UNIVERSITY
OF OSLO

Master's Thesis

Performance Optimization in ACTS

Optimization of the computing performance of the inner detector track seeding algorithm used by ATLAS for High Luminosity LHC data

Thomas Aksdal

Computational Science: Physics
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences



Spring 2026

Thomas Aksdal

Performance Optimization in ACTS

Optimization of the computing performance of the inner detector track seeding algorithm used by ATLAS for High Luminosity LHC data

Supervisors:
Prof. Farid Ould-Saada
Dr. James Catmore

Abstract

The High-Luminosity upgrade of the Large Hadron Collider (HL-LHC) will increase the instantaneous luminosity and associated pile-up to levels that impose unprecedented performance requirements on all stages of the event reconstruction chain. Track seeding, which constructs the initial trajectory candidates supplied to subsequent track-finding algorithms, has received comparatively limited, systematic optimisation within the ACTS framework, despite constituting a significant fraction of the total reconstruction time.

In this thesis, a two-phase profiling strategy is employed to identify and mitigate performance bottlenecks in the ACTS seeding algorithm. The first phase consists of lightweight sampling-based profiling for hotspot identification, followed in the second phase by source-level instrumentation and annotation to enable detailed root-cause analysis. Based on these studies, five targeted optimisations are proposed and evaluated using wall-clock timing, Callgrind instruction-count measurements, Heaptrack-based memory profiling, and physics-performance validation on a simulated dataset at HL-LHC pile-up conditions.

Of the five proposed modifications, three satisfy both the runtime improvement and physics-preservation criteria. These are combined into a new seeding implementation, `Seeding3`, which achieves a wall-clock speedup of 11.6× over the `Seeding2` baseline while preserving equivalent physics performance. This corresponds to an increase in single-thread throughput from 1.7 to 19.3 events per second. These results demonstrate that the seeding stage possesses substantial headroom for performance optimisation when guided by systematic profiling, and that the adopted methodology is readily generalisable to other components of the reconstruction chain.

Preface

This thesis is submitted as partial fulfilment of the requirements for a Master's degree in Computational Science: Physics at the Department of Physics, University of Oslo. The work has been carried out in the High Energy Particle Physics group, under the supervision of Farid Ould-Saada, James Catmore, and Eirik Gramstad.

I would like to thank my supervisors for their guidance, patience, and expertise throughout the project. Our Monday meetings, on the occasions I didn't cancel them, kept the work on track and were always genuinely useful.

I will miss this place. The student area at the Department of Chemistry has been my life for the past five years. There have been many evenings spent frantically working on the last details before a deadline, but they have been some of the best moments, shared together with the best people. I am going to miss sitting in the office, waiting for someone to open that door and startle me for the hundredth time. It is not the thing I will miss the most, but it is something I will miss nonetheless.

The photographs below are a record of some of the student association boards I have been a part of during my time here at the university. The time I spent as a member of these associations made these years what they were, and was sometimes the only reason I was able to get out of bed in the morning. Menageriet, KFU, and Proton represent perhaps the time I am most proud to have spent here. They have given me a great deal of frustration, sleepless nights, and more responsibility than I bargained for, but I would give a lot to get the chance to do it all again.
A special thanks to some of the people I will always remember from my time here:

Selma, Live. Ivar, Iris. Ayla, Tirill. Sanam, Richard; Marie, Fredrik. Sindre, Viktor. Mariann, Per Sivert. Espen, Marte. Lea, Alexandra, Kristian, Marko; Trygve, Mathilde. Ava, Torstein. Lille Thomas, Tuva, Matthew, Line; Martine Cornelia, Linda. Ella, Kim. Yani, Tina. Kevin, Melika, and so many more.

Line Association Members



From left to right, top to bottom: Proton V22, Menageriet H22, Menageriet H23, Menageriet V24, Menageriet H24, KFU H25, Menageriet H25, Menageriet V26.

Contents

1 Introduction 1
1.1 Track Reconstruction Challenges at High Luminosity 1
1.2 Seeding in Track Reconstruction 3
1.3 Research Questions 5
1.4 Thesis Structure 6
2 Background 7
2.1 HL-LHC 7
2.2 ATLAS Inner Tracker 9
2.3 Track Reconstruction 9
2.3.1 The Reconstruction Pipeline 10
2.3.2 Track Parameterization 10
2.3.3 Computational Bottlenecks 11
2.4 ACTS Framework 12
2.4.1 The ACTS Seeding Algorithm 13
2.5 Performance Metrics 15
2.5.1 Computational Performance Metrics 15
2.5.2 Physics Performance Metrics 15
3 Introduction to Profiling Tools 17
3.1 Types of Profiling 17
3.1.1 Statistical Sampling Profilers 17
3.1.2 Instrumentation-Based Profilers 18
3.2 Core Profiling Tools 19
3.3 Limitations of Profiling Tools. 25
4 Datasets. 27
4.1 Profiling Dataset 27
4.1.1 Event Characteristics 27
4.2 Physics Validation Dataset 31
4.2.1 Event Configuration 31
4.3 Strip Spacepoint Processing 32
5 Methods 33
5.1 Experimental Environment 33
5.1.1 Computing Infrastructure. 33
5.1.2 Software Environment. 34
5.1.3 Thesis Code Repository and Reproducibility 36
5.2 Profiling Methodology 37
5.2.1 Experimental Workflow 37
5.2.2 Statistical Analysis of Performance Measurements 40

5.3 Physics Validation Methodology . . . 44
5.3.1 Optimisation Categories and Validation Strategy . . . 44
5.3.2 Evaluation Metrics . . . 44
5.3.3 Code-Level Optimisation Validation . . . 45
5.3.4 Algorithm-Level Performance Evaluation . . . 45
5.3.5 Interpretation . . . 45
5.3.6 Parameter Selection Methodology . . . 45
6 Results . . . 49
6.1 Baseline Performance Comparison: `Seeding` vs. `Seeding2` . . . 49
6.2 Detailed Profiling of Seeding2 . . . 59
6.2.1 Phase 1: Hotspot Identification . . . 59
6.2.2 Phase 2: Source-level Investigation . . . 64
6.2.3 Summary of Optimisation Targets . . . 67
6.3 Code-Level Optimisations . . . 68
6.3.1 O1 — `createDoubletsImpl` Inner-Loop Reformulation . . . 68
6.3.2 O2 — Early-Exit in `createPixelTripletTopCandidates` . . . 70
6.3.3 O3 — Removing Sort-Path Indirection for Inlining . . . 73
6.3.4 O4 — Upfront Vector Reservation . . . 75
6.3.5 Cumulative Impact . . . 79
6.4 Algorithm-Level Optimisations . . . 81
6.5 The `Seeding3` Implementation . . . 90
7 Conclusion . . . 97
7.1 Summary of Findings . . . 97
7.2 Limitations . . . 100
7.3 Future Work . . . 100
7.4 Closing Remarks . . . 101
A Build Instructions . . . 103
B Supplementary Figures and Data . . . 107
B.1 Baseline Comparison: `Seeding` vs. `Seeding2` . . . 107
B.2 Detailed `Seeding2` Profiling . . . 111
B.3 Algorithm-Level Optimisations . . . 129
B.4 `Seeding3` Proposal . . . 132

List of Figures

1.1 Reconstruction wall time per event as a function of pile-up $\langle\mu\rangle$ 2
1.2 Track reconstruction chain from space points to reconstructed tracks . . . 4

2.1 Simulated HL-LHC event in the ATLAS ITk geometry at $\langle\mu\rangle = 140$. 8
2.2 The five global track parameters at the perigee. 11
2.3 Triplet search neighborhood in the (r, z) plane. 13

3.1 Example `pprof` text output. 20
3.2 Example `pprof` call graph. 20
3.3 Example flame graph visualisation 21
3.4 Example line-level profiling output. 21
3.5 Example Heaptrack flame graph. 24

4.1 Per-event multiplicity distributions for the profiling dataset. 29
4.2 Reconstructed-track kinematic distributions for the profiling dataset. 30

5.1 Two-phase profiling workflow. 38
5.2 Runtime residuals as a function of execution order. 38

6.1 Waterfall decomposition of per-event execution time for `Seeding` and `Seeding2`. 53
6.2 CPU flamegraphs, `Seeding` vs. `Seeding2` on pixel data. 57
6.3 Per-event execution time vs. N_{SP} for `Seeding` and `Seeding2`. 58
6.4 CPU flamegraph for `Seeding2` on pixel data. 63
6.5 Seeding efficiency and reconstructed tracks vs. η for cylindrical and spherical configurations. 83
6.6 Fake and duplicated track counts vs. η for cylindrical and spherical configurations. 84
6.7 Stage-level waterfall decomposition: Seeding3 vs. Seeding2. 93
6.8 Per-event execution time vs. N_{SP} for Seeding2 and Seeding3. 94

B.1 CPU flamegraph for `Seeding` on strip data 108
B.2 Callgraph for `Seeding` on pixel data 109
B.3 Callgraph for `Seeding` on strip data 110
B.4 CPU flamegraph for `Seeding2` on strip data 112
B.5 Callgraph for `Seeding2` on pixel data 113
B.6 Callgraph for `Seeding2` on strip data. 114
B.7 Seeding performance vs. p_T 129
B.8 Seeding performance vs. ϕ 129
B.9 Spherical scan: η-bin width 130

B.10 Spherical scan: bottom-layer η-bin neighbour radius 130
B.11 Spherical scan: top-layer η-bin neighbour radius 130
B.12 Spherical scan: ϕ-bin deflection coverage 130
B.13 Spherical scan: maximum radial separation 131
B.14 Spherical scan: maximum impact parameter 131
B.15 Spherical scan: maximum seeds per middle space point 131
B.16 Spherical scan: compatible-seed limit 131
B.17 CPU flamegraph for `Seeding2` 133
B.18 CPU flamegraph for `Seeding3` 134
B.19 Callgraph for `Seeding2` 135
B.20 Callgraph for `Seeding3` 136

List of Tables

3.1 Comparison of profiling methodologies. 18
3.2 Selected Callgrind metrics. 23
3.3 Selected Heaptrack metrics 24

4.1 Event-averaged occupancy and track multiplicity for the profiling dataset. . . 28

5.1 Hardware and system specifications of HEPP02. 34
5.2 Profiling and analysis tools used in this work. 35

6.1 Overall `Seeding` vs. `Seeding2` performance. 50
6.2 Workload comparison between `Seeding` and `Seeding2`. 51
6.3 Top 10 functions, `Seeding` vs. `Seeding2` on pixel data. 54
6.4 Top 10 functions, `Seeding` vs. `Seeding2` on strip data. 55
6.5 Linear fit coefficients for per-event execution time vs. N_{SP}. 58
6.6 Stage-level timing breakdown for `Seeding2`. 60
6.7 Top `Seeding2` functions on pixel data. 61
6.8 Top `Seeding2` functions on strip data. 62
6.9 Summary of identified hotspots and their Phase 2 observations. 67
6.10 Wall-clock and Callgrind measurements for O1. 69
6.11 Physics validation for O1. 70
6.12 Wall-clock and Callgrind measurements for $O2$. 72
6.13 Physics validation for O2. 72
6.14 Wall-clock and Callgrind measurements for O3. 74
6.15 Physics validation for O3. 75
6.16 Heaptrack summary for O4. 77
6.17 Wall-clock and Callgrind measurements for O4. 78
6.18 Physics validation for O4. 78
6.19 Cumulative O1+O2 wall-clock and Callgrind measurements. 80
6.20 Physics validation for cumulative O1+O2. 81
6.21 Spherical specific parameters compared to cylindrical baseline. 83
6.22 Grid statistics for cylindrical baseline and spherical configurations. 85
6.23 Callgrind performance counters for cylindrical and spherical configurations. . 86
6.24 Callgrind per-function breakdown for cylindrical and spherical configurations. 87
6.25 Heaptrack peak memory and allocations for cylindrical and spherical configurations. 89
6.26 Overall wall-clock comparison: `Seeding2` vs. `Seeding3`. 91
6.27 Workload comparison: Seeding2 vs. Seeding3. 92
6.28 Scaling fit parameters for Seeding2 and Seeding3. 94

B.1 Top-10 functions by flat CPU time, `Seeding2` vs. `Seeding3` 132

Declaration of Use of Generative Artificial Intelligence

Generative artificial intelligence (AI) tools, including large language models (LLMs) such as ChatGPT, Claude, and DeepSeek, were used in the preparation of this thesis. Their use was limited to supportive and non-decisive tasks. The following outlines how these tools were applied:

Planning: AI models were consulted to provide suggestions on the structure and organization of chapters and sections. The author reviewed and adapted these suggestions when developing the final outline.

Drafting: AI models were occasionally used to generate example text or rough drafts for inspiration. All such material was thoroughly rewritten and verified by the author.

Text improvement: Selected sentences or paragraphs were input to the models to obtain feedback on clarity, conciseness, and depth. Any proposed edits were evaluated, and only changes approved by the author were incorporated.

Coding assistance: In programming-related parts of the thesis, AI tools were used to generate code examples or for specific commands. All generated code was tested, modified, and validated by the author.

Source searching: Large language models were also used as a supplementary aid for locating potential references and relevant literature. When seeking background material or related work, the models were prompted to suggest keywords, topics, or representative papers within the research area. These suggestions served only as starting points for manual verification. All cited sources were subsequently obtained through conventional academic databases and cross-checked for authenticity, accuracy, and relevance before inclusion in the thesis.

All use of generative AI was conducted in accordance with the University of Oslo's guidelines on academic integrity. The author remains fully responsible for the content, analysis, and conclusions presented in this thesis.

Chapter 1

Introduction

1.1 Track Reconstruction Challenges at High Luminosity

The upcoming High-Luminosity Large Hadron Collider (HL-LHC) era, scheduled to begin in 2030, will push particle physics experiments into an unprecedented data regime. The collision rate will increase by a factor of 6–7 compared to the current LHC [28], presenting severe computational challenges for event reconstruction and data processing. For ATLAS (A Toroidal LHC ApparatuS), one of the LHC's flagship detectors, this translates to up to 200 simultaneous proton–proton interactions per bunch crossing (the average per-bunch-crossing pile-up, denoted $\langle\mu\rangle$), which is nearly four times the approximately 55 collisions observed in previous runs. [28].

