Performance Engineering in High Energy Physics Software: The ATLAS Offline χ^2 Track Fitter

Stephen Nicholas Swatman

December 2019

Supervisors
dr. A.L. Varbanescu
dr. P.M. Kluit

University of Amsterdam
Vrije Universiteit Amsterdam

CERN-THESIS-2019-442
31/12/2019

Abstract

As the ATLAS high energy physics experiment gears up for the HL-LHC upgrade, the performance of the software used to process the experiment's massive amount of data grows more and more important. Models predict that, when HL-LHC becomes operational in 2026, the amount of CPU time required to process the data produced by the ATLAS experiment will increase by up to a factor of nine; this increase exceeds by far the predicted computing budget. In this thesis, we present the results of a research project investigating the feasibility and applicability of performance engineering techniques on HEP software, using the ATLAS offline track fitting software as a guiding example. We discuss the challenges involved in optimising such software from a software and performance engineering perspective. Of particular interest are ways in which traditional performance analysis techniques can be adapted to obtain detailed and meaningful performance data for such software. Indeed, we find that not all techniques can be applied naively to give maximally meaningful results. From the results of performance analysis using techniques modified to fit this particular use case, we design and implement a series of relatively simple optimising code transformations. These optimisations approximately double the performance of the fitting code, thus bringing it closer to the performance required for HL-LHC.

Acknowledgements

Many people were involved in some capacity with the coming about of this thesis. Most important of all to this project were my supervisors, Ana Varbanescu and Peter Kluit; I owe them both my undying gratitude for their support over the last months. Ana has helped me greatly by keeping my frantic mind (and pen!) on track, by giving me rigorous feedback on all my work, and by providing me with so many opportunities, including the opportunity to do this project in the first place. Peter was always there to help guide me into the fascinating, esoteric and often mind-boggling world of particle physics and quantum mechanics, and he always allowed me to bounce ideas off of him. Without either of them, I would not have been able to complete this project. My appreciation also goes out to Ana Oprescu for volunteering to be the second reader for this thesis, which has become somewhat more lengthy and wordy than I had hoped!

I would like to extend my gratitude to the Nikhef institute, and in particular the Nikhef ATLAS group; my appreciation goes out to everyone there. I spent the vast majority of the time working on this thesis at Nikhef, and I am thankful for the pleasant working environment they have provided. I would like to thank in particular Wouter Verkerke for accepting me into his group, as well as Roel Aaij for making this project possible at all. In addition, I greatly appreciate the help of Ivo van Vulpen, Tristan du Pree, and Paul de Jong. To Bryan, Peter, Rahul, Ashley, Jordy, Kees, Alessio, and Bram: thanks for making the ATLAS student room such a great place to work.

Of course, I must also thank the ATLAS collaboration outside of Nikhef: it's an international collaboration and this project is based in its entirety on their shoulders. My gratitude goes out to, amongst others and in no particular order, Edward Moyse, Attila Krasznahorkay, Shaun Roe, Stefano Rosati, William Leight, Andreas Salzburger, Anthony Morley, Markus Elsing Goetz Gaycken, James Catmore, and Scott Snyder.

Furthermore, I would like to thank the Parallel Computing Systems group at the University of Amsterdam for providing me with the opportunity to present my work there. Thanks to Stijn Heldens for inviting me to present my work at the Netherlands eScience Center.

On a personal note, I want to thank all of the people who I have the pleasure of

calling my friends for supporting me over the years. Mom and dad, you've done more for me than I could ever hope to thank you for on paper. Lia, I'm sorry that you weren't with us long enough to see me finish my thesis; thank you for always being so interested and supportive. I know you would have been very proud of me.

Contents

1. Introduction

Dozens of meters below the ground near the Swiss-French border lies a twenty-seven kilometre long tunnel that houses the *Large Hadron Collider*, the largest and most powerful particle accelerator ever constructed. This extraordinary machine is managed by the *European Laboratory for Particle Physics* (CERN), and its purpose is to push the boundaries of human knowledge about the fundamental particles that make up the universe and everything that exists around us. In the collider, small packets of protons are accelerated to approach the speed of light and compressed into the width of a human hair. These packets are made to collide with one another, and the resulting collisions are so energetic that new particles are created out of nothing but the momentum of the original particles. Through such high energy collisions it is possible for physicists to produce never before observed particles, such as the Higgs boson which was discovered in the LHC in 2012 [1]. This particle was theorised half a century earlier, and the LHC was the first collider powerful enough to produce it. While the behaviour of the Higgs boson is rather bizarre, esoteric and certainly outside the scope of this thesis, it suffices to say that its discovery was of immense value to the particle physics community. In fact, it was perhaps the most significant discovery within the field in several decades.

Notably, the LHC by itself does not produce much in the way of interesting physics research. In order to study what happens when particles collide in the LHC, we need to build detector devices which can probe the behaviour of collision events. Along the tubes of the Large Hadron Collider lie seven such experiments, the largest of these is *A Toroidal LHC ApparatuS* (ATLAS, [7]), and ATLAS is also the topic of this thesis. ATLAS is a so-called general-purpose experiment, and it is as such looking out for any new and interesting phenomena it can find. This contrasts with most other experiments, which have a much more specific goal. The Higgs boson was discovered independently by both ATLAS and the other general-purpose LHC experiment, the *Compact Muon Solenoid* (CMS).

The Large Hadron Collider represents the peak of electrical engineering and particle physics, but we should certainly not disregard the amount of computation involved in such an experiment. Beam packets collide every 25 ns, or at a rate of 40 MHz. Each

of these events is roughly 320 MB in size, giving a total raw data rate of $12.8\,\mathrm{PB\,s^{-1}}$. Performing a full analysis of each of these events would, in terms of computation, be a nigh impossible task today, let alone when the collider first came into operation. To combat this deluge of data, we see the application of ideas from many fields in computer science. For example, ATLAS heavily relies on FPGAs to discard uninteresting events and reduce the event rate from the aforementioned 40 MHz to 75 kHz. Layers of sophisticated software-based filters running on distributed systems reduce the data rate further to 1000 Hz. This enormously reduced number of events is stored, in a compressed format, on disk for later analysis, but storing even this tiny fraction of all events still presents a massive challenge in distributed data storage. After the data is stored, it is processed using complex algorithms and machine learning techniques. At each and every one of these steps, it is critically important that the software is not only as accurate as possible, but also as performant as possible.

The current system is capable of handling the data output of the ATLAS experiment as it stands now, but this is likely to change in the near future. To keep the LHC relevant until its eventual planned decommissioning in 2038, a large upgrade is underway to increase the luminosity of the collider. That is to say, there will be an increase in the number of colliding particles and the sensors in the experiments will become more sensitive. All in all, this will lead to an increase in the amount of data produced by the experiment. This increase in data volume will need to be managed, preferably without spending massive amounts of money on more computing hardware. To this end, the multi-purpose *Athena* software package, which is used for many different ATLAS computing tasks, will need to be optimised. We have embarked on a performance engineering project to increase the computational performance a specific part of this software: the track fitting algorithms, particularly in the context of offline muon reconstruction.

In this thesis, we attempt to answer the research question whether we can apply methods of analysis and optimisation from the performance engineering body of knowledge to increase the computational performance of high energy physics software at ATLAS without sacrificing the quality of the output. By studying the χ^2 minimisation track fitting algorithm in particular, we hope to provide a useful example for how such techniques can be applied to the particle physics domain. Our approach in answering the main research question is two-fold. Firstly, we aim to gain a better understanding of how the existing software performs, by means of benchmarking and profiling. The gathering of such data, which to the best of our knowledge has not been done before, will provide insight into performance hot spots and possible ways to improve

the performance of the software. Secondly, we seek to apply targeted optimisations to improve the performance of the software. We aim to thoroughly quantify the effectiveness of these improvements through empirical performance analysis.

The remainder of this thesis is structured as follows. Chapter 2 will provide the necessary background information on the Large Hadron Collider, the ATLAS experiment, particle physics, applicable algorithms, and previous work by other authors. Chapter 3 describes the Athena software package in detail, in order to give a clear understanding of its architecture and software quality. Chapter 4 describes our experimental setup, as well as the ways in which we build, run, benchmark, and validate the code. Chapter 5 presents an analysis of the performance of the existing Athena software, including benchmarks, models, and profiling. Chapter 6 describes the changes that we have made to the software, and it explores the performance benefits of those changes. Finally, Chapter 7 presents our conclusions, as well as our recommendations for future work.

2. Background

By its very nature, our project exists on the border of computer science and physics. In addition, the work is embedded in the ATLAS collaboration, which encompasses an incredible amount of previous work in the fields of mechanical engineering, civil engineering, physics, and computing. To impart an understanding of the context of this thesis, therefore, it is necessary for us to provide some background knowledge on many different fields. In this chapter, we will attempt to describe in very rough detail the ATLAS experiment, the LHC, relevant particle physics, track reconstruction algorithms, performance engineering, as well as previous work on the topic of this thesis.

2.1. The ATLAS Experiment

The ATLAS experiment is a large particle physics experiment positioned along the *Large Hadron Collider* or LHC. ATLAS is a general purpose experiment, and its goal therefore is to look out for any previously unknown physical phenomena; it is not bound to a particular niche like some of the other LHC experiments. It is cylindrical in shape, 44 m long, and 22 m in diameter. Along the central axis of this cylinder runs the beam pipe, a straight section of the 27 km long collider circuit. At the very centre of the experiment lies one of the interaction points of the LHC, where large magnets cause particle beams to cross. When such crossings occur particles may collide, creating additional particles out of the collision energy. These resulting particles travel outwards from the centre of the experiments, through thousands of detectors, which register the positions of passing particles [2].

Figure 2.1 shows a schematic breakdown of the ATLAS experiment. In the centre of the experiment, closest to the collision point, lies the inner detector. This is the area of the experiment where detectors are most numerous and precise. This is also the part of the experiment where we encounter the largest number of particles. Outside the inner detector lie the calorimeters, which are used to determine the kinetic energy of a particle. Even further away from the centre we find the toroidal and solenoidal magnets, which project a powerful magnetic field that permeates the entire experiment.

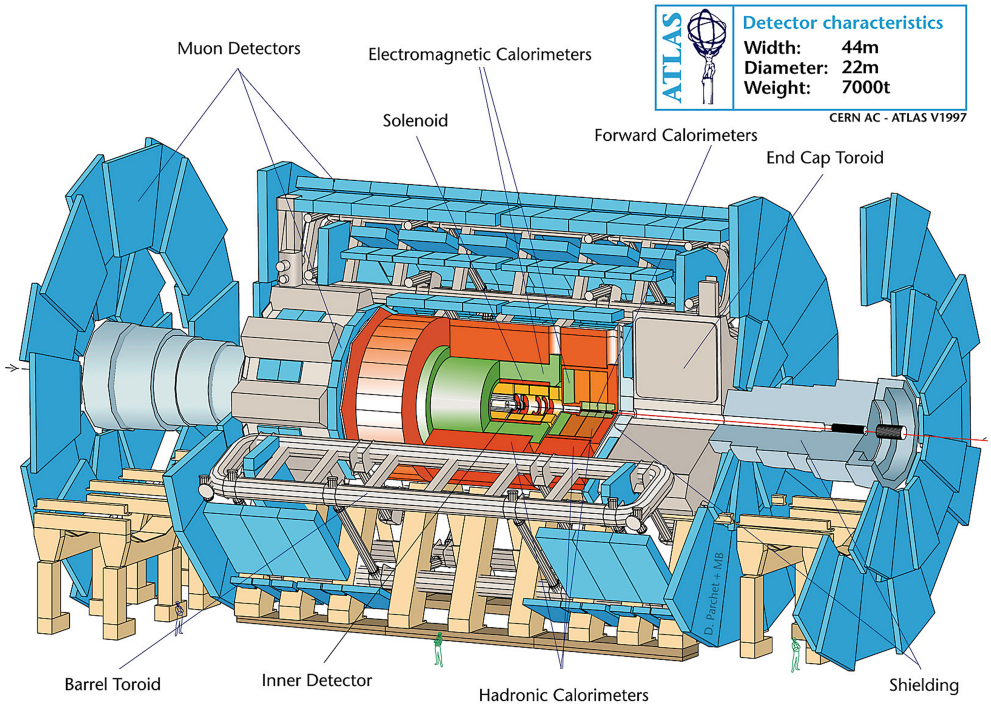


Figure 2.1.: A schematic cross section of the ATLAS experiment. Source: CERN Infographics Library.

Finally, we see the muon detectors. Muons (which we will discuss in more detail in Section 2.2) are some of the only particles that are able to travel this far through the experiment; most other particles come to a stop at or before the calorimeters. It is these muons that are of particular interest for this thesis, and we will be focusing our efforts primarily on fitting for the muon system.

2.1.1. HL-LHC

When operating a particle collision experiment, there are diminishing returns in the value of running the experiment for prolonged periods of time. In order for the Large Hadron Collider to produce further valuable physics data, there is an upgrade underway which is referred to as *High-Luminosity LHC* or HL-LHC. In high energy physics, luminosity represents the number of collisions over a period of time[1]. The goal of HL-LHC is to increase the luminosity of the collider, resulting in a factor five increase in the number of collisions. It is expected that the HL-LHC upgrade will be ready for operation at the start of run 4 in 2026 [3].

[1]The actual definition of luminosity is more complicated, involving also the collision cross section.

HL-LHC is critical to ensure the continued relevance of the LHC, and it is key to maximally exploiting the capabilities of the experiment. However, the upgrade also presents several significant challenges. For the purposes of this thesis, we will disregard the monumental challenges in terms of engineering and focus solely on the computational challenges of HL-LHC. The primary computing challenge inherent in HL-LHC is one of data volume, and the processing time of events scales superlinearly in the number of collisions. Indeed, it is predicted that the upgrade will cause a factor nine increase in computing power required to process all of the data. This means that the required computing resources will far exceed what is expected to be available when HL-LHC is finished [12, 6].

2.2. Particle Physics

To attempt to summarise an entire field of study in a single section of a thesis would be a fool's errand, but most foolish of all would be to attempt to summarise in such a way the field of particle physics. Still, some basic understanding of the field may be necessary to understand the rest of this work, and the goal of this chapter is to provide that knowledge. Specifically, we will briefly discuss the particles that make up our universe, their properties, and their interactions with matter.

2.2.1. Brief Historical Summary

The urge to understand how the matter that surrounds us at all times is constructed goes back to ancient times. As but one example of many, the famous philosopher Aristotle posited that all matter was a sort of highly receptive substance that could take various forms (including those of air, fire, water and earth) through the application of the fundamental forces of heat and cold, fluidity and solidity [32]. While this view on matter obviously does not hold up to the rigorous scientific processes of today, it is one of the first of many models that have been proposed in an attempt to explain the nature of matter.

Such models are of tremendous importance to particle physics, and they permeate the history of the field. In a way, the study of particle physics is cyclical; it starts with theorists developing a model which explains the experimentally observed phenomena known at that time. That model then leads to new experiments being developed to test the predictive power of that model. These new experiments inevitably reveal new phenomena that are not adequately explained by the existing models, and the cycle

starts over again [26].

Following this cycle backwards from the current understanding of particle physics takes us to a point in time even before the aforementioned theory by Aristotle. Ancient Greek and Indian philosophers developed theories that postulated the existence of indivisible, elementary particles that constituted all matter. While the finer details of these theories do not match what we know today, these theories paved the way for the theory of *atomism* (from ancient Greek for *indivisible*). The theory of atomism would lie dormant for over two thousand years, waiting for the development of experimental technology advanced enough to test it.

The first modern model of the atom would come from English chemist John Dalton at the start of the nineteenth century. Dalton's theory maintained that atoms were indivisible, hard spheres of matter, and he applied his theory mostly to the field of chemistry by measuring the relative weight of different kinds of atoms. Still, this theory renewed the interest in atomism and increased its credibility as a theory for explaining matter.

A major turning point came a hundred years later, at the start of the twentieth century, at the hand of English scientist Joseph Thomson. Thomson experimentally determined that the supposedly indivisible atom had to be made up of smaller constituent parts. Notably, Thomson was not the first to predict the existence of subatomic particles, but he was the first to experimentally discover them. The particle discovered by Thomson was the electron, a negatively charged particle, the movement of which we use every day in the form of electricity. The discovery of the electron came at a time where theories were developed that electrical charge must also have a fundamental unit, and the electron fit well into these theories.

It is at this point in history that the cycle of theory and experiment starts to accelerate. Thomson's model of the atom (the *plum pudding model*) consisted of negatively charged electrons suspended in a positively charged substance like plums in a pudding. This model was quickly experimentally discredited and replaced by the Rutherford model in which electrons orbit a small, dense, and presumably indivisible nucleus. The Rutherford model was then improved upon by Bohr to create the Rutherford-Bohr orbital model.

From here, our understanding of matter would go on to change in many more ways. Firstly, it would be discovered that subatomic particles do not behave according to the classical mechanics that govern objects at ordinary scale. Instead, these particles behave in strange, esoteric, and unintuitive ways that seem so divorced from the objects we see around us that a new class of mechanics had to be developed to describe

them: quantum mechanics. Secondly, the theory of the indivisible nucleus would not stand for long. In 1932, it was discovered that the atomic nucleus is made up of protons and neutrons, collectively called nucleons. In 1964, a theory was developed that nucleons in turn are made up of even smaller particles, up quarks and down quarks, and experimental evidence for this theory came about only a few years later.

2.2.2. Particles and Forces

Today, quarks and electrons are still considered elementary particles, but the division of ordinary matter into electrons, up, and down quarks paints a picture that is far from complete. Indeed, these three particles are only a few of many elementary particles in the *standard model*, a model of physics that describes very well (but not perfectly) the nature of matter. The derivation of this model is far outside of the scope of this thesis, but we must superficially explore it to make clear the scientific context of this work.

The up and down particles fall into a category of particles referred to as quarks, while the electron is a type of particle referred to as a lepton. Another lepton is the so-called electron neutrino; the electron and the electron neutrino are often found together in interactions, but are otherwise dissimilar. The electron neutrino has very little mass, no electrical charge and it interacts very little with ordinary mass, making it hard to detect. In fact, a neutrino could pass through a piece of lead one light year thick and it would most likely not interact with a single lead atom! It is only through the incredible number of neutrinos that bombard the earth at any given time that we have any hope of detecting any at all.

These two quarks (up and down) and two leptons (the electron and the electron neutrino) make up the first generation of the class of particles known as fermions. The matter that we see around us is made up of these first generation particles, but there are two more generations. These generations become increasingly unstable and their particles generally exists only briefly before decaying into first generation particles. The second generation of fermions consists of the charm and strange quarks as well as the muon and the muon neutrino leptons. The third generation consists of the top and bottom quarks, and the tauon and tauon neutrino. While we don't come across second and third generation particles in ordinary matter, they do occur naturally in the world around us for brief periods of time. For example, high energy particles from space constantly bombard the Earth's atmosphere, and collisions between these particles and the particles making up the atmosphere can lead to the creation of second and third generation particles.

The twelve fermions described above are complemented by the so-called bosons, or

force-carrying particles. Bosons mediate the forces that allow fermions to interact with each other and they critically prevent the universe from (quite literally) falling apart. Five bosons are known to exist, and the Higgs boson is the most recently discovered one, being experimentally confirmed in 2012. Perhaps the most widely known boson is the photon, commonly known as the particle that is responsible for light. The photon is the particle that mediates the electromagnetic force, and whenever objects interact with each other by electromagnetism (such as when they are attracted or repelled by a magnet), that is the result of an interaction mediated by photons. When two particles with the same charge approach each other, they begin to repel through the repeated process of one particle emitting some of its own momentum in the form of a photon and the other particle absorbing that photon to gain momentum. The emission and absorption of photons changes the direction in which the repelling particles travel, resulting in the behaviour we would expect to see.

The other bosons include the gluon, which mediates the strong force; the strong force is the force that holds quarks together to form nucleons. While the strong force is the strongest of all forces, its operates only on extremely short ranges, and as such we do not experience it directly in day to day life. The W boson and Z boson mediate the weak force; like the strong force, it has limited range and is therefore not directly noticeable at macroscopic scale. The weak force still plays an important part in nature, though, as it allows quarks to change flavours (for example, it allows an up quark to turn into a down quark), and it thereby facilitates certain types of radioactive decay. As an aside, it is perhaps for the best that the strong and weak forces are of limited range – even the weak force is twenty-five orders of magnitude stronger than gravity, and we should consider ourselves lucky that it is only gravity that acts on us when we fall off our bikes.

One might wonder which particle mediates gravity. Indeed, we know that there are bosonic particles that mediate every fundamental force other than gravity. It is still an open question whether gravity is mediated by a bosonic particle or if it is caused by some other effect. If a particle is responsible for gravity, it is unlikely to be discovered any time soon – particles are detected through their interactions with matter, and as mentioned earlier gravity interacts only weakly with matter. Therefore, modern detectors have an infinitesimally small chance of actually detecting a hypothetical gravity particle, or *graviton*. The Higgs boson is known to exists but does not mediate a fundamental force. The Higgs boson is the reason some of the other bosons have any mass at all, but the mechanism behind this is far outside the scope of this work.

Finally, it is important to note that not all particles interact with all four fundamental

forces, and the 'willingness' of a particle to interact with different forces determines how difficult it is to detect. Particles with electric charge interact with the electromagnetic force, and it is this force that makes particles detectable; particles with so-called colour charge interact with the strong force; finally, most particles interact with the weak force due to their iso-spin. Charged leptons, such as muons, interact with gravity, the electromagnetic force, and the weak force, and are relatively easily detected. Uncharged leptons, on the other hand, interact only with the weak force and gravity, and are therefore harder to detect, as discussed earlier. Quarks interact with all four fundamental forces, and notably cannot exist on their own in a so-called free state[2].

2.2.3. Particle Interactions with Matter

In a particle collider experiment like ATLAS, the objective is to understand what is happening when a high energy particle collision occurs. This, of course, requires ways to measure how particles behave, as well as methodologies for dealing with all kinds of unwanted physical phenomena. As a poignant example of a particle detection technique that remains fairly close to everyday life, we will consider first the so-called nuclear emulsion. A nuclear emulsion plate is much like a traditional photographic plate, but instead of recording an image, it records the tracks of particles travelling through it. A particle travelling through a nuclear emulsion infers an electric charge to some of the particles it passes by through a process called *ionisation*, and this permanently alters the molecules in the emulsion. The emulsion is then developed just like a photograph, and particle tracks become visible to the naked eye!

Of course, rapidly changing out nuclear emulsion plates is not a feasible way to operate a modern particle physics experiment where millions of events happen every second, but the concept is not too dissimilar from the technology used in ATLAS: charged particles are detected due to the electric charge they deposit in particles they pass by. This is the interaction with matter that makes the detection of particles possible at all. Instead of using photographic emulsion, modern detectors use silicon chips (analogous to how we now use silicon-based digital cameras instead of film-based analogue ones) or compartments of gas to detect particles. In ATLAS, *Pixel* and *SCT* detectors are silicon based, while *TRT* and *MicroMegas* detectors are gas-based.

Detecting particles alone is far from the complete story, and the matter interactions that make it possible to detect particles have some complications that make it non-

[2] If one were to try and separate quarks that are bound together by the strong force, one would need to insert so much energy into the system that the quarks can create additional quarks from the vacuum to form new bound states.

trivial to understand particle tracks. Consider, for example, non-charged particles, such as neutrons n or uncharged kaons K^0. These particles do not ionise other particles, and as such they are invisible to nuclear emulsions and modern detectors alike! To reconstruct the path of uncharged particles, we must either infer their parameters from the interaction that created them, or we can hope that they decay into charged particles elsewhere in the detector. Furthermore, particles do not travel in a straight path away from the center of the detector. Rather, there are several physical effects which can alter the track of a particle, sometimes in unpredictable ways. For one, magnetic fields influence the tracks of any charged particles travelling through them. ATLAS has a particularly powerful magnetic field, and while the magnetic field in the ATLAS experiment is predictable (the value of the field in any given point is known fairly precisely), it is not uniform. This complicates tasks like propagating a particle through the magnetic field significantly, as this must be done numerically rather than analytically.

An example of an unpredictable effect results from the previously discussed ionisation: If a charged particle is to deposit a charge to some other particle, it loses some of its energy, and therefore some of its momentum, changing the track of the particle. To make matters worse, particles do not interact solely with detectors (*active* material), but they can also interact with non-detecting (*passive*) materials such as support structures; if a particle travels through passive material it will change course without us being able to detect it. If the material is of appreciable thickness, it is likely that many so-called scatterings will occur, and the particle leaves the material at an angle with Gaussian probability. This process is referred to as multiple scattering. Finally, highly energetic charged particles such as high energy electrons can also alter their momentum through the spontaneous emission of a photon when travelling through a medium. This is referred to as *bremsstrahlung*, German for *braking radiation*.

