University of Wuppertal
School of Mathematics and Natural Science
Experimental Particle Physics

Master of Science
Computersimulation in Science

Performance Optimization of the ATLAS Detector Simulation

Master Thesis

Martin Errenst
1010100

Wuppertal, 05.09.2016

First corrector: Prof. Dr. Christian Zeitnitz
Second corrector: Dr. Torsten Harenberg

Abstract

In the thesis at hand the current performance of the ATLAS detector simulation, part of the *Athena* framework, is analyzed and possible optimizations are examined. For this purpose the event based sampling profiler *VTune Amplifier* by Intel is utilized. As the most important metric to measure improvements, the total execution time of the simulation of $t\bar{t}$ events is also considered. All efforts are focused on structural changes, which do not influence the simulation output and can be attributed to CPU specific issues, especially front end stalls and vectorization. The most promising change is the activation of profile guided optimization for Geant4, which is a critical external dependency of the simulation. Profile guided optimization gives an average improvement of 8.9% and 10.0% for the two considered cases at the cost of one additional compilation (instrumented binaries) and execution (training to obtain profiling data) at build time.

Contents

1 Introduction

Simulation plays a prominent role in modern physics experiments. It allows scientists to study the inherent properties of large and complicated measurement devices and to test theoretical hypotheses against real data. The ATLAS (***A T****oroidal* ***LHC A****pparatu****S***) detector is one of four larger experiments at the *Large Hadron Collider* (LHC), CERN [Col08]. It introduces a great demand for extensive simulation due to its complexity and the necessity for high statistics.

For this purpose the simulation framework *Athena* [Col10] was created, which is based on the *Gaudi* framework [Bar+01] and utilizes *Geant4* [Ago+03] to simulate the physical processes. Athena allows for a very diverse usage including event generation, simulation of particles trajectories, deposited energy in active detector material and reconstruction of particle events, which is done in the same fashion as with real measurement data. This modularity is achieved by wrapping the C++ core components of the framework with python scripts that allow for dynamic loading and usage of each component.

Event generation and digitization requires much less computational costs, which motivates the investigation of optimizations in the simulation step [Col10].

The general aim of this thesis is to analyze the current performance of the Athena framework and to identify and implement possible ways of optimization. For this purpose Intel VTune Amplifier, an event based sampling profiler, is utilized. This should be done in a structural fashion that keeps numerical computations unchanged, such that simulation results remain the same. If the simulation would change in such ways, new validation of the physical behavior is necessary.

The focus of this project lies on CPU specific topics, especially the investigation of front end stalls and vectorization. Most promising optimizations are the utilization of profile guided optimization and several smaller changes that arise from following profiling results.

Since there is another student, Tobias Wegner from Hochschule Niederrhein, working on the same project [Weg16], the project has been divided into two parts, where I am focusing on CPU issues. Due to the nature of computation, our parts are tightly entangled and we are therefore working on some aspects together and share results. Those sections are marked accordingly.

Chapter 2 comprises a brief description of the ATLAS detector and the physical processes, as well as hard- and software fundamentals that are needed for the discussion of optimizations. It also includes a detailed discussion of the Athena framework and of previous optimization efforts. Chapter 3 presents the utilized tools, the benchmark test setup and a detailed discussion of the simulation performance before any optimizations are applied. In chapter 4 I am explaining several attempts of performance improvements in a case by case study. Chapter 5 summarizes the changes that are done and includes a performance analysis with all improvements applied.

2 Foundation

This chapter summarizes all essentials to discuss the ATLAS simulation and possible performance optimizations. Section 2.1 gives an introduction to the CPU architecture and the instruction decoding process, as well as other CPU specific topics. This is followed by a brief introduction to particle physics in section 2.2 and the real ATLAS detector, described in 2.2.3. A detailed discussion of the Athena framework in 2.3 is followed by a brief section on previous optimizations in section 2.4.

2.1 CPU Architecture and Instruction Decoding

In order to discuss performance issues and possible improvements, it is necessary to elaborate on the underlying CPU architecture and to show how these performance issues come about.

The following description is based on the *Intel Optimization Reference Manual* [Cor16] and focuses primarily on the instruction pipeline. Figure 2.1 gives an overview of the Sandy Bridge core. It is divided in the *in-order* and *out-of-order* sections. While the in-order part is comprised of the instruction decoding pipeline and similar functions that have to respect the order in which instructions are processed, the out-of-order engine can execute instructions in parallel through several ports that give access to the *Arithmetic Logical Unit* (ALU) and other (vector or memory) operations.

Memory data and instructions are cached in separate $32\,kB$ level 1 (L1) caches. Both receive their data from the (unified) $256\,kB$ L2 cache. These caches are distinct for each core of the CPU, and are fed by the global *Last Level Cache* (LLC). In the following the L1 instruction cache will be abbreviated with *ICache*.

2.1.1 Instruction Execution

Instructions are processed in several steps:

1. The *Branch Prediction Unit* (BPU) fetches a block of code from:
    - Decoded ICache
    - ICache, via decoding pipeline

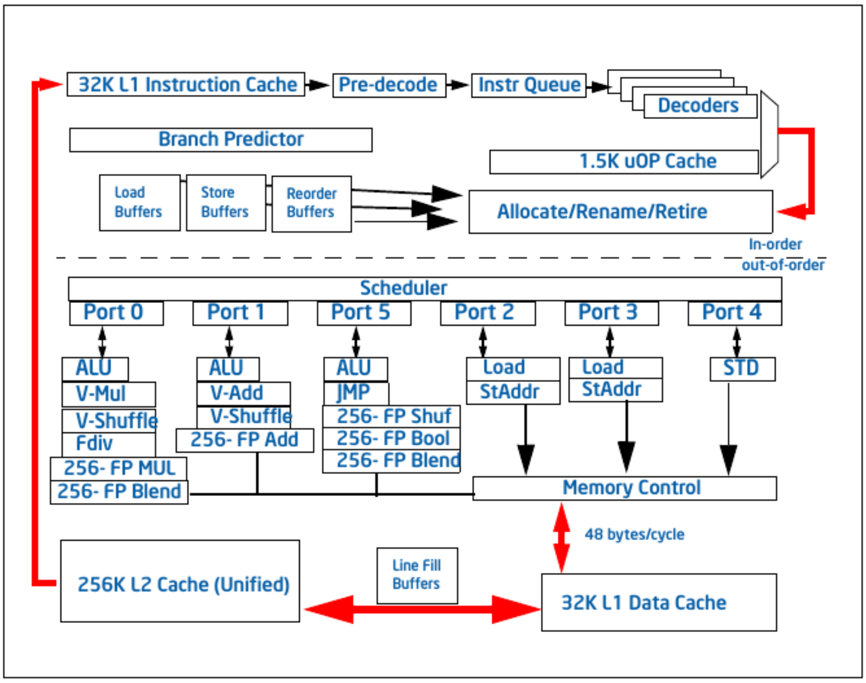


Figure 2.1: Schematic of the Intel Sandy Bridge core architecture and pipeline functionality [Cor16].

- L2 Cache, LLC or memory, if necessary

2. Corresponding *micro operations* (uOps) are send to the rename/retirement block. Entering the scheduler in program order, but are processed & deallocated in data flow order. Branch mispredictions are signaled at branch execution ⇒ front end resteers to new branch

3. Memory operations are managed/reordered for parallelism. Cache misses result in fetches from higher levels in the hierarchy

4. Exceptions are signaled at the (attempted) retirement of the faulting instruction

Most of these stages need further explanation, since the components partly implicate other features, which can have an impact on the program performance.

The *BPU* determines the next block of code which has to be executed, thus predicting the correct path in branching sections of the code. It can efficiently predict conditional branches, direct (fixed address) and indirect (computed address) jumps and returns. This prediction is based on previous branches and their jump addresses. Mispredictions result in front end resteers, implying that the pipeline has to be flushed and filled with new instructions from the correct branch. This automatically introduces performance penalties, since a coherent output of decoded instructions is essential to utilize the complete core. The BPU even allows for execution of branch related code, before the branch is taken and therefore utilizing the out-of-order engine appropriately.

The ICache, or *Instruction Cache*, is a 32 kB cache to allow for faster access to hot code sections, i.e. frequently executed functions. If the reused proportions of the program instructions are compact and completely fit into the ICache, no overhead for access to higher memory hierarchy levels has to be paid.

The *Legacy Decode Pipeline* decodes instructions to micro operations, which are stored in the Decoded ICache and send to the uOps queue. The pipeline consists of the ICache, *Instruction Translation Lookaside Buffer* (ITLB) – translating virtual memory addresses into physical addresses – Instruction Predecoder, the Instruction Queue and four Decode Units. A miss during the ITLB lookup results in a seven cycles penalty. The Predecoder can take 16 B of aligned instructions per cycle, meaning that unaligned instruction data can reduce the throughput.

During this stage two performance improving methods can be applied, *Micro-* and *Macrofusion*. The former describes the process of fusing multiple uOps into a single *complex uOp*, while the later describes the combination of two instructions into a single uOp.

The *Decoded Instruction Cache*, also called Micro Operation Cache, is another cache level, which provides an increased uOp bandwidth at lower latency and power consumption than the legacy pipeline. Intel mentions an average hit rate of 80%, nearly 100% for hotspots [Cor16]. It also reduces the latency penalty if a branch misprediction occurs, since it can store decoded instructions for different branches simultaneously. The Decoded ICache stores its uOps in 32 *sets*, with 8 *ways* containing 6 uOps each, resulting in a total of 1536 micro operations. These sets and ways have to meet certain restrictions, e.g. only two branches per way, a complex uOp consumes a complete way, etc. If these restrictions cannot be met for a specific code section, e.g. with heavy and dense branching, the Legacy Instruction Pipeline has to be used instead.

Both, the Legacy Instruction Pipeline and the Decoded ICache, push micro operations to the *uOp queue*, which issues the in-order instructions to the rename/retirement unit. This queue uncouples the in-order from the out-of-order section and ensures that 4 uops are delivered each cycle, provided that the front end can supply enough instructions. On this level, the *Loop Stream Detector* (LSD) can detect small loops, with less than 28 instructions, and locks these uOps in the queue until a branch misprediction occurs.

The execution core is *superscalar* and can process instructions out-of-order in the *out-of-order engine* (OOE). The OOE can detect dependency chains between instructions and execute computations out of order, while maintaining correct data flow. It is comprised of the rename/retirement block and the Scheduler. The *Renamer* moves uOps from the front end to the execution core and eliminates false dependencies among uOps. uOps are queued in the *Scheduler* until all source operands are present. Afterwards they are dispatched to available execution units as close to a FIFO order as possible. The *Retirement Unit* handles faults and exceptions, which occur during execution.

2.1.2 SIMD Capabilities

Modern processors have different instruction sets to offer parallel, SIMD (*Single Instruction Multiple Data*), operations. These packed computations work on separate floating point registers, specially designed for this purpose.

The Sandy Bridge architecture provides the SSE, SSE 2, 3, 4.1, 4.2 and AVX instruction sets, that work on up to 256-bit wide data simultaneously. This is equivalent to four double floating point operations per cycle.

To utilize these vector capabilities, critical sections of the program can either be written in assembly by hand or the autovectorization features of various compilers can be applied. When using autovectorization, (nested) loops should be arranged in a way that dependencies between intermediate results are avoided. Conditional branches inside loops also have a negative impact on autovectorization and should be avoided.

Usual `gcc` flags are `-ftree-vectorize` (included in `-O3`), and `-ftree-vectorizer-verbose=2` to print the output of the autovectorizer, allowing for verification if certain loops are vectorized or not.

2.1.3 Ivy Bridge CPU

The test computer has an *Ivy Bridge* XEON CPU, which is essentially a *die shrink* of the *Sandy Bridge* architecture with a few additional features. Changes in the Ivy Bridge architecture with respect to Sandy Bridge are:

- Hardware prefetch (of data from memory)
- zero latency register MOV, executed during instruction decoding
- uOp-queue holds 28 entries per logical processor. 56 if hyper threading is disabled or the other logical core is inactive
- *Loop Stream Detector* (LSD) (see below) can handle larger loop structures than 28 instructions
- Latency & throughput of some instructions are improved (e.g. 256 bit packed fp divide, sqrt, . . .)

Given that the analyzed Athena setup is only used in single threaded mode, I do not discuss multi-threading technologies, e.g. hyper-threading and others.

2.1.4 Coding Guidelines Based on the CPU Architecture

Many features are either automatically used on a hardware level or implemented on an assembly / compiler level. Spotting or influencing this behavior in a high level programming language is very indirect and has to be identified through profiling.

There are a few guidelines for the structure in high level code that can be deduced by the consideration of the architecture. To utilize the Decoded Instruction Cache, hot sections should not contain more than 500 instructions, 1000 if hyperthreading is disabled or the program is with 100% certainty single threaded. Loops with many instructions should not be unrolled if the resulting code exceeds that 500 instruction limit. Dense branching should be avoided, to meet the way-restrictions of the Decoded ICache. Very small loops should be analyzed, if they meet the requirements of the LSD.

Long *Dependency Chains*, sequential computations where each operation depends on the previous one, should be avoided or reordered, such that out-of-order execution becomes more likely.

ICache misses occur when the required instruction is not present in the L1I Cache. This can be the case when the code is highly fragmented between hot and cold sections or if a function results in a large working set and is therefore not fitting well in the ICache.

Inlining functions can either degrade or improve the instruction cache performance [Goo]. This increases code locality, i.e. reduces fragmentation between hot and cold code sections and therefore makes the code more cache efficient. On the other hand it increases the code size that can lead to increased occurence of cache misses. The *Google C++ Style Guide* suggests to only inline frequently used functions with ≤ 10 lines of code.

Front end stalls due to instruction decoding issues are a complex and global phenomenon, that is not easily influenced by single changes in a high programming language. It is suggested, that *Profile guided optimization* (PGO), as discussed in section 3.1.3, and other function reordering compiler options should be investigated in this context [Corc].

2.2 Physics Background

The ATLAS experiment is a fundamental research project in the area of elementary particle physics. I am following chapters 1 and 2 of Griffiths' *Introduction to Elementary Particles* [Gri08] to give an overview of the Standard Model. To describe the interactions of particles with matter I will follow chapter 6, “Energy Deposition in Media”, from *Introduction to Nuclear and Particle Physics* by Das and Ferbel [DF03].

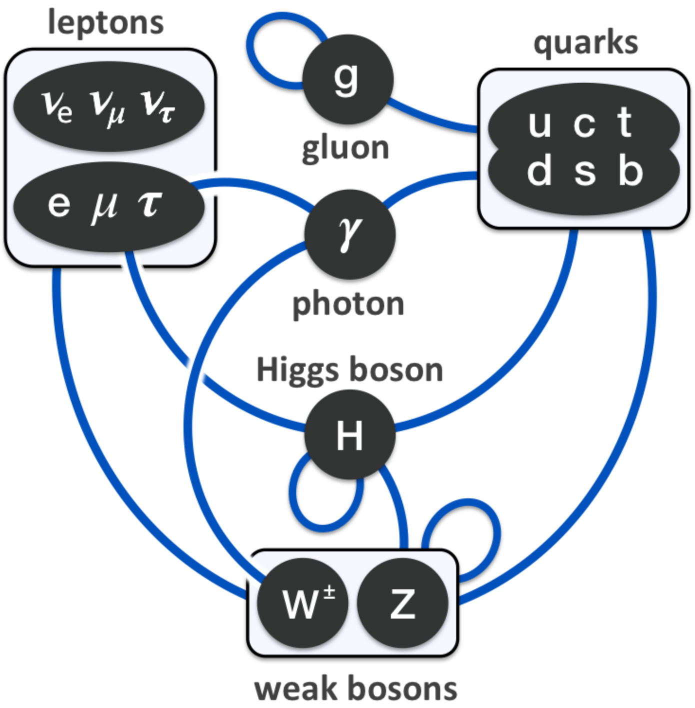


Figure 2.2: Particles of the Standard Model and their interactions [Dre14].

2.2.1 Standard Model

The Standard Model of elementary particle physics describes all known elementary particles and their interaction with each other. There are six *leptons*, six *quarks*, their antiparticles, four *gauge bosons* — that act as carriers of the strong, weak and electromagnetic interactions — and the Higgs particle. Leptons and quarks are sorted into three generations, where each generation behaves similar in certain characteristics, e.g. same charge, but with larger masses than the previous generation. Each generation also introduces special quantum characteristics, for example strangeness and lepton numbers.

Leptons are the *electron*, *muon* and *tau* particle, with an electrical charge of −1 each, and the corresponding neutral *neutrinos*. The quark families are: *up* and *down*, *charm* and *strange*, *bottom* and *top*, with a charge of $-\frac{1}{3}$ and $\frac{2}{3}$ respectively.

In figure 2.2 you can see that these particles interact through gauge bosons, which are a representation of the three fundamental forces described by the Standard Model. The *photon*, responsible for electromagnetic interactions, couples to leptons, quarks and the $W^{\pm}$ bosons, i.e. every particle with electrical charge $\neq 0$. The *gluon*, responsible for strong interactions, couples to all quark families and itself. $W^{\pm}$ and Z are the gauge bosons of the weak interaction. The Higgs boson is responsible for the mass of elementary particles through the Higgs Mechanism (chapter 10.9 [Gri08]).

The mentioned anti-particles (e.g. anti-leptons, anti-quarks) are characterized through opposite behavior with respect to electrical charge and other properties – e.g. lepton number, color charge – while the characterization through possible interactions etc., stays the same.

All these particles and their interactions make up every observable matter and describe our current understanding of particle physics.

2.2.2 Interactions of Particles with Matter

Although ideal experiments should not distort the measured process in question, particle detectors can solely detect particles through interactions. Most deposited and detected energy comes from electromagnetic interactions and is detected through measuring currents or with light sensitive devices. Charged particles can *ionize* the medium in their path, which produces a measurable current. Another process is the *excitation* of atoms and molecules. In this case the excited atom emits a photon, when dropping back to ground levels. Lightweight particles, like electrons, can also lose energy through *bremsstrahlung*, i.e. emitting photons, when accelerated in the electromagnetic field of atoms in the medium.

Important parameters for these interactions are charge and mass from the penetrating particles as well as the atomic number and molecule structure of the medium.

The *radiation length* X_0 is defined as the distance an electron travels, before its energy drops to $\frac{1}{e}$-th of it's original energy. It is a constant which depends on the atomic mass and atomic number of the medium and can be used to estimate the properties of irradiated matter.

Photons can interact through three processes, where each one is dominant for a different energy scale. These three effects are the *photoelectric effect*, *compton scattering* and *pair production*. The photoelectric effect describes the absorption of a photon by a bound electron, such that its kinetic energy is enough to leave the atomic bound state. Compton scattering is often described as the interaction of a photon with a free electron. Pair production is only relevant for photons with a large energy, such that it can produce a positron-electron pair.

Neutral hadronic particles, e.g. neutrons, can interact with the irradiated matter through strong interactions, i.e. scattering with nuclei elastically or inelastically. In general hadronic particles, e.g. protons, neutrons, mesons, interact with other hadronic matter in the irradiated medium and can produce very complex cascades with a wide variety of involved particles, also called *hadronic shower*.

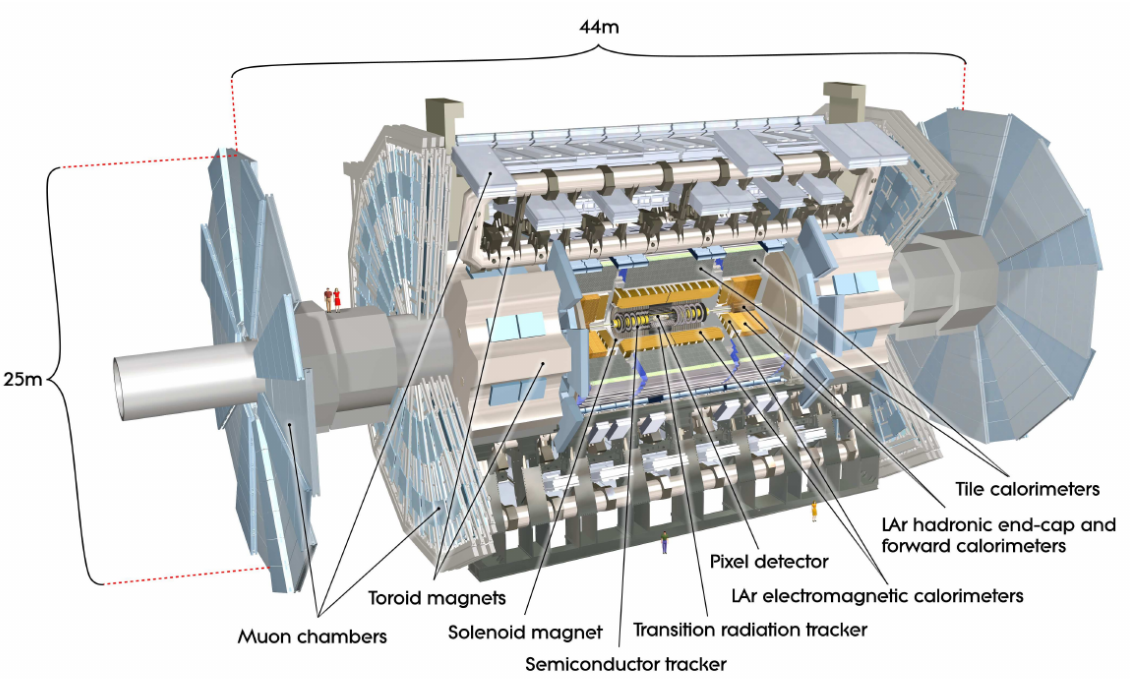

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 2.3: Overview of the ATLAS detector [Col08], p 4.

2.2.3 The ATLAS Detector

The ATLAS detector is one of two general purpose detectors at the LHC, CERN. It is build to identify a broad range of particles, created by proton-proton collisions at 7 and up to 13 TeV. It is constructed symmetrically around the interaction point in its center and consists of cylindrical layers around that origin. To give an overview, I am following the first chapter of [Col08].

In figure 2.3 you can see all major components and the physical dimensions of the detector. Most components are part of the *tracking* detector, *calorimetry* and the *muon spectrometer*.

The inner detector, including the Pixel, SCT (silicon microstrip) and Transition Radiation (TRT) detectors, is responsible for tracking the trajectories of traversing particles. Its main task is to measure the momentum of primary and secondary particles.

The main purpose of calorimeters is particle identification through measurement of deposited energy. There are separate calorimeters for electromagnetic and hadronic particle interactions, due to their different behavior in matter. There are Lead-Liquid Argon (LAr) calorimeters to capture electromagnetic showers and tile calorimeters, as well as LAr calorimeters to detect hadronic showers.

A good impulse resolution for muons is required for certain physics studies. For that reason there is a dedicated muon identification system at the outer regions of the detector. It also implements additional trigger features for automated event selection.

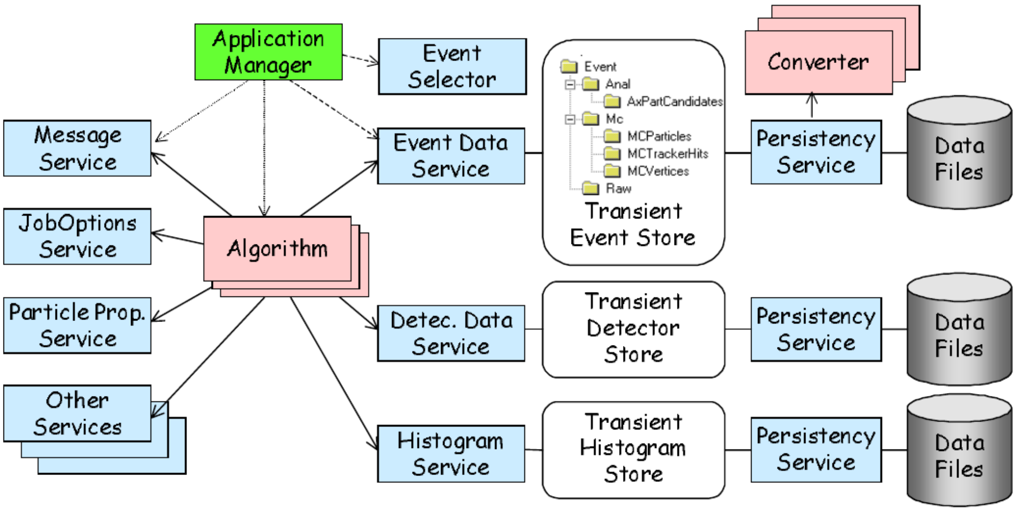


Figure 2.4: Gaudi schema of central software components [CM01].

2.3 Description of the Athena Framework

The ATLAS software framework *Athena* [Col10], adapts the *Gaudi* framework [Bar+01], that implements C++ algorithms and objects, which are dynamically loaded and wrapped with python scripts.

Figure 2.4 gives an overview about the core components of a Gaudi application. Not necessarily all components are used within Athena, but their central concept and their roles become clear. Prominent examples are the *Application Manager* that controls the program execution, the *Algorithms*, which implement the operations to be executed on event data, and services like the *Message Service*, *JobOptions Service* and loader services, which transfer data from persistent storage to transient stores.

The algorithms are combined in one or many *Algorithm Sequences*, which are applied to each event once during the *eventloop*. An *event* is the collection of all primary particles, after a generated collision with all immediate decays. Each algorithm has access to tools and services, which might be called several times during one iteration of the eventloop.

In the context of this thesis, the process of propagating the initial particles through the detector geometry is called *simulation*. The resulting trajectories are called *tracks*. Each track is computed in many *steps*, whereat the step length is dependent on the local material, material boundaries, physics process and accuracy considerations.

Digitization is the simulation of currents and voltages of detector electronics, to produce a file format identical to real detector data. This way all following tools, e.g. reconstruction, work identically on simulated and real data. As a speciality, the simulated data also contain *truth* information, which is initially created during the generation and is also processed throughout the full simulation chain. The truth record holds information of important interactions that are simulated, such that the reconstruction can be validated for each process afterwards. For the benchmark setup, I read the generated events from a precomputed file and solely focus on the simulation step.

2.3.1 Athena Sequence Diagram

Figure 2.5 illustrates the sequential execution of a typical Athena simulation job. A few inexact simplifications are present, but improve the clarity of the diagram itself. I will point these out in the following description.

Athena itself is a python script that is dynamically loading necessary libraries and evaluates the *jobOptions file*, which is used to define the behavior of the current job.

The *ApplicationManager* (AppMgr) is created afterwards and responsible for bootstrapping the whole execution. It manages the most common *services* (Svc), but not all services, which might be the interpretation from the diagram. Every Algorithm can create services at later times during execution. Some important services are not mentioned in the diagram, e.g. the StoreGateSvc which is responsible for the access to persistent data.

The initialization of the detector geometry is also a simplified process and is in reality more complicated. The most important part which was omitted in the diagram is the conversion of the geometry format in a Geant4 format, once the G4AtlasAlgorithm is created and initialized.

The AppMgr creates the *EventLoopManager*, which is responsible for managing the algorithm sequence and its constituents. Each algorithm is created and initialized in the order of this sequence. Some user controlled variables are set during the initialization phase, using the *JobOptionsSvc*, which holds all options defined in the beginning by the jobOptions file. The G4AtlasAlgorithm is creating the environment for the execution of the Geant4 part during the eventloop. As another simplification the Geant4 Event-, Tracking- and Stepping manager are omitted here but further explained in section 2.3.2.

Each algorithm can implement a `beginRun()` and `endRun()` function, which are executed right before and after the eventloop. In the eventloop itself the `execute()` is executed for each algorithm on the current event data in the order defined by the algorithm sequence. Data access, e.g. for the geometry data, reading/writing event data, getting magnetic field data, happens solely through corresponding services. After the eventloop, each parent instance is responsible for the termination of its children, which is somewhat simplified in this diagram. Every algorithm can implement a `finalize()` method that is responsible for its own clean up work.

The algorithm `CCAPIAlg` is part of a special package that Tobias Wegner and I created together. It utilizes the Intel *Collection Control API* [Cora] to start and stop the collection process of VTune in the already mentioned `beginRun()` and `endRun()` functions. This way the initialization and finalization of the simulation can be skipped from profiling. The diagram shows a different configuration which starts and stops the collection process in

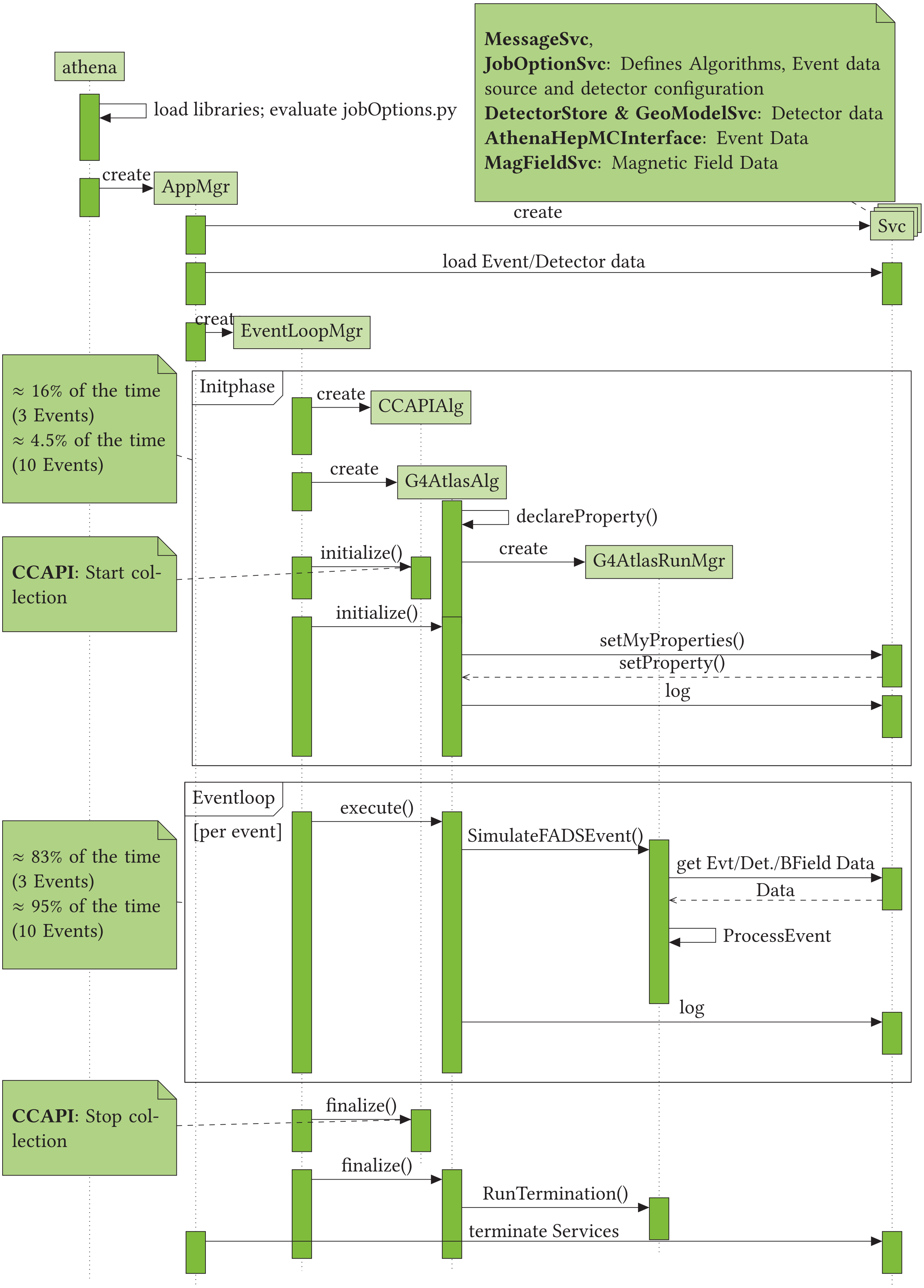


Figure 2.5: Athena sequence diagram, based on a diagramm in the Gaudi Users Guide [CM01], page 33.

the initialization and finalization method. I also implemented time logs to get the included timing information, using the MessageSvc and <chrono>.

Notes in the diagram show a rough estimation of the relative time the application spends in each phase. Since the time is only measured after the constructor call of the CCAPI algorithm all processes before are not measured. This is indicated with the initphase box, showing that we only measure the time spent inside this region and in the eventloop.

With 3 events we spent roughly 16% in the initphase and 83% in the eventloop, compared to the total execution time, which is retrieved from the log file. For 10 events the ratio shifts towards the eventloop with 4.5% spent in the initphase and 95% in the eventloop. Since a typical simulation job consists of 50 or more events [Col10], the importance of optimizing the eventloop and the execute() functions of the algorithms, as well as possible service calls, becomes very clear.

2.3.2 Description of Geant4

Geant4 (G4) is a software toolkit to simulate the interaction between particles and matter. It is widely used in various fields of science and medical applications. To give an introduction to the core component of the ATLAS simulation, I am following [Ago+03], which describes the state of Geant4 in the year 2003. Later paragraphs will update this picture and point more specifically to the implementation for Athena.

Geant4 follows an object oriented design, where geometry and materials, particles, particle interactions, tracking, hits, event- and track managers are represented as classes.

One *run* can have multiple events. An *event* is the collection of all primary particles, produced by the generators, including the option of *simulation truth*, which stores the real interactions at each stage, such that the reconstruction can be validated afterwards. *Tracking* is the process of transporting (*stepping*) a particle through the detector. At each *step* the particle can go through three different types of *actions*, which happen *at rest* (e.g. decay), *along step* (e.g. energy loss) and *post step*, which implement the physical processes. The user can specify own actions through *UserActions*.

The G4TrackingManager is an interface class, responsible to handle data between the event- and the tracklevel. This is done through message passing between the G4EventManager and hierarchical lower levels. Stepping actually happens in the G4SteppingManager. A single track is represented as an instance of the G4Track class and holds information like the current position, passed time since beginning of the stepping process, identification of the current geometrical volume, etc. Tracks are stacked on three different stacks, *urgent*, *waiting* and *postpone*, to organize the processing priority of the tracks during the eventloop.

The geometry is implemented using a voxel based method, dividing the space into cubical volume elements (voxels) and a tree based map. The SmartVoxel technique divides each mother volume into a one dimensional virtual division (sliced along a specific axis). Each slice contains pointers to their subvolumes. The best slicing axis is chosen by heuristics and subdivisions containing the same volumes are merged.

Charged particles have to be transported in an electromagnetic field, which is numerically approximated in the whole detector volume. The trajectory curve is approximated, while the accuracy parameters can be controlled by the user.

In 2006 some essential features have been implemented and a short description is given here, following [All06].

The *Run manager* allows to control the Geant4 configuration, and can read preproduced events from file.

Regions are implemented to allow for a division of the detector geometry into subdetectors. Each region can have different parameters, e.g. the threshold for secondary particle production or integration accuracy. With the `G4Navigator` a new abstraction of the navigation within the detector geometry has been implemented.

To improve the propagation of particles in magnetic fields, different fields can be attached to different geometry volumes, allowing for a different accuracy or particle behavior per region.

Since Geant4 is usually a hard compiled C++ application, it has to be wrapped with python code to provide modifiable parameters during runtime in the Athena framework [Col10]. The FADS (Framework Atlas Detector Simulation) wraps several Geant4 classes to allow for a dynamic selection and configuration without recompilation.

FADS objects can be translated to G4 equivalents and G4 can be accessed through services and the G4Algorithm. The eventloop is implemented as a wrapped service that provides additional handles. To process the input events from the generators, these can be converted from the HepMC to the default G4 format. After the eventloop is processed some analysis is done to ensure that the process finished without any errors.

2.3.3 Common Data Formats

In this thesis I will focus on the simulation step in figure 2.6. The input to the simulation is done in the *HepMC* format and can be filtered to fit certain needs, e.g. leptonic decays, missing energy above certain values [Col10]. These are the particles, produced by a simulated particle interaction in an event generator, i.e. simulating the collision process at the interaction point of the detector. These generated events can be computed in the current job or read from file. Filtered particles are processed together with its MCTruth data.

Output of the Geant4 Atlas simulation is given in the *hits file* format, e.g. `atlasG4.hits.pool.root`, which contains the deposited energy at a certain geometry of the detector, packed with the MCTruth data and other meta data, like the simulation

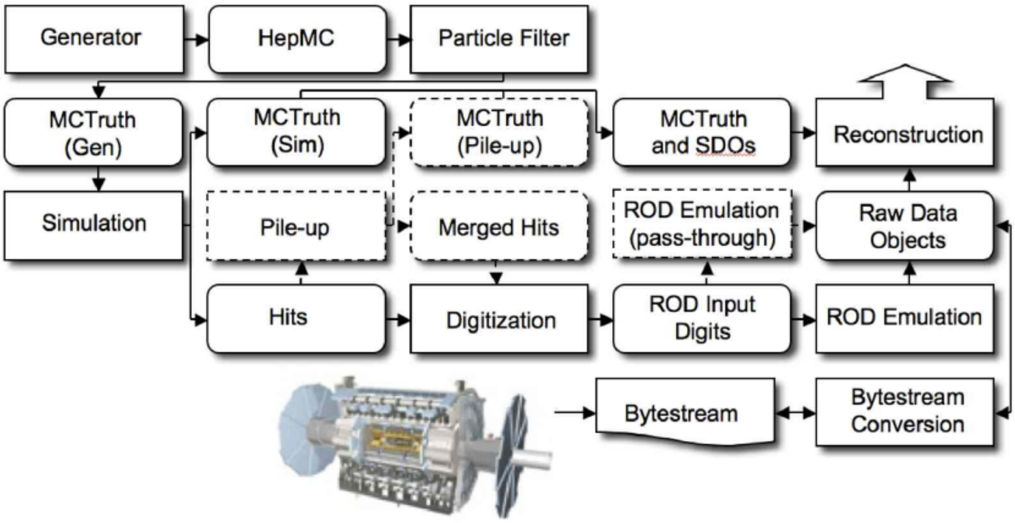


Figure 2.6: Data flow of Athena. Square boxes indicate operations and rounded boxes data formats [Col10].

configuration. Its size is mentioned to be ≈ 2.5 MB/event [Col10]. Own observations are hinting towards a size of ≈ 1 MB/event. To reduce the file size, showers in the calorimeter have been merged.

The digitization step is split from the simulation since the workload per event is much higher during the simulation than during digitization. A typical digitization job consists of $\sim 1k$ Events, while a typical simulation job is working on 50 - 200 events.

2.3.4 Remarks on the Detector Geometry

The simulation supports different layouts of the detector geometry, including test beam setups and different states throughout the build phase of the detector [Col10]. This data is loaded into a `GeoModel` compliant format, which is not only used by the simulation but also for digitization and reconstruction.

The model consists of *solids*, *logical volumes* and *physical volumes*. Solids are basic shapes without a position. Logical volumes are solids with properties, e.g. physical material and corresponding characteristics. Physical volumes are logical volumes, located at a specified position. Some approximations for dead materials, cables and cooling pipes, are done to simplify the model.

To simplify the placement of large structures, a single logical volume can be repeated several times through *volume parametrization*. Physical volumes can be nested and can have defined dependencies to create subdetectors as distinct parts of the whole detector. Each subdetector is responsible for including all of its own materials and elements.

A unique identification of the current location of a particle is necessary and overlaps between volumes have to be avoided. This can be achieved through small gaps between volumes, at the additional cost of extra intermediate steps during simulation for each traversing particle [Rim+08b].

Two databases handle the detector geometry information in every Athena job. The *Atlas Geometry DB* stores basic constants, volume dimensions, rotations, material, tags and links to external files, while the *Atlas Condition DB* stores the conditions, dead channels and misalignments of the detector. Both have version controlled data and a tree structure.

During the initialization of the simulation, the GeoModel has to be translated into a Geant4 compliant format. The GeoModel description can be released afterwards to improve the memory footprint. Each particle stores its location in the G4 geometry description to speed up the lookup time during the stepping loop.

2.4 Previous Optimizations

There have been previous efforts to improve the performance of Athena. Reports on these were published in April and December 2008 and evaluate the performance at the start of ATLAS data taking. Another report was published in 2010. I will briefly summarize their findings in this chapter and describe the state of the simulation code at previous times.

In *First Report of the Simulation Optimization Group* [Rim+08b] the main motivation is justifying the use of Geant4 and to propose possible changes. A good proportion of the investigation is dealing with different physics models and time-/rangecuts for low interacting particles. These cuts describe limits after which the corresponding particles are completely dumped at their current position to reduce computational costs. As an example, reducing the timecut of thermal neutrons to 150 ns improved the issue that the HITS file was significantly larger with the preferred physics model at that time.

The stepper solves an ordinary differential equation (ODE) for each next step of a particle in a magnetic field. The parameters for this solver are changeable and adjusted differently for each subdetector to reduce computational costs through unnecessary accuracy. This could be further improved by reducing the order of the solver (4th order Runge Kutta), but analysis showed that accuracy errors can accumulate to significant offsets for muons with ≥ 1200 steps. A different solver could be used for each particle type.

Final Report of the Simulation Optimization Task Force [Rim+08a] discusses optimizations that were evaluated and implemented. A crucial improvement was the removal of string comparisons in core and detector packages, which were executed every step. The total number of steps easily reaches several millions. (see 3.3.3 and table 3.2) Strings have a more complex structure than simple data types, thus very frequently executed functions should not contain such operations, if not necessary. Name checks could be implemented by checks of integer encoded IDs, etc.

The GeoModel representation of the detector is translated to the Geant4 format during initialization and should be released afterwards. An estimation is that roughly 100 MB of RAM could be saved by this process, but this was not done at that time.

To improve the output file size of the HITS file, the above mentioned neutron timecut of 150 ns was approved and additionally hits in the silicon detectors are stored as a collection along the minimum ionizing particle (MIP) path, instead of individual hits. This removed duplicate information and reduced the storage size by a factor of 2.

A new stepper dispatcher allows for the selection of stepping parameters for each particle type and region. The accuracy of electrons in the calorimeters is relaxed, while high precision of muons in the tracker and calorimeters is maintained. Up to 20% of the total simulation time is spend in magnetic field functions and a second order RK integrator would reduce the calls to the magnetic field functions from 4 to 2. If a uniform field over the length of one step is assumed, this can be further decreased to only one function call per step. Therefore decreasing the accuracy of the integration step for particles where it is possible, can increase the performance significantly.

The benchmarking was done on a silent machine, since the runtime RMS on the CERN batch system was at 5 − 10%, which makes subsequent tests not very comparable. On these silent machines the RMS was about 1%. The tool `taskset` was used to lock the simulation process to a specific core and avoid cache misses due to shifts between different cores.

Final Report of the ATLAS Detector Simulation Performance Assessment Group [Apo+10] is the final report on another optimization period in 2010. The simulation performance assessment group lists various methods that consume much CPU/memory resources as first targets for further improvement.

Several Geant4 features are discussed that could reduce the number of simulation steps and potentially reduce the CPU time, at the cost of a possible impact on the physics. Allocations and memory usage is an important factor in this discussion. Given that the focus of this thesis lies on CPU behavior, the description of these findings will be skipped.

The most important performance metric mentioned is the CPU time for 50 $t\bar{t}$ events.

Athena was instrumented with callgrind to count CPU cycles and instruction/function call rates to isolate hotspots.

Some example hotspots that are found by this approach are the stepping function and a function to determine the physical step length of the `G4SteppingManager` class.

As a conclusion the analysis showed that 26% of the time is spent with magnetic field related operations, including `G4AtlasRK` and the stepper dispatcher. 15% is spent with geometric functions, dealing with computations of distances between certain points.

The last section of this report suggest that other tools should be utilized to analyze cache- and instruction misses, identifying frond end stalls as a major topic of interest.

3 Performance Analysis

This chapter discusses utilized tools in section 3.1 and the benchmark test setup in section 3.2. Both sections give a necessary foundation for the performance analysis in section 3.3, which defines a reference state for all attempted optimizations.

3.1 Utilized Tools

The most important tools for this project are the profiler Intel VTune Amplifier, own scripts for log file parsing and the GCC compiler. Other scripts offered by Athena are discussed later, when first needed.

3.1.1 VTune Amplifier

Intels *VTune Amplifier* is an *Event Based Sampling profiler* that is capable of analyzing the run time performance of a compiled program with little overhead. To describe its functionality, I am following an article about the *General Exploration* profiling mode [Mar15].

VTune utilizes the *Performance Monitoring Units* (PMU) of the CPU to extract *hardware events*. These events cover a wide variety of CPU operations and indicate what the CPU is doing at a specific point in time. Examples are counters for taken branches and branch misprediction, as well as simple counters for instructions, that are retired in the retirement unit and eventually executed.

VTune uses *Event-Based Sampling* (EBS) to profile the program execution. In this process the PMU registers issue an interrupt, when a certain limit in the event counters is met. At this interrupt the *instruction pointer* (IP) is read and all hardware events of the previous interval will be assigned to the corresponding instruction, function call and library. This is a statistical process which only has significance if enough samples are collected, i.e. the workload/time of the program execution is large enough.

As a benefit, EBS introduces only a very small overhead. Intel refers to an average overhead of 2% [Corb]. Given that the number of PMUs is limited and some *metrics* — a computational combination of a range of hardware events — require many hardware events to be computed, the total number of collected distinct events can easily exceed the number of present PMUs. This problem is circumvented either through multiple runs, which might be an issue if the program behavior is not reproducible, or through *multiplexing*. With multiplexing the collected hardware events are periodically swapped, such that all events can be collected at a smaller rate. This introduces a greater statistical uncertainty.

Another possible complication is *event skid*, which describes the wrong assignment of hardware events to a different IP, several instructions after the original IP. This can happen due to delays in the interrupt handling.

With hardware events and corresponding metrics, VTune can offer a lot of information. It is capable of identifying the hotspots, where most of the time in the application is spent, as well as offering information of more complex nature, such as L1/L2 hits/misses, branch mispredictions, instruction misses, FPU utilization and so on.

VTune allows for several profiling methods that cover different aspects of the program analysis. These profiles carry preselected collections of events that are already combined into complex metrics. In this project two profiles are used, *HPC* and *General Exploration*, that cover vectorization/floating point operations and a general overview about CPU and cache behavior.

Intel offers the *Collection Control API* (CCAPI) to control the data collection process of VTune, which includes three simple C++ functions that resume, pause or start the collection when called. To utilize the CCAPI, the application must be instrumented by hand with these function calls and recompiled and linked against VTune supplied libraries. This allows to selectively profile only certain parts of the application in question.

3.1.2 Run- & Parse Scripts

I have written two scripts to automate the timing measurements of the simulation. The first, `poetAtiming.sh`, is a bash script that takes the corresponding jobOptions file and the number of repetitions of the simulation as input parameters. As an output it redirects the log of each run in a separate text file, named `run<X>.log`. It is important to note that the log level has to be set to `INFO` within the jobOptions file. For more details on how to set up the benchmark test setup, please refer to section 3.2.

The second script, `poetTparse.py`, is a python script that parses all logfiles inside the folder specified by the only input argument. The log files contain timing information from several services that are already implemented in the Athena simulation. These metrics are the `total time`, the `total G4 time`, spent in Geant4, and the `average time per event` for that run.

These values are averaged over all runs and printed together with its maximum and minimum, as well as its standard deviation. For better interpretation and identification of unexpected behavior, a plot of the `total time` and `total G4 time` is automatically produced. For this feature the `matplotlib` package is required.

3.1.3 GCC Compiler Optimizations

Many optimization strategies involve compiler options that trigger different behavior during compilation. The most important features for this thesis are discussed here. A complete list of all available features is given at [Frec].

The `-O` flags are collections of many optimization features of different levels. While `-O2` is usually considered to be unproblematic, and used as the default in all Athena components, `-O3` activates quite aggressive features, that can impact on numerical results of floating point computations and potentially change the programs result, for example `-O3` includes autovectorization and `-ffast-math`.

Autovectorization can be activated individually, like any other feature, through a special flag, which is in this case `-ftree-vectorize`. The autovectorizer allows for different verbosity levels to evaluate the vectorization of specific code sections, settable with `-ftree-vectorizer-verbose=n` (e.g. $n = 2$).

PGO and LTO

Profile guided optimization is a method to optimize programs in several steps [Wic+14]. PGO is comprised of 3 steps:

1. Instrumentation
2. Training
3. Optimization

First, `-fprofile-generate=<PATH>` is used to instrument the program binary to collect profiling data at `<PATH>`.

In the second step, the instrumented binary is executed with common use cases, to produce the profiling data.

The last step is recompiling the program with `-fprofile-use=<PATH>` to utilize the heuristic data and to give the compiler better options for optimization decisions. These decisions effect methods as inlining, block ordering within functions, branch ordering and cache access when arrays are used within loops [Wic+14]. All PGO flags must be included during compilation and linking.

Although this process can produce better binaries, it comes at the cost of a significant larger compilation time, since at least 2 compilations and one instrumented binary execution is necessary. The build overhead might be reasonable when the application execution time is of a large magnitude.

Another possible problem could be the over adaptation to the test case, when the application in question is used in a very broad and generic fashion.

Link time optimization (LTO) allows the compiler to optimize the whole program at link time, meaning that optimization is not only done per module, but also considers relations between modules [Freb]. To activate LTO the `-flto` compiler flag has to be stated during compilation and linking of the application. It is not activated at any `-O` level.

3.2 Usage of the Benchmark Test Setup

```
$ cd /path/to/workspace/
$ setupATLAS
$ asetup 20.3.2.1,here
$ athena --preloadlib= \
  $ATLASMKLLIBDIR_PRELOAD/libintlc.so.5:\
  $ATLASMKLLIBDIR_PRELOAD/libimf.so \
  jobOptions.G4Atlas.py &> athenarun.log &
```

Listing 3.1: Set up of the Athena benchmark test case.

In order to start a simulation job, Athena has to be set up properly, which is presented in listing 3.1.

The first three commands prepare the working directory and set up a local session for a simulation job. `setupATLAS` makes several tools and files from the AFS and CVMFS ([Bun+10]) available. These are distributed file systems used in CERNs computing environments. This is implicitly done by setting certain environment variables and aliases to commands, including `asetup`. `asetup` is responsible for setting up a local – hence the option `,here` – version (20.3.2 with patch 1) of Athena. There are other possible options, e.g. `dbg` to make debugging symbols available, which are not present for every release.

After this step the `athena` command is available, which is just a link to a python script, placed in the current versions folder hierarchy on the CVMFS. `athena` starts a simulation job through the *jobOptions* file with possible additional parameters, in this case to preload different math and C libraries from Intel to improve the performance. The jobOptions file provides all necessary information for a well defined simulation job, e.g. which detector condition or what event data should be used. An important aspect of the jobOptions file is to define the *algorithm sequence*, which should work on the event data. In our benchmark problem the jobOption file includes the G4Atlas (Geant4) algorithm, which sets up Geant4 in an Athena compatible way. A more detailed description of the jobOptions file can be found in appendix A.1.

To ensure modularity, all of Athenas software is bundled in *packages*. An Athena *release* (e.g. 20.3.2) consists of certain versions of these packages, such that the packages can be changed and maintained individually. Unless a certain package of the current Athena release is checked out and altered, the working directory only holds output and auxiliary files, while the remote packages on the CVMFS are used.

cmt is a tool to facilitate the process of package retrieval and compilation. With the command `cmt co` a certain version of a package can be checked out and recompiled with possible code changes. This process creates the `InstallArea` directory in the working directory, which has a higher priority in the look up process of finding libraries. This way the locally placed versions of all components are preferred over the remote ones on the AFS/CVMFS. Every requirement of a package is defined in its *requirements file*, which is used to define additional dependencies or from the default setup deviating compilation flags.

The benchmark problem for this thesis will consist of the Athena version 20.3.2.1 and the jobOptions file printed in appendix A.1. This is done through following the steps in listing 3.1. The jobOptions file retrieves $t\bar{t}$-events from the AFS, as well as certain condition and simulation flags. It is beneficial to place the event input file in a local directory to avoid unnecessary AFS access. Finally this setup includes the Geant4 algorithm to propagate the particles, read from the events input file, through the detector. This step does not include digitization.

3.2.1 Profiling the Benchmark with VTune

To profile Athena with VTune, it can either be attached to a running process or it starts the application itself. For the latter some minor pitfalls should be avoided:

- Choose the correct working directory, otherwise VTune tries to write to read only paths on the AFS

- Choose a working directory on a local drive, access times to AFS directories can introduce greater timing fluctuations

- Do not use environmental variables, e.g. in the path to the `--preload` option. Write them explicitly

- Avoid the “Basic Hotspot" analysis

To elaborate on the last point, this analysis type gets stuck for an unknown reason on a `ld --verify` process. This process can be killed manually and the simulation will work without further issues, but since the “Basic Hotspot” analysis is not the preferred type, no further investigations of this issue are done. The main difference between this and other analysis types is that this type does not utilize the kernel module, i.e. runs in “user”-mode, which could give a hint of why this problem occurs.

Table 3.1: Timings averaged over 10 runs with 50 events each with/without taskset.

	without taskset	with taskset
Total time	$7728.0 \pm 52.3\,\mathrm{s}(\approx 0.68\%)$	$7938.0 \pm 100.6\,\mathrm{s}(\approx 1.17\%)$
G4 time	$7654.9 \pm 56.1\,\mathrm{s}(\approx 0.73\%)$	$7870.0 \pm 91.2\,\mathrm{s}(\approx 1.16\%)$
average-per-event-time	$153.67 \pm 1.13\,\mathrm{s}(\approx 0.74\%)$	$158.03 \pm 1.85\,\mathrm{s}(\approx 1.17\%)$

3.3 Performance Analysis of the Current Simulation

Every test, mentioned in this thesis, was running on a silent, i.e mostly idle, workstation, provided by the OpenLab at Cern. This machine has two Ivy Bridge Intel Xeon E5-2695 v2 CPUs, running at $2.4\,GHz$ ($3.2\,GHz$ in turbo mode). The system has therefore 2×12 cores with hyperthreading enabled and access to $64\,GB$ of memory.

As described in section 3.2, our benchmark problem is using a specific Athena version – and implicitly a specific Geant4 version – and the jobOptions file, described in appendix A.1.

3.3.1 Timing with Taskset

The most important metric of program execution performance is the total execution time of the complete simulation process. This time will not be the same on each run, since the *operating system* (OS) can randomly interrupt the process and reschedule it in different, non-deterministic ways. To get a measure of these variances I wrote a script that executes the same Athena job N times and computes the average total time, its variance and the same for the Geant4 part of the simulation. The scripts are explained in section 3.1.2.

These timing measurements are collected from the simulation log files, which are only gathered, if the `INFO` log level is set. A potential statistical error of these timing values is represented in the variance over all runs, together with other statistical disrupters, e.g. the influence of the OS scheduler. Possible systematic errors should not be relevant, since I am only interested in relative improvements with respect to a reference measurement.

As mentioned in section 2.4, previous performance measurements were done using the linux tool `taskset` [Rim+08a]. To evaluate the necessity of this, 10 runs with 50 events each are executed, using the Athena version 20.3.5.1. Through comparisons it can be shown that taskset is in fact not needed and the Linux process scheduler can be trusted. The execution time as well as the variance increase significantly, using taskset, as shown in table 3.1.

10 runs are not a large sample set to estimate the standard deviation, but even with a corrected (unbiased) sample standard deviation of the total time, $s_{\text{unbiased}} = \frac{100.6}{c_4(10)} \approx 102.8$, the difference to the reference without taskset is still $\geq 2\sigma$ [Wik].

3.3.2 FLOPs Estimation

Tobias Wegner made a detailed analysis to roughly estimate the GFLOPs (*giga floating point operations per second*) during the Athena simulation job [Weg16]. I will reproduce his considerations here to give a certain perspective of the CPU performance.

With a theoretical maximum RAM bandwidth of 59.7 *GB/s*, eight Byte can be transferred in

$$t_{8B} = \frac{8\,\mathrm{B}}{59.7 \times 10^{9}\,\frac{\mathrm{B}}{\mathrm{s}}} = 1.34 \times 10^{-10}\,\mathrm{s}$$

Since one clockcycle, in turbo mode, takes

$$t_{clockcycle} = \frac{1}{3.2\,\mathrm{GHz}} = 3.125 \times 10^{-10}\,\mathrm{s}$$

a maximum of 3.2 *GFLOPs* is possible, assuming that one operation per cycle is possible.

When using the SIMD capabilities of the processor, 4 double or 8 single operations, 256 bit wide registers, 4 or 8 ops per cycle are feasible. In this case the data transfer rate would be the limiting factor.

Tobias Wegner also computed the GFLOPs, based on hardware events, collected with the profiler `perf`[Weg16]. 90% of all floating point operations are unpacked, i.e. no SIMD operations. The events indicate a rate of 853 *MFLOPs*, which is ~ 25% of the theoretical maximum, ~ 6% if four flops per cycle are considered. VTune profiles give a FLOPs value of 860 *MFLOPs*, which can be seen as in good agreement with Tobias Wegners result, since this is just a rough estimation.

The theoretical maximal FLOPs is a poor metric to measure the performance of a given program, since it is never reachable and always limited by practical issues. A few percent of that maximum is already considered to be “good”. This estimation also does not include the information value of the computation. Redundant operations can perform well w.r.t the FLOPs, but a program that does a lot of redundant operations is not considered to have good general performance.

Nevertheless does this estimation substantiate the impression that Athenas simulation process is already with good performance.

3.3.3 Analysis of Geant4 Structures

To gather information on the behavior of Geant4 structures, especially the number of tracks per event as well as steps and time per track, Tobias Wegner instrumented the G4 code to retrieve this data and store it in a binary format [Weg16]. I wrote a python script, `poetRAWparse.py`, to recollect this data and produce histograms that can characterize the typical behavior of the simulation in a descriptive fashion.

Figure 3.1 is showing the computation time in μs and number of steps per track, as a 2D

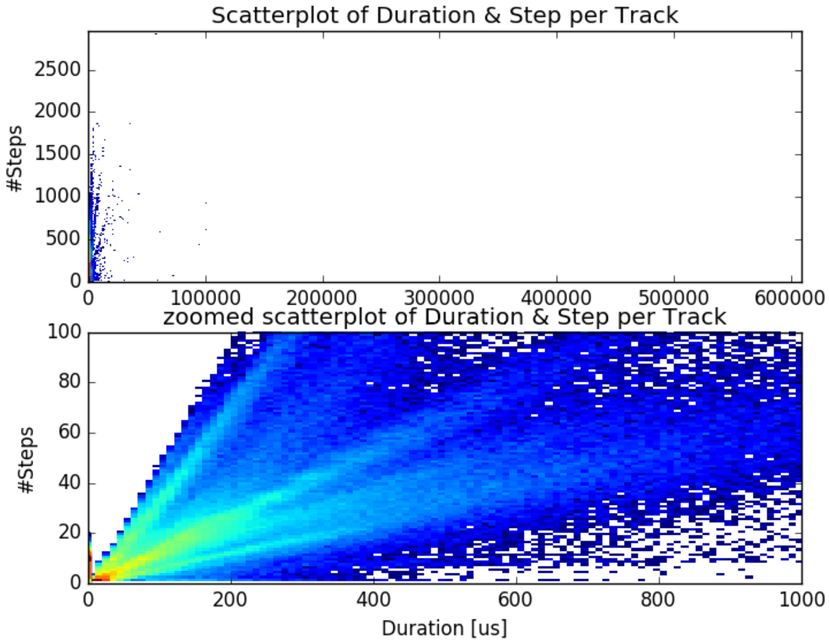


Figure 3.1: 2D logarithmic histogram of all tracks w.r.t. step count & computation time

histogram for the first event of the test data. The second figure shows a zoomed fraction of the graph above, to focus on the active region.

This histogram is highly dependant on the physical characteristics of the underlying, generated event, since different collisions are expected to produce different secondary particles.

Nevertheless two findings become clear:

- The largest fraction of all tracks are processed in a very short time ($< 200\,\mu s$) for a few steps (< 40).
- The computation time per track has different correlations with respect to the number of steps.

In figure 3.2 the same data is presented, separated by particle type, as a scatter plot. The data is summed along both axis into two 1D histograms over the step count and the duration per track.

The 1D histograms clarify that in fact most tracks have a duration $< 20\,\mu s$ and only a single step. These are most probably all secondary particles, which are produced with an energy below a lower bound, i.e. being dumped at their position in their first step.

By coloring the data with respect to the particle type one can see that charged particle tracks (e.g. leptons (red)) take more computational time than uncharged particles (e.g. photons (magenta)). This is expected, since access to the magnetic field data takes time, which is not necessary for neutral particle tracks.

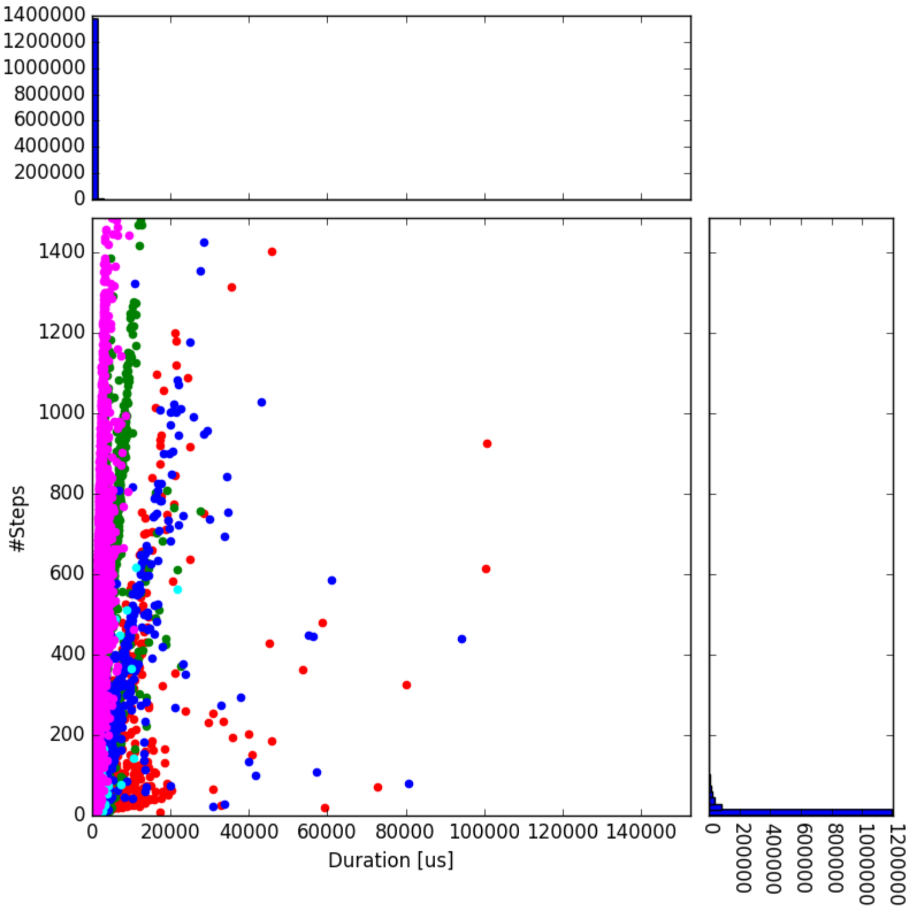


Figure 3.2: 2D Histogram of all tracks w.r.t. stepcount & time, colored by particletype: red: leptons, green: baryons, blue: mesons, cyan: nuclei, magenta: gamma

Table 3.2: Durations and track / step counts.

	Event 1	Event 2
Primary tracks	298	361
Total secondary tracks	1506605	1383613
Average steps per track	8.970	10.506
Total steps	13487189	14512911

As discussed in the previous section 3.3.2, one clock cycle takes 3.125×10^{-10} s ≈ 0.3 ns. Assuming one operation per clock cycle, this would imply that a 20 μs track invokes at most $\frac{20}{0.0003} \approx 66667$ operations. This could be an indication that these type of tracks introduce mostly organizational work.

Assuming an execution time of 105 s for the first event, as found in an example run log, and 1000000 tracks with $\approx 10\,\mu$s each, representing the fraction of single step secondary particles, these particles already take up $\frac{1}{105} \approx 0.95\%$ of the total execution time.

Tobias Wegner also extracted accurate data about the track and step count of the first two events, which is given in table 3.2 [Weg16].

3.3.4 VTune Profile Analysis

The Intel VTune profiler is used for a better hotspot identification and a more in depth analysis of the code issues. For all profiles discussed below, the Athena version 20.3.2.1, with 20 events is used.

Profiles can be collected with and without call stack information, helping to understand the caller / callee structure of the program. Collecting the call stacks introduces a significant data and time overhead (e.g. $\sim$ 3400s instead of 2653s for 20 events).

Each profile is controlled with the CCAPI, as discussed in section 2.3.1. This helps to limit the data taking to scaling portions of the program and avoid profiling constant parts during initialization. Depending on the order of the algorithm sequence and the configuration of the CCAPI algorithm, some initialization code could be included, during the G4 algorithm instance creation, as it is shown in the sequence diagram 2.5. To circumvent this the VTune collection process is started in the `beginRun()` and stopped in the `endRun()` method, which are executed directly before and after the eventloop.

Several abbreviations appear in the discussions of VTune profiles that should be explained beforehand:

- **CPI**: Clockticks per instruction retired – Gives the ratio of how many instructions are issued per clocktick. A CPI of 0.25 is the theoretical maximum of 4 instructions per clock cycle

- **FE Bound**: Front End Bound – Fraction of time the CPU stalls because of issues with instruction decoding and throughput
- **BE Bound**: Back End Bound – Fraction of time the CPU stalls because of data/memory access problems
- **Bad Speculation**: Fraction of time the CPU stalls because of branch mispredictions
- **Retired**: Last fraction compared to the three mentioned above. It gives the fraction of time where the CPU does not stall and instructions are retired (executed necessary instructions)
- **I$ misses**: ICache misses – Misses in the L1 instruction cache
- **FPU utilization**: Fraction of (floating point) operations with scalar and packed (vectorized) instructions per cycle
- **MITE**: Micro instruction translation engine – Translates instructions into corresponding uOps

To focus on separate CPU specifics, two profiling types are chosen: *General Exploration* (GE) for a general overview over a wide range of hardware metrics and *HPC Performance* for a focus on vectorization and overall floating point computation performance.

Table 3.3 lists a selection of functions, which are either identified as hotspots, i.e. consuming a large proportion of the overall CPU time, or exhibiting a pathological computational behavior of some sort.

In the following I will give a few remarks on each mentioned metric and how this hints towards certain optimization strategies. These indications should be understood as a starting points and each investigation probably leads to its own unique insights.

A CPI rate below 1 can be considered as quite performant. A high CPI rate (> 2) could indicate wasteful and unnecessary operations, but could also be attributed to memory bandwidth issues.

If low, the packed floating point operation metric can hint at the possibility for vectorization.

High ICache miss rates suggest that the structure of the executable code can be improved, i.e. moving hot code sections and reduce general instruction code size of that section.

Bad speculation can be avoided by reordering the if/else branches in the order of their execution probability or by making the data more predictable, e.g. through sorting.

Not all cells in table 3.3 are populated due to different focus on certain aspects or because of missing information in different profiling modes.

Table 3.3: Selection of noticible functions, found in Athena profiles and their assigned characteristic behavior

Function	CPU time	CPI Rate	FPU utilization (scalar/packed flop)	I$ misses	FE BW (MITE)	Bad Spec.
`LArWheelCalculator_Impl ::DistanceCalculatorSaggingOff ::DistantceToTheNeutralFibre`	**165.866** *s*	0.718	7.6% (0.541/0.084)	0.038	-	-
`LArWheelCalculator::parameterized_sin`	119.178 *s*	0.534	7.6% (0.648 / 0.011)	0.049	-	-
`MagField::AtlasFieldSvc::getField`	107.016 *s*	1.007	3.9% (0.407 / 0.035)	0.082	-	-
`sincos`	49.811 *s*	1.028	15.9% (0.115 / **1.170**)	0.139	-	-
`G4PolyconeSide::Inside`	39.854 *s*	0.537	5.7% (0.537 / 0.000)	0.046	-	-
`G4AtlasRK4::Stepper`	32.094 *s*	0.671	5.1% (0.427 / 0.000)	0.166	-	-
`G4PolyconeSide::DistanceAway`	31.86 *s*	0.825	3.2% (0.294 / 0.000)	0.091	-	10%
`G4PhysicsVector::SplineInterpolation`	24.993 *s*	1.390	3.8% (0.342 / 0.000)	0.298	-	-
`LArWheelSolid::search_for_nearest_point`	24.125 *s*	0.820	7.8% (0.667 / 0.000)	-	-	17.4%
`G4CrossSectionDataStore::GetCrossSection`	23.248 *s*	1.708	0.8% (0.079 / 0.000)	**0.557**	-	-
`G4Navigator::LocateGlobalPointAndSetup`	21.769 *s*	1.744	0.9% (0.069 / 0.000)	0.056	14.6% (0.188)	-
`G4Transportation::AlongStepGetPhysicalInteractionLength`	18.419 *s*	1.569	0.9% (0.078 / 0.000)	0.467	-	-
`LArWheelCalculator_Impl::WheelFanCalculator <LArWheelCalculator_Impl::SaggingOff_t>:: DistanceToTheNearestFan`	17.35 *s*	0.694	10.2% (0.355 / 0.468)	-	-	-
`G4VEmProcess::PostStepGetPhysicalInteractionLength`	13.516 *s*	**2.301**	0.8% (0.070 / 0.000)	**0.586**	13.3% (0.231)	-
`G4VProcess::SubtractNumberOfInteractionLengthLeft`	12.605 *s*	**3.465**	0.5% (0.052 / 0.000)	-	-	-
`G4ParticleChange::CheckIt`	-	-	-	0.509	-	-
`G4Navigator::ComputeStep`	-	-	-	0.464	-	-
`G4NormalNavigation::ComputeStep`	-	-	-	0.280	22.1% (0.177)	-
`G4SteppingManager::InvokeAlongStepDoItProc`	-	-	-	-	24.9% (0.123)	-
`LArWheelSolid::out_iteration_process`	-	0.820	-	0.126	-	22.6%
`G4UnionSolid::Inside`	-	0.725	-	-	-	15.6%

The selection of functions in table 3.3 allows for general statements about the behavior of the Athena simulation.

