Optimization and validation of Geant4 detector simulation software for the ATLAS experiment at the LHC

Einar Elén

THESIS SUBMITTED FOR THE DEGREE OF MASTER OF SCIENCE
PROJECT DURATION: 12 MONTHS

SUPERVISED BY OXANA SMIRNOVA AND CATERINA MARCON
WITH ADDITIONAL SUPERVISION FROM THE CERN IT DEPARTMENT
BY DAVID SMITH AND MARKUS SCHULZ

Division of Particle Physics
Department of Physics
Faculty of Science
Lund University
May 2020

Abstract

Simulations of large detectors such as the ATLAS detector at the LHC (Large Hadron Collider) are compute-intensive projects, and are according to current internal estimates expected to be consuming 40 % of the total ATLAS CPU resources in 2020, creating a need for software optimization strategies. We investigate build configurations of the Geant4 simulation software to find if a) a performance improvement can be made by switching build configuration and b) if such build configuration switches produce the same physics results. We performed measurements of CPU time per event and energy deposited per event using both the ATLAS detector geometry and the CMS detector geometry on two types of hardware. We find potential performance improvements using static linking of the Geant4 libraries of up to ∼19 % using unsafe build options and up to ∼10 % using safe build options. We observe three sources of differences in the average energy deposition per event of order ∼0.1 %. These sources include the expected unsafe optimization methods but also differences between hardware platforms and from changing the compiler that is used to build the simulation.

Contents

Glossary

Athena A software framework for most of ATLAS computing.

ATLAS *A Toroidal LHC ApparatuS:* One of the four large experiments at the LHC. A general purpose particle physics detector. See Section 2.1.4

Aurora Computing cluster at LUNARC.

Bremsstrahlung Breaking radiation. Emission of photons when a charged particle is accelerated by the electric field of nuclei in a medium.

CERN *Organisation européenne pour la recherche nucléaire/European Organization for Nuclear Research:* European subatomic physics laboratory.

Clang A free and open source C and C++ compiler frontend provided by the LLVM project.Specifically for this project, this refers to the C++ compiler, sometimes referred to as clang++.

CLHEP *A Class Library for High Energy Physics:* A collection of common utility classes for high energy physics programming in C++.

CMS *Compact Muon Solenoid:* One of the four large experiments at the LHC. A general purpose particle physics detector.

Configuration A combination of five parameters used to describe one method of compiling and linking the benchmark and validation code used in this project. The parameters and how they are labeled in this thesis can be found in 3.4.

EM Electromagnetic

EMEC *Electromagnetic Endcap Calorimeter:* An electromagnetic calorimeter in ATLAS.

FCal *Forward Calorimeter:* One of the outer calorimeters of ATLAS.

GCC *Gnu Compiler Collection:* A collection of free and open source compilers from the GNU project. Specifically for this project, this refers to the C++ compiler, sometimes referred to as G++.

GDML *Geometry Description Markup Language:* An XML-based markup language for describing detector geometries for Geant4.

Geant4 *GEometry and Tracking:* A simulation software toolkit for subatomic physics.

GOT Global Offsets Table

HEC *Hadronic Encap Calorimeter:* A part of the outer calorimeters of ATLAS.

HepExpMT A standalone benchmarking application for high energy physics simulations using Geant4.

HS06 *HepSpec06*: A standard unit for computing performance used in high energy physics.

ICC *Intel C++ Compiler:* A C and C++ compiler frontend provided by Intel. Specifically, in this project, this refers to the C++ compiler, sometimes referred to as icpc.

IEEE Institute of Electrical and Electronics Engineers

Implementation-defined behavior Parts of the C++ programming language for which the behavior of a program depends on the concrete implementation of the language. Guaranteed to be consistent throughout the program and must be documented and defined by the implementation. One example is the size of an integer.

ISO International Organization for Standardization

LHC *Large Hadron Collider:* A particle accelerator located at CERN.

LTO *Link-Time Optimization:* An optimization strategy where all optimization is delayed until link-time providing additional optimization opportunities at the cost of build time. Sometimes referred to as interprocedural optimization or whole program optimization.

LUNARC Lund university's center for scientific and technical computing.

Monte Carlo A class of methods for solving numerical problems by utilizing random number generation. See Section 2.2.1.

ODR *One definition rule:* A rule in the C++ programming language stating that a valid program contains no more than one definition of e.g., a function .

PIC Position Independent Code

PLT Procedure Linkage Table

ROOT A data analysis framework commonly used in high energy physics.

SCT *Semiconductor Trackers/Silicon Microstrip Trackers:* A part of the inner detector of ATLAS.

Tile Calorimeters A central hadronic calorimeter in ATLAS.

TRT *Transition Radiation Tracker:* A part of the inner detector of ATLAS.

UB *Undefined/Unspecified behavior:* Parts of the C++ programming language for which the international standard imposes no requirements (undefined) or for which the behavior of a program depends on the concrete implementation of the language (unspecified). Undefined and unspecified behavior is often used interchangeably with each other and implementation defined behavior. For example, accessing an array out of bounds is undefined behavour, while the order of evaluation of function arguments is unspecified.

1 Introduction

1.1 Purpose and motivation

Researching the fundamental parts of our universe has been one of mankind's longest running endeavors. These days, in order to be able to probe further into nature, particle physicists require conditions mimicking that of the early universe which can be found in either extreme astrophysical phenomena or in the experimental halls of particle colliders such as the LHC (Large Hadron Collider). At the LHC, protons are accelerated in two beams and brought to collide within one of four detectors, providing sufficient energy to produce events that, when studied, can deepen our understanding of the microcosmos.

Large experiments such as ATLAS (A Toroidal LHC AparatuS) [1] and CMS (Compact Muon Solenoid) [2] at the LHC generate unprecedented volumes of raw data. However, in order to interpret the output of such instruments, we need to have a model of both the background and the characteristics of the instruments. With large systems, analytical solutions for describing expected properties are not going to be feasible. Rather, so-called Monte Carlo simulations are used to compute everything from initial reactions to detector response. Monte Carlo simulations (see Section 2.2.1) are a family of methods that rely on probabilities and random number generation to model systems.

In ATLAS, Monte Carlo simulations of the detector consume a large portion of the available computing resources. Current internal estimates predict detector simulations to be consuming $\sim 40\,\%$ of the total 4.5 HS06 CPU budget for ATLAS in 2020. As the rate of data generated by the experiment is increasing, our computing capabilities have to be able to keep up. If we fail to keep pace, we risk being bound by our computing capabilities rather than our experiments.

The software used to perform the detector simulations in ATLAS is based on a simulation toolkit called Geant4 (Geometry and tracking 4) [3]. Geant4 is a large international software project, and while improving the detector simulation performance by improving the source code of either Geant4 or the ATLAS simulation implementations is important, it is a long-term and expensive process. An alternative and orthogonal approach to improving performance is to change the way that the simulations are built in terms of compilation and linking (see Section 2.3.1).

There are theoretical reasons to expect that runtime performance could be improved by performing such changes. However, without proper measurements to confirm, it is possible that the improvement is minimal, and thus not worth losing the benefits of the current way of building the simulations. Even worse, the changes could reduce performance. Therefore, it is going to be important for the experiment to properly investigate and measure the performance characteristics of simulations built with different methods with as realistic as possible scenarios.

Beyond ensuring that a change would provide a substantial performance improvement, it is important that the change does not change the physics results from the simulations. First of all, the results must not be wrong. A program that runs fast but wrong is not hard to create but is not particularly useful. Furthermore, if different ways of building the same software produce different output for a given input, the possibility of replicating any simulation might not be possible. It is, therefore, going to be important for the experiment to ensure that any transformation it applies to its simulations produces the same results.

1.2 Goals

There are two parts to this project:

1. Investigate the potential for improving runtime performance of simulation of large detectors through a combination of the following build time methods:
    - Statically linking the Geant4 library and its companion libraries (Expat, Xerces-C++, CLHEP (Class Library for High Energy Physics)) rather than dynamically (see Section 2.3.2).
    - Upgrading or using a different compiler to build the simulation libraries (see Section 2.3.1).
    - Changing the so-called optimization level used by the compiler when building the project (see Section 2.3.1).
    - Instructing the compiler to use so-called unsafe math optimizations or to use specialized instructions (see Section 2.3.1)
    - Using additional optimization methods such as LTO (Link Time Optimization), which sacrifice the time it takes to compile the simulations to potentially improve runtime performance (see Section 2.3.2).
2. Perform measurements that check if physics results from the simulations change when applying any of the above mentioned methods, and if they do, estimate the relative impact of such differences.

2 Theory and background

2.1 A brief introduction to experimental particle physics

The focus of this project is detector simulations in high energy physics in general and for the ATLAS detector in particular. In order to understand what such simulations need to model for an experiment like ATLAS, we will introduce the basics of the Standard Model in Section 2.1.1, and how the particles and forces of the Standard Model interact with matter in Section 2.1.2. Equipped with this knowledge, we can discuss the basic components of modern detectors that Geant4 has to simulate. As we will see, Geant4 and detectors rely on a few simple parts of the Standard Model as it deals with the products of an initial interaction and not how the initial interaction is generated, which is where the more complicated theoretical parts find their home. In general, we will leave out material that is not relevant for getting an overview of ATLAS detector simulations for brevity. For a more complete picture we refer the reader to literature such as [4] for Section 2.1.1 and [5] for Section 2.1.2 and Section 2.1.3.

2.1.1 The Standard Model

The Standard Model of particle physics is the name of our current model of the microcosmos. It describes experimental results to an astonishing degree. At its heart are a couple of families of particles and their interactions. We will review these to highlight their characteristic observable properties.

The quarks are lightweight fermions not observed directly but as constituent parts of composite particles called hadrons, such as the proton. While the quarks carry fractional electric charge, compositions of quarks can be both charged or neutral. The quarks are bound together tightly due to the strong force which is mediated through the gluon. The reach of the strong force is technically infinite, but due to mutual interactions of the gluons, its effective reach is short. As we do not observe free quarks, interactions involving the strong force typically produce new hadrons. With enough energy available, strong interactions result in showers of hadrons.

The leptons, unlike the quarks, are free fermions. Half of the leptons carry charge (the electron, the muon, and the tauon) while the remaining, called the neutrinos, only couple to the weak force. Because of this, neutrinos barely interact with matter, and we typically do not observe neutrinos directly in our experiments but infer their presence through missing energy. The remaining leptons are similar to each other but have different masses, with the muon being ~ 200 times more massive than the electron. As we shall see in Section 2.1.2, this mass difference allows the heavy leptons to penetrate much further through matter than the electron and its anti-particle the positron.

We have already mentioned the gluon, which mediates the strong force. Correspondingly there are bosons for both the electromagnetic and weak forces. While the weak interaction and its three bosons, W^+, W^-, Z^0, are typically negligible in particle detection, the photon, typically denoted γ, which mediates the electromagnetic force, is a key part of particle detection.

At the detectors at the LHC, see Section 2.1.4, the Standard Model is studied by colliding protons or heavy ions with each other. However, most collisions will not result in anything interesting being produced. In fact, the LHC does not accelerate individual protons but bunches of them. In the interesting events, a hard scattering occurs between partons, constituent quarks or gluons, in a proton where a large transfer of momentum allows for the production of a particle of interest such as the Higgs boson. The hard scattering event is distinguished from everything else that happens in the collision, such as soft parton interactions, which is referred to as the underlying event.

2.1.2 Interactions of particles with matter

The forces that play a part in detector physics are typically the strong and electromagnetic forces. We will briefly discuss the primary interactions involved in detectors for high energy physics experiments and corresponding concepts.

The interactions that are relevant in a given system can be viewed as a function primarily of the particle species/matter constituents and the energy and momentum scale. For a proton with low momentum, the probability of strong interaction with another proton as part of the early stages of nuclear fusion is negligible due to the repulsion of the electromagnetic Coulomb force. However, for protons traveling with relativistic velocity in the core of a star, the momentum scale allows rare events to occur with enough frequency to allow for life in the universe. On the flip side, the electromagnetic force plays a far less prominent role for the neutron, where the strong and weak interactions are the most relevant forces.

When discussing the probability of interactions between radiation and matter, physicists typically discuss in terms of cross-sections, denoted σ, which is a characteristic function of the interaction itself. The total probability of an interaction occurring is given by the cross-section and the relevant parameters of the system, such as the thickness of a material that a particle passes through.

Heavy charged particles, such as the proton or muon, primarily loose energy in a medium by exciting or ionizing atomic electrons and secondarily through strong interactions with nuclei. These electrons can receive enough energy to create further interactions within the materials and are then referred to as δ-rays, which are a key component in many detectors. For heavy neutral particles, the primary interactions are with the nuclei. As the cross-section for these interactions is lower than the ionization of their charged counterparts, neutral particles typically have greater penetrating powers.

For light charged particles, electrons and positrons, the same interactions are available as for their heavier cousins. However, due to the much smaller mass, bremsstrahlung becomes an important factor in the equation. Bremsstrahlung can be thought of in a classical model as the emission of photons when a charged particle is accelerated in an electric field, such as the field of a nucleus. The cross-section for bremsstrahlung, σ_{rad}, scales as

$$\sigma_{rad} \propto \left(\frac{e^2}{mc^2}\right)^2 \tag{1}$$

which is why bremsstrahlung is typically insignificant even for the fairly lightweight muon for which the cross-section is a factor $\sim 40\,000$ lower than the electron.

As mentioned in Section 2.1.1, photons are capable of passing through and interacting with matter. There are three primary processes involved. The photon can be absorbed by an atomic electron leading to ionization, called the photoelectric effect, or reemission of one or more photons; they can scatter with free electrons, and, if their energy is greater than the invariant mass of an electron-positron pair, decay into an electron-positron pair. The charged particles produced in pair creation can then produce new photons through bremsstrahlung, which in turn can pair produce further as long as $E_\gamma \geq 1.022\,\text{MeV}$, producing an electromagnetic shower in the detector.

2.1.3 Detector fundamentals

Detectors are highly specialized instruments designed to measure a specific property of a type of radiation. Experiments, therefore, are typically constructed by layering detectors that each produce some observable, which, when combined, can be reconstructed into an initial state.

Trackers are detectors that are used to determine where a charged particle has traveled. Tracking is typically done by measuring a small energy deposition at multiple points and interpolating the route that the particle must have gone. If this is done in a homogeneous magnetic field, the momentum of a charged particle can be estimated through the radius of curvature inside the tracking volume. Tracking is usually the first part of a detector as further components typically stop the incoming particles, and knowledge about the precise location of interaction points, so-called vertices, is crucial for the further interpretation of an event. Certain types of trackers can further be used to provide particle identification through time-of-flight measurements or a particle's characteristic ionization energy loss.

Beyond momentum and position, one of the most important components to be able to reconstruct an event is the measurement of a particle's energy. The detector components responsible for measuring energy are called calorimeters. An effective calorimeter needs to be able to both completely stop the particle and measure the deposited energy. Homogeneous calorimeters are constructed of materials capable of performing both functions. In contrast, sampling calorimeters consist of layers of material where one material breaks the incoming particle and produces showers that are then measured by a layer of a different material.

For photons, electrons, and positrons, the strategy for energy measurement in an electromagnetic calorimeter is to utilize the pair-production and bremsstrahlung showers, which will halt an incoming photon, electron, or positron but not their heavier cousins. The material used to produce electromagnetic showers will typically not be well suited for hadrons, and therefore, electromagnetic calorimeters are usually followed up by calorimeters designed to stop the remaining particles. The only particles that remain will be muons, which can be identified by another layer of tracking, and the weakly interacting neutrinos whose existence has to be inferred.

2.1.4 The ATLAS detector and the LHC

The LHC is currently the world's largest particle accelerator. It is a circular accelerator for protons and heavy ions located at the border of France and Switzerland and hosts four major experiments as part of the CERN (European Organization for Nuclear Research) laboratory. This project focuses primarily on the ATLAS experiment, an overview of which can be found in Figure 1 and Table 1, and in part on the CMS experiment that are two general-purpose detectors. ATLAS is a layered instrument surrounding the beampipe in which the collisions occur in accordance with the principles discussed earlier. Closest to the beampipe is a tracking region called the inner detector, which is covered by first EM (Electromagnetic) calorimetry and then hadronic calorimetry. The choice of calorimeter varies from region to region of the experiment as they have different requirements. For example, tracks that are perpendicular to the beampipe require much higher precision than those passing through near the beampipe. Finally, ATLAS has detectors to measure any escaping muons. A description of CMS in detail can be found in [2] and of ATLAS can be found in [1].

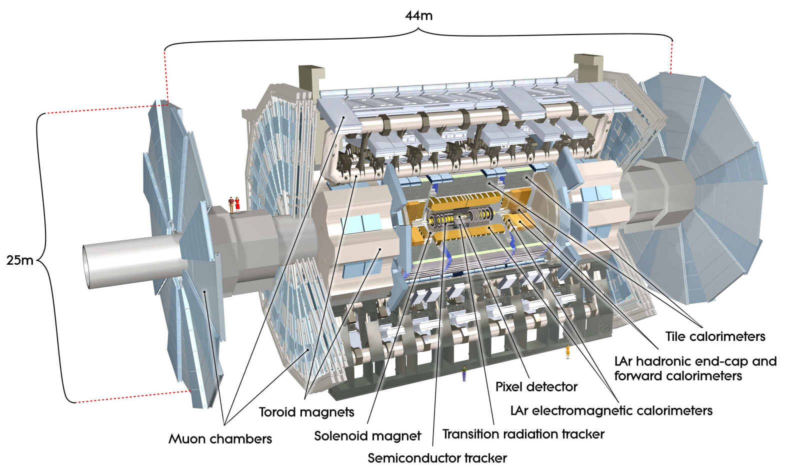

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 1: Overview of the ATLAS detector. The inner detector, which primarily does tracking, can be seen in the center surrounded by both EM and hadronic calorimetry [1].

The coordinate system commonly used when discussing ATLAS places the origin at the collision point, with the z-axis along the beamline, the y-axis oriented upwards, and the x-axis pointing parallel along the radius of the collider. Angular coordinates are defined as θ, the polar angle from the beam axis, and ϕ, the azimuthal angle going around the beam axis. In practice, the coordinates y Equation (2) and η Equation (3) called rapidity and pseudorapidity are used to characterize the polar coordinate. As the name suggests, pseudorapidity is an approximation for rapidity. The coverage of detector components in ATLAS are usually referred to using the corresponding range in η.

$$y = \frac{1}{2}\frac{E + p_z}{E - p_z} \tag{2}$$

$$\eta = -\ln\tan\left(\frac{\theta}{2}\right) \tag{3}$$

For small polar angles where particle momentum is primarily along the beampipe, $\lim\big|_{\theta\to 0}\eta = \infty$ while for large values of θ $\lim\big|_{\theta\to 90}\eta = 0$. The regions that different components of ATLAS cover are usually described in terms of η, so it is important to keep in mind that large values mean closer to the beampipe and that the scale is not linear.

Table 1: An overview of the detector components of ATLAS, their purpose, and the corresponding η regions they cover. Multiple values for η range correspond to separate parts of the same detector system. Adapted from [1].

Name	Type	η coverage
Pixel detectors	Tracking	$\|\eta\| < 2.5$
Semiconductor Tracker (SCT)	Tracking	$\|\eta\| < 2.5$
Transition Radiation Tracker (TRT)	Tracking	$\|\eta\| < 2.0$
Barrell EM calorimeter	EM calorimetry	$\|\eta\| < 1.475$
Electromagnetic Endcap Calorimeter (EMEC)	EM calorimetry	$1.375 < \|\eta\| < 2.5$ and $2.5 < \|\eta\| < 3.2$
Tile calorimeters	Hadronic calorimetry	$\|\eta\| < 1.0$ and $0.8 < \|\eta\| < 1.7$
Hadronic Endcap Calorimeter (HEC)	Hadronic calorimetry	$1.5 < \|\eta\| < 3.2$
Forward Calorimeter (FCal)	Hadronic calorimetry	$3.1 < \|\eta\| < 4.9$
Muon spectrometer	Tracking & Muon ID	$\|\eta\| < 1.05$ and $1.05 < \|\eta\| < 2.7$ and $2.0 < \|\eta\| < 2.7$ and $\|\eta\| < 2.7$

2.2 Data and simulation

2.2.1 The role of simulation in experimental physics

In order to interpret the data our experiments generate, we need a model of both the expected outcome of an initial collision and what signal we should expect from our instruments. The primary tool for this job is the Monte Carlo method, both for event generation and detector simulation. In this thesis, the focus will lie on the second half, detector simulation, as it is currently the most CPU intensive part of ATLAS computing.

Monte Carlo methods are a family of computing techniques based on probability and random number generation. In order to understand the usefulness of Monte Carlo for understanding physics systems, consider an isolated hypothetical particle Q at rest that we wish to simulate. For simplicity, let us assume that the only allowed interaction is a single decay channel with some mean lifetime τ. Given an initial population of $N(t=0) = N_0$ particles, how will the evolution of the system look?

Since the decay process is random in nature, we cannot make any simple deterministic calculation of when each particle will decay. However, since we know τ, we can give a probability, P, that a given particle will undergo decay given a small unit of time, $\delta t \ll \tau$, is

$$\tau P = \delta t \Leftrightarrow P = \frac{\delta t}{\tau} \tag{4}$$

If we discretize time in units of δt, we can model the remaining population without computing the theoretical value for N at a given time point by iterating and at each step, allowing each particle a chance to decay with probability according to Equation (4). We do this by generating a random number, x_i, from a uniform distribution in the range $[0,1]$ for each particle, Q_i, and if $x_i \geq \frac{\delta t}{\tau}$, we consider the particle to have decayed, and decrease the number of remaining particles.

If we repeat this process for a sufficient number of generations each of length δt, we can plot N as a function of t and obtain a distribution that matches what we would have observed if we performed a similar experiment in nature or the theoretical distribution that we could have calculated by integrating a differential form of Equation (4) and using $N(t_0) = N_0$ as our boundary condition[1].

Despite not being able to predict when any given particle would decay, the Monte Carlo method has allowed us to describe our physical system without any theoretical calculations, and therein lies the power of the Monte Carlo method. When we are no longer able to find analytical solutions to describe our physical systems, Monte Carlo stops being just interesting and starts becoming mandatory. Monte Carlo is not without limits though, we still require a robust understanding of the underlying system such as the cross-sections of interactions, and with increased complexity, the computing resources required to simulate a system can increase rapidly.

2.2.2 The simulation package Geant4

The current primary software package for simulations of particles interacting with matter is Geant4 [3]. As the name suggests, Geant4 is the successor of the Geant3 package and was developed to allow for similar performance but enabling more physicists to develop their own simulation applications.

A Geant4 simulation can be broken down into steps. First, a geometry description is evaluated and optimized. In a general Geant4 simulation, parts of the geometry can be designated as sensitive regions that can record interactions and the response of the particular material in the region. In our simulations, the XML-based GDML (Geometry Description Markup Language) format [6] is used to describe our detectors. Geant4 is also responsible for maintaining definitions of the materials of the geometry and their properties.

[1]If the reader has not performed any similar computing experiment before, we would recommend trying it out as the results can be quite eye-opening. Remember to keep $\delta t \ll \tau$.

In addition to the geometry, the system also requires a choice of physics models to be used and a description of what magnetic fields will be present. Physics models are always at their core approximations of a part of nature, and different models will better represent different systems. In Geant4, this is described by the choice of physics list, which contains a set of processes and models tuned for some specific application.

After this, Geant4 leaves possibilities to add arbitrary code to run both before and after specific steps in the process. These opportunities are collectively referred to as user actions and are registered as part of the setup process. For example, relevant for this project, a user can register an action to occur at the start of an event to mark the current time and another action at the end of the event, which registers the time that has elapsed.

Once all setup is done, a run can take place. A run is simply a collection of events, which is the basic unit of computation in Geant4. An event describes an initial state, e.g., what particles are present and what is their momentum, and the results of their interactions while traveling through the detector geometry. An event can be provided by data, by an event generator, or by Geant4 itself.

Given the initial state, Geant4 will then model the transportation of each initial particle and any particles it produces while interacting with the detector. Particles are transported in steps, which can be both a spatial step, for a moving particle, or a unit of time, for particles at rest, which is where physics processes take place in Geant4. A physics process can either take place at rest, at the end of the step, or along the step for continuous processes. Each physics process proposes a step, and the tracking system then decides which action will occur. Since Geant4 tracks individual particles, processes such as calorimetry, which involve the creation of showers of particles, therefore incur a heavy cost in simulations.

Geant4 distinguishes between physics processes and models. The former describes a particular initial and final state with its corresponding parameters, such as cross-section, while the latter manages the production of secondary particles in order to allow for different models for any given physics process. During transportation, each process will calculate an interaction distance based on the underlying probability distribution and a uniformly generated random number. The process which returns the smallest interaction distance is selected, and its action is performed. If the action is an interaction or decay, the particle is killed as new secondary particles are created; otherwise, it will undergo another process.

If one or more steps occur within a sensitive region of the detector geometry, a snapshot of the physical interactions that were involved can be registered as a hit. Furthermore, Geant4 provides a representation of the output of the detector called a digit that can be constructed either from hits, from other digits, or both. The hits are registered as part of the event object, so processing can be deferred and even done for several events at once in order to simulate more realistic situations where events are well separated.

A powerful feature of the digitization and hits based simulation system is that it can produce output in the same format as the experiment, which allows researchers to run simulated events through the same pipeline as for reconstruction and analysis as the measured data. Unlike the measurements, the simulated digitization can be provided with information about the initial event, so-called truth-data.

2.2.3 The ATLAS simulation data workflow

The scale of computing required to deal with the output of instruments like ATLAS necessitates specialized computing models. In ATLAS, the chosen computing model is Grid computing with computing resources distributed world-wide across an array of computing resources. We will review here how ATLAS manages its simulation components. The details of the overarching ATLAS software project and the computing framework, Athena, which manages it all, are described in [7] while further details on the simulations can be found in [8].

Simulation in Athena for ATLAS is conceptually divided into three distinct steps. First, events need to be generated along with their immediate decay products in the event generation step. Next, the generated events are fed into the detector and physics portion of the full Geant4 simulation process. Finally, the results are put through the simulated digitization process in Geant4, which produces output in terms of voltages and currents from the sensitive regions of the detector. The output can be produced either in an object-based format or in a format identical to that of the actual detector.

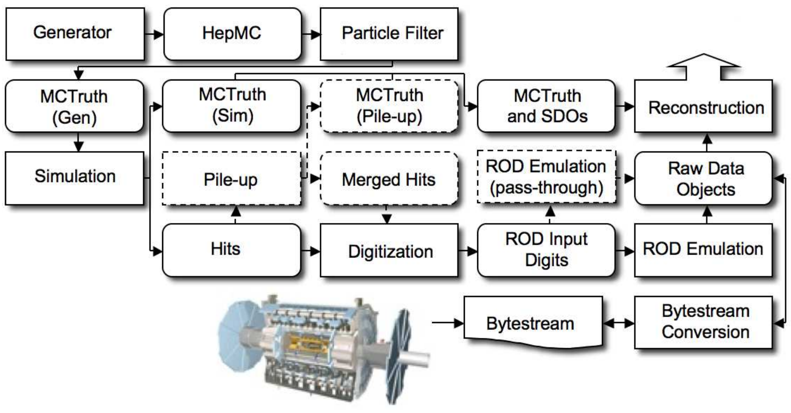


Figure 2: Overview of the complete ATLAS simulation process and its relationship to the output of the experiment originally from [8]. The locations where Monte Carlo-truth data can be stored are shown as separate steps, and optional paths are marked with dashed outlines.

An overview of the entire process, including all the potential steps, can be seen in Figure 2 or, in detail, in [8]. In the first step, events are generated from a Monte Carlo program which simulates the initial collision and its immediate decay products called, unsurprisingly, an event generator into a standardized format called HepMC. Not every event type is going to be relevant to any one particular analysis, and uninteresting events can be discarded here already. Truth information from the generator can be perpetuated through the simulation chain, including, e.g., initial particles.

Following the generation step, the events are fed into the Geant4 simulation, which records hits and truth data from the simulation. Optionally, multiple events can be combined into one event at this step in order to simulate the so-called pile-up effect of multiple collisions occurring at once, again storing truth data about the process. If pile-up simulation was performed, the cumulative hits are merged and sent to the final step. The primary reason for simulating pile-up by combining events rather than generating initial events with multiple collisions is to reduce the CPU load of the simulations as Geant4 tracks each initial particle individually.

The third and final step of the simulation chain is digitization. Given either hits from individual events or a collection of merged hits from pile-up, the digitization process converts hits into detector responses, called digits. In producing the digits, the process will also overlay noise and other effects that are going to be part of the non-idealized physical detector, such as potential long-lived particles. The digits are then fed to an emulation layer of the detector electronics' read out drivers. The final output is a format called raw data object files that can be converted to and from the bytestream produced by the detector.

2.3 The production of scientific software

This section is intended to be a brief introduction to compilers, linkers, and optimizations. For the interested reader, see Appendix A and [9, 10]. An overview of the complete process from source code to executable can be found in Figure 3.

2.3.1 Compilers and optimization

Scientific software is not in principle different from any other kind of software. Source code is written in a human-readable format called a programming language, which is then processed into instructions to the hardware in a binary file, often called binary for short. This can be either in a format that can be run immediately, called an executable binary or executable for short, or a format containing code that other programs can use called library files. There exist a plethora of languages that are optimized for different purposes, but they can, in general, sorted into one of two categories: Compiled or interpreted languages.

The analogy with language provides for a useful metaphor for understanding the differences between the two. If you need to communicate with someone who does not share any languages with you, you will need external help. One way to do this is to use a translator who takes what you want to say and translates it directly into a static form that can then be passed on to the person you wish to communicate with. Alternatively, you can make use of an interpreter who will perform the translation as you speak. Compiled languages communicate your thoughts based on a translator called a compiler while interpreted languages, perhaps unsurprisingly, do the same thing live like an interpreter. This project focuses on compiled languages, C++ in particular, as these are the basis of simulation packages such as Geant4.

By default, compilers will perform a naive translation from the programming language to the machine instructions. This is helpful for understanding what your program will actually do and for debugging as there will, mostly, be a one to one translation from what you wrote to what the computer will run. However, this is unlikely to be the most efficient way to perform the task you wanted and, if you ask for it, the compiler can perform transformations, called optimizations, with the goal to improve efficiency with respect to some metric, usually how fast the program runs. Generally, the transformations are supposed to produce a binary that produces the same output as the unoptimized version. However, as we shall see, there are cases where it can be desirable to perform optimizations that can change the program output.

Most compilers allow users to specify exactly which transformations are allowed, but also provide groups of transformations called optimization levels. In the compilers used in this project, there are numbered optimization levels (labeled `-O0`,`-O1`,`-O2`, and `-O3`) that apply successively more optimizations. Here, the higher numbers imply the optimizations of the lower numbers. There are also specialized optimization levels that work well for some specific purpose or priority, such as `-Os`, which prioritizes creating small executables.

The last optimization level discussed highlights an important principle for using these optimization capabilities; There are always tradeoffs when choosing an optimization level, and one such important tradeoff is the time/space tradeoff. When forced to choose, the `-Os` setting will prioritize creating a small executable while `-O3` will do everything it can to produce code that, in principle, should run as fast as possible.

The compiler is, however, not omniscient, and optimization decisions are made based on heuristics, so without measuring, it is impossible to know which setting will run the fastest. For example, the aggressive optimizations in `-O3` could produce code that does not fit the faster parts of the processor, which then has to use the, sometimes orders of magnitude, slower parts of the hardware. Compilers also are bound to do what you told them to do and cannot perform transformations such as changing the traversal order of a matrix, which on modern computers does not perform the same by a longshot, unless it can prove, definitively, that there is no way that you could observe the difference.

In this thesis, we will primarily discuss three kinds of optimizations, two of which will be referred to as unsafe optimizations, which refers to their theoretical impact on output. The safe optimizations are the types of optimizations you get from the standard optimization levels. They only perform transformations that are guaranteed not to alter the output of the program. More specifically, they only perform transformations that preserve IEEE (Institute of Electrical and Electronics Engineers) and ISO (International Organization for Standardization) rules and specifications for math functions and operations.

Next up and first in the unsafe category is architecture-specific optimizations. By default, most compilers will produce binaries that run on multiple types of machines with the same underlying architecture. This is useful for portability but prevents the compiler from making use of instructions that are not available on the lowest common denominator in the architecture family. An important category of these instructions is the so-called vector instructions that perform multiple computations in one instruction. The vector instructions are the main reason why these optimizations are considered unsafe, as the order of their operations is not guaranteed, and floating-point arithmetic is not associative. Other than the safety aspect, the main challenge with these optimizations is that you need knowledge of the hardware that the code will run on as it will risk crashing on systems where particular instructions are not available and otherwise running slow. In this thesis, we will commonly refer to these optimizations as the native optimizations as we tell the compiler to use whatever the native architecture of the system it is running on.

The third category is the fast-math category. These are primarily optimizations for floating-point mathematics that may violate strict standards compliance. Some allow the compiler to rearrange floating-point calculations in ways that would reduce the number of necessary operations, some can replace common operations with instructions, and some let the compiler assume that your code will not rely on some parts of floating-point behavior such as infinities.

A final complication that is specific to C++ and related languages is something which is called UB (Undefined Behavior) [2] [11]. Some things are either forbidden by the language to rely upon or do, or that the language simply cannot specify the behavior of. The dangers of UB are often stated with hyperbole[3], but a much more helpful way to think about it is that a program invoking UB is incorrect. This matters for our purpose both because compilers are allowed to optimize assuming that undefined behavior does not occur and as the output of two different compilers is not guaranteed to be the same if the software contains or relies upon UB. In other words, if the program is incorrect, the safe optimizations are no longer safe. A further introduction and a famous demonstration of how a program with UB can disprove Fermat's last theorem can be found in Appendix B.

2.3.2 Linking

Complex programs such as scientific analysis code are made by stitching together code that has been written by someone else, so-called libraries, that is combined with the researcher's code. Furthermore, code is typically split up into different subparts called translation units. In order to make compilations take less time, compilers process each such translation unit independently into object files, which allows builds to be parallelized and reduces the amount of code that needs to be recompiled when one translation unit is changed. At the end stage of a build process, we, therefore, find ourselves with a group of object files and files from our libraries, and we need some process for merging them into the executable. This process is linking and has room for interesting tradeoffs in its own right, which we will review here based on [9, 10].

[2]Technically speaking, there are different categories in addition to UB that are often collectively referred to as UB, such as implementation-defined behavior and the somewhat unhelpfully named unspecified behavior. The difference can be important but is omitted in this project for simplicity.

[3]It is not uncommon to hear that programs invoking UB are allowed to make your male cat pregnant, erase your hard drive, or summon nasal demons. While technically true, what is much more likely is that they simply will produce wrong output or, much worse, produce what looks like correct output but isn't.

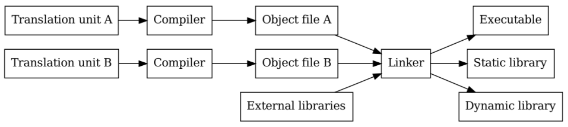


Figure 3: Simplified overview of the model of separate compilation and linking with two translation units producing either an executable, a static library, or a dynamic library. Each translation unit is first parsed, translated, and optimized independently by a compiler into an object file. The object files are then combined with any libraries that the program makes use of into the final product. If LTO is utilized, the optimization step is moved from the compiler and into the linker.

When it comes to the library files, they come in two different formats for two separate ways of linking, static and dynamic linking. Static linking is the more straightforward method. A useful mental model for static linking is that the linker cuts and pastes the relevant parts of the object files and the libraries together and leaving the other parts be. In fact, static libraries are just collections of object files.

Dynamic linking, on the other hand, is a bit more complicated. When a program is dynamically linked, the linker, rather than inserting the functions and values into the program, will record what functions or values are needed from the dynamic libraries. Not putting the functions directly into the executable allows the same binary to use different versions of the same library without recompiling and linking the executable. On Linux systems, the necessary setup to combine the executable and the libraries is done at runtime by a program called the dynamic loader.

There are two methods for performing this setup on Linux systems, load-time relation, and PIC (Position Independent Code). The latter of the two is the most commonly used and is what we will use in this project. With PIC, a layer of indirection is inserted into the executable. When the program starts, the dynamic loader loads the dynamic libraries but does not look up any individual function that is to be used. Whenever a function in a dynamic library is to be used, a call to a function in an area called the PLT (Procedure Linkage Table) is called instead. This function first does an indirect jump into a region called the GOT (Global Offsets Table), which will on the first entrance return back to the PLT-code. The PLT-code will then continue into the dynamic loader. The dynamic loader will look up the address in the dynamic library and write it back into the location in GOT before proceeding to call the original function. This allows the subsequent calls to the function to avoid the overhead of looking up addresses, but the initial indirection remains.

A simple, but useful, mental model to remember how dynamic linking and loading works is that the linker leaves a note in the executable with everything that is missing that the loader then goes and looks up at runtime.

A consequence of the separate compilation of translation units is that languages like C++ face a challenge that other languages do not. How can the language ensure that a variable, function, or another part of the language is defined the same way throughout every translation unit? In C++ the resolution of this problem is the ODR (One definition rule) [11]. ODR can, with much simplification, be understood as stating that a valid program does not contain different definitions but not requiring that the language implementation, in general, identify if it occurs. The details of the rule are complex and can be found in the current ISO specification of the language. ODR violations can be the source of subtle and serious errors.

Beyond static and dynamic linking, the link-time provides an additional opportunity for improving runtime performance. A disadvantage of the model of separate compilation of translation units is that the compiler only has vision of a limited section of the program and will be unable to make certain assumptions. For example, consider Listing 1 where a global variable, such as the global integer `pi`, might be constant throughout a given translation unit, which could allow the compiler to remove redundant code, such as in `use.cxx`. However, it could still be possible that the variable is modified in another translation unit, such as `might_modify.cxx` in our example, so the compiler cannot perform the optimization.

Since the linker sees the entire program at once, it does not face the same visibility restrictions as the compiler, and if optimization is deferred until link-time, more optimization opportunities may present themselves. This method is called LTO (Link Time Optimization) [4]. In Listing 1, the branch in the `relies_on_i` function can safely be removed, and the entire function reduced to `return 3;`. In more realistic scenarios, the dependencies that have to be calculated to find out if an optimization is safe are much more complicated. In addition to providing novel optimization opportunities, LTO can potentially allow compilers to diagnose ODR violations.

```
// header.h
// Define a global variable
int pi = 3;
// use.cxx
#include "header.h"
int relies_on_i () {
    if (pi == 3) {
        return pi;
    }
    // This never happens!
    return 0;
}
// might_modify.h
#include "header.h"
// might_modify.cxx
#include "header.h"
void might_modify_i() {
    //  pi = 4;
}
```

Listing 1: Example pseudo-code demonstrating how LTO can allow optimizations that are impossible in the standard model of separate compilation of translation units. The comments, i.e., text following `//`, designate what part of the code belongs to which of the files involved.

[4]LTO is often used interchangeably with whole program optimization and interprocedural optimization. We make no distinguishment between the three in this project.

2.4 Challenges with measuring computing performance

When measuring the performance of software, or benchmarking as it is often referred to, there are a couple of ways that things can go wrong, and it can be hard to detect if something is incorrect. Going into all the potential pitfalls is beyond the scope of this thesis, but we will discuss some basic principles and some of the more nefarious issues that are unique to measuring computing performance. Together with a solid understanding of basic statistics, such as can be found in [12], this material should provide the reader with sufficient knowledge to analyze basic performance benchmarks.

First of all, we need to make sure that we are producing a controlled environment. There are two parts to this problem, how can we ensure that our process is as independent of the host system as possible, and how do we control the build. For the second part, we need to monitor the actual compiler and linker invocations that our system produces, which has been done manually in this project. A significant source of problems here that one has to take care of is that build systems, being helpful, will set recommended optimization flags for you. In normal scenarios, this is what you want, but for us, this is a problem since build systems will typically append their flags to the ones we provide, and compilers will use the last flag given to them (i.e., `-O3 -O2` is equivalent to `-O2`).

Measuring itself has some caveats of its own. First, we need to consider what we measure when we time an event. In addition to the time it took to run the event, t, there can be background caused by the rest of the system, $t_{background}$, and the time it takes to perform the measurement, $t_{overhead}$. The time we measure, $t_{measured}$, can thus be represented as

$$t_{measured} \approx t + t_{overhead} + t_{background} \tag{5}$$

The benchmark we utilize in this project, see 3.6, however, assumes

$$t \gg t_{overhead} + t_{background} \Rightarrow \tag{6}$$
$$t_{measured} = t \tag{7}$$

There are further issues that are not covered by this simple model that we have to pay close attention to. First of all, t is not going to be consistent across runs even with a single-threaded measurement. This is because the time that the program is allowed to be on the CPU is split up into slices by the kernel. These slices can be split onto different cores, which can have different properties. This effect can produce an additional spread in the t-distribution but can be mitigated, to some degree, by pinning the process to a single core. Finally, the act of measuring itself can affect the time an event takes. One way this can happen is by affecting the CPU caches. Caches are small but fast pieces of memory where recently accessed data can remain. This is excellent for a process that keeps doing the same thing, but introducing timing code into the system can interfere with this. In pathological cases, this can have orders of magnitude of impact on the execution time. An in-depth discussion of these issues is beyond the scope of this text, but the interested reader is encouraged to explore literature such as [13].

If Equation (6) does not hold, there is no clear cut solution; however, there are steps that can be taken to mitigate the situation. The measurement can be amortized, offsetting the impact of $t_{overhead}$ by measuring the time it takes to perform a set number of events rather than a single event. A disadvantage with amortization is that it can hide some of the features of the distribution of your events. Additionally, care can be put into reducing $t_{overhead}$ by performing as few operations as possible, in particular, avoiding I/O operations, such as by storing timing results locally while measuring and processing the events only after the simulation is complete.

3 Methodology

3.1 Overview

We have performed measurements of both time per event and energy deposited per event in Geant4 simulations with both ATLAS and CMS detector geometries. These measurements were made on two hardware platforms, the Aurora computing cluster at LUNARC in Lund and a standalone machine at CERN, which we will refer to as PMPE in this thesis (Section 3.3). To perform the measurements, a common toolchain, containing both compiler, linker, and relevant system libraries, was bootstrapped on each machine. This toolchain is then used to compile three versions of the GCC (GNU Compiler Collection) compiler as well as the Clang compiler.

Using these compilers and a version of the ICC (Intel C++ Compiler) compiler provided by the host system, the Geant4 library, three of its dependencies, the HepExpMT benchmark, and a modified version of HepExpMT for physics validation (Section 3.6) were compiled once for every configuration (Section 3.4) in an isolated environment (Section 3.5). Once compiled, the benchmark and validation programs were run for 5000 isotropically distributed 50 GeV π^- events each (Section 3.5).

The benchmark recorded CPU time per event, t_b, while the validation code recorded CPU time per event, t_v, and energy deposited per event, E. In order to be able to spot differences originating in the random number generation, the last random number generated, X, in the benchmark program is also recorded. Any difference in random number generation occurring in the process will result in a different value for X. Finally, the validation code is run one more time for each configuration to be able to see if the output is stable.

3.2 History and initial work

Initial work was done using geometry for CMS and the inner detector of ATLAS as we had yet to acquire a useful GDML file for the full ATLAS detector. The initial measurements were made using multiple threads, i.e., splitting the work into chunks that can run at the same time on the hardware, but no significant difference in the relative performance characteristics could be seen between runs with multiple or a single thread. For validation purposes, the order of events is important to ensure reproducibility, and single-threaded runs were therefore necessary. We, therefore, decided to focus on single-threaded runs for both validation and benchmark runs.

3.3 Hardware used

In order to avoid the quirks of any particular hardware configuration, we performed measurements on two platforms. The Aurora cluster at LUNARC in Lund is a shared computing cluster where each node has 2 Intel Xeon E5-2650 v3 CPUs, (2.3 GHz base frequency, 10 physical cores per socket, 25 MB cache). The standalone machines at CERN used in the project, referred to as PMPE in this thesis, are dedicated to the benchmark and validation. The CPUs on PMPE are dual socket Intel Xeon E5-2630v3, (2.4 GHz base frequency, 8 physical cores per socket, 20 MB cache). Both the Aurora and PMPE CPUs are from the Haswell microarchitecture generation.

3.4 Configurations

We define a configuration as a combination of a

- A compiler
- If the libraries are compiled as dynamic or static libraries
- What optimization flags are used
- If the native architecture flags are used
- If LTO was used

We use three versions of the GCC compiler, 4.9.4, 6.2.0, 8.3.0. GCC version 4 is the system compiler on CentOS 6 and CentOS 7, two distributions used on scientific computing clusters, and is, therefore, what physicists often will be using as it is what you get by default. We wanted to look at later versions of GCC as well to see if there was a performance difference or, perhaps more importantly, in the physics output. As potential alternatives to GCC, we have used the most recent versions available of the Clang compiler (10.0.0) and Intel's ICC (19.0.5.281).

For optimization levels, we look at `-O2`, which is what is often set by default, the more aggressive `-O3`, `-Os` which optimizes for small binary size, and `-Ofast`, which is equivalent to `-O3` but with additional unsafe math optimizations. There have been two exceptions during this project. First, LTO and `-Os` do not work together with Clang. Due to a bug in our version of Geant4 and the specific version of ICC on Aurora causing the compilation process to fail, no configurations with ICC could be tested on Aurora.

In total, we have looked at 156 configurations. To keep things brief we use the following naming scheme for the configurations

Compiler-{D/S}Flag[N][L]

where {D/S} represents either static (S) or dynamic (D) linking, N represents if native architecture flags were used, and L represents if LTO was used. As an example, gcc4-SOfastNL is the GCC 4.9.4 compiler, static linking, `-Ofast`, and native architecture and LTO. We will be using the gcc4-DO2 configuration as the reference configuration.

We will further make use of three additional categories. We will refer to any configuration which uses either `-Ofast` or native architecture flags as unsafe configuration. Correspondingly, the remaining configurations we will categorize as safe. However, the ICC compiler uses some unsafe methods by default, and it will be important to be able to separate the safe configurations which use ICC from those that use GCC or Clang. Otherwise, it is possible that the unsafe methods in ICC would artificially increase the performance of the safe category. Therefore, we classify all members of the safe category that are not using ICC as very safe configurations.

3.5 Setup and building

We have developed a simple system for reproducible Geant4 performance measurements. As we want to perform measurements on two separate platforms, which have different models for execution, we had to develop a system where adding support for other systems is trivial. In the system we produced, adding support for a different model of execution requires at most defining a couple of shell script functions and a couple of environment variables.

In order to reduce the dependency on the underlying operating system, we start by setting up a local filesystem on each machine and bootstrap a small but sufficient toolchain for compiling our project and their dependencies. There are limitations to this procedure as the lowest parts of the computing stack, such as the C library and the kernel, are not possible to swap out in this way. However, most other important parts of the system, such as compiler and binary utilities, can be replaced. For a detailed listing, see Appendix C.

With this setup, we compile CLHEP [14], expat, xerces-c++, Geant4, HepExpMT (see Section 3.6), and our validation code individually for each configuration. The compilation is performed in an isolated environment, details of which are described in Appendix C, and the output from each configuration is logged and monitored manually to ensure that we measure what we set out to measure. One caveat to this procedure regards the Intel compiler, which we do not have access to the source code for and, therefore, cannot isolate as well as the remaining compilers. When compiling HepExpMT and the validation code, we also set the initial seed value for the random number generators to a set value, 6446.

In order to reduce noise from other processes running on the machines, the code was run on a compute node with exclusive access on Aurora. Since the code does not make use of multiple threads, we can reduce noise on the PMPE machines by turning off hyperthreading and pinning each run to an individual physical core. Turning off hyperthreading means that we only use the physical cores on the CPU, which reduces the risk of cache effects. Pinning the process to a CPU reduces the impact of the operating system scheduler moving the process between CPUs.

3.6 Benchmarking with HepExpMT and validation

Once every setup is compiled, the actual benchmarking and validation can take place. We use the HepExpMT benchmark [15], originally designed for benchmarking Geant4 simulations for CMS, and a version which we modified for validation. HepExpMT is a simple benchmarking program that constructs a geometry from a GDML-file, generates events according to instructions, and measures the time it takes from the start of an event to the end of the event, t_b. The validation version adds additional logic to store the total energy, E, deposited throughout the event by summing up the event deposited in each step as well as recording the time per event, t_v. The rationale for the method of calculating the energy deposited is that we do not have access to the sensitive detector regions that are used by the experiment in our GDML-files, so we cannot determine if an energy deposition would have been registered.

The validation code uses ROOT [16] to store its data, while HepExpMT prints it to standard output. Neither method is optimal with respect to $t_{overhead}$ or potential observer effects, but given our assumption in Equation (6) it should suffice. For measurements with less complex geometries, other strategies such as amortization would be required.

If the validation runs would show that there are differences in energy deposition per event between configurations, we want to be able to investigate the source of the differences. In Monte Carlo simulations, any perturbation in the way that random numbers are generated will produce a different output. Differences arising purely from the sequence of random numbers would produce a different sample from the same underlying distribution and should converge to the same distribution given enough events. We utilize that HepExpMT records the final random number produced, denoted X, to get insight into if differences in energy deposition are due to a difference in random number generation. If there are differences in random number generation between two configurations, then the corresponding values of X will differ.

In total, we use 5000 50 GeV isotropically distributed π^- events for both the CMS and ATLAS geometries. We run the validation code twice for each configuration on both Aurora and PMPE, so we can ensure that the output is reproducible, and the benchmark code one for each configuration on both Aurora and PMPE. The physics list used was `FTFP_BERT` for both CMS and ATLAS.

3.7 Analysis

For each configuration, we calculate an average CPU time per event in the benchmark, μ_{t_b}, average CPU time per event in the validation run, μ_{t_v}, and average energy deposited in the detector per event, μ_E. We then calculate the average gain or loss in the benchmark

$$\Delta t_b = \frac{\mu_{t_b} - \mu_{t_b}^{ref}}{\mu_{t_b}^{ref}} \tag{8}$$

where $\mu_{t_b}^{ref}$ is the average CPU time per event in the reference configuration (gcc4-DO2). Similarly, the average gain or loss in the validation run with respect to the reference average, $\mu_{t_v}^{ref}$, is calculated as

$$\Delta t_v = \frac{\mu_{t_v} - \mu_{t_v}^{ref}}{\mu_{t_v}^{ref}} \tag{9}$$

For μ_E we are interested in the relative absolute size of any difference between μ_E and the average energy deposited per event in the reference configuration, μ_E^{ref}

$$\Delta E = \frac{\left|\mu_E - \mu_E^{ref}\right|}{\mu_E^{ref}} \tag{10}$$

We found that there appeared to be two distinct classes of events, one fast and one slow in both the t_v and t_b distributions and that the complete distribution could be well described by a fit of a combination of a Landau distribution, for fast events, and a Gaussian distribution, for slow events. We characterize the full distribution by the average CPU time per event, the location parameter of the Landau distribution of the fast events, denoted μ_{fast}, and the mean of the Gaussian distribution denoted μ_{slow}.

4 Results

4.1 Performance measurements

We present here results for the first part of the project, measuring the effect of alternative configurations on Geant4 simulations with ATLAS and CMS detector geometries. We first review the baseline configurations with gcc4-DO2, which can be found in Figures 4 and 5, and their characteristics.

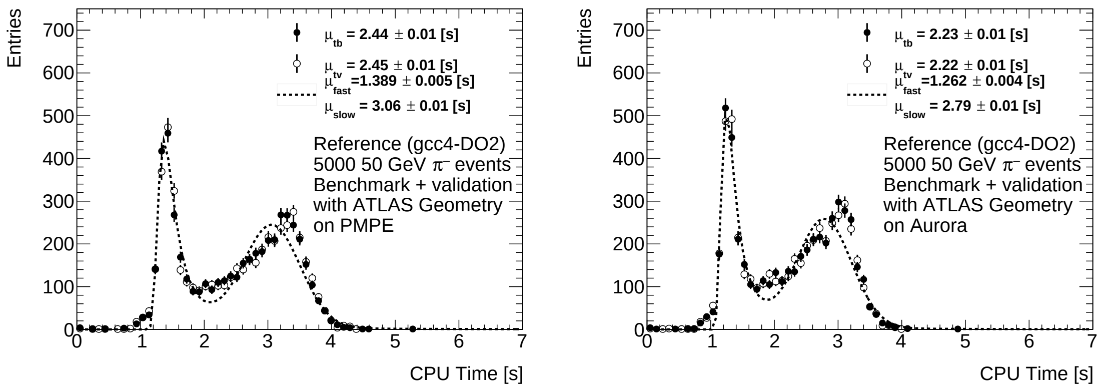


Figure 4: Reference measurements with the benchmark and validation code using the ATLAS geometry and PMPE hardware (left) and Aurora hardware (right). Average CPU time per event in benchmark on PMPE $\mu_{t_b} = 2.44(1)\,\mathrm{s}$ and on Aurora $\mu_{t_b} = 2.23(1)\,\mathrm{s}$. Average CPU time per event in validation runs on PMPE $\mu_{t_v} = 2.45(1)\,\mathrm{s}$ and on Aurora $\mu_{t_v} = 2.22(1)\,\mathrm{s}$.

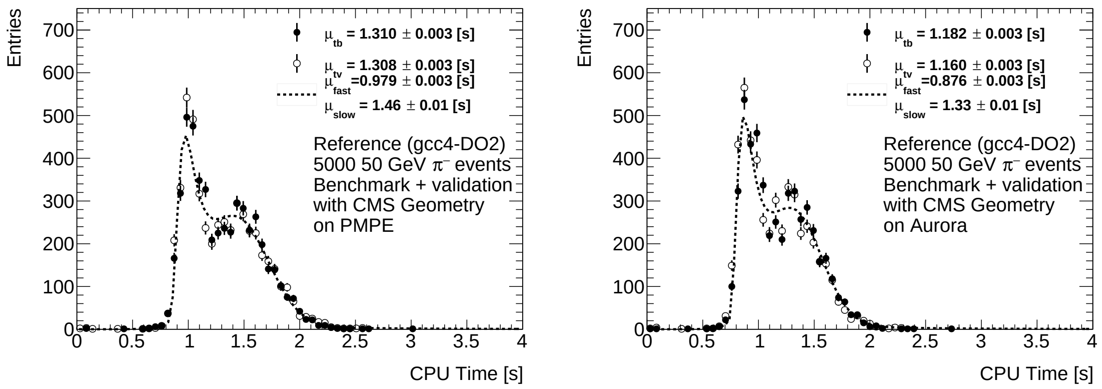


Figure 5: Reference measurements with the benchmark and validation code using the CMS geometry and PMPE hardware (left) and Aurora hardware (right). Average CPU time per event in benchmark on PMPE $\mu_{t_b} = 1.310(3)\,\mathrm{s}$ and on Aurora $\mu_{t_b} = 1.182(3)\,\mathrm{s}$. Average CPU time per event in validation runs on PMPE $\mu_{t_v} = 1.308(3)\,\mathrm{s}$ and on Aurora $\mu_{t_v} = 1.160(3)\,\mathrm{s}$.

For both ATLAS and CMS, the μ_{t_b} is roughly 10 % shorter on Aurora than on PMPE. No significant difference between μ_{t_b} and μ_{t_v} can be observed for either reference configuration, indicating that either Equation (6) holds for both measurements or none of them. The combined Gaussian and Landau distribution fit for the benchmark distributions matches our measurements reasonably well.

For ATLAS, the fast events are located around $\mu_{fast} = 1.389(5)\,\mathrm{s}$ (PMPE), and $\mu_{fast} = 1.262(4)\,\mathrm{s}$ (Aurora), while the slow events are located around $\mu_{slow} = 3.06(1)\,\mathrm{s}$ (PMPE) and $\mu_{slow} = 2.79(1)\,\mathrm{s}$ (Aurora). With CMS, the peaks are closer together but still distinct, with $\mu_{fast} = 0.979(3)\,\mathrm{s}, \mu_{slow} = 1.46(1)\,(s)$ (PMPE) and $\mu_{fast} = 0.876(3)\,\mathrm{s}, \mu_{slow} = 1.33(1)\,(s)$ (Aurora).

Given the information from the reference runs, we can now begin to consider performance improvements. We will only review a representative selection here. However, the complete results are available in Appendix D. We will look in particular at static and dynamic versions of four configurations for each compiler, the small size optimizing `-Os`, a basic `-O2`, a version with unsafe optimizations `-OfastN`, and the same version but including LTO. These measurements are shown in Figure 6 for GCC 4.9.4 and GCC 6.2.0, Figure 7 for GCC 8.3.0 and Clang, and Figure 8. From these configurations, we can obtain the following key observations:

- `-Os` will, in general, perform worse than the corresponding `-O2`, and in some cases much worse. For example, the performance of dynamically linked `-Os` for GCC 4 is on average $\Delta t_b \approx -15\,\%$. The primary exception is Clang, where it tends to perform the same as `-O2`, for example, $\Delta t_b \approx 0\,\%$ for dynamically linked `-Os` with Clang.
- Performance improvements with GCC are, in general, more stable across geometry and platform than ICC and, particularly, Clang.
- Statically linked configurations perform better in general. For example, for GCC 4, the average for statically linked `-O2` is $\Delta t_b \approx 8\,\%$ when compared to its dynamically linked counterpart, the reference.
- The unsafe optimizations perform better in general. For example, the relative improvement for `-DOfastN` with GCC 8 is $\Delta t_b \approx 8\,\%$.
- There seems to be little performance improvement or loss from LTO, except for ICC and the CMS geometry, where performance drops by $\sim 50\,\%$.
- There are exceptions to all of the above.

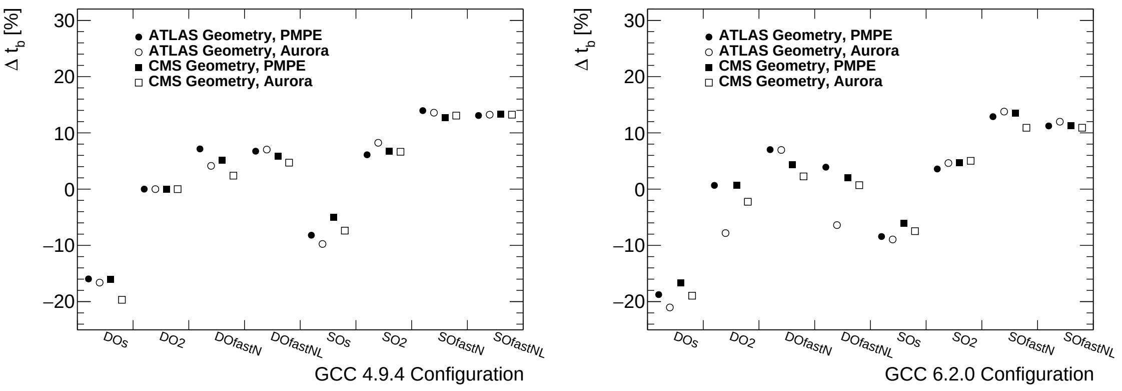
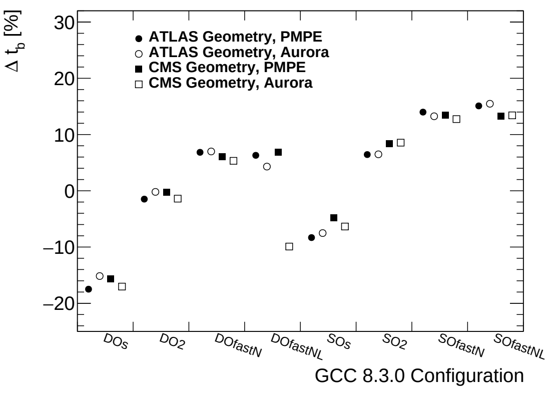


Figure 6: Relative performance improvements, Δt_b, for the GCC 4.9.4 (left) and GCC 6.2.0 compilers (right) for dynamic and static versions of four configurations. Results are relatively similar across the two compilers, although there are some differences for ATLAS geometry on Aurora (`-DO2` and `-DOfastN`).



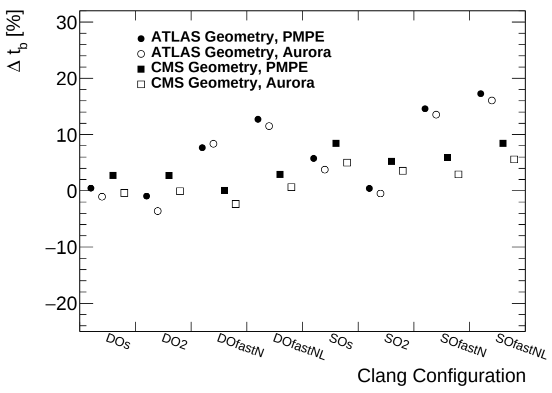


Figure 7: Relative performance improvements, Δt_b, for the GCC 8.3.0 (left) and Clang compilers (right) for dynamic and static versions of four configurations. The performance characteristics for GCC 8.3.0 is similar to those in Figure 6 but quite different from that of Clang. Notably, the `-Os` performance is much worse for GCC. Furthermore, Clang performs better on PMPE than on Aurora, and better with the CMS geometry for `-Os` and `-O2`, but worse for `-OfastN` and `-OfastNL`.

As shown in Figure 8, something goes wrong for ICC with LTO and the CMS geometry. Figure 9 demonstrates the difference between one such configuration with LTO on or off. In the LTO scenario, there is a long tail after the initial peak rather than the second peak. Furthermore, the fast peak is moved from $\sim 0.77\,s$ to $\sim 1.0\,$s. This behavior is unique to ICC and the CMS geometry, which is curious as there is no obvious reason as to why CMS should be different from ATLAS here.

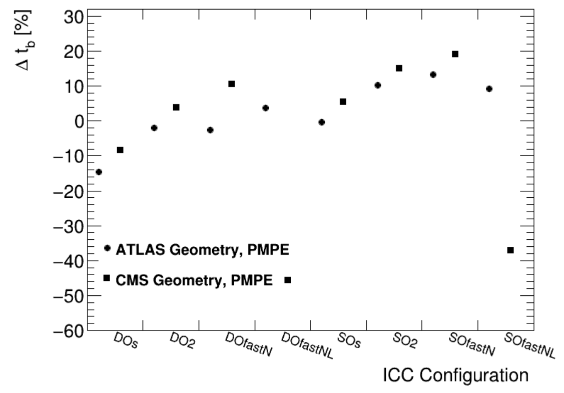
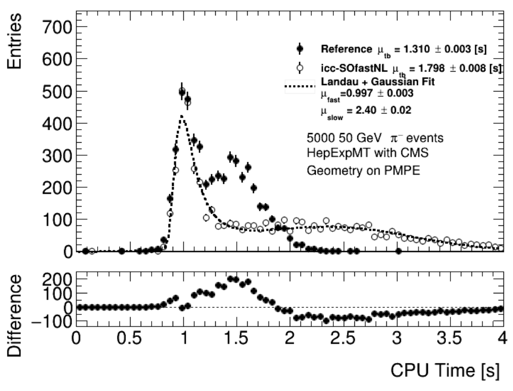


Figure 8: Relative performance improvement, Δt_b, for dynamic and static versions of four configurations using the ICC compiler. No measurements could be made on Aurora with ICC due to a bug in the version of Geant4 used in the project. ICC performs better in general with the CMS geometry, except `-OfastNL` where ICC with the CMS geometry performs far below other configurations.



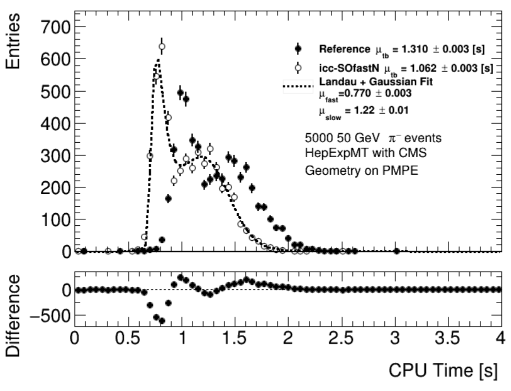


Figure 9: Performance measurement with HepExpMT for one of the ICC configurations, which performed drastically below average with the CMS geometry when using LTO (left) than without (right). Average CPU time per event in benchmark $\mu_{t_b} = 1.798(8)$ s (left), $\mu_{t_b} = 1.062(3)$ s (right). Combined Gaussian and Landau fit for two separate event categories estimates average CPU time per fast event $\mu_{fast} = 0.997(3)$ s (left), $\mu_{fast} = 0.770(3)$ s (right), and average CPU time per slow event $\mu_{slow} = 2.40(2)$ s (left), $\mu_{slow} = 1.22(1)$ s (right).

So far, we have looked in some detail at a limited set of configurations. To get an overview of the remaining material, we present the best performing configurations in Tables 2 to 9. We will for brevity not include corresponding tables for the safe and very safe categories, refer to Appendix D for the remaining measurements, but we will still describe their corresponding characteristics. The overall results can be summarized as follows:

- The overall best performing configuration in each safety category was
  - **Unsafe:** $\Delta t_b = 18.9\,\%$, icc-SOfastN
  - **Safe:** $\Delta t_b = 15.0\,\%$, icc-SO2
  - **Very safe:** $\Delta t_b = 9.5\,\%$, clang-SO3L
- On average, of the 20 top-performing configurations in each hardware platform and geometry category
  - 96 % (unsafe), 85 % (safe), 78 % (very safe), were statically linked
  - 66 % (unsafe) used `-Ofast`
  - 70 % (unsafe) user native architecture optimizations
- On average, of the 20 worst performing configurations in each hardware platform and geometry category
  - 71 % were dynamically linked
  - 79 % used `-Os`

For ATLAS geometry on PMPE the range in the 20 top-performing configurations, see Table 2, was $\Delta t_b = 17.3\,\%$ for clang-SOfastNL to $\Delta t_b = 9.5\,\%$ for gcc4-SO3N while the corresponding range in worst performing configurations, see Table 3, was $\Delta t_b = -9.3\,\%$ for icc-DOsN to $\Delta t_b = -20.1\,\%$ for gcc6-DOsNL. For the safe configurations, the best performing configuration with $\Delta t_b = 10.2\,\%$ was icc-SO2. Discarding the ICC compiler, the best performing configuration was clang-SO2L with $\Delta t_b = 8.1\,\%$.

Of the top 20 performing configurations, 19 unsafe, 18 safe, and 15 very safe were statically linked, and of the unsafe, 15 were compiled with `-Ofast`, 13 were using native architecture flags, and 8 were using LTO. Of the 20 worst performing configurations, 15 were dynamically linked, 19 were compiled with `-Os`, 9 were using native architecture flags, and 12 were using LTO.

Table 2: Top 20 performing configurations with respect to Δt_b with ATLAS geometry on PMPE

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
clang-SOfastNL	2.02(1)	17.3	2.07(1)	42.91(4)	0.19
gcc8-SOfastNL	2.07(1)	15.1	2.19(1)	42.92(4)	0.17
clang-SOfastN	2.09(1)	14.6	2.12(1)	42.87(5)	0.30
gcc8-SOfastN	2.10(1)	14.0	2.14(1)	42.93(4)	0.15
gcc4-SOfastN	2.10(1)	14.0	2.15(1)	42.98(4)	0.03
icc-SOfastN	2.12(1)	13.3	2.16(1)	42.93(4)	0.14
gcc4-SOfastNL	2.12(1)	13.1	2.16(1)	43.02(4)	0.05
gcc6-SOfastN	2.13(1)	12.9	2.13(1)	42.95(4)	0.11
clang-SOfastL	2.13(1)	12.8	2.21(1)	42.99(4)	0.00
clang-DOfastNL	2.13(1)	12.7	2.13(1)	43.07(4)	0.17
icc-SOfast	2.16(1)	11.8	2.20(1)	42.98(4)	0.02
gcc6-SOfastNL	2.17(1)	11.2	2.23(1)	42.96(4)	0.08
gcc8-SOfast	2.18(1)	10.8	2.21(1)	42.96(4)	0.09
gcc6-SOfast	2.18(1)	10.7	2.24(1)	42.94(4)	0.13
gcc8-SOfastL	2.19(1)	10.4	2.21(1)	43.05(4)	0.12
icc-SO2	2.19(1)	10.2	2.24(1)	42.97(4)	0.05
clang-SO2NL	2.20(1)	10.0	2.24(1)	43.04(4)	0.12
icc-SO3	2.20(1)	10.0	2.25(1)	42.97(4)	0.05
gcc6-SO3N	2.20(1)	9.9	2.22(1)	43.00(4)	0.01
gcc4-SO3N	2.21(1)	9.5	2.26(1)	43.07(4)	0.18

Table 3: 20 worst performing configurations with respect to Δt_b with ATLAS geometry on PMPE

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
icc-DOsN	2.67(1)	−9.3	2.62(1)	42.96(4)	0.08
gcc4-SOsNL	2.70(1)	−10.6	2.70(1)	42.95(4)	0.10
gcc8-SOsL	2.73(1)	−11.5	2.73(1)	42.99(4)	0.00
gcc8-DOsN	2.74(1)	−12.0	2.74(1)	42.96(4)	0.08
gcc6-SOsNL	2.76(1)	−12.9	2.70(1)	43.00(5)	0.01
gcc4-SOsL	2.77(1)	−13.5	2.73(1)	42.99(4)	0.00
gcc6-DOsN	2.79(1)	−14.3	2.77(1)	42.96(4)	0.08

Continued on next page

Table 3: 20 worst performing configurations with respect to Δt_b with ATLAS geometry on PMPE

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc4-DOsN	2.79(1)	−14.4	2.82(1)	42.95(4)	0.11
icc-DOs	2.80(1)	−14.7	2.90(1)	42.97(4)	0.05
icc-DOsL	2.82(1)	−15.6	2.75(1)	42.97(4)	0.05
gcc4-DOs	2.83(1)	−16.0	2.83(1)	42.99(4)	0.00
gcc4-DOsNL	2.86(1)	−17.1	2.88(1)	43.01(4)	0.03
gcc8-DOsNL	2.86(1)	−17.1	2.88(1)	42.98(4)	0.04
gcc6-SOsL	2.86(1)	−17.2	2.78(1)	42.99(4)	0.00
gcc8-DOs	2.87(1)	−17.5	2.86(1)	42.99(4)	0.00
gcc4-DOsL	2.90(1)	−18.6	2.92(1)	42.99(4)	0.00
gcc6-DOs	2.90(1)	−18.8	2.86(1)	42.99(4)	0.00
gcc6-DOsL	2.91(1)	−19.1	2.92(1)	42.99(4)	0.00
gcc8-DOsL	2.92(1)	−19.4	2.86(1)	42.99(4)	0.00
gcc6-DOsNL	2.94(1)	−20.1	2.84(1)	42.96(4)	0.09

For ATLAS geometry on Aurora the range in the 20-top performing configurations, see Table 4, was $\Delta t_b = 16.1\,\%$ for clang-SOfastNL to $\Delta t_b = 9.8\,\%$ for gcc4-SOfastL while the corresponding range in worst performing configurations, see Table 5, was $\Delta t_b = -9.6\,\%$ for gcc8-SOsNL to $\Delta t_b = -33.8\,\%$ for gcc4-DOsNL. For the safe configurations, the best performing configuration with $\Delta t_b = 8.24\,\%$ was gcc4-SO2.

Of the top 20 performing configurations, 19 unsafe and 16 safe were statically linked, and of the unsafe, 14 were compiled with `-Ofast`, 15 used native architecture flags, and 11 used LTO. Of the 20 worst performing configurations, 13 were dynamically linked, 19 were compiled with `-Os`, 9 used native architecture flags, and 12 used LTO.

Table 4: Top 20 performing configurations with respect to Δt_b with ATLAS geometry on Aurora

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
clang-SOfastNL	1.87(1)	16.1	1.85(1)	42.91(4)	0.19
gcc8-SOfastNL	1.89(1)	15.5	1.89(1)	42.92(4)	0.17
gcc6-SOfastN	1.93(1)	13.8	1.92(1)	42.97(4)	0.06
clang-SOfastL	1.93(1)	13.7	1.98(1)	42.99(4)	0.00
gcc4-SOfastN	1.93(1)	13.6	1.87(1)	42.86(5)	0.31
clang-SOfastN	1.93(1)	13.5	1.92(1)	42.87(5)	0.30
gcc8-SOfastN	1.94(1)	13.2	1.90(1)	42.93(4)	0.15
gcc4-SOfastNL	1.94(1)	13.2	1.92(1)	42.95(4)	0.09
gcc6-SO2N	1.94(1)	13.1	1.95(1)	42.90(4)	0.22
gcc6-SOfastNL	1.97(1)	12.0	1.93(1)	42.87(5)	0.28
gcc8-SOfastL	1.97(1)	11.7	1.93(1)	43.05(4)	0.12
clang-DOfastNL	1.98(1)	11.5	1.99(1)	43.07(4)	0.17
gcc6-SOfast	1.99(1)	11.0	2.20(1)	42.94(4)	0.13
gcc8-SOfast	1.99(1)	10.7	2.02(1)	42.96(4)	0.09
gcc8-SO3NL	2.00(1)	10.7	1.96(1)	42.90(4)	0.21
gcc8-SO3N	2.00(1)	10.4	2.06(1)	42.97(4)	0.05

Continued on next page

Table 4: Top 20 performing configurations with respect to Δt_b with ATLAS geometry on Aurora

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc4-SO3NL	2.00(1)	10.4	1.99(1)	42.88(4)	0.27
gcc4-SO3N	2.01(1)	10.1	1.98(1)	43.07(4)	0.18
gcc6-SO2NL	2.01(1)	10.1	1.91(1)	42.96(4)	0.08
gcc4-SOfastL	2.01(1)	9.8	2.01(1)	42.91(4)	0.18

Table 5: 20 worst performing configurations with respect to Δt_b with ATLAS geometry on Aurora

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-SOsNL	2.45(1)	−9.6	2.42(1)	42.91(4)	0.20
gcc4-SOs	2.45(1)	−9.8	2.53(1)	43.03(4)	0.08
gcc4-SOsNL	2.48(1)	−10.8	2.42(1)	42.95(4)	0.10
gcc8-DOsN	2.51(1)	−12.3	2.46(1)	42.96(4)	0.08
gcc4-SOsL	2.51(1)	−12.3	2.49(1)	42.99(4)	0.00
gcc8-SOsL	2.51(1)	−12.4	2.49(1)	42.99(4)	0.00
gcc8-DO3	2.53(1)	−13.3	2.48(1)	42.99(4)	0.00
gcc4-DOsN	2.54(1)	−13.6	2.47(1)	42.95(4)	0.11
gcc6-SOsL	2.54(1)	−13.9	2.55(1)	42.99(4)	0.00
gcc6-DOsN	2.55(1)	−14.3	2.53(1)	42.96(4)	0.08
gcc6-SOsNL	2.56(1)	−14.5	2.50(1)	43.00(5)	0.01
gcc8-DOs	2.57(1)	−15.2	2.54(1)	42.99(4)	0.00
gcc8-DOsNL	2.59(1)	−16.0	2.52(1)	42.98(4)	0.04
gcc4-DOs	2.60(1)	−16.6	2.56(1)	42.99(4)	0.00
gcc6-DOsNL	2.64(1)	−18.2	2.58(1)	42.96(4)	0.09
gcc8-DOsL	2.65(1)	−18.8	2.60(1)	42.99(4)	0.00
gcc6-DOsL	2.68(1)	−20.0	2.64(1)	42.99(4)	0.00
gcc6-DOs	2.70(1)	−21.0	2.57(1)	42.99(4)	0.00
gcc4-DOsL	2.76(1)	−23.6	2.58(1)	42.99(4)	0.00
gcc4-DOsNL	2.99(1)	−33.8	2.83(1)	43.01(4)	0.03

For CMS geometry on PMPE the range in the 20-top performing configurations, see Table 6, was $\Delta t_b = 18.9\,\%$ for icc-SOfastN to $\Delta t_b = 10.1\,\%$ for gcc8-SO2NL while the corresponding range in worst performing configurations, see Table 7, was $\Delta t_b = -15.7\,\%$ for gcc8-DOs to $\Delta t_b = -57.9\,\%$ for icc-DO2L. For the safe configurations, the best performing configuration with $\Delta t_b = 15\,\%$ was icc-SO2. Discarding the ICC compiler, the best performing configuration was clang-SO3L with $\Delta t_b = 9.5\,\%$.

Of the top 20 performing configurations, 19 unsafe, 17 safe, and 15 very safe were statically linked, and of the unsafe, 13 were compiled with `-Ofast`, 13 used native architecture instructions, and 5 used LTO. Of the 20 worst performing configurations, 14 were dynamically linked, 8 were compiled with `-Os`, 16 used LTO (note: primarily ICC), and 8 using native architecture flags.

Table 6: Top 20 performing configurations with respect to Δt_b with CMS geometry on PMPE

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
icc-SOfastN	1.062(4)	18.9	1.077(4)	42.98(4)	0.02
icc-SOfast	1.095(4)	16.4	1.114(4)	42.95(4)	0.05
icc-SO2	1.115(4)	14.9	1.116(4)	42.98(4)	0.00
icc-SO2N	1.126(4)	14.1	1.138(4)	42.97(4)	0.00
gcc6-SOfastN	1.133(4)	13.5	1.168(4)	42.99(4)	0.04
gcc8-SOfastN	1.134(4)	13.4	1.141(4)	43.03(4)	0.12
icc-SO3	1.136(4)	13.3	1.150(4)	42.98(4)	0.00
gcc4-SOfastNL	1.137(4)	13.2	1.142(4)	42.96(4)	0.03
gcc8-SOfastNL	1.138(4)	13.2	1.161(4)	43.00(4)	0.05
gcc4-SOfastN	1.143(4)	12.7	1.135(4)	42.96(4)	0.04
icc-SO3N	1.144(4)	12.7	1.143(4)	42.97(4)	0.00
gcc6-SOfastNL	1.163(4)	11.3	1.177(4)	42.99(4)	0.03
gcc4-SOfast	1.163(4)	11.2	1.178(4)	42.89(4)	0.21
gcc8-SOfastL	1.167(4)	10.9	1.188(4)	42.99(4)	0.03
gcc6-SOfast	1.169(4)	10.8	1.207(4)	42.93(4)	0.10
gcc8-SOfast	1.173(4)	10.5	1.187(4)	42.89(4)	0.19
gcc4-SO3N	1.173(4)	10.5	1.183(4)	42.92(4)	0.12
icc-DOfastN	1.173(4)	10.4	1.191(4)	42.96(4)	0.05
gcc6-SO3N	1.177(4)	10.2	1.173(4)	42.93(4)	0.10
gcc8-SO2NL	1.178(4)	10.1	1.186(4)	42.92(4)	0.13

Table 7: 20 worst performing configurations with respect to Δt_b with CMS geometry on PMPE

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-DOs	1.515(5)	−15.7	1.519(5)	42.98(4)	0.00
gcc4-DOsN	1.520(5)	−16.0	1.488(5)	42.93(4)	0.11
gcc4-DOs	1.521(5)	−16.1	1.486(5)	42.98(4)	0.00
gcc6-DOsL	1.522(5)	−16.2	1.585(6)	42.98(4)	0.00
gcc6-DOs	1.529(5)	−16.7	1.531(5)	42.98(4)	0.00
gcc8-DOsL	1.534(5)	−17.1	1.512(5)	42.98(4)	0.00
gcc6-DOsNL	1.536(5)	−17.2	1.497(5)	42.99(4)	0.03
gcc4-DOsL	1.585(5)	−20.9	1.557(5)	42.98(4)	0.00
icc-SOfastNL	1.843(12)	−40.7	1.840(13)	42.92(4)	0.14
icc-SO2NL	1.857(12)	−41.7	1.871(13)	42.98(4)	0.00
icc-SO2L	1.899(13)	−44.9	1.890(13)	43.00(4)	0.05
icc-SO3NL	1.906(13)	−45.5	1.888(13)	42.99(4)	0.03
icc-SOfastL	1.926(13)	−47.0	1.899(13)	42.89(4)	0.19
icc-SO3L	1.933(13)	−47.5	1.951(13)	43.00(4)	0.05
icc-DO2NL	1.987(14)	−51.6	2.016(14)	43.03(4)	0.12
icc-DOfastNL	2.000(14)	−52.6	1.985(13)	43.02(4)	0.10
icc-DO3NL	2.028(14)	−54.8	2.001(14)	43.03(4)	0.12
icc-DOfastL	2.028(13)	−54.8	2.006(13)	42.96(4)	0.05
icc-DO3L	2.047(14)	−56.2	2.050(14)	43.01(4)	0.08
icc-DO2L	2.069(14)	−57.9	2.020(14)	43.01(4)	0.08

For CMS geometry on Aurora the range in the 20 top-performing configurations, see Table 9, was $\Delta t_b = 13.3\,\%$ for gcc8-SOfastNL to $\Delta t_b = 8.8\,\%$ for gcc4-SO2NL while the corresponding range in worst performing configurations, see Table 8, was $\Delta t_b = -10.5\,\%$ for gcc4-SOL to $\Delta t_b = -22.6\,\%$ for gcc4-DOsL. For the configurations, the best performing configuration with $\Delta t_b = 8.75\,\%$ was clang-SO2L.

Of the top 20 performing configurations, 20 unsafe and 17 safe were statically linked, and of the unsafe, 11 were compiled with `-Ofast`, 15 used native architecture flags, and 10 used LTO. Of the 20 worst performing configurations, 15 were dynamically linked, 17 were compiled with `-Os`, 14 used LTO, and 9 used native architecture flags.

Table 8: 20 worst performing configurations with respect to Δt_b with CMS geometry on Aurora

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc4-SOsL	1.306(4)	−10.5	1.265(5)	42.98(4)	0.00
gcc8-SOsL	1.307(4)	−10.6	1.274(5)	42.98(4)	0.00
gcc4-SOsNL	1.309(4)	−10.7	1.258(5)	42.87(4)	0.25
gcc6-SOsL	1.321(5)	−11.8	1.293(5)	42.98(4)	0.00
gcc6-SOsNL	1.328(5)	−12.4	1.279(5)	42.92(4)	0.13
gcc6-DO2NL	1.333(5)	−12.8	1.285(4)	42.93(4)	0.10
gcc8-DOsN	1.337(5)	−13.1	1.321(5)	42.92(4)	0.13
gcc4-DO3L	1.346(4)	−13.9	1.296(4)	42.98(4)	0.00
gcc4-DOsN	1.362(5)	−15.2	1.298(4)	42.93(4)	0.11
gcc8-DOsNL	1.374(5)	−16.3	1.323(5)	43.03(4)	0.12
gcc4-DOsNL	1.383(5)	−17.0	1.332(5)	43.00(4)	0.06
gcc8-DOs	1.383(5)	−17.0	1.340(5)	42.98(4)	0.00
gcc6-DOsN	1.385(5)	−17.2	1.337(5)	42.89(4)	0.19
gcc8-DOsL	1.386(5)	−17.3	1.376(5)	42.98(4)	0.00
gcc6-DOsNL	1.392(5)	−17.8	1.341(5)	42.99(4)	0.03
gcc6-DO3L	1.398(4)	−18.3	1.308(4)	42.98(4)	0.00
gcc6-DOs	1.406(5)	−18.9	1.356(5)	42.98(4)	0.00
gcc4-DOs	1.415(5)	−19.7	1.338(5)	42.98(4)	0.00
gcc6-DOsL	1.423(5)	−20.4	1.367(5)	42.98(4)	0.00
gcc4-DOsL	1.450(5)	−22.6	1.385(5)	42.98(4)	0.00

Table 9: Top 20 performing configurations with respect to Δt_b with CMS geometry on Aurora

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-SOfastNL	1.024(4)	13.3	1.020(4)	43.00(4)	0.05
gcc4-SOfastNL	1.026(4)	13.2	1.034(4)	42.84(4)	0.31
gcc8-SO2NL	1.026(4)	13.2	1.012(4)	42.92(4)	0.13
gcc4-SOfastN	1.027(4)	13.1	1.025(4)	42.97(4)	0.03
gcc8-SO2N	1.028(3)	13.0	1.005(4)	42.91(4)	0.15
gcc8-SOfastN	1.031(4)	12.7	1.027(4)	43.03(4)	0.12
gcc6-SOfastNL	1.053(4)	10.9	1.032(4)	42.91(4)	0.16
gcc6-SOfastN	1.053(4)	10.9	1.031(4)	42.92(4)	0.13
gcc8-SO3NL	1.055(4)	10.7	1.060(4)	42.91(4)	0.15
gcc4-SOfast	1.057(4)	10.6	1.050(4)	42.89(4)	0.21
gcc4-SO3NL	1.058(4)	10.5	1.030(4)	43.00(4)	0.06
gcc4-SOfastL	1.062(4)	10.2	1.046(4)	43.04(4)	0.14
gcc8-SO3N	1.066(4)	9.8	1.064(4)	42.93(4)	0.11
gcc6-SO3NL	1.068(4)	9.7	1.064(4)	42.96(4)	0.04
gcc4-SO3N	1.070(4)	9.5	1.046(4)	42.92(4)	0.12
gcc8-SOfastL	1.070(4)	9.5	1.053(4)	42.99(4)	0.03
gcc8-SOfast	1.071(4)	9.4	1.055(4)	42.89(4)	0.19
gcc4-SO2N	1.073(4)	9.2	1.076(4)	42.94(4)	0.09
gcc6-SOfast	1.075(4)	9.0	1.088(4)	42.93(4)	0.10
gcc4-SO2NL	1.078(4)	8.8	1.062(4)	42.96(4)	0.04

4.2 Physics validation

For the second part of the project, we have carried out measurements of the energy deposited per event, E, distribution for each configuration. Furthermore, we have recorded the final random number generated by HepExpMT, X, for each configuration in order to determine if any differences in random number generation have occurred throughout the simulation process. We will present here both the reference results as well as demonstrate three seemingly distinct categories of ΔE.

The reference E distributions for both ATLAS and CMS can be found in Figure 10. The distribution is similar between the two detectors and identical between hardware platforms. The shape of the distribution is a single peak with a small bump at near $E = 50\,\text{GeV}$, corresponding to events that deposited all of their energy during the simulation. It is important to note here that the validation code considers any energy deposited through any interaction regardless of if the location in the detector is capable of measuring energy.

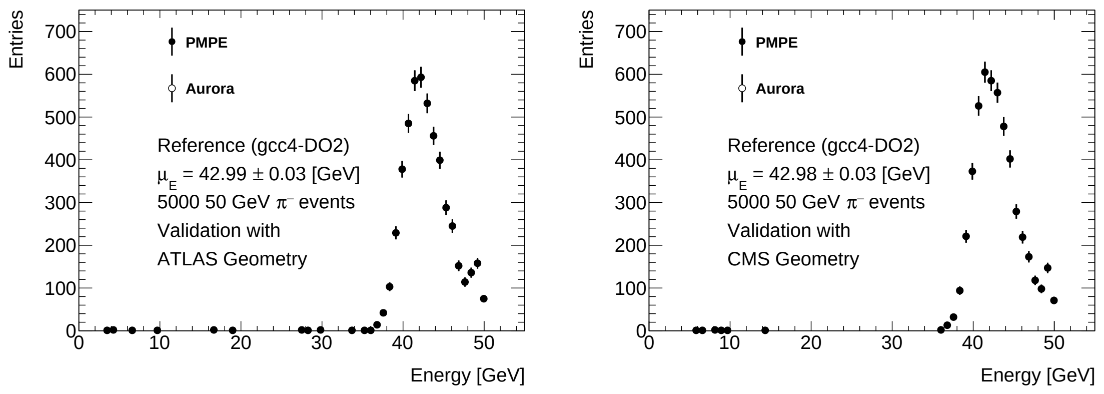


Figure 10: Reference energy measurements with the validation code using the ATLAS geometry (left) and the CMS geometry (right) on both Aurora and PMPE. Average energy deposited within the geometry per event is $\mu_E = 42.99(3)\,\text{GeV}$ (ATLAS) and $\mu_E = 42.98(3)\,\text{GeV}$ CMS. Note: The distributions within the figures are identical event by event.

While compiling with LTO, GCC 6 and 8 both report multiple ODR violations in the various Geant4 manager classes and some in its internal libraries. Energy distributions are event by event identical within a given configuration both between Aurora and PMPE and between two independent runs for all configurations except those listed in Table 10.

The final random number generated in HepExpMT was

- $X = 0.713087$ (ATLAS), $X = 0.70476$ (CMS) for the Clang and ICC compilers
- $X = 0.137428$ (ATLAS), $X = 0.231357$ (CMS) for the GCC compilers

This difference indicates that somewhere throughout the simulation process, the sequence of random numbers that are generated by each compiler family diverges. As the reproducibility of a Monte Carlo experiment is dependent on the stability of its random number generators, which is why an arbitrary but defined number is used to start the random number generation, finding evidence of instability is concerning. This observed difference does not tell us where it originates or which, if any, of the sequences is correct.

We observe three seemingly distinct sources of ΔE throughout our measurements of roughly equal magnitude:

1. **Unsafe optimizations:** Applying unsafe math optimizations through `-Ofast`, using native architecture instructions, or using the ICC compiler.
2. **Choice of compiler:** Configurations using one of the GCC compilers will produce different output from configurations using Clang, even for safe configurations.
3. **Platform differences:** For a set of configurations using `-Ofast` with GCC 4 and 6, we see differences between hardware platforms for the same configurations (see Table 10). Notably, one of the configurations in this category does not use native architecture optimizations, where it could be expected to see platform-dependent differences.

For the unsafe optimizations category and choice of compiler category respectively, a comparison between the E distribution of gcc8-SOfastNL with the CMS geometry and clang-DO2 with the ATLAS geometry on PMPE with the corresponding reference configuration can be found in Figure 11. For both examples, the distributions are clearly distinct from the reference distribution but still similar, $\Delta E = 0.1\,\%$. Finally, a list of all nine configurations in the platform category can be found in Table 10. All of the platform category configurations are using `-Ofast` with either GCC 4 or 6, and all but one, gcc4-DOfastL, uses native architecture optimizations.

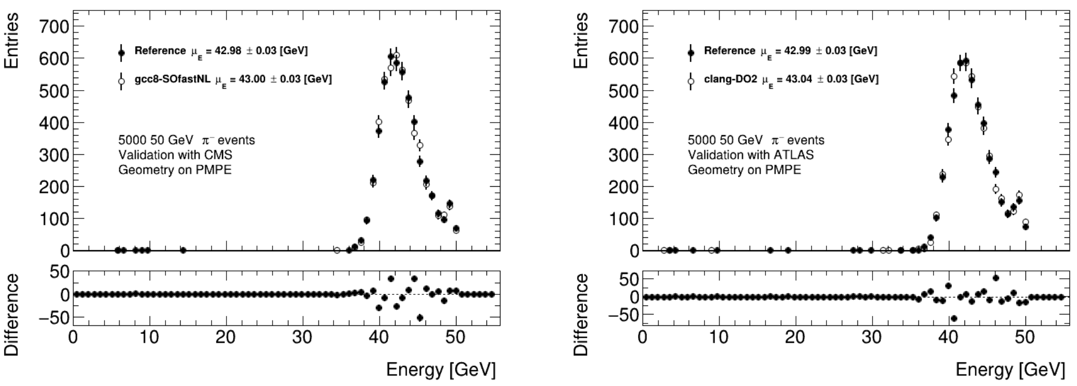


Figure 11: Energy deposition in two configurations exhibiting two of the seemingly distinct categories of sources of ΔE compared with the corresponding reference configurations. The gcc8-SOfastNL (left) uses both native architecture optimizations and the unsafe math optimizations while clang-DO2 (right) uses a different compiler than the reference configuration. Average energy deposited per event is $\mu_E = 43.00(3)\,\mathrm{GeV}$ (left) and $\mu_E = 43.03(3)\,\mathrm{GeV}$ (right). Corresponding absolute relative difference with respect to the reference configuration $\Delta E = 0.1\,\%$ (left) $\Delta E = 0.1\,\%$ (right).

Table 10: Average energy deposited per event for all configurations where the E distribution differed between Aurora and PMPE. This behavior is unique for the GCC 4.9.4 and GCC 6.3.0 compilers using `-Ofast`. Notably, one of the nine configurations is not using native architecture optimizations, where such differences could be expected.

Configuration	ATLAS, PMPE (GeV)	ATLAS, Aurora (GeV)	CMS, PMPE (GeV)	CMS, Aurora (GeV)
gcc4-DOfastN	42.93	42.97	42.95	42.99
gcc4-DOfastNL	42.92	42.91	43.00	42.93
gcc4-SOfastL	42.95	42.91	43.03	43.04
gcc4-SOfastN	42.98	42.86	42.96	42.97
gcc4-SOfastNL	43.02	42.95	42.96	42.84
gcc6-DOfastN	42.98	42.89	42.91	42.98
gcc6-DOfastNL	42.95	42.93	42.95	43.02
gcc6-SOfastN	42.95	42.97	42.99	42.92
gcc6-SOfastNL	42.96	42.87	42.99	42.91

5 Discussion

5.1 Physics validation

We observe at least three sources of ΔE in the E measurements. As expected, the unsafe configurations, native architecture flags and `-Ofast`, and configurations with ICC, which use unsafe methods by default, do not produce the same output as the corresponding safe configurations. However, while the safe options of the three GCC compilers produce event-wise identical output to each other and all safe configurations with Clang produce event-wise identical output to each other, the GCC family and Clang do not produce the same output ($\Delta E = 0.1\,\%$).

What distinguishes GCC and Clang (and ICC) is that they output different values for X. Notably, there is no difference between safe and unsafe methods to be seen in X. As discussed in Section 2.3.1, compilers are allowed to optimize based on UB. If there is UB either in Geant4 or the underlying random number generator code from CLHEP, it is possible for two compilers to produce different output.

As can be seen in Table 10, we observe a platform-specific difference in the E distribution for some of the `-Ofast` configurations with GCC 4.9.4 and GCC 6.2.0. This is a bit troubling as accounting for what hardware simulations will run on is going to be a lot harder than with the build configurations. Furthermore, one of the configurations that is part of Table 10 does not use native architecture flags, gcc4-DOfastL. If platform dependence was limited to runs with the native architecture flags, these results would not be outside of what we expected. Given that the issue seems limited to the two earlier versions of GCC, it is possible that this is a bug which has since been patched. We could also perform the validation runs on hardware which differs significantly from Aurora and PMPE, which both run Intel CPU's from the same generation.

As $\Delta E < 0.35\,\%$ for all configurations, for compute-intensive simulations with a precision that cannot resolve such differences using unsafe configurations could well be considered. Regardless, the difference between the GCC and Clang compilers that we have observed do imply that simulated data that does not report how it was built might not be perfectly reproducible. Whether or not our source of ΔE is UB in either Geant4 or CLHEP, it is not surprising that large and complex software can contain these kinds of bugs. Therefore, if we do not know the compiler that was used to build any piece of scientific software of similar scope, we cannot know if optimizations based on UB might have altered the results.

5.2 Performance of Geant4 simulations

As can be seen in Figure 4 and Figure 5, there are at least two kinds of events present, a fast and a slow kind. A straight-forward explanation of this phenomenon lies in the detector geometry. Our π^- particles are distributed isotropically, and their trajectories will, therefore, have different η. The detectors that any given η direction point through are very different (see Table 1 for details). As discussed in Section 2.2.2, Geant4 performance is expected to be sensitive to the number of particles produced as a π^- passes through the detector; the length traveled through calorimeters which produce particle showers should have a clear impact on performance. Therefore. it seems likely that the event categories correspond to trajectories passing through either the barrel or end cap calorimeters. Our GDML file for ATLAS also lacks the EMEC (Electromagnetic Endcap Calorimeter) calorimetry in the forward region.

While it is hard to draw overarching conclusions, and there are exceptions to every rule, there are some observations we can make with some confidence. Among the top-performing configurations for both ATLAS and CMS with respect to Δt_b, we find: statically linked configurations (96 %), configurations using `-Ofast` (66 %), and configurations using native architecture optimizations (70 %). On the flip side, the slower region is dominated by dynamic linking (71 %) and `-Os` (79 %).

These results are not surprising, optimizing for small size is not going to be helpful when your code is already large, and, as discussed in Section 2.3.1, there is a runtime component of dynamic linking which has a prize. Finally, the `-Ofast` and native architecture optimizations are primarily optimizations of floating-point code, which is a large portion of what simulation code like Geant4 is occupied with.

It is interesting to compare the relative performance improvements presented in Figures 6 to 8. In general, results are more stable across hardware platforms and geometry with the GCC compilers, although there are outliers such as gcc6-DOfastNL in Figure 6. Builds with Clang, as can be seen in Figure 7, improved more on PMPE than on Aurora, and there are several cases where the configuration performs better with one geometry than the other.

As can be seen in Table 7 or Figure 8, something goes wrong with several ICC configurations where LTO is used with performance drops in the range $-58\,\% < \Delta t_b < -40\,\%$ for the CMS geometry. Comparing the distributions in Figure 9, we see that the pathological distribution is significantly different. Rather than a sharp peak for fast events and a broader peak for slow events, there is a single peak with a long tail. It is hard to speculate about the sources of this issue without further research. Regardless, this result highlights the importance of measuring before and after switching configurations. It is regrettable that we were unable to compile Geant4 with the version of ICC on Aurora as it would be interesting to see if this issue occurs on both hardware platforms.

We do not see any significant difference in runtime performance that can be attributed to the choice of compiler or usage of LTO in these studies. However, performance is not the only reason why we could benefit from utilizing newer compilers or techniques such as LTO. As we have seen, LTO has already provided information regarding potential ODR violations. These warnings were only available in the later versions of GCC. Furthermore, the C++ language is evolving, and being able to use more recent versions of compilers can allow physicists to write code that is more likely to be correct without giving up the performance advantages of compiled languages. As discussed earlier, GCC 8 does not seem to produce the same issue as the earlier versions regarding platform differences in Table 10, suggesting that something might have been patched.

5.3 Outlook

Our primary purposes have been to investigate the potential for improving the runtime performance of Geant4 simulations for ATLAS using different build time methods and perform validation of the physics output of these methods. Studying any individual method in detail or investigating any of the curious observations, such as non-zero ΔE with safe methods, in detail is beyond the scope of this project. However, the measurement setup we have created could be incorporated into future studies and we can, based on our theoretical discussion and our observations, here propose some potential avenues.

The observed differences in X and corresponding non-zero ΔE between safe configurations with the GCC family and Clang is arguably the most significant result of our validation measurements. While the resulting relative difference is small ($\Delta E = 0.1\,\%$), the core issue could, if part of software which is used throughout ATLAS such as CLHEP or Geant4, produce worse errors elsewhere. This is particularly true if the core issue is UB.

The validation code could be instrumented to record the next random number in the event action sections, either at select events or for every event. This would provide guidance for further investigations. If the recorded numbers are different before the first event, there is no need to look into the event processing parts of Geant4. If the difference appears first during the event processing state, the source could lie in a specific physics process or model. Since the process is deterministic, the events near the first deviation could then be explored in further detail.

Another fairly simple method that could determine if the source of UB lies in the random number generation of CLHEP is the following. As Geant4 is modular in nature, we can exchange the current random number generator for a generator that produces a known stream of random numbers, e.g., by reading from a preallocated array. If the source of UB lies in the random number generation, then X for runs with this new generator should be equal across compilers. As the validation code only uses standard Geant4 features, this should be trivial to implement. A similar method that works if the difference is due to UB in CLHEP would be to compile CLHEP without optimizations. If the differences in X disappear, we have our culprit.

Diagnosing UB is challenging, especially if the UB in question is ODR violation. We already have some evidence for ODR violation in Geant4 from GCC version 6 and 8 when using LTO. There are further methods that could be integrated into our setup that could provide significant help. GCC and Clang both ship with so-called sanitizer options for both address/memory safety and diagnosing common sources of UB by inserting compiler guided instrumentation code into the binary that can help out.

The scope of this project has been limited to a single particle, π^-, with a specific energy. It would be a good idea to study the same setup with different particles and energies or to study different physics by altering the physics list that we use. To study this setup further, one could attempt to identify either what parts of the simulation code are consuming the most CPU resources or what parts seem to benefit more or less from the kinds of transformations we have studied. This could be done using operating tools such as perf, which our setup already has support for.

A simple method for investigating this difference further, which would be straight forward to integrate into the current validation code, would be recording η for each event along with the t_v measurement and separating the two regions. In order to ensure that we do not violate our assumption in Equation (6) regarding $t_{measured}$, it would be recommended that the current measurement of E be disabled during the measurement. If possible, finding a GDML file which includes the EMEC calorimeters could also provide a better picture.

`-Ofast` is associated with some degree of ΔE. However, as mentioned in Section 2.3.1, `-Ofast` is a group of optimizations. It would be interesting to compare these options on their own. It is possible that there are options that contribute more to ΔE but provide little performance improvement in which case, we could mitigate some of the energy difference and still get most of the benefits of `-Ofast`.

5.4 Conclusion

We have carried out a broad survey of 156 configurations to see if the performance of Geant4 simulations for the ATLAS experiment can be improved at build time while simultaneously performing validation of the results. The project setup that we have developed, thanks to building an isolated environment, has proven to be easy to adapt to new environments with drastically different properties.

First and foremost, using statically linked libraries seems to produce a consistent performance improvement, although measurements before and after will always be necessary given examples such as Figure 9, without affecting the physics results. However, doing so is not going to be easy, Athena was developed with “Extensive use of dynamic libraries” as one of its three goals [7]. Researching how static linking can be introduced into the ATLAS software ecosystem while maintaining the simplicity for users that Athena currently provides could prove to be a fruitful endeavor.

We do not believe it is going to be worth the effort to continue studying `-Os`. However, `-Ofast` has shown to be capable of producing decent performance improvements. We have proposed some ways that we could study `-Ofast` further and potentially minimize ΔE while retaining some of the performance improvements.

We have seen some evidence of native architecture flags providing a performance improvement on its own. However, given that implementing these optimizations is likely much harder on a large scale, using the wrong instructions can either slow down or crash a system, further studies of utilizing them might not be worth the effort. For use on standalone systems, however, it is possible that individual researchers can gain performance by using the flags in their computing. As always, measurements of the impact before and after turning on these optimizations will be important to avoid pathological cases.

The observation of both hardware-dependent ΔE and ΔE in safe configurations has interesting implications, even though their magnitude is relatively small. While the differences likely are negligible in most situations, any results relying on simulations such as ours that do not report how they were built and on what hardware they ran on might not be perfectly reproducible. This is not to say that these results are unusable, we would be in serious trouble if it was, but there is an inherent uncertainty originating in how we build our projects that we should be aware of and document. For the second category, potential UB or ODR violations are serious; a program containing them has no guarantees to be correct. It is possible that what is the source of a small ΔE today can become much more serious as ATLAS software evolves.

In this project, we have found that runtime performance can be improved with $\sim 19\,\%$ in the best case and likely $\sim 10\,\%$ at least if the build method is chosen well. With all of this in mind, our recommendations for the ATLAS experiment are:

- Continue research into the possibility of using static libraries in Athena. Static linking seems to provide consistent performance benefits without producing any ΔE.
- Consider utilizing unsafe configurations for simulations where the required resolution can allow it, and testing before and after to ensure that there is a performance gain, and that situations like Figure 9 are avoided.
- Consider reporting build configuration when reporting simulated data if reproducibility is to be claimed.
- Consider integrating LTO with recent versions of compilers into the development and testing of ATLAS simulation software to be able to catch issues such as ODR.
- Investigate the source of ΔE in the safe configurations with Clang to be able to determine its severity, in particular with respect to UB and ODR violations.

6 Acknowledgments

I would like to take this opportunity to thank several people. First of all, Caterina Marcon and Oxana Smirnova for providing me with a very interesting project, and for their supervision and support along the way. I have learned a lot throughout this process. I am especially grateful to Caterina, who managed to obtain the GDML files for ATLAS, which this project would have been limited without. Secondly, I would like to thank David Smith and Markus Schulz for both their support with my project and for the time I got to spend working in their offices. Hopefully, the next student that visits your offices will not have their stay cut short by a worldwide pandemic.

7 Bibliography

References

[1] ATLAS Collaboration. The atlas experiment at the cern large hadron collider. *Journal of Instrumentation*, 3(08):S08003–S08003, 2008.

[2] The CMS Collaboration. The cms experiment at the cern lhc. *Journal of Instrumentation*, 3(08):S08004–S08004, 2008.

[3] S. Agostinelli et al. Geant4-a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[4] Bogdan Povh, Klaus Rith, Christoph Scholz, Frank Zetsche, and Werner Rodejohann. *Particles and Nuclei*. Graduate Texts in Physics. Springer Berlin Heidelberg, 2015.

[5] William R. Leo. *Techniques for Nuclear and Particle Physics Experiments*. Springer Berlin Heidelberg, 1994.

[6] R. Chytracek, J. Mccormick, W. Pokorski, and G. Santin. Geometry description markup language for physics simulation and analysis applications. *IEEE Transactions on Nuclear Science*, 53(5):2892–2896, 2006.

[7] ATLAS Collaboration. *ATLAS Computing: technical design report*. Technical Design Report ATLAS. CERN, Geneva, 2005.

[8] ATLAS Collaboration. The atlas simulation infrastructure. *The European Physical Journal C*, 70(3):823–874, 2010.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[10] John R Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[11] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth edition, December 2017.

[12] Edward L. Robinson. *Data Analysis for Scientists and Engineers*. Princeton University Press, Princeton, 2017.

[13] Tan Li, Yufei Ren, Dantong Yu, and Shudong Jin. Analysis of numa effects in modern multicore systems for the design of high-performance data transfer applications. *Future Generation Computer Systems*, 74:41 – 50, 2017.

[14] Leif Lönnblad. Clhep-a project for designing a c++ class library for high energy physics. *Computer Physics Communications*, 84(1-3):307–316, 1994.

[15] Steven Farrell, Andrea Dotti, Makoto Asai, Paolo Calafiura, and Romain Monnard. Multithreaded geant4 on the xeon-phi with complex high-energy physics geometry. 1 2016.

[16] Rene Brun and Fons Rademakers. Root - an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1-2):81–86, 1997.

[17] John Regehr. C compilers disprove fermat's last theorem. `https://blog.regehr.org/archives/140`, 4 2010. Accessed: 2020-03-04.

Appendix

A Compiler optimization by example

We will be demonstrating compiler optimization briefly here using pseudo-assembly code. To get started, let's look at the syntax we will be using and the instructions we will see

```
.L5
```

this is a label that other parts of the code can jump to.

```
sub eax eax
```

This is an instruction. The target, i.e. where the result will be stored, is the first argument. The purpose of the second argument depends on the instruction but we will refer to it as the source. Here, it corresponds to the value that is to be subtracted from the target. Many instructions will in addition to performing the operation set a number of flags. In our example, this could be a flag which specifies if the previous operation resulted in a negative number. In this example, both the optional source and target arguments are registers (the same in this case). Registers are small pieces of memory that can hold a single piece of data such as a number or an address.

Two important categories of instructions that we will encounter are the jumps and moves.

```
mov rax rdi
mov QWORD PTR [rbp-24], rdi
```

Mov, as the name suggests, takes a value from its source and puts it into the target. While mov is straight forward to understand, the arguments it takes can be quite interesting. In the first example, we move a value from one register into another. In the second example, we are calculating an address ([rbp-24] would be the address corresponding to value in rbp - 24) and using this location as our target (QWORD PTR [rbp-24]). Effectively, we are carrying out two steps in one.

```
cmp edx, eax
jle .L5
```

Jump instructions will jump to a label depending on a condition. The condition is not part of the jump instruction but instead, the jump instructions will use the flags set by the previous instruction. In our example, we calculate (edx - eax) and then jump to .L5 if the result is zero or negative (jle: jump if less than or equal).

Finally, there is the lea instruction. Lea stands for Load Effective Address and is primarily intended to be used to calculate addresses. However, it can be utilized to perform arithmetic

```
mov rax 4
lea rdx [0 + rax * 4]
```

The lea instruction here calculates the value that would correspond to the address 0 + 4*4 and stores it in rdx.

Now we can get started with some optimizations. We will be working with the following piece of C-style C++ code.

```
void bubblesort(int* arr, int n) {
    for (int i = n-1; i > 0; i--) {
        for (int j = 1; j < i+1; ++j) {
            if (arr[j] > arr[j+1]) {
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

We can convert this to assembly with GCC as follows (assuming the function is stored in bubblesort.cxx)

```
g++ -masm=intel -c bubblesort.cxx -S -o - | c++filt | grep -vE '\s+\.[a-z]'
```

The first flag tells GCC which assembly dialect we want to use. -S is the flag that tells GCC to stop after generating assembly, and -o - means put the output directly to standard output. We then do some filtering to make things easier to read.

By default, the compiler will generate a lot of boilerplate which we will skip here. The reader is encouraged to try things out for themselves if they are interested.

The arguments of the function are passed in the registers rsi and rdi. rdi is our array, and rsi is n. Note that registers beginning with r are 64-bit registers but have a corresponding name for 32-bit values beginning with e (e.g. rsi, esi are the same register). rbp is a register containing information about where the stack starts.

A.1 The unoptimized code

At -O0, most of the generated assembly is spent computing addresses. Registers are, in general, not used to store values for very long, so the same computation has to be performed multiple times. Additionally, a fair amount of instructions are spent managing 32-bit values in 64-bit registers.

```
bubblesort(int*, int):
.LFB0:
        ;; Stack magnagement. Decrement stack pointer (top
    ;; of stack is at low address)
        ;; Store current base pointer, i.e. base of previous
    ;; stack, at the current top of the stack (bottom of
    ;; new stack frame)
    push    rbp
        ;; Copy current stack pointer, top of previous
    ;; stack, into the base pointer
    mov     rbp, rsp
        ;; Copy function call variables from register into
    ;; local variable
        ;; rdi is the address of the array to sort
        ;; esi has the value of n
    mov     QWORD PTR [rbp-24], rdi
    mov     DWORD PTR [rbp-28], esi
        ;; Copy n from stack into eax
    mov     eax, DWORD PTR [rbp-28]
        ;; Subtract one and store into stack variable i
    sub     eax, 1
    mov     DWORD PTR [rbp-4], eax
.L6:
        ;; Jump to end if i is zero or negative
    cmp     DWORD PTR [rbp-4], 0
    jle     .L7
        ;; j = 1
    mov     DWORD PTR [rbp-8], 1
.L5:
        ;; Store i in register and compare with j. Jump to end
        ;; of inner loop if i - j < 0
        mov     eax, DWORD PTR [rbp-4]
    cmp     eax, DWORD PTR [rbp-8]
    jl  .L3
        ;; Copy j into register
    mov     eax, DWORD PTR [rbp-8]
        ;; Extend size of the register to 64 bytes
    cdqe
        ;;  Store the result of the computation 0 + j * 4 in
    ;; rdx
    lea     rdx, [0+rax*4]
        ;; Copy value of arr (an address) into rax
    mov     rax, QWORD PTR [rbp-24]
        ;; Add 0 + j * 4 into the base address of arr
    add     rax, rdx
        ;; Copy value att arr[0 + 0 + j * 4], i.e.
        ;; arr[j] into register
    mov     edx, DWORD PTR [rax]
        ;; Store j into register and extend size to 64-bit
    mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; k = j + 1
    add     rax, 1
        ;; Compute 0 + (k) * 4, i.e. 0 + (j+1) * 4
    lea     rcx, [0+rax*4]
        ;; Copy address of arr into rax
    mov     rax, QWORD PTR [rbp-24]
        ;; Compute &arr[j+1] into rax
    add     rax, rcx
        ;; Store arr[j+1] in eax
    mov     eax, DWORD PTR [rax]
        ;; Compute edx - eax and jump if negative or zero
    cmp     edx, eax
    jle     .L4
        ;; Copy j into register and extend
    mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; Compute 0 + j * 4
    lea     rdx, [0+rax*4]
        ;; Copy address of arr into register
    mov     rax, QWORD PTR [rbp-24]
        ;; Compute arr[j] and store into variable on stack
    ;; (tmp)
    add     rax, rdx
    mov     eax, DWORD PTR [rax]
    mov     DWORD PTR [rbp-12], eax
        ;; Copy j into register, extend and add 1, compute
    ;; corresponding length into memory
    mov     eax, DWORD PTR [rbp-8]
    cdqe
    add     rax, 1
    lea     rdx, [0+rax*4]
        ;; Compute address of arr[j+1]
    mov     rax, QWORD PTR [rbp-24]
    add     rax, rdx
    mov     edx, DWORD PTR [rbp-8]
        ;; Perform sign-extension for a register other than
    ;; rax/eax
    movsx   rdx, edx
        ;; Compute &arr[j+1] into rcx
    lea     rcx, [0+rdx*4]
    mov     rdx, QWORD PTR [rbp-24]
    add     rdx, rcx
        ;; Turn the value back to 32-bits
    mov     eax, DWORD PTR [rax]
        ;; Store arr[j+1] into arr[j]
    mov     DWORD PTR [rdx], eax
        ;;
    mov     eax, DWORD PTR [rbp-8]
    cdqe
    add     rax, 1
    lea     rdx, [0+rax*4]
    mov     rax, QWORD PTR [rbp-24]
    add     rdx, rax
    mov     eax, DWORD PTR [rbp-12]
        ;; Store arr[j] into arr[j+1]
    mov     DWORD PTR [rdx], eax
.L4:
        ;; j++
    add     DWORD PTR [rbp-8], 1
    jmp     .L5
.L3:
    ;; i--
        sub     DWORD PTR [rbp-4], 1
    jmp     .L6
.L7:
    nop
    pop     rbp
    ret
.LFE0:
```

A.2 Local optimization

Okay, that was a lot of assembly. First, we will split the code into blocks. A block is defined by a single entry and single output. We will then see what optimizations we can perform within a block without any assumptions regarding other blocks. These are called local optimizations.

First out we have the stack management and setup block

```
bubblesort(int*, int):
.LFB0:
    push    rbp
    mov     rbp, rsp
    mov     QWORD PTR [rbp-24], rdi
    mov     DWORD PTR [rbp-28], esi
    mov     eax, DWORD PTR [rbp-28]
    sub     eax, 1
    mov     DWORD PTR [rbp-4], eax
```

We can begin by removing the stack management (-fomit-stack-pointer, turned on at -O1). We already have n in a register, but we cannot assume that the memory location that we store it into is not read later. We can skip one read from memory by moving n directly from esi to eax.

```
mov QWORD PTR [rbp-24], rdi
mov DWORD PTR [rbp-28], esi
mov eax, esi
sub eax, 1
mov DWORD PTR [rbp-4], eax
```

Not much shorter, but definitely more efficient. We now have three blocks that are trivial, so we show them all here as one.

```
;; First trivial block
.L6:
    cmp     DWORD PTR [rbp-4], 0
    jle     .L7
;; Second trivial block
    mov     DWORD PTR [rbp-8], 1
;; Third trivial block
.L5:
    mov     eax, DWORD PTR [rbp-4]
    cmp     eax, DWORD PTR [rbp-8]
    jl  .L3
```

Now we get to the point where we will be doing some computation. This is the first part of the inner for-loop. First, we load 1 into j (1 was stored at rbp-8 in the second trivial block). Next, we calculate the offset into the array which j corresponds to using the lea-instruction. We then load arr[j] and repeat the process for j+1. Once everything is loaded, we compare and jump to the j++ if the condition fails.

```
    ;; 1, j (load)
        mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; 2, &[j]
    lea     rdx, [0+rax*4]
        ;; 3, &arr (load)
    mov     rax, QWORD PTR [rbp-24]
        ;; 4, &arr + &[j]
    add     rax, rdx
        ;; 5, arr[j] (load)
    mov     edx, DWORD PTR [rax]
        ;; 6, j (load)
    mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; 7, j+1
    add     rax, 1
        ;; 8, &[j+1]
    lea     rcx, [0+rax*4]
        ;; 9, &arr (load)
```

```
mov     rax, QWORD PTR [rbp-24]
    ;; 10, &arr + &[j+1]
add     rax, rcx
    ;; 11, arr[j+1] (load)
mov     eax, DWORD PTR [rax]
cmp     edx, eax
jle     .L4
```

In this block, we compute the following things:

1. j (load), r8
2. &[j]
3. &arr (load), r9
4. &arr + &[j]
5. arr[j] (load)
6. j (load), r8
7. j+1
8. &[j+1]
9. &arr (load), r9
10. &arr + &[j+1]
11. arr[j+1] (load)

We compute (load) j and &arr twice, we could keep it in a register instead. Let us use r8 and r9.

```
;; 1, j (load)
    mov     r8, DWORD PTR [rbp-8]
cdqe
    ;; 2, &[j]
lea     rdx, [0+rax*4]
    ;; 3, &arr (load)
mov     r9, QWORD PTR [rbp-24]
    ;; 4, &arr + &[j]
add     rdx, r9
    ;; 5, arr[j] (load)
mov     edx, DWORD PTR [rax]
    ;; 6, j + 1
add     r8, 1
    ;; 7, &[j+1]
lea     rcx, [0+r8*4]
    ;; 8, &arr + &[j+1]
add     rcx, r9
    ;; 9 arr[j+1] (load)
mov     eax, DWORD PTR [rcx]
cmp     edx, eax
jle     .L4
```

We have saved 2 instructions. We will be able to reuse this idea in the next block. Beware, this is a big one. We are now inside the if-statement.

```
    ;; 1, j (load)
    mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; 2, &[j]
    lea     rdx, [0+rax*4]
        ;; 3, &arr (load)
    mov     rax, QWORD PTR [rbp-24]
        ;; 4, &arr + &[j]
    add     rax, rdx
        ;; 5, arr[j] (load)
    mov     eax, DWORD PTR [rax]
        ;; 6, tmp = arr[j] (store)
    mov     DWORD PTR [rbp-12], eax
        ;; 7, j (load)
    mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; 8, j + 1
    add     rax, 1
        ;; 9, &[j+1]
    lea     rdx, [0+rax*4]
        ;; 10, &arr (load)
    mov     rax, QWORD PTR [rbp-24]
        ;; 11 &arr + &[j+1]
    add     rax, rdx
        ;; 12, j (load)
    mov     edx, DWORD PTR [rbp-8]
    movsx   rdx, edx
        ;; 13, &[j]
    lea     rcx, [0+rdx*4]
        ;; 14, &arr (load)
    mov     rdx, QWORD PTR [rbp-24]
        ;; 15, &arr + &[j]
    add     rdx, rcx
        ;; 16, arr[j] (load)
    mov     eax, DWORD PTR [rax]
        ;; 17, arr[j+1] = arr[j] (store)
    mov     DWORD PTR [rdx], eax
        ;; 18, j (load)
    mov     eax, DWORD PTR [rbp-8]
    cdqe
        ;; 19, j+1
    add     rax, 1
        ;; 20, &[j]
    lea     rdx, [0+rax*4]
        ;; 21, &arr (load)
    mov     rax, QWORD PTR [rbp-24]
        ;; 22, &arr + &[j]
    add     rdx, rax
        ;; 23, tmp (arr[j]) (load)
    mov     eax, DWORD PTR [rbp-12]
        ;; 24, arr[j+1] = tmp (store)
    mov     DWORD PTR [rdx], eax
```

The computation this time is as follows

1. j (load), r8
2. &[j]
3. &arr (load), r9
4. &arr + &[j], r10
5. arr[j] (load), arr[r10], r13
6. tmp = arr[j] (store), tmp = arr[r10]
7. j (load), r8
8. j + 1, r11
9. &[j+1]
10. &arr (load), r9
11. &arr + &[j+1], r12
12. j (load), r8
13. &[j]
14. &arr (load), r9
15. &arr + &[j], r10
16. arr[j] (load), arr[r10], r13
17. arr[j] = arr[j+1] (store), arr[r10] = arr[r12]
18. j (load), r8
19. j + 1, r11
20. &[j+1]
21. &arr (load), r9
22. &arr + &[j+1], r10
23. tmp (load)
24. arr[j+1] = tmp (store), arr[r12] = tmp

We can perform the same type of optimizations as in the previous block but in a much larger scale. We can get rid of the second read of tmp but we cannot remove the store as we cannot know that it is not used elsewhere.

We can also move things around to make it look prettier than before.

```
    ;; 3, &arr (load)
    mov     r9, QWORD PTR [rbp-24]
        ;; 1, j (load)
    mov     r8, DWORD PTR [rbp-8]
    cdqe
        ;; 2, j + 1 (part 1)
    mov     r11, r8
    add     r11, 1
        ;; 2, &[j]
    lea     r10, [0+r8*4]
    ;; 9, &[j+1]
    lea     r12, [0+r11*4]
        ;; 4, &arr + &[j]
    add     r10, r9
    ;; 11 &arr + &[j+1]
    add     r12, r9
        ;; 5, arr[j] (load)
    mov     r13, DWORD PTR [r10]
        ;; 5, arr[j+1] (load)
        mov     r15, DWORD PTR [r12]
        ;; 6, tmp = arr[j] (store)
    mov     DWORD PTR [rbp-12], r13
        ;; 17, arr[j] = arr[j+1] (store)
    mov     DWORD PTR [r10], r13
        ;; 24, arr[j+1]
    mov     DWORD PTR [r12], r13
```

We now have three more trivial blocks. First, the block incrementing j, then the block decrementing i, and finally the return statement

```
;; Fourth trivial block, j++
.L4:
    add     DWORD PTR [rbp-8], 1
    jmp     .L5
;; Fifth trivial block, i--
.L3:
    sub     DWORD PTR [rbp-4], 1
    jmp     .L6
;; Sixth trivial block
.L7:
    nop
    ret
.LFE0:
```

A.3 Preparing for global optimization

We would like to make each block easier to compare with each other. We can do so if we try to make similar expressions look the same. We store j in r8, j+1 in r9, &arr in r10, &[j] and (&[j] + &arr) in r11, &[j+1] and (&[j+1] + &arr) in r12, arr[j] in r13, arr[j+1] in r14, and i in r15

```
        ;; Block 1
bubblesort(int*, int):
.LFB0:
    mov QWORD PTR [rbp-24], rdi
    mov DWORD PTR [rbp-28], esi
    mov r15, DWORD PTR [rbp-28]
    sub r15, 1
    mov DWORD PTR [rbp-4], r15
        ;; Block 2
.L6:
    cmp     DWORD PTR [rbp-4], 0
    jle     .L7
        ;; Block 3
        mov     DWORD PTR [rbp-8], 1
        ;; Block 4
.L5:
    mov     r15, DWORD PTR [rbp-4]
    cmp     r15, DWORD PTR [rbp-8]
    jl  .L3
        ;; Block 5
        mov     r10, QWORD PTR [rbp-24]
    mov     r8, DWORD PTR [rbp-8]
    cdqe
    lea     r11, [0+r8*4]
    add     r11, r10
    add     r8, 1
    lea     r12, [0+r8*4]
    add     r12, r10
    mov     r13, DWORD PTR [r11]
    mov     r14, DWORD PTR [r12]
    cmp     r13, r14
    jle     .L4
        ;; Block 6
        mov     r8, DWORD PTR [rbp-8]
        cdqe
    mov     r10, QWORD PTR [rbp-24]
    mov     r9, r8
    add     r9, 1
    lea     r11, [0+r8*4]
    lea     r12, [0+r9*4]
    add     r11, r10
    add     r12, r10
    mov     r13, DWORD PTR [r11]
    mov     r14, DWORD PTR [r12]
    mov     DWORD PTR [rbp-12], r13
    mov     DWORD PTR [r11], r14
    mov     DWORD PTR [r12], r13
        ;; Block 7
.L4:
    add     DWORD PTR [rbp-8], 1
    jmp     .L5
        ;; Block 8
.L3:
    sub     DWORD PTR [rbp-4], 1
    jmp     .L6
        ;; Block 9
.L7:
    nop
    ret
.LFE0:
```

A.4 Global optimization

We can now get into some global optimization. First, we will consider global common subexpression elimination. If an expression occurs in multiple blocks and isn't changed, we can get rid of redundant loads and stores.

1. We can move the load of arr out of the whole thing, it is never changed. In fact, we do not actually need to store it on the stack. We can let it remain in the register it was passed in through.
2. We can (with some care) within the inner loop store j, j+1, and the addresses computed with these in a single register for each. In fact, neither j nor j + 1 needs to be stored on the stack.
   - i, j, and j+1 can be stored in the same register through the entire procedure.
     - j and j + 1 are only used and never updated between block 4 and block 6. They are updated in block 7 and block 3 and there is no possible flow which could cause problems.
   - The addresses and their computations corresponding to j and j + 1 can be computed at block 5 as no path through block 5 or block 6 can cause troubles.

```
        ;; Block 1
bubblesort(int*, int):
.LFB0:
    ;; mov QWORD PTR [rbp-24], rdi
        ;; Keep arr in rdi
    mov r15, esi
    sub r15, 1
        ;; Block 2
.L6:
    cmp     r15, 0
    jle     .L7
        ;; Block 3
        mov     r8, 1
        ;; Block 4
.L5:
    cmp     r15, r8
    jl  .L3
        ;; Block 5
        mov     r9, r8
    add     r9, 1
        lea     r11, [0+r8*4]
    add     r11, rdi

    lea     r12, [0+r9*4]
    add     r12, rdi

    mov     r13, DWORD PTR [r11]
    mov     r14, DWORD PTR [r12]
    cmp     r13, r14
    jle     .L4
        ;; Block 6
    mov     DWORD PTR [rbp-12], r13
    mov     DWORD PTR [r11], r14
    mov     DWORD PTR [r12], r13
        ;; Block 7
.L4:
    add     r8, 1
    jmp     .L5
        ;; Block 8
.L3:
    sub     r15, 1
    jmp     .L6
        ;; Block 9
.L7:
    nop
    ret
.LFE0:
```

This is already looking much better. Now that we have code that is easier to wrap your head around, we can remove the redundant adds in our address calculations and perform them within the instruction. We can then remove the store of addresses into registers and keep the whole thing in the instructions themselves.

```
        ;; Block 1
bubblesort(int*, int):
.LFB0:
    ;; mov QWORD PTR [rbp-24], rdi
        ;; Keep arr in rdi
    mov r15, esi
    sub r15, 1
        ;; Block 2
.L6:
    cmp     r15, 0
    jle     .L1
        ;; Block 3
        mov     r8, 1
        ;; Block 4
.L5:
    cmp     r15, r8
    jl          .L3
        ;; Block 5
        mov     r9, r8
    add     r9, 1
    mov     r13, DWORD PTR      [rdi+r8*4]
    mov     r14, DWORD PTR [rdi+r9*4]
    cmp     r13, r14
    jle     .L4
        ;; Block 6
    mov     DWORD PTR [rbp-12], r13
    mov     DWORD PTR [rdi+r8*4], r14
    mov     DWORD PTR [rdi+r9*4], r13
        ;; Block 7
.L4:
    add     r8, 1
    jmp     .L5
        ;; Block 8
.L3:
    sub     r15, 1
    jmp     .L6
        ;; Block 9
.L7:
    nop
    ret
.LFE0:
```

Next optimization pass we can apply is induction variable elimination. There will never be a reason to have j + 1 stored separately from j and similar for the computed addresses.

```
        ;; Block 1
bubblesort(int*, int):
.LFB0:
    ;; mov QWORD PTR [rbp-24], rdi
        ;; Keep arr in rdi
    mov r15, esi
    sub r15, 1
        ;; Block 2
.L6:
    cmp     r15, 0
    jle     .L1
        ;; Block 3
        mov     r8, 1
        ;; Block 4
.L5:
    cmp     r15, r8
    jl          .L3
        ;; Block 5
    mov     r13, DWORD PTR      [rdi+r8*4]
    mov     r14, DWORD PTR [rdi+r8*4 + 4]
    cmp     r13, r14
    jle     .L4
        ;; Block 6
    mov     DWORD PTR [rbp-12], r13
    mov     DWORD PTR [rdi+r8*4], r14
    mov     DWORD PTR [rdi+r8*4 + 4], r13
        ;; Block 7
.L4:
    add     r8, 1
    jmp     .L5
        ;; Block 8
.L3:
    sub     r15, 1
    jmp     .L6
        ;; Block 9
.L7:
    nop
    ret
.LFE0:
```

Next up is dead code elimination. Now that we are looking at the entire code, we can get rid of that icky store from earlier that we could not remove when looking locally.

```
        ;; Block 1
bubblesort(int*, int):
.LFB0:
    ;; mov QWORD PTR [rbp-24], rdi
        ;; Keep arr in rdi
    mov r15, esi
    sub r15, 1
        ;; Block 2
.L6:
    cmp     r15, 0
    jle     .L1
        ;; Block 3
        mov     r8, 1
        ;; Block 4
.L5:
    cmp     r15, r8
    jl          .L3
        ;; Block 5
    mov     r13, DWORD PTR      [rdi+r8*4]
    mov     r14, DWORD PTR [rdi+r8*4 + 4]
    cmp     r13, r14
    jle     .L4
        ;; Block 6
    mov     DWORD PTR [rdi+r8*4], r14
    mov     DWORD PTR [rdi+r8*4 + 4], r13
        ;; Block 7
.L4:
    add     r8, 1
    jmp     .L5
        ;; Block 8
.L3:
    sub     r15, 1
    jmp     .L6
        ;; Block 9
.L7:
    nop
    ret
.LFE0:
```

And we are done. For this case, the most important optimizations were removing redundant address calculations and global common subexpression elimination. However, keep in mind that even though we have optimized this bubblesort code thoroughly, we cannot fix the overarching issue that the code author decided to use bubblesort.

B Understanding undefined behavior

Let us start out with a simple example which could happen in some analysis code. We have some class that represents a particle and can be constructed from a const char* (i.e. these -> “”). We have written a function which checks if a particle, p, is a lepton (we disregard neutrinos in this example to keep it brief). We do this by creating an array with all particles that are leptons and we then intend to loop through the array and check if there is a match.

However, we make a small mistake in the loop. Our array has six elements but we have accidentally let the loop go through seven elements. Reading out of bounds of an array is UB. Imagine that you are the compiler and you are faced with this code. You look at the loop condition and realize that if we ever enter the seventh (i=6) case, we will read out of bounds of the leptons-array. Since UB is not allowed to happen, this must mean that the person who wrote the code has written the code in such a way that we always find a match before the seventh case.

The only way to leave the loop is through the return-statement. You then reason that, since the author must have written code where we never trigger UB, the only way out of the function is through the return true;-statement. You can therefore remove the entire function and replace it with return true;

```
bool is_lepton(const Particle& p) {
    Particle leptons[] = {"e-", "e+", "m-", "m+", "t-", "t+"}; // Ignore neutrinos
    // accidental <= instead of <
    for (int i = 0; i <= 6 ;++i ) {
        if (p == leptons[i]) {
            return true;
        }
    }
    return false;
}
// Lots of code
int main (){
    if (is_lepton("pi-"))
        return 0;
    return 1;
}
```

When compiled with optimizations, Clang will return 1 as we would have expected. GCC however, will return 0.

This next example is adapted from a blog post from John Regehr (University of Utah) [17]. Compiled with Clang and optimizations turned on, this will output “Fermat has been disproven!” while it will keep on going when compiled with GCC.

As infinite loops are undefined in C++, the compiler observes an infinite loop with only one way out, “return true”. Therefore, since infinite loops are undefined, this means that at some point, our comparison must be true and the whole function can be reduced to “return true”. Note that the loop itself is undefined, so this would work the same way if we had infinite size integers.

```
#include <cmath>
#include <iostream>
bool is_fermat_disproven() {
    int a, b, c = 1;
    int n = 3;
    while (true) {
       if ((std::pow(a,n) + std::pow(b,n)) == std::pow(c,n)) {
          return true;
       }
       // do some interesting math here to update a,b,c,n
    }
    return false;
}
int main ( ) {
    if(is_fermat_disproven()) {
        std::cout << "Fermat has been disproven!\n";
    } else {
        std::cout << "Fermat has not been disproven!\n";
    }
    return 0;
}
```

C Environment details and software versions

C.1 Environment variables that need to be managed

The following environment variables need to be changed to match either the toolchain, when compiling the compilers, or the compiler that is to be used in a particular configuration.

`CC`, `CXX`, `AS`, `AR`, `NM`, `RANLIB`, `OBJCOPY`, `LD`, `CMAKE_AR`, `CMAKE_RANLIB`, `CMAKE_NM`, `CMAKE_C_COMPILER`, `CMAKE_CXX_COMPILER`, and `CMAKE_LINKER`, all need to point to their corresponding tool. Notably, the versions of AR, NM, RANLIB, need to point to the wrapper versions that support LTO, called gcc-ar, gcc-nm, and gcc-ranlib.

The following environment variables need to be limited to prevent components of the host system to be used instead of the local toolchain.

`ROOTSYS`, `PATH`, `LIBRARY_PATH`, `LD_LIBRARY_PATH`, `CMAKE_PREFIX_PATH`, `PERL5LIB`, `PERLLIB`, `PYTHONPATH`, and `CMAKE_MODULE_PATH`.

To save on disk space, we download the required data files for Geant4 into a single location rather than letting each configuration download its own version. If this is done, the following environment variables need to be set to the corresponding location.

`G4NEUTRONHPDATA`, `G4LEDATA`, `G4LEVELGAMMADATA`, `G4RADIOACTIVEDATA`, `G4ABLADATA`, `G4ENSDFSTATEDATA`, `G4NEUTRONXSDATA`, `G4PIIDATA`, `G4SAIDXSDATA`, and `G4REALSURFACEDATA`,

During compilation, the important environment variables are `CFLAGS`, `CXXFLAGS`, `CMAKE_C_FLAGS`, `CMAKE_CXX_FLAGS`, and `LDFLAGS`.

In order to get around CMake appending its own optimization flags, we set `CMAKE_CXX_FLAGS_RELEASE` and `CMAKE_C_FLAGS_RELEASE` manually to “”.

C.2 Software used in the setup

Table 11: Software compiled as part of each configurations. Note the source locations may change over time in which case using the version number will be the way to reproduce the system.

Package	Version	Source
HepExpMT	0.9.4	`https://cernbox.cern.ch/index.php/s/3MNrtycrirGalYa/download`
Geant4	10.02.p02	`https://cernbox.cern.ch/index.php/s/3MNrtycrirGalYa/download`
CLHEP	2.4.1.3	`https://proj-clhep.web.cern.ch/proj-clhep/dist1/clhep-2.4.1.3.tgz`
expat	2.2.9	`https://github.com/libexpat/libexpat/releases/download/R_2_2_9/expat-2.2.9.tar.gz`
xerces-c++	3.2.2	`http://www.apache.org/dist/xerces/c/3/sources/xerces-c-3.2.2.tar.gz`

Table 12: Core components of the local toolchain, version number, and source location. Includes the compiler which is used to build the compilers used in our measurements Note the source locations may change over time in which case using the version number will be the way to reproduce the system.

Package	Version	Source
Toolchain compiler	9.2.0	`ftp://ftp.gnu.org/gnu/gcc/gcc-9.2.0/gcc-9.2.0.tar.gz`
libtool	2.4	`ftp://ftp.gnu.org/gnu/libtool/libtool-2.4.tar.gz`
pkgconfig	0.29.2	`https://pkg-config.freedesktop.org/releases/pkg-config-0.29.2.tar.gz`
zlib	1.2.11	`https://www.zlib.net/zlib-1.2.11.tar.gz`
help2man	1.47.12	`https://ftp.gnu.org/gnu/help2man/help2man-1.47.12.tar.xz`
bzip2	1.0.8	`ftp://sourceware.org/pub/bzip2/bzip2-1.0.8.tar.gz`
xz	5.2.4	`https://tukaani.org/xz/xz-5.2.4.tar.gz`
readline	8.0	`ftp://ftp.gnu.org/gnu/readline/readline-8.0.tar.gz`
binutils	2.34	`ftp://ftp.gnu.org/gnu/binutils/binutils-2.34.tar.xz`
bash	5.0	`https://ftp.gnu.org/gnu/bash/bash-5.0.tar.gz`
m4	1.4.18	`https://ftp.gnu.org/gnu/m4/m4-1.4.18.tar.gz`
texinfo	6.7	`https://ftp.gnu.org/gnu/texinfo/texinfo-6.7.tar.gz`
automake	1.16	`https://ftp.gnu.org/gnu/automake/automake-1.16.tar.gz`
autoconf	2.69	`https://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz`
bison	3.5	`https://ftp.gnu.org/gnu/bison/bison-3.5.tar.gz`
gettext	0.20	`https://ftp.gnu.org/gnu/gettext/gettext-0.20.tar.gz`
flex	2.6.4	`https://github.com/westes/flex/files/981163/flex-2.6.4.tar.gz`
cmake	3.16.3	`https://github.com/Kitware/CMake/releases/download/v3.16.3/cmake-3.16.3.tar.gz`

Table 13: Utility packages used in the local toolchain, version number, and source location. Note the source locations may change over time in which case using the version number will be the way to reproduce the system.

Package	Version	Source
make	4.3	`https://ftp.gnu.org/gnu/make/make-4.3.tar.gz`
file	5.38	`ftp://ftp.astron.com/pub/file/file-5.38.tar.gz`
curl	7.68.0	`https://curl.haxx.se/download/curl-7.68.0.tar.gz`
util-linux	2.35	`https://mirrors.edge.kernel.org/pub/linux/utils/util-linux/v2.35/util-linux-2.35.tar.xz`
git	2.9.5	`https://mirrors.edge.kernel.org/pub/software/scm/git/git-2.9.5.tar.gz`
coreutils	8.31	`https://ftp.gnu.org/gnu/coreutils/coreutils-8.31.tar.xz`
gawk	5.0.1	`https://ftp.gnu.org/gnu/gawk/gawk-5.0.1.tar.gz`
grep	3.4	`https://ftp.gnu.org/gnu/grep/grep-3.4.tar.xz`
sed	4.8	`https://ftp.gnu.org/gnu/sed/sed-4.8.tar.gz`

Continued on next page

Table 13: Utility packages used in the local toolchain, version number, and source location. Note the source locations may change over time in which case using the version number will be the way to reproduce the system.

Package	Version	Source
gzip	1.10	`https://ftp.gnu.org/gnu/gzip/gzip-1.10.tar.gz`
zstd	1.4.4	`https://github.com/facebook/zstd/releases/download/v1.4.4/zstd-1.4.4.tar.gz`
tar	1.32	`ftp://ftp.gnu.org/gnu/tar/tar-1.32.tar.gz`
wget	1.9	`ftp://ftp.gnu.org/gnu/wget/wget-1.9.tar.gz`
findutils	4.7.0	`https://ftp.gnu.org/gnu/findutils/findutils-4.7.0.tar.xz`
diffutils	3.7	`https://ftp.gnu.org/gnu/diffutils/diffutils-3.7.tar.xz`
less	5.51	`http://www.greenwoodsoftware.com/less/less-551.tar.gz`
which	2.21	`ftp://ftp.gnu.org/gnu/which/which-2.21.tar.gz`
psmisc	23.3	`https://sourceforge.net/projects/psmisc/files/psmisc/psmisc-23.3.tar.xz/download`
elfutils	0.178	`https://sourceware.org/ftp/elfutils/0.178/elfutils-0.178.tar.bz2`

Table 14: Compilers and their common libraries used in the local for compiling our configurations, version number, and source location. For Clang, the package is downloaded from git directly using the tag noted in the version number column. Note the source locations may change over time in which case using the version number or tag will be the way to reproduce the system.

Package	Version	Source
mpc	1.1.0	`https://ftp.gnu.org/gnu/mpc/mpc-1.1.0.tar.gz`
mpfr	4.0.2	`https://ftp.gnu.org/gnu/mpfr/mpfr-4.0.2.tar.gz`
gmp	6.2.0	`https://ftp.gnu.org/gnu/gmp/gmp-6.2.0.tar.xz`
cloog	0.18.1	`https://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1.tar.gz`
gcc	4.9.4	`ftp://ftp.gnu.org/gnu/gcc/gcc-4.9.4/gcc-4.9.4.tar.gz`
gcc	6.2.0	`ftp://ftp.gnu.org/gnu/gcc/gcc-6.2.0/gcc-6.2.0.tar.gz`
gcc	8.3.0	`ftp://ftp.gnu.org/gnu/gcc/gcc-8.3.0/gcc-8.3.0.tar.xz`
clang	10.0.0	`https://github.com/llvm/llvm-project`
icc (PMPE)	19.0.5.281	N/A

Table 15: Programming languages, interpreters and libraries used in the local toolchain, version number, and source location. Note the source locations may change over time in which case using the version number will be the way to reproduce the system.

Package	Version	Source
perl	5.30.1	`https://www.cpan.org/src/5.0/perl-5.30.1.tar.gz`
XML-Parser	2.46	`https://cpan.metacpan.org/authors/id/T/TO/TODDR/XML-Parser-2.46.tar.gz`
python	3.8.1	`https://www.python.org/ftp/python/3.8.1/Python-3.8.1.tar.xz`
libffi	3.3	`https://github.com/libffi/libffi/releases/download/v3.3/libffi-3.3.tar.gz`
expat	2.2.9	`https://github.com/libexpat/libexpat/releases/download/R_2_2_9/expat-2.2.9.tar.gz`
z3	4.8.7	`https://github.com/Z3Prover/z3/archive/z3-4.8.7.tar.gz`
root	6.18.04	`https://root.cern/download/root_v6.18.04.source.tar.gz`
audit	2.8.5	`https://github.com/linux-audit/audit-userspace/archive/v2.8.5.tar.gz`
swig	4.0.1	`https://downloads.sourceforge.net/swig/swig-4.0.1.tar.gz`

D Supplementary data

Table 16: All measurements with ATLAS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
clang-SOfastNL	2.02(1)	17.3	2.07(1)	42.91(4)	0.19
gcc8-SOfastNL	2.07(1)	15.1	2.19(1)	42.92(4)	0.17
clang-SOfastN	2.09(1)	14.6	2.12(1)	42.87(5)	0.30
gcc8-SOfastN	2.10(1)	14.0	2.14(1)	42.93(4)	0.15
gcc4-SOfastN	2.10(1)	14.0	2.15(1)	42.98(4)	0.03
icc-SOfastN	2.12(1)	13.3	2.16(1)	42.93(4)	0.14
gcc4-SOfastNL	2.12(1)	13.1	2.16(1)	43.02(4)	0.05
gcc6-SOfastN	2.13(1)	12.9	2.13(1)	42.95(4)	0.11
clang-SOfastL	2.13(1)	12.8	2.21(1)	42.99(4)	0.00
clang-DOfastNL	2.13(1)	12.7	2.13(1)	43.07(4)	0.17
icc-SOfast	2.16(1)	11.8	2.20(1)	42.98(4)	0.02
gcc6-SOfastNL	2.17(1)	11.2	2.23(1)	42.96(4)	0.08
gcc8-SOfast	2.18(1)	10.8	2.21(1)	42.96(4)	0.09
gcc6-SOfast	2.18(1)	10.7	2.24(1)	42.94(4)	0.13
gcc8-SOfastL	2.19(1)	10.4	2.21(1)	43.05(4)	0.12
icc-SO2	2.19(1)	10.2	2.24(1)	42.97(4)	0.05
clang-SO2NL	2.20(1)	10.0	2.24(1)	43.04(4)	0.12
icc-SO3	2.20(1)	10.0	2.25(1)	42.97(4)	0.05
gcc6-SO3N	2.20(1)	9.9	2.22(1)	43.00(4)	0.01
gcc4-SO3N	2.21(1)	9.5	2.26(1)	43.07(4)	0.18
clang-DOfastL	2.21(1)	9.5	2.27(1)	42.87(4)	0.28
gcc4-SO2NL	2.22(1)	9.3	2.24(1)	42.93(4)	0.14
clang-SOfast	2.22(1)	9.3	2.24(1)	42.96(4)	0.09
icc-SOfastNL	2.22(1)	9.2	2.28(1)	42.94(4)	0.12
gcc8-SO2NL	2.22(1)	9.1	2.25(1)	43.01(4)	0.04
icc-SO2NL	2.22(1)	9.1	2.25(1)	42.91(4)	0.20
gcc4-SOfast	2.23(1)	8.8	2.24(1)	42.85(5)	0.34
gcc6-SOfastL	2.24(1)	8.4	2.26(1)	42.96(4)	0.08
gcc8-SO3N	2.24(1)	8.2	2.24(1)	42.97(4)	0.05
clang-SOsN	2.24(1)	8.2	2.28(1)	43.04(4)	0.12
clang-SO2L	2.24(1)	8.1	2.31(1)	43.04(4)	0.12
gcc6-SO2NL	2.25(1)	8.1	2.27(1)	42.96(4)	0.08
gcc4-SO2N	2.25(1)	8.0	2.30(1)	42.92(5)	0.18
gcc8-SO2N	2.25(1)	8.0	2.24(1)	43.04(4)	0.12
icc-SO2N	2.25(1)	8.0	2.32(1)	43.01(4)	0.04
gcc4-SO3	2.25(1)	7.9	2.29(1)	42.99(4)	0.00
clang-DOfastN	2.26(1)	7.7	2.24(1)	42.87(5)	0.30
gcc4-SOfastL	2.26(1)	7.5	2.23(1)	42.95(4)	0.10
gcc4-DOfastN	2.27(1)	7.1	2.27(1)	42.93(4)	0.15
gcc6-DOfastN	2.27(1)	7.0	2.28(1)	42.98(4)	0.04
gcc8-DOfastN	2.28(1)	6.8	2.31(1)	42.92(4)	0.18
clang-DO2NL	2.28(1)	6.8	2.32(1)	43.04(4)	0.12
gcc4-DOfastNL	2.28(1)	6.7	2.30(1)	42.92(4)	0.17
clang-DO3NL	2.28(1)	6.7	2.31(1)	43.04(4)	0.12
gcc8-SO3	2.28(1)	6.6	2.28(1)	42.99(4)	0.00

Continued on next page

Table 16: All measurements with ATLAS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-SO2	2.29(1)	6.4	2.35(1)	42.99(4)	0.00
gcc4-SO2L	2.29(1)	6.4	2.34(1)	42.99(4)	0.00
gcc8-DOfastNL	2.29(1)	6.3	2.27(1)	42.98(4)	0.03
gcc4-SO2	2.29(1)	6.1	2.34(1)	42.99(4)	0.00
gcc6-SO3	2.29(1)	6.1	2.31(1)	42.99(4)	0.00
gcc8-SO2L	2.30(1)	6.0	2.29(1)	42.99(4)	0.00
clang-SO3N	2.30(1)	5.9	2.34(1)	43.04(4)	0.12
clang-SOs	2.30(1)	5.8	2.35(1)	43.04(4)	0.12
gcc8-SO3NL	2.31(1)	5.6	2.37(1)	42.90(4)	0.21
clang-SO2N	2.31(1)	5.5	2.32(1)	43.04(4)	0.12
gcc6-SO2L	2.31(1)	5.4	2.33(1)	42.99(4)	0.00
icc-SO2L	2.31(1)	5.3	2.34(1)	42.97(4)	0.06
clang-DOfast	2.33(1)	4.8	2.38(1)	42.96(4)	0.09
gcc8-DOfast	2.33(1)	4.8	2.32(1)	42.88(4)	0.27
icc-SOfastL	2.33(1)	4.7	2.36(1)	42.94(4)	0.12
clang-SO3	2.33(1)	4.6	2.38(1)	43.04(4)	0.12
icc-SO3N	2.33(1)	4.6	2.34(1)	43.01(4)	0.04
icc-SO3NL	2.34(1)	4.4	2.31(1)	42.98(4)	0.03
clang-SO3NL	2.34(1)	4.3	2.28(1)	43.04(4)	0.12
gcc6-SO2N	2.35(1)	4.0	2.29(1)	42.90(4)	0.22
gcc6-DOfastNL	2.35(1)	3.9	2.34(1)	42.95(4)	0.10
clang-DO2L	2.35(1)	3.9	2.39(1)	43.04(4)	0.12
clang-SO3L	2.35(1)	3.8	2.39(1)	43.04(4)	0.12
gcc8-DO3N	2.35(1)	3.8	2.36(1)	42.94(4)	0.11
gcc4-DOfastL	2.35(1)	3.7	2.39(1)	42.93(4)	0.14
icc-DOfastNL	2.35(1)	3.7	2.35(1)	43.00(4)	0.01
gcc4-SO3NL	2.35(1)	3.7	2.33(1)	42.88(4)	0.27
gcc4-DOfast	2.35(1)	3.7	2.37(1)	42.98(4)	0.04
gcc6-SO2	2.36(1)	3.6	2.31(1)	42.99(4)	0.00
icc-SOsN	2.36(1)	3.6	2.39(1)	42.98(4)	0.03
gcc6-DO2NL	2.36(1)	3.6	2.41(1)	42.93(4)	0.15
gcc6-DO2N	2.36(1)	3.5	2.36(1)	42.98(5)	0.04
gcc4-DO2NL	2.36(1)	3.3	2.38(1)	42.92(4)	0.16
gcc8-DOfastL	2.36(1)	3.2	2.41(1)	43.03(4)	0.08
gcc8-SO3L	2.36(1)	3.2	2.42(1)	42.99(4)	0.00
gcc4-DO3NL	2.37(1)	3.2	2.35(1)	42.94(4)	0.13
clang-DO3L	2.37(1)	3.2	2.45(1)	43.04(4)	0.12
gcc6-DOfast	2.38(1)	2.8	2.35(1)	43.04(4)	0.11
gcc8-DO2N	2.38(1)	2.6	2.39(1)	42.84(4)	0.35
gcc4-SO3L	2.39(1)	2.4	2.43(1)	42.99(4)	0.00
gcc4-DO3N	2.39(1)	2.3	2.38(4)	42.94(4)	0.13
gcc8-DO2NL	2.39(1)	2.3	2.38(1)	42.99(4)	0.01
clang-DO3N	2.39(1)	2.1	2.44(1)	43.04(4)	0.12
icc-DOfast	2.39(1)	2.0	2.39(1)	43.01(4)	0.04
gcc6-DOfastL	2.39(1)	2.0	2.40(1)	42.92(4)	0.17
icc-DO3NL	2.40(1)	1.8	2.39(1)	43.00(4)	0.02
gcc6-DO3NL	2.40(1)	1.7	2.38(1)	42.93(4)	0.15

Continued on next page

Table 16: All measurements with ATLAS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
clang-DOsN	2.40(1)	1.7	2.39(1)	43.04(4)	0.12
gcc8-DO3NL	2.40(1)	1.6	2.35(1)	42.96(4)	0.08
gcc6-SO3NL	2.41(1)	1.4	2.43(1)	42.99(4)	0.00
icc-SO3L	2.41(1)	1.4	2.50(1)	42.97(4)	0.06
gcc4-DO2L	2.41(1)	1.3	2.45(1)	42.99(4)	0.00
gcc6-DO3N	2.41(1)	1.2	2.40(1)	42.99(4)	0.02
gcc8-DO3	2.42(1)	0.8	2.43(1)	42.99(4)	0.00
gcc6-DO2	2.43(1)	0.7	2.44(1)	42.99(4)	0.00
icc-DOfastL	2.43(1)	0.6	2.40(1)	42.95(5)	0.11
clang-DO2N	2.43(1)	0.6	2.48(1)	43.04(4)	0.12
clang-DOs	2.43(1)	0.5	2.46(1)	43.04(4)	0.12
gcc6-DO3	2.43(1)	0.4	2.44(1)	42.99(4)	0.00
clang-SO2	2.43(1)	0.4	2.43(1)	43.04(4)	0.12
gcc4-DO3	2.43(1)	0.4	2.43(1)	42.99(4)	0.00
icc-SOsNL	2.43(1)	0.4	2.40(1)	42.93(4)	0.15
gcc4-DO3L	2.44(1)	0.2	2.43(1)	42.99(4)	0.00
icc-DO2NL	2.44(1)	0.2	2.41(1)	42.99(4)	0.02
gcc4-DO2	2.44(1)	0.0	2.45(1)	42.99(4)	0.00
clang-DO3	2.45(1)	−0.3	2.49(1)	43.04(4)	0.12
icc-SOs	2.45(1)	−0.3	2.48(1)	42.97(4)	0.05
icc-SOsL	2.45(1)	−0.3	2.48(1)	42.93(4)	0.14
gcc8-DO3L	2.45(1)	−0.4	2.43(1)	42.99(4)	0.00
clang-DO2	2.47(1)	−0.9	2.50(1)	43.04(4)	0.12
gcc4-DO2N	2.47(1)	−1.0	2.35(1)	42.98(4)	0.04
gcc8-DO2	2.48(1)	−1.5	2.48(1)	42.99(4)	0.00
icc-DO2	2.49(1)	−1.9	2.46(1)	42.97(4)	0.05
icc-DO3L	2.49(1)	−1.9	2.49(1)	42.94(4)	0.13
gcc6-DO2L	2.49(1)	−2.0	2.44(1)	42.99(4)	0.00
gcc6-DO3L	2.50(1)	−2.5	2.46(1)	42.99(4)	0.00
icc-DOfastN	2.51(1)	−2.5	2.40(1)	43.00(4)	0.03
gcc6-SO3L	2.51(1)	−2.7	2.47(1)	42.99(4)	0.00
icc-DO2L	2.52(1)	−3.0	2.47(1)	42.94(4)	0.13
gcc8-SOsN	2.53(1)	−3.7	2.55(1)	42.98(4)	0.03
icc-DO3	2.54(1)	−3.8	2.42(1)	42.97(4)	0.05
gcc6-SOsN	2.54(1)	−3.9	2.58(1)	43.01(4)	0.04
gcc4-SOsN	2.54(1)	−4.1	2.62(1)	43.01(4)	0.04
icc-DO2N	2.56(1)	−4.9	2.50(1)	42.90(4)	0.21
gcc8-DO2L	2.60(1)	−6.4	2.44(1)	42.99(4)	0.00
icc-DO3N	2.62(1)	−7.3	2.66(1)	42.96(4)	0.09
gcc4-SOs	2.64(1)	−8.2	2.76(1)	43.03(4)	0.08
icc-DOsNL	2.64(1)	−8.3	2.69(1)	42.99(4)	0.00
gcc8-SOs	2.65(1)	−8.3	2.65(1)	43.03(4)	0.08
gcc6-SOs	2.65(1)	−8.4	2.66(1)	42.99(4)	0.00
gcc8-SOsNL	2.67(1)	−9.2	2.68(1)	42.91(4)	0.20
icc-DOsN	2.67(1)	−9.3	2.62(1)	42.96(4)	0.08
gcc4-SOsNL	2.70(1)	−10.6	2.70(1)	42.95(4)	0.10
gcc8-SOsL	2.73(1)	−11.5	2.73(1)	42.99(4)	0.00

Continued on next page

Table 16: All measurements with ATLAS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-DOsN	2.74(1)	−12.0	2.74(1)	42.96(4)	0.08
gcc6-SOsNL	2.76(1)	−12.9	2.70(1)	43.00(5)	0.01
gcc4-SOsL	2.77(1)	−13.5	2.73(1)	42.99(4)	0.00
gcc6-DOsN	2.79(1)	−14.3	2.77(1)	42.96(4)	0.08
gcc4-DOsN	2.79(1)	−14.4	2.82(1)	42.95(4)	0.11
icc-DOs	2.80(1)	−14.7	2.90(1)	42.97(4)	0.05
icc-DOsL	2.82(1)	−15.6	2.75(1)	42.97(4)	0.05
gcc4-DOs	2.83(1)	−16.0	2.83(1)	42.99(4)	0.00
gcc4-DOsNL	2.86(1)	−17.1	2.88(1)	43.01(4)	0.03
gcc8-DOsNL	2.86(1)	−17.1	2.88(1)	42.98(4)	0.04
gcc6-SOsL	2.86(1)	−17.2	2.78(1)	42.99(4)	0.00
gcc8-DOs	2.87(1)	−17.5	2.86(1)	42.99(4)	0.00
gcc4-DOsL	2.90(1)	−18.6	2.92(1)	42.99(4)	0.00
gcc6-DOs	2.90(1)	−18.8	2.86(1)	42.99(4)	0.00
gcc6-DOsL	2.91(1)	−19.1	2.92(1)	42.99(4)	0.00
gcc8-DOsL	2.92(1)	−19.4	2.86(1)	42.99(4)	0.00
gcc6-DOsNL	2.94(1)	−20.1	2.84(1)	42.96(4)	0.09

Table 17: All measurements with ATLAS geometry on Aurora sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
clang-SOfastNL	1.87(1)	16.1	1.85(1)	42.91(4)	0.19
gcc8-SOfastNL	1.89(1)	15.5	1.89(1)	42.92(4)	0.17
gcc6-SOfastN	1.93(1)	13.8	1.92(1)	42.97(4)	0.06
clang-SOfastL	1.93(1)	13.7	1.98(1)	42.99(4)	0.00
gcc4-SOfastN	1.93(1)	13.6	1.87(1)	42.86(5)	0.31
clang-SOfastN	1.93(1)	13.5	1.92(1)	42.87(5)	0.30
gcc8-SOfastN	1.94(1)	13.2	1.90(1)	42.93(4)	0.15
gcc4-SOfastNL	1.94(1)	13.2	1.92(1)	42.95(4)	0.09
gcc6-SO2N	1.94(1)	13.1	1.95(1)	42.90(4)	0.22
gcc6-SOfastNL	1.97(1)	12.0	1.93(1)	42.87(5)	0.28
gcc8-SOfastL	1.97(1)	11.7	1.93(1)	43.05(4)	0.12
clang-DOfastNL	1.98(1)	11.5	1.99(1)	43.07(4)	0.17
gcc6-SOfast	1.99(1)	11.0	2.20(1)	42.94(4)	0.13
gcc8-SOfast	1.99(1)	10.7	2.02(1)	42.96(4)	0.09
gcc8-SO3NL	2.00(1)	10.7	1.96(1)	42.90(4)	0.21
gcc8-SO3N	2.00(1)	10.4	2.06(1)	42.97(4)	0.05
gcc4-SO3NL	2.00(1)	10.4	1.99(1)	42.88(4)	0.27
gcc4-SO3N	2.01(1)	10.1	1.98(1)	43.07(4)	0.18
gcc6-SO2NL	2.01(1)	10.1	1.91(1)	42.96(4)	0.08
gcc4-SOfastL	2.01(1)	9.8	2.01(1)	42.91(4)	0.18
gcc8-SO2N	2.02(1)	9.6	2.05(1)	43.04(4)	0.12
gcc6-SO3NL	2.02(1)	9.5	2.05(1)	42.99(4)	0.00

Continued on next page

Table 17: All measurements with ATLAS geometry on Aurora sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc6-SO3N	2.02(1)	9.4	1.99(1)	43.00(4)	0.01
gcc4-SO2NL	2.03(1)	9.3	2.02(1)	42.93(4)	0.14
gcc6-SOfastL	2.03(1)	9.3	2.07(1)	42.96(4)	0.08
gcc8-SO2NL	2.04(1)	8.7	2.06(1)	43.01(4)	0.04
gcc4-SOfast	2.04(1)	8.5	2.07(1)	42.85(5)	0.34
clang-DOfastN	2.05(1)	8.4	2.06(1)	42.87(5)	0.30
gcc4-SO2	2.05(1)	8.2	2.12(1)	42.99(4)	0.00
gcc4-SO2N	2.05(1)	8.0	2.08(1)	42.92(5)	0.18
gcc6-DO2N	2.06(1)	7.8	1.98(1)	42.98(5)	0.04
gcc4-SO3	2.06(1)	7.6	2.07(1)	42.99(4)	0.00
clang-SOfast	2.07(1)	7.5	2.00(1)	42.96(4)	0.09
gcc4-SO3L	2.07(1)	7.3	2.18(1)	42.99(4)	0.00
clang-SO3NL	2.07(1)	7.2	2.03(1)	43.04(4)	0.12
gcc6-DO2NL	2.07(1)	7.1	2.02(1)	42.93(4)	0.15
gcc4-DOfastNL	2.08(1)	7.0	2.15(1)	42.91(4)	0.19
gcc8-DOfastN	2.08(1)	7.0	2.03(1)	42.92(4)	0.18
gcc6-SO3	2.08(1)	7.0	2.11(1)	42.99(4)	0.00
gcc6-DOfastN	2.08(1)	7.0	2.09(1)	42.89(5)	0.24
clang-SO2NL	2.08(1)	6.9	2.09(1)	43.04(4)	0.12
gcc8-SO3L	2.08(1)	6.9	2.06(1)	42.99(4)	0.00
gcc4-SO2L	2.08(1)	6.8	2.12(1)	42.99(4)	0.00
gcc8-SO2	2.09(1)	6.5	2.19(1)	42.99(4)	0.00
clang-SO2L	2.09(1)	6.4	2.17(1)	43.04(4)	0.12
clang-SOsN	2.10(1)	6.2	2.09(1)	43.04(4)	0.12
clang-SO2N	2.10(1)	6.1	2.16(1)	43.04(4)	0.12
gcc6-SO3L	2.11(1)	5.4	2.16(1)	42.99(4)	0.00
gcc8-DOfast	2.12(1)	5.2	2.16(1)	42.88(4)	0.27
gcc8-SO3	2.12(1)	4.9	2.09(1)	42.99(4)	0.00
clang-SO3L	2.12(1)	4.9	2.12(1)	43.04(4)	0.12
gcc8-SO2L	2.13(1)	4.7	2.11(1)	42.99(4)	0.00
gcc6-SO2	2.13(1)	4.6	2.15(1)	42.99(4)	0.00
clang-DO2NL	2.14(1)	4.3	2.15(1)	43.04(4)	0.12
gcc6-SO2L	2.14(1)	4.3	2.14(1)	42.99(4)	0.00
gcc8-DOfastNL	2.14(1)	4.3	2.14(1)	42.98(4)	0.03
clang-DO3NL	2.14(1)	4.3	2.17(1)	43.04(4)	0.12
gcc4-DOfastN	2.14(1)	4.1	2.12(1)	42.97(4)	0.05
clang-SO3	2.14(1)	4.1	2.18(1)	43.04(4)	0.12
gcc4-DO2NL	2.15(1)	3.9	2.23(1)	42.92(4)	0.16
clang-SOs	2.15(1)	3.8	2.20(1)	43.04(4)	0.12
gcc6-DOfast	2.15(1)	3.5	2.19(1)	43.04(4)	0.11
gcc4-DO3N	2.16(1)	3.4	2.19(3)	42.94(4)	0.13
gcc6-DOfastL	2.16(1)	3.3	2.21(1)	42.92(4)	0.17
gcc8-DO2N	2.16(1)	3.2	2.18(1)	42.84(4)	0.35
gcc8-DO3NL	2.16(1)	3.1	2.18(1)	42.96(4)	0.08
gcc4-DOfast	2.16(1)	3.1	2.19(1)	42.98(4)	0.04
gcc8-DO3N	2.17(1)	2.9	2.18(1)	42.94(4)	0.11
gcc8-DOfastL	2.17(1)	2.8	2.19(1)	43.03(4)	0.08

Continued on next page

Table 17: All measurements with ATLAS geometry on Aurora sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc4-DOfastL	2.18(1)	2.4	2.19(1)	42.93(4)	0.14
clang-DOfast	2.18(1)	2.4	2.23(1)	42.96(4)	0.09
gcc4-DO3NL	2.19(1)	2.1	2.16(1)	42.94(4)	0.13
clang-SO3N	2.19(1)	2.0	2.26(1)	43.04(4)	0.12
clang-DO2L	2.19(1)	1.8	2.22(1)	43.04(4)	0.12
clang-DOsN	2.19(1)	1.8	2.24(1)	43.04(4)	0.12
clang-DOfastL	2.21(1)	1.0	2.23(1)	42.87(4)	0.28
gcc4-DO3L	2.21(1)	0.9	2.24(1)	42.99(4)	0.00
gcc6-DO3N	2.22(1)	0.6	2.20(1)	42.99(4)	0.02
gcc6-DO3NL	2.22(1)	0.4	2.22(1)	42.93(4)	0.15
clang-DO2N	2.23(1)	0.1	2.27(1)	43.04(4)	0.12
gcc4-DO2N	2.23(1)	0.0	2.24(1)	42.98(4)	0.04
gcc8-DO3L	2.23(1)	0.0	2.22(1)	42.99(4)	0.00
gcc4-DO2	2.23(1)	0.0	2.22(1)	42.99(4)	0.00
gcc6-DO3	2.23(1)	0.0	2.23(1)	42.99(4)	0.00
clang-DO3L	2.24(1)	−0.2	2.25(1)	43.04(4)	0.12
gcc8-DO2	2.24(1)	−0.2	2.25(1)	42.99(4)	0.00
clang-DO3N	2.24(1)	−0.3	2.25(1)	43.04(4)	0.12
clang-SO2	2.24(1)	−0.5	2.21(1)	43.04(4)	0.12
gcc4-DO3	2.25(1)	−0.6	2.25(1)	42.99(4)	0.00
clang-DOs	2.26(1)	−1.1	2.29(1)	43.04(4)	0.12
gcc6-DO2L	2.27(1)	−1.6	2.26(1)	42.99(4)	0.00
clang-DO3	2.29(1)	−2.5	2.32(1)	43.04(4)	0.12
gcc6-DO3L	2.30(1)	−2.9	2.29(1)	42.99(4)	0.00
gcc4-DO2L	2.30(1)	−3.1	2.28(1)	42.99(4)	0.00
gcc6-SOsN	2.31(1)	−3.2	2.39(1)	43.01(4)	0.04
clang-DO2	2.31(1)	−3.6	2.29(1)	43.04(4)	0.12
gcc8-DO2L	2.32(1)	−3.9	2.21(1)	42.99(4)	0.00
gcc8-SOsN	2.32(1)	−3.9	2.34(1)	42.98(4)	0.03
gcc6-DOfastNL	2.38(1)	−6.4	2.39(1)	42.93(5)	0.15
gcc4-SOsN	2.39(1)	−6.8	2.35(1)	43.01(4)	0.04
gcc8-SOs	2.40(1)	−7.5	2.42(1)	43.03(4)	0.08
gcc6-DO2	2.41(1)	−7.8	2.27(1)	42.99(4)	0.00
gcc6-SOs	2.43(1)	−8.9	2.45(1)	42.99(4)	0.00
gcc8-DO2NL	2.43(1)	−9.0	2.44(1)	42.99(4)	0.01
gcc8-SOsNL	2.45(1)	−9.6	2.42(1)	42.91(4)	0.20
gcc4-SOs	2.45(1)	−9.8	2.53(1)	43.03(4)	0.08
gcc4-SOsNL	2.48(1)	−10.8	2.42(1)	42.95(4)	0.10
gcc8-DOsN	2.51(1)	−12.3	2.46(1)	42.96(4)	0.08
gcc4-SOsL	2.51(1)	−12.3	2.49(1)	42.99(4)	0.00
gcc8-SOsL	2.51(1)	−12.4	2.49(1)	42.99(4)	0.00
gcc8-DO3	2.53(1)	−13.3	2.48(1)	42.99(4)	0.00
gcc4-DOsN	2.54(1)	−13.6	2.47(1)	42.95(4)	0.11
gcc6-SOsL	2.54(1)	−13.9	2.55(1)	42.99(4)	0.00
gcc6-DOsN	2.55(1)	−14.3	2.53(1)	42.96(4)	0.08
gcc6-SOsNL	2.56(1)	−14.5	2.50(1)	43.00(5)	0.01
gcc8-DOs	2.57(1)	−15.2	2.54(1)	42.99(4)	0.00

Continued on next page

Table 17: All measurements with ATLAS geometry on Aurora sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-DOsNL	2.59(1)	−16.0	2.52(1)	42.98(4)	0.04
gcc4-DOs	2.60(1)	−16.6	2.56(1)	42.99(4)	0.00
gcc6-DOsNL	2.64(1)	−18.2	2.58(1)	42.96(4)	0.09
gcc8-DOsL	2.65(1)	−18.8	2.60(1)	42.99(4)	0.00
gcc6-DOsL	2.68(1)	−20.0	2.64(1)	42.99(4)	0.00
gcc6-DOs	2.70(1)	−21.0	2.57(1)	42.99(4)	0.00
gcc4-DOsL	2.76(1)	−23.6	2.58(1)	42.99(4)	0.00
gcc4-DOsNL	2.99(1)	−33.8	2.83(1)	43.01(4)	0.03

Table 18: All measurements with CMS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
icc-SOfastN	1.062(4)	18.9	1.077(4)	42.98(4)	0.02
icc-SOfast	1.095(4)	16.4	1.114(4)	42.95(4)	0.05
icc-SO2	1.115(4)	14.9	1.116(4)	42.98(4)	0.00
icc-SO2N	1.126(4)	14.1	1.138(4)	42.97(4)	0.00
gcc6-SOfastN	1.133(4)	13.5	1.168(4)	42.99(4)	0.04
gcc8-SOfastN	1.134(4)	13.4	1.141(4)	43.03(4)	0.12
icc-SO3	1.136(4)	13.3	1.150(4)	42.98(4)	0.00
gcc4-SOfastNL	1.137(4)	13.2	1.142(4)	42.96(4)	0.03
gcc8-SOfastNL	1.138(4)	13.2	1.161(4)	43.00(4)	0.05
gcc4-SOfastN	1.143(4)	12.7	1.135(4)	42.96(4)	0.04
icc-SO3N	1.144(4)	12.7	1.143(4)	42.97(4)	0.00
gcc6-SOfastNL	1.163(4)	11.3	1.177(4)	42.99(4)	0.03
gcc4-SOfast	1.163(4)	11.2	1.178(4)	42.89(4)	0.21
gcc8-SOfastL	1.167(4)	10.9	1.188(4)	42.99(4)	0.03
gcc6-SOfast	1.169(4)	10.8	1.207(4)	42.93(4)	0.10
gcc8-SOfast	1.173(4)	10.5	1.187(4)	42.89(4)	0.19
gcc4-SO3N	1.173(4)	10.5	1.183(4)	42.92(4)	0.12
icc-DOfastN	1.173(4)	10.4	1.191(4)	42.96(4)	0.05
gcc6-SO3N	1.177(4)	10.2	1.173(4)	42.93(4)	0.10
gcc8-SO2NL	1.178(4)	10.1	1.186(4)	42.92(4)	0.13
clang-SO2NL	1.183(4)	9.7	1.152(4)	42.96(4)	0.05
clang-SO3NL	1.184(4)	9.6	1.160(4)	42.96(4)	0.05
gcc8-SO3N	1.185(4)	9.6	1.184(4)	42.93(4)	0.11
clang-SO3L	1.186(4)	9.5	1.198(4)	42.96(4)	0.05
clang-SO2L	1.190(4)	9.2	1.172(4)	42.96(4)	0.05
gcc8-SO2N	1.192(4)	9.0	1.183(5)	42.91(4)	0.15
gcc6-SOfastL	1.192(4)	9.0	1.200(4)	42.97(4)	0.02
gcc4-SO2NL	1.193(4)	9.0	1.195(4)	42.96(4)	0.04
icc-SOsN	1.195(5)	8.8	1.199(5)	42.93(4)	0.11
clang-SOfastNL	1.199(5)	8.5	1.192(5)	42.92(4)	0.13
clang-SOs	1.199(4)	8.5	1.216(4)	42.96(4)	0.05

Continued on next page

Table 18: All measurements with CMS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc8-SO2	1.200(4)	8.4	1.241(4)	42.98(4)	0.00
gcc6-SO2NL	1.200(4)	8.4	1.197(4)	42.98(4)	0.00
gcc4-SO2N	1.201(4)	8.4	1.214(4)	42.94(4)	0.09
gcc6-SO3	1.201(4)	8.3	1.216(4)	42.98(4)	0.00
gcc4-SO3	1.202(4)	8.2	1.208(4)	42.98(4)	0.00
clang-SOsN	1.202(4)	8.2	1.183(4)	42.96(4)	0.05
clang-SO3	1.203(4)	8.2	1.219(4)	42.96(4)	0.05
gcc4-SOfastL	1.206(4)	8.0	1.177(4)	43.03(4)	0.12
clang-SO3N	1.208(4)	7.8	1.184(4)	42.96(4)	0.05
gcc8-SO2L	1.212(4)	7.5	1.207(4)	42.98(4)	0.00
gcc4-SO2L	1.213(4)	7.4	1.241(4)	42.98(4)	0.00
clang-DO3NL	1.216(4)	7.2	1.198(4)	42.96(4)	0.05
gcc8-SO3	1.216(4)	7.2	1.213(4)	42.98(4)	0.00
clang-SO2N	1.221(4)	6.8	1.181(4)	42.96(4)	0.05
gcc8-DOfastNL	1.221(4)	6.8	1.243(4)	42.97(4)	0.00
gcc4-SO2	1.221(4)	6.8	1.233(4)	42.98(4)	0.00
clang-DO2NL	1.227(4)	6.4	1.219(4)	42.96(4)	0.05
gcc6-SO2L	1.230(4)	6.1	1.233(4)	42.98(4)	0.00
gcc8-DOfastN	1.231(4)	6.1	1.259(4)	42.86(4)	0.27
clang-DO3L	1.232(4)	6.0	1.231(4)	42.96(4)	0.05
clang-SOfastN	1.233(5)	5.9	1.226(5)	42.95(4)	0.05
gcc4-DOfastNL	1.234(4)	5.8	1.265(4)	43.00(4)	0.07
gcc8-SO3NL	1.238(4)	5.5	1.301(4)	42.91(4)	0.15
icc-SOs	1.239(5)	5.4	1.244(5)	42.98(4)	0.00
clang-SO2	1.241(4)	5.3	1.238(4)	42.96(4)	0.05
gcc4-DOfastN	1.243(4)	5.1	1.234(4)	42.95(4)	0.05
gcc4-SO3NL	1.245(4)	5.0	1.237(4)	43.00(4)	0.06
icc-DO2N	1.247(4)	4.8	1.247(4)	42.98(4)	0.00
gcc6-SO2	1.249(4)	4.7	1.222(4)	42.98(4)	0.00
clang-SOfastL	1.252(5)	4.5	1.264(6)	42.95(4)	0.06
gcc6-DOfastN	1.253(4)	4.4	1.259(4)	42.91(4)	0.15
gcc8-DOfast	1.253(4)	4.4	1.253(4)	42.96(4)	0.04
gcc6-SO2N	1.254(4)	4.3	1.200(4)	42.94(4)	0.07
clang-DO2L	1.259(4)	3.9	1.224(4)	42.96(4)	0.05
gcc4-DOfast	1.259(4)	3.9	1.267(4)	42.95(4)	0.07
icc-DO2	1.261(5)	3.8	1.214(4)	42.98(4)	0.00
gcc8-DO3N	1.263(4)	3.6	1.260(4)	42.98(4)	0.02
gcc8-DO2N	1.264(4)	3.5	1.253(4)	43.07(4)	0.22
clang-DOsN	1.264(4)	3.5	1.263(4)	42.96(4)	0.05
gcc6-DO2N	1.267(4)	3.3	1.302(4)	42.94(4)	0.09
clang-DO3N	1.268(4)	3.2	1.266(4)	42.96(4)	0.05
clang-DO3	1.270(4)	3.0	1.284(4)	42.96(4)	0.05
gcc4-SO3L	1.271(4)	3.0	1.281(4)	42.98(4)	0.00
gcc4-DO2N	1.271(4)	3.0	1.254(4)	42.98(4)	0.01
gcc8-DO3NL	1.272(4)	2.9	1.266(4)	42.96(4)	0.04
clang-DOfastNL	1.272(5)	2.9	1.250(5)	43.00(4)	0.05
icc-DOfast	1.272(5)	2.9	1.244(4)	43.03(4)	0.14

Continued on next page

Table 18: All measurements with CMS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc4-DO3NL	1.272(4)	2.9	1.278(4)	42.97(4)	0.00
gcc4-DOfastL	1.272(5)	2.9	1.286(4)	43.01(4)	0.08
gcc4-DO3N	1.273(4)	2.8	1.253(4)	42.97(4)	0.01
gcc8-DOfastL	1.273(4)	2.8	1.288(4)	42.94(4)	0.09
clang-DOs	1.274(4)	2.7	1.287(4)	42.96(4)	0.05
clang-DO2	1.276(4)	2.6	1.297(5)	42.96(4)	0.05
clang-DO2N	1.276(5)	2.6	1.266(4)	42.96(4)	0.05
clang-SOfast	1.278(6)	2.5	1.290(6)	42.97(4)	0.01
gcc6-SO3NL	1.278(4)	2.5	1.319(4)	42.96(4)	0.04
gcc6-DOfast	1.279(4)	2.4	1.266(4)	42.90(4)	0.18
gcc4-DO2NL	1.282(4)	2.2	1.261(4)	43.03(4)	0.13
gcc6-DOfastNL	1.284(4)	2.0	1.261(4)	42.95(4)	0.05
gcc8-DO2NL	1.284(4)	2.0	1.268(4)	42.98(4)	0.02
gcc6-DO3N	1.284(4)	2.0	1.317(4)	42.99(4)	0.02
gcc6-DO2NL	1.285(5)	1.9	1.287(4)	42.93(4)	0.10
gcc8-SO3L	1.291(4)	1.4	1.295(4)	42.98(4)	0.00
gcc8-DO3	1.293(4)	1.3	1.304(5)	42.98(4)	0.00
gcc4-DO3	1.295(4)	1.2	1.302(5)	42.98(4)	0.00
gcc8-DO3L	1.296(4)	1.1	1.290(5)	42.98(4)	0.00
gcc4-DO2L	1.296(4)	1.1	1.307(5)	42.98(4)	0.00
gcc6-DO2	1.301(4)	0.7	1.326(5)	42.98(4)	0.00
gcc4-DO3L	1.301(4)	0.7	1.307(5)	42.98(4)	0.00
icc-DO3	1.306(5)	0.3	1.286(5)	42.98(4)	0.00
gcc8-SOsN	1.308(5)	0.2	1.343(5)	42.94(4)	0.08
clang-DOfastN	1.309(6)	0.1	1.298(5)	42.95(4)	0.05
gcc4-DO2	1.310(4)	0.0	1.308(5)	42.98(4)	0.00
gcc6-DOfastL	1.310(4)	0.0	1.290(4)	42.92(4)	0.14
gcc6-DO3	1.311(4)	0.0	1.313(5)	42.98(4)	0.00
gcc8-DO2L	1.313(5)	−0.2	1.336(5)	42.98(4)	0.00
icc-SOsNL	1.313(5)	−0.2	1.282(5)	42.96(4)	0.05
gcc8-DO2	1.314(4)	−0.3	1.300(5)	42.98(4)	0.00
gcc6-DO3NL	1.316(40)	−0.4	1.283(4)	42.99(4)	0.03
gcc6-DO2L	1.316(5)	−0.4	1.306(5)	42.98(4)	0.00
gcc6-DO3L	1.323(5)	−1.0	1.325(5)	42.98(4)	0.00
clang-DOfastL	1.323(6)	−1.0	1.332(6)	43.04(4)	0.14
gcc6-SOsN	1.334(5)	−1.8	1.338(5)	43.05(4)	0.16
gcc4-SOsN	1.351(5)	−3.1	1.341(5)	43.04(4)	0.15
icc-SOsL	1.356(48)	−3.5	1.391(49)	42.96(4)	0.04
clang-DOfast	1.357(6)	−3.6	1.367(6)	42.97(4)	0.01
gcc6-SO3L	1.357(4)	−3.6	1.323(5)	42.98(4)	0.00
icc-DOsNL	1.366(5)	−4.3	1.353(5)	42.89(4)	0.19
gcc8-SOs	1.373(5)	−4.8	1.382(5)	42.98(4)	0.00
gcc4-SOs	1.376(5)	−5.0	1.419(5)	42.98(4)	0.00
icc-DO3N	1.378(5)	−5.1	1.347(5)	42.92(4)	0.13
icc-DOsN	1.378(5)	−5.2	1.412(5)	42.94(4)	0.08
gcc8-SOsNL	1.387(5)	−5.9	1.415(6)	42.95(4)	0.06
gcc6-SOs	1.389(5)	−6.0	1.386(5)	42.98(4)	0.00

Continued on next page

Table 18: All measurements with CMS geometry on PMPE sorted by Δt_b

Configuration	μ_{t_b} (s)	Δt_b (%)	μ_{t_v} (s)	μ_E (GeV)	ΔE (%)
gcc4-SOsNL	1.407(5)	−7.4	1.403(5)	42.87(4)	0.25
gcc6-SOsNL	1.407(5)	−7.4	1.412(5)	42.92(4)	0.13
icc-DOs	1.423(5)	−8.6	1.421(5)	42.98(4)	0.00
gcc8-SOsL	1.428(5)	−9.0	1.439(5)	42.98(4)	0.00
gcc4-SOsL	1.431(5)	−9.2	1.427(5)	42.98(4)	0.00
icc-DOsL	1.439(5)	−9.8	1.410(5)	42.98(4)	0.00
gcc6-DOsN	1.477(5)	−12.7	1.493(5)	42.89(4)	0.19
gcc8-DOsN	1.479(5)	−12.9	1.485(5)	42.92(4)	0.13
gcc6-SOsL	1.485(5)	−13.3	1.426(5)	42.98(4)	0.00
gcc4-DOsNL	1.500(5)	−14.5	1.516(5)	43.00(4)	0.06
gcc8-DOsNL	1.505(5)	−14.9	1.487(5)	43.03(4)	0.12
gcc8-DOs	1.515(5)	−15.7	1.519(5)	42.98(4)	0.00
gcc4-DOsN	1.520(5)	−16.0	1.488(5)	42.93(4)	0.11
gcc4-DOs	1.521(5)	−16.1	1.486(5)	42.98(4)	0.00
gcc6-DOsL	1.522(5)	−16.2	1.585(6)	42.98(4)	0.00
gcc6-DOs	1.529(5)	−16.7	1.531(5)	42.98(4)	0.00
gcc8-DOsL	1.534(5)	−17.1	1.512(5)	42.98(4)	0.00
gcc6-DOsNL	1.536(5)	−17.2	1.497(5)	42.99(4)	0.03
gcc4-DOsL	1.585(5)	−20.9	1.557(5)	42.98(4)	0.00
icc-SOfastNL	1.843(12)	−40.7	1.840(13)	42.92(4)	0.14
icc-SO2NL	1.857(12)	−41.7	1.871(13)	42.98(4)	0.00
icc-SO2L	1.899(13)	−44.9	1.890(13)	43.00(4)	0.05
icc-SO3NL	1.906(13)	−45.5	1.888(13)	42.99(4)	0.03
icc-SOfastL	1.926(13)	−47.0	1.899(13)	42.89(4)	0.19
icc-SO3L	1.933(13)	−47.5	1.951(13)	43.00(4)	0.05
icc-DO2NL	1.987(14)	−51.6	2.016(14)	43.03(4)	0.12
icc-DOfastNL	2.000(14)	−52.6	1.985(13)	43.02(4)	0.10
icc-DO3NL	2.028(14)	−54.8	2.001(14)	43.03(4)	0.12
icc-DOfastL	2.028(13)	−54.8	2.006(13)	42.96(4)	0.05
icc-DO3L	2.047(14)	−56.2	2.050(14)	43.01(4)	0.08
icc-DO2L	2.069(14)	−57.9	2.020(14)	43.01(4)	0.08