Photons can interact with matter in even more ways. For example, a photon and an electron may interact with each other, altering the momentum and direction of both particles, in processes dubbed Thomson scattering and Compton scattering. Via the photoelectric effect, a photon may knock an electron loose from an atom, creating electrical charge. Finally, a highly energetic photon may, in the vicinity of matter, spontaneously decay into an electron and a positron in an effect called pair production. It is these effects that make tracking of particles in a detector a non-trivial task. The technical details these effects is not within the scope of this work, but we mention them to show how many factors add uncertainty to particle collision events.

2.3. Track Reconstruction

When particles travel through the ATLAS experiment, it is impossible to record their position continuously. Rather, the data that is available only describes which detector elements were triggered. To reconstruct the actual particle tracks is the core goal of track reconstruction. The process of track reconstruction, from raw detector data to track parameters, consists of roughly three steps: space-point finding, track finding, and track fitting. The aim of this section is to describe these processes.

2.3.1. Space-point Finding

The simplest possible model for a particle detector would be a device outputting a binary signal; high if a particle was detected and low otherwise. During reconstruction, we could retrieve the position of our detector element from a spatial description of the entire experiment to conclude that a particle must have passed through that particular location. This is the essence of space-point reconstruction: transforming signals from thousands of individual detectors to points in space through which we assume a particle has travelled.

In practice, detector elements are more complicated, and they come in several very different designs. The so-called Pixel detectors, for example, behave somewhat similarly to a digital camera; a large grid of pixels[3] record the position of particles through them relative to the position of the pixel detector as a whole. Compared to our earlier hypothetical particle detector this adds an additional step to the space-point finding process.

A different type of detector element is the so-called *Transition Radiation Tube* or TRT. TRT detectors are cheaper (and less accurate) than Pixel detectors, and are used further away from the main collision point where the cost of using more precise detectors would be prohibitive. A TRT detector consists of a long gas-filled straw with a wire running along its central axis. When a charged particle passes through the straw, the gas becomes excited and this charge is detected as it travels through the wire. TRT detector readouts indicate how far along the tube the particle was detected, as well as the distance between the wire and the particle's point of closest approach to it. It follows that one cannot retrieve a single space point from a TRT detector, and that the readout indicates instead a circle of possible positions of the particle. The resolution of

[3] The size of each pixel in a Pixel detector is much larger than a pixel in a digital camera, at roughly $50 \times 400\,\mu\text{m}$. These pixels are also designed to be read out forty million times per second, and are hardened to resist extreme amounts of radiation.

uncertain results from TRT detectors is outside the scope of this thesis (it is not track *fitting*, after all), but it illustrates some of the challenges that make space-point finding non-trivial.

Space-point finding is subject to observational errors, both systematic and random. Systematic error can manifest in inaccurate data about the positions of detector elements, and random error can be due to a lack of precision in the detector elements. Known errors are passed onto the track finding and fitting algorithms, as they are critically important for achieving high quality results. It is also possible for detectors to fire spuriously, or to not fire at all when a particle passes through them; there is no hope of resolving such errors in the space-point finding step, and this responsibility is delegated mostly to track finding.

2.3.2. Track Finding

When the space-point finding step is complete, we are left with a large number of coordinates through which we assume particles have passed. If an event would involve only a single particle, we could immediately try to fit a curve to these points, but this is not the case in reality; an event can contain hundreds of individual particles, and they do not conveniently leave a business card at the detector elements to allow us to uniquely identify them. To have any hope of fitting accurate track parameters to points, we must first separate those points into clusters that we assume belong to the same particle. That is the essence of track finding.

Track finding is an expensive combinatorial operation, the complexity of which scales superlinearly with the number of space points in its input. On the other hand, it also lends itself well to parallel computation. First, space-points are grouped into so-called *seeds* of three points each that are likely to belong to the same track[4], and very rough estimates for the track parameters are generated. Then, seeds with similar parameters (and therefore likely matches) are combined into track candidates. This two-phase approach to track finding helps mitigate the *combinatorial explosion* inherent to the technique. For an event with one thousand tracks, the track finding algorithm generates roughly two thousand track candidates from some twenty thousand seeds [14]. Each track candidate has a rough estimate for the parameters of the track, which will be used and refined by the track fitting algorithm.

Track finding is prone to noise in the form of cosmic radiations. Even though the ATLAS detector is many meters under the ground, high energy particles from space

[4] In theory, any three points can belong to the same track; one could always find a curve through them. However, only likely candidate seeds are selected

can and do still penetrate into the experiment. Such particles can cause detectors to fire, and they introduce space-points that do not belong to collision particles at all. Track candidates for cosmic rays can be reconstructed as normal and discarded in the track fitting step. Spurious detector firings can be eliminated during the track finding step, since it is unlikely that useful seeds will be generated with them.

2.3.3. Track Fitting

Finally, we arrive at the main topic of this thesis: track fitting. Once the track finding algorithm has generated track candidates with rough estimates of their track parameters, the goal of track fitting is to refine those parameters such that they most closely fit the observed data. The closeness of fit is quantified using the χ^2 measure, defined over N degrees of freedom (in the case of track fitting, detector hits) such that p_i is the measured position in detector i, $\hat{p}_i$ is the expected position for detector i according to track parameters, and σ_i is the measurement error of detector i:

$$\chi^2 = \sum_{i=1}^{N} \frac{(p_i - \hat{p}_i)^2}{\sigma_i^2}$$

The quality of the fit is iteratively improved by changing the track parameters. Between iterations, the values of p_i are constant, while the values of $\hat{p}_i$ vary. To understand how the values of $\hat{p}_i$ change when the track parameters are modified, it is important to know how the expected positions are calculated.

Recalling that we know the positions at which particles were detected (p_i), we know with some error margin where a particle passed through a detector. Since we also know in which order the particle passed through the detector elements, it follows that there must exist smaller track sections between each pair of detector elements hit in sequence. The way expected track positions are calculated is by starting on one detector element, at the position where that detector detected a particle, and plotting the track section as we would expect it to behave given the estimated track parameters from there to the next detector. If we trace the predicted track section, we get a predicted position for that track section's intersection with the next detector element. It is this position that serves as the estimated position in the calculation of the χ^2 measure.

Because of the many physical effects dictating the behaviour of particles (described in Section 2.2), we must use numerical methods to trace track sections between two surfaces. The numerical tracing of a track according to a set of track parameters is referred to as *transportation*. At the core of this are two component mechanisms: *propagation* and *navigation*. Propagation refers to the use of Runge-Kutta methods

to account for the highly heterogeneous magnetic field that permeates the ATLAS experiment. Navigation refers to the process of accounting for non-active material in the experiment; during propagation, we may encounter material such as support columns, cabling, or even detectors that erroneously failed to fire (holes) which can alter path travelled by the particle. It is the responsibility of a navigation tool to account for such materials by breaking the track segment between two active detector elements into several even smaller track segments.

The process of refining the track parameters continues until a certain threshold of quality is reached, or until the number of iterations exceeds a certain maximum. In brief, the track fitting algorithm is executed as follows:

1. Start with a rough prediction for the track's parameters.
2. For each detector hit, navigate and propagate onto the next detector using the current prediction for the track's parameters.
3. Calculate the χ^2 of the resulting points. If the value is acceptable, return a success. If the value is not acceptable and the maximum number of iterations is reached, return an error.
4. Refine the track's parameters, and continue at step 2.

2.4. Performance Engineering

Performance engineering is a field within software engineering that concerns itself with the non-functional efficiency of software. In other words, performance engineering aims to modify software such that it gains as much possible value from the hardware on which it is executed. Usually, this means reducing the latency of software, or increasing it's throughput, but it can also relate to monetary cost, power draw, or memory use.

What gives rise to the need for performance engineering is the ever increasing complexity of computing, in terms of both software and hardware. When electronic computers first became available, they were already complex machines and technological marvels in their own right, of course, but over the many decades since, computers have grown to an almost unimaginable level of complexity, and this has made it much more difficult to write performant software. Where processors had a limited set of available instructions before, they now have thousands. Where memory used to be uniform before, it is now divided into complex hierarchies. And where machine code was once written by hand, it is now generated automatically by optimising compilers

which are of unprecedented complexity themselves. These are only a few ways in which the complexity of performance-aware computing has increased over time.

Someone aiming to increase the performance of a computer system has a large array of tools available; performance may be gained through simple means such as enabling certain compiler flags, or through extensive rewriting and refactoring of code. In fact, the number of ways that code can be optimised for performance is so large that it can be hard to see the forest for its trees. Besides providing the techniques to implement more performant code, the performance engineering body of knowledge therefore also provides methodologies to determine which optimisations are suitable to which parts of a code base, and to determine which segments of a large code volume are worth optimising in the first place. Such analysis methods include benchmarking, performance modeling, and profiling.

The performance engineering process is archetypically cyclical; it consists of the repeated analysis and subsequent modification of a computing system either until the desired results are achieved, until there is no further possibility of improvement, or until time and money run out. During each iteration of this cycle, the result of the analysis is used to determine where and which code transformations should be made. That is to say, we aim to make *targeted* optimisations that most effectively make use of the person-power available for the performance engineering process.

The aim of this thesis is to investigate how well the aforementioned cycle, as well as the associated analysis and optimisation methodologies, apply to the software used in the field of high energy physics. Indeed, the distinct architecture and code characteristics of such software may hinder the use of certain techniques, and it may require us to take a somewhat different approach to the performance engineering process. By documenting the exploratory process of combining performance engineering and high energy physics, we hope to narrow the gap between these fields, such that we can more easily apply performance engineering concepts to particle physics software in the future.

2.5. Related Work

Computational performance has been an important issue that has received a lot of attention since well before the LHC first came online. The ATLAS collaboration has a vast amount of hardware and software, and certainly not all previous work is related to the offline Athena software. Much of it is focused, for example, on the triggers. Even if we constrain our search to include only works that investigate the performance of the

Athena software, we must consider that this is in itself a very large software package that can perform many different tasks other than reconstruction. Indeed, we will touch on the Athena package as a whole in more detail in Chapter 3.

Much previous performance engineering work was done on the simulation component of Athena, which is responsible for generating simulated events for validation and testing. Work on the simulation component includes two separate optimisation task forces in 2008 [28] and in 2010 [4]. A master thesis was also written on the topic by Errenst [11]. While the work by Errenst is focused on simulation and not tracking, it has still been a valuable resource for understanding the architecture of Athena. A thesis by Langenberg [23] examines the computational performance of the Athena offline software as a whole; due to the broad scope of that work, its relevance to our much more narrow specific research is limited.

On the topic of reconstruction specifically, we find that most previous work has been on track finding as opposed to track fitting. Indeed, track finding lends itself far better to parallelisation, heterogeneous computing and techniques such as machine learning, while track fitting has inherently sequential properties, and is not easily captured by artificial intelligence methods.

We note a previous work by Du Pree [10] that examines, and attempts to improve, the performance of the `GlobalChi2Fitter` specifically[5]. The work examines specifically the performance of matrix inversion operations in the track fitting algorithm, and attempts to replace them by the more efficient versions provided by the Eigen linear algebra library. A speedup by a factor of two is reported, but to the best of our knowledge this concerns only the performance of the matrix inversion and not the track fitter as a whole; as we will see in Section 5, the speedup on the scale of the track fitting algorithm as as whole was likely significantly less. Notably, the code developed in the context of the work by Du Pree is no longer present in the code base, and the matrix inversion has been replaced by an even more efficient Cholesky decomposition.

Papers have also been published studying the computational performance of software used in other LHC experiments, such as those by Pérez and Couturier [27] and Ticse et al. [34], which are focused on the LHCb experiment. While software architectures and algorithms do differ significantly zbetween ATLAS and other experiments, these works examine the applicability of more general performance optimisation techniques in the context of particle physics, which matches the goals of this work. In particular, these works conclude that the application of SIMD architectures can lead to significant

[5] This work is of special importance to us, since it was the catalyst that sparked additional research into the computational performance of this specific part of the code.

speedups, even in vertical configurations.

3. The *Athena* Software Package

Athena is the primary offline data processing software package used in the ATLAS collaboration. It is a large, multi-purpose software package with a strongly defined framework for data processing. Understanding the architecture of Athena is of critical importance to the performance engineering process. The purpose of this section, therefore, is to examine the Athena software from a software architecture and engineering point of view.

3.1. Architecture Overview

At its core, Athena is designed to be a complete software package for almost all offline computing that is required in the ATLAS experiment. This means that the tracking software that we are working on resides in the same code base as code that is used for vastly different applications such as event simulation. In order to organise the large amount of code necessary to support all the kinds of computation that Athena must be able to perform, the system is built on the *Gaudi* framework. Gaudi is a framework for developing high energy physics software, and was developed at CERN. It is used as a foundation for the software used in several other LHC experiments, such as LHCb.

The Gaudi framework encourages the development of structured and organised physics applications by forcing developers to write independent and interchangeable algorithms, tools, and services. Instead of combining a large number of algorithms on the source code level, computation is controlled by the end user specifying a list of algorithms that is to be executed on each input data point. By factoring all use cases that Athena must support into separate algorithms and services, the code volume and complexity of individual parts of the code can be kep more easily under control.

Within Gaudi terminology, *an algorithm* refers to a transformation of data, designed to be executed in sequence with other algorithms, to transform the input data to the desired output format. Algorithms read and write data from a data storage mechanism managed by Gaudi, but the externally visible behaviour is as though algorithms are passing data between each other directly.

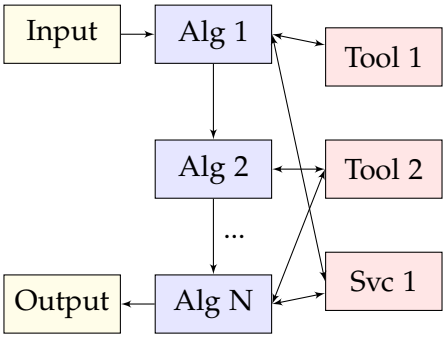


Figure 3.1.: Architecture of an Athena job. Algorithms are composed in sequence, each using zero or more tools and services. Input is passed between the algorithms sequentially, until there are no more algorithms left to execute on them.

Tools also represent transformations of data, but do not necessarily operate in sequence with algorithms, and they do not necessarily interact directly with data in Gaudi's data storage mechanism. Tools can be used by algorithms or other tools directly, and they are used in the same way as any other callable. Tools are, ideally, stateless. Multiple instances of the same tool can exist simultaneously and they may be configured in the same way. Tool instances can be declared *public*, such that all algorithms can use them, or they can be made *private*, in which case only the algorithm owning them has access to their functionality.

Services are stateful singleton objects that are used by many different algorithms or tools. For example, all logging in Athena is done through the `MessageSvc` service. Most services are designed to interact with the outside world in some way, although this line can become somewhat blurred. The magnetic field service (`MagFieldSvc`), for example, allows algorithms and tools to find the direction and strength of the magnetic field inside the detector at any point in space. Even though such a component has tool-like properties, it also interacts with a large external file describing the magnetic field, and it is therefore defined as a service.

The Athena software consists of roughly 2100 modules, each containing algorithms, tools, and services. Each module is compiled into shared objects which are loaded on demand at run-time. The separation of functionality into so many individual components has important benefits: it enforces a separation of concerns between modules, and it speeds up the Athena development cycle (see Section 3.2). Upon execution, the Athena software will load the required shared objects from a local build first if such a build is available, and it can source any missing objects from a remote

repository containing a complete build of the software. In the ATLAS experiment, a continuous integration system builds the full Athena software every night and stores the compiled modules on a shared file system (the *CERN Virtual Machine File System*, or CVMFS) that is accessible to Athena users and developers.

To ensure interoperability between modules, interfaces are widespread in Gaudi and Athena. Through the use of interfaces, modules can be written that interact with other modules while being agnostic of the exact implementation of the modules they rely on. Some tools and services in Athena are members of a tool family, meaning that there are alternative tools and services available that perform the same operation albeit with different internal implementations. An algorithm that interacts with a global fitting tool, for example, would be written in such a way that it interacts with the `IGlobalTrackFitter` interface. At run time, the user's job configuration file would determine which of the available implementations, and which instance of that implementation, is used.

3.2. Build System

The Athena software package is built using the *CMake* meta build system. Customarily, an implementation of the *make* build system is used to then build the software, although support is also provided for the *Ninja* build system. The recommended compilers for compiling Athena come from the GNU compiler collection, which provides compilers for C++ (`g++`) and Fortran (`gfortran`), the most commonly used compiled programming languages in Athena (see Table 3.4).

Owing to the large code volume of Athena, it would be disastrous for the development process if Athena had to be fully recompiled for each small change; it can take many hours to build a complete Athena release, even on modern hardware using parallel compilation. The fact that Athena is separated into many different modules as mentioned in Section 3.1 helps remedy this issue, as developers can choose to compile only those modules that are relevant to the changes they aim to make to the software. All other modules can be retrieved from an existing shared build at run time. To ensure that any arbitrary subset of modules can be successfully compiled, each individual module has its own complete build system.

A partial build of Athena is usually performed through one of two methods. The first method is a sparse checkout, in which the version control system is configured to store on the file system only a subset of the source code. The Athena source code is managed using the *git* version control system, and this method is referred to as a

sparse checkout[1]. The build system will then attempt to build every module for which the source code is available. In cases where header files are required from modules that were not checked out, the build system will attempt to retrieve those files from a build on the CVMFS file system, similar to how missing shared objects are supplied.

The second method for performing a partial build of Athena is referred to as a *sparse compilation*. When sparsely compiling an Athena release, whether or not a module is compiled is determined by a configuration file rather than the on-disk presence of the required source code. The configuration file consists of rules indicating whether a module should or should not be compiled, and the build system omits any module that is not configured to be compiled. In general, a sparse compilation will require more disk space in order to store the source code for the project (which is approximately 480 MB in size)[2], but carries the significant advantage of being more reproducible; distributing or sharing a sparse compilation configuration file is trivial, whereas sparse checkouts are configured on the level of the version control system and require more effort to distribute.

3.3. Multi-Core Processing

In order to benefit from multi-core systems, Athena employs a multi-processing model that is referred to as *Athena Multi-Processing* or *AthenaMP*. AthenaMP implements a batch processing model where multiple events are processed in parallel. In recent years, the growth in the number of cores in most machines has exposed an important flaw in the multi-processing model of Athena, however; under a multi-processing model, each process maintains its own copy of the state of the system. The state of the Athena software can consume a lot of memory since the software must contain, amongst other things, a complete and detailed model of the geometry of the detector. Some relief for this problem was found by sharing copy-on-write memory pages between processes, but the memory footprint of AthenaMP still proves to be too high to be practical.

To truly fix the memory sharing problem, Athena is now moving towards a multi-threaded approach referred to as *AthenaMT*. Like AthenaMP, AthenaMT will implement batch parallelism on the event level. Under a multi-threaded design, parallel threads can

[1] Sparse checkouts are a feature of git itself, documented at `https://git-scm.com/docs/git-read-tree#_sparse_checkout`.

[2] In practice, this is not such a significant disadvantage; for git to produce a sparse checkout of a repository, it must still store the full change history, and as such even files that are not checked out will occupy space on disk. Furthermore, it is technically possible to do a sparse compilation of a sparsely checked out repository, so the two techniques are not mutually exclusive.

share much of the required data, saving a significant amount of memory. Unfortunately, such a drastic paradism shift is complicated by the fact that most of the existing modules (of which there are over two thousand) are not written with thread safety in mind. Many modules are not re-entrant or maintain state in a way that is not thread safe.

As of the time of writing, it is not yet possible to run Athena in a multi-threaded configuration. In fact, merely enabling the multi-threaded processing model with a single thread causes the program to malfunction. A significant effort is currently under way to improve the thread safety of Athena. Throughout our work, we have resolved thread safety problems wherever we discovered them and we have taken great care not to introduce new race conditions and other thread safety issues to the code base.

One unfortunate consequence of the AthenaMT processing model is that it limits our design space when working to optimise the fitting code. Indeed, the AthenaMT batch processing will operate at event level and the tools that we are working on will always be called in a parallel context. Any attempts to implement thread-level parallelism in the tools would be redundant, as all available cores would be used to process additional events in parallel. While it is possible to posit specific circumstances in which thread-level parallelism in tools could lead to benefit (for example, in cases where the batch load between cores is unbalanced), we believe the additional development time, added complexity, and scheduling overhead makes it not worthwhile to pursue thread-level parallelism within tools.

3.4. Portability Requirements

The Athena software is run on a wide variety of machines, sometimes with vastly different CPU architectures and capabilities; the software is run on the computing infrastructure at CERN, the computing infrastructure at many of the institutes connected to ATLAS, the personal computers of scientists, and even by volunteers at home through the BOINC programme. A list of processor architectures used to run Athena is given in Table 3.1. To ensure problem-free operation of Athena on all machines, strict portability requirements are maintained. The source code should be written in a platform-agnostic manner and, as such, we cannot use platform-specific optimisations in our project.

One problem with strictly enforcing portability is that it can prevent the software from benefiting from many recent advancements in CPU architecture. A significant part of the performance increase in hardware over the past decade has been due to the introduction of new instruction set extensions. These extensions must remain unused

Table 3.1.: Number of Athena jobs by microarchitecture of the processor used. The data represented was gathered in 2018 and is taken from Snyder [31]. Information on the capabilities of the microarchitectures listed is taken from the gcc compiler manual [13].

Year	Architecture		SSE3	SSSE3	SSE4.1	SSE4.2	SSE4a	AVX	AVX2	AVX512F	FMA4	FMA3	Number of Jobs	
2003	AMD	K8	✓										871771	0.40%
2006	Intel	Merom	✓	✓									1541711	0.70%
2007	Intel	Penryn	✓	✓	✓								7497405	3.43%
2007	AMD	K10	✓				✓						2852463	1.30%
2008	Intel	Nehalem	✓	✓	✓	✓							7771807	3.55%
2010	Intel	Westmere	✓	✓	✓	✓							26683828	12.20%
2011	Intel	Sandy B.	✓	✓	✓	✓		✓					15816191	7.23%
2011	AMD	Bulldozer	✓	✓	✓	✓	✓	✓			✓		5644664	2.58%
2011	AMD	Bobcat	✓	✓			✓						81	0.00%
2012	Intel	Ivy Bridge	✓	✓	✓	✓		✓					22677742	10.37%
2012	AMD	Piledriver	✓	✓	✓	✓	✓	✓			✓	✓	5939365	2.72%
2013	Intel	Haswell	✓	✓	✓	✓		✓	✓			✓	55864736	25.54%
2013	Intel	Silvermont	✓	✓	✓	✓							757	0.00%
2013	AMD	Jaguar	✓	✓	✓	✓	✓	✓					199	0.00%
2014	Intel	Broadwell	✓	✓	✓	✓		✓	✓			✓	52040830	23.79%
2014	Intel	Airmont	✓	✓	✓	✓							96	0.00%
2014	AMD	Steamroller	✓	✓	✓	✓	✓	✓			✓	✓	1597	0.00%
2014	AMD	Puma	✓	✓	✓	✓	✓	✓					57	0.00%
2015	Intel	Skylake[1]	✓	✓	✓	✓		✓	✓			✓	11522901	5.27%
2015	AMD	Excavator	✓	✓	✓	✓	✓	✓	✓		✓	✓	923	0.00%
2016	Intel	Kaby Lake	✓	✓	✓	✓		✓	✓			✓	46880	0.02%
2016	Intel	Goldmont	✓	✓	✓	✓							174	0.00%
2016	Intel	KNL	✓	✓	✓	✓		✓	✓	✓		✓	447097	0.20%
2017	Intel	Coffee Lake	✓	✓	✓	✓		✓	✓			✓	7314	0.00%
2017	AMD	Zen	✓	✓	✓	✓	✓	✓	✓			✓	183627	0.08%
	Other												1340776	0.61%
	Total												218754992	100.00%

[1] Some Skylake processors support AVX512F, but we do not have sufficient information to distinguish between processor models that do and those that don't. For this reason, we assume a lack of AVX512F for all Skylake processors in this dataset.

Table 3.2.: Number of jobs executed on processors with selected instruction set extensions. Data derived from Table 3.1.

ISE	Number of Jobs	
SSE3	217414216	99.39%
SSSE3	213689982	97.68%
SSE4.1	212148190	96.98%
SSE4.2	204650785	93.55%
SSE4a	14622976	6.68%
AVX	170194123	77.80%
AVX2	120114308	54.91%
AVX512F	447097	0.20%
FMA4	11586549	5.30%
FMA3	126055270	57.62%

in Athena in order to maintain backward compatibility with old processors. At the time of writing, the Athena software is built to support generic x86-64 processors. This means that only very old instruction set extensions are used, such as MMX and SSE2. Luckily, a recent push has been made to step away from this way of building the software, and work is ongoing to include SSE3, SSSE3, SSE4.1 and SSE4.2 into the list of required instruction set extensions. As we can see in Table 3.2, only 6.45% of jobs are run on processors without support for SSE4.2. This decision represents a trade-off between dropping support for older processors (and thus reducing the total available processing power), and more effectively exploiting newer hardware. Interestingly, the GCC compiler supports function multiversioning, where a single function is compiled multiple times into the same binary with support for different micro-architectures, such that the right function can be loaded at run-time; this could allow Athena to support a wide range of hardware while also making good use of newer instruction set extensions. However, this is not currently enabled in the Athena build system, and it would require significant effort to enable.

Support for instruction set extensions is not only a build-level issue, but also impacts the development on source level. On source level, it is possible to write code in such a way that it is not compatible with all CPU architectures. For example, code written with inline assembly or some intrinsic functions can lead to undefined instruction errors. When implementing optimisations in ways which can impact the portability of the software, it is of critical importance to supply the compiler with alternative versions of the code that can be used in case support for a particular instruction set extension is not available. There are at least two ways to exploit newer instructions on source code

level which do not have portability problems. Firstly, external libraries can be used that either provide versions of commonly used functions that are written with modern ISEs in mind. The Athena library package provides two libraries that do this: *Eigen* [15] provides vectorised implementations of a wide variety of linear algebra functions and *Vc* [22] provides a lower-level way to interact with data vectors. Both of these libraries relieve the developer of the requirement to write multiple versions of the same code in order to be compatible with a wide range of hardware; the logic to determine which version of the code should be compiled is contained within the libraries themselves. It is also possible to leave the code unchanged and to rely on the compiler to vectorise the code automatically. Unfortunately, this process can be unreliable, and we therefore prefer to use the aforementioned libraries..

3.5. Code Quality

Performance engineering can be viewed as a form of code maintenance and, as such, it can be insightful to first approach the software to be optimised from a software maintainability point of view. Indeed, IEEE defines maintainability as "the ease with which a software system or component can be modified to ... improve performance ..." [17]. That is the purpose of this section; we will apply the methodologies and metrics related to maintainability as found in the software engineering body of knowledge to predict any difficulties we may face during the optimisation process.

International standard ISO/IEC 25010:2011 [18] posits an abstract definition of the notion of quality of software. According this standard, software quality has eight core characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. Of these, only performance efficiency, maintainability, and portability are in the scope of this thesis. The main matter of this thesis is focused on the performance efficiency, and the portability of the Athena software is discussed in Section 3.4. The aspect of maintainability is subdivided further into five sub-characteristics: modularity, reusability, analysability, modifiability, and testability.

While the standard roughly defines three layers (the abstract notion of software quality, the eight constituent characteristics, and sub-characteristics), it does not provide a concrete methodology for quantitatively describing these layers. Heitlager, Kuipers, and Visser [16] expand on the standard by adding two additional layers to the model. They posit that each sub-characteristic of the software is determined by one or more properties of the code (an example given is that the modifiability of the code is impacted

by its complexity), and that each property is measured through one or more metrics (for example, complexity may be measured using the metric of the code's cyclomatic number).

It is worth noting that the work of Heitlager, Kuipers, and Visser is based on the predecessor to ISO/IEC 25010:2011, ISO/IEC 9126-1:2001 [19], and these two standards differ significantly. The most significant changes within the scope of maintainability are the addition of the modularity and reusability sub-characteristic as well as modifiability, a combination of the sub-characteristics that are referred to as stability and changeability in ISO/IEC 9126-1:2001.

We shall limit our investigation of software sub-characteristic to analysability and modifiability (in ISO/IEC 25010:2011 terminology); we deem that the other sub-characteristics of maintainability are outside the scope of this work. To compute a score for these sub-characteristics from code properties, Heitlager, Kuipers, and Visser proposes an impact matrix that indicates which properties indicate which sub-characteristics. Table 3.3 shows this impact matrix, as well as a conservative interpretation of it in the framework of ISO/IEC 25010:2011. We find that analysability is impacted by code volume, duplication, unit size, and unit testing; modifiability is impacted by unit complexity, duplication, and unit testing.

Finally, each code property is quantified using one or more metrics. Heitlager, Kuipers, and Visser provides at least one metric for each property, but the list is by no means exhaustive. For the measurement of code volume, two metrics are proposed: lines of code of the entire system (LOC), and person-years of effort required to recreate the system. For the unit complexity property, one metric is proposed: cyclomatic complexity per unit. For duplication, a single metric is proposed: the percentage of code that lives in duplicated units of six or more lines. For unit size, the proposed metric is again lines of code, but per unit rather than in the entire system. Finally, for unit testing, two metrics are proposed: the test coverage percentage and the number of assertion statements. For most metrics, the authors provide a method to convert from measured quantities to a five-point rating system (from -- to ++) to simplify the rating process.

Before we make the step to quantifying the code metrics, it is worthwhile to take a step back and observe from a global point of view the subdivision of code quality that is given by the combination of ISO/IEC 25010:2011 and the work of Heitlager, Kuipers, and Visser. Our model for software quality now consists of five distinct layers. At the top lies the abstract notion of code quality, subdivided into eight characteristics of code quality. Each characteristic is divided into between two and six sub-characteristics.

Table 3.3.: An adaptation of Figure 3 from Heitlager, Kuipers, and Visser [16]. Each row represents one of the constituent sub-characteristics of maintainability and each column represents a code property. A cell containing a checkmark indicates that the property corresponding to that column impacts the sub-characteristic corresponding to that row.

	Volume	Unit complexity	Duplication	Unit size	Unit testing
ISO/IEC 9126-1:2001					
Analysability	✓		✓	✓	✓
Changeability		✓	✓		
Stability					✓
Testability		✓		✓	✓
ISO/IEC 25010:2011					
Analysability	✓		✓	✓	✓
Modifiability[1]		✓	✓		✓
Reusability[2]	Undetermined				
Testability	✓			✓	✓
Modularity[2]	Undetermined				

[1] Computed by combining the changeability and stability sub-characteristics from ISO/IEC 9126-1:2001;
[2] Not detailed in Heitlager, Kuipers, and Visser.

Each sub-characteristic is impacted by one or more code properties, and each code property is finally measured by one or more code metrics. If the metrics are quantifiable and each step in the chain of subdivisions has some way of meaningfully combining its constituent parts, it is possible to quantify the abstract notion of code quality from a series of low-level code metrics. A schematic overview of the constituent hierarchy of maintainability is given in Figure 3.2.

In the remainder of this chapter, we will evaluate five of the code metrics: cyclomatic complexity per unit, six-line duplicates, lines of code per unit and in total, and person-years of work. We choose to omit metrics relating to unit testing, as there are no unit tests in the Athena code base. When we have gathered values for these metrics, we will compute upwards along the hierarchy to come to a statement about the maintainability

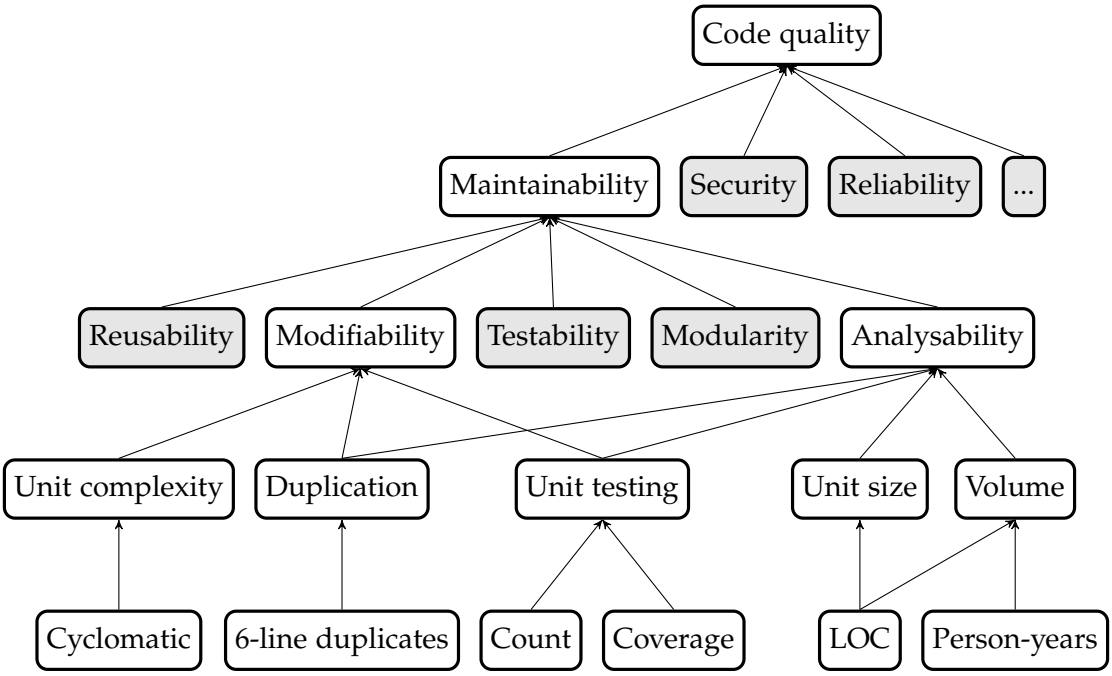


Figure 3.2.: A complete schematic overview of the constituent hierarchy of code quality. Shaded nodes indicate aspects that we consider outside the scope of this chapter.

of the Athena code base. Whenever possible, we will attempt to compute these metrics at three levels: firstly at the level of the entire Athena code base (therefore including a lot of code that is not relevant to our project), secondly at the level of the `Tracking` super-package, and finally at the level of the `TrkGlobalChi2Fitter` package (the core of our project). All metrics will be measured using the code as it exists at the point of commit `971046`, dated May 13 2019.

3.5.1. Lines of Code

Lines of code (LOC) is likely the simplest metric of the ones that we discuss in this section. An initial analysis of code line count was performed using the open source `cloc` utility [9]. This yielded a total of 5.7 million lines of code (excluding blank or comment lines) across fifty-six thousand files with recognised file extensions. The `cloc` utility was able to detect thirty-eight unique languages, of which C++ (including headers), Python, and XML were the most common with 3.9 million lines (3.0 million lines of non-header files), 1.1 million lines, 342 thousand lines respectively. Other commonly found languages include Fortran 77, Fortran 90, CMake, and the Bourne

Table 3.4.: Total code volume by programming language for the complete Athena software, the `Tracking` super-package and the `TrkGlobalChi2Fitter` package.

	LOC by code base		
Language	Athena	`Tracking`	`GlobalChi2Fitter`
Programming			
C++	3.0 million	156 thousand	8 thousand
Python	1.2 million	5 thousand	–
C/C++ Header	924 thousand	56 thousand	<1 thousand
Fortran	71 thousand	–	–
sh/bash	60 thousand	<1 thousand	–
CMake	56 thousand	3 thousand	<1 thousand
Other	39 thousand	<1 thousand	–
Markup			
XML	342 thousand	<1 thousand	–
Qt	29 thousand	–	–
Other	29 thousand	<1 thousand	–
Miscellaneous	2 thousand	–	–
Total	5.7 million	221 thousand	9 thousand

(Again) Shell programming language. The `Tracking` super-package contains a much smaller number of lines of C++: 156 thousand. The `TrkGlobalChi2Fitter` package has a lower code volume still: almost nine thousand lines of C++ code. A more extensive breakdown of the number of lines of code per package and per language is given in Table 3.4.

As noted previously, we are not only interested in total code line numbers, but we also with to compute the number of lines of code per code unit. Indeed, the total number of code lines contributes to the *volume* property, and the number of code line per unit contributes to the *unit size* property. Unfortunately, the `cloc` utility does not support counting of lines per unit, and thus we resort to a different tool: `lizard` [36]. Like `cloc`, `lizard` is open source and supports a wide range of languages. Unlike `cloc`, however, it supports more in-depth analysis with metrics like lines of code per unit, and cyclomatic complexity. It also supports detection of code duplication. It is worth noting that code volume numbers are not directly comparable between `cloc` and `lizard` since the two utilities use slightly different methodologies to determine what is and what is not a line of code.

In terms of code volume per unit, we observe substantial differences between the `TrkGlobalChi2Fitter` package and the rest of the Athena code base. Both the entire

Athena code base and the `Tracking` super-package are host to a large number of small code units: the mean volume per unit is 15.9 and 15.6 lines per unit, respectively. The `TrkGlobalChi2Fitter` has relatively more high volume units, with a mean unit volume of 46.0 lines of code. These differences are also shown in Figure 3.3. Curiously, Heitlager, Kuipers, and Visser [16] do not provide any limit on the number of lines of code per unit. Therefore, we cannot concretely assess the quality of the Athena software in terms of unit size.

3.5.2. Person-Years of Work

To compute the number of person-years of work required to reproduce the Athena source code, we refer to the figures found in Section 3.5.1, and an extensive list of programming languages by productivity presented by Jones [21]. Unfortunately, of the three major languages found in Athena, the aforementioned productivity table only has information on C++, and we were unable to find a more recent, similar resource. For C++, the stated number of statement lines per function point is 53, so the 3.0 million lines of non-header C++ contain an estimated fifty-six thousand function points. Furthermore, Jones estimates the "level" of C++ to be 6.0, meaning that a programmer should be able to write up to around 20 code points per month, or 240 function points per year. Hence, we estimate that the C++ component of Athena alone comprises over 235 person-years. Heitlager, Kuipers, and Visser [16] consider systems with more than 160 person-years to be "extremely large", receiving the lowest possible maintainability rating of `--`. The `Tracking` super-package corresponds to approximately 12 years of programming, enough to earn the second-best rating of `+`. The `TrkGlobalChi2Fitter` package contributes less than one person-year of work. This is enough to earn the highest rating of `++`.

3.5.3. Cyclomatic Complexity

Cyclomatic complexity or McCabe complexity is a widely used measure for the complexity of code initially presented by McCabe [25]. Put simply, the cyclomatic complexity of a unit of code is equal to the number of independent paths through that unit. In the seminal paper on the subject, cyclomatic complexity was found to be a good indicator of the testability of code, and subsequent research also showed correlations with the ease and speed with which programmers could understand and modify code [8].

McCabe recommended to keep the cyclomatic complexity of units at or below ten, and a consensus has since formed around this limit. Bray et al. posit four classes of

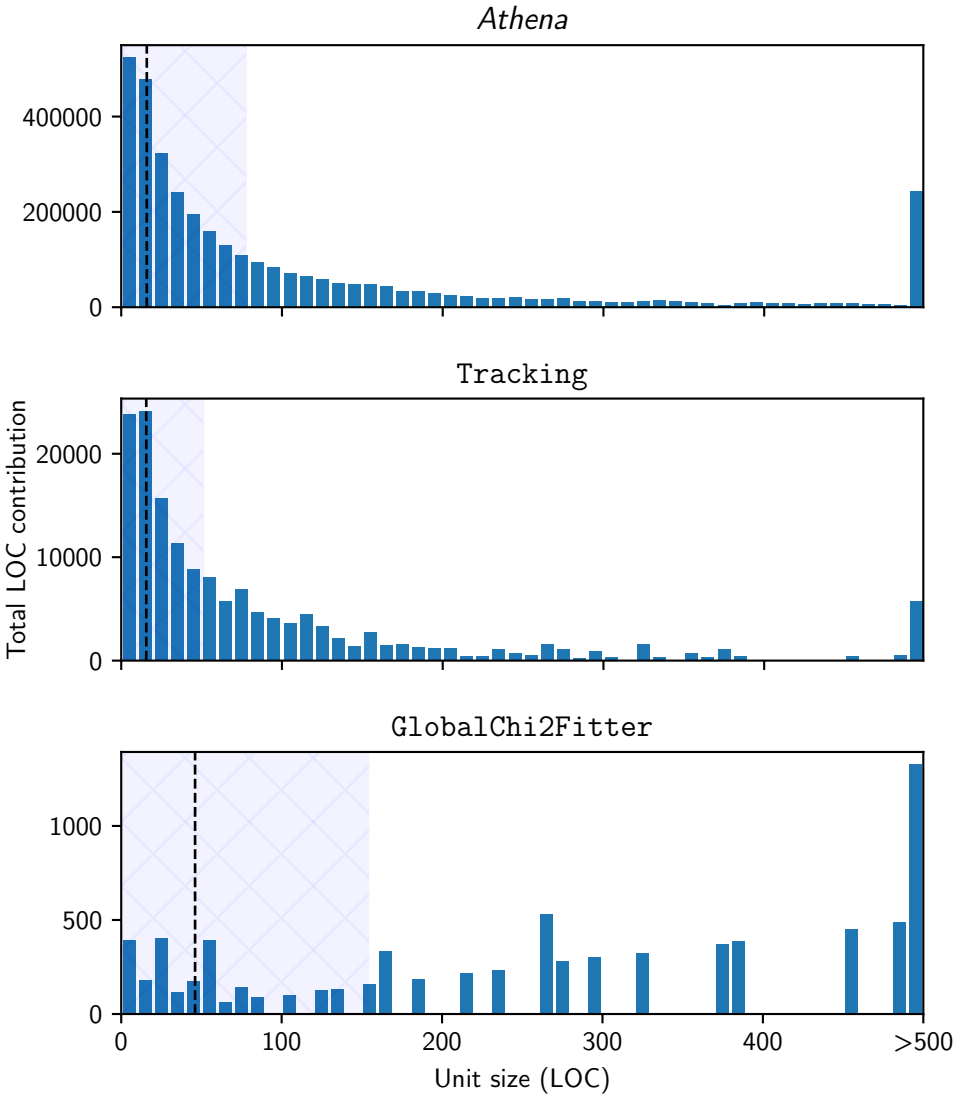


Figure 3.3.: Distribution of per-unit code volume for the entire Athena software, the `Tracking` super-package and the `GlobalChi2Fitter` package. The `GlobalChi2Fitter` has a mean unit volume almost three times higher than the rest of Athena (46.0 LOC per unit compared to 15.9). Shaded areas cover the area within one standard deviation from the mean.

Table 3.5.: Analysis of unit complexity classes in the Athena code base. Results given in terms of unit count (number of functions within each complexity class) as well as relative number of lines of code (number of lines of code within functions of that complexity class).

	Code base		
Complexity	Athena	`Tracking`	`GlobalChi2Fitter`
Unit count			
0 – 10	200091 (93.47%)	9341 (92.97%)	137 (80.12%)
10 – 20	8232 (3.85%)	438 (4.36%)	12 (7.02%)
20 – 50	4491 (2.10%)	205 (2.04%)	8 (4.68%)
> 50	1256 (0.59%)	63 (0.63%)	14 (8.19%)
Relative lines of code			
0 – 10	1990494 (58.46%)	89259 (55.73%)	1171 (14.87%)
10 – 20	493654 (14.50%)	26864 (17.16%)	609 (7.74%)
20 – 50	525538 (15.43%)	23508 (15.01%)	1026 (13.03%)
> 50	395447 (11.61%)	18939 (12.10%)	5067 (64.36%)

units, where units with complexity less than ten are considered simple and at little risk of faults; units with complexity between ten and twenty are considered more complex and at moderate risk of faults; units with complexity between twenty and fifty are considered complex and at high risk of faults; finally, units with complexity over fifty are considered very complex, untestable and at a very high risk of faults.

We use the `lizard` utility to compute both the number of code units that fall into each of the four complexity classes as well as the total number of lines of code that these units contain. The results of this analysis are shown in Table 3.5. It is clear from these results that the amount of high complexity code is much higher in the `TrkGlobalChi2Fitter` package than in the rest of the code. Indeed, a staggering 64.36% of all lines of code in the `TrkGlobalChi2Fitter` package are contained within units with very high complexity and should be considered to be at a very high risk of faults. The package also contributes over a quarter of the high complexity relative code volume within the `Tracking` super-package while contributing less than a twentieth of the total code volume.

Heitlager, Kuipers, and Visser [16] propose a rating schema based on the relative code volume per complexity class where any software with more than five percent very-high complexity code is given the lowest rating of `--`. Under this schema, the Athena package as a whole, the `Tracking` super-package and the `TrkGlobalChi2Fitter` package would all receive the lowest possible rating.

3.5.4. Six-Line Duplicates

To find the duplication rate in the Athena code base, we once again use the `lizard` utility. Our methodology is to find all duplicates of six lines or more and to count the number of lines in such duplicates. We exclude the first entry in each list of duplicates, since it is necessary to have a single copy of the code in order for it to be reusable. The `lizard` utility ignores certain types of formatting such as white space to improve the accuracy of the duplication detection. In addition, this analysis can detect block of code that are not identical but extremely similar, such as blocks of code that are identical apart from the names of variables. Notably, this is a departure from the methodology proposed by Heitlager, Kuipers, and Visser, who only consider exact matches to be duplicates.

We find that the `Tracking` super-package has a duplication rate (ratio of duplicate lines to the total number of lines) of 27.5% (--), and the `TrkGlobalChi2Fitter` has a duplication rate of 15.0% (-). Unfortunately, we were not able to complete an analysis for the entire Athena code base, as the computational resources required for a duplication analysis on such a large volume of code were not readily available. Although a larger code base is inherently more likely to contain duplicates due to its volume of code, it is possible that more effort was made in the rest of the Athena code base to avoid duplication. Therefore, we cannot make predictions in good faith about the duplication rate of the Athena code base.

3.5.5. Combined Results

Before we combine the results of the previously measured properties into ratings for the analysability and modifiability sub-characteristics of the software, we must first address two shortcomings in the data we have gathered. Firstly, as mentioned in Section 3.5.1, we do not have a concrete schema for converting between unit size metrics and a unit size property rating. In lieu of such a schema, we opt to assign a rating for the unit size property equal to the rating for the unit complexity for each unit. The reason for this is the strong correlation between the lines of code per unit and the cyclomatic complexity of a unit [20]. We observe a similar correlation in the Athena code base, with a Pearson moment of correlation between lines of code per unit and complexity per unit of 0.529 for the entire Athena code base, 0.906 for the `Tracking` super-package and 0.986 for the `TrkGlobalChi2Fitter` package. Secondly, as mentioned in Section 3.5.4, we are missing a rating for the duplication property in Athena. To resolve this, we opt to simply omit it from our evaluation.

In order to calculate the rating for a sub-characteristic, we calculate the average rating of all properties that are deemed to correlate with the sub-characteristic as shown in Table 3.3. For example, to calculate the modifiability rating for a certain segment of the code, we take the average of the ratings for code volume, unit complexity and unit size. Using this methodology, we arrive at the lowest possible rating of `--` for the entire Athena code base in terms of both analysability and modifiability. For the `Tracking` super-package and the `TrkGlobalChi2Fitter` package, we arrive at a rating of - for both sub-characteristics. A complete breakdown of ratings is given in Table 3.6.

In conclusion, we have applied a structured methodology in order to gauge the ease with which we should be able to analyse and make changes to the Athena code base. Since we consistently arrive at a low rating for both analysability and modifiability, we should be prepared for a rather arduous process in which analysis is hindered by code quality and where implementation time may contribute a significant overhead to the overall project workload.

Table 3.6.: ...

	Volume	Unit complexity	Duplication	Unit size[1]	Unit testing	Rating
Athena	--	--	?	--	--	
Analysability	✓		✓	✓	✓	--[2]
Modifiability	✓	✓		✓		--
`Tracking`	+	--	--	--	--	
Analysability	✓		✓	✓	✓	-
Modifiability	✓	✓		✓		-
`GlobalChi2Fitter`	++	--	-	--	--	
Analysability	✓		✓	✓	✓	-
Modifiability	✓	✓		✓		-

[1] The unit size property is set to be equal to the unit complexity property due to the lack of a concrete rating schema and the strong correlation between the two properties; [2] Rating based on partially unknown data.

4. Experimental Design

In this chapter, we will describe the way in which the experiments presented in this work were conducted. We will detail the hardware on which experiments were performed, the exact state of the code used, the way in which it was built and compiled. The aim of this section is firstly to make the results of this thesis as reproducible as possible, and secondly to provide some insight into which factors need to be taken into account when conducting a software optimisation project. All experiments discussed in this thesis are performed *in situ*. That is to say, the analysis is performed on a full running copy of the Athena source code. During the initial stages of our project, we believed that it may be viable to extract the fitting code from the remainder of the Athena software to conduct *in vitro* experiments, but this was deemed impractical due to the high degree of coupling to other modules in Athena.

4.1. Hardware and Operating System Description

We conduct all our experiments on the *Stoomboot* computing cluster which is owned and operated by Nikhef. The cluster is managed using the TORQUE resource manager (TORQUE is an implementation of the *Portable Grid System* or PBS) [33], and it contains twenty-four standard compute nodes and two additional GPU nodes. As far as we know, each standard node is identical, containing a single AMD EPYC 7551P CPU and 256 GB of memory. The AMD EPYC 7551P processor is based on the *Zen* microarchitecture and uses the *Naples* core. This is a 32 core processor with support for simultaneous multithreading, allowing up to 64 threads. It supports many modern instruction set extensions including SSE4.2, AVX, and AVX2. However, the processor does not support AVX-512, and the support for AVX2 is implemented using joined 128-bit execution ports as opposed to the dedicated 256-bit or even 512-bit circuitry found in many competing Intel processors and more recent AMD Zen 2 processors. It has a base clock frequency of 2.00 GHz and a maximum boost frequency of 3.00 GHz up to twelve cores and 2.55 GHz when more than twelve cores are in use. The processor features, per core, a 32 kB L1 data cache and a 64 kB L1 instruction cache. In addition. each core has its

own 512 kB L2 cache. Finally, a 8 MB L3 cache is allocated to each *core complex* (*CCX*); the EPYC 7551P contains eight core complexes of four cores each, for a total of 64 MB of L3 cache.

The Stoomboot nodes run on the *CentOS 7* operating system; CentOS 7 is the *de facto* standard for ATLAS computing since the recent migration from the now-obsolete *Scientific Linux 6* distribution. The nodes use version 3.10.0 of the Linux kernel.

4.2. Package Selection

To build the software during our project, we used the CMake build system that is standard for the Athena package. However, there are several ways in which our build process differs from the default build process. This section is meant to highlight these differences, such that our build process can be more easily replicated.

The first consideration is which parts of Athena are compiled and which are not. As mentioned in Chapter 3, the Athena project consists of over two thousand modules and contains over five million lines of code. This means that compiling the complete Athena software takes such a long time that it is not conducive to a project where quick iteration is desired. Compiling the entire software package takes several hours even on dedicated continuous integration machines. In an optimisation project, it is often useful to rapidly profile and benchmark many variations on the same code, and as such it is impractical to have such long build times.

To reduce compilation times, it is possible to compile only a subsection of the Athena software. This is achieved either by retrieving only a subsection of the source code, or by specifying during the build process which packages to build and which ones to ignore. In theory, this allows developers to compile only the parts of the code that they have modified, saving time. Any missing modules can be retrieved as precompiled shared objects from the shared repository as described in Section 3.1.

In practice, compiling only modified modules is unfortunately not always adequate for benchmarking the performance impact of changes. Many routines in the code base are defined in header files or inlined, meaning that they are not compiled solely into the module to which they belong. To consistently gauge the impact of changes made to such a function, one would have to compile every single module using it. One header function might even call another header function (possibly one from a different module), quickly creating a complicated system of connections between modules such that it is difficult to determine what to compile and what not to. In some cases, changes to code in one module might even cause errors in other, non-compiled modules if the

change makes the newer modules binary incompatible with the older ones. Although this should in theory be prevented by the architecture of Athena, careless programming practices have made this a real problem for Athena development.

In this thesis, we have adopted a broad-stroke approach and opted to compile a larger subsection of the source code than might be strictly necessary. While this means that compilation times are longer, we increase the accuracy of our benchmarks such that the performance observations are closer to what would be observed in a full build of the software. For this thesis, we have chosen to build three entire package categories: `Tracking`, `MagneticField`, and `MuonSpectrometer`. These three categories contain a total of 208 individual packages. On top of that, we compile many packages from other sections of the code base, either because they contained bugs that needed to be fixed, or to provide compatibility with changes we made in the aforementioned packages. A full list of packages that we have chosen to compile is given in Appendix B.

4.3. Build Environment

Within the ATLAS collaboration (as well as the rest of the LHC experiments), it is customary to compile software against a so-called *LHC Computing Grid* (LCG) environment. LCG environments are versioned and contain a large number of software packages, including C++ libraries, a Python interpreter, Python libraries, and more. Each LCG version is distributed for different architectures, operating systems, compilers, and release types. By making LCG environments available through the shared CVMFS network file system, this makes it possible to make a consistent build environment available for all developers around the world.

Our project is compiled against LCG 95, the recommended version for compiling Athena as of April 1, 2019. We use the version for the x86-64 architecture, the CentOS 7 operating system, the GCC 8 compiler, and the non-debug build (the full name of this environment is `x86_64-centos7-gcc8-opt`). A full list of packages and their versions in this release of LCG is available at `http://lcginfo.cern.ch/release_packages/x86_64-centos7-gcc8-opt/95/`. The most important piece of software not included in LCG is the compiler. As far as we are aware, the most common compiler for Athena is GCC, and releases of GCC are made available alongside LCG environments on the shared file system. Throughout this thesis, we use GCC version 8.3.0.

4.4. Build Options

While Athena packages can be individually configured to use different compiler options, the vast majority of them are compiled with a default set of flags. Throughout this thesis, we used the `RelWithDebInfo` build type, which optimises in the same way as a normal release but with additional debugging info. Omitting compiler warning flags, this means that the software is compiled with `-DNDEBUG` to disable assertions which impact performance, `-O2` as the optimisation level[1], `-g` to include debugging information, `-fPIC` to produce position-independent code (necessary for shared objects), `-pthread` to enable AthenaMT, and `-std=c++17` to use the ISO/IEC 14882:2017 C++ standard (commonly known as C++17).

GCC will use default values for many options if they are not explicitly specified. For the sake of brevity, we will not list all of them. However, it is worth reiterating that the compiler defaults to use a generic x86-64 architecture. This means that the binaries produced sacrifice performance to be backward-compatible with any x86-64 CPU.

In this thesis, we make several modifications to the default Athena compilation flags. We have made these changes either out of necessity for profiling, or because we believe they more accurately show the performance improvements that can be achieved through optimisation. An example of a change that was necessary for profiling was the preservation of frame pointers. A frame pointer is a part of a stack frame that is used in stack unwinding; when a frame is popped of the stack, the frame pointer guides the stack pointer to the new top frame on the stack, and this process can be repeated to unwind the entire stack. In many cases, it is possible for the compiler to omit the frame pointer from the stack frame, freeing up a register and reducing register pressure. However, frame pointers can be useful tools for profiling and debugging, making their omission undesirable. Therefore, we build the Athena software with the `-fno-omit-frame-pointer` flag, which disables the frame pointer omission optimisation. The use of frame pointers in profiling will be further detailed in Section 5.2.1.

Furthermore, we have opted to move away from the generic x86-64 compatibility requirement and we have instead configured the compiler to use the microarchitecture corresponding to the machine on which the code will be run using the `-march=znver1` flag. It is worth noting that this also implicitly sets the `-mtune=znver1` flag, which means that the compiler will optimise using cost models specifically tuned to the chosen microarchitecture. We believe that compiling for modern CPU architectures will show

[1] More information about GCC 8.3.0 optimisation levels can be found at `https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/Optimize-Options.html`

more clearly how much performance can be gained by fully utilising the capabilities of modern processors. Naturally, both the baseline benchmark and the benchmarks of our optimised code were done using this compiler option (unless otherwise noted) to provide a fair comparison.

Finally, the change to target modern CPU architectures exposes an additional problem in the configuration of Athena which requires additional compilation flags to resolve. Compiling parts of the code with support for 256-bit or 512-bit vector registers causes the Eigen library to emit code for types that is ABI-incompatible with code compiled for the same types with 128-bit vector registers. Specifically, Eigen assumes 16-byte memory alignment when compiled for 128-bit registers (the default) and 32-byte or 64-byte alignment when compiled for 256-bit or 512-bit registers respectively. Since 32-byte or 64-byte aligned memory is always 16-byte aligned but the reverse is not necessarily true, objects using compiled for different vector widths may contain Eigen types that are not compatible with each other.

To resolve this issue, we have opted to enforce 16-byte alignment in Eigen, regardless of the vector register size. For systems using AVX or AVX-512, this means that unaligned memory will be used and that the program may incur a performance penalty [24]. Still, this is the only way to ensure ABI-compatibility without compiling the entire Athena project. The compiler flag used is `-DEIGEN_MAX_ALIGN_BYTES=16`. Enforcing 16-byte alignment is for all intents and purposes a way to make the code backwards compatible. It is alternatively possible to make the code forward-compatible by enforcing 64-byte alignment throughout the entire Athena code base. Since 64-byte aligned memory is trivially 32-byte aligned and 16-byte aligned, this would allow any microarchitecture to operate without slowdown at the cost of slightly increased memory usage due to increased padding. We have proposed this change on the official ATLAS issue board, but it has not been implemented at the time of writing.

4.5. Software State

Athena is a continuously evolving piece of software, with hundreds of merge requests open at any time and dozens of them being merged into the software every day. Our development takes place on the most up-to-date branch of the software (the `master` branch, for the upcoming release of Athena 22). To provide a reproducible baseline for our optimisation efforts, it is crucial that we describe as precisely as possible the initial state of the software, as well as any changes we have made to it that are not directly related to optimisation. The baseline version of the Athena software used is a

snapshot of how the code was on April 1, 2019. This version of the software is tagged as `nightly/master/2019-04-01T2157`, and refers to the commit hash `7a5ace5c97` in the project's version control system. This version of the software was chosen as it accurately represents the state of the software as it existed at the start of our project.

The chosen version of the software was not free of bugs, however, and several patches needed to be applied to ensure that the code was in a state fit to be executed and tested. Firstly, several patches were backported from a later version of the `master` branch, fixing an error that caused the physics performance summaries described in Section 4.7 to report incorrect results. These fixes were critical to ensure that we would be able to correctly evaluate the physics performance of our optimised version of the software. Then, several segmentation violation errors had to be resolved. We are unsure how these errors entered the code in the first place, as Athena merge requests are automatically tested. It may be the case that a rather obscure set of configuration parameters in our job settings brought these bugs to light. We developed two patches that resolved these faults, and those patches were quickly merged into the main branch of the software. Several less critical bugs were also discovered, fixed, and those fixes were suggested into the `master` branch.

We also applied several patches containing code style changes. When developing these changes, we were careful to avoid changing the performance of the code wherever possible. These code style changes and some refactoring were included to improve the ease of development, since code quality is a significant issue in the Athena source code (see Section 3.5). Some changes to ensure thread safety were also included, and the build system was configured to automatically check the thread safety of some of the modules that this thesis primarily focuses on. Several changes were made to the call graph of the software to aid profiling tools. Finally, we added some benchmarking mechanisms to the software, such that we can more accurately measure the performance of the relevant code. For example, code was added that logs the execution time of each fitter call, and dumps each of those records to a file at the end of the reconstruction job.

4.6. Input Data

In particle physics experiments, no two collision events are the same, and different events can have vastly different computational performance characteristics. One event might contains orders of magnitude more particles than another, for example, and take much longer to process. In an ideal world, we would characterise the performance of the track fitting algorithm on all possible classes of events, but this would be an

insurmountable task. It is crucial, therefore, to have a series of events that can be used as input that are well-described, somewhat uniform and well-suited for performance experiments.

In this thesis, we have opted to use a dataset of simulated di-muon events; that is to say events in which a decay takes place of one particle into two muons. There are three primary benefits to using this dataset. Firstly, since the events are simulated, we have access to the so-called *truth data*, which is the true simulated path of the particle through the detector. This is one way to verify the correctness of the program and any changes we make to it. Secondly, these events have roughly the same structure and therefore should behave at least somewhat similarly in terms of performance, giving more consistent performance measurements. Finally, the muon data fits well with the broader goal of this work, which is to optimise the track fitting software specifically for the muon system.

The simulated data was kindly provided by ATLAS muon system researcher Stefano Rosati, and can be found in the ATLAS data repository as `group.det-muon.DiMuon10_100GeV.RDO.rel21_3_12.FullATLAS.v001_EXT1`. The data contains particles with a relatively high transverse momentum in the range 10 GeV to 100 GeV and the simulation was done with the full ATLAS geometry, including the New Small Wheel, an upgrade to the muon spectrometer system that is in the process of being installed at the time of writing.

4.7. Output Validation

When making changes to code in order to improve its performance, it is vital to ensure that the output of the program remains valid. Indeed, code that produces incorrect results is of little use to anyone, regardless of how performant it may be. The exact definition of output validity, and the method by which it is determined, differs from project to project. The purpose of this section, therefore, is firstly to identify possible properties of the program output that we can use to determine if the output is valid, and secondly to establish acceptable thresholds of deviation from the reference output.

In the Athena tracking software, we identify two possible methods to check whether the output of our modified code is still valid. The first method follows from the fact that tracks can be characterised by a quintuple (the *magic five*) describing the perigee position of the particle, as well as its angle and momentum. There are various combinations of five variables that can be used to describe a track. Athena dedicates three variables to the position (in global polar coordinates) of the perigee, one variable

to the pseudo-rapidity, and one variable to the momentum of the particle. By comparing the quintuples describing each track before and after making changes to the code, we can verifying that the difference between them lies within an acceptable threshold. We can then be confident that the output of the program is still valid. Unfortunately, this method fails in practice, since the execution of some fits is entirely dependent on the output of other fits; if a fit returns a value that is non-problematically different from the reference (for example, due to floating point imprecision), this can lead to fits being performed with different parameters in the future. While the resulting final fits would be functionally identical, we would be unable to verify the correctness of the intermediate fits against the reference output. In other words, we can say that the fitter reacts chaotically to changes to its input, making it impractical to verify on this level.

When running Athena on simulated data, as we do in this thesis, the software has access so so-called *truth tracks* which describe precisely the path a particle travelled through the detector (this is known precisely because the data is simulated). After processing all input events, the Athena process reports statistics about the physics performance of the run. That is to say, it reports how many tracks were correctly reconstructed, as well as how many fakes (reconstructed tracks that were not present in the actual data) were reconstructed, and several more metrics. The most important metrics that are reported are the efficiency, which is the fraction of all truth tracks that were accurately reconstructed, and the fake rate, which is the number of fake tracks that were reconstructed, on average, per event. In other terms, the efficiency corresponds to the recall of the reconstruction, and the fake rate equals the false positive rate. A truncated example of a result summary is given in Listing 4.1. It is important to note that the result summary counts only final fits, and each final fit can be the result of a hundred or more partial fits. By verifying that the physics performance metrics of the code after optimisation are similar to the metrics before optimisation, we have a very practical way of verifying that our changes to the code are correct.

We will consider now the question of what are acceptable difference thresholds for this metric. Ideally, we would consider valid only those results which are exactly identical to the reference values, but such a threshold would be difficult to maintain in practice. Since track reconstruction involves such a complex interplay between different functions and processes, minute numerical differences can propagate to non-negligible differences in output. Note that this does not necessarily mean that the new output is incorrect, or less valid than the reference output. Limited-precision floating point arithmetic is inherently inexact for most values, and any reference value will also have suffered numerical inaccuracies. While we must guard against large differences as well

Listing 4.1: An truncated example of a results summary as generated by Athena following a reconstruction job on simulated data.

```
Summarizing results for track collection CombinedMuonTracks  500 events:
  number of tracks                              0   avg per event 0
  number of truth tracks                      820   avg per event 1.64
  number of good tracks                       816   avg per event 1.632
  number of missed multi station tracks         4   avg per event 0.008
  number of missed one station tracks           0   avg per event 0
  number of fake tracks                        59   avg per event 0.118
  number of high pt fake tracks                59   avg per event 0.118
  number of low pt fake tracks                  0   avg per event 0
  number of SL fake tracks                      0   avg per event 0
  number of matched fake tracks                 0   avg per event 0
  number of tracks with lost momentum          12   avg per truth track 0.01463
  number of tracks missing precision layer     33   avg per truth track 0.04024
  number of tracks missing trigger layer        7   avg per truth track 0.008537
  number of tracks missing chamber             59   avg per truth track 0.07195
  efficiency: 0.9951  fake rate 0.118  high pt 0.118  low pt 0
  missed track rate (per truth track) 0.004878 secondary efficiency 0
```

as structural biases, we cannot assume that a result is invalid simply because its output does not exactly match the reference output. We have chosen to allow a maximum absolute degradation in efficiency of 0.5%, and a maximum increase in the fake rate of 0.01 tracks per event. Any result that compares to the reference less favourably than the threshold is rejected. Any result that performs better than the reference is, of course, acceptable.

It is important to note that the results of Athena reconstructions job are not perfectly deterministic. We observe that identically configured jobs with identical binaries can result in slightly different outputs. The differences are significant enough that they can impact the efficiencies and fake rates of reconstruction jobs. We are unsure whether this behaviour is expected (for example, due to time-based seeding of random number generators) or if it is caused by a bug in the software. Still, we must consider these non-deterministic effects when evaluating the physics performance of different versions of the Athena source code. Fortunately, the effects are rather small; the difference between the highest and lowest efficiency in an experiment consisting of 50 runs using the base software state describes in Section 4.5 amounts to 0.10% (with a standard deviation from the mean of 0.03%), whereas the range in fake rates amounts to 0.005 fakes per event (with a standard deviation of 0.0011 fakes per event), roughly 2.74% of the mean value.

4.8. Experimental Noise

In many performance engineering experiments, it is possible to benchmark code in such a way that the results are relatively consistent. This usually requires control over the host machine to ensure that core (and other resource) contention is not an issue, and it requires that the program's performance does not depend heavily on external processes. Neither of these conditions are necessarily met in our experimental setup, so it is important that we elaborate on why this is not the case, and what levels of noise we can expect in our measurements.

Since we are running our experiments on a shared computing cluster, some sources of noise are inherently present in our experiments. One seemingly obvious reason would be that we may be assigned nodes with differing CPU types, but this is not actually a problem since, to the best of our knowledge, all nodes have identical hardware setups. Rather, an issue can be that a node assigned to our benchmarks might simultaneously be in use by another user[2]. Although the scheduler pins each job to a fixed core to reduce scheduling overhead, we may still be competing with other users for resources such as disk and network bandwidth. Furthermore, Athena spawns several auxiliary processes such as memory monitors; if only a single thread is available to both Athena and these additional processes, performance may degrade due to competition on the thread. Theoretically, degraded performance may also result from Athena and its auxiliary processes to be running on processes in separate NUMA domains, but we consider it unlikely that this will result in any statistically significant differences. Finally, as noted in Section 4.1, the clock frequency of the processors used can vary significantly depending on the number of cores in use. However, Stoomboot nodes appear to have their clock frequency adjustment mechanisms disabled, and the clock frequencies of nodes appears to be pinned to the lowest available value of 2.00 GHz[3].

In order to measure the noise generated involved in our experiments, we have performed fifty identical reconstruction jobs, scheduled by the grid resource management software, in order to best simulate conditions as they would be during an ordinary reconstruction job. This experiment was repeated four times at different points in time to gauge not only the level of noise, but also its variability. We find that the results are relatively consistent both between the different experiments, as well as between samples

[2] While it would be possible to request a whole node to ourselves, this would be inconsiderate towards other users looking to use the grid.

[3] We do not know for sure that this is the case, but we have monitored the clock frequencies of nodes over extended periods of time and we did not observe any instance of the CPU clock frequencies exceeding this value.

within the same experiment. Indeed, the total fitting time differs by no more than 0.76% between experiments at different times in the day. Furthermore, the percentage standard deviation from the mean for each of the four times the experiment was executed range between 1.19% and 2.02%. Over the entire range of samples ($N = 200$), we find a percentage standard deviation from the mean fitting time of 1.63%. We therefore expect our results to be consistent and reliable despite the nature of experimental environment.

4.9. Job Configuration and Submission

To ensure that our results are as reproducible as possible, it is vital that we describe in detail the way our benchmarking jobs were submitted to the grid, in terms of both Athena job configuration as well as the PBS grid job parameters. The Athena job options are given in their entirety in Appendix A; these configurations were provided to us on the premise of them being representative of how muon system reconstruction jobs are commonly configured. The Athena jobs were additionally configured to reconstruct exactly 1000 events. We believe that this number of events presents a good balance between the total length of each reconstruction job and the representation of a wide range of different events[4].

In regards to the PBS grid submission system, our job submission parameters are relatively simple. Indeed, all nodes in the default grid queues are identical, and as such there is no need to differentiate between hardware configurations to obtain accurate results. Each job was configured to request two threads on a single node to minimise the impact of auxiliary Athena processes (see Section 4.8), and the requested wall clock time was two hours. In practice, the average total job run time was approximately forty minutes, but jobs would occasionally significantly exceed this average, presumably due to IO-related issues.

[4]Each event has different characteristics, and these can impact the performance behaviour of the reconstruction significantly. By selecting a large number of events to reconstruct, we ensure a sufficiently diverse set of such behaviours.

5. Performance Analysis

The purpose of this chapter is to gain a deeper insight into the performance behaviour of the `TrkGlobalChi2Fitter` code. To that end, we will then applying more advanced profiling techniques to gain an understanding of the performance hot spots, possible areas of improvement, and how to meaningfully benchmark it. We will also conduct benchmarks on the fitting code as a whole, and we will attempt to create a model for its performance. All analyses in this chapter are conducted in accordance with Chapter 4.

5.1. Benchmarking

It is possible to split the code of the `TrkGlobalChi2Fitter` package into roughly two parts: the core in which most of the actual computation takes place, and the prologue in which data is converted from a format compatible with the interface for fitting tools to a format that is compatible with the core. There exists a multitude of methods to enter the prologue, but they all lead to the same single entry point to the core algorithms. This makes benchmarking the core functionality far easier and less prone to errors than benchmarking the prologue and core together. In addition, we feel that this method is more representative of the performance of the fitting algorithm itself; converting between data formats is a necessity not because because the fitting algorithm intrinsically requires it but rather due to the design of the system. For this reason, we choose to benchmark only the core of the fitting tool in this chapter. We opt to keep the design of our initial benchmarks extremely simple, using a high-resolution monotonic wall clock to measure the time at the start and end of the execution of the core fitting code. This methodology is resilient, and provides results that match closely with the observable performance of the software.

The results of our initial benchmarking are shown in Figure 5.1. It is immediately apparent that individual calls to the fitting method can vary significantly in performance; run time ranges between roughly 100 µs and 20 ms, a range of more than two orders of magnitude. Extreme values are observed as low as 18 µs and as high as 78 ms, but we consider these statistical anomalies. The mean run time is 515 µs. The multimodal

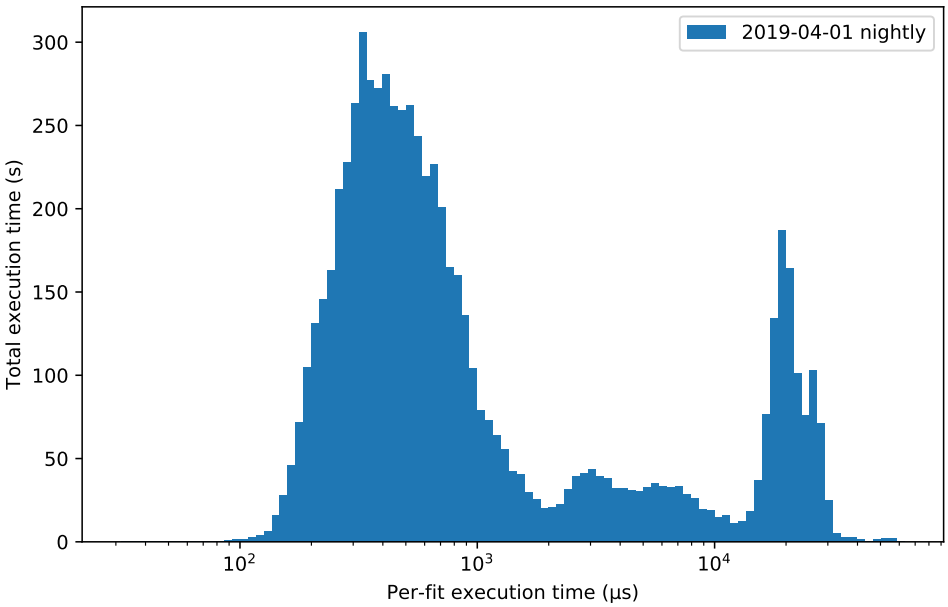


Figure 5.1.: Results of benchmarking the run time of individual calls to the `GlobalChi2Fitter::my_fit` method. The vertical axis represents the total run time contribution of each bin.

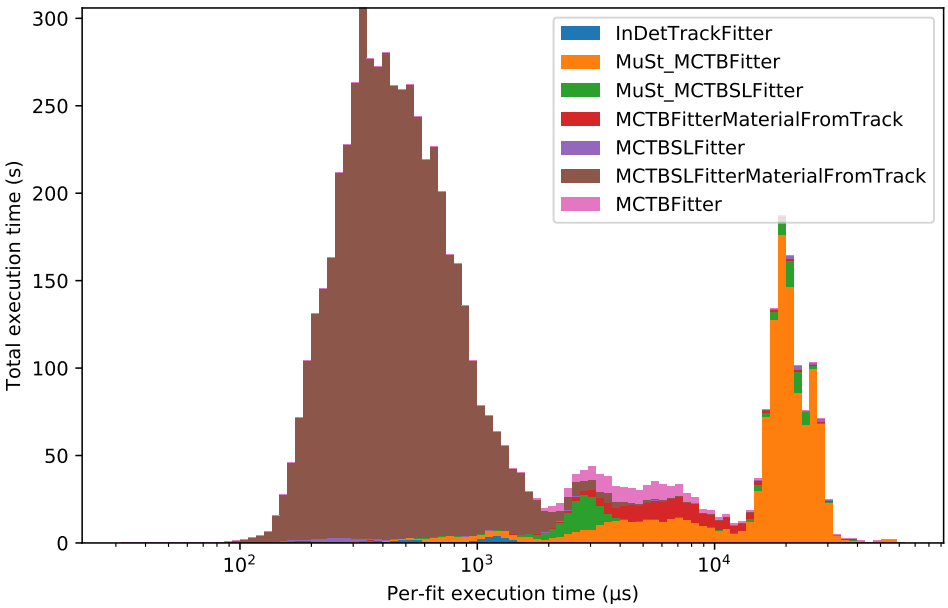


Figure 5.2.: Results shown in Figure 5.1, separated by the name of the fitting tool to show how significant the difference in performance is between different tool instances.

distribution of the data suggests that simple, descriptive statistical methods are insufficient to describe the behaviour of track fitter's performance. It could indicate that there are some hidden structures or patterns in the input data, but the true explanation for this behaviour lies in the architecture of the Athena software

As we will see in more detail in Section 5.2, the multimodal nature of our benchmarking data stems from the fact that multiple, separately configured tools can exist simultaneously in the same reconstruction job. Using techniques described in the next section, we can distinguish individual fits by the fitting tool used. The benchmarking data separated like this is given in Figure 5.2 and Table 5.1. We observe that the fit execution times for each tool are, generally speaking, monomodal and more easily described than the distribution for all tools combined. It is clear from the data that some fitters contribute to the total run-time by performing many short fits, while others contribute by performing far fewer fits that each take more time. The fact that these fitting tools behave so differently will be key to our future optimisation efforts, and we will need to take this into account when applying optimisations to the code.

Fitter name	Total	Per-fit	Fits
`MCTBFitter`	9.2 s	3963.6 ± 2019.7 µs	2309
`MCTBFitterMaterialFromTrack`	12.4 s	5740.3 ± 3514.5 µs	1958
`MCTBSLFitter`	2.0 s	436.7 ± 1949.9 µs	4599
`MCTBSLFitterMaterialFromTrack`	322.8 s	395.3 ± 252.5 µs	816726
`MuSt_MCTBFitter`	75.9 s	8518.9 ± 8850.4 µs	8910
`MuSt_MCTBSLFitter`	11.8 s	1770.6 ± 3368.6 µs	6665
`InDetTrackFitter`	2.1 s	699.3 ± 360.7 µs	2978
Combined	435.0 s	515.4 ± 1355.6 µs	844145

Table 5.1.: Execution times of different fitting tools. Total fitting time and number of fits are reported for a single reconstruction job.

5.2. Profiling

Profiling software can provide critical insight into its performance behaviour, but it can also be significantly more complex than simple benchmarking; relatively simple timers no longer suffice, and we will need to employ more advanced profiling tools. For our analyses throughout this chapter, our tool of choise will be the Linux `perf` performance profiling subsystem. There are some additional challenges presented by the nature of the Athena software which we will need to tackle in order to gain meaningful profiling results. Specifically, we observe two challenges to overcome: recording stack information in such a large code base, and the separation of vastly different performance behaviours that arise from the same code base. The purpose of this section is to elaborate on these challenges, our attempts to tackle them, and to show the results of our profiling efforts.

5.2.1. Call Stack Recording

Since the tracking code involves many intricate interactions between functions, it is critical that our analysis includes call graph data. The version of `perf` used in our analysis supports three distinct methods for gathering call graph data: using frame pointers, using hardware last branch records, and using DWARF debugging information.

Using frame pointers for call graph analysis is perhaps the most obvious choice of the three. Indeed, the primary goal of the frame pointer is to assist in unwinding the stack during normal program execution. However, omission of frame pointers is a very common compiler optimisation which renders this method call graph analysis

impossible. Since the frame pointer is stored in a register, there is a significant benefit to be gained in terms of register pressure by omitting it whenever possible, and this optimisation quickly became widespread. Nowadays, most processors have enough registers that the benefit of frame pointer omission is less noticeable, but it is still a common optimisation. Frame pointer omission is also performed by the Athena build system, making this method of call graph analysis impractical unless it is disabled for the *entire* Athena code base.

Some modern Intel processors contain a last branch record (LBR) facility, a fixed size circular buffer implemented in hardware that records jumps in program flow with very little overhead. A profiling tool can access the contents of the LBR and unwind it to recover the stack of the program. This allows for a method of unwinding the stack with low overhead. Unfortunately, several downsides mean that LBR is impractical for profiling Athena. To the best of our knowledge, no processors are available at the time of writing with an LBR buffer capacity of more than thirty-two. This means that the LBR method for analysing call graphs is impractical for programs with all but the shallowest call graphs. Additionally, the AMD processors used for the benchmarking in this thesis do not support LBR at all.

Finally, it is possible to perform call graph analysis using the DWARF debugging format. Specifically, the Call Frame Information (CFI) can be used by profiling tools to retrace the call chain. DWARF debugging information is commonly included in binaries, even when they are not specifically compiled with debugging symbols, and they are also available for Athena releases. There are two main problems with the use of DWARF CFI for call graph analysis, however. Firstly, there is a significant overhead associated with the use of DWARF CFI; the overhead so high that profiling a full reconstruction job becomes unpractical. Secondly, we have found call graph analysis to be highly unreliable, and we frequently encountered critical error in `perf` that rendered the DWARF method of call graph analysis useless. It is unclear to us whether this is an issue that specifically impacts our version of the `perf` subsystem, or if this is a more widespread problem.

In summary, there are significant problems with all three methods of call graph analysis supported by `perf`. Without a way to analyse the complex call chains of Athena, however, profiling would be mostly meaningless. Luckily, we can mitigate the aforementioned issues with analysis by frame pointer to make it useful in the context of the Athena software, and we propose two ways of doing so. Recalling that the primary problem with this form of call graph analysis for Athena is that the pre-compiled object files omit frame pointers, the simplest solution would be to manually and completely

compile Athena with frame pointer omission disabled, and to use those object files in lieu of the object files generated by the continuous integration service. Note that this does not imply that the development-compilation cycle must be performed on the complete Athena code base as we warn against in Section 4.2. Rather, the object files with frame pointer omission disabled would act as static fall-back objects and they would not need to be recompiled.

If compiling an entire release of Athena is not possible or not desirable, it is possible to achieve the desired effect by compiling only a subsection of the code with frame pointer omission disabled. Since there is no inherent incompatibility between code with and without frame pointers, it is possible for functions with and without them to exist in the same call graph. Unwinding such mixed stacks results in a call graph containing correct frame pointers (derived from the value of the frame pointer register, usually EBP on x86) as well as bogus ones (derived from that same register, which is treated as a general-purpose register in code without frame pointers and may therefore contain arbitrary data). Any sequence of non-bogus frame pointers still provides meaningful information about the call graph, and if we ensure that the control flow of a given function with frame pointers can only ever reach other functions with frame pointers, the data allows for accurate call graph analysis of the software. To ensure this, we must expand the list of symbols compiled with frame pointers enabled to encompass all possible callees (direct or indirect) of the function in question.

An obvious problem with the aforementioned approach is that it requires knowledge about the call graph, which is the very kind of information that we are attempting to gain; there is a circular dependency. We posit that this can be resolved by bootstrapping the call graph analysis process with one of at least two other forms of call graph data. Firstly, it is possible to use call graph data generated by a static code analysis tool; such analysis does not provide insight into the performance impact of parts of the call graph, but it can be used to determine which functions must be compiled with frame pointers. Secondly, it is possible to iteratively approximate a solution to the question which functions need frame pointers for accurate analysis. Indeed, we can usually detect bogus call graph information, as the reconstructed function pointers will lie outside of the text segments in memory[1]. By identifying bogus children in a call graph, we can add those to the list of modules compiled with frame pointers, and iterate this until there are no more bogus nodes remaining.

[1] Since the register usually used to store frame pointers is used as a general purpose register in code with frame pointer omission, it is possible for this register to point, by chance or by design, at a valid location in a text segment. This may make it significantly harder to detect bogus frame pointers, but we do not consider this edge case in this work.

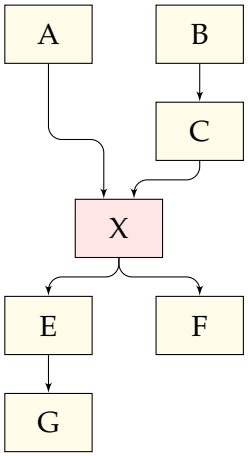
5.2.2. Disambiguation of Performance Classes

In Athena, we observe a many-to-one mapping of calling contexts to performance behaviours due to the use of differently configured tools. Specifically, some 312 unique calling contexts map onto a much smaller number of behaviours. Luckily, Athena gives us a way to identify on a source code level which tool is being used, but such luxury is not necessarily always afforded to performance analysts in this situation, and it is therefore worthwhile to consider whether we can automatically classify call contexts based on the performance characteristics of the code. In this section, we describe a simple method of classification that provides adequate results for the Athena use case. This methodology that is designed to aid performance analysis in niche situations that fit the following criteria:

1. A single unit of code is called or used in many different parts of the source code.
2. The different contexts in which the aforementioned unit of code is called are presumed to form classes based on their input parameters, or the properties thereof, such that contexts in the same class exhibit similar performance behaviour.
3. Profiling on a class-by-class basis of the code aids the analysis of the performance of the code, for example when the number of calling contexts is so large that they cannot be individually examined.

At the core of our analysis methodology lies the idea that we can represent the performance characteristics of a unit of code in terms of structures that can be quantitatively compared and grouped. A prime candidate for such a structure is a feature vector in a high-dimensional vector space; such vectors are very well studied, and measures of similarity as well as clustering methods exist for vector spaces of arbitrary dimensionality. If we manage to meaningfully express the performance of calling contexts of a function in terms of their position in a vector space, we can determine how similar the performance of a function is between two contexts, and we can automatically group contexts into classes.

To express the performance characteristics of a function within a certain calling context in terms of a feature vector, we use call graph analysis information as generated by a profiling tool. We will assume that this information comes in the form of a set U of tuples (λ, N), where λ is a list describing the full call stack as it existed at a point of measurement, and N describes the number of times an event was observed with call stack λ. Figure 5.3 shows a simple example of profiling a toy program and representing the output in this format.



(a) A simple toy program, showing the call graph around target function `X`

λ	N
`A;X;`	5
`A;X;E;`	1
`A;X;E;G;`	4
`A;X;F;`	9
`B;C;X;`	3
`B;C;X;E;`	6
`B;C;X;E;G;`	1
`B;C;X;F;`	2

(b) An example of profiling output that the program in Figure a may produce. Here, λ represents a call chain, and N equals the number of events observed in that call chain.

Figure 5.3.: A simple example of how profiling data can be represented for a simple toy program.

Given a particular function target t, we find all of the aforementioned tuples (λ, N) where λ is of the form $p \frown t \frown q$, where $a \frown b$ is the concatenation of two segments of the call stack. That is to say, we find all tuples where the call stack consists of some prefix subsection of the stack p, followed by the target function t, followed by a suffix q. We collect all encountered values of p into an ordered collection P, and we collect all values encountered for q into another ordered collection Q. The ordering of these collections does not matter, as long as it is consistent throughout the process. For the example given in Figure 5.3, we find $P = \{A, B \frown C\}$ and $Q = \{\emptyset, E, F, E \frown G\}$.

The construction of the collection P gives us all (observed) contexts in which the function t is called, and the collection Q represents all the observed callee stacks of t. We construct a $|P| \times |Q|$ matrix V in which element $V_{i,j}$ is equal to the total event count for suffix Q_j in the context of prefix P_i:

$$V_{i,j}(U;t) = \sum_{(p \frown t \frown q,\ N)\ \in\ U\ :\ P_i = p,\ Q_j = q} N$$

We then normalise our matrix V row-wise, such that the each row vector of our normalised matrix lies on a $|Q|$-dimensional unit hyper-sphere. This normalisation step is necessary to ensure that two contexts with the same performance behaviour occupy a similar position in the performance signature vector space even if one context more frequently calls the target function than the other. Indeed, when we ignore

measurement errors, if two contexts have similar performance behaviour, but one represents s times more events than the other, this equates to the vector representation of the first context being equal to the vector representation of the second vector scaled by scalar s. Normalising each row of the matrix V means that the performance characteristics become invariant under the usage frequency of the code, which is a highly desirable property for this type of analysis. Notably, this normalisation does not necessarily negate the effect of differing input sizes. Continuing our example from Figure 5.3, we obtain the following matrix:

$$\begin{bmatrix} 0.451 & 0.090 & 0.812 & 0.361 \\ 0.424 & 0.849 & 0.283 & 0.141 \end{bmatrix}$$

This concludes the construction of our row-vector matrix in which each row represents the performance characteristics of a specific function within a specific context as an element of a vector space. We will refer to each such row as a *performance signature*. When applying this methodology to profiling data obtained from the Athena code, the resulting matrix has size 312×1855. In other words, we have 312 performance signatures, each of which is a vector in 1855 dimensions. When working with information represented as elements of vector spaces, one has a vast array of statistical tools at their disposal for analysing that data. In the remainder of this section, we will explore how the application of three of such methods to performance signatures can enrich our performance analysis and help solve niche profiling problems.

The first, and perhaps the simplest way in which we can use performance signatures is to determine the similarity between the performance behaviours of a unit of code in two different contexts. We theorise that two contexts with similar performance signatures will react similarly to changes in input, and to optimisations in the code itself. A common measure of similarity is that of cosine similarity, and we find it to be well suited for an analysis of this kind. The cosine similarity between two vectors is defined as the cosine of the angle between the two vectors, and a higher cosine similarity indicates that two vectors are more similar. Figure 5.4 shows a visual matrix of the cosine similarity between every pair of the 312 prefixes observed in our profiling of an Athena reconstruction job. The plot shows 'islands' of highly similar performance signatures, implying that there exist clusters of similar signatures in the original data set.

Secondly, we can visualise the performance characteristics of different contexts in a way that is more intuitively understood by humans. A performance signature vector can have an arbitrary number of dimensions, and this makes it difficult to visualise

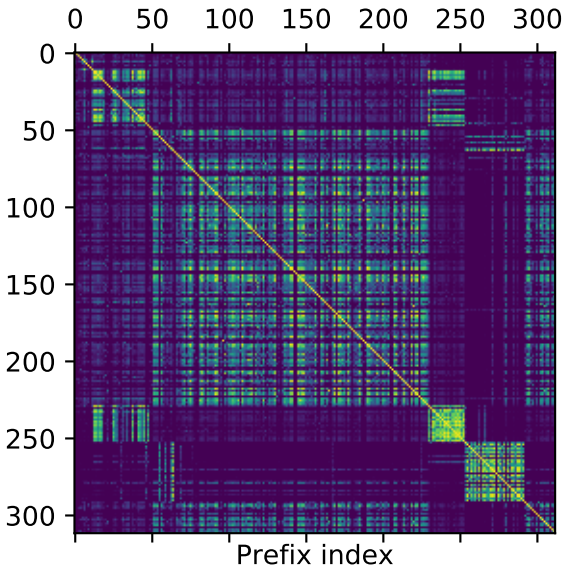
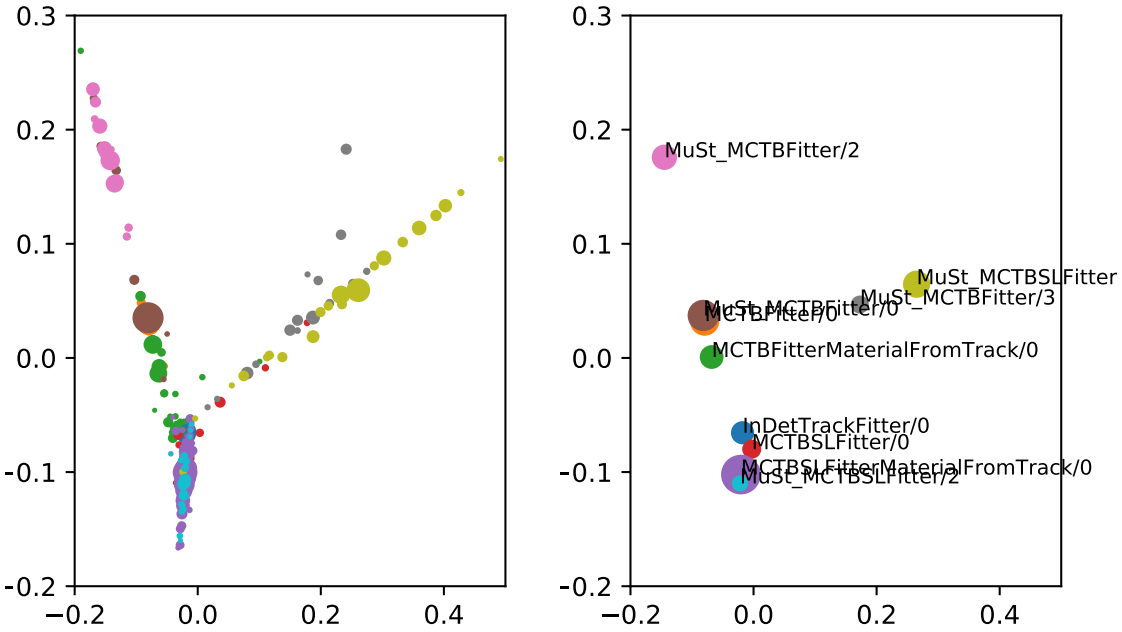


Figure 5.4.: Cosine similarity matrix for the performance signatures of 312 call stack prefixes as measured in an Athena reconstruction run. Brighter segments indicate more similar prefixes. This plot indicates shows that the data exhibits strong clustering tendencies.

them directly. However, we can apply dimensionality reduction techniques to such data to reduce its dimensionality from several thousands to two or three while maintaining much of the structure of the data. In this work, we will apply a *principal component analysis* methodology as described by Wold, Esbensen, and Geladi [35]. In a principal component analysis, m-dimensional data is mapped onto a n-dimensional vector space (where n is smaller than m) defined by the n eigenvectors of the dataset with the largest corresponding eigenvalues. Since a dimensionality reduction technique aims to preserve the structure of the original data as much as possible, we assume that signatures which are similar in the full dimensionality dataset will also lie close together in the lower dimensionality visualisation.

We have applied a principal component analysis to reduce the dimensionality of our 1855-dimensional Athena profiling data to two dimensions, resulting in Figure 5.5a. In two dimensions, the performance signatures seem to radiate outward from a central point in three distinct and roughly straight lines. The meaning of this pattern is unknown to us at this time, we consider it an interesting and unexpected result that is open for interpretation. We also notice strong clustering of signatures associated to the same Athena tool into four categories. We see this as verification that, in the particular



(a) Each of the 312 unique prefixes found profiling an Athena reconstruction job. Points are coloured according to the fitter name as reported by the Athena software.

(b) The same data, but with truncated prefixes such that tools map one-to-one unto prefixes. Essentially, this represents the central points of the different clusters shown in Figure 5.5a.

Figure 5.5.: Reductions of the 1855-dimensional performance signature vector space onto two dimensions using a principal component analysis technique. In both figures, points are coloured using additional information provided by Athena, such that points with similar colour represent the same tool. Colouring is consistent between the two plots.

case study of Athena, this technique can be applied to find similar performance behaviours even without additional annotation. It is interesting to note that the data represented here is the same as one would display in, say, a flame graph. The difference being that visualisation by flame graph provides more insight into the structure of each individual data point, while this PCA method has more more comparative power.

To briefly recap, we have empirical data on the performance characteristics of 312 different calling contexts for a particular fitting function. By representing the behaviour of each of these contexts as a feature vector, we can find similar vectors, and we can visually represent how different contexts relate to each other in terms of performance. By visualising the data in such a way, and by subsequently annotating this data

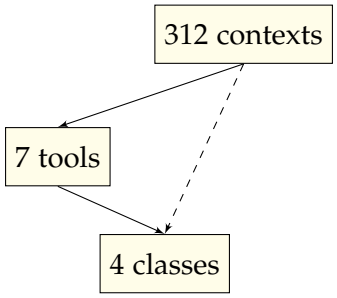


Figure 5.6.: Relationships between calling contexts, Athena tools, and performance classes. The dashed mapping is hidden, and the aim is to reconstruct it, even if the mappings between contexts and tools, and tools and classes are not available.

using additional information provided by the Athena software, we find that the 312 calling contexts map in some way onto four distinct performance classes. Figure 5.6 serves to show this relationship visually. The objective, now, is to find the mapping between calling contexts and performance classes automatically, without additional tool annotations.

If tool name annotation is unavailable, it may be possible to automatically reconstruct clusters of similarly behaving calling contexts through the use of clustering algorithms. Figure 5.7 shows an automatically generated clustering of the data. The clustering was generated using a mini batch K-means algorithm[2] configured to find four clusters. This number was found by manually examining the two-dimensional representation of the data, as mentioned before. If we compare the results of this clustering with this 'ground truth', we find that the automatic clustering performs relatively well; it achieves a Cohen's kappa $\kappa = 0.795$. This suggests that the methodology described in this section can be a useful tool to reconstruct mappings between calling contexts and classes of performance behaviour.

Throughout this section, we have expressed call stacks in terms of functions, but this need not necessarily be the case. If the required data is available, it would be possible to perform a similar analysis using, for example, branch record information; this may be able to provide a richer and more in-depth analysis, especially if individual functions in the analysed code are highly complex. Similarly, the definition of measurement events in terms of profiling is intentionally left vague. A natural fit would be to use CPU cycles here, but any measure that the profiling tool of choice supports may be used

[2]The reason for using a mini batch K-means algorithm as opposed to a regular K-means algorithm is technical in nature, as we experienced crashes trying to train a regular K-means algorithm.

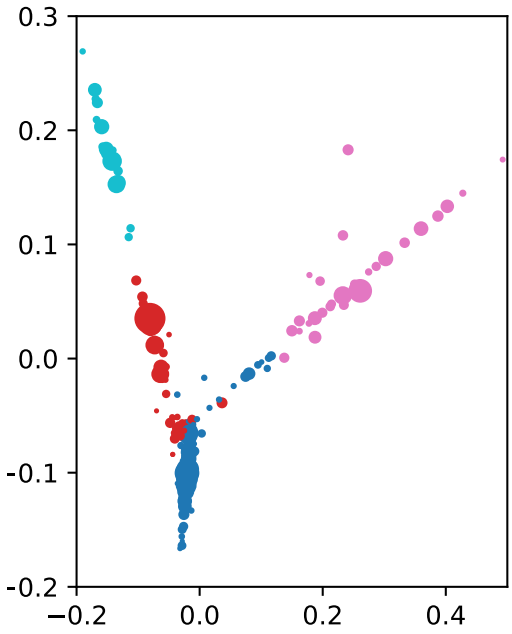


Figure 5.7.: Automatic clustering of the performance signature data. The clustering was performed with a weighted mini batch K-means algorithm (configured to find four clusters) on the data with full dimensionality, even though the visualisation is in two dimension.

here; it may be more valuable to perform such an analysis in terms of cache misses, for example. It may even be possible to widen the row-matrix such that each suffix corresponds not to a single column, but rather to multiple columns, each of which representing a different measure.

5.2.3. Filtering in Profiling Tools

By default, profiling tools like `perf` are not well suited to filtering data based on a large number on call chains. While `perf` allows us to filter call graph data by matching them to a regular expression, this method is tedious and impractical when the goal is to analyse a large collection of call chains. To solve this problem, we have opted to inject additional information into the call graph of the Athena software, such that profiling tools can use this additional information to disambiguate between performance classes and tools.

Since Athena is written mostly in C++, we can easily annotate the call graph using the templating system. By transforming the targeted fitting function into a templated function, we can create arbitrarily many versions of it. All versions of the function are functionally identical, and they differ only in name. The fitting code was augmented with trampoline code that calls one of several instances of the fitting code depending on which fitter tool is used. In other words, one of several functionally identical versions of the fitting function is chosen at run-time, and we can easily configure our profiling tools to display only the results pertaining to a particular set of template parameters. This allows us to distinguish the different Athena tools and profile them separately, even when the number of calling contexts is large.

5.2.4. Results

Figure 5.8 shows the distribution of CPU cycles across different parts of the tracking code. The data shown is taken over an entire Athena reconstruction job, and we do not disambiguate between tools here. We observe that the subdivision into the following seven categories covers the vast majority of cycles: the tracking code itself, the Eigen linear algebra library, track extrapolation, propagation, the detector description, the magnetic field service, and build-in mathematical functions. Segments of code not included in these seven categories, marked yellow, contribute only 7.62% of the total CPU time overhead.

We assume that both the Eigen linear algebra library functions as well as the mathematical functions included in the chosen C++ standard library are already highly

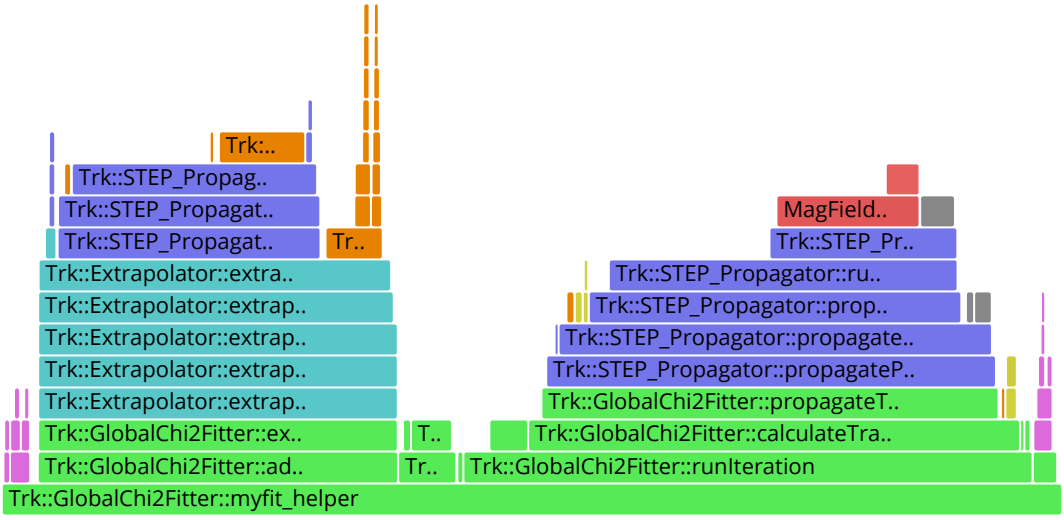


Figure 5.8.: Flame graph showing the CPU time composition of the `Trk::GlobalChi2Fitter` fitting class. Call graphs are truncated such that the call graph information prior to the fitter call is discarded. Green areas represent fitting code, purple areas represent Eigen linear algebra operations, light blue areas represent track extrapolation, dark blue areas represent propagation code, orange areas represent detector description code, red areas represent magnetic field code, grey areas represent built-in mathematical functions, and yellow areas represent all other parts of the code.

optimised, and we do not expect to be able to speed up these methods in any meaningful way. Of course, it is still possible to reduce the CPU time contribution of these functions by reducing the number of times they are called.

We observe that the overhead of the fitting code itself is comparatively minor. Rather, most of the CPU cycles are spent in the propagator and in the extrapolator. The extrapolator, in turn, contributes little overhead and has the propagator as one of its largest contributors of CPU cycles. Therefore, most of the overhead is introduced by the propagation code as well as some of its callees, the detector description code and the magnetic field service. The precise distribution of overhead is given in Table 5.2.

An interesting observation is that the matrix inversion or decomposition operations performed by the track fitter are only a small component of the total CPU time. The overhead of these is captured completely in the Eigen library category, which in itself only contributes 5.76% of the total overhead[3]. This is surprising, as it goes against the commonly accepted idea that the inversion and decomposition of matrices is the primary source of overhead in track fitting as described in Section 2.5. We conclude that such operations, while they may be a performance bottleneck in an idealised track fitting environment in which parameter propagation is free, are not a source of significant overhead in practice. Of course, previous optimisation has contributed to reducing the overhead of equation solving, and we expect that the overhead was larger in the past.

The top performance *hot spots* in the code are shown in Table 5.3. The most important observation from this data is that CPU time is distributed over a large number of functions, and very few functions contribute more than 10% of the total CPU time. Considering that the maximum speedup that can be achieved optimising the overhead of a function that contributes a fraction f of the total execution time is $\frac{1}{1-f}$, we find that the optimisation of a single function could only ever lead to a speedup of 1.18 times. Therefore, if we wish to achieve significant speedups, we will either need to focus our efforts on many different functions, or we will need to modify the calling relationships between functions to reduce the calling frequency of high overhead functions.

While Figure 5.8 provides some insight into the performance of the fitting code, it does nothing to disambiguate the performance of different fitting tools in the code. Using the methodology described in Section 5.2.3, we can produce flame graph for each tool individually. These graphs are shown in Figure 5.9. It is interesting to observe that tools which are positioned closely together in Figure 5.5b tend to have similar

[3]We have verified that the inversion and decomposition operations are explicitly visible in call graphs and that they are not inlined by the compiler.

Table 5.2.: Distribution of CPU time overhead (self-cost) across different segments of the code. This data should be considered a rough breakdown, as many functions are declared inline – should the compiler choose to follow these directions, the breakdown may be skewed. In that case, the magnetic field service and the Eigen library are likely to constitute a larger fraction of the total CPU time overhead.

Category	Overhead
Propagation	32.88%
Detector description	17.64%
Track fitting	14.47%
Magnetic field	14.01%
Built-in math	7.16%
Eigen	5.76%
Extrapolation	0.46%
Miscellaneous	7.62%

Table 5.3.: Distribution of CPU time overhead (self-cost) per function. Only the top ten functions are shown. Similar reservations to Table 5.2 apply. Since Eigen library functions are templated, there may exist many instances of the same kernel. For example, kernels operating on 3×3 row-major matrices may appear separately from kernels operating on 4×4 matrices, or 3×3 column-major matrices.

Function	Category	Overhead
`Trk::STEP_Propagator::rungeKuttaStep`	Propagation	15.39%
`Trk::STEP_Propagator::propagateWithJacobian`	Propagation	13.36%
`MagField::AtlasFieldSvc::getField`	Magnetic field	10.68%
`Trk::PlaneSurface::straightLineDistanceEst`	Detector des.	9.32%
`Trk::Volume::inside`	Detector des.	4.65%
`__ieee754_atan2_avx`	Built-in math	4.41%
`Trk::GlobalChi2Fitter::errors2`	Track fitting	3.81%
`Trk::GlobalChi2Fitter::calculateDerivatives`	Track fitting	3.65%
`BFieldCache::getB`	Magnetic field	3.31%
`Eigen::internal::gebp_kernel<...>`	*Eigen*	1.87%
Remainder		29.55%

flame graphs. For example, Figures 5.9b and 5.9f show very similar compositions, and lie very closely together in the two-dimensional performance signature space. Some counterexamples to this exist, such as Figures 5.9a and 5.9d, which are rather dissimilar despite being close together in the signature vector space. We believe that this may be due to a very low number of samples for the `InDetTrackFitter`.

Performance profiling can be done on far more granular levels than we have demonstrated here. Examination of the caching behaviour of code, or profiling on instruction level, for example, can be immensely powerful. However, the purpose of this chapter is to provide an insight into how the code behaves on a macroscopic scale, and fine-grained analysis does not effect such insights. Rather, we will apply lower level profiling techniques where appropriate in Chapter 6 where appropriate.

5.3. Modelling

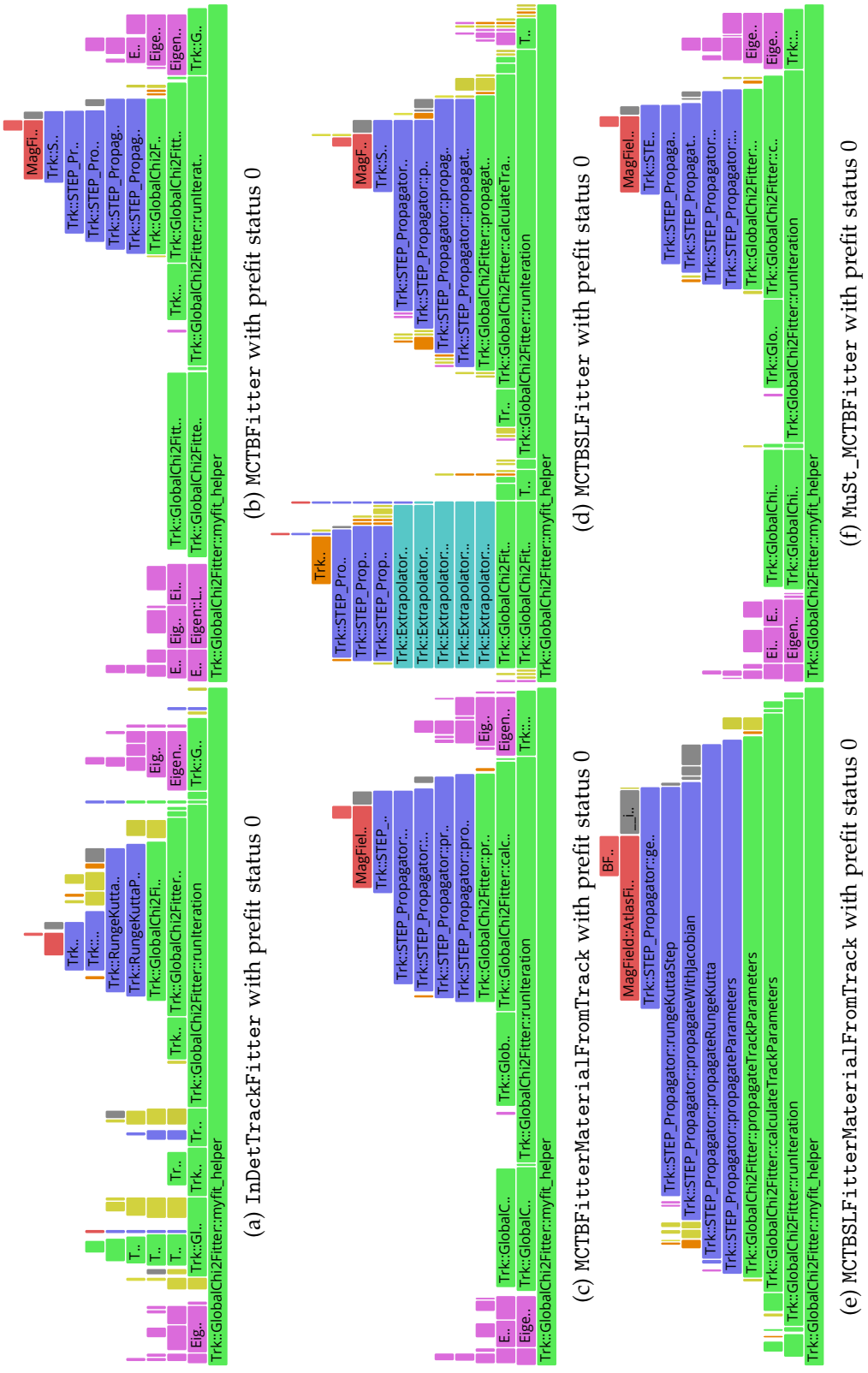
It is often useful to create a model for the performance of a piece of software in order to further ones understanding of its behaviour. The benefit of creating a performance model is two-fold: a model can be a useful tool to predict program performance (that is to say, it has predictive power), and it can act as a rough profiling tool, giving insight into the factors that impact the performance of the software (descriptive power).

Customarily, a performance model is a function that maps properties of the input of the software (for example, the size of the input, or the average degree of an input graph) to an estimated execution time. Ideally, a model is derived analytically as such models generally have high predictive and descriptive power. In cases where the construction of an analytical model is infeasible for one reason or another, statistical methods can be used to create models with high predictive power and diminished descriptive power. Recent developments in artificial intelligence methods have added many new tools to the statistical modelling body of knowledge, but such methods come at the cost of an even larger decrease in descriptive power.

If a performance model maps input parameters to expected run time, we must first determine which parameters exist in the Athena tracking software, and it is here that we encounter one of the consequences of the high complexity of the software: the number of input parameters that can impact the performance of the tracking code is very large. The `TrkGlobalChi2Fitter` tool has forty-four configuration parameters by itself, and depends on other tools such as extrapolators, navigators, and magnetic field services that have configurable parameters of their own. Then, there are properties of the input data itself that impact performance, some of which are known a priori

and some of which are not, such as the starting position and momentum of the track, and the number of detector hits along the track path. Finally, there are some variables which are also not known a priori such as the number of fitting iterations a track will require in order to reach the desired fit quality.

When considering the `TrkGlobalChi2Fitter` code, we believe that the code is too complex to model analytically; such a model would have an impractical number of parameters, and we cannot infer the values of some of these parameters analytically. Furthermore, we have attempted to create statistical models for the performance of the `TrkGlobalChi2Fitter` tool but we were unable to find any models with appreciable predictive power. We believe that this is due to the chaotic nature of the fitter's run time in relation to the fitter's input parameters. Indeed, extremely minor changes to the input of the fitter can result in vastly different performance. As an example, consider a particle travelling through the detector at some angle relative to the beam line. If we alter the angle of this particle by only a tiny amount, the particle may suddenly pass through many more material volumes, requiring many more propagation steps. For a statistical performance model to have significant predictive power, it must somehow capture the highly complex geometry of the whole ATLAS experiment, as well as some of the rules of particle physics. We believe that the creation of such a model in not not feasible within the scope of this work.



(a) InDetTrackFitter with prefit status 0

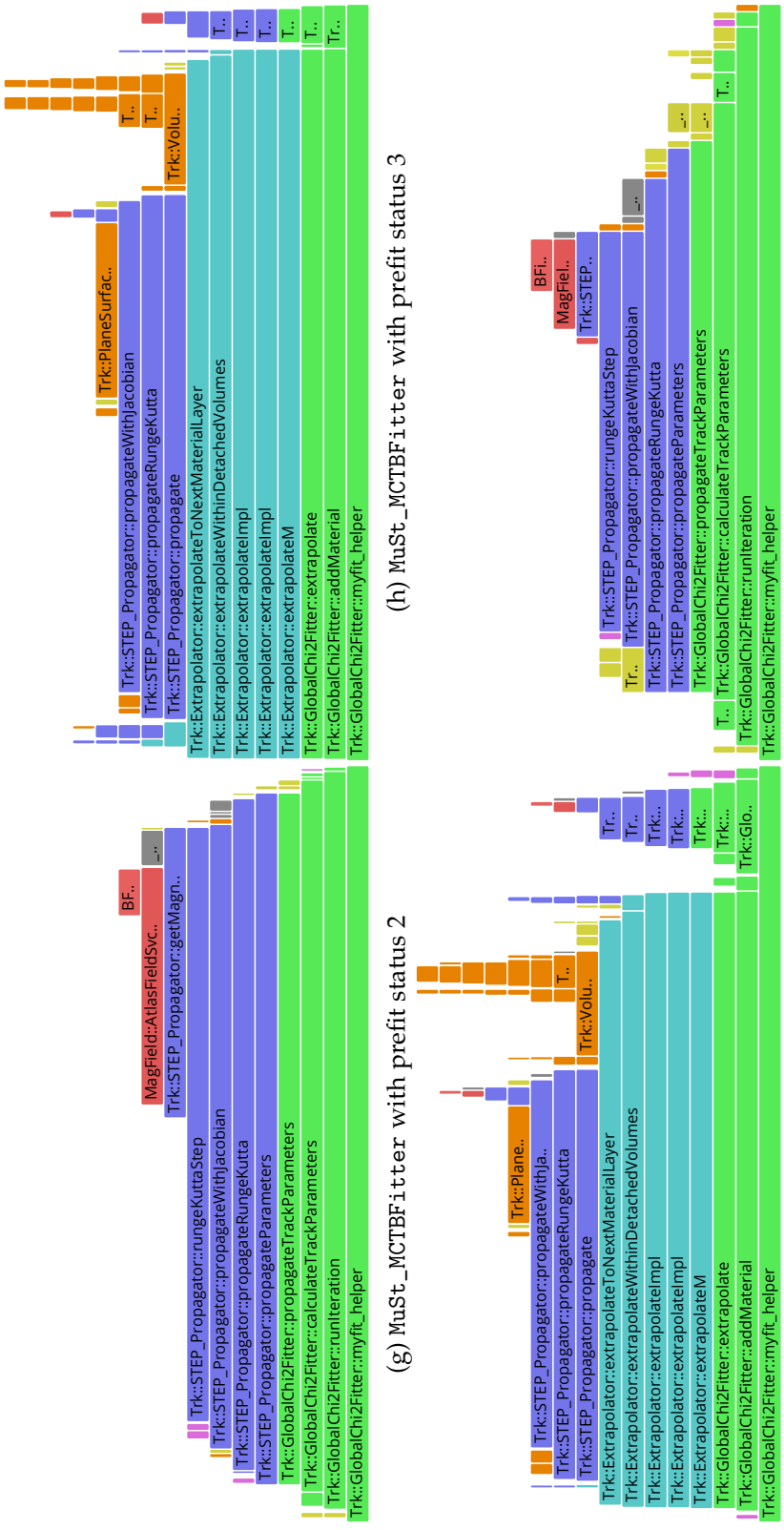
(b) MCTBFitter with prefit status 0

(c) MCTBFitterMaterialFromTrack with prefit status 0

(d) MCTBSLFitter with prefit status 0

(e) MCTBSLFitterMaterialFromTrack with prefit status 0

(f) MuSt_MCTBFitter with prefit status 0



(g) `MuSt_MCTBFitter` with prefit status 2

(h) `MuSt_MCTBFitter` with prefit status 3

(i) `MuSt_MCTBSLFitter` with prefit status 0

(j) `MuSt_MCTBSLFitter` with prefit status 2

Figure 5.9.: Flamegraphs of the unique tools and prefit settings profiled in an Athena reconstruction job. Call graphs are truncated such that the fitting code acts as the root. Colouring is consistent with Figure 5.8.

6. Performance Optimisation

We conclude from the previous chapter that a significant amount of the total CPU time spent on track fitting is spent not on the fitting code itself but rather on the other tools that it calls. It would therefore be ineffective to focus our efforts solely on the fitting code, and we must expand our efforts to other tools and services to have any hope of achieving significant speedups. Recalling that the majority of CPU cycles is spent in the magnetic field service, the detector description, and the propagator code, we focus our efforts on these three areas as well as the fitting code itself.

In this section, we will first examine the possible methodologies that we can apply to optimise the Athena track fitting code, addressing advantages and possible challenges along the way. In some cases, we will also provide a more in-depth explanation of how a particular transformation works. Then, we will detail how we have applied these methodologies to the aforementioned parts of the Athena code base. Finally, we will report on the total performance gain that was achieved by applying these optimisations.

6.1. Methods

The number of different optimising transformations that can be applied to code is vast; modern optimising compilers have hundreds of configuration flags to enable and disable all kinds of optimisations, and those cover only optimisations that compilers can perform automatically. One possible way of arranging performance optimisation techniques is to do so along a spectrum ranging from *high level* techniques to *low level* techniques. In this arrangement, high level optimisations are those that require mainly domain knowledge (in this case, knowledge of particle physics), such as changes to the overall design of the software, or the selection of different, more efficient algorithms. Low level optimisations, on the other hand, rely more on a technical knowledge of computer architectures, and include techniques such as peephole optimisation and the application of data-level parallelism.

Given our very limited knowledge of particle physics, it is immediately apparent that is will be difficult for us to make very high level changes to the code base; if niche,

more efficient designs or algorithms were to exist, what hope would we have to know about them approaching the problem as computer scientists? Similarly, applying low level optimisations has several problems as well. Firstly, many low level optimisations can be done automatically be compilers these days, and optimising compilers are only becoming more advanced. Secondly, many low level optimisations are platform specific, and we have seen in Section 3.4 that Athena has strong portability requirements.

6.1.1. Task-level Parallelism

A common, and potentially very powerful optimisation is task-level parallelism, usually in the form of multi-threading. Task-level parallelism occupies an interesting place in our spectrum of optimisations; it is usually possible to introduce it to code without having a thorough understanding of the scientific domain of the code, but it is at the same time too complex for an optimising compiler to consistently perform automatically. This means it would be an excellent candidate for our thesis, were it not for the fact that task-level parallelism is already being implemented on a batch level in the form of AthenaMT (see Section 3.3). Enabling AthenaMT would ideally saturate the hardware threads of the machine, making it inefficient to introduce an additional layer of thread-level parallelism in our fitting code.

6.1.2. Data-level Parallelism

Parallelism is a profoundly important concept for the performance of modern processors, and it extends much further than just thread-level parallelism. One other example of parallelism is data-level parallelism in which the same operation is executed on multiple pieces of data in parallel (also referred to as *SIMD*, for *single instruction multiple data*). For example, a programmer may want to add a scalar number to each element of a data vector, or perform a pair-wise multiplication of two data vectors. Such use cases are rather common in many kinds of computing, including the processing of multimedia data and scientific computing.

The key advantage of data-level parallelism becomes apparent when we consider that modern floating point units can perform operations on vectors in less time than it would take to perform the same operation on each element individually in a scalar manner. The Zen architecture used for the benchmarks in this thesis can perform arithmetic on vectors of eight single-precision floating point numbers with the same

latency as performing arithmetic on scalar single-precision floating point numbers[1]. In fact, there is no dedicated scalar floating point circuitry at all in this architecture, and floating point operations on scalars are merely specialised cases of vector operations. Recent Intel processors that support the AVX-512 instruction set extension are capable of operating on vectors of width sixteen.

As mentioned previously, scientific computing is a field that is often well suited for SIMD architectures. In the case of track fitting, we use a lot of linear algebra operations which are intrinsically data-parallel. However, the C++ programming language does not provide native support for vector arithmetic, and the common BLAS standard for linear algebra is too esoteric for many developers. This has lead to a wild growth of scalar implementations of algorithms that are well suited to vectorisation.

When we discard the intrinsic data-parallel properties of algorithms, we pay a great price in terms of both performance and maintainability. We lose performance because we lose the ability to use SIMD instruction set extensions as detailed previously, and we lose maintainability because the code we write is further removed from its abstract mathematical definition. In recent years, optimising compilers have evolved to attempt to reconstruct the intrinsic data-parallel properties of scalar code, but automatic vectorisation is far from perfect; a single conditional branch inside a loop can prevent a compiler from emitting optimal code. While compilers are still improving in this regard, we believe that we are still far away from a compiler that can perfectly infer the higher level properties from scalar code alone.

Converting scalar code to vector code can be difficult; not only does the programmer need to have sufficient knowledge of the algorithms to rewrite them in a more abstract fashion, they also need to have the technical skill to ensure that the right instructions will be emitted while keeping the code portable. The latter requirement is often met by using external libraries that are developed specifically for performance and portability. The recommended library for this in Athena is Eigen, which is an easy to use template library that interfaces with either its internal BLAS implementation or a drop-in external one. The Eigen library handles portability issues by selecting kernel implementations that are compatible with the targeted microarchitecture, and it contains an internal optimisation engine to find the most performant algorithms.

[1] It is worth noting that the Zen architecture performs 256-bit operations by employing two 128-bit floating point execution ports at the same time. Therefore, vector operations exert a higher port pressure than scalar operations, and the throughput is slightly higher for vector operations than it is for scalar operations. Processors with dedicated 256-bit vector circuitry do not suffer from this.

6.1.3. Memory Access Patterns

Nearly every computer program or algorithm interacts with the machine's memory in one way or another, and it is critical for a computer's memory to be both performant and sufficiently large. Over the years, computer memory systems have grown more and more complex, seeking a balance between capacity, performance and cost. Of particular importance for this thesis are memory hierarchies, which are nearly ubiquitous nowadays. At the core of this idea lies the fact that the cost of memory scales proportionally with both its capacity and speed of access, such that it is impractical to construct a computer with memory that is both very fast and very large. Instead, modern computers contain several different memories ranging from very fast but very small, such the processor registers, to very large but very slow, such as magnetic disks or even tapes. We will assume for now that the data required to run our fitting algorithm has already been retrieved from permanent storage, meaning that the slowest part of the hierarchy is the computer's main memory.

Between the computer's main memory and the processor's registers lie several intermediate memories, commonly referred to as caches. It is rather common, but by no means universal, for a processor to have three layers of cache (L1, L2, and L3, where L1 is the smallest, and L3 is the largest), and this is also true for the AMD EPYC 7551P processor used for the analysis in this work. The purpose of cache memories is to exploit the tendency of memory access patters to have some form of *locality*. For example, when memory is accessed, it is likely that that same memory will be accessed again in the near future (temporal locality), and it is also likely that nearby addresses will be accessed in the future (spatial locality). Thus, whenever a modern processor accesses an address in the computer's main memory, the data at that address, as well as some of the surrounding addresses, are copied into the processor's caches for faster access in the future. It is even possible for the processor to fetch data into its caches before it is explicitly asked to do so through a mechanism called prefetching; a prefetching processor will use models to predict which addresses are likely to be accessed in the near future and it will preemptively retrieve them from main memory.

Hardware caching is, in some ways, disconnected from software development; there is no need for the developer to indicate on a source-code level which data to cache and which data to flush from the caches. While it is possible to instruct the processor to manipulate the caches in certain ways, the vast majority of caching is handled by the processor itself on a hardware level. This does not mean that we should forget about memory access patterns completely, however. Indeed, it is not difficult at all to

write code that is *cache hostile*, meaning that the performance benefit of having caches is severely diminished. When writing code without hardware caching in mind, we may even compromise the performance of our caches unintentionally. For example, we might write code that accesses addresses in memory in a seemingly random order, reducing the locality of our algorithms, even when this is not necessary.

We observe, through a reading of the relevant source code, many instances of potential memory access pattern problems in Athena. Most commonly, such problems arise from excessive heap allocation; when allocating memory on the heap, the memory allocator has no knowledge of how it will be accessed. Separately heap allocating many smaller objects (such vectors or small matrices) that may need to be accessed sequentially can lead to unnecessarily random access patterns on that data. When attempting to improve the performance of a computer program, we should pay attention to its memory access patterns, and specifically to replace cache hostile code by equivalent code that is more cache friendly.

6.2. Application

In this section, we will briefly list some of the transformations made to the code of the track fitting, the magnetic field service, the detector description, and the propagation algorithms. Since the number of changes is quite large, and many of them are conceptually similar, we will only examine in depth a few examples.

6.2.1. Track Fitting

Within the track fitting code, we find many naive implementations of widely used algorithms such as matrix multiplication. In order to more effectively make use of data parallelism and memory locality, these parts of the code were removed and replaced by calls to library methods. Potentially inefficient data structures were also replaced by contiguous ones to improve performance. In addition to optimisation, we have also made an attempt to refactor the fitting code, focusing on deduplication and factoring very large methods into smaller ones. The goal of this effort has been to improve the code quality, and to thus make it easier to make modifications to the code, both for ourselves as well as for anyone who may wish to make modifications to the code in the future. However, it must be stated that the code quality of the track fitter is still very low even after refactoring, and there is a lot of work still to be done in this regard. Since refactoring is not a primary goal of our project, we did not pursue this further.

6.2.2. STEP Propagator

We recall the idea that particle tracks are constructed from individual hits on active detector elements from Section 2.3. To this end, wee need methods to interpolate the path that a particle took to travel between two detector elements. As particles travel through the ATLAS experiment, predictable but highly variable forces continually impact their direction of travel; principally, the magnetic field alters the direction of travel of charged particles as they move through space through a mechanism called the Lorentz force. The equation of motion for a particle of charge q with momentum p through a magnetic field $\vec{B}$ is defined as follows:

$$\frac{d^2\vec{r}}{dt^2} = \frac{q}{p}\left(\frac{d\vec{r}}{dt} \times \vec{B}(\vec{r})\right)$$

If the magnetic field is homogeneous, solving this equation is not so difficult, but since the magnetic field can vary significantly in strength and direction, it is generally impossible to analytically calculate the trajectory of a particle, and numerical methods must be used. We refer to such methods as propagation methods; two such methods, the STEP propagator and the Runge Kutta propagator, are commonly used in muon reconstruction, and we will discuss them in this chapter, as well as the next.

STEP is an acronym for *simultaneous track and error propagation*, meaning that this algorithm propagates not only the position of a particle over time, but also the error in that position. Since the STEP propagator propagates error while also propagating the track position, this saves an error propagation step at the end of the process. Numerically, the STEP propagator is similar to the Runge Kutta propagator, and both propagators implement a fourth-order Runge-Kutta-Nystrøm method [29].

As with the fitting code itself, we make heavy use of data-parallel techniques to improve the performance of the STEP propagation code. The core compute kernels of the STEP code make extensive use of floating point arithmetic, and they are as such excellent candidates for vectorisation. Excluding the magnetic field service queries, a single full-field Runge-Kutte-Nystrøm step receives as input one 3×3 matrix, one 6×7 matrix, and two vectors of length 3, for a total of 57 floating point values. We estimate that the number of floating point operations performed is roughly four hundred, and as such it depends somewhat on the underlying CPU architecture whether this code is compute-bound or memory-bound.

Superfluous Full-Field Propagation

An interesting observation follows from the individual profiling of fitting tools as described in Chapter 5. Particularly, the separate analysis of the `MCTBSLFitter` and `MCTBSLFitterMaterialFromTrack` reveals the existence of a significant performance-related bug. Indeed, the `SL` component in the name of these fitters indicates that they are designed to perform so-called *straight-line* propagation. In straight-line propagation, the magnetic field is disregarded because it is deemed unnecessary; straight-line propagation can be used in areas of the detector where the magnetic field is weaker, or for rough, lower precision fits where the quality of the parameters is not so relevant – in general, each final fit is preceded by hundreds of coarser fits, many of which may be straight-line fits.

There is a discrepancy between these fitters having the aforementioned straight-line fitting property and their measured performance characteristics, however, as there is a considerable overhead contribution from the magnetic field service in both cases; in a straight-line propagation tool, we would not expect to see this contribution. Investigation of the propagation source code reveals the source of this behaviour. The propagation interface (which all propagation tools must implement) specifies that most propagation methods accept a type `Trk::MagneticFieldMode` which enumerates four options: `FullField` to perform propagation with the full magnetic field, `ConstantField` to use a constant magnetic field, `FastField` to use a two-dimensional field map, and `NoField` to ignore the magnetic field entirely. The critical point is that the implementation of `Trk::STEP_Propagator` ignores this parameter completely and always performs a full-field propagation.

It is impossible for us to determine for certain how this performance bug was ever introduced to Athena, but we believe it to be a consequence of the organically grown nature of the code base over more than a decade. Older works by Salzburger [30, 29] reveal that the Athena code base used to contain a dedicated `Trk::StraightLinePropagator` class to perform straight-line propagation. However, this class is no longer available in the current code base of Athena; presumably it was removed, and the straight-line propagation mechanism was transferred to the other propagators. The STEP propagator appears to have been forgotten in this transition, as it did not receive any straight-line propagation functionality – the Runge Kutta propagator did get updated. It is unknown to us when this change was made, since the current version control system history does not go back far enough to determine this.

The implementation of straight-line propagation in the STEP code not only eliminates

the overhead of repeatedly calling the magnetic field service, it also eliminates the need to apply computationally expensive Runge Kutta pairs. In straight-line propagation, a single step of the Euler method is sufficient to propagate over any distance. Therefore, we also affect significant overhead savings in terms of the propagation code itself.

The discovery of this performance bug highlights the richness of performance analysis in which we profile each tool separately, and we see it as validation that this is a worthwhile endeavour in a project such as Athena, even if it necessitates a much more complicated analysis process. Indeed, had we profiled only on a source-code level, the magnetic field overhead would have blended in with the magnetic field queries of non-straight-line propagators, and the bug would have remained hidden.

6.2.3. Runge-Kutta Propagator

The Runge Kutta propagator (`Trk::RungeKuttaPropagator`) is another tool implementing the propagation interface. It is mathematically similar to the STEP propagator (with the exception that it does not propagate error), but the code was developed independently and is therefore vastly different. Much like the STEP propagator, it suffers from maintainability issues, due to the scalar nature of its programming. While the Runge Kutta propagator is not used by the tracking software in our job configuration, it is possible to configure Athena to use this fitter, and therefore there is still value in examining and optimising the performance of this class. Our work on the Runge-Kutta Propagator relies heavily on vectorisation. A move towards vertical vectorisation allows us to simultaneously increase the performance of the code while also improving its readability. Large swathes of scalar code were replaced by more semantically meaningful vector code, reducing the code volume by a factor three or more. A migration of the Runge-Kutta propagator code was performed for the two most important computation kernels in its code base, `rungeKuttaStep` and `rungeKuttaStepWithGradient`.

6.2.4. Detector Description

The detector description module provides classes that represent the material components of the ATLAS experiment. For example, the detector description provides classes for surfaces of many different shapes, and volumes such as cylinders, cubes, or even complex volumes formed from the union of two simpler volumes. Compared to the previously discussed components of the Athena software, which are complicated numerical methods, the detector description is composed of many much simpler functions. Although these functions are cheaper to execute, they are called far more often, and

therefore they still contribute significantly to total execution time. Since relatively little floating point arithmetic is involved, the angle of attack should be somewhat different than described before; instead of attempting to maximise floating point performance through the use of low-level architectural techniques, we may benefit much more from small improvements to branching behaviours or caching.

One interesting feature of the detector description code is that polymorphism is heavily exploited in this part of the code base. The abstract volume bound class is sub-classed, for example, for cylinders, trapezoids, cuboids, combined volumes, and more. Surfaces, likewise, can take the form of cylindrical surfaces, discs, ellipses, cones, and more. Most of these classes implement core functionality in different ways, meaning that we face many a large number of code units, and we will need to optimise them separately. As always, it is critical to first analyse the performance of these code units, such that we can prioritise optimisation of those that contribute most to the CPU time overhead.

Analysis

Profiling the detector description code is a difficult and rather inexact task, since there is a significant amount of function inlining present in the code base. This means that many functions can 'meld' together, and there is no practical way to separate them. An attempt was made to remove inlining keywords for profiling purposes, but this lead to so many linking errors that we deemed it impractical. Still, we can conclude from profiling data that the following two functions constitute the majority (85.68%) of the CPU time consumed by the detector description:

`Trk::PlaneSurface::straightLineDistanceEstimate` Used to calculate the distance between a plane surface and an object described by a position and a direction of travel. Returns the distance from that object to the surface both in absolute terms, as well as along the given direction of travel. For some surface types (notably curved ones), the object may have two intersection points with the surface, in which case the second one is also returned. Figure 6.2 visually shows the geometric meaning of this function.

`Trk::Volume::inside` Used to determine whether a given point lies within a volume. Returns a boolean.

Of these two, `Trk::PlaneSurface::straightLineDistanceEstimate` represent the smaller challenge in terms of profiling and optimisation; it is a member of the

Figure 6.1.: Flame graph of the detector description code.

`PlaneSurface` class, which is a concrete implementation of `Surface`, and as such there is no additional polymorphism present. This function has only simple getter-functions as callees, which are fully inlined. `Trk::Volume::inside`, on the other hand, has many possible implementations. Each `Trk::Volume` object has a member of type `Trk::VolumeBounds`, which describes the boundaries of the volume. The `Trk::Volume::inside` method mostly defers computation to the `Trk::VolumeBounds::inside` method, which is polymorphic. For example, determining whether a point in space is inside a cuboid volume and a cylindrical volume involves calling the `Trk::Volume::inside` for both, which will then call the `inside` methods of `Trk::CuboidVolumeBounds` and `Trk::CylinderVolumeBounds`, respectively. There is an additional intermediate stage present, as the program must determine at runtime which implementation of `inside` to use. Due to the inlining of the `inside` functions for all volume boundary classes, they are effectively hidden from profiling tools.

`Trk::Volume::inside`

The source code for `Trk::Volume::inside` is presented in its entirety in Listing 6.1. The code contains only five lines, but presents some opportunities for minor improvements; because the use of this method is so frequent, we believe it is worthwhile to examine these opportunities. In the source code for the `inside` method, we identify seven operations that may contribute to the execution time of this code:

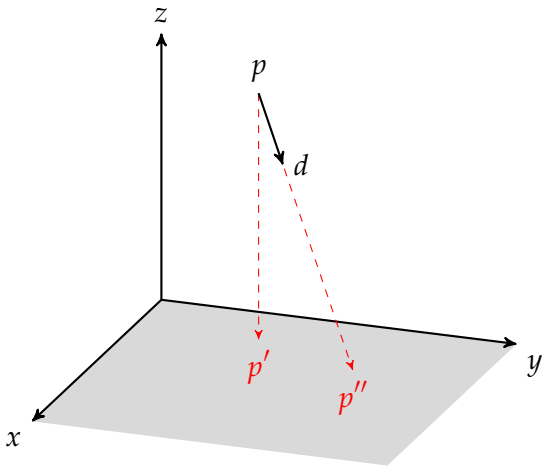


Figure 6.2.: Visualisation of the geometric meaning of `straightLineDistanceEstimate` for plane surfaces. A particle at position p travelling in direction d projects onto the surface at the xy plane (shaded grey) at p' and intersects it at p''. A distance solution contains the distances from p to p' and p''.

Listing 6.1: Source code for `Trk::Volume::inside`.

```
1 bool Trk::Volume::inside(const Amg::Vector3D& gp, double tol) const
2 {
3     if (!m_transform) {
4       return (volumeBounds()).inside(gp, tol);
5     }
6     Amg::Vector3D posInVolFrame((transform().inverse())*gp);
7     return (volumeBounds()).inside(posInVolFrame, tol);
8 }
```

1. The check whether the data member `m_transform` is a null pointer. The performance of this check is highly dependent on the branching behaviour of the code. Although modern CPUs have strong branch prediction capabilities, it can still be helpful to hint the compiler in certain directions if we are certain that one path will be more common than another. This allows the compiler to ensure that the commonly accepted branch lies within instruction cache boundaries and it can prevent pipeline flushes. Empirical evidence suggests that the branch in which a transformation matrix *is* present is vastly more common, and this is hinted to the compiler through the use of the `__builtin_expect` annotation.

2. The retrieval of the volume bounds through the call to `volumeBounds()`. This is an extremely simple inlined method that produces a reference from a shared pointer and returns it. We expect the overhead here to be negligible. Interestingly, the code uses a custom implementation of smart pointers in lieu of the smart pointers included in the C++ standard.

3. The retrieval of the transformation matrix through the call to `transform()`. Another trivial getter method, although it does contain another check on the `m_transform` member. Since this function is inlined, we expect the compiler to optimise this comparison away.

4. The inversion of the aforementioned transformation matrix. This is a possible source of significant overhead. Matrix inversion can be costly, even if it is only being performed on small 4×4 matrices. For this reason, we have chosen to cache the inverse transformation matrix for each volume object. We trade some some memory for a possibly significant reduction in execution time by doing so.

5. The matrix-vector multiplication of the aforementioned inverse matrix and the position vector. This is handled entirely by the Eigen library. As before, we assume that this library is sufficiently optimised that there is little we can do to directly improve it.

6. The storage the matrix resulting from the aforementioned matrix-vector multiplication into memory. This is a potentially unnecessary operation. Eigen is designed around lazy evaluation, and employs built-in cost models to determine whether the result of a computation should be materialised into a temporary or not. The implementation shows in Listing 6.1 forces early materialisation. A possibly more efficient solution would be to pass the result of the matrix-vector multiplication directly to the boundary's `inside` method. However, this change

Table 6.1.: Number of times the `inside` method is called for a 200 event reconstruction job, per volume boundary class.

Class	Count	Fraction
`PrismVolumeBounds`	2214821	32.44%
`SubtractedVolumeBounds`	1757115	25.73%
`CombinedVolumeBounds`	1549834	22.70%
`SimplePolygonBrepVolumeBounds`	680766	9.97%
`TrapezoidVolumeBounds`	278387	4.08%
`CuboidVolumeBounds`	173002	2.53%
`BevelledCylinderVolumeBounds`	110791	1.62%
`CylinderVolumeBounds`	62986	0.92%
`DoubleTrapezoidVolumeBounds`	105	0.00%
Total	6827807	100.00%

would require extensive modification of the volume boundary source code and was therefore omitted.

7. The check whether the resulting transformed position vector lies within the volume bounds. Small optimisations were implemented for most of the volume types. Most interesting of these relate to the branching volume types, and we will discuss these further in the remainder of this section.

In total, we observe nine distinct types of volume boundaries that are used in the fitting code. Some of these are used much more frequently than the others. The number of times the `inside` method is called for each class is given in Table 6.1. It is worth noting that the branching volumes, `CombinedVolumeBounds` and `SubtractedVolumeBounds`, are always dependent on at least two other volumes, and as such a call to their `inside` methods will always invoke at least two calls to that method in other classes. Additionally, given n branching volumes, there must be at minimum $n + 1$ non-branching volumes contained within them. Given the 3306949 branching volumes, we find that at least 3306950 out of 3520858 non-branching volumes are accessed indirectly through a branching volume.

The properties of branching volume boundaries lend themselves well to optimisation by predicate short-circuiting. To understand how, we must first further dissect the branching volume classes to examine how they operate. Indeed, branching volumes can have three separate geometric meanings; they can imply the union of two volumes (represented by `CombinedVolumeBounds`), they can represent the intersection of two

volumes (also represented by `CombinedVolumeBounds`, but with the `m_intersection` member enabled), or they can represent the relative complement of two volumes. We note the following equivalences for a point p inside a branching volume consisting of volumes A and B:

Union $p \in A \cup B \iff p \in A \vee p \in B$

Intersection $p \in A \cap B \iff p \in A \wedge p \in B$

Complement $p \in A \setminus B \iff p \in A \wedge \neg p \in B$

It is clear that each of these definitions allows short circuiting. If, for example, we wish to determine whether p is inside a union volume of A and B, it is sufficient to know that p lies in either of the two, and we do not need to examine the other. Short circuiting is currently implemented in the detector description code (be it only by virtue of short circuiting being a specification-defined feature of the `&&` and `||` operators), but the order in which the two dependent volumes are checked is always the same: left first, then right. We believe that it is possible to do better then this if we leverage the available knowledge of the underlying volumes.

Ideally, we would first evaluate the dependent volume with the lowest cost. For example, if a branching volume has another branching volume as its left child and a non-branching volume on the right side, it would be more efficient to evaluate the non-branching volume first. A branching volume with two non-branching children of different types may wish to first evaluate the child with the lower computational overhead. Finally, a branching volume with two branching children would benefit from computing first the child with the lowest depth.

Analysis of the actual behaviour of the branching volume nodes reveals some interesting behaviour. Firstly, we observe not a single intersection node; roughly 61% of branching nodes are of the union type, and the remaining 39% are of complement type. Some 51% of branching nodes have at least one branching child. Interestingly, the branching child is virtually always the left child, and only 0.04% of all branching nodes with at least one branching child have a branching right child; virtually all branching volume trees are unbalanced to the left. The deepest tree found in our experimentation had a total depth of 20. This lends some validity to the idea that it might be more efficient to evaluate the right child first. However, it must also be noted that only a small fraction of all `inside` calls for branching volumes return true: 0.56%. The remaining calls return false. For a union volume, this is the worst possible case since it requires the traversal of both children. In cases where union volumes returned a true value,

this could be determined from the right hand child 36.19% of the time. For relative complement nodes, the negative return value could be determined from the left hand child in 99.78% of cases, and it could be determined from the right hand child in 0.58% of cases; the optimisation is therefore largely non-viable for complement nodes, and the overhead of checking which node has a deeper tree would likely outweigh any benefits.

We conclude, therefore, that the proposed optimisation will not yield significant performance gains given the current state of the code. Should the frequency with which `inside` calls on branching volumes return true increase, it may be worthwhile to reconsider this. Given the large number of negative return values for branching volumes (as well as volumes in general), we suggest an optimisation through the use of bounding boxes. If we were to store a bounding box for each volume, we could eliminate many unnecessary calls. Indeed, if a point is not within the bounding box of a volume, is is impossible for that point to lie within the volume. Conversely, if a point lies within the bounding box, it *may* lie within the volume itself, and a more expensive complete computation must be performed to verify this. The main benefit from this approach comes from generating bounding boxes for branching volumes, as this may allow us to skip the traversal of the entire tree in a large number of cases.

To implement axis-aligned bounding boxes for each volume, we must first define bounding boxes for each of the seven non-branching volume types. This implies finding two vectors in the global coordinate space, $p_{min} = (x_1, y_1, z_1)$ and $p_{max} = (x_2, y_2, z_2)$, such that $x_1 \leq x' \leq x_2$, $y_1 \leq y' \leq y_2$, and $z_1 \leq z' \leq z_2$ for all points $p' = (x', y', z')$ in the volume. For polyhedral volume types (`Cuboid`, `DoubleTrapezoid`, `Prism`, `SimplePolygonBrep`, and `Trapezoid`) this is trivial; these volumes are defined by a set of space points, and the bounding box is defined simply by taking the minimum and maximum in each of the three dimensions amongst these points. For the two remaining non-polyhedral volume types, bounding box computation is somewhat more complicated, but the overhead is not necessarily relevant as this computation needs only be performed once. Given bounding box definitions for each of the seven non-branching volume types, it is trivial to construct bounding boxes for branching volumes, too. For unions, p_{min} can be defined as the pairwise minimum of the associated vectors in the children, and p_{max} can similarly be defined as the pairwise maximum. For intersections, p_{min} is the pairwise maximum and p_{max} is the pairwise minimum. For a relative complement volume $A \setminus B$, the bounding box is simply equal to the bounding

box for A^2.

Another benefit of being able to find a bounding box for every volume is that it makes it possible to construct a bounding volume hierarchy of all volumes in the detector description. As we have seen in Section 6.2.2, the tracking software incurs significant overhead retrieving the set of volumes near a given point. It does this inefficiently, retrieving far more volumes than strictly necessary, resulting in repeated iteration of irrelevant volumes. A bounding volume hierarchy could make it possible to efficiently, and more selectively return volumes near a given point, possibly resulting in significant performance gains. Due to time constraints, we were unable to implement a bounding box architecture for the tracking detector description. Still, we believe that this is a potentially valuable avenue of attack for future optimisation research in this area.

`Trk::PlaneSurface::straightLineDistanceEstimate`

The `straightLineDistanceEstimate` function differs somewhat from other functions we have discussed so far in that it is a simple and relatively unambiguous piece of code. Although different surface types have their own straight line distance estimate functions, the one belonging to `Trk::PlaneSurface` is the only one which contributes significant overhead. The source code for the function is given in Listing 6.2. In terms of operations that may impact the performance of the code, we identify the following: two getter methods (`normal` and `center`) which both retrieve a three-element vector, two dot products, one vector subtraction, one comparison, and one scalar division. As before, we rely on the compiler and the Eigen library to produce high-quality assembly code for the mathematical operations.

Rather, we will focus our efforts on the result sanitation (lines 11 through 17) and the getter methods on lines 6 and 8. The result sanitation attempts to catch possible errors when the value of `A` is exactly equal to zero, as division is performed on line 19 where `A` is the divisor. However, this check is both redundant and may degrade the performance of this function. The check is redundant since the likelihood of a double-precision dot product returning exactly zero is astronomically small, since the values that are represented by double precision numbers are so dense around the zero point. In our testing, we were unable to find a single example where the variable `A` took the value of exactly zero. Additionally, in the extremely unlikely event that the dot product would return exactly zero, this is not necessarily a problem. Perhaps counter-intuitively,

[2] It is possible to further reduce the volume of these bounding boxes, for example by using arbitrarily oriented minimum bounding boxes. However, the goal here is not necessarily the minimisation of the volume but rather composability and low overhead when determining whether a point lies inside it.

Listing 6.2: Source code for `Trk::PlaneSurface::straightLineDistanceEstimate`.

```
1  Trk::DistanceSolution
2  Trk::PlaneSurface::straightLineDistanceEstimate(const Amg::Vector3D& pos,
       const Amg::Vector3D& dir) const
3  {
4    static const double tol = 0.001;
5
6    const Amg::Vector3D& N = normal();
7
8    const double d = (pos - center()).dot(N);
9
10   const double A = dir.dot(N);
11   if (A == 0.) {
12     if (fabs(d) < tol) {
13       return Trk::DistanceSolution(1, 0., true, 0.);
14     } else {
15       return Trk::DistanceSolution(0, d, true, 0.);
16     }
17   }
18
19   return Trk::DistanceSolution(1, d, true, -d / A);
20 }
```

division by zero is well defined in the IEEE 754 standard; division of a non-zero real number by positive zero (the value that the code attempts to guard against) returns positive infinity.

The aforementioned section of code threatens the performance of the function as a whole in two ways, even if it is never executed. Firstly, it adds an additional comparison instruction which has an overhead in and of itself, and it may lead to pipeline flushes. Secondly, it increases the size of the resulting machine code, possibly making it harder for the processor to keep the code in its instruction cache. We have therefore chosen to remove the guarding code.

The two getter methods, `normal` and `center`, are also an important performance consideration. Indeed, these methods are not trivial getters, and they contain additional logic for cases where the members representing the normal and center vectors do not yet exist. In such cases, these vectors are first computed, cached, and returned. In practice, this means that at later points in a reconstruction job virtually all calls to this method simply return the cached vectors, and we would do well to optimise this common path. As in other parts of the Athena source code, the structure of such simple conditional statements are not optimal for a priori known most common paths. As before, we use compile-time hints to improve the structure of the generated assembly code to improve branching performance in ways that the runtime branch predictor

cannot.

In addition, we note that the cached vectors are allocated on the heap. This means that these vectors can be stored essentially anywhere on the heap, possibly far away from where the rest of the surface object is stored. To increase locality and reduce the overhead of memory access, we have opted to redesign the `Trk::Surface` class such that the normal and center vectors are embedded within the objects, rather than the objects storing a reference to vectors elsewhere in memory[3]. Whenever an object of the class is then accessed, the center and normal vectors are most likely already in the processor's cache, saving up to two memory accesses.

6.3. Computational Performance Improvement

Comparing the performance of the track fitting code before and after the application of the optimisations described in this chapter, we observe that the optimised version does indeed perform better than the non-optimised version of the code. The average total fitting time per job decreased from 435.0 ± 7.1 s to 219.7 ± 3.5 s, implying a speedup by factor 1.98. If we consider the mean execution time of individual fits, we find a decrease from 515.4 µs to 236.6 µs, giving a slightly higher speedup of 2.17. The source of this discrepancy is due to the diminished accuracy of intermediate fits, which is expected. To achieve final fits of the same quality, we find that we need to perform 10.0% more intermediate fits. Of course, the improvement to the performance of individual fits outweighs the increased number of fits.

In Chapter 5, it was found that many of the individual fitting tools behaved differently, and we hypothesised that they would respond differently to optimisation. We observe that this is, in fact, the case. Fitters which represented a more significant fraction of the total execution time before optimisation (Figure 5.2) were optimised more heavily, and the speedup for these fitting tools is, in general, higher than for others. Table 6.2 shows the performance after optimisation of each of the fitting tools. Table 6.3 shows the comparison between the original and the optimised versions of the code. The range in improvement between fitters is large. The highest reported increase in mean fitting time

[3] This redesign was a rather complicated process. Athena is comprised of modules such that one should, in theory, be able to change the code within one module while retaining full ABI compatibility with all others. However, this is not the case in practise; for whatever reason, the Athena source code includes many functions in header files and explicitly inlined functions. These functions can cross the boundaries between modules and end up being compiled in modules other than the one in which they were defined. When this happens, modifications in the origin module may cause incompatibilities with other modules under sparse compilation. We were required to vastly increase the number of compiled modules in order to be able to change the structure of the `Trk::Surface` class.

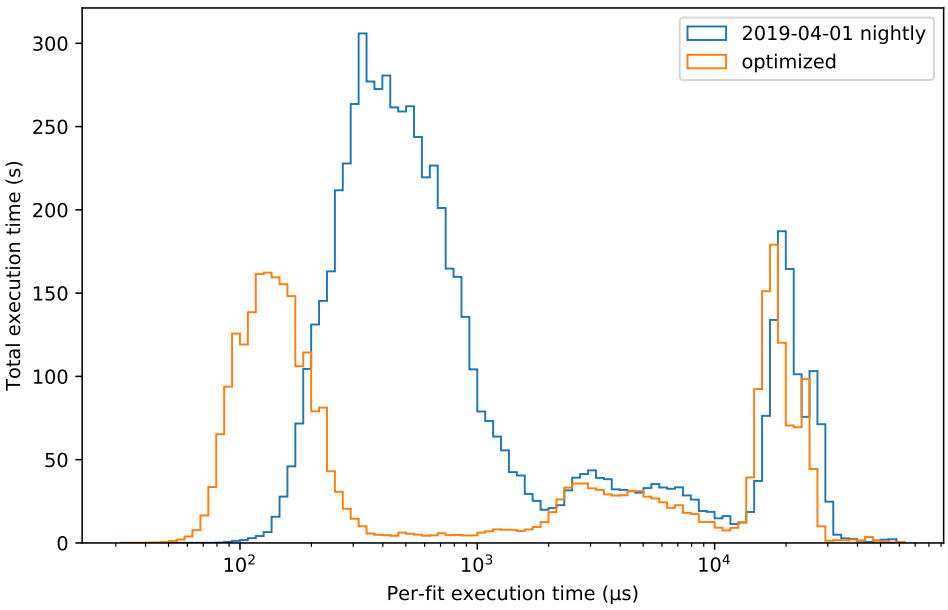


Figure 6.3.: Comparison in terms of performance between the original Athena track fitting code and the optimised version described in this thesis. An appreciable shift towards the left (lower fitting times) is observed across the entire range of per-fit execution times, but the effect is much stronger in some ranges than in others.

is a factor 2.98, although there are also fitter tools for which no speedup is observed at all, or for which we observe even a minor decrease in performance. We hypothesise that this is caused by some fitters reacting negatively to changes in code that provide speedups in other fitters. As long as the performance as a whole increases we do not consider this to be problematic, and the fitter in question is one of the smallest contributors to total fitting time. Figure 6.3 displays how the distribution of execution times is shifted before and after optimisation.

6.4. Physics Performance Validation

As mentioned in Section 4.7, it is of critical importance to verify that the physics performance of our optimised code is not worse than the original code. To this end, we have ran reconstruction jobs with the original version of the code, as well as our optimised version of the code. Identical configuration and input data were used in

Fitter name	Total	Per-fit	Fits
`MCTBFitter`	7.8 s	$3387 \pm 16\,441$ µs	2309
`MCTBFitterMaterialFromTrack`	10.1 s	5102.3 ± 3847.2 µs	1994
`MCTBSLFitter`	1.4 s	308.1 ± 1650.7 µs	4598
`MCTBSLFitterMaterialFromTrack`	119.7 s	132.8 ± 71.1 µs	901333
`MuSt_MCTBFitter`	68.7 s	7672.4 ± 7939.3 µs	8957
`MuSt_MCTBSLFitter`	9.8 s	1476.6 ± 2858.2 µs	6633
`InDetTrackFitter`	2.1 s	710.5 ± 358.4 µs	2978
Combined	219.7 s	236.6 ± 1163.8 µs	928802

Table 6.2.: Execution times of different fitting tools. Total fitting time and number of fits are reported for a single reconstruction job. For performance statistics before optimisation, see Table 5.1.

Fitter name	Total	Per-fit	Fits
`MCTBFitter`	1.170	1.170	1.000
`MCTBFitterMaterialFromTrack`	1.105	1.125	0.982
`MCTBSLFitter`	1.418	1.418	1.000
`MCTBSLFitterMaterialFromTrack`	2.698	2.977	0.906
`MuSt_MCTBFitter`	1.104	1.110	0.995
`MuSt_MCTBSLFitter`	1.205	1.199	1.005
`InDetTrackFitter`	0.984	0.984	1.000
Combined	1.980	2.179	0.909

Table 6.3.: Relative performance of the optimised version of the code compared to the base version, obtained by dividing the data in Table 5.1 by the data in Table 6.2.

both cases. Additionally, fifty reconstruction jobs were performed for each of the two versions of the code. The full comparison between the physics results for the original and the modified versions of Athena are displayed in Table 6.4. We observe a degradation in terms of reconstruction efficiency of no more than 0.12 p.p., and the fake rate has increased by no more than 0.001 tracks per event. Both of these values are well within the thresholds established in Section 4.7, and as such we do not believe the physics performance has been impacted in any significant way. In some cases, the physics performance *improved*, but not in a significant manner.

Table 6.4.: Comparison of the physics performance of a reconstruction job of 1000 di-muon events before and after optimisation. Values that differ are emboldened. Note that for some metrics a higher number is better, and for others a lower number is preferred.

	Track Collection							
	μ Spectrometer		Extrapolated μ		Combined μ		MS Only μ	
Metric	Old	New	Old	New	Old	New	Old	New
Truth tracks	1963.0	1963.0	1963.0	1963.0	1656.0	1656.0	1963.0	1963.0
Good tracks	1927.0	**1926.0**	1925.0	**1922.8**	1649.2	1649.2	1761.1	**1760.7**
Missed tracks								
Multi-station	30.0	**31.0**	32.0	**34.2**	5.8	5.8	194.9	**195.3**
Single-station	6.0	6.0	6.0	6.0	1.0	1.0	7.0	7.0
Fake tracks								
High p_T	160.4	**158.9**	185.2	**185.3**	127.0	**128.0**	1.0	**2.1**
Low p_T	15.0	**16.3**	1.9	**0.5**	0.0	0.0	0.0	0.0
Straight-line	57.0	**57.1**	4.0	**4.2**	0.0	0.0	0.0	0.0
Incomplete tracks								
Lost momentum	34.0	**33.8**	44.0	**42.8**	27.0	**26.1**	19.0	**18.0**
Missing precision	68.0	**67.5**	71.9	**70.4**	63.4	**62.3**	61.1	**60.2**
Wrong precision	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Missing trigger	13.0	13.0	13.0	13.0	11.0	11.0	11.0	11.0
Wrong trigger	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Missing chamber	173.0	173.0	174.0	**174.3**	131.0	**131.3**	154.0	**154.3**
Wrong chamber	3.0	3.0	3.0	3.0	2.0	2.0	3.0	3.0
Efficiency	0.9817	**0.9812**	0.9806	**0.9794**	0.9959	0.9959	0.8971	**0.8970**
Fake rate	0.2324	**0.2323**	0.1911	**0.1899**	0.1270	**0.1280**	0.0010	**0.0020**

7. Conclusion

The goal of this thesis has been to investigate if and how performance engineering methods can be applied to high energy physics software in order to prepare that software for the significantly higher data output of modern experiments such as the HL-LHC. We highlight in this chapter our main findings and contributions, and discuss future work opportunities.

7.1. Main findings

In particular, we have investigated the applicability of the performance engineering body of knowledge on the χ^2 minimisation track fitting algorithm in the ATLAS Athena software. We have investigated the applicability of analysis and optimisation, as well as the existing performance of the software, and we have used the results of that analysis to make informed and targeted changes to the source code with the aim of more effectively utilising the available computing hardware.

It has been mostly in the analysis phases of our project that the nature of the software has been a challenge. The use of analysis tools like profilers and benchmarking tools was hindered in many ways by the architecture, design, implementation, and even the build system of Athena. It was immediately noticeable that the quality of the Athena code base could be a significant hurdle in the optimisation process, and a study of software quality from a software engineering point of view has confirmed this. In other cases, challenges were technical in nature, such as the difficulty in recording call graph data. In other cases still, they required a more fundamental shift in the way we think about and apply tools, such as the need to disambiguate between many different performance patterns in identical code. Throughout this thesis, we have made an effort to describe these challenges, as well as the methods used to tackle them. In this process, we believe to have presented some novel, if highly niche, methodologies.

Through optimisation, we have increased the performance of the software, in the context of muon reconstruction, by roughly a factor two without impacting the quality of the output in any significant way. It must be noted that it is particularly difficult

to make definitive statements about the performance gain as a whole, however, as the input parameter space is immensely large; the performance gain from the work presented in this thesis might be lower of higher for other workloads. Although the achieved result is not sufficient to meet the factor nine requirement for HL-LHC, we hope that it will help to reach that goal by 2026. It is said that Rome was not built in one day, and the same goes for such a massive undertaking as preparing Athena for HL-LHC. Aside from improving the performance of the Athena software, we have provided an example of how performance engineering can be applied to high energy physics software; a case study, so to say, that we hope might help bridge the remaining performance gap.

If we consider the optimisations made to the software, we have not used any particularly cutting-edge techniques. Indeed, most of the code transformations are rather elementary in nature. The fact that a doubling of performance can be achieved in such performance-critical software through only simple optimisations is, in itself, a fascinating result. It indicates that many more such opportunities for simple optimisations may be present in the software. We suspect that these opportunities have not been capitalised upon simply because they are so hard to find in the Athena code base. With performance analysis being hindered by the architecture and quality of the software, even the simplest optimisations (*free lunch* optimisations) may become difficult to find in a six million line code base. If the ATLAS collaboration is to meet its computing performance requirements for HL-LHC, finding these opportunities will be critically important, and for that we will need tools to analyse the performance of Athena in meaningful ways.

7.2. Future Work and Challenges

A consistent red thread throughout this project has been how low code quality can negatively impact the maintenance of software. In our analysis of the software quality of Athena we concluded that the maintainability of the code is severely hindered by factors like high complexity, and large units of code. With the LHC scheduled to keep running for at least 18 more years, the cost in terms of development time of this code quality issue will only increase.

Many of the optimisations described in this thesis rely on vertical vectorisation. In many cases, vertical vectorisation was the only vectorisation technique that could be applied while preserving the general architecture of the software. Unfortunately, vertical vectorisation is not necessarily arbitrarily scalable. Indeed, much of the vector

arithmetic done in Athena uses three-element double-precision vectors. Since such vectors are 192 bits in size, they can benefit from the increase in vector size of AVX (which increases the width of vectors to 256 bits from 128 bits), but they are unlikely to perform better under AVX-512 or hypothetical future instruction set extensions with even wider vector registers. However, the delay at which even SSE4.2 is planned to be supported gives reason to believe that vector instruction set extensions with vector width greater than 256 will not be supported for a while[1].

[1] SSE4.2 was introduced with the Intel Nehalem architecture in 2008, and support for these instructions is not yet present in Athena as of 2019.

Bibliography

[1] Georges Aad et al. "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 1–29.

[2] Georges Aad et al. "The ATLAS experiment at the CERN large hadron collider". In: *Jinst* 3 (2008), S08003.

[3] Giorgio Apollinari et al. "High Luminosity Large Hadron Collider HL-LHC". In: *arXiv preprint arXiv:1705.08830* (2017).

[4] J. Apostolakis et al. "Final report of the ATLAS detector simulation performance assessment group". In: *CERN/PH/SFT, CERN-LCGAPP-2010-01* (2010).

[5] Michael Bray et al. *C4 Software Technology Reference Guide-A Prototype.* Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1997.

[6] Simone Campana and Torre Wenaus. *The ATLAS computing challenge for HL-LHC*. Tech. rep. ATL-COM-SOFT-2016-085, 2016.

[7] ATLAS Collaboration et al. *The ATLAS experiment at the CERN large hadron collider*. 2008.

[8] B. Curtis et al. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics". In: *IEEE Transactions on Software Engineering* SE-5.2 (Mar. 1979), pp. 96–104. DOI: 10.1109/tse.1979.234165.

[9] Al Danial. *cloc – Count Lines of Code*. 2006. URL: https://github.com/AlDanial/cloc.

[10] Tristan Arnoldus Du Pree. *Global Chi2 Fitter Status*. ATLAS Collaboration. Nov. 7, 2017. URL: https://indico.cern.ch/event/677168/contributions/2772208/ (visited on 10/14/2019).

[11] Martin Errenst. *Performance Optimization of the ATLAS Detector Simulation*. Tech. rep. 2016.

[12] *Expected Tracking Performance of the ATLAS Inner Tracker at the HL-LHC*. Tech. rep. ATL-PHYS-PUB-2019-014. Geneva: CERN, Mar. 2019. URL: https://cds.cern.ch/record/2669540.

[13] *gcc manual: x86 Options*. URL: https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/x86-Options.html.

[14] Heather Gray. "Track reconstruction in the ATLAS experiment". Internal ATLAS slide deck. Mar. 2016. URL: https://indico.cern.ch/event/504284/contributions/2023875/.

[15] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org/.

[16] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability". In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. IEEE, Sept. 2007. DOI: 10.1109/quatic.2007.8.

[17] *IEEE Standard Glossary of Software Engineering Terminology*. DOI: 10.1109/ieeestd.1990.101064.

[18] *Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2011.

[19] *Software engineering – Product quality – Part 1: Quality model*. Standard. Geneva, CH: International Organization for Standardization, June 2001.

[20] Graylin Jay et al. "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship." In: *JSEA* 2.3 (2009), pp. 137–143.

[21] Capers Jones. *Programming Languages Table*. 1996. URL: http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm.

[22] Matthias Kretz and Volker Lindenstruth. "Vc: A C++ library for explicit vectorization". In: *Software: Practice and Experience* 42.11 (Dec. 2011), pp. 1409–1430. DOI: 10.1002/spe.1149.

[23] Robert Langenberg. "Analysis and Optimization of the Offline Software of the ATLAS Experiment at CERN". Dissertation. Technische Universität München, 2017.

[24] Chris Lomont. "Introduction to Intel Advanced Vector Extensions". In: *Intel White Paper* (2011), pp. 1–21.

[25] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. DOI: 10.1109/tse.1976.233837.

[26] Yoichiro Nambu. "Particle physics in perspective". In: *Nuclear Physics A* 629.1-2 (Feb. 1998), pp. 3–8. DOI: 10.1016/s0375-9474(97)00659-3.

[27] Daniel Hugo Cámpora Pérez and Ben Couturier. "SIMD studies in the LHCb reconstruction software". In: *Journal of Physics: Conference Series*. Vol. 664. 9. IOP Publishing. 2015, p. 092004.

[28] A. Rimoldi et al. *Final report of the simulation optimization task force*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2008.

[29] A. Salzburger. *The ATLAS track extrapolation package*. Tech. rep. 2007.

[30] A. Salzburger. "The track extrapolation package in the new ATLAS tracking realm". In: (2005).

[31] Scott Snyder. "Old CPUs and new architectures". Internal ATLAS slide deck. Oct. 2018. URL: https://indico.cern.ch/event/646945/contributions/3153403/.

[32] Robert Sokolowski. "Matter, Elements and Substance in Aristotle". In: *Journal of the History of Philosophy* 8.3 (1970), pp. 263–288. DOI: 10.1353/hph.2008.1672.

[33] Garrick Staples. "TORQUE resource manager". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06*. ACM Press, 2006. DOI: 10.1145/1188455.1188464.

[34] Royer Ticse et al. *An SIMD parallel version of the VELO Pixel track reconstruction for the LHCb upgrade*. Tech. rep. 2013.

[35] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal component analysis". In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.

[36] Terry Yin. *lizard*. 2006. URL: https://github.com/terryyin/lizard.

A. Experiment Job Options

In Athena, jobs are configured by Python scripts referred to as *job options*. What follows are the job options used in the experiments conducted for this thesis. The file `pre.py` is included first, followed by an automatic configuration step in which Athena configures several other parameters. Finally, the file `post.py` is executed.

listings/pre.py

```
1  from MuonRecExample.MuonRecFlags import muonRecFlags, muonStandaloneFlags
2  from AthenaCommon.AthenaCommonFlags import athenaCommonFlags
3
4  muonRecFlags.doFastDigitization=False
5  muonRecFlags.useLooseErrorTuning.set_Value_and_Lock(True)
6  muonRecFlags.doCSCs=False
7  muonRecFlags.doNSWNewThirdChain=True
8  muonRecFlags.dosTGCs=True
9  muonRecFlags.doMicromegas=True
10
11 muonStandaloneFlags.printSummary=True
12 muonRecFlags.doTrackPerformance=True
13 muonRecFlags.TrackPerfSummaryLevel=2
14 muonRecFlags.TrackPerfDebugLevel=5
15
16 rec.doTrigger=False
17 rec.doInDet=True
18 rec.doCalo=True
19 rec.doLArg=True
20 rec.doEgamma=False
21 rec.doTile=True
22 rec.doLucid=False
23 rec.doZdc=False
24 rec.doJetMissingETTag=False
25
26 rec.doMuon=True
27
28 from MuonCombinedRecExample.MuonCombinedRecFlags import
       muonCombinedRecFlags
```

```
29
30 muonCombinedRecFlags.doCaloTrkMuId=True
31 muonCombinedRecFlags.doMuGirlLowBeta=False
32 muonCombinedRecFlags.doMuGirl=True
33 muonCombinedRecFlags.doCombinedFit=True
34 muonCombinedRecFlags.doMuonSegmentTagger=True
35
36 svcMgr.MessageSvc.infoLimit    = 100000000
37 svcMgr.MessageSvc.defaultLimit = 100000000
38 svcMgr.MessageSvc.debugLimit   = 100000000
39 svcMgr.MessageSvc.verboseLimit = 100000000
```

listings/post.py

```
1  from AthenaCommon.AlgSequence import AlgSequence
2  topSequence = AlgSequence()
3
4  if rec.doMuon:
5    ToolSvc.MuonStraightLineExtrapolator.Propagators = ['Trk::
       STEP_Propagator/MuonStraightLinePropagator']
6    ToolSvc.MuonStraightLineExtrapolator.STEP_Propagator = 'Trk::
       STEP_Propagator/MuonStraightLinePropagator'
7
8    ToolSvc.MCTBSLFitterMaterialFromTrack.ExtrapolationTool = 'Trk::
       Extrapolator/MuonStraightLineExtrapolator'
9    ToolSvc.MCTBSLFitterMaterialFromTrack.PropagatorTool = 'Trk::
       STEP_Propagator/MuonStraightLinePropagator'
10
11   ToolSvc.MuonTrackCleaner_seg.Extrapolator = 'Trk::Extrapolator/
       MuonStraightLineExtrapolator'
12
13   ToolSvc.MuonClusterSegmentFinderTool.SLFitter = 'Trk::GlobalChi2Fitter/
       MCTBSLFitterMaterialFromTrack'
14   ToolSvc.MuonTrackCleaner_seg.Fitter = 'Trk::GlobalChi2Fitter/
       MCTBSLFitterMaterialFromTrack'
15
16   ToolSvc.CombinedMuonTrackBuilder.MuonHoleRecovery = ""
17   ToolSvc.CombinedMuonTrackBuilderFit.MuonHoleRecovery = ""
18   ToolSvc.MuonTrackTruthTool.DoSummary=True
19   topSequence.MuonStandalonePerformanceAlg.DoTrackDebug = 6
20   topSequence.BeamBackgroundFiller.Enable = False
21
22   from AthenaCommon.AlgSequence import AlgSequence
23   job = AlgSequence()
24   from MuonPRDTest.MuonPRDTestConf import *
```

```
25
26    job+=NSWPRDValAlg('NSWPRDValAlg', OutputLevel = INFO)
27    NSWPRDValAlg.OutputLevel = INFO
28    NSWPRDValAlg.doTruth = True
29
30    NSWPRDValAlg.doMuEntry = True
31
32    NSWPRDValAlg.doMMHit = True
33    NSWPRDValAlg.doMMDigit = True
34    NSWPRDValAlg.doMMRDO = True
35    NSWPRDValAlg.doMMPRD = True
36    NSWPRDValAlg.doMMFastDigit = False
37
38    NSWPRDValAlg.doSTGCHit = True
39    NSWPRDValAlg.doSTGCDigit = True
40    NSWPRDValAlg.doSTGCRDO = True
41    NSWPRDValAlg.doSTGCPRD = True
42    NSWPRDValAlg.doSTGCFastDigit = False
43
44    from GaudiSvc.GaudiSvcConf import THistSvc
45    ServiceMgr += THistSvc()
46    ServiceMgr.THistSvc.Output = [ "NSWPRDValAlg DATAFILE='NSWPRDValAlg.
        full.ntuple.root' OPT='RECREATE'" ]
47
48  from AthenaCommon.ConfigurationShelve import saveToAscii
49  saveToAscii("configCB.txt")
```

B. Sparse Compilation Module Listing

The following is an integral copy of the sparse compilation module selection file that was used to build Athena in this thesis. Lines starting with - exclude modules while lines starting with +. Wildcards are used to match any module located in certain directories. Somewhat unintuitively, the build system uses only the first line mentioning a particular module; any subsequent attempts to include or exclude the module are ignored.

listings/sparse_compilation.txt

```
# Not sure why, but these two fail to build.
-Tracking/Acts/.*
-MuonSpectrometer/MuonCnv/MuonByteStream

+Tracking/.*
+MagneticField/.*

# Necessary for bug fixes.
+Reconstruction/MuonIdentification/MuonCombinedRecExample
+MuonSpectrometer/MuonValidation/MuonRecValidation/.*
+MuonSpectrometer/MuonTruthAlgs
+MuonSpectrometer/MuonReconstruction/MuonTrackMakers/.*

# Necessary for changes to Trk::Surface.
+Calorimeter/CaloTrackingGeometry
+Calorimeter/CaloTrackUtils
+Event/xAOD/.*
+InnerDetector/InDetCosmics/.*
+InnerDetector/InDetDetDescr/.*
+InnerDetector/InDetDigitization/.*
+InnerDetector/InDetEventCnv/.*
+InnerDetector/InDetMonitoring/.*
+InnerDetector/InDetRecAlgs/.*
+InnerDetector/InDetRecEvent/.*
+InnerDetector/InDetRecTools/.*
+InnerDetector/InDetTrigRecAlgs/.*
+InnerDetector/InDetValidation/.*
```

```
28 +LArCalorimeter/LArTrackingGeometry
29 +MuonSpectrometer/Amdcsimrec/.*
30 +MuonSpectrometer/MSVertexReconstruction/.*
31 +MuonSpectrometer/MuonAlignment/.*
32 +MuonSpectrometer/MuonCalib/.*
33 +MuonSpectrometer/MuonCnv/.*
34 +MuonSpectrometer/MuonDetDescr/.*
35 +MuonSpectrometer/MuonDigitization/.*
36 +MuonSpectrometer/MuonGeoModel
37 +MuonSpectrometer/MuonGeoModelTest
38 +MuonSpectrometer/MuonReconstruction/.*
39 +MuonSpectrometer/MuonValidation/.*
40 +Reconstruction/egamma/.*
41 +Reconstruction/iPat/.*
42 +Reconstruction/MuonIdentification/.*
43 +Reconstruction/RecoTools/.*
44 +Reconstruction/TrackCaloClusterRec/.*
45 +Reconstruction/TRT_Rec
46 +Reconstruction/VKalVrt/.*
47 +TileCalorimeter/TileTrackingGeometry
48 +Trigger/TrigAlgorithms/.*
49 +Trigger/TrigFTK/.*
50 +Trigger/TrigHypothesis/.*
51 +Trigger/TrigTools/.*
52
53 # Exclude everything else.
54 -.*
```