Even the largest hotspots still take less than 170 *s*, which is less than 7% of the total execution time, 2653 *s*. This shows that the simulation is a complex composition of many different functions, which are all involved in the inner level of the eventloop.

The `LArWheel*` functions are, together with `MagField::AtlasFieldSvc :: getField` and `sincos`, the only non-Geant4 functions present. Since `sincos` is a math library function, included in `libimf`, it can be considered to be already well optimized. This observation only allows for a division of work, where the focus either lies on the Athena packages involved in `LarWheel*` or `MagField*` operations on the one side, or on Geant4 code on the other side.

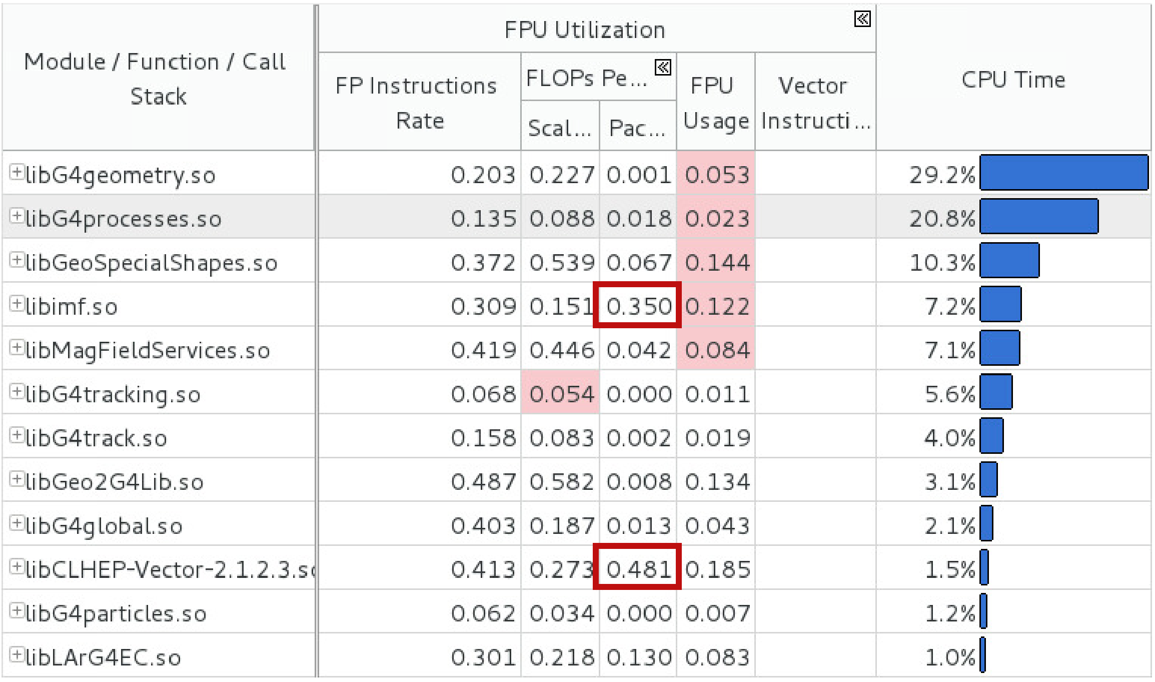
The presented Geant4 functions mostly cover geometry and particle operations. Their behavior could be very dependent on the input data, since different particles could exhibit different on average behavior and typically traverse different sections of the geometry.

Most functions have a quite good CPI rate, suggesting that there is not much potential for improvement, implying that no redundant and unnecessary computations are done. Exceptions are `G4VEmProcess::PostStepGetPhysicalInteractionLength` and `G4VProcess::SubtractNumberOfInteractionsLengthLeft`, which have a relatively high CPI value of 2.301 and 3.465 respectively.

It should be noted, that none of the mentioned G4 function utilizes packed floating point instructions, which becomes also clear by comparing the amount of packed floating point operations per cycle, for each module in figure 3.3. This is suggesting that vectorization could improve the situation. `sincos` is an example of a highly vectorized function with 1.170 packed FLOP.

Most of the G4 functions have a rather high rate of instruction cache misses, e.g. `G4CrossSectionDataStore::GetCrossSection` (0.557). This metric is computed as the ratio of stalls due to instruction fetching problems divided by the total count of CPU clocks spend in this function (minus minor correction term). A rate of ~ 0.5 means that the function spends half of its clockticks with stalls due to ICache misses. Together with the mentioned front end bandwidth issues, this is suggesting that FE stalls in G4 functions are a serious limitation to the simulation performance, as already stated in section 2.4.

Four methods are shown with a bad speculation value of 10% and above, indicating the percentage a function deals with consequences of bad branch predictions. All four are geometry functions computing distances or evaluating if a particle is currently inside a given shape or not. This process involves very unpredictable input data, since their branching behavior is solely dependent on the current trajectory of the particle, which may be considered as random in this case.



Module / Function / Call Stack	FPU Utilization					CPU Time
	FP Instructions Rate	FLOPs Pe...		FPU Usage	Vector Instructi...	
		Scal...	Pac...			
⊞libG4geometry.so	0.203	0.227	0.001	0.053		29.2%
⊞libG4processes.so	0.135	0.088	0.018	0.023		20.8%
⊞libGeoSpecialShapes.so	0.372	0.539	0.067	0.144		10.3%
⊞libimf.so	0.309	0.151	0.350	0.122		7.2%
⊞libMagFieldServices.so	0.419	0.446	0.042	0.084		7.1%
⊞libG4tracking.so	0.068	0.054	0.000	0.011		5.6%
⊞libG4track.so	0.158	0.083	0.002	0.019		4.0%
⊞libGeo2G4Lib.so	0.487	0.582	0.008	0.134		3.1%
⊞libG4global.so	0.403	0.187	0.013	0.043		2.1%
⊞libCLHEP-Vector-2.1.2.3.s	0.413	0.273	0.481	0.185		1.5%
⊞libG4particles.so	0.062	0.034	0.000	0.007		1.2%
⊞libLArG4EC.so	0.301	0.218	0.130	0.083		1.0%

Figure 3.3: FPU utilization per module according to VTune HPC profile. Red squares mark modules with a high count of vectorized functions.

Most of these results are not to be read too literally. Due to the nature of event based sampling, hardware counters can be assigned to wrong functions and statistical fluctuations influence each measurement observably.

To roughly estimate the fluctuations of VTune metrics, 10 identical Athena runs with no additional changes have been profiled. Table 3.4 gives the average of the global metrics from the general exploration summary view, averaged over all runs.

It becomes clear that the front end bound and back end bound metrics are heavily fluctuating and not very reliable in this context. Interestingly they are observed to change anti-proportionally to each other, which is obvious when considering the average of both values summed together and its corresponding standard deviation. This is hinting towards a non deterministic effect that is influencing the caching hierarchy. That influencing factor is not identified, but it could be linked to the fact that access to remote file systems is common during the simulation.

Other metrics, e.g. ICache misses, are more reliable on the global scale. This is somewhat different on a local scale, e.g. for the instruction cache misses of the `G4AtlasRK4::Stepper` method. The average of this metric is 0.212 ± 0.037, which displays with ∼ 17.4% larger fluctuations than the global ICache miss rate of 0.278 ± 0.004 (∼ 1.4%). It is likely that undeterministic effects cause the hardware events to be assigned to different functions in each run, thus local fluctuations occur, while the global count of the corresponding events remains more or less constant. This is also an unproven conjecture.

Table 3.4: Average result and standard deviation of 10 VTune profiles for the reference simulation.

Metric	Average $\pm\sigma$
Clockticks	$8,233,950M \pm 47,220M$
Instruction retired	$7,894,664M \pm 1,134M$
CPI rate	1.043 ± 0.006
Front end bound	**35.18% ± 6.24%**
Back end bound	**34.96% ± 6.32%**
FE + BE	**70.14% ± 0.34%**
I$ misses	0.278 ± 0.004
Bad speculation	3.93% ± 0.08%
Retiring	25.81% ± 0.31%

4 Discussion of Possible Improvements

In this chapter I will discuss all attempted performance improvements. Each section includes the motivation for a proposed change, which is tested and evaluated on a case by case basis.

Based on the analysis in section 3.3.4, two major optimization approaches are identified:

1. Investigate stalls due to front end issues, especially because of ICache misses
2. Investigate the possibility of vectorization

4.1 Vectorization

Vectorization adds a layer of parallelization in the central processing unit and is typically several times faster than the serial equivalent, if it can be applied to critical code sections. These code sections have to be of a certain form, to be formulated in the *single instruction multiple data* (SIMD) pattern, which is common to parallelization on that level. The following subsections cover the global autovectorization of Geant4 and a example problem to discuss a common approach to prepare a code section for the autovectorizer.

4.1.1 Global Vectorization in Geant4

Continuing on the realization that no vectorization is done within Geant4 functions, it is an obvious first step to activate autovectorization and recompile Geant4 to see if anything can be gained that way without any further adjustment.

Approaches in this manner have the inherent assumption that enough code sections are easily vectorizable in their current state. This might not be the case if the program structure is dominated by high abstraction patterns with only few vectorizable loops and dominating access to changing memory data. Simulating the trajectories of particles in the detector geometry might very well fall in the later category.

The VTune HPC profile states, that no AVX instructions are utilized, which are present on the local test machine. This could be due to compatibility reasons of the binary, such that Athena uses SSE2 as a largest common denominator between all involved computing systems. Nevertheless it might be profitable to investigate potential improvements by utilizing higher instruction sets.

The considered Geant version is `4.10.1.patch01.atlas02`, with additional minor patches by the ATLAS project, also used in Athena 20.3.2.1. It can be build using the `cmake` build system by choosing a (separate) build folder and call `cmake /path/to/G4source`.

The cmake configuration is stored in a local file, `CMakeCache.txt`, which can be altered directly, although the usage of configuration tools like `ccmake /path/to/G4source` is recommended to retain a coherent configuration. Alternative compile-/linker flags can be set for the complete project in this way.

GCC autovectorization is implicitly activated with `-O3`, which also includes other aggressive optimization methods. This could alter the simulation results by reordering floating point operations, e.g. with `-ffast-math`, and is therefore potentially dangerous. If the default optimization level, `-O2`, is kept, the autovectorization can be activated through `-ftree-vectorize` and `-ftree-vectorizer-verbose=2`, which increases the verbosity level to include loop specific output, allowing for an in-depth analysis of each code section.

To possibly improve the performance on AVX capable hardware, the `-mavx` flag can be included. This could allow for more advanced usage of the available hardware, which is not possible if the binary is compiled for SSE2 compatibility (default case).

After compilation, shared object files are stored in the subfolder `outputs/library/Linux-g++/` of the build directory and can be copied to the InstallArea of the Athena test session, as described in section 3.2.

Listing 4.1 gives an example selection of log messages of successful and failed autovectorization for `G4PhysicsVector`. The functions located at lines 163 and 129 of that file are `::Store` and `::CopyData`. Both contain easy vectorizable loops that iterate over all elements of a vector. Notes of non vectorized proportions are mainly referring to small preparation sections in front of the loops that access remote object data.

This log file contains over 6.5 million lines and requires automated evaluation with grep and regular expressions for quick extraction of meaningful global data. Otherwise it offers detailed descriptions, if one is searching for information on the autovectorization of specific code sections.

A total of 1989 loops are vectorized throughout Geant4 and external compiled dependencies. This does not tell much about the potential gain, since the vectorized loops can be insignificant in execution time, compared to other sections of not vectorized code.

To test the naive vectorization approach against a reference measurement, Geant4 has been compiled with `-O2 -mavx -ftree-vectorize -ftree-vectorizer`

```
...
.../global/management/src/G4PhysicsVector.cc:163: note:
   vectorized 1 loops in function.
.../global/management/src/G4PhysicsVector.cc:166: note: not
   vectorized: not enough data-refs
 in basic block.
...
.../global/management/src/G4PhysicsVector.cc:129: note:
   vectorized 3 loops in function.
.../global/management/src/G4PhysicsVector.cc:131: note: can’
   t determine dependence between
vec_9(D)->type and this_11(D)->type
...
```

Listing 4.1: Examples of the verbose vectorizer output for `G4PhysicsVector`.

```
acmd.py diff-root BEFORE.hits.pool.root AFTER.hits.pool.root
    --ignore-leaves
        RecoTimingObj_p1_EVNTtoHITS_timings
        RecoTimingObj_p1_HITStoRDO_timings
        RecoTimingObj_p1_RAWtoESD_timings
    --error-mode resilient
```

Listing 4.2: Script to show differences in HITS files

`-verbose=2` and only with `-O2`. All libraries are copied into the local `InstallArea` folder for both versions, to generate a mostly identical setup.

Measured timings are printed in table 4.1 and show that there is no significant improvement. In fact the vectorized version appears to take longer than the reference version, but this cannot be validated with the present data, since the mean values lie within each others error intervals and are not clearly distinguishable.

The resulting HITS file of the AVX and the SSE2 version are identical to the Geant4 version with default compile options. This can be checked by invoking the command in listing 4.2.

Table 4.1 also includes the timings for a `-mSSE2` version, to estimate a possible performance difference between both vector instruction sets. It appears to take a little more time than the AVX version, but it is also only 1.5σ from the reference. For this reason a

Table 4.1: 25 runs for the AVX version, 30 runs for the reference.

	Vectorized AVX	Vectorized SSE2	Reference
Total time	2692.1 ± 18.1 s	2705.4 ± 17.8 s	2679.2 ± 18.2 s
G4 time	2617.2 ± 18.2 s	2625.0 ± 17.3 s	2603.5 ± 18.2 s
average / event	131.7 ± 0.9 s	132.2 ± 0.9 s	131.2 ± 0.9 s

naive vectorization of Geant4 version is not a suited test to compare differences between AVX and SSE2.

I see two reasons why a naive vectorization of Geant4 might not improve the total performance:

1. Memory access is the limiting factor
2. Current design is not suited for autovectorization

Regarding 1, it is possible that most of Geant4 function are already limited by memory operations and data throughput. Vectorization does not really help in this situation since the data access to caches and memory is not improved with packed operations.

The second point refers to the possibility that current design patterns result in bad vectorizable loops. Currently, 21519 loops are not vectorized because of “bad loop form”. In this case it would be most beneficial to identify the largest hotspots of the not vectorized loops and to bring them into a autovectorizable form.

VTune general exploration profiles of the AVX, SSE2 and reference version show that the global amount of packed floating point operations per cycle does not increase significantly. All the changes appear to be in the range of statistical fluctuations. This also indicates that naive activation of autovectorization does not effect global performance and only a small fraction of the Geant4 code is effected.

4.1.2 Vectorization of the Magnetic Field

When looking through the simulation hotspots I encountered the `BFieldCache::getB` method, which is part of `MagField::AtlasFieldSvc::getField`, the ATLAS magnetic field service. A selection of the function is printed in listing 4.3. Since this particular function takes ~ 107 s, $\sim 4\%$ of the total time, I found this to be a good example case to discuss vectorization.

The selected loop is only accessing data of the current iteration i and has therefore no dependencies between iterations. Additionally several multiplications of subsequent array values are required in each step. Both of these characteristics are hinting towards vectorization.

After discussing this example in the ATLAS Simulation Software meeting, one of the participants, Elmar Ritsch, mentioned that he investigated exactly this function before and provided me with his test setup. I am reproducing a selection of his approaches here for a step by step case study. Unfortunately it turns out, that the conversion from `float m_field` to `const short field` is the most critical part of this loop and that vectorization plays only a minor role.

The different considered cases are:

```
/* ... */
for ( int i = 0; i < 3; i++ ) { // z, r, phi components
    const short *field = m_field[i];
    Bzrphi[i] = m_scale*(
           gz*( gr*( gphi*field[0] + fphi*field[1] )
              + fr*( gphi*field[2] + fphi*field[3] ))
              + fz*( gr*( gphi*field[4] + fphi*field[5] )
              + fr*( gphi*field[6] + fphi*field[7] )));
}
/* ... */
```

Listing 4.3: `BFieldCache::getB` reference.

```
for ( unsigned i = 0; i < loop_count; i++ ) {
    field[i] = gphi*m_field[i*2] + fphi*m_field[i*2+1];
}

for ( unsigned i = 0; i < 3; i++ ) {
    // z, r, phi components
    Bzrphi[i] = m_scale*(
        gz*( gr*( field[i*4+0] ) + fr*( field[i*4+1] )) +
        fz*( gr*( field[i*4+2] ) + fr*( field[i*4+3] )));
}
```

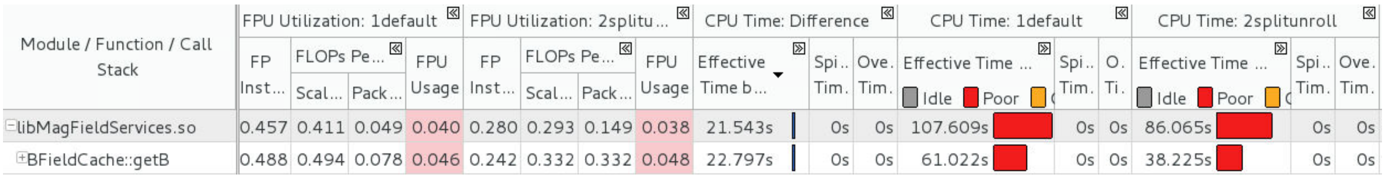
Listing 4.4: `BFieldCache::getB` with split loops.

1. Default code for Athena 20.3.2.1 as a reference
2. Split loop into two loops, such that one can be vectorized
3. Same loop split with additional combined multiplications
4. Vectorized first loop and unrolled `bzrphi` computation
5. `float->short` conversion outside of the loop

A VTune profile of the reference implementation shows that `::getField` utilizes only ~ 0.005 packed and ~ 0.294 scalar floating point operations per cycle. This suggests that vectorization could in fact improve the performance.

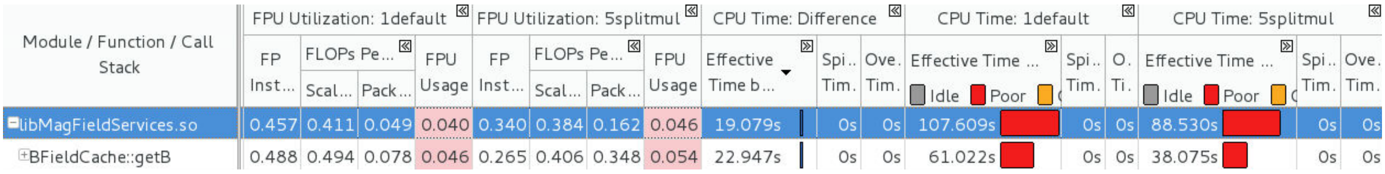
Listing 4.4 shows the second case with a split loop, such that field values can be computed in a vectorized fashion. Compilation with `-ftree-vectorize` successfully vectorizes the first loop and other parts of the package. The second loop is not vectorized due to a small iteration count. It is important to note that the conversion from `float` to `short` is also removed, by changing the type of `field`, in this example.

Figure 4.1 presents the VTune comparison of the reference and the split case for the specific function as well as for the total module. `BFieldCache::getB` now uses 0.332



Module / Function / Call Stack	FPU Utilization: 1default				FPU Utilization: 2splitu…				CPU Time: Difference			CPU Time: 1default			CPU Time: 2splitunroll		
	FP Inst…	FLOPs Pe…		FPU Usage	FP Inst…	FLOPs Pe…		FPU Usage	Effective Time b…	Spi.. Tim.	Ove. Tim.	Effective Time …	Spi.. Tim.	O. Ti.	Effective Time …	Spi.. Tim.	Ove. Tim.
		Scal…	Pack…			Scal…	Pack…					Idle Poor			Idle Poor		
libMagFieldServices.so	0.457	0.411	0.049	0.040	0.280	0.293	0.149	0.038	21.543s	0s	0s	107.609s	0s	0s	86.065s	0s	0s
BFieldCache::getB	0.488	0.494	0.078	0.046	0.242	0.332	0.332	0.048	22.797s	0s	0s	61.022s	0s	0s	38.225s	0s	0s

Figure 4.1: VTune comparison of reference and split case.



Module / Function / Call Stack	FPU Utilization: 1default				FPU Utilization: 5splitmul				CPU Time: Difference			CPU Time: 1default			CPU Time: 5splitmul		
	FP Inst…	FLOPs Pe…		FPU Usage	FP Inst…	FLOPs Pe…		FPU Usage	Effective Time b…	Spi.. Tim.	Ove. Tim.	Effective Time …	Spi.. Tim.	O. Ti.	Effective Time …	Spi.. Tim.	Ove. Tim.
		Scal…	Pack…			Scal…	Pack…					Idle Poor			Idle Poor		
libMagFieldServices.so	0.457	0.411	0.049	0.040	0.340	0.384	0.162	0.046	19.079s	0s	0s	107.609s	0s	0s	88.530s	0s	0s
BFieldCache::getB	0.488	0.494	0.078	0.046	0.265	0.406	0.348	0.054	22.947s	0s	0s	61.022s	0s	0s	38.075s	0s	0s

Figure 4.2: VTune comparison between reference and split & combined multiplication case.

instead of 0.078 packed FLOP per cycle. The time spent in this function is reduced from ~ 61.0s to ~ 38.2s.

Interestingly the global metrics changed significantly. The simulation is 48% (before 40%) stalled due to FE issues and the BE bound decreased in a similar manner from ~ 29.8% to ~ 21.6%. For getB the same change becomes visible by a increasing ICache miss rate from ~ 0.06 to ~ 0.11. These metrics can be misleading, as it is discussed in section 3.3.4. Nevertheless they could be an indication, that this function suffers more from memory issues than missing vectorization, i.e. the removed conversion of field from float to short is the significant improvement here.

Vectorization can have an impact on the instruction decoding procedure, since autovectorization produces different code that can in general behave worse or better in terms of code size and alignment.

To make the second loop more predictable and possibly vectorizable, combined multiplications of the coefficients gz, gr, fz and fr are precomputed. This is shown in listing 4.5 and it can be seen that in each iteration of the second loop 4 multiplications and additions have to be done with sequential data in field.

The log file shows that the first loop is vectorized, but the second one is not due to a small iteration count.

Figure 4.2 is a screenshot of the VTune results of the corresponding profile. The ratio of packed operations as well as the time is roughly the same as in the previous case, including the statistical fluctuation of the VTune collection process.

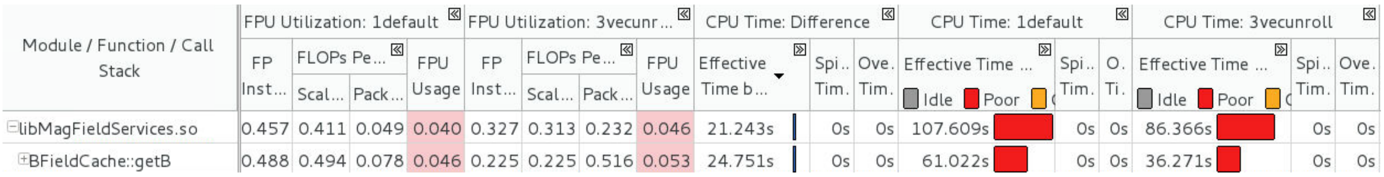
The global metrics change once again. According to the VTune profile this version has ~ 32% FE stalls instead of the usual ~ 40% of the reference. The BE bound metric rises accordingly. Local metrics for that function are very comparable to the previous case. The retired instructions for that section increases from ~ 32% to ~ 49%, while the ICache misses increase from ~ 0.06 to ~ 0.09.

```
const float gzgr = gz*gr;
const float gzfr = gz*fr;
const float fzgr = fz*gr;
const float fzfr = fz*fr;

// autovectorizable loop over 3 components (z, r, phi)
// and 4 corner pairs to compute intermediate result
const unsigned loop_count = 3*4;
float field[loop_count];
for ( unsigned i = 0; i < loop_count; i++ ) {
    field[i] = gphi*m_field[i*2] + fphi*m_field[i*2+1];
}

for ( unsigned i = 0; i < 3; i++ ) { // z, r, phi components
    Bzrphi[i] = m_scale*(
        gzgr*field[i*4+0] + gzfr*field[i*4+1] +
        fzgr*field[i*4+2] + fzfr*field[i*4+3] );
}
```

Listing 4.5: `BFieldCache::getB` with split loops and combined multiplications.



Module / Function / Call Stack	FPU Utilization: 1default				FPU Utilization: 3vecunr...				CPU Time: Difference			CPU Time: 1default			CPU Time: 3vecunroll		
	FP Inst...	FLOPs Pe... Scal...	FLOPs Pe... Pack...	FPU Usage	FP Inst...	FLOPs Pe... Scal...	FLOPs Pe... Pack...	FPU Usage	Effective Time b...	Spi.. Tim.	Ove. Tim.	Effective Time ... Idle Poor	Spi.. Tim.	O. Ti.	Effective Time ... Idle Poor	Spi.. Tim.	Ove. Tim.
⊟libMagFieldServices.so	0.457	0.411	0.049	0.040	0.327	0.313	0.232	0.046	21.243s	0s	0s	107.609s	0s	0s	86.366s	0s	0s
⊞BFieldCache::getB	0.488	0.494	0.078	0.046	0.225	0.225	0.516	0.053	24.751s	0s	0s	61.022s	0s	0s	36.271s	0s	0s

Figure 4.3: VTune comparison of reference and unrolled case.

As a third case, the coefficients of the computation can be considered as constants in the `field` computation (see Listing 4.6). The for loop is again vectorized and the second loop was unrolled by hand, since it has too few iterations to be vectorized.

Loop unrolling can improve the performance, when the loop count is low or if the loop can be partially unrolled, i.e. computing two or more iterations at once. Removing the loop reduces the usual loop overhead, which introduces branching implicitly. It is also possible that unrolled loops are better at utilizing the out of order execution engine.

Figure 4.3 shows the VTune profile results for this case. The fraction of packed FP operations has increased to 0.516, while the time spent in `::getB` is reduced by 2 seconds. The reduction in time might be attributed to the fluctuation of VTune measurements.

Global metrics for FE stalls are reduced and for BE stalls are increased by $\sim 30\%$, which is very hard to explain by such a small change in the code. It is likely that some undetermined influences, e.g. file access, memory organization and OS processes, have a significant impact on these metrics. Other metrics, e.g. execution time per function, are observed to be more consistent.

```
// autovectorizable loop over 3 components (z, r, phi)
// and 4 corner pairs to compute intermediate result
const unsigned loop_count = 3*4;
const float coeff[loop_count] =
            { gzgr, gzfr, fzgr, fzfr,
              gzgr, gzfr, fzgr, fzfr,
              gzgr, gzfr, fzgr, fzfr };
float field[loop_count];
for ( unsigned i = 0; i < loop_count; i++ ) { // line 79
    field[i] = coeff[i] * ( gphi*m_field[i*2] + fphi*m_field
       [i*2+1] );
}

Bzrphi[0] = m_scale * ( field[ 0] + field[ 1] + field[ 2] +
   field[ 3] );
Bzrphi[1] = m_scale * ( field[ 4] + field[ 5] + field[ 6] +
   field[ 7] );
Bzrphi[2] = m_scale * ( field[ 8] + field[ 9] + field[10] +
   field[11] );
```

Listing 4.6: `BFieldCache::getB` with vectorization and unrolled loop.

A repetition of the VTune profile for this case shows that the differences can be large even for the same test case. Front end stalls are now ∼ 47% instead of the ∼ 6% measured in the first profile. The BE bound is reduced by the same proportion. I expect that a parallel process has influenced the memory access behavior during profiling, thus shifting the percentage of stalls towards the BE bound metric.

The above changes reorder the present floating point computations. Due to finite precision of computer arithmetic, reordered floating point operations can lead to different results. Since the underlying "cascade"-like nature of the simulation is very sensible to such small differences, a reordering may result in totally different numerical behavior and requires new validation of physical correctness.

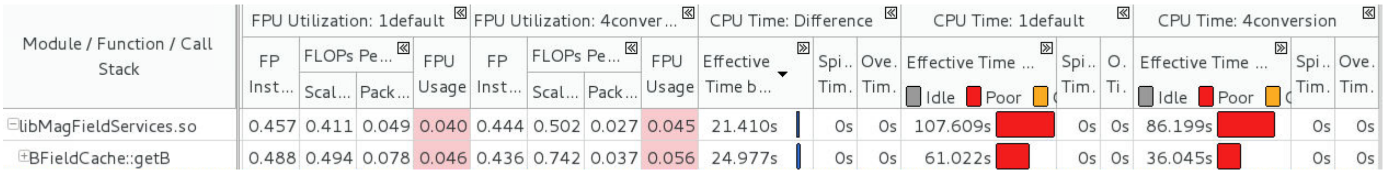
As a test it is sufficient to compare the resulting HITS files with the Athena auxiliary tool `acmd.py diff-root <HITS1> <HITS_reference>`. The vectorization and loop unrolling really produces a different simulation output in this case.

To identify the impact of the conversion between `float` and `short`, a last test case is implemented as given in listing 4.7. This case is identical to the reference case, except for the type change of `field`.

The VTune result in figure 4.4 clearly shows that the time improvement in that function can only be attributed to the removed type conversion. A conversion between floating point and integer types in such a critical part is very expensive. The global metrics are almost identical to the first profile of the unrolled case, which could indicate, that the influencing parallel process was still present during the profiling of this case. This, however,

```
for ( int i = 0; i < 3; i++ ) { // z, r, phi components
    const float *field = m_field[i]; // not short, but float
        !
    Bzrphi[i] = m_scale*(
        gz*( gr*( gphi*field[0] + fphi*field[1] ) +
             fr*( gphi*field[2] + fphi*field[3] )) +
        fz*( gr*( gphi*field[4] + fphi*field[5] ) +
             fr*( gphi*field[6] + fphi*field[7] )));
}
```

Listing 4.7: `BFieldCache::getB` with no conversion of `field`



Module / Function / Call Stack	FPU Utilization: 1default				FPU Utilization: 4conver...				CPU Time: Difference			CPU Time: 1default			CPU Time: 4conversion		
	FP Inst...	FLOPs Pe...		FPU Usage	FP Inst...	FLOPs Pe...		FPU Usage	Effective Time b...	Spi.. Tim.	Ove. Tim.	Effective Time ...	Spi.. Tim.	O. Ti.	Effective Time ...	Spi.. Tim.	Ove. Tim.
		Scal...	Pack...			Scal...	Pack...					▨ Idle 🟥 Poor 🟧			▨ Idle 🟥 Poor 🟧		
⊟libMagFieldServices.so	0.457	0.411	0.049	0.040	0.444	0.502	0.027	0.045	21.410s	0s	0s	107.609s 🟥	0s	0s	86.199s 🟥	0s	0s
⊞BFieldCache::getB	0.488	0.494	0.078	0.046	0.436	0.742	0.037	0.056	24.977s	0s	0s	61.022s 🟥	0s	0s	36.045s 🟥	0s	0s

Figure 4.4: VTune comparison of reference and no-conversion case.

could not be verified.

A smaller proportion of packed FP operations per cycle, 0.037 instead of 0.078 for the reference, shows that vectorization is not critical for this function. It also gives an impression of the magnitude of fluctuations in this metric type, since both versions should have the same amount of packed operations.

The total time spent in this function is again $\sim$ 36 seconds, which also shows that the improved timing of 2 seconds in figure 4.3 is most likely caused by fluctuations. Additionally this change does not influence the numerical computation and thus produces the exact same HITS output file.

The presented example is unfortunately not measurably improved by vectorization, but benefits from the removal of an unnecessary type conversion. It is nevertheless a good show case example to discuss the step by step approach of identifying important parts of the program and shows common techniques, that aim for easy autovectorization during compilation.

4.2 Front End Issues

The performance analysis in section 3.3 revealed that front end stalls are a major problem of the Athena simulation. In this section, PGO and other approaches are discussed to deal with that issue.

Table 4.2: List of test cases for PGO & LTO. All timing data collected with 20 runs, 200 runs for the 300μ. 20* marks different $t\bar{t}$ events, that are obtained by introducing an offset of 50 events to the same event file.

Nr.	PGO	LTO	#Ev.	Trained w/	HITS	Total Time (% of ref)	Time ref
1a	X		20	20 Ev.	✓	2429.7 ± 17.9 s (∼ 91.3%)	2660.7 ± 17.9 s
2a	X		20	50 Ev.	✓	2373.6 ± 10.8 s (∼ 89.2%)	2660.7 ± 17.9 s
3		X	20	20 Ev.	✓	2697.6 ± 13.5 s (∼ 101.4%)	2660.7 ± 17.9 s
4	X	X	20	20 Ev.	✓	2458.2 ± 12.6 s (∼ 92.4%)	2660.7 ± 17.9 s
1b	X		50	20 Ev.	✓	6171 ± 44 s (∼ 91.4%)	6750 ± 45 s
2b	X		50	50 Ev.	✓	6069 ± 22 s (∼ 89.9%)	6750 ± 45 s
1c	X		20*	20 Ev.	✓	2817.3 ± 11.5 s (∼ 90.6%)	3109.5 ± 14.8 s
2c	X		20*	50 Ev.	✓	2823.6 ± 22.6 s (∼ 90.8%)	3109.5 ± 14.8 s
5	X		300μ	20 Ev.	✓	138.3 ± 1.2 s (∼ 94.9%)	145.8 ± 1.3 s

4.2.1 Profile Guided and Link Time Optimization in Geant4

Profile guided optimization and link time optimization are two compiler technologies that can improve the front end performance (see section 2.1.1 and 3.1.3). Since PGO and LTO are effecting the inlining, branch and block order, it is likely that the front end stall rate decreases and the overall performance improves measurably.

In this discussion the Geant version `4.10.1.patch01.atlas02` will be evaluated in many possible combinations of PGO, LTO and training data to investigate the influence on performance metrics as the total execution time, ICache misses and branch misprediction.

Table 4.2 gives a list of all the considered test cases in this discussion. Test 1 and 2 a,b and c are runs with 20, 50 and 20* $t\bar{t}$-events, respectively. The executed binary is trained with 20 and 50 $t\bar{t}$-events, as marked in the table. These tests cover the situation of over adaptation during training and should allow for a better understanding if this is happening for the same type of input data ($t\bar{t}$-events). All timings are given in seconds and compared to the reference measurement, which is the same Geant4 version compiled with default flags.

Tests 3 and 4 evaluate the effects of LTO and its combination with PGO. A last case tests the with $20t\bar{t}$ events trained simulation with single muon events.

The HITS file is checked for equality between the reference and alternative versions of all test cases.

It becomes clear that PGO and LTO do not effect the resulting HITS file of the simulation. This is expected, given that only features like inlining and branch / block orders are applied.

The average time improvement by applying PGO is ∼ 91.1% for the 20 event trained version and ∼ 90.0% for the 50 event trained version, relative to the corresponding reference. The 50 event trained version is therefore measurably faster than the 20 event trained application.

All test cases benefit from PGO, regardless of the applied training data. 20 different $t\bar{t}$ events benefit even better from the 20 event trained binary, than the 20 original events. The fact that the original 20 events also benefit from including additional 30 events indicates that more profiling data could be considered as better. There is no over adaptation observable for these test cases.

The μ events exhibit also a better performance with the PGO binary. It should be noted that this optimization is with $\sim 94.9\%$ of its reference less effective, compared to the $t\bar{t}$ events. This could hint at an over adaptation to the $t\bar{t}$ type events. Another interpretation could be single tracks offer less potential for these kind of optimizations and multi track events benefit from them at a higher rate. This can be tested by comparing the total time spent in the Geant4 section of the simulation, what is also collected by the log file and parse scripts. That comparison yields an improvement of $\frac{58.5\,\mathrm{s}}{63.2\,\mathrm{s}} \approx 92.6\%$, which confirms this assumption to some extent. The first $20t\bar{t}$ test case has a ratio of $\frac{2353.7}{2581.0} \approx 91.2\%$ for the total G4 time and does not really grow as much as the single muon case.

Single muon events have a much smaller count of tracks to be processed. For this reason the execution time in the Geant4 part of the simulation is also much smaller and shrinks to the same magnitude of time spent in the initialization phase. The initialization phase is independent of improvements in Geant4 binaries and therefore has a more dominant impact on the total execution time.

A VTune profile of the $20t\bar{t}$ event trained Geant4 version shows that the overall metrics differ clearly. The percentage of stalls due to front end issues drops from 46.5% to 10.2%, which should not be taken as the best metric to indicate a significant change, since this metric fluctuated a lot in previous profiles. Interesting is the increase of retired instructions from 26.3% to 28.2%. Most fluctuations appeared between the FE and BE bound metric, while “Bad Speculation” and the percentage of retired instruction remained stable. The CPI rate for the complete simulation is reduced from 1.031 to 0.955.

Table 4.3 shows a selection of expensive Geant4 functions with their assigned metrics. The reference value is given in parenthesis for each case. Even when the values are considered to be very uncertain, a general trend of improvement is observable. The VTune profile confirms the assumption that PGO does improve the front end characteristics of a given application. Branch prediction seems to work better and instruction cache misses are reduced.

In general is PGO an effective feature to address front end issues and increase the overall performance of Athena significantly. LTO alone and in combination with PGO seems to have a negative effect and increases the total time for both test cases.

Table 4.3: Function and assigned metrics for a VTune profile of the $20t\bar{t}$ trained Geant4 version. Reference values of a previous profile are given in parenthesis for comparison.

Function	CPI Rate	I$ misses	Branch Misprediction
`G4PhysicsVector::Value`	1.475 (1.659)	0.243 (0.300)	6.1% (7.0%)
`G4Navigator::ComputeStep`	1.161 (1.409)	0.441 (0.445)	2.3% (2.1%)
`-"-::LocateGlobalPointAndSetup`	1.664 (1.753)	0.240 (0.331)	3.4% (4.4%)
`G4PolyconeSide::Inside`	0.515 (0.575)	0.061 (0.048)	0.0% (0.0%)
`G4SteppingManager::Stepping`	1.122 (1.048)	0.362 (0.509)	4.5% (6.5%)
`-"-::DefinePhysicalStepLength`	0.968 (1.182)	0.197 (0.290)	3.7% (10.6%)
`G4AtlasRK4::Stepper`	0.590 (0.628)	0.143 (0.242)	0.0% (0.0%)
`G4PropagatorInField::ComputeStep`	1.607 (1.813)	0.401 (0.433)	0.0% (0.0%)

4.2.2 Inlining parameterized_sin

To investigate the effects of inlining and defining macro functions on front end stalls, the implementation of `LArWheelCalculator::parameterized_sin` is examined more closely (see listing 4.8), which is with ~ 3.6% of the execution time one of the identified hotspots. This function is used in geometry approximations of the liquid argon calorimeter wheel. Given that the sine and cosine are required very often, but are not needed in high precision, their computation can be replaced by a 5th order polynomial with a smaller impact on the computation time. The function also computes the cosine by the relation

$$cos(\alpha) = \sqrt{1 - sin(\alpha)^2} \quad .$$

Since this function is executed very frequently even a tiny improvement might have a measurable impact on the total execution time. For this reason the total execution time is measured for an inlined, a macro and a reference version. A version where each approximated sine and cosine is replaced by `std::sin` and `std::cos` resulted in significantly longer times for the first two events of a run and did not terminate during the third event. For this reason this approach has not been investigated further.

Inlining a function replaces the function call with the function body in place, thus removing the function call overhead. A macro is a preprocessor directive that replaces the function call before compilation with the function body. The macro version of the function is expected to be as fast as the inlined function [Frea] and both versions could influence the total performance.

For the inlined version, the function is annotated with the `inline` keyword and placed in the corresponding header file, as usually done with inlined functions. This keyword does not force the compiler to inline, but should be considered as a suggestion. Usually

```
void LArWheelCalculator::parameterized_sin(
                    const double r,
                    double &sin_a,
                    double &cos_a) const {
    const double r2 = r*r;
    const double r3 = r2*r;
    const double r4 = r2*r2;
#if LARWC_SINCOS_POLY > 4
    const double r5 = r4*r;
#endif
    sin_a = m_sin_parametrization[0]
          + m_sin_parametrization[1]*r
          + m_sin_parametrization[2]*r2
          + m_sin_parametrization[3]*r3
          + m_sin_parametrization[4]*r4
#if LARWC_SINCOS_POLY > 4
          + m_sin_parametrization[5]*r5
#endif
    ;
    cos_a = sqrt(1. - sin_a*sin_a);
}
```

Listing 4.8: LArWheelCalculatorGeometry.cxx - parameterized_sin

Table 4.4: 20 runs for the macro and inlined version, 65 runs for the reference.

	Inline	Macro	Reference
Total time	2631.9 ± 22.7 s	2688.6 ± 21.5 s	2644.3 ± 20.8 s
G4 time	2552.0 ± 22.5 s	2609.4 ± 22.3 s	2565.3 ± 20.9 s
average / event	128.5 ± 1.2 s	131.6 ± 1.1 s	129.2 ± 1.1 s

the compiler is very good at deciding if a function should be inlined or not. It can be forced with the `__attribute__((always_inline))` annotation, which is is done here.
`-Winline` can be used to activate warnings, if a function could not be inlined during compilation, what is not observed for the inlined `parameterized_sin`.

Listing 4.9 shows the macro implementation that still allows for a selection of the 4th or 5th order polynomial during compilation. A 5th order polynomial is the default.

VTune profiles of the inlined and reference version exhibit too small differences, which are most likely to be covered by the statistical fluctuations of the metrics. The macro version cannot be profiled with VTune at all, since the code for that specific function is placed everywhere, where it is called, thus the time spent “in” this function is distributed between all callers. For that reason the total execution time is considered, given in table 4.4.

```
#if LARWC_SINCOS_POLY > 4
#define PSIN(r,sin_a,cos_a)      \
    const double r2 = r*r;       \
    const double r3 = r*r*r;     \
    const double r4 = r*r*r*r;   \
    const double r5 = r*r*r*r*r; \
    sin_a = lwc()->m_sin_parametrization[0]   \
        + lwc()->m_sin_parametrization[1]*r   \
        + lwc()->m_sin_parametrization[2]*r2  \
        + lwc()->m_sin_parametrization[3]*r3  \
        + lwc()->m_sin_parametrization[4]*r4  \
        + lwc()->m_sin_parametrization[5]*r5; \
    cos_a = sqrt(1. - sin_a*sin_a);
#else
#define PSIN(r,sin_a,cos_a)    \
    const double r2 = r*r;     \
    const double r3 = r*r*r;   \
    const double r4 = r*r*r*r; \
    sin_a = lwc()->m_sin_parametrization[0];  \
        + lwc()->m_sin_parametrization[1]*r;  \
        + lwc()->m_sin_parametrization[2]*r2; \
        + lwc()->m_sin_parametrization[3]*r3; \
        + lwc()->m_sin_parametrization[4]*r4; \
    cos_a = sqrt(1. - sin_a*sin_a);
#endif
```

Listing 4.9: Macro version of parameterized_sin

It can be seen, that the macro implementation is measurably slower than the original one. The inlined function appears to be faster than the reference, but uncertainties of the measurements do not allow for a definitive judgement. Front end stalls are already a present issue and inlining can both improve or worsen the situation. While inlining improves code locality, i.e. less fragmented hot code sections, it also increases the total code size, which again can induce ICache misses. The behavior has to be evaluated for each case uniquely through profiling. In this case a negative impact is observed.

4.3 Different Optimization Attempts

This section holds minor optimization attempts that cover the discussion of dependency breaking in `parameterized_sin` and the investigation of the `G4AtlasRK4::Stepper` method.

4.3.1 Dependency Breaking in parameterized_sin

The function `LArWheelCalculator::parameterized_sin`, listing 4.8, as discussed in the previous section 4.2.2, is also an interesting example to discuss dependency chains. The computation of the 5th order polynomial is done in one C++ statement and it is interesting to test, if a dependency chain is present and if intermediate variables can improve the behavior of this function.

The effect of dependency chains is discussed in section 2.1.

In listing 4.8, this dependency chain could be split by adding intermediate variables for each term of the polynomial and possibly compute the exponents of `r` only using `r`. I have implemented this version, and a second, where the dependency between the computation of `r2,r3` etc., is maintained.

The resulting times and metrics of the changes can be seen in table 4.5. The total and G4 time are obtained by the log-parse script for 20 runs with 20 events each. The values for the VTune metrics are also produced with 20 events and should not be taken too literally, due to its sample based nature.

The total time clearly shows that these changes are in fact no improvement to the code and increase the computation time instead. Introduced variables probably increase the memory footprint of this code section and prevent the compiler from doing optimizations which are otherwise applicable. It is likely that the compiler is capable to detect dependency chains on this scale and introduces changes, like the above mentioned only increase the complexity with no potential gain at all.

Table 4.5: Timings for `parameterized_sin` with changes vs. reference. No timings taken for the second version, since it is already clear that no improvement is made.

	Dep. breaking	Dep. breaking in r	Reference
Total time	2761.5 ± 12.9 s	-	2644.3 ± 20.8 s
G4 time	2682.9 ± 13.7 s	-	2565.3 ± 20.9 s
Clockticks	$494,114M$	$470,590M$	$462,803M$
Instructions Retired	$526,767M$	$491,857M$	$492,239M$
CPI Rate	0.938	0.957	0.940

4.3.2 G4AtlasRK4

`G4AtlasRK4::Stepper`, given in listing A.2, appendix A.2, is a special implementation of the stepper function, used during the Geant4 stepping process in the ATLAS Athena simulation. This method implements the influence of a magnetic field on a charged particle trajectory. It is a special implementation of a 4th order Runge Kutta integrator, that evaluates the magnetic field at the 2nd and 4th intermediate step.

The simulation spends $\sim 1\%$ of the total time in this function and a huge improvement by any changes is not to be expected, since it already has a quite good CPI rate of ~ 0.6, executing almost two instructions per cycle.

Given that this function was investigated in previous optimization efforts, see section 2.4, and because of its essential position in the simulation – being executed for every step of a charged particle – it is still interesting for further investigation. The call count for this function could easily exceed the millions, as is shown in section 3.3.

The present Runge Kutta implementation requires certain fractions of the step length S, which is computed as $\frac{S}{2} = 0.5 * S$, $\frac{S}{4} = 0.25 * S$. This way an expensive division operation can be replaced with a less costly multiplication, while maintaining the exact same floating point operation. That is not possible for $\frac{S}{6}$, which has to involve a explicit division, given that no exact finite representation of the inverse exists. A naive replacement with the inexact multiplication $\frac{S}{6} \approx 0.1666666666666666 * S$ resulted in a different HITS file, thus changing the results of the simulation, while not giving a measurably speed up. The digits of precision are chosen to achieve the closest 64 bit (double) representation of the fraction $\frac{S}{6.0}$.

Another attempt to influence the computation on this level, is to define the fraction $\frac{1}{6}$ as a static variable of this function. This way the division has to be done only once and $\frac{S}{6}$ can be implemented using a multiplication, which should behave identical to the naive implementation above.

Listings 4.10 and 4.11 show two example functions and their compiled assembler code, which has been obtained with an online compiler for gcc version 4.4.7 [God]. Even for this

```
double Stepper (const double S) {
  static double os = 1/6.0;
  double S6 = S * os;
  return S6;
}
double Stepper2 (const double S) {
  double S6 = S *(1/6.);
  return S6;
}
```

Listing 4.10: Example implementation in C++

```
Stepper(double):
    pushq   %rbp
    movq    %rsp, %rbp
    movsd   %xmm0, -24(%rbp)
    movsd   Stepper(double)::os(%rip), %xmm0
    movsd   -24(%rbp), %xmm1
    mulsd   %xmm1, %xmm0
    movsd   %xmm0, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    %rax, -32(%rbp)
    movsd   -32(%rbp), %xmm0
    leave
    ret
Stepper2(double):
    pushq   %rbp
    movq    %rsp, %rbp
    movsd   %xmm0, -24(%rbp)
    movsd   -24(%rbp), %xmm1
    movsd   .LC0(%rip), %xmm0
    mulsd   %xmm1, %xmm0
    movsd   %xmm0, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    %rax, -32(%rbp)
    movsd   -32(%rbp), %xmm0
    leave
    ret
Stepper(double)::os:
.LC0:
    .long   1431655765
    .long   1069897045
```

Listing 4.11: Example implementation as compiled assembler

older gcc version, the two implementations behave exactly the same. For this reason it can be assumed, that such optimizations are already applied by the compiler and readability should be valued above similar optimization attempts in the high level language.

Two changes addressing other sections of the Runge Kutta implementation involves reformulating certain sections to allow for packed/vectorized instructions and fusing certain computation of intermediate values, that are done several times ($\sim 2-5$ times), into an intermediate variable. Both attempts result in longer execution times.

The impression of an already well performing function proves to be true and no significant improvement can be done with small structural changes to this section. Replacing a division by a multiplication with a static variable of the fraction could influence the performance positively, but such a change has to be done in many frequent called routines to show its effect on a large scale.

5 Conclusion

To summarize the improvement, a final performance analysis is done, where the VTune profile and total execution time of the improved Athena version is compared to the reference version.

This evaluation is assessed afterwards to motivate and discuss the potential usage of these methods in future Athena releases. A short section on the outlook and potential follow up projects embeds this thesis further in the context of the Athena Simulation.

5.1 Performance Analysis After Changes

The performance analysis of the reference version in section 3.3 is identifying front end stalls and missing vectorization as the two major points of interest for this project. Given that PGO is the only considered optimization feature that successfully improves the total execution time, the following summarizing performance analysis is solely based on PGO (4.2.1). It improves the total execution time by roughly 8.9% to 10.0%, depending on the amount of training data, as it can be seen in table 5.1.

The reference version exhibits a FLOP rate of 853 MFLOPs from Tobias Wegners *perf* estimation and 860 MFLOPs in previous VTune profiles (see section 3.3.2). A VTune profile of the improved simulation shows a value of 970 MFLOPs, which appears to be a significant improvement.

A HPC profile of the PGO variant shows a similar low rate of packed instruction utilization within Geant4 libraries, as observed in reference profiles. Nevertheless, the rate of scalar floating point instructions has increased in comparison to the reference profile, which is in agreement with the increased global FLOP rate.

Table 5.1: Timing results for the 20 $t\bar{t}$/50 $t\bar{t}$ events trained PGO version, executed with 20 events, averaged over 20 runs.

	20 $t\bar{t}$	50 $t\bar{t}$	**Reference**
Total time	2429.7 ± 17.9 s	2373.6 ± 10.8 s	2660.7 ± 17.9 s
G4 time	2353.7 ± 18.6 s	2297.0 ± 11.3 s	2581.0 ± 17.6 s
average per event	118.5 ± 0.9 s	115.7 ± 0.6 s	130.0 ± 0.9 s

The general exploration profile took 2452.5 s, which is ∼ 200 s faster than the reference version, discussed in section 3.3.4. Table 4.3 shows a selection of Geant4 functions and their CPI rate, ICache misses and occurrences of branch mispredictions for the improved and reference version. These values substantiate the impression that PGO achieves the better execution times by improving the front end performance.

5.2 Optimization Assessment and Outlook

Previous optimization efforts and an ongoing development process left Athena in an already high performance state. Simple errors, that cause costly performance penalties and happen in every development process, are likely to be discovered already.

I expect that any performance optimization at this state is either obtained through new, or unused, technology or through time-consuming refactoring to better incorporate modern software architecture concepts, e.g. mutli-threading and vectorization.

The enhancement of applying PGO to Geant4 appears to be significant enough that its application to the Athena project should be further investigated. With PGO, a selection of functions showed much better CPI rates and fewer ICache misses. This technology is therefore one method to address the front end stalls, which are observed in the reference case.

The dual compilation and training process of PGO introduces a non negligible overhead to the build process. It involves at least 2× the compilation time and one additional simulation run. If the simulation is executed on a much larger scale, compared to its build time, it could still be viable to introduce that overhead and benefit from better adapted binaries. This is especially true when PGO is only applied to external dependencies, e.g. Geant4, which are not recompiled with each Athena release, but only each time a new Geant4 version is implemented.

Another potential problem is the selection of the best test data that is used during training. It has to represent the most common simulations, but should also improve the performance for a very broad and generic usage. Given that worse performance for uncommon use cases is not to be expected, this is only another parameter to optimize the simulation as much as possible.

This leads to a proposition for follow up work to this thesis project, which is the investigation of the PGO training data. This study should answer questions of the best type and number of events that produce the fastest binaries for a diverse set of application data (different primary particles). In this thesis $50t\bar{t}$ provided the best results, but it could be also worthwhile to optimize the build process as well and reduce the number events, used for training, as much as possible.

Another topic that could lead to further insight is a more extensive comparison of

AVX and SSE2 instruction sets with different code sections, that are more suitable for vectorization. A more comprehensive study of LTO, inlining of critical functions and bad speculation could also lead to better performing code.

Currently there are new versions of Athena and Geant under development, that incorporate multi-threading (AthenaMT [Ste+16]) and vectorization (GeantV [Ama+15]).

Multi-threading Athena could introduce better memory management between simulation jobs and reduce the overall memory footprint.

GeantV could improve the simulation performance per job by refactoring Geant code to better implement vectorization and make use of SIMD capabilities in modern processors.

5.3 Summary

The primary goals of this thesis are to analyze the CPU performance of the ATLAS simulation framework Athena and investigate potential optimizations. Because of the six month time frame of this project, these changes are focusing on structural, local, changes to the code and do not involve physical or architectural modifications that require intensive physics validation or larger development live cycles.

The performance analysis reveals most promising gains by examining vectorization and reasons for front end stalls due to ICache misses. This analysis also shows that the performance is already in a good state, due to previous optimization efforts and an ongoing development process.

Intel VTune Amplifier, an event based sampling profiler, is used to analyze the application on a detailed level, but is also prone to large statistical fluctuations. These unreliabilities are attributed to the large complex structure of the Athena simulation and to the fact that the “silent” test system is never behaving fully deterministically and can influence the measurements significantly.

The most reliable metric for improvement is the total execution time, that is collected for each test setup N times to get an average total time. This method is only accurate enough to reveal changes that differ by at least $\sim 1\%$ from the reference data.

Profile guided optimization is the most prominent investigated method, that allows for an improvement of 8.9% / 10.0%. It is technically easy to implement, but introduces an overhead during the build process, that might not be negligible. Additionally an instrumented version of the application has to be trained in a way to represent the most common use case of the simulation, which potentially is not trivial to decide. Nevertheless does PGO offer a good improvement and should be included in future Athena development cycles. Minor changes to various components of Athena and Geant4 are discussed that barely influence the total performance at all.

Bibliography

[Ago+03] S. Agostinelli et al. "GEANT4: A Simulation toolkit". In: *Nucl. Instrum. Meth.* A506 (2003), pp. 250–303. DOI: 10.1016/S0168-9002(03)01368-8.

[All06] J Allison. "Geant4 developments and applications". In: *IEEE Trans. Nucl. Sci.* 53 (2006), p. 270. URL: http://cds.cern.ch/record/1035669.

[Ama+15] G Amadio et al. "The GeantV project: preparing the future of simulation". In: *J. Phys.: Conf. Ser* 664.FERMILAB-CONF-15-598-CD. 7 (2015), 072006. 8 p. URL: https://cds.cern.ch/record/2134619.

[Apo+10] J. Apostolakis et al. *Final Report of the ATLAS Detector Simulation Performance Assessment Group.* Tech. rep. 2010-03. URL: http://lcgapp.cern.ch/project/docs/reportATLASDetectorSimulationPerformance2010.pdf.

[Bar+01] G. Barrand et al. "GAUDI - A software architecture and framework for building HEP data processing applications". In: *Comput. Phys. Commun.* 140 (2001), pp. 45–55. DOI: 10.1016/S0010-4655(01)00254-5.

[Bun+10] P Buncic et al. "CernVM – a virtual software appliance for LHC applications". In: *J. Phys.: Conf. Ser.* 219 (2010), p. 042003. URL: https://cds.cern.ch/record/1269671.

[CM01] M. Cattaneo and P. Maley. *Gaudi; LHCb Data Processing Applications Framework; Users Guide.* 9th ed. 2001. URL: http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/GUG/GUG.pdf (visited on 2016-05-10).

[Col08] The ATLAS Collaboration. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *JINST* 3 (2008), S08003. DOI: 10.1088/1748-0221/3/08/S08003.

[Col10] The ATLAS Collaboration. "The ATLAS Simulation Infrastructure". In: *Eur.Phys.J.* C70 (2010), pp. 823–874. arXiv: 1005.4568 [physics.ins-det].

[Cora] Intel Corporation. *Collection Control API.* URL: https://software.intel.com/en-us/node/596660 (visited on 2016-08-04).

[Corb] Intel Corporation. *Hardware Event-based Sampling Collection.* URL: `https://software.intel.com/en-us/node/544067` (visited on 2016-05-19).

[Corc] Intel Corporation. *ICache Misses.* URL: `https://software.intel.com/en-us/node/544435` (visited on 2016-08-05).

[Cor16] Intel Corporation. *Intel 64 and IA-32, Architectures Optimization Reference Manual.* 2016-01. URL: `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[DF03] A. Das and T. Ferbel. *Introduction to Nuclear and Particle Physics.* 2nd ed. Singaproe 596224: World Scientific, 2003.

[Dre14] Eric Drexler. *Elementary Particle Interactions in the Standard Model.* Image File. 2014. URL: `https://en.wikipedia.org/wiki/File:Elementary_particle_interactions_in_the_Standard_Model.png` (visited on 2016-05-13).

[Frea] Inc. Free Software Foundation. *An Inline Function Is As Fast As a Macro.* URL: `https://gcc.gnu.org/onlinedocs/gcc-4.0.1/gcc/Inline.html` (visited on 2016-08-15).

[Freb] Inc. Free Software Foundation. *GCC Wiki: Link Time Optimization.* URL: `https://gcc.gnu.org/wiki/LinkTimeOptimization` (visited on 2016-08-16).

[Frec] Inc. Free Software Foundation. *Options That Control Optimization.* URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (visited on 2016-08-16).

[God] Matt Godbolt. *GCC Explorer.* URL: `https://gcc.godbolt.org` (visited on 2016-08-20).

[Goo] Google. *Google C++ Style Guide.* URL: `https://google.github.io/styleguide/cppguide.html` (visited on 2016-08-08).

[Gri08] David Griffiths. *Introduction to Elementary Particles.* 2nd ed. Weinheim: Wiley-VCH, 2008.

[Mar15] Jackson Marusarz. *Understanding How General Exploration Works in Intel® VTune™ Amplifier XE.* 2015-02-09. URL: `https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe` (visited on 2016-05-19).

[Rim+08a] A Rimoldi et al. *Final Report of the Simulation Optimization Task Force*. Tech. rep. ATL-SOFT-PUB-2008-004. ATL-COM-SOFT-2008-023. Follow up to note ATL-SOFT-PUB-2008-002. Geneva: CERN, 2008-12. URL: http://cds.cern.ch/record/1151298.

[Rim+08b] A Rimoldi et al. *First Report of the Simulation Optimization Group*. Tech. rep. ATL-SOFT-PUB-2008-002. ATL-COM-SOFT-2008-004. Geneva: CERN, 2008-04. URL: http://cds.cern.ch/record/1097789.

[Ste+16] Graeme Stewart et al. "Multi-threaded Software Framework Development for the ATLAS Experiment". In: (2016-01). URL: https://cds.cern.ch/record/2120835.

[Weg16] Tobias Wegner. "Laufzeitoptimierung der Detektorsimulation des ATLAS Experiments am CERN". Thesis of simultaneous project, unreleased during creation of bibilography. MA thesis. HS Niederrhein, 2016.

[Wic+14] Baptiste Wicht et al. "Hardware Counted Profile-Guided Optimization". In: (2014). arXiv: 1411.6361.

[Wik] Wikipedia. *Unbiased Estimation of Standard Deviation*. URL: https://en.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation (visited on 2016-08-05).

List of Figures

List of Tables

List of Listings

A Code Examples

A.1 jobOptions File

An example jobOptions file is shown in listing A.1. This python script is used to define how an Athena job is executed. This includes options for services and algorithms, as well as the input and output data. An important concept is the algorithm sequence, which is executed on all events during the Eventloop. This sequence is initialized in line 3, while line 4 sets the log level of the MassageSvc. To gather timing information the log level should be set to "INFO".

The following lines define detector flags and condition tags, describing the state of the detector. Line 18 and following set the input file for previously generated events and the output of the HITS file, as well as the maximum number of events to be used in the Eventloop. Additional simulation flags control additional simulation data, e.g. the physics list and random number generators. In the end, at line 40 and 41, the Geant4 Algorithm is inserted in the algorithm sequence.

The jobOptions file collects all external controllable configurations for a simulation job. For this reason listing A.1 should only be considered as an example, which is very close to the discussed benchmark problem in this thesis.

Lines 47 to 49 show the additional implementation of the VTune CCAPI algorithm, that controls the collection process to only gather data during the eventloop.

```
## Job options file for Geant4 ATLAS detector simulations
from AthenaCommon.AlgSequence import AlgSequence
topSeq = AlgSequence()
ServiceMgr.MessageSvc.OutputLevel = WARNING #FATAL, INFO, DEBUG

## Detector flags
from AthenaCommon.DetFlags import DetFlags
DetFlags.ID_setOn()
DetFlags.Calo_setOn()
DetFlags.Muon_setOn()
DetFlags.Truth_setOn()

## Global conditions tag
from AthenaCommon.GlobalFlags import jobproperties
jobproperties.Global.ConditionsTag = "OFLCOND-RUN12-SDR-21"

## AthenaCommon flags
from AthenaCommon.AthenaCommonFlags import athenaCommonFlags
# remote AFS event input file:
# athenaCommonFlags.PoolEvgenInput = [’/afs/cern.ch/atlas/
#     offline/ProdData/16.6.X/16.6.7.Y/ttbar_muplusjets-
#     pythia6-7000.evgen.pool.root’]
# local copy:
athenaCommonFlags.PoolEvgenInput = [’../ttbar_muplusjets-pythia6-7000.evgen.pool.root’]
athenaCommonFlags.PoolHitsOutput = "atlasG4.hits.pool.root"
athenaCommonFlags.EvtMax = 50
#athenaCommonFlags.SkipEvents = 50

## Simulation flags
from G4AtlasApps.SimFlags import simFlags
simFlags.load_atlas_flags()
simFlags.RandomSvc = ’AtDSFMTGenSvc’
simFlags.SimLayout=’ATLAS-R2-2015-03-01-00_VALIDATION’
simFlags.PhysicsList = ’FTFP_BERT’
simFlags.EventFilter.set_On()
simFlags.CalibrationRun.set_Off()

# Frozen showers in the FCAL
simFlags.LArParameterization=3

# Use the new magnetic field service
simFlags.MagneticField.set_Value_and_Lock(’AtlasFieldSvc’)

from G4AtlasApps.PyG4Atlas import PyG4AtlasAlg
topSeq += PyG4AtlasAlg()
from VTune_CCAPI.VTune_CCAPIConf import CCAPI_Alg
topSeq += CCAPI_Alg("VTune_CCAPI")
topSeq.VTune_CCAPI.resumeAtBeginRun = True
print topSeq
```

Listing A.1: Example jobOptions file.

A.2 G4AtlasRK4

The `G4AtlasRK4::Stepper` function is given in listing A.2. It is a special Geant4 implementation of the integrator that estimates the trajectory of a charged particle in a magnetic field. Only two evaluations of the magnetic field are required in the intermediate points 2 and 4.

```
void G4AtlasRK4::Stepper
(const G4double P[],const G4double dPdS[],G4double Step,
   G4double Po[], G4double Err[]) {
  G4double R[3] = {   P[0],    P[1] ,     P[2]};
  G4double A[3] = {dPdS[0], dPdS[1], dPdS[2]};

  m_iPoint[0]=P[0]; m_iPoint[1]=P[1]; m_iPoint[2]=P[2];
  G4double S  =     Step   ;
  G4double S5 =  .5*Step   ;
  G4double S4 = .25*Step   ;
  G4double S6 =     Step/6.;

  // John A  added, in order to emulate effect of call to
     changed/derived RHS
  m_mom   = sqrt(P[3]*P[3]+P[4]*P[4]+P[5]*P[5]);
  m_imom  = 1./m_mom;
  m_cof   = m_fEq->FCof()*m_imom;

  // Point 1
  G4double K1[3] = {m_imom*dPdS[3],m_imom*dPdS[4],m_imom*
     dPdS[5]};

  // Point2
  G4double p[4] = {R[0]+S5*(A[0]+S4*K1[0]),
                   R[1]+S5*(A[1]+S4*K1[1]),
                   R[2]+S5*(A[2]+S4*K1[2]),
                   P[7]                   }; getField(p);
  G4double A2[3] = {A[0]+S5*K1[0],A[1]+S5*K1[1],A[2]+S5*K1
     [2]};
  G4double K2[3] = {(A2[1]*m_field[2]-A2[2]*m_field[1])*
     m_cof,
                    (A2[2]*m_field[0]-A2[0]*m_field[2])*
                       m_cof,
                    (A2[0]*m_field[1]-A2[1]*m_field[0])*
                       m_cof};

  m_mPoint[0]=p[0]; m_mPoint[1]=p[1]; m_mPoint[2]=p[2];
  // Point 3 with the same magnetic field
  G4double A3[3] = {A[0]+S5*K2[0],A[1]+S5*K2[1],A[2]+S5*K2
     [2]};
  G4double K3[3] = {(A3[1]*m_field[2]-A3[2]*m_field[1])*
     m_cof,
                    (A3[2]*m_field[0]-A3[0]*m_field[2])*
```

```
                       m_cof,
                    (A3[0]*m_field[1]-A3[1]*m_field[0])*
                       m_cof};

  // Point 4
  p[0] = R[0]+S*(A[0]+S5*K3[0]);
  p[1] = R[1]+S*(A[1]+S5*K3[1]);
  p[2] = R[2]+S*(A[2]+S5*K3[2]);            getField(p);
  G4double A4[3] = {A[0]+S*K3[0],A[1]+S*K3[1],A[2]+S*K3[2]};
  G4double K4[3] = {(A4[1]*m_field[2]-A4[2]*m_field[1])*
     m_cof,
                    (A4[2]*m_field[0]-A4[0]*m_field[2])*
                       m_cof,
                    (A4[0]*m_field[1]-A4[1]*m_field[0])*
                       m_cof};

  // New position
  Po[0] = R[0]+S*(A[0]+S6*(K1[0]+K2[0]+K3[0]));
  Po[1] = R[1]+S*(A[1]+S6*(K1[1]+K2[1]+K3[1]));
  Po[2] = R[2]+S*(A[2]+S6*(K1[2]+K2[2]+K3[2]));

  m_fPoint[0]=Po[0]; m_fPoint[1]=Po[1]; m_fPoint[2]=Po[2];

  // New direction
  Po[3] = A[0]+S6*(K1[0]+K4[0]+2.*(K2[0]+K3[0]));
  Po[4] = A[1]+S6*(K1[1]+K4[1]+2.*(K2[1]+K3[1]));
  Po[5] = A[2]+S6*(K1[2]+K4[2]+2.*(K2[2]+K3[2]));

  // Errors
  Err[3] = S*fabs(K1[0]-K2[0]-K3[0]+K4[0]);
  Err[4] = S*fabs(K1[1]-K2[1]-K3[1]+K4[1]);
  Err[5] = S*fabs(K1[2]-K2[2]-K3[2]+K4[2]);
  Err[0] = S*Err[3]                       ;
  Err[1] = S*Err[4]                       ;
  Err[2] = S*Err[5]                       ;
  Err[3]*= m_mom                          ;
  Err[4]*= m_mom                          ;
  Err[5]*= m_mom                          ;

  // Normalize momentum
  G4double N = m_mom/sqrt(Po[3]*Po[3]+Po[4]*Po[4]+Po[5]*Po
     [5]);
  Po[3]*=N    ;
  Po[4]*=N    ;
  Po[5]*=N    ;
  Po[7] =P[7];
}
```

Listing A.2: `G4AtlasRK4::Stepper`