In this high-density environment, charged-particle track reconstruction becomes both a cornerstone and a major computational challenge of modern collider experiments. Track reconstruction enables precise measurements of particle momentum, electric charge, and decay vertices, information that is vital for fundamental physics research. In collider detectors such as ATLAS, these tracks are essential for reconstructing physics objects including electrons and conversion photons [10], muons [16], τ leptons [13], and jets [12]. They also play a crucial role in identifying jets containing b and c hadrons [8] and in determining event-level quantities that depend on these reconstructed objects. The goal of track reconstruction software is therefore to determine particle trajectories with high accuracy while maintaining manageable computational demands. However, the extreme pile-up and data complexity of the HL-LHC require significant upgrades to both detector hardware and reconstruction software to preserve ATLAS's scientific capabilities.

On the hardware side, the experiment will replace its current Inner Detector with a new all-silicon Inner Tracker (ITk), designed for higher granularity and radiation hardness [28]. On the software side, the upgrade is driven less by a fundamental incapacity of the legacy reconstruction code, which could in principle be incrementally adapted to HL-LHC conditions, than by the accumulated maintenance burden of a code base developed over more than two decades. The legacy implementation relies on overly deep inheritance hierarchies and tightly coupled components, and several of its original authors are no longer active in the project. These factors make sustained development and the integration of new techniques (such as GPU parallelisation and machine-learning-assisted reconstruction) increasingly costly to support inside the legacy framework. Paying down this technical debt by adopting a modern, modular replacement is therefore the primary motivation for the software upgrade, with scalability under HL-LHC pile-up a secondary concern, though closely related.

Figure 1.1 illustrates how the legacy reconstruction software scales on the current Inner Detector, where the per-event runtime grows steeply with $\langle\mu\rangle$. The figure does not predict the legacy code's behaviour on the upgraded ITk geometry at HL-LHC pile-up, which would require a dedicated ITk-geometry study, but what it does show is that the existing implementation's measured scaling on the present detector is already unfavourable. Combined with the maintenance considerations above, this provides further motivation for moving to a framework whose design prioritises both code health and scalable performance.

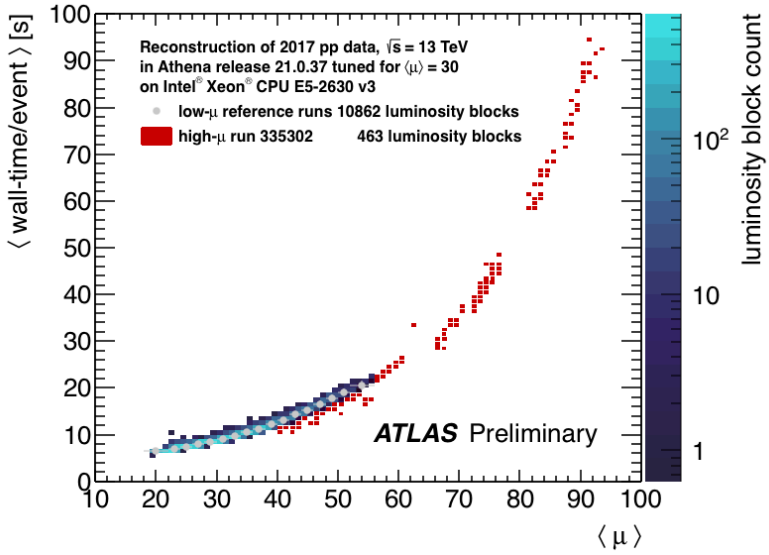


Figure 1.1: Reconstruction wall time per event as a function of the average number of interactions per bunch crossing $\langle\mu\rangle$, for the current Inner Detector with default tracking cuts. The legacy implementation exhibits steep growth with pile-up; the ACTS implementation scales approximately linearly. The measurement reflects the current detector geometry and should not be read as a projection of either implementation's behaviour on the upgraded ITk under HL-LHC pile-up, for which dedicated studies are reported in [17].

To address both the maintenance burden and the scaling concerns, the ATLAS collaboration, together with the broader High-Energy Physics (HEP) community, has developed a modern, modular alternative: ACTS (A Common Tracking Software) [6]. ACTS provides a parallel, geometry-agnostic framework designed for flexibility, performance, and maintainability, and is now shared across multiple HEP experiments. Within the ITk geometry the framework achieves stable behaviour, scaling approximately linearly with increasing pile-up. This architectural redesign provides the foundation on which HL-LHC track reconstruction can be built and refined.

Even with this redesigned foundation, the extreme data volumes of the HL-LHC make continuous optimisation of every component of the reconstruction pipeline worthwhile. The upgraded collider is projected to collect ten times more data in its first decade than the LHC collected in fourteen years, requiring processing of billions of collision

events. In this context, even small inefficiencies in reconstruction algorithms translate to substantial computing costs and processing delays. Significant effort within ACTS and the wider community has been directed at high-leverage parallelisation, including GPU offloading via `traccc` [26] and machine-learning-assisted track finding such as the GNN4Itk project [20]. The CPU performance of one particular stage, track seeding, has so far attracted less dedicated profiling attention than other components of the chain. Track seeding forms the foundation of the reconstruction pipeline, providing the initial trajectory estimates that guide all downstream processing, and a systematic profiling study of the ACTS seeding implementation therefore represents an opportunity for targeted CPU optimisation in the run-up to the HL-LHC era.

1.2 Seeding in Track Reconstruction

Among the multiple stages of the track reconstruction pipeline (seeding, track finding, and track fitting, illustrated in Figure 1.2), seeding plays the foundational role in creating charged-particle trajectories. A seed is an initial estimate of a particle trajectory, constructed from triplets of space points (SPs). A space point is the three-dimensional position reconstructed from one or more *hits*, the raw signals recorded when a charged particle deposits energy in an active sensor element. The seeding stage identifies triplets of space points across consecutive detector layers that are geometrically consistent with a charged-particle trajectory, providing the starting point for the downstream track-building process.

The seeding stage is a natural optimisation target because it directly influences both the efficiency and speed of the entire reconstruction chain, for two main reasons. First, if no valid seed is found for a given particle, its track cannot be reconstructed at all. Second, seeding is inherently combinatorial: it involves evaluating numerous combinations of detector space points to identify those likely originating from the same particle trajectory. For instance, with 10^4 space points per detector layer and three layers, a naive approach would require evaluating $(10^4)^3 = 10^{12}$ triplet combinations. To mitigate this, the ACTS framework employs a grid-based spatial binning strategy, which effectively reduces the computational scaling from $\mathcal{O}(N^3)$ to approximately $\mathcal{O}(N)$ by restricting the search to localized bins. This method substantially decreases the number of candidate triplets to be tested while maintaining sensitivity to genuine particle trajectories.

In realistic HL-LHC conditions the situation becomes even more challenging. Multiple overlapping collisions can produce tens of thousands of space points per layer, leading to a rapid increase in potential seed candidates. Each additional detector layer or increase in event multiplicity further amplifies the problem, as the algorithm must examine an increasing number of false combinations to identify the relatively few that correspond to real tracks. Generating too many seeds not only raises computational costs but can also degrade reconstruction performance by introducing fake or duplicate tracks.

Thus, the seeding algorithm must carefully balance high efficiency for genuine tracks with controlled CPU usage on combinatorial background. Achieving this balance is vital to maintain tracking performance in the dense, high-pile-up environment expected at the new collider, making seeding a natural and high-impact target for performance optimization. The computational intensity of this stage, combined with its foundational role in the reconstruction chain, underscores the value of understanding and optimising its performance.

This thesis presents a profiling study and optimisation effort targeting the computational bottlenecks of the ACTS seeding algorithm. While the framework provides the necessary architectural foundation for HL-LHC tracking, the extreme data volumes projected for this era make targeted profiling of every component worthwhile. This thesis systematically investigates the CPU performance of the ACTS seeding algorithm with the explicit goal of identifying and implementing optimisations that significantly increase processing throughput without degrading physics performance. By identifying and addressing specific inefficiencies, this research contributes to ensuring that track reconstruction remains computationally feasible under future operating conditions, thereby safeguarding the experiment's physics potential.

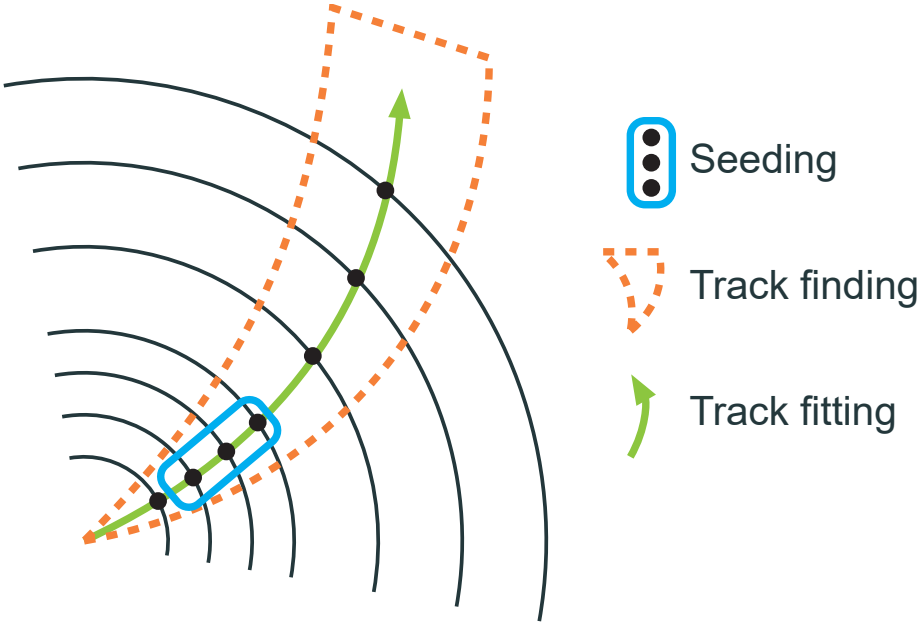


Figure 1.2: Illustration of a track reconstruction chain, starting from space points and proceeding through seeding, track finding, and track fitting to fully formed tracks. Taken from the ACTS documentation, "Tracking in a nutshell" [2].

1.3 Research Questions

This thesis investigates the CPU performance of the ACTS track seeding algorithm to identify optimizations that increase throughput without compromising physics performance. It addresses the following research questions:

RQ1. Profiling the original implementation. Where does the seeding algorithm spend most of its runtime, and what are the main computational bottlenecks?

RQ2. Developing optimizations. Can code-level or algorithmic optimizations improve runtime without degrading seeding quality metrics (efficiency, fake rate, duplicate rate)?

RQ3. Comparing implementations. How do the combined optimizations compare quantitatively to the baseline?

RQ4. Evaluating practical implications. What are the implications for HL-LHC-scale data processing?

We hypothesize that systematic profiling of the ACTS seeding algorithm will reveal specific computational bottlenecks, and that targeting these bottlenecks will yield a statistically significant reduction in execution time without degrading physics performance.

1.4 Thesis Structure

The remainder of this thesis is organized as follows.
Chapter 2 provides background on the High-Luminosity LHC environment, the ATLAS Inner Tracker, charged-particle track reconstruction, and the ACTS framework, with emphasis on the seeding algorithm and the performance metrics used in this work.

Chapter 3 introduces the profiling tools and methodologies used to analyze the computational performance of the ACTS seeding algorithm.

Chapter 4 describes the datasets used for profiling and physics validation.

Chapter 5 presents the experimental methods, including the computing environment, profiling workflow, statistical analysis of performance measurements, and physics validation procedures.

Chapter 6 presents and discusses the performance results and the impact of the implemented optimizations.

Chapter 7 summarizes the main conclusions and outlines directions for future work.
Supplementary material, including additional figures and technical details, is provided in the appendices.

Chapter 2

Background

This chapter provides background on the experimental and software context relevant to this thesis. It begins with an overview of the High-Luminosity Large Hadron Collider (HL-LHC) and its new Inner Tracker (ITk), outlining the motivation for more efficient tracking algorithms under high-luminosity conditions. The track reconstruction pipeline is then described, emphasizing the central role of the seeding stage as the foundation of the reconstruction process. The chapter concludes with an introduction to the ACTS framework and the performance metrics used to validate the computational and algorithmic changes presented in this work.

2.1 HL-LHC

The Large Hadron Collider (LHC) is one of the largest scientific projects ever undertaken. Since it began operations in 2010, it has been a cornerstone of fundamental particle physics research, culminating in the discovery of the Higgs boson. To extend its discovery potential into the 2030s and beyond, a major upgrade is underway [18], resulting in the High-Luminosity LHC (HL-LHC). In particle physics, the luminosity of a collider determines the rate of particle collisions; a higher luminosity therefore enables the study of extremely rare processes.

Upgrading such a large and complex machine is a time-consuming process. The effort, which spans the period of Long Shutdown 3 (LS3) from 2026 to 2029, has necessitated new surface and underground infrastructure. Scheduled to begin operations in 2030, the HL-LHC will dramatically increase the collider's output, raising its instantaneous luminosity by a factor of five and its integrated luminosity by a factor of ten compared to the original LHC design. This will allow the ATLAS and CMS (Compact Muon Solenoid) experiments to collect a total integrated luminosity of approximately $3000\,\text{fb}^{-1}$ over the first decade of the upgraded collider's operation, a sixfold increase over the roughly $500\,\text{fb}^{-1}$ collected during the LHC's first 14 years.

The primary challenge of this increased luminosity is the dramatic rise in pile-up ($\langle\mu\rangle$), the average number of simultaneous proton–proton interactions per bunch crossing. While the current LHC operates with $\langle\mu\rangle \approx 40$–$60$, the HL-LHC will face $\langle\mu\rangle = 140$ during Run 4 (starting around 2030) and $\langle\mu\rangle = 200$ in Run 5 [14]. This high-pile-up environment, illustrated in Figure 2.1, makes event reconstruction profoundly difficult. Each recorded event contains the overlapping debris from up to 200 independent collisions, creating a dense environment where thousands of particle tracks cross, merge, and obscure each other. This complicates critical tasks such as identifying the primary vertex from which particles originate and disambiguating individual particle trajectories.

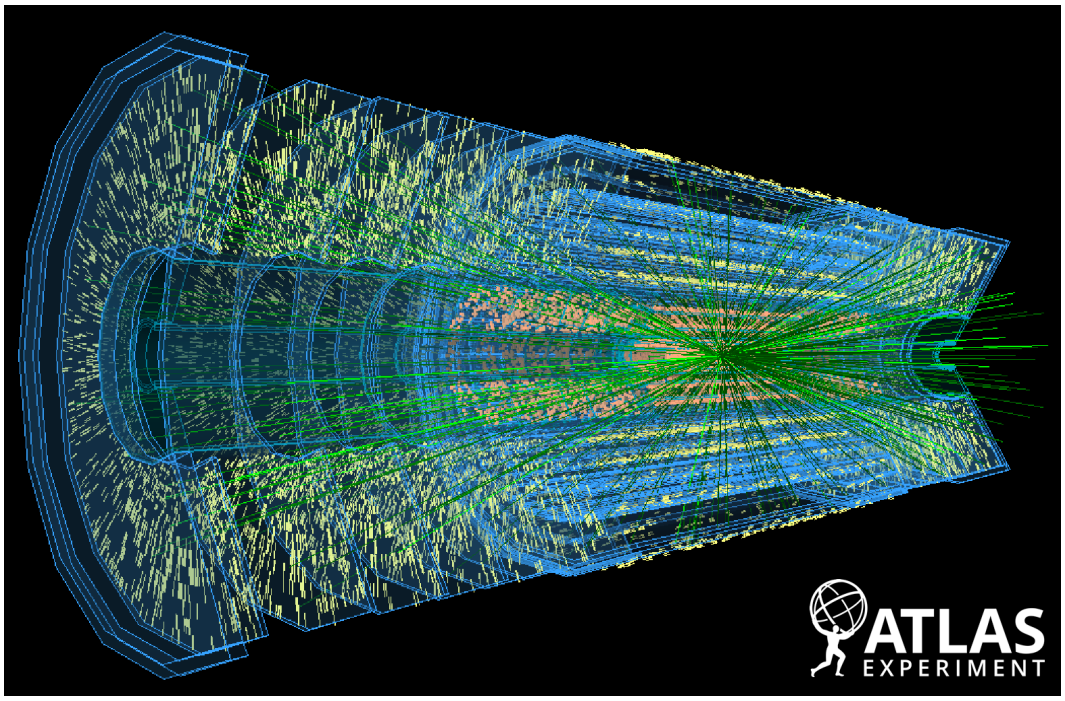


Figure 2.1: Simulated event in the intended layout of the ATLAS Inner Tracker (ITk), showing 40 signal muons overlaid with minimum-bias events consistent with an average pile-up of 140 collisions per bunch crossing. Reconstructed tracks are shown in green; pixel and strip clusters are shown in orange and yellow, respectively. Taken from [9].

To fully exploit this physics opportunity, the tracking software must evolve to handle these extreme conditions. Particle tracking involves reconstructing the paths of charged particles as they traverse the detector's layers. This process becomes exponentially more difficult with higher pile-up due to its combinatorial nature. The combinatorial explosion is sharpest in the initial track seeding stage: naively matching three space points across N hits per layer yields N^3 candidate triplets to evaluate, so the number of candidate trajectories scales as the cube of the hit count. Inefficient algorithms would lead to significant data loss, slower processing speeds, or higher rates of misidentified particles, ultimately undermining the HL-LHC's core scientific objectives.

Improved software efficiency is not just a matter of managing the data flow; it is what enables the finer-grained physics analyses the collider is designed for, such as precise Higgs coupling measurements and searches for rare beyond-Standard-Model processes. Superior tracking allows for more accurate reconstruction of these events. The collider's high luminosity is a mixed blessing: it promises a new era of precision physics, but this promise is dependent on developing equally high-performance computational tools. Without continual improvements in tracking software efficiency, the HL-LHC's hardware investments cannot deliver their full physics potential, and it is this software frontier that this thesis seeks to advance. On the hardware side, the ATLAS experiment is addressing the challenge with a completely new Inner Tracker, discussed in Section 2.2.

2.2 ATLAS Inner Tracker

The ATLAS experiment, one of two general-purpose detectors at the Large Hadron Collider, relies on its inner tracking system to reconstruct charged-particle trajectories from collision events. This innermost subsystem records spatial measurements as particles traverse successive detector layers, providing the fundamental input for the track reconstruction algorithms that determine particle momenta, charges, and interaction vertices.

To meet the demands of the High-Luminosity LHC, ATLAS is replacing its current Inner Detector with a completely new Inner Tracker (ITk) [28]. The scale of this upgrade is substantial: the ITk will instrument approximately $180\,\mathrm{m}^2$ with more than five billion readout channels, compared to the current detector's $63\,\mathrm{m}^2$ and roughly 100 million channels. This nearly threefold increase in instrumented area and fiftyfold increase in channel count reflects the fundamental challenge of operating in the high-pile-up environment of the HL-LHC.

Unlike the present system, which combines silicon and gaseous detectors, the ITk will be entirely silicon-based. The layout consists of five pixel barrel layers close to the interaction point and four silicon strip barrel layers at larger radii, with corresponding disk structures in the forward regions. Coverage extends to $|\eta| = 4$[1], significantly beyond the current $|\eta| = 2.5$ limit, ensuring at least nine measurement points per track in the barrel region and up to thirteen in the end-caps. Simulations show that even at $\langle\mu\rangle = 200$, the ITk's tracking efficiency within the original acceptance matches that of the current detector at $\langle\mu\rangle = 38$, with around 85% efficiency in the newly instrumented high-$|\eta|$ region [28].

This enhanced granularity directly addresses the high-density environment of the HL-LHC, but simultaneously creates a significant computational challenge. More detector channels and finer segmentation translate to more recorded space points per event, precisely the input that drives the combinatorial complexity of track seeding. While the ITk hardware ensures that high-quality measurements survive the radiation environment and detector occupancy of 200 simultaneous collisions, the reconstruction software must efficiently process the resulting data volume. It is this software challenge, and specifically the optimization of the track seeding stage, that motivates this thesis.

2.3 Track Reconstruction

Track reconstruction transforms raw detector signals into precise particle trajectories, enabling measurements of momentum, charge, and decay vertices essential for physics analysis. At the HL-LHC, where up to 200 simultaneous collisions produce tens of thousands of space points per event, accurate and efficient tracking becomes especially demanding. The reconstruction process follows a multi-stage pipeline where each step builds upon the previous one, making early stages, particularly track seeding, foundational to overall performance.

[1] Pseudorapidity is defined as $\eta = -\ln!\left[\tan(\theta/2)\right]$, where θ is the polar angle measured from the beam axis. In this convention, $\eta = 0$ corresponds to the direction perpendicular to the beam, while $|\eta| \to \infty$ approaches the beam axis.

2.3.1 The Reconstruction Pipeline

The reconstruction chain consists of four sequential stages, illustrated in Figure 1.2:

Space point formation converts detector signals into three-dimensional coordinates. When a charged particle traverses silicon sensors, it deposits energy that is digitized and localized. In pixel detectors, each hit provides a direct 3D measurement. In strip detectors, hits from layers are combined to resolve the coordinate perpendicular to the strips, forming "space points" with improved positional accuracy.

Track seeding generates initial trajectory hypotheses from compatible triplets of space points. The seeding algorithm identifies combinations of points from different detector layers that satisfy geometric and kinematic constraints consistent with a charged particle on a helical trajectory. This is fundamentally a combinatorial problem: with N space points per layer, naive triplet formation scales as $\mathcal{O}(N^3)$. Seeds provide approximate track parameters (position, direction, and momentum) that guide the subsequent track-building. Because no track can be reconstructed without a seed, and because poor-quality seeds waste computational resources or produce fake tracks, seeding directly determines both the efficiency and speed of the entire reconstruction chain.

Track building extends seeds by progressively associating compatible space points from additional detector layers. Starting from the seed's parameter estimate, the algorithm searches for hits along the predicted trajectory, accounting for detector geometry, magnetic field effects, and multiple scattering. This stage constructs candidate tracks by iteratively incorporating measurements that improve the trajectory fit.

Track fitting refines candidate tracks to produce final, high-precision trajectories. Using all associated space points, the fit optimizes the track parameters, typically employing Kalman filtering or χ^2 minimization to obtain the best estimate of the particle's momentum, charge, and impact parameters. The output is a set of fitted tracks characterized by five parameters defined at the perigee, the point of closest approach to the beam line.

2.3.2 Track Parameterization

A fitted track is described by five global parameters defined at the perigee [21]:

- d_0: transverse impact parameter (distance from the beam line in the transverse plane).
- z_0: longitudinal impact parameter (position along the beam axis).
- ϕ: azimuthal angle of the track momentum.
- θ: polar angle of the track momentum.
- q/p: particle charge divided by momentum magnitude.

These parameters, illustrated in Figure 2.2, fully define the track's kinematics and geometry, forming the basis for particle identification, vertex reconstruction, and physics analysis.

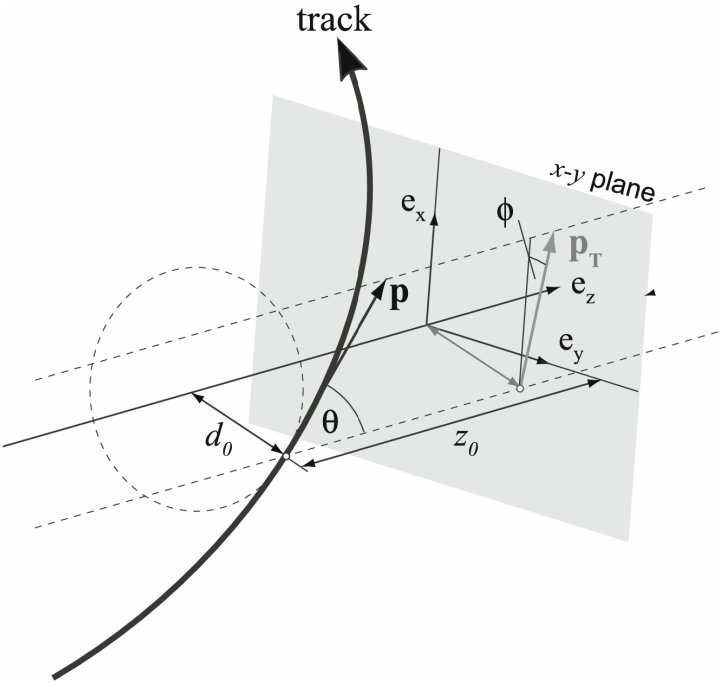


Figure 2.2: Illustration of the five global track parameters d_0, z_0, ϕ, θ, and q/p, defined relative to the perigee, the point of closest approach to the beam line. Reproduced from [21].

2.3.3 Computational Bottlenecks

The combinatorial nature of seeding makes it the primary computational bottleneck in the reconstruction pipeline. While track building and fitting also scale with the number of hits, their computational cost is bounded by the number of seeds. In contrast, seeding must evaluate a vast space of possible triplet combinations before any filtering can occur. At $\langle\mu\rangle = 200$, a single event may contain tens of thousands of space points across all detector layers (the concrete counts for the datasets used in this thesis are given in Chapter 4). Even with spatial binning strategies that reduce effective complexity from $\mathcal{O}(N^3)$ to approximately $\mathcal{O}(N)$, the absolute number of candidate triplets remains very large.

This bottleneck has direct consequences: inefficient seeding either limits physics performance (if too many genuine seeds are rejected) or consumes excessive CPU resources (if too many fake candidates are produced). Modern tracking frameworks such as ATLAS Run 3 achieve 70–85% reconstruction efficiency for isolated tracks through carefully tuned seeding cuts and ambiguity resolution [15]. However, maintaining this performance under HL-LHC conditions requires continuous algorithmic optimization. The ACTS framework, described in Section 2.4, provides the architectural foundation for this optimization effort, but realizing its full potential demands detailed profiling and targeted improvements to individual components, beginning with the seeding stage.

2.4 ACTS Framework

Modern high-energy physics experiments require reconstruction software that is both computationally efficient and adaptable to evolving detector technologies and computing architectures. Legacy tracking frameworks, often developed for specific experiments and hardware configurations, have become increasingly difficult to maintain and extend. The ATLAS Inner Detector tracking software, developed in the early 2000s, shows these limitations clearly: a largely monolithic architecture with deep inheritance hierarchies andThe ATLAS Inner Detector tracking software, developed in the early 2000s, shows these limitations clearly: a largely monolithic architecture with deep inheritance hierarchies and limited parallelisation has made the introduction of new techniques (such as GPU offloading or modern multi-threading) increasingly costly. The runtime behaviour of the legacy implementation on the current Inner Detector is shown by the steeply-rising curve in Figure 1.1, where the per-event time grows by an order of magnitude as $\langle\mu\rangle$ rises from 15 to 60.

To address these challenges, the ATLAS collaboration and the broader high-energy physics community developed A Common Tracking Software (ACTS) [2, 3, 5, 6]. ACTS is a modular, experiment-independent C++ framework designed for thread-safe and scalable particle tracking across diverse detector geometries. By replacing experiment-specific code with reusable, well-tested components, ACTS supports sustainable development and consistent performance optimisation across the HEP community.

The framework employs C++ templates for type safety and compile-time optimisation, enabling experiment-agnostic algorithms that operate across different coordinate systems without code duplication. Functionality is organised into independent modules (geometry, magnetic-field handling, propagation, track finding, seeding, vertexing, and others), each with clear interfaces that let components be replaced or extended without affecting the rest. All algorithmic modules are stateless and thread-safe at the API level, so that frameworks built on top of ACTS can process events in parallel without contending on internal state.

ACTS reconstruction wall time grows roughly linearly with pile-up on the current Inner Detector, whereas the legacy implementation rises steeply over the same range (Figure 1.1). Both curves come from 2017 pp data reconstructed on the current ID geometry, with each stack running its own production tracking configuration; the comparison is therefore at fixed geometry but unmatched cuts. It should not be read as a projection to the upgraded ITk under HL-LHC pile-up, where the finer-grained, silicon-only layout and the HL-LHC tracking configuration developed for it change both the absolute timing and the scaling. What Figure 1.1 does establish is that ACTS's scaling on a real detector is much flatter than the legacy code's, a strong indication that it is a viable foundation for HL-LHC reconstruction once the ITk-specific work is layered on top.

The architectural redesign also serves a longer-term purpose independent of pile-up scaling: it pays down the technical debt of a two-decade-old code base and establishes a maintainable foundation on which future optimisations, including those described in this thesis, can be developed and shared across experiments. ACTS has been adopted as the production tracking framework by ATLAS for the HL-LHC ITk, and is also in use at sPHENIX, Belle II, and FASER, demonstrating its flexibility across different collision environments and detector configurations [2].

2.4.1 The ACTS Seeding Algorithm

The ACTS seeding algorithm [5] identifies initial trajectory hypotheses by forming and evaluating triplets of space points from consecutive detector layers. An effective seeding strategy must balance three competing objectives: high efficiency (finding at least one seed per reconstructable particle), low fake rate (minimising seeds from random hit combinations), and low duplicate rate (limiting multiple seeds per particle). Achieving this balance in the dense HL-LHC environment, where tens of thousands of space points coexist per event, requires careful algorithmic design.

Algorithm Structure

The seeding process operates in two stages: the Seed Finder constructs and evaluates candidate triplets, and the Seed Filter ranks and selects final seeds based on quality metrics.

The Seed Finder begins by iterating over space points in a designated middle detector layer. For each middle space point, it searches for compatible points in inner (bottom) and outer (top) layers. To manage the $\mathcal{O}(N^3)$ combinatorial complexity of naive triplet formation, space points are projected onto a rectangular grid in the (r, z) plane, and searches are restricted to neighbouring grid bins (Figure 2.3). This spatial binning reduces the effective complexity to approximately $\mathcal{O}(N)$ with respect to the number of middle-layer space points [5], making the algorithm scalable to high-occupancy events.

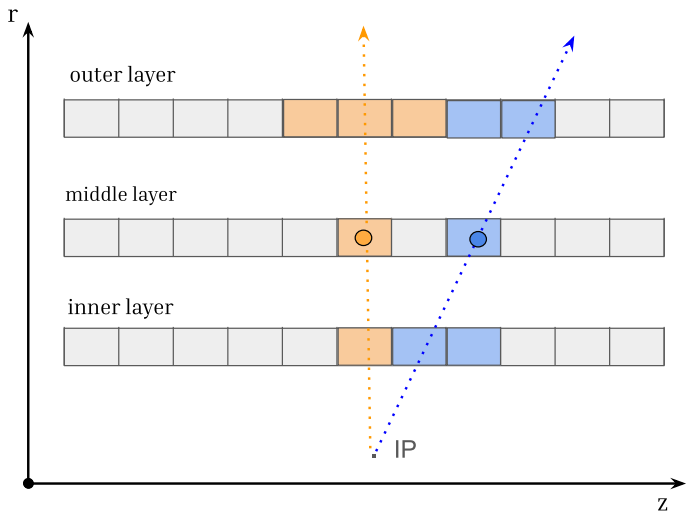


Figure 2.3: Triplet search in the (r, z) plane. Colored bins indicate the search neighborhood. Taken from [5].

For each candidate bottom–middle–top triplet, a series of geometric and kinematic compatibility checks are performed:

Pair selection. Bottom–middle and middle–top pairs are tested for compatibility with the interaction point and required trajectory properties. Configurable cuts on

pseudorapidity (η), longitudinal position (z), and inter-point distance reject incompatible combinations early.

Helix evaluation. To assess whether three points lie on a valid helical trajectory, a conformal mapping transforms the problem from three dimensions to two. Space points (x, y, z) are mapped to conformal coordinates:

$$u = \frac{x}{x^2 + y^2}, \qquad v = \frac{y}{x^2 + y^2}. \tag{2.1}$$

In this space, a circle in the transverse plane becomes a straight line $v = Au + B$. The helix diameter $2R$ relates to the line parameters through

$$(2R)^2 = \frac{A^2 + 1}{B^2}. \tag{2.2}$$

Since transverse momentum is proportional to the radius of curvature ($p_T = 0.3 \cdot B_z \cdot R$ for B_z in tesla and R in metres), a minimum diameter cut directly enforces a minimum p_T threshold:

$$\frac{A^2 + 1}{B^2} > (2R_{\min})^2 = \left(\frac{2\, p_T^{\min}}{0.3 \cdot B_z} \right)^2. \tag{2.3}$$

Compatibility checks. The triplet is evaluated for consistency with originating from the interaction region (small transverse impact parameter d_0) and for collinearity in the (r, z) plane, accounting for expected multiple scattering effects.

Triplets passing these cuts are assigned a seed weight, as described below, and forwarded to the Seed Filter.

Seed Filtering and Selection

The Seed Filter ranks candidates using a configurable quality metric. The default seed weight combines several factors:

$$w = c_1 \cdot N_t - c_2 \cdot d_0 - c_3 \cdot |z_0| + (\text{detector-region terms}), \tag{2.4}$$

where N_t is the number of compatible top space points, d_0 is the estimated transverse impact parameter, and z_0 is the estimated longitudinal vertex position. The constants c_1, c_2, and c_3 are tunable parameters that balance different quality criteria. The intuition for the signs is geometric: genuine prompt tracks originate close to the primary interaction region, so they should have small d_0 and small $|z_0|$, and they should leave many compatible hits along the predicted trajectory, so N_t should be large. The weight therefore rewards seeds consistent with a prompt-vertex origin (positive N_t coefficient) and penalises seeds that look like secondary interactions or random combinations of unrelated hits (negative d_0 and $|z_0|$ coefficients). The "detector-region terms" are configurable additive bonuses or penalties applied per detector region, typically used to give seeds formed entirely within the precision-pixel layers a small bonus relative to seeds that span pixel and strip sub-detectors, or to weight central seeds differently from forward seeds where hit density and resolution differ.

In practice, the constants c_1, c_2, c_3, and the detector-region terms are tuned empirically on simulated samples: a grid or sequential single-parameter scan is run over a representative event set, and the configuration that maximises seeding efficiency at a target fake rate is kept. ATLAS uses separate configurations for the pixel and strip

subdetectors, since their hit densities and impact-parameter distributions differ; finer η- or z-dependent tuning is possible but is not exposed in the default configuration that serves as the baseline in this thesis.

When multiple seeds share the same bottom and middle space points, only the highest-weighted candidates are retained, subject to a configurable maximum. This pruning step is essential for controlling combinatorial output while prioritising the most promising trajectory hypotheses for downstream track building.

Algorithmic Complexity

The grid-based binning strategy is the key enabler of near-linear scaling. For each middle-layer space point, only space points in adjacent bins of the bottom and top layers are examined. If k_b and k_t denote the average number of space points in these relevant bins, the computational cost per middle point is $\mathcal{O}(k_b \cdot k_t)$, which is effectively constant for fixed bin sizes and moderate occupancy. The total cost therefore scales as $\mathcal{O}(N_m)$, where N_m is the number of middle-layer space points.

ACTS parallelises reconstruction at the event level: multiple events are processed concurrently across threads, while the seeding pipeline within each event runs sequentially. The thread safety of the algorithmic modules is what makes this safe. This thesis works within a single thread and focuses on reducing the constant factors inside the $\mathcal{O}(N_m)$ scaling, rather than on the question of parallelising the algorithm itself.

2.5 Performance Metrics

A seeding algorithm has to be evaluated on two fronts: computational cost (runtime and resource use) and physics performance (how accurately it reconstructs particle trajectories). This section establishes the metrics used to evaluate the ACTS seeding implementation and to validate that optimisations improve computational performance without degrading reconstruction quality.

2.5.1 Computational Performance Metrics

Computational performance is assessed using standard timing metrics that characterise the execution efficiency of the reconstruction algorithms.

Execution time is reported throughout this thesis as wall-clock time (T_{wall}), the total elapsed time from the start to the completion of a task. CPU time (T_{CPU}), defined as the cumulative time the processor spends executing instructions, is approximately equal to the wall-clock time for the CPU-bound benchmarks considered here, that is, $T_{\text{CPU}} \approx T_{\text{wall}}$. It therefore provides no additional information, and wall-clock time alone is used.

2.5.2 Physics Performance Metrics

Physics performance metrics quantify how well the seeding algorithm recovers true particle trajectories, with Monte Carlo truth as the reference. A *truth particle* (equivalently, a true or reconstructable particle) is a Monte Carlo charged particle that satisfies the standard reconstructability requirements: it originates from the interaction

region, lies within the detector acceptance, and leaves hits on at least the minimum number of layers required by the seeding algorithm.

Because this work is restricted to the seeding stage, no downstream track building or fitting is applied. The seeds produced by the algorithm are therefore the reconstructed objects against which truth matching is performed, and the metrics below are referred to as *seed metrics*: matching is done at the seed–particle level rather than the hit level used in full track-finding studies. In the ACTS performance-collector implementation, these quantities are computed by the shared track-finder writer with seeds playing the role of reconstructed tracks; the per-track denominators in the code are the per-seed denominators here. Hit-level metrics require a fully built track and are out of scope for a seeding-only evaluation, so the three seed-level metrics below characterise the seeding output.

- **Efficiency** (ε). The fraction of truth particles for which at least one truth-matched seed is produced. High efficiency means the algorithm captures nearly all reconstructable particles.
- **Fake ratio** (f). The fraction of produced seeds that fail the ACTS truth-matching criteria: no single truth particle accounts for a sufficient fraction of the constituent measurements. A low fake ratio means the algorithm rarely produces spurious seeds from random hit combinations.
- **Duplicate ratio** (d). The fraction of produced seeds that are redundant matches to a truth particle already represented by another, higher-quality seed (with seed quality defined by the ACTS seed-filter ranking). A low duplicate ratio means the algorithm produces close to one representative seed per particle.

Formally,

$$\varepsilon = \frac{N_{\text{matched}}}{N_{\text{particles}}}, \tag{2.5}$$

$$f = \frac{N_{\text{fake}}}{N_{\text{seeds}}}, \tag{2.6}$$

$$d = \frac{N_{\text{dup}}}{N_{\text{seeds}}}, \tag{2.7}$$

where $N_{\text{particles}}$ is the number of reconstructable truth particles in the event, N_{seeds} the total number of seeds produced, N_{matched} the number of truth particles with at least one matched seed, N_{fake} the number of seeds that fail truth matching, and N_{dup} the number of seeds that match a truth particle already represented by a higher-quality seed.

A good seeding configuration combines high efficiency with low fake and duplicate ratios. Computational optimisations are therefore accepted only if all three metrics are preserved within statistical uncertainty relative to the baseline.

Chapter 3

Introduction to Profiling Tools

Profiling is the systematic measurement and analysis of software execution behavior with the goal of identifying performance bottlenecks and optimization opportunities. In the context of High-Luminosity LHC tracking, where reconstruction algorithms must process events with extreme detector occupancy, profiling is an essential tool for for meaningful performance optimization.

This chapter introduces the profiling methodologies and tools used to analyze the CPU performance of the ACTS seeding algorithm (Section 2.4.1). While the tools discussed are general-purpose, the presentation is guided by the specific characteristics of track seeding: a highly combinatorial workload with complex memory access patterns and strong sensitivity to event-by-event variation. The emphasis is therefore on profiling approaches that can reliably identify dominant runtime contributions while minimizing measurement bias.

3.1 Types of Profiling

Software profilers can be broadly categorized by how execution information is collected. This thesis employs two complementary approaches: statistical sampling and instrumentation-based profiling. Each provides different insights into algorithm behavior and is applied at different stages of the optimization workflow.

3.1.1 Statistical Sampling Profilers

Statistical sampling profilers interrupt the program at regular intervals and record the current call stack. Each sample is a snapshot of what the program is doing at that moment; over many samples, the fraction of samples in which a given function is on the stack approximates the fraction of runtime it consumes. Because each sample captures the full call stack, the chain of nested calls leading to the currently executing instruction, aggregating across samples also yields a probabilistic call graph, with each edge $A \to B$ weighted by the number of samples in which A called B directly. The graph is approximate, since it only reflects paths the sampler happened to catch, but converges to the true call-frequency structure as more samples are collected.

The primary advantage of sampling profilers is their low overhead, typically imposing only a small performance cost [29]. This makes them well suited for profiling production code and long-running applications. However, because sampling provides statistical estimates rather than exact measurements, short-lived functions that execute between samples may be missed.

This limitation can be mitigated by adjusting the sampling frequency, which introduces a trade-off between measurement accuracy and profiling overhead. Each sample takes a stack walk and a buffer write, both effectively constant per sample, so total overhead is linear in the sampling rate. Higher frequencies (e.g. $1000\,\mathrm{Hz}$) therefore capture more detailed execution behaviour at the cost of greater intrusion on the measurement, while lower frequencies (e.g. $100\,\mathrm{Hz}$) reduce overhead at the cost of statistical precision. The default sampling frequency used in this work is $100\,\mathrm{Hz}$, corresponding to approximately one sample every $10\,\mathrm{ms}$, which provides a reasonable balance between measurement fidelity and runtime overhead for most applications."

3.1.2 Instrumentation-Based Profilers

Instrumentation-based profilers record every function call and instruction executed by a program, providing deterministic measurements but often introducing significant runtime overhead [33].

Despite this slowdown, instrumentation profilers are extremely useful for detailed analysis of specific code regions, particularly after sampling profilers have identified potential performance bottlenecks. Several approaches can be used to introduce instrumentation: during compilation (requiring the program to be rebuilt), by modifying the compiled executable, or by executing the program within a simulation environment such as Valgrind. Each method involves different trade-offs between ease of setup, measurement accuracy, and runtime overhead.

Table 3.1 summarizes the key trade-offs between statistical sampling and instrumentation-based profiling approaches.

Table 3.1: Comparison of statistical sampling and instrumentation-based profiling, summarising the trade-offs between the two approaches.

Characteristic	**Sampling**	**Instrumentation**
Overhead	1–5%	10–50× slowdown
Accuracy	Statistical estimate	Deterministic
Short-lived functions	May miss	Captures all
Setup complexity	Minimal	Requires rebuild or framework
Use case	Initial exploration	Detailed diagnosis
Call graph detail	Approximate	Complete

The choice between these approaches follows a natural workflow: sampling profilers are used to identify performance bottlenecks with minimal overhead, while instrumentation tools provide the detailed measurements required for targeted optimization. This thesis employs both methods within the two-phase profiling strategy described in Section 5.2.

3.2 Core Profiling Tools

This work adopts a two-phase profiling strategy. Lightweight sampling tools are first used to identify performance-critical regions of the seeding algorithm. More detailed instrumentation-based tools are then applied to these regions to diagnose underlying causes and guide optimisation decisions.

gperftools (Google Performance Tools)

gperftools provides a low-overhead, sampling-based CPU profiler designed to identify performance hotspots under realistic execution conditions. Rather than instrumenting every function call or instruction, the profiler periodically samples the program counter during execution and records the active call stack. Aggregating these samples yields a statistical approximation of where wall-clock time is spent, typically with only 1–5% runtime overhead [29]. This makes gperftools well suited for profiling large workloads and input datasets where preserving realistic execution behaviour is essential.

In the profiling methodology used in this thesis, gperftools serves as the primary tool for hotspot identification. Its low perturbation ensures that observed bottlenecks reflect genuine performance characteristics rather than artefacts introduced by profiling itself. Profiling is enabled by setting the `CPUPROFILE` environment variable to the desired output path, with the sampling frequency set to 100 Hz via `CPUPROFILE_FREQUENCY`. The resulting profile is not interpreted directly; instead, the recorded samples are analysed using the `pprof` tool.

pprof

`pprof` reconstructs call stacks from sampled data and presents execution costs through several complementary visualisation modes. These views are used iteratively, progressing from coarse-grained, high-level identification of expensive regions to fine-grained localisation of costs within specific functions. Each mode highlights a different aspect of program behaviour and provides a distinct perspective on program execution.

Text view. The text view presents profiling results as a tabular summary of functions with their self and cumulative sample counts, typically sorted by cumulative cost. The five columns reported by `pprof` are: `flat` (the time spent executing instructions inside this function itself, excluding any callees); `flat%` (the same quantity as a percentage of total samples); `sum%` (the cumulative sum of `flat%` read top-to-bottom down the list, useful for answering "the top n functions cover what share of total runtime?"); `cum` (this function's time plus the time of all functions it calls); and `cum%` (the same as a percentage). For leaf functions, those that do not call anything visible in the profile, `flat` and `cum` are identical: no callee time is attributed to them. This view provides a compact overview of where execution time is concentrated and enables precise comparison between functions. It is particularly useful for identifying dominant hotspots in large profiles and supports caller and callee inspection for limited exploration of call relationships.

Call graph. The call graph view presents hierarchical function relationships with execution time attributed along caller–callee paths. Costs are reported both as *self time*, representing samples attributed directly to a function, and *cumulative time*, which includes all callees. This distinction is essential for determining whether a function is intrinsically expensive or simply acts as a dispatcher for costly operations. In practice,

```
flat  flat%   sum%        cum   cum%
13.6  30.1%  30.1%       13.6  30.1% Acts::SeedFinder::createSeeds
12.4  27.4%  57.5%       12.4  27.4% Acts::SeedFinder::filterSeeds
 8.1  17.9%  75.4%        8.1  17.9% std::vector::push_back
 5.3  11.7%  87.1%        5.3  11.7% Acts::SeedFinder::getCompatibleDoublets
```

Figure 3.1: Example `pprof` text output showing the most time-consuming functions. `flat` is self-time (sample count attributed to the function itself); `cum` is cumulative time including all callees; `sum%` is the running cumulative percentage of `flat%` from the top. In this example all four functions are leaves at the level shown in the profile, so `flat` equals `cum` and `flat%` equals `cum%` for each row.

this view is used to identify which call paths dominate execution time and where optimisation effort is most likely to have the largest impact.

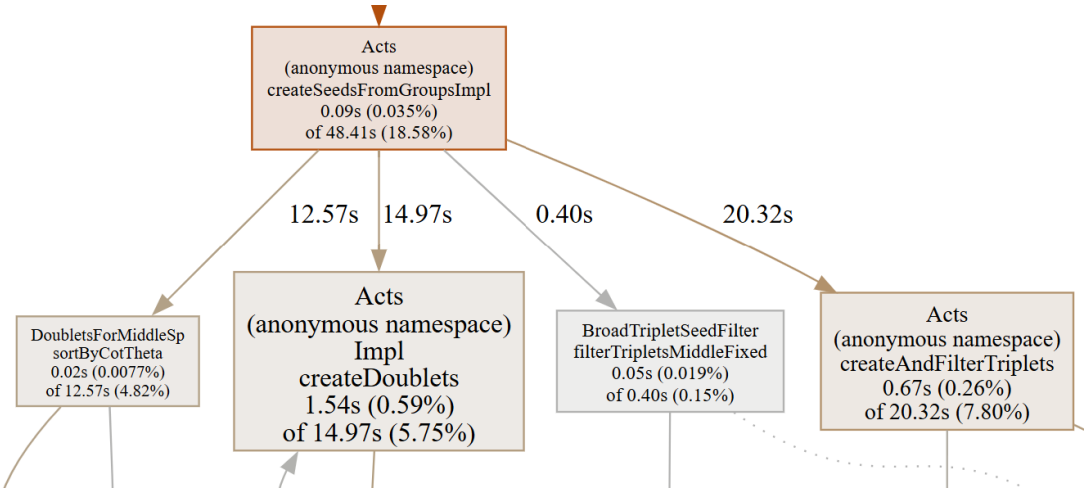


Figure 3.2: Example `pprof` call graph showing hierarchical function relationships and time attribution.

Flame graph. Flame graph visualisations [30] provide a structural overview of program execution by encoding cumulative cost as horizontal width and call stack depth as vertical height. Each rectangle represents a function, with wider regions indicating higher cumulative sample counts. Within each stack level, functions are ordered alphabetically rather than temporally, enabling pattern recognition at the expense of execution order information.

Flame graphs are particularly effective for revealing the overall "shape" of performance bottlenecks, such as wide plateaus indicating dominant subsystems, deep towers suggesting excessive call depth or recursion, and fragmented layers that may indicate frequent function calls or missed inlining opportunities.

Line-level profiling. Once dominant functions or call paths have been identified, `pprof` supports line-level profiling to localise costs within individual functions. When debug information is available, sampled execution time can be attributed to specific source-code lines, highlighting loops, branches, or statements responsible for the majority of a function's runtime. This mode is applied selectively to confirmed hotspots, enabling

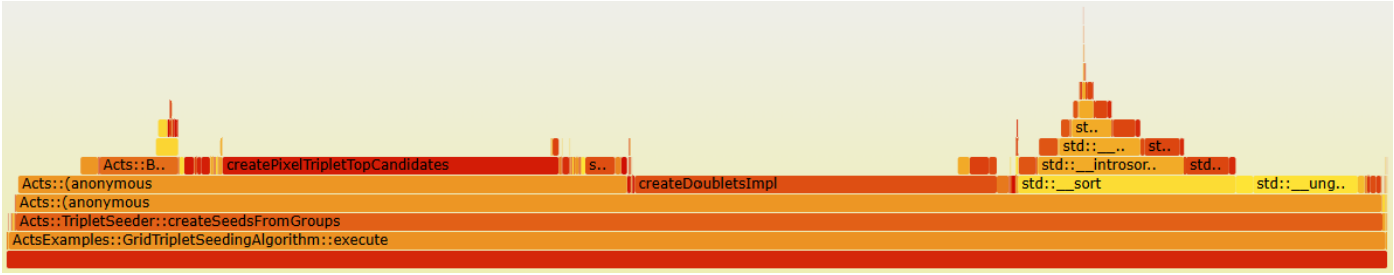


Figure 3.3: Example flame graph visualisation showing the call stack hierarchy with width proportional to sample count.

targeted micro-optimisations without prematurely focusing on code with negligible overall performance impact.

```
Total: 12.5s
ROUTINE ======================== processRequest
  3.20s  (25.6%)   for i := 0; i < len(items); i++ {
  7.85s  (62.8%)       result += expensiveComputation(items[i])
  0.45s  ( 3.6%)   }
```

Figure 3.4: Example line-level profiling output showing time attributed to individual lines of code.

Although sampling-based profiling may miss very short-lived functions or infrequently executed code paths, its robustness and minimal overhead make gperftools an effective foundation for performance analysis. In this thesis, all detailed investigations using instrumentation-based profilers are guided by hotspots first identified with gperftools and `pprof`, ensuring that optimisation efforts remain grounded in observed wall-clock behaviour.

Valgrind: A Framework for Dynamic Binary Instrumentation

While sampling-based profilers such as gperftools are effective at identifying where execution time is spent, they provide limited insight into why a particular code path is expensive. For detailed diagnosis of performance bottlenecks, this thesis employs profiling tools built on the Valgrind dynamic binary instrumentation framework [33].

Valgrind is not a profiler itself but a runtime instrumentation platform that executes programs within a synthetic CPU environment. At runtime, native machine instructions are translated into an intermediate representation, instrumented with analysis code, and executed in a controlled environment. This design enables precise, deterministic observation of program behaviour at the granularity of individual instructions, memory accesses, and function calls, without requiring source-code modification.

The primary trade-off of this approach is significant runtime overhead. Depending on the analysis tool, Valgrind-based profiling typically incurs a slowdown of approximately 10–50× compared to native execution. This perturbation affects cache behaviour, branch prediction, and thread scheduling, making Valgrind unsuitable for establishing absolute performance measurements. Instead, its strength lies in comparative analysis: explaining the underlying causes of bottlenecks previously identified by low-overhead profilers and guiding targeted optimisation efforts.

Callgrind: Instruction-Level Profiling, Cache Simulation, and Call Graph Analysis

Callgrind is a Valgrind tool [33] that provides detailed instruction-level profiling together with optional simulation of cache hierarchies and branch prediction. It records the total number of executed instructions, memory accesses, cache misses, and branch mispredictions, while attributing these costs to individual functions and call paths.

Conceptually, the reported quantities can be interpreted as follows: instruction counts measure total computational work, cache misses quantify memory locality and data movement costs, and branch mispredictions capture control-flow inefficiencies. In contrast to sampling-based profilers, which provide statistical estimates of runtime, Callgrind yields deterministic counts of these observables and decomposes them across the program structure.

Callgrind additionally constructs a complete call graph, distinguishing between *self cost* (instructions executed directly within a function) and *inclusive cost* (including all callees). This enables identification of computational hotspots, inefficient call patterns, and unnecessary abstraction overhead.

The underlying cache model is simplified and does not capture hardware features such as prefetching, out-of-order execution, or complex replacement policies. Consequently, absolute miss rates may differ from hardware counter measurements. However, relative comparisons between profiling runs reliably identify code regions with poor locality or inefficient execution patterns, provided the comparisons are performed on the same hardware and the same Valgrind build, so that any modelling biases cancel in the ratio.

Due to its instrumentation-based design, Callgrind introduces substantial runtime overhead, typically 20–50×. It is therefore applied selectively to representative workloads or isolated code regions to explain performance bottlenecks identified by low-overhead profiling tools.

In this thesis, Callgrind is invoked with cache simulation, branch simulation, jump collection, and instruction-level dumps enabled (`-cache-sim=yes` activates the D1/Ll cache model so that data-access miss counters are populated; `-branch-sim=yes` enables the branch-predictor model and the corresponding misprediction counters; `-collect-jumps=yes` records jump and conditional-branch information needed for control-flow analysis within a function; `-dump-instr=yes` produces instruction-level output that allows the cost of each machine instruction to be inspected when source-line attribution is ambiguous). Instrumentation is deferred at startup (`-instr-atstart=no`) and activated from within the seeding module using the `CALLGRIND_START_INSTRUMENTATION` macro, so that ACTS initialisation is excluded from the measurement. Callgrind output can additionally be inspected interactively using KCachegrind or its Qt-only build QCachegrind, which provide flat profile, caller–callee, call graph, and source-annotation views over the same profile data. In this thesis, the metrics in Table 3.2 are reported directly from the textual `callgrind_annotate` output.

Heaptrack: Memory Allocation Profiling

While CPU profiling identifies where execution time is spent, memory allocation behaviour can independently impose significant performance and scalability limitations. Frequent dynamic allocations and high peak memory usage can degrade performance in two main ways: *cache pollution*, where allocator bookkeeping and freshly returned memory blocks evict the program's hot data from the CPU caches, and *memory pressure*,

Table 3.2: Selected Callgrind metrics used to assess computational cost, memory behaviour, and control-flow efficiency. Each metric is interpretable as a physical observable: instruction counts measure total work, cache misses quantify costly data-fetch delays, and branch mispredictions signal disruptions to the execution pipeline.

Metric	Definition / Interpretation
Ir	Total executed instructions; a direct measure of computational work, analogous to counting the elementary operations required to complete a calculation.
Data accesses (Dr + Dw)	Total memory load and store operations, reflecting how frequently the program reads from or writes to memory.
LLC data misses (DLmr + DLmw)	Accesses not satisfied by any cache level, requiring a fetch from main memory. These high-latency events are analogous to retrieving a resource from a distant location rather than a local cache.
L1 instruction-cache miss reads (I1mr)	Occur when the next instruction is absent from the L1 instruction cache, indicating poor code locality: execution jumps between distant code regions rather than proceeding sequentially.
Branch mispredictions (Bcm)	Occur when the processor incorrectly predicts a conditional branch outcome, discarding speculative work and stalling the pipeline, analogous to taking the wrong path and having to backtrack.

where the program's resident set grows large enough to push the operating system into more aggressive paging and, in extreme cases, swap or out-of-memory conditions. A third mechanism, allocator contention between threads competing for a shared lock on each malloc/free, is relevant in multithreaded runs but does not apply to the single-threaded measurements in this thesis. Heap behaviour is analysed with Heaptrack, a low-overhead allocation profiler [35].

Heaptrack records dynamic memory allocations performed by a program by intercepting calls to allocation functions such as `malloc`, `free`, `new`, and `delete`. For each allocation, Heaptrack captures the allocation size, timestamp, and full call stack using stack unwinding. Allocation events are streamed to disk during execution and analysed post-mortem, enabling attribution of memory usage to specific code paths.

Unlike Valgrind-based memory profilers, which run the program through a synthetic CPU and re-execute every instruction inside an interpreter loop, Heaptrack runs the program natively on the real CPU and intercepts only the small set of allocator entry points via `LD_PRELOAD`. Native execution is preserved for all non-allocator instructions, so the perturbation is restricted to the allocator call sites themselves. This design typically introduces only a 2–5× runtime overhead, low enough to profile realistic workloads and long-running executions, allowing memory behaviour to be studied under conditions that closely resemble production runs.

In this thesis, Heaptrack is used to evaluate two key aspects of memory behaviour: peak heap memory usage and allocation count. Peak heap memory captures the maximum amount of heap memory simultaneously in use during execution, serving as an indicator of overall memory footprint and a constraint for scalability on memory-limited systems. Allocation count reflects the total number of dynamic memory allocations performed during execution, capturing allocation overhead that can manifest as allocator contention and cache pollution. Heaptrack is run through its native wrapper with default

flags, and all allocations made by the child process are recorded to a compressed trace. Table 3.3 provides definitions of these two metrics.

Visualisation tools provided by Heaptrack include flame graphs weighted by allocation size or allocation count, as well as timeline views showing how memory usage evolves throughout program execution. These representations are particularly effective for identifying allocation hotspots and the execution phases in which peak memory usage occurs.

Table 3.3: Heaptrack metrics used for evaluating memory behaviour and optimisation effects. The table focuses on peak memory footprint and allocation overhead, which are the indicators most relevant for the performance analysis in this thesis.

Metric	Definition
Peak Heap Memory	Maximum heap memory simultaneously in use during program execution, indicating overall memory footprint.
Allocation Count	Total number of dynamic memory allocations performed during execution, indicating allocation overhead.



Figure 3.5: Example Heaptrack flame graph showing allocations weighted by their contribution to total allocated memory [35].

Heaptrack is used here to follow up on memory issues spotted during CPU profiling: hotspots dominated by allocation overhead, or workloads with unnecessarily high peak memory. The typical fixes fall into three categories: reusing objects to cut down on dynamic allocations, swapping in a custom allocator or allocation scheme where the

default one is a bottleneck, and moving short-lived allocations from the heap to the stack where possible.

Heaptrack only tracks heap allocations, not stack ones. This would be a serious blind spot for code dominated by stack-resident locals, but the seeding workload studied here keeps its main data structures (the space-point grid, the doublet and triplet containers, the seed-output buffers) on the heap, so the limitation does not affect the analyses that follow. Distinguishing identical allocations from different call paths also takes some care when reading the call-stack output. Within these limits, Heaptrack strikes a good balance between observability and perturbation, and complements the sampling-based CPU profiling used elsewhere in this work.

3.3 Limitations of Profiling Tools

All profiling techniques introduce measurement artefacts, since the act of observation perturbs program execution. The tools employed in this work trade realism for observability in different ways: sampling-based profilers preserve native execution at the cost of statistical uncertainty, while instrumentation-based tools yield deterministic counts but execute programs in a synthetic environment with substantial slowdown. No single profiler tells the whole story; each one exposes a different aspect of program behaviour. The limitations outlined below motivate the staged methodology described in Section 5.2.

Aggregation effects. Profiling data is typically aggregated over the full run, which can hide variability between phases or execution paths and underweight rare but expensive events. The runs profiled here use representative workloads, and individual hot components are examined at finer granularity where needed.

Symbolisation. Accurate attribution of profile samples to source code requires debug information. Compiler optimisations like inlining and code reordering blur function boundaries and complicate the instruction-to-source mapping, which can distort how cost appears to be distributed across functions.

Complementary scope. Each profiler captures only a subset of performance-relevant behaviour, and the three tools used here are chosen to be complementary rather than overlapping. Sampling profilers preserve realistic execution but can miss short-lived or infrequently executed paths. Instrumentation-based tools give deterministic instruction-level detail at the cost of perturbing execution and relying on simplified hardware models; their results are therefore only directly comparable when measurements are taken on the same hardware and the same Valgrind build, so that any modelling biases cancel in the ratio. Heaptrack covers heap allocation independently of either, with the heap-only scope already discussed above. Reliable analysis therefore depends on combining these tools rather than relying on any one of them in isolation.

Chapter 4

Datasets

This chapter introduces the datasets used to measure the computational and physics performance of ACTS track seeding. Two are used: a *profiling dataset* for runtime characterisation and timing measurements of algorithmic and code-level optimisations, and a *physics validation dataset* used to confirm that those optimisations preserve reconstruction quality and to measure the computational performance of changes that affect physics behaviour. Neither is a full production sample. Both are representative test samples, realistic enough to test the algorithm under production-like conditions, while supporting reproducible implementation-to-implementation comparisons.

The goal of this work is not to characterise the detector's intrinsic performance or physics reach, but to optimise the computational behaviour of the seeding algorithm while keeping the physics performance metrics defined in Section 2.5.2 within tolerance.

4.1 Profiling Dataset

The profiling dataset comprises 50 simulated events supplied directly by a collaborator within the ACTS project [1]. These events were provided as a pre-generated sample intended primarily for algorithm development and performance evaluation.

All events in this dataset originate from a single simulation production job, so geometry, reconstruction configuration, and underlying simulation conditions are internally consistent within the sample. The choice of 50 events was made upstream: it is sufficient to exhibit substantial event-to-event fluctuations in detector occupancy and the associated combinatorial complexity, and small enough to support multiple profiling cycles with low-overhead performance analysis tools. The sample is treated strictly as a computing-test input. The sample is not large enough to be statistically representative of HL-LHC data in any meaningful physics sense, and no such claim is made; for the present purpose, relative timing and hotspot comparisons between implementations of the same algorithm, it does not need to be.

4.1.1 Event Characteristics

As an initial exploration of the sample, distributions of the truth-level particles, space points, clusters, and reconstructed tracks can be drawn from the profiling dataset. Table 4.1, Figure 4.1, and Figure 4.2 provide a concise overview of these characteristics, including event-to-event fluctuations and the kinematic distributions of the reconstructed tracks. The individual quantities plotted in Figures 4.1 and 4.2 are described in the respective figure captions. The sample is too small to determine whether these

distributions are realistic in any rigorous sense; the most that can be said is that they are representative of the kind of high-occupancy, HL-LHC-like input the seeding algorithm is expected to face.

Table 4.1: Event-averaged detector occupancy and track multiplicity for the 50-event profiling dataset. CV denotes the coefficient of variation (standard deviation / mean × 100). Pixel space points† correspond one-to-one with pixel clusters by detector geometry; the lower space-point count for the strip detector relative to its clusters reflects the merging of adjacent strip hits into shared space points.

Quantity	Mean ± SD	CV (%)	Min	Max
Reconstruction				
Truth particles	$103,836 \pm 9,206$	8.9	84,592	123,352
Tracks	$2,007 \pm 217$	10.8	1,496	2,505
Pixel detector				
Clusters	$244,135 \pm 22,410$	9.2	196,141	289,892
Space points†	$244,135 \pm 22,410$	9.2	196,141	289,892
Strip detector				
Clusters	$230,042 \pm 19,497$	8.5	176,076	270,848
Space points	$90,449 \pm 10,786$	11.9	68,480	114,546

These summary statistics serve two purposes. First, they show that the dataset exhibits substantial event-to-event variation in detector occupancy and the associated combinatorial complexity, which is what is needed to exercise the seeding algorithm under representative, computationally demanding conditions. Second, they provide quantitative context for interpreting the profiling measurements, since execution time and hotspot structure are expected to depend on the underlying event complexity. The values reported are not intended as a physics validation of the underlying simulation.

Metric extraction and calculations. From the profiling dataset, the following quantities are extracted directly from the ROOT file via the `GNN4ITk` tree. Per-event truth-particle and track multiplicities are read from the `nPartEVT` and `nTRK` branches respectively. Clusters are classified as pixel or strip based on the `CLhardware` field, which takes the values `PIXEL` or `STRIP`. Space points are classified analogously using the `SPCL2_index` field: a value of -1 denotes a pixel space point, whereas any other value denotes a strip space point. This classification scheme is consistent with the `RootAthenaDumpReader`, the component used throughout this work to read the profiling dataset, and reproduces the space-point multiplicities observed during execution of the seeding algorithm.

Track kinematic observables η, p_T, and ϕ are computed from the reconstructed track-momentum components p_x, p_y, and p_z, accessed via the `TRKperigee_momentum` branch, using the standard relations

$$p_T = \sqrt{p_x^2 + p_y^2}, \qquad \eta = \ln\left(\frac{p + p_z}{p_T}\right), \qquad \phi = \operatorname{atan2}(p_y, p_x), \tag{4.1}$$

where $p = \sqrt{p_x^2 + p_y^2 + p_z^2}$ denotes the total momentum magnitude. This is the

momentum-coordinate form of pseudorapidity, $\eta = -\ln\tan(\theta/2)$. The same track population is used as input to the seeding algorithm.

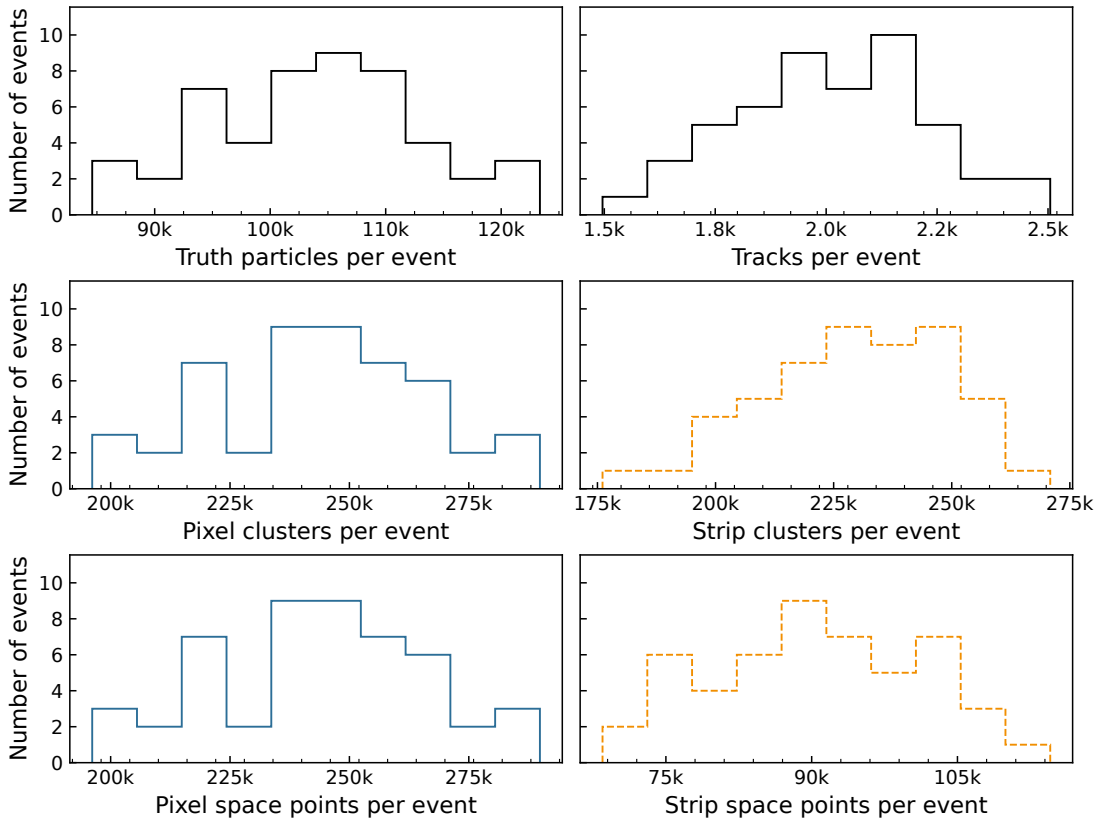
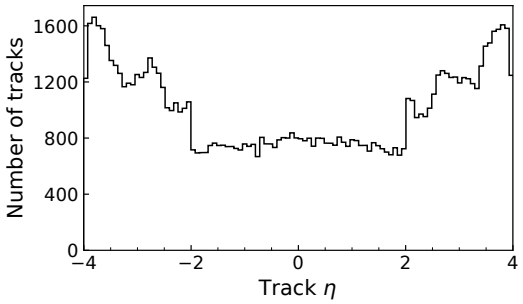
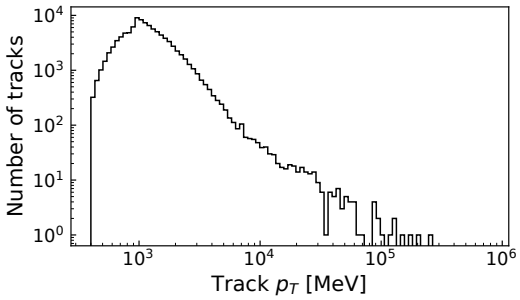


Figure 4.1: Per-event multiplicity distributions for the 50-event profiling dataset. Top row: truth-particle multiplicity (left) and reconstructed-track multiplicity (right). Middle row: cluster multiplicity for the pixel (left) and strip (right) detectors. Bottom row: space-point multiplicity for the pixel (left) and strip (right) detectors. Pixel and strip quantities are shown in solid blue and dashed orange respectively; detector-agnostic quantities (truth particles and tracks) are shown in black.

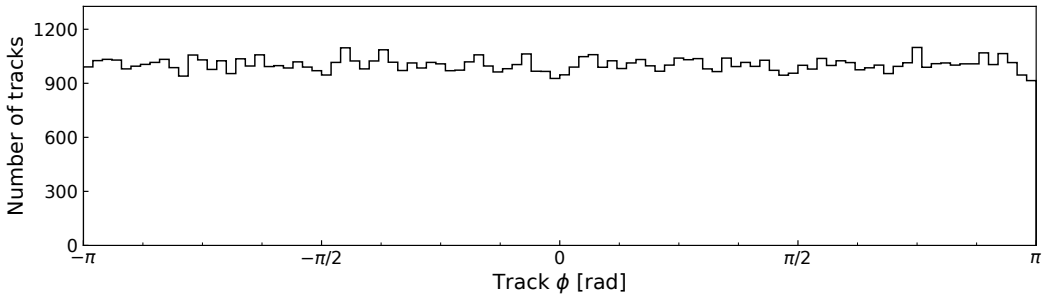
Usage in computational profiling. The profiling dataset is used to characterise the computational performance of code-level optimisations, changes to the implementation of the seeding algorithm that do not alter its physics behaviour. The two datasets differ not in size or representativeness, but in infrastructure: the profiling sample is a stand-alone collection of space points without the surrounding ACTS validation pipeline (truth-matching, reconstructable-particle selection, seed-quality bookkeeping), and the seeding algorithm can be run directly on it from the `RootAthenaDumpReader`. This makes it ideal for repeated profiling runs of a code-level change, where the only quantities of interest are wall-clock time and the Callgrind/Heaptrack hotspot structure, and where verifying physics performance is unnecessary because the algorithm's outputs are by construction unchanged. By contrast, an optimisation that does alter physics behaviour (a new grid layout, a modified cut, a different filtering rule) has to be measured on the physics-validation dataset, where the full truth-matching infrastructure makes it possible to confirm that efficiency, fake ratio, and duplicate ratio are preserved alongside the timing change (Section 4.2).



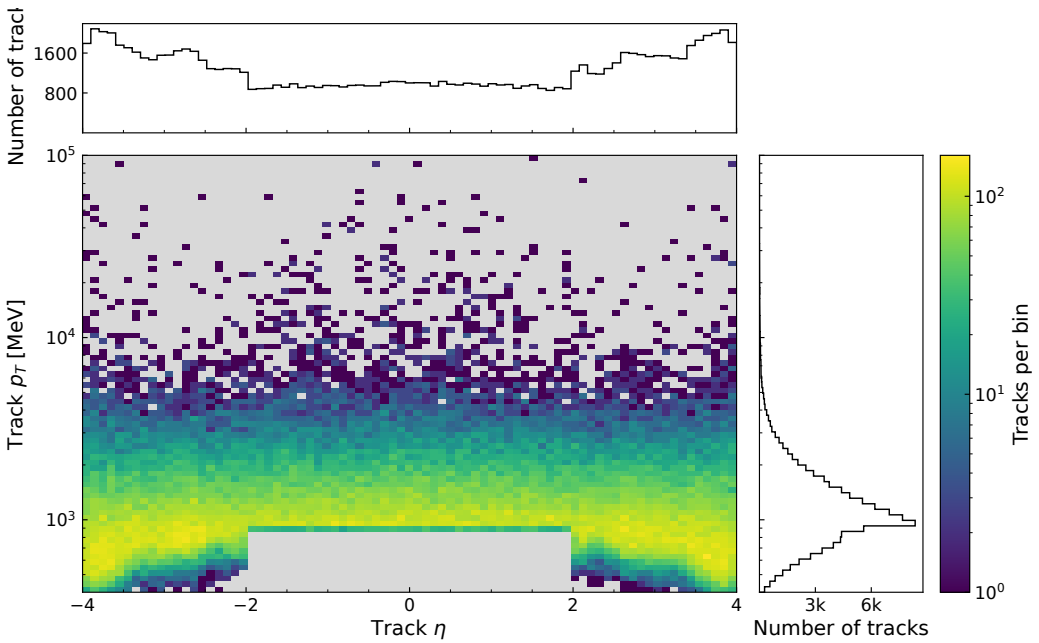
(a) Pseudorapidity η.



(b) Transverse momentum p_T.



(c) Azimuthal angle ϕ.



(d) p_T versus η with marginal η (top) and p_T (right) projections.

Figure 4.2: Reconstructed-track kinematic distributions for the 50-event profiling dataset. The η shape reflects the geometric acceptance of the ITk, which extends to $|\eta| = 4$. The p_T spectrum is steeply falling and shown on a logarithmic scale. The ϕ distribution is approximately uniform, consistent with the azimuthal symmetry of the detector; significant deviation from flatness would indicate detector inefficiencies or other issues in the sample. In panel (d), bin colour encodes track density on a logarithmic scale; bins shown in light grey contain zero reconstructed tracks. The empty band in the central region ($|\eta| \lesssim 2$, $p_T \lesssim 900$ MeV) reflects a minimum-p_T selection applied during track reconstruction.

All profiling runs on this dataset use identical algorithm configurations, by which we mean the user-tunable parameters of the seeding algorithm (cuts, tolerances, grid bin sizes, seed-filter weights, and so on) together with the runner script that drives the measurement. Holding all of these fixed ensures that observed performance variations originate solely from code-level or algorithmic modifications rather than from changes in input data or configuration parameters.

While absolute timing measurements are intrinsically dependent on the specific dataset used, the use of a single, consistent dataset ensures that relative changes in execution time and in the distribution of computational hotspots can be reliably attributed to the implemented optimisations.

4.2 Physics Validation Dataset

The physics validation dataset comprises 50 simulated events generated under HL-LHC-like conditions. Each event consists of a $t\bar{t}$ hard scatter, generated with PYTHIA8 [19] from the `Top:qqbar2ttbar=on` process, overlaid with 200 minimum-bias PYTHIA8 events to produce an average pile-up of $\langle\mu\rangle = 200$. The primary vertex of each interaction is sampled from a Gaussian beam-spot model with standard deviations $\sigma_{x,y} = 0.0125$ mm and $\sigma_z = 55.5$ mm. Physics validation focuses on the seed metrics defined in Section 2.5.2: efficiency, fake ratio, and duplicate ratio. The sample size is configurable, but 50 events provides sufficiently small statistical uncertainties for meaningful physics-performance comparisons while keeping overall processing time manageable.

This dataset also supports timing measurements for optimisations that modify the algorithm in ways that affect physics behaviour. For such optimisations, timing on the profiling dataset alone is insufficient, as that sample does not support the physics-metric evaluation required to confirm that performance remains within acceptable tolerances. Using the physics-validation dataset for both purposes ensures that any optimisation of this kind is characterised jointly: the timing benefit is quantified on the same sample on which its physics impact is assessed. For optimisations that do not alter physics behaviour, computational performance is measured exclusively on the profiling dataset, and the physics-validation dataset is used only to confirm that physics metrics remain unchanged relative to the baseline.

4.2.1 Event Configuration

Following generation, particle propagation through the detector is handled by the FATRAS fast-simulation framework [23], with the resulting hits and clusters fed into the standard ACTS reconstruction chain. The pile-up level of $\langle\mu\rangle = 200$ corresponds to the anticipated High-Luminosity LHC (HL-LHC) operating conditions and is set explicitly in the execution script that drives the dataset.

Unlike the profiling dataset, physics validation requires detector geometry information, as truth matching and physics-performance metrics depend on the full reconstruction chain. For this purpose, the Open Data Detector (ODD) geometry is used [7, 27]. ODD is a generic and simplified detector geometry developed by the ACTS collaboration as an experiment-independent reference. It enables validation of algorithmic behaviour without introducing detector-specific complexity, while significantly reducing simulation and reconstruction time and thereby facilitating rapid prototyping. Although ODD simplifies certain geometric details, it preserves the essential characteristics relevant for tracking and seeding performance, provided that

algorithmic optimisations do not rely on detector-specific features. The optimisations in this thesis target algorithm-level inefficiencies that do not depend on the detector layout, so the ODD geometry is sufficient for the physics-validation studies here.

The seeding algorithm and reconstruction configuration are kept identical to those used in the profiling studies (Section 4.1), ensuring that any observed differences in physics performance can be attributed to algorithmic changes rather than to differences in detector configuration or reconstruction settings.

4.3 Strip Spacepoint Processing

Both the profiling and physics-validation datasets contain pixel and strip space points, which the seeding algorithm processes as separate collections. In both `Seeding` and `Seeding2`, strip space points are run through the pixel algorithm code path: `useDetailedDoubleMeasurementInfo = false` in `Seeding`, `useStripInfo = false` in `Seeding2`. The per-triplet strip coordinate correction , which uses strip-detector geometry metadata to refine the approximate space-point positions produced by strip clustering, is therefore not applied during seeding.

This is forced by the input data formats. The profiling-dataset ROOT file (GNN4ITk dump format) is missing the strip-detector geometry fields required by the strip-aware code path: `topStripVector`, `bottomStripVector`, `stripCenterDistance`, and `topStripCenter` [20]. The format was designed for GNN-based tracking rather than ACTS seeding, so these fields were never part of the dump. The physics-validation dataset, produced under the Open Data Detector (ODD) [27] geometry, supplies the full ACTS particle/hit/measurement/space-point chain consumed by the standard examples reader, but likewise does not provide the strip metadata required by the strip-aware code path. With the strip-specific fields absent, the strip-aware code path cannot be exercised on either dataset.

Because the same pixel-path-for-strip configuration applies to `Seeding` and `Seeding2` alike, the computational performance comparisons between the two implementations remain valid: the two algorithms are doing the same work, with the same inputs, in the same configuration. The strip *physics* results reported in this thesis should however be interpreted as the pixel algorithm applied to strip space points, not as the full strip-aware reconstruction that would be used in a production environment with a complete geometry description.

Chapter 5

Methods

This chapter describes the methods used to measure performance reproducibly. Computational performance is quantified with bootstrap-based confidence intervals, which separate real algorithmic gains from system-level variability. Physics performance is compared between baseline and optimised implementations run under identical conditions, so that any difference can be attributed to the change itself, and the reconstruction quality is either preserved or traded off against speed according to criteria defined in advance.

5.1 Experimental Environment

All profiling and performance evaluations were carried out using the ACTS framework within a controlled computational environment to ensure reproducibility and to reduce systematic variations to a minimum. Comprehensive build instructions are provided in Chapter A.

5.1.1 Computing Infrastructure

All experiments were performed on the HEPP02 computing node of the High Energy Physics Platform (HEPP) at the University of Oslo. The hardware and system configuration are summarised in Table 5.1. All entries correspond to documented hardware or software specifications; no measured performance values are included.

The node is equipped with approximately 2 TiB of RAM (roughly 1 TiB per socket) connected via 8 DDR4 channels per socket, yielding a theoretical peak memory bandwidth on the order of 190–205 GB/s per socket depending on the installed DIMM speed. This is substantially in excess of the bandwidth demands of the seeding workload considered in this thesis. The optimisations studied target instruction count, cache locality, and algorithmic structure rather than sustained memory throughput. Memory bandwidth is therefore not a limiting factor for the measurements reported here and is not analysed separately.

Containerisation. All measurements were conducted within an Apptainer container to ensure a consistent and reproducible software environment, isolated from variations in the host system. Apptainer is a containerisation platform specifically designed for high-performance computing (HPC) environments. The container image, based on AlmaLinux 9.7, encapsulates all required runtime dependencies and ensures compatibility across systems with CVMFS access. This container-based approach

Table 5.1: Hardware and system specifications of the HEPP02 computing node.

Component	Specification
CPUs	2 × AMD EPYC 7702 (64 cores/socket)
Instruction set	x86_64
Microarchitecture	Zen 2 (Rome)
Sockets	2
Cores	128 physical / 256 logical (SMT)
Frequency [GHz]	2.0 base, up to 3.35 boost
L1I/core [KiB]	32
L1D/core [KiB]	32
L2/core [KiB]	512
L3/socket [MiB]	256
Memory	≈ 2 TiB total (1 TiB/socket), 8 DDR4 channels/socket
Host OS	RHEL 8.10
Container runtime	Apptainer 1.4.4
Container OS	AlmaLinux 9.7

mitigates dependency conflicts and library version inconsistencies, thereby facilitating reproducible profiling results across heterogeneous computing facilities.

5.1.2 Software Environment

Software stack and dependencies. External dependencies (compiler toolchain, numerical libraries, data-format libraries, detector frameworks) were taken from the LCG 105 stack on CVMFS, build `x86_64-el9-gcc13-opt`. This view supplies `gcc` 13.1.0, `Boost` 1.82.0, `Eigen` 3.4.0, `ROOT` 6.30, `DD4hep` 1.26, `Geant4` 11.2, `Pythia8`, and the `Python` 3.9 interpreter used for the experiment scripts. The runtime environment was initialised in every shell by sourcing the LCG view:

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-el9-gcc13-opt/setup.sh
```

ACTS framework. The ACTS framework itself is not provided by LCG; it was cloned from the upstream repository at release version **v44.0.0** and compiled locally from source following the official ACTS build guidelines [4]. CMake ($\geq$ 3.14) was used as the configuration system, with GNU Make as the build driver. All commands shown in this section are wrapped in the `environment/setupACTS.sh` script in the accompanying repository, , which also installs a static `git-lfs` binary if one is not already present; this is required for the OpenDataDetector submodule, whose classifier weights are stored as LFS blobs.

Core dependencies.

- **GCC 13.1.0** — C++20 toolchain (required by ACTS v44.0.0 for `std::format`).
- **Boost** $\geq$ 1.77 — with `filesystem`, `program_options`, and `unit_test_framework`.
- **Eigen** $\geq$ 3.3.7 — linear algebra.
- **Pythia 8** — $t\bar{t}$ event generation for the ODD pipeline.
- **ONNX Runtime** — inference backend for the ODD duplicate-seed classifier.
- **gperftools** — CPU profiling; installed locally rather than from LCG 105.

Build configuration. The ACTS library was compiled in **Release** mode (`-O3 -DNDEBUG`) to match production deployment conditions and to report peak achievable performance, with the following CMake options:

```
-DCMAKE_BUILD_TYPE=Release
-DACTS_BUILD_EXAMPLES=ON
-DACTS_BUILD_EXAMPLES_PYTHON_BINDINGS=ON
-DACTS_BUILD_EXAMPLES_PYTHIA8=ON
-DACTS_BUILD_EXAMPLES_ROOT=ON
-DACTS_BUILD_EXAMPLES_DD4HEP=ON
-DACTS_BUILD_PLUGIN_DD4HEP=ON
-DACTS_BUILD_PLUGIN_JSON=ON
-DACTS_BUILD_PLUGIN_ROOT=ON
-DACTS_BUILD_PLUGIN_ONNX=ON
-DACTS_BUILD_FATRAS=ON
-DACTS_BUILD_ODD=ON
-DACTS_ENABLE_CPU_PROFILING=ON
-DACTS_GPERF_INSTALL_DIR=<GPERF_INSTALL_PATH>
```

This configuration enables CPU profiling (via linked `gperftools`), support for the DD4hep detector-description framework (required by the OpenDataDetector), Pythia8-based event generation for the ODD pipeline, ONNX inference for the ODD duplicate-seed classifier, ROOT and JSON I/O, and Python bindings to drive the profiling and physics-validation experiments from script.

Profiling Tools

The following software tools were employed for performance profiling. A comprehensive description of their configuration and usage is provided in Section 5.2. All profiling tools were accessed via the LCG 105 software stack.

Table 5.2: Profiling and analysis tools used in this work.

Tool	Source	Version	Purpose
gperftools	LCG 105	bundled	Sampling-based CPU profiling; hotspot identification.
pprof	gperftools	bundled	Profile analysis and visualisation.
Valgrind	standalone	3.22.0	Dynamic binary instrumentation framework.
Callgrind	Valgrind	bundled	Instructions, memory, and control-flow behaviour.
Heaptrack	standalone	1.3.0	Memory allocation profiling.

5.1.3 Thesis Code Repository and Reproducibility

Repository overview. All software artefacts developed for this thesis are publicly available at https://github.com/Aksth070600/acts-seeding-optimization. Archival snapshots are deposited on Zenodo under the concept DOI 10.5281/zenodo.20126578, which always resolves to the latest archived release. The snapshot matched to this thesis is `v1.0.2` (10.5281/zenodo.20128085). The repository contains the modified ACTS source files, data-generation scripts that drive the profiling workflows, figure-generation scripts that consume the profiler output, and pipeline drivers that sequence the two. ACTS source modifications live under `acts-overrides/`, organised by algorithm variant (`Seeding/`, `Seeding2/`, `Seeding3/`) and by optimisation family (`CodeOptimizations/`, `AlgorithmOptimizations/`), with a `clean/` subdirectory that restores every patched file to its baseline before any variant is applied.

Versioning and variant management. ACTS is built from a pinned upstream version (`v44.0.0`) by `environment/setupACTS.sh`. Each measurement variant is encoded as a self-contained subdirectory under `acts-overrides/` rather than as a separate branch or git tag; the `utils/CopyDir.sh` helper, invoked by the data-generation pipeline before every build, overlays the variant onto the baseline ACTS source tree. The mapping from each thesis figure or table to the script (and therefore the variant) that produces it is recorded in `figure-map.md` at the repository root. The exact state of the repository used for the figures and tables in this thesis is tagged `v1.0.2` and archived as the corresponding Zenodo snapshot above.

Reproducibility instructions. Stepwise reproduction instructions live in the top-level `README.md` and in per-directory `README.md` files under `environment/`, `acts-overrides/`, `data-gen/`, `figure-gen/`, and `pipelines/`. On a host with the LCG/Apptainer environment of Section 5.1.1 and a local gperftools install, reproduction reduces to `bash environment/setupACTS.sh` (one-time; clones, configures, and builds ACTS) followed by `source environment/setup.sh` (per-shell). `python3 pipelines/all.py` then drives the full data-generation and figure-generation chain end-to-end, while `python3 pipelines/figures_all.py` regenerates only the analysis figures and tables from existing raw profiler output. Each pipeline step is idempotent: it re-uses existing outputs and re-runs only what is missing or stale, so partial reproductions are cheap.

Licensing and data availability. All original code in the repository is released under the BSD 3-Clause licence. Modifications to the ACTS framework retain ACTS's original licence; the upstream project itself is unaffected by anything in `acts-overrides/`. The GNN4ITk profiling dataset of Section 4.1 is access-restricted — it is an internal ATLAS Athena dump supplied by the collaboration — and is not redistributed with the repository. Pipelines that depend on it self-skip when the dump is unavailable, so the ODD-only chapters (using the publicly available OpenDataDetector dataset of Section 4.2, which ships with ACTS via CVMFS) can be reproduced on any host with the required environment. Absolute runtimes reported in this thesis depend on hardware and software configuration; relative speedups, the ranking of optimisations, and physics-validation results are expected to be reproducible.

5.2 Profiling Methodology

The profiling methodology is designed to yield statistically robust and reproducible estimates of computational performance under steady-state execution conditions. The primary objective is to distinguish genuine performance improvements from variations induced by system noise, initialisation effects, or measurement artefacts.

A central principle of this methodology is to keep three concerns separate: locating where time is spent, measuring *how much faster* an optimisation makes the algorithm, and confirming *that the optimisation mechanically did what it was supposed to do.* Hotspot discovery is performed using low-overhead sampling on realistic input, because the perturbation of the running program must be small enough that the observed profile remains representative of normal execution. Reported speedups are computed from manual wall-clock timers placed around the relevant algorithmic region and analysed statistically over repeated runs and events. Mechanical confirmation is performed using instrumentation-based tools such as Callgrind and Heaptrack, which can count instructions, branches, cache behaviour, or heap allocations more directly, but at the cost of substantially perturbing execution time.

Mixing these roles would give misleading results. Instrumentation-based timing is biased by tool overhead and is therefore unsuitable for reported wall-clock speedups. Conversely, sampling-based profiling is statistical and can miss short or infrequently executed paths, making it unsuitable for confirming that a specific instruction-count or allocation change occurred. The methodology therefore uses sampling to locate hotspots, manual timing to quantify reported performance differences, and instrumentation only to confirm that an implemented change had the intended mechanical effect.

To ensure high statistical confidence, all measurements use the 50-event dataset described in Section 4.1. Space points are read from pre-computed ROOT files, so seeding is evaluated independently of the upstream detector simulation and clustering stages.

Instrumentation-based tools are reserved for a narrow confirmatory role: verifying that a proposed fix produced the expected mechanical change after implementation, rather than quantifying its performance impact. This two-phase workflow is designed to ensure that profiling overhead does not bias the measured performance or the inferred speedups.

5.2.1 Experimental Workflow

Warm-up Phase

A workload's first execution can incur one-time costs, including library loading, allocator initialisation, cache population, and branch-predictor training, that do not reflect steady-state behaviour. The per-event seeding runtime was therefore examined as a function of execution order to check that these effects do not bias the reported timings.

The per-event seeding runtime t is dominated by a linear dependence on the space-point multiplicity N_{SP}: each space point contributes a roughly constant amount of work, on top of a small fixed per-event overhead. This linearity is not assumed, but observed. It is shown explicitly in Figure 6.3, where per-event wall time is plotted against N_{SP} and is well described by a straight line. A linear model $t = aN_{\mathrm{SP}} + b$ was therefore fitted to the per-event timings, and the residuals $r = t - (aN_{\mathrm{SP}} + b)$ were used as a workload-corrected measure of per-event runtime. The fit is good ($R^2 \geq 0.99$ for both pixel and strip), so residuals close to zero with no trend in execution order indicate that

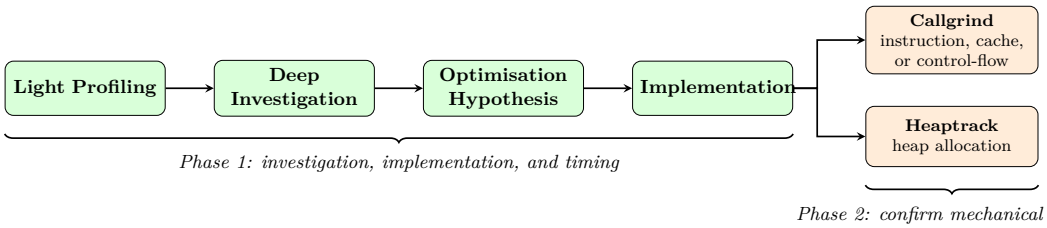


Figure 5.1: Profiling workflow. In Phase 1, `gperftools` sampling locates hotspots in the baseline, which are then investigated using ranked function lists, call graphs, flame graphs, and source annotations to form concrete optimisation hypotheses. After implementation, reported performance comparisons are derived from manual wall-clock timers instrumented directly in the code and evaluated statistically over repeated runs and events. They are not derived from `gperftools` profile samples. In Phase 2, Callgrind and Heaptrack are applied as independent confirmatory tools. Depending on the nature of the change, one or both may be run to verify that the expected mechanical change, in instruction counts, cache behaviour, control flow, or heap allocation, has taken effect.

no detectable warm-up effect remains. Any systematic drift would instead point to a residual dependence on execution order.

Figure 5.2 shows the per-event residuals, averaged across five independent runs, together with a centred ten-event running mean. The running mean uses a window of ±5 events around each point and is shrunk at the edges of the run: the first event therefore averages itself and the next five events, the second event averages itself, the previous event, and the next five events, and so on. No systematic trend is evident in either the pixel or strip datasets. The running mean remains approximately flat and centred on zero over the whole sequence, with no sign of elevated cost at the start of the run, and the deviation magnitudes are comparable throughout. This is inconsistent with the monotonic decay typical of a warm-up phase.

Any initialisation effects are therefore either negligible or indistinguishable from event-to-event variability in the residuals. No events are excluded from the reported timings, and the results can be taken as representative of steady-state performance.

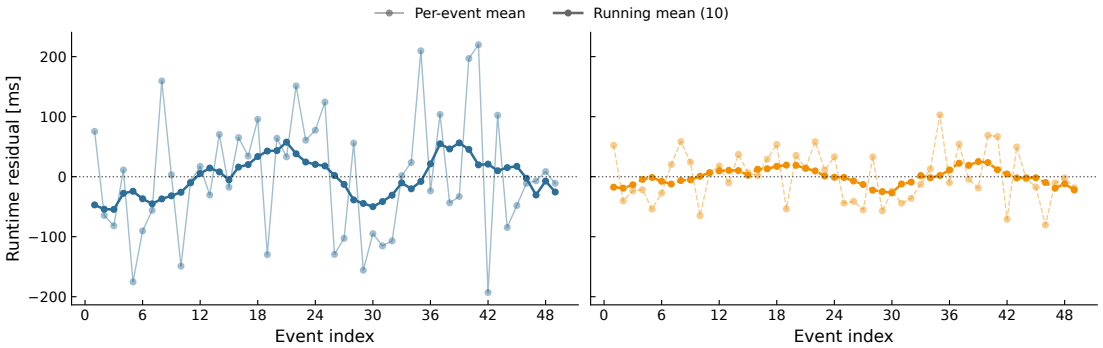


Figure 5.2: Runtime residuals $r = t - (a\,N_{\mathrm{SP}} + b)$ relative to a linear workload model fitted to the per-event timing measurements, shown for pixel (left) and strip (right) seeding. Thin lines show the per-event mean residual across five independent runs; thick lines show a centred ten-event running mean (±5 events around each point, truncated at the run boundaries so that event 1 spans events 1–6, event 2 spans events 1–7, and so on). The horizontal dotted line marks zero residual, i.e. perfect agreement with the linear workload model. No systematic decrease in residual is seen at early execution indices, indicating the absence of a detectable warm-up phase.

Phase 1: Lightweight Profiling, Hotspot Investigation, and Timing

Five independent steady-state executions are performed per configuration. During each profiling run, sampling-based CPU profiling is recorded at 100 Hz using `gperftools` (see Section 3.2). These profiles are used to identify where execution time is spent and to guide optimisation work; they are not used directly to compute reported speedups.

Profile data are analysed using multiple complementary views: ranked cumulative function lists, hierarchical call graphs, flame-graph visualisations, and line-level source annotations. Functions are investigated in decreasing order of cumulative runtime contribution until concrete optimisation hypotheses have been formed. Call graphs and flame graphs localise cost within the call tree, while source-level annotations identify the dominant instruction groups within each function. This investigation is sufficient to form a concrete optimisation hypothesis for each hotspot without requiring instrumentation-based tools.

After an optimisation has been implemented, its performance effect is measured using manual wall-clock timers instrumented directly into the code around the seeding stage. These timer measurements are collected for the baseline and optimised implementations over the same 50-event dataset and repeated runs. All reported speedups and confidence intervals are derived from these manual-timer measurements using the statistical procedure described in Section 5.2.2.

Phase 2: Implementation Confirmation

After each optimisation is implemented, one confirmatory Callgrind or Heaptrack run, or both, is performed depending on what mechanical effect the change is expected to produce:

- **Callgrind** provides the per-function instruction counts (`Ir`) and branch mispredictions (`Bcm`) shown in the results-chapter metrics tables, read from `callgrind_annotate` at two scopes: per event, meaning the total over the profiling dataset, and per function, meaning the cost attributed to the routines modified by each optimisation. Because Callgrind is instrumentation-based and deterministic, it resolves local changes that the wall-clock timer cannot, in particular inner-function work reductions masked by per-iteration timer overhead. Invocation details are given in Section 3.2.
- **Heaptrack** is used when the change targets memory-allocation behaviour. The Heaptrack summary is read to confirm that an expected reduction appears at the predicted call site. For example, when an up-front capacity reservation is intended to eliminate `std::vector` reallocations, Heaptrack is used to check that the corresponding allocation source no longer appears as a producer of new allocations and that the total allocation count for the affected function drops by the expected amount. Invocation details are given in Section 3.2.

Phase 2 is qualitative and confirmatory rather than quantitative. It establishes that the implementation change is mechanically correct; all reported performance numbers come from the manual-timer measurements in Phase 1. Because instrumentation profiling is slow and substantially perturbs wall-clock execution time, these analyses are run once over the full dataset, or over a representative subset when appropriate, and are not used to derive speedup estimates. Reproduction instructions for the Phase 2 pipelines are provided in the thesis code repository (Section 5.1.3).

5.2.2 Statistical Analysis of Performance Measurements

Performance differences are evaluated statistically rather than from single measurements. The analysis proceeds in three steps: per-event speedups are computed from paired baseline and optimised timings, these speedups are aggregated across events using the geometric mean, and the uncertainty of the aggregate speedup is quantified using a non-parametric bootstrap over events. This section describes each step in turn.

Throughout, the word *run* denotes one complete execution of the seeding pipeline over the entire 50-event dataset, and the word *event* denotes one collision event within such a run. The dataset of measurements therefore consists of $r = 5$ runs, each producing one per-event timing for every one of the $N = 50$ events, for a total of $r \cdot N = 250$ per-event timings per configuration.

Per-Event Speedup

For each of the $N = 50$ events, the $r = 5$ runs yield r paired timing measurements for the baseline and optimised implementations under the same benchmark conditions. Let

$$T^{(i)}_{\text{baseline}} = \{t^{(i)}_1, \ldots, t^{(i)}_r\} \quad \text{and} \quad T^{(i)}_{\text{optimised}} = \{t'^{(i)}_1, \ldots, t'^{(i)}_r\}$$

denote these measurement sets for event i, where the subscript $j \in \{1, \ldots, r\}$ indexes the run that produced the measurement.

The r measurements collected for a given event are repeated measurements of the same input, not independent event samples. Treating them as independent observations would therefore artificially inflate the sample size. The repeated measurements are instead averaged to produce one timing estimate per event and configuration:

$$\bar{t}^{(i)}_{\text{baseline}} = \frac{1}{r}\sum_{j=1}^{r} t^{(i)}_j, \quad \bar{t}^{(i)}_{\text{optimised}} = \frac{1}{r}\sum_{j=1}^{r} t'^{(i)}_j.$$

Although measurement-to-measurement variability across the r runs is consistently small in practice, the averaging provides a principled per-event estimate regardless of the observed variance. The event-level speedup is then

$$S_i = \frac{\bar{t}^{(i)}_{\text{baseline}}}{\bar{t}^{(i)}_{\text{optimised}}}. \tag{5.1}$$

Geometric Mean Aggregation

Performance improvements are multiplicative: a 1.5× speedup on one event and a 2.0× speedup on another combine geometrically, not arithmetically. The natural summary statistic is therefore the geometric mean of the event-level speedups, computed via the log-transform

$$\log S_i = \log \bar{t}^{(i)}_{\text{baseline}} - \log \bar{t}^{(i)}_{\text{optimised}}. \tag{5.2}$$

The overall speedup estimate is

$$\hat{S} = \exp\left(\frac{1}{N}\sum_{i=1}^{N} \log S_i\right), \tag{5.3}$$

the geometric mean of the per-event speedups. This is the standard summary for ratio-based performance metrics: it respects the multiplicative structure of speedups, and

it is invariant under inversion. The geometric mean of baseline-over-optimised is the reciprocal of the geometric mean of optimised-over-baseline, so the reported speedup does not depend on which implementation is chosen as the reