtToolUBC* m_TrtTool;
  // boolean value that specifies whether TrtToolUBC was initialized successfully
  bool m_TrtToolInitSuccess;
  // Track refit container
  std::string m_UnslimmedTracksContainerName;

  ////////
  //////// ATHENA function equivalents (call from corresponding function)
  ////////
  StatusCode initializeTrtToolUBC();
  int finalizeTrtToolUBC();

  ////////
  //////// Conditions Database interaction functions
  ////////

  // Function for updating the TRT conditions database entries
  // Adapted from TRT_DriftFunctionTool.cxx
  StatusCode update(IOVSVC_CALLBACK_ARGS);

  ////////
  //////// Wrapper functions for calling TrtToolUBC
  ////////

  // A wrapper intermediary for the TrtToolUBC::TRT_FEbeta function
  // Meant to be called from LowBetaAlg.cxx, via InDet::LowBetaAlg::ChargedSMPindicators
  // Returns a vector of results from TRT_FEbeta where:
  //   vector[0] = LikelihoodBeta
  //   vector[1] = LikelihoodError
  std::vector<float> callTrtToolUBC(const Trk::Track& track);
```

```
    // Gather all of the necessary data that the TRT_FEbeta function takes as inputs
    // Sets all of the input arguments
    StatusCode parseDataForTrtToolUBC(const Trk::Track& track, std::vector<int>* TRT_bitpattern,
                std::vector<int>* TRT_bec, std::vector<int>* TRT_strawlayer, std::vector<int>* TRT_layer,
                std::vector<float>* TRT_t0, std::vector<float>* TRT_R, std::vector<float>* TRT_R_track,
                std::vector<float>* TrackX, std::vector<float>* TrackY, std::vector<float>* TrackZ,
                float* RecPt, float* RecEta);

  };

}//end of namespace
#endif // LOWBETAALG_H
```

A.2 TrtToolsWrapper.cxx (Wrappers Body)

This file contains all of the wrapper methods discussed in chapter 3, and is the core part of my project. These methods are the interaction layer that allows LowBetaAlg, which is an Athena algorithm, and TRTCHAMP, a private algorithm, to communicate. The file has been slightly modified to fit onto the page.

```
#include "InDetLowBetaFinder/LowBetaAlg.h"
#include "InDetLowBetaFinder/TrtToolsUBC.h"

// Athena Control and Services
#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/ISvcLocator.h"
#include "StoreGate/StoreGateSvc.h"

// Track headers
#include "TrkTrack/Track.h"
#include "TrkTrack/TrackCollection.h"
#include "TrkParticleBase/TrackParticleBaseCollection.h"
#include "TrkParticleBase/TrackParticleBase.h"
#include "Particle/TrackParticleContainer.h"
#include "Particle/TrackParticle.h"

// Other headers
#include "InDetRIO_OnTrack/TRT_DriftCircleOnTrack.h"
#include "InDetPrepRawData/TRT_DriftCircle.h"
#include "InDetIdentifier/TRT_ID.h"
```

```
// Prints the input of TRT_FEbeta() to file (hard-set)
// This is a temporary debug method, and may be removed later
int printTrtToolUBCDebugFile(std::vector<int> TRT_bitpattern,
        std::vector<int> TRT_bec, std::vector<int> TRT_strawlayer,
        std::vector<int> TRT_layer, std::vector<float> TRT_t0,
        std::vector<float> TRT_R, std::vector<float> TRT_R_track,
        std::vector<float> TrackX, std::vector<float> TrackY,
        std::vector<float> TrackZ, float RecPt, float RecEta);


StatusCode InDet::LowBetaAlg::initializeTrtToolUBC()
{
        // Declare the track refit container
        declareProperty("UnslimmedTracksContainer",
                m_UnslimmedTracksContainerName="CombinedInDetTracks");

        // Create the TrtTool
        m_TrtTool = new TrtToolUBC();

        // Load the default prior values
        m_TrtTool->TRT_LoadDefaultPriors();

        // Register a function for obtaining priors from the TRT conditions database
        const DataHandle<CondAttrListCollection> collectionHandle;
        StatusCode SC = detStore()->regFcn(&InDet::LowBetaAlg::update,
                this,collectionHandle,"/TRT/Calib/MLbetaPriors");
        if (SC.isFailure())
          ATH_MSG_WARNING("Callback registration failed for LowBetaAlg - priors." +
                "  Using default prior values.");
        else
          ATH_MSG_INFO("Registered callback for updating LowBetaAlg - priors");

        return StatusCode::SUCCESS;
}

int InDet::LowBetaAlg::finalizeTrtToolUBC()
{
        // Done with the TrtTool
        delete m_TrtTool;

        return 0;
}
```

```
// Function for updating the TRT conditions database entries
// Adapted from TRT_DriftFunctionTool.cxx
StatusCode InDet::LowBetaAlg::update(IOVSVC_CALLBACK_ARGS_P(I,keys))
{
        const bool doDebug = false;      // set to true to print to file below
        const char* DEBUGFILE_C = "/afs/cern.ch/user/s/sschramm/testarea/16.0.2/ +
                "InnerDetector/InDetRecAlgs/InDetLowBetaFinder/debugPriors.out";
        FILE* dFile = NULL;

        ATH_MSG_INFO("Updating priors for the LowBetaAlg likelihood beta estimator");

        // Callback function to update priors when condDB data changes:
        for(std::list<std::string>::const_iterator key=keys.begin();
                key != keys.end(); ++key)
        ATH_MSG_DEBUG("IOVCALLBACK for key " << *key << " number " << I);

        // Read the priors
        const CondAttrListCollection* collection;
        StatusCode SC = detStore()->retrieve(collection, "/TRT/Calib/MLbetaPriors");
        if (SC.isFailure() || collection == 0)
        {
          ATH_MSG_ERROR("A problem occurred while reading a conditions database object.");
          return StatusCode::FAILURE;
        }
        else // Successfully retrieved
        {
          // Ensure that the collection is the same size as the priors
          // (Make sure we're replacing everything, not just a portion of the priors)
          if (collection->size() != m_TrtTool->TRT_NumPriors())
            ATH_MSG_WARNING("Unexpected number of priors retrieved from the condDB " +
                        "(got " << collection->size() << ", expected " <<
                        m_TrtTool->TRT_NumPriors() << ").  Using defaults.");
          else
          {
            // If debugging, open the output file and print the header
            if (doDebug)
              dFile = fopen(DEBUGFILE_C,"w");
            if (dFile != NULL)
              fprintf(dFile,"#####\n##### prior[etaIndex][barrelOrEndcap]" +
                        "[radiusIndex]\n#####\n\n");

            int channel;
            char name[25];
            double* bitValues;

            // Loop over the changes
            for (CondAttrListCollection::const_iterator iter = collection->begin();
```

```
    iter != collection->end(); ++iter)
{
  channel = abs(iter->first);

  // channel will be in one of two forms
  // if channel >= 100000:
  //        channel is in the form 1AABCC (each one is a single base-10 digit)
  //        AA = eta bin index (0 to 99)            ["i" in TRT_LoadPriors]
  //        B = straw type, barrel(0) or endcap(1)  ["k" in TRT_LoadPriors]
  //        CC = radius bin index (0 to 99)         ["l" in TRT_LoadPriors]
  //  else if 0 <= channel < 100000:
  //        channel is in the form ABC (each one is a single base-10 digit)
  //        A = eta bin index (0 to 9)              ["i" in TRT_LoadPriors]
  //        B = straw type, barrel(0) or endcap(1)  ["k" in TRT_LoadPriors]
  //        C = radius bin index (0 to 9)           ["l" in TRT_LoadPriors]
  int etaIndex;
  int barrelOrEndcap;
  int radiusIndex;
  if (channel >= 100000)
  {
    channel -= 100000;
    etaIndex = channel/1000;
    barrelOrEndcap = (channel%1000)/100;
    radiusIndex = channel%100;
  }
  else
  {
    etaIndex = channel/100;
    barrelOrEndcap = (channel%100)/10;
    radiusIndex = channel%10;
  }

  if (    (etaIndex >= 0 && etaIndex <= TrtToolUBC::NETABINS)
       && ( barrelOrEndcap == 0 || barrelOrEndcap == 1)
       && (radiusIndex >= 0 && radiusIndex <= TrtToolUBC::NRFEBINS) )
  {
    const coral::AttributeList &list = iter->second;
    bitValues = (double*)malloc(sizeof(double)*24);

    for (int i = 0; i < 24; i++)
    {
      sprintf(name,"TRT_bit_%d",i);
      bitValues[i] = list[name].data<double>();
      // If debug, print to file
      if (dFile != NULL)
        fprintf(dFile,"prior[%d][%d][%d][TRT_bit_%d] = %10f\n",etaIndex,
            barrelOrEndcap,radiusIndex,i,list[name].data<double>());
```

```
                }
                // If debug, print a new line to file
                if (dFile != NULL)
                  fprintf(dFile,"\n");

                m_TrtTool->TRT_UpdatePriorValues(radiusIndex,etaIndex,
                                barrelOrEndcap,bitValues);
                free(bitValues);
              }
              else if (dFile != NULL) // If debug, print a warning line to file
                fprintf(dFile,"Unexpected result!  Got channel of %d (eta=%d," +
                        "BoE=%d,rad=%d)\n\n",channel,etaIndex,barrelOrEndcap,radiusIndex);

              // If debug, close the file, as we're done now
              if (dFile != NULL)
                fclose(dFile);
            }
          }
        }

        return StatusCode::SUCCESS;
}

// A wrapper intermediary for the TrtToolUBC::TRT_FEbeta function
// Meant to be called from LowBetaAlg.cxx, via InDet::LowBetaAlg::ChargedSMPindicators
// Returns a vector of results from TRT_FEbeta where:
//       vector[0] = LikelihoodBeta
//       vector[1] = LikelihoodError
std::vector<float> InDet::LowBetaAlg::callTrtToolUBC(const Trk::Track& track)
{
        // Return variable
        std::vector<float> LikelihoodValues;

        // Variables to be determined before calling TRT_FEbeta
        //Raw bitpattern
        std::vector<int> TRT_bitpattern;
        //Barrel or endcap index
        std::vector<int> TRT_bec;
        //Strawlayer index
        std::vector<int> TRT_strawlayer;
        //Layer index
        std::vector<int> TRT_layer;
        //t0 values
        std::vector<float> TRT_t0;
        //Drift circle radius
        std::vector<float> TRT_R;
        //Local track radius
```

```
        std::vector<float> TRT_R_track;
        //x position of the hit
        std::vector<float> TrackX;
        //y position of the hit
        std::vector<float> TrackY;
        //z position of the hit
        std::vector<float> TrackZ;
        //pt
        float RecPt;
        //eta
        float RecEta;


        // Load the priors from a file instead of using the conditions db or the defaults
        //const std::string fileName = "/afs/cern.ch/user/s/sschramm/testarea/16.0.2/" +
        //                "InnerDetector/InDetRecAlgs/InDetLowBetaFinder/" +
        //                "InDetLowBetaFinder/TRTpriors.MC.root";
        //m_TrtTool->TRT_LoadPriors(fileName);


        // Gather/parse all of the neccessary data
        StatusCode SC = parseDataForTrtToolUBC(track,&TRT_bitpattern,&TRT_bec,
                &TRT_strawlayer,&TRT_layer,&TRT_t0,&TRT_R,&TRT_R_track,
                &TrackX,&TrackY,&TrackZ,&RecPt,&RecEta);
        if (SC.isFailure())
        {
          LikelihoodValues.clear();
          LikelihoodValues.push_back(-997);
          LikelihoodValues.push_back(-997);
          return LikelihoodValues;
        }

        // We now have everything we need



        // Create the debug file
        //printTrtToolUBCDebugFile(TRT_bitpattern,TRT_bec,TRT_strawlayer,TRT_layer,
        //                TRT_t0,TRT_R,TRT_R_track,TrackX,TrackY,TrackZ,RecPt,RecEta);

        // Now call the actual function we want
        LikelihoodValues = m_TrtTool->TRT_FEbeta(TRT_bitpattern,TRT_bec,
                TRT_strawlayer,TRT_layer,TRT_t0,TRT_R,TRT_R_track,
                TrackX,TrackY,TrackZ,RecPt,RecEta);

        return LikelihoodValues;
}
```

```
// Gather all of the necessary data that the TRT_FEbeta function takes as inputs
// Sets all of the input arguments
StatusCode InDet::LowBetaAlg::parseDataForTrtToolUBC(const Trk::Track& track,
                        std::vector<int>* TRT_bitpattern, std::vector<int>* TRT_bec,
                        std::vector<int>* TRT_strawlayer, std::vector<int>* TRT_layer,
                        std::vector<float>* TRT_t0, std::vector<float>* TRT_R,
                        std::vector<float>* TRT_R_track, std::vector<float>* TrackX,
                        std::vector<float>* TrackY, std::vector<float>* TrackZ,
                        float* RecPt, float* RecEta)
{
        // Variable for non-refit drift circles, to be used as a check
        std::vector<float> TRT_R_notFit;

        // Clear all the vectors
        TRT_bitpattern->clear();
        TRT_bec->clear();
        TRT_strawlayer->clear();
        TRT_layer->clear();
        TRT_t0->clear();
        TRT_R->clear();
        TRT_R_track->clear();
        TrackX->clear();
        TrackY->clear();
        TrackZ->clear();
        TRT_R_notFit.clear();




        // Get the non-refit track particle drift circle radii for checking
        // our refits later
        const DataVector<const Trk::TrackStateOnSurface>* hits =
                track.trackStateOnSurfaces();
        if (hits)
        {

          // Loop over the hits
          DataVector<const Trk::TrackStateOnSurface>::const_iterator hitIterEnd =
                hits->end();
          DataVector<const Trk::TrackStateOnSurface>::const_iterator hitIter =
                hits->begin();
          for (; hitIter != hitIterEnd; ++hitIter)
          {
            //only include this hit if we can call a TRT drift circle out of a
            //measurementBase object based on it
            const Trk::MeasurementBase *measurement = (*hitIter)->measurementOnTrack();
```

```
    if (measurement)
    {
      // Get drift circle (ensures that hit is from TRT)
      const InDet::TRT_DriftCircleOnTrack *driftCircle =
                dynamic_cast<const InDet::TRT_DriftCircleOnTrack*>(measurement);
      if (driftCircle)
      {
        // Check the raw data
        if (driftCircle->prepRawData())
        {
          float driftCircleRadius = -999;
          driftCircleRadius = driftCircle->localParameters()[Trk::driftRadius];
          TRT_R_notFit.push_back(driftCircleRadius);
        }
      }
    }
  }
}

// Determine pt, eta, phi
// Perigee exists if we got here (checked for in
// InDet::LowBetaAlg::ChargedSMPindicators)
const Trk::TrackParameters* perigee = track.perigeeParameters();
const HepVector& parameterVector = perigee->parameters();
double qOverP  = parameterVector[Trk::qOverP];
double theta   = parameterVector[Trk::theta];
double phi0    = parameterVector[Trk::phi0];
if (tan(theta/2.0) < 0.0001)
{
  ATH_MSG_DEBUG("TrtToolUBC aborting due to theta value (tan(theta/2) < 0.0001");
  return StatusCode::FAILURE;
}
double eta = -log(tan(theta/2.0));
if (qOverP == 0.0)
{
  ATH_MSG_DEBUG("TrtToolUBC aborting due to momentum value (q/p == 0)");
  return StatusCode::FAILURE;
}
double pt  = fabs(1.0/qOverP)*sin(theta);

// Set pt and eta
*RecPt  = pt;
*RecEta = eta;
```

```
// Check if the track refit container is in storegate
if (!evtStore()->contains<TrackCollection>(m_UnslimmedTracksContainerName))
{
  // Warn the user that the algorithm failed
  ATH_MSG_WARNING("StoreGate does not contain the CombinedInDetTracks container");
  return StatusCode::FAILURE;
}

// Get the track refit container from storegate
// (We've already checked that it's there)
const TrackCollection* unslimmedTracks(0);
StatusCode SC = evtStore()->retrieve(unslimmedTracks,m_UnslimmedTracksContainerName);
if (SC.isFailure() || !unslimmedTracks)
{
  ATH_MSG_WARNING("Could not retrieve CombinedInDetTracks container");
  return StatusCode::FAILURE;
}

// need to loop over refit container to find the right refit track,
// NOT 1:1 with track particles!
float refitEta = -999;
float refitPhi = -999;
float etaphiCone = 0;
float bestCone = 9999;
TrackCollection::const_iterator MATCH = unslimmedTracks->begin();
for( TrackCollection::const_iterator ITR = unslimmedTracks->begin();
        ITR!=unslimmedTracks->end(); ++ITR)
{
  const Trk::MeasuredPerigee *aMeasPer =
        dynamic_cast<const Trk::MeasuredPerigee*>((*ITR)->perigeeParameters () );
  etaphiCone = sqrt(pow( -log(tan(aMeasPer->parameters()[Trk::theta]/2))
                - eta, 2) + pow( aMeasPer->parameters()[Trk::phi0] - phi0,2));
  if( etaphiCone < bestCone )
  {
    bestCone = etaphiCone;
    refitEta = -log(tan(aMeasPer->parameters()[Trk::theta]/2));
    refitPhi = aMeasPer->parameters()[Trk::phi0];
    MATCH = ITR;
  }
}
```

```
// Check for refit hits (aka track states on surfaces, tsos)
const DataVector<const Trk::TrackStateOnSurface>* refitHits =
        dynamic_cast<const DataVector<const Trk::TrackStateOnSurface>*>
        ((*MATCH)->trackStateOnSurfaces());
if (refitHits)
{
  DataVector<const Trk::TrackStateOnSurface>::const_iterator
        hitIterEnd = refitHits->end();
  DataVector<const Trk::TrackStateOnSurface>::const_iterator
        hitIter = refitHits->begin();
  // Loop over hits
  for (; hitIter != hitIterEnd; ++hitIter)
  {
    //only include this hit if we can call a TRT drift circle out of a
    //measurementBase object based on it
    const Trk::MeasurementBase *measurement = (*hitIter)->measurementOnTrack();
    if (measurement)
    {
      // Get drift circle (ensures that hit is from TRT):
      const InDet::TRT_DriftCircleOnTrack *driftCircle =
                dynamic_cast<const InDet::TRT_DriftCircleOnTrack*>(measurement);
      if (driftCircle)
      {
        // Check the raw data
        if (driftCircle->prepRawData())
        {
          // Raw bit pattern
          TRT_bitpattern->push_back(driftCircle->prepRawData()->getWord());

          // TRT ID information
          Identifier DCoTId = driftCircle->identify();
          int bec = m_trtId->barrel_ec(DCoTId);
          int strawLayer = m_trtId->straw_layer(DCoTId);
          int layer = m_trtId->layer_or_wheel(DCoTId);
          TRT_bec->push_back(bec);
          TRT_strawlayer->push_back(strawLayer);
          TRT_layer->push_back(layer);

          // Get TRT calibration from the database, t0
          Identifier TRTlocal = m_trtId->straw_id(bec,
                        m_trtId->phi_module(DCoTId),layer,
                        strawLayer,m_trtId->straw(DCoTId));
          double t0 = m_trtconddbsvc->getT0(TRTlocal);
          TRT_t0->push_back(t0);

          // Get drift cirlce radius
          float driftCircleRadius = -999;
```

```
                driftCircleRadius = driftCircle->localParameters()[Trk::driftRadius];
                TRT_R->push_back(driftCircleRadius);

                // Get local track radius (track-anode distance)
                const Trk::TrackParameters* hitParam=((*hitIter)->trackParameters());
                float localTrackRadius = -999;
                if(hitParam)
                  localTrackRadius = hitParam->parameters()[Trk::driftRadius];
                TRT_R_track->push_back(localTrackRadius);


                // Get the x, y, and z positions of the hit
                TrackX->push_back(driftCircle->globalPosition().x());
                TrackY->push_back(driftCircle->globalPosition().y());
                TrackZ->push_back(driftCircle->globalPosition().z());
              }
            }
          }
        }
      }

      // Compare the non-refit and refit drift circle radii
      if (TRT_R_notFit.size() != TRT_R->size())
        bestCone = 9999;
      else
        for(unsigned int i = 0; i < TRT_R->size(); i++)
          if ((*TRT_R)[i] != TRT_R_notFit[i])
            bestCone = 9998;

      return StatusCode::SUCCESS;
}
```

```
// Prints the input of TRT_FEbeta() to file (hard-set)
// This is a temporary debug method, and may be removed later
int printTrtToolUBCDebugFile(std::vector<int> TRT_bitpattern,
                std::vector<int> TRT_bec, std::vector<int> TRT_strawlayer,
                std::vector<int> TRT_layer, std::vector<float> TRT_t0,
                std::vector<float> TRT_R, std::vector<float> TRT_R_track,
                std::vector<float> TrackX, std::vector<float> TrackY,
                std::vector<float> TrackZ, float RecPt, float RecEta)
{
        static int trackNum = 1;
        const char* FILENAME_C = "/afs/cern.ch/user/s/sschramm/testarea/16.0.2/" +
                "InnerDetector/InDetRecAlgs/InDetLowBetaFinder/debugFile.out";
        FILE* outFile;
        unsigned int i;

        if (trackNum == 1)
          outFile = fopen(FILENAME_C,"w");
        else
          outFile = fopen(FILENAME_C,"a");

        if (outFile == NULL)
          return -1;

        fprintf(outFile,"#\n#Track Number %d\n#RecPt = %f, RecEta = %f\n#\n",
                trackNum++,RecPt,RecEta);

        fprintf(outFile,"%%Start TRT_bitpattern:\n");
        for (i = 0; i < TRT_bitpattern.size(); i++)
          fprintf(outFile,"\t%d\n",TRT_bitpattern[i]);
        fprintf(outFile,"%%End TRT_bitpattern:\n");

        fprintf(outFile,"%%Start TRT_bec:\n");
        for (i = 0; i < TRT_bec.size(); i++)
          fprintf(outFile,"\t%d\n",TRT_bec[i]);
        fprintf(outFile,"%%End TRT_bec:\n");

        fprintf(outFile,"%%Start TRT_strawlayer:\n");
        for (i = 0; i < TRT_strawlayer.size(); i++)
          fprintf(outFile,"\t%d\n",TRT_strawlayer[i]);
        fprintf(outFile,"%%End TRT_strawlayer:\n");

        fprintf(outFile,"%%Start TRT_layer:\n");
        for (i = 0; i < TRT_layer.size(); i++)
          fprintf(outFile,"\t%d\n",TRT_layer[i]);
        fprintf(outFile,"%%End TRT_layer:\n");

        fprintf(outFile,"%%Start TRT_t0:\n");
```

```
        for (i = 0; i < TRT_t0.size(); i++)
          fprintf(outFile,"\t%f\n",TRT_t0[i]);
        fprintf(outFile,"%%End TRT_t0:\n");

        fprintf(outFile,"%%Start TRT_R:\n");
        for (i = 0; i < TRT_R.size(); i++)
          fprintf(outFile,"\t%f\n",TRT_R[i]);
        fprintf(outFile,"%%End TRT_R:\n");

        fprintf(outFile,"%%Start TRT_R_track:\n");
        for (i = 0; i < TRT_R_track.size(); i++)
          fprintf(outFile,"\t%f\n",TRT_R_track[i]);
        fprintf(outFile,"%%End TRT_R_track:\n");

        fprintf(outFile,"%%Start TrackX:\n");
        for (i = 0; i < TrackX.size(); i++)
          fprintf(outFile,"\t%f\n",TrackX[i]);
        fprintf(outFile,"%%End TrackX:\n");

        fprintf(outFile,"%%Start TrackY:\n");
        for (i = 0; i < TrackY.size(); i++)
          fprintf(outFile,"\t%f\n",TrackY[i]);
        fprintf(outFile,"%%End TrackY:\n");

        fprintf(outFile,"%%Start TrackZ:\n");
        for (i = 0; i < TrackZ.size(); i++)
          fprintf(outFile,"\t%f\n",TrackZ[i]);
        fprintf(outFile,"%%End TrackZ:\n");

        fclose(outFile);

        return 0;
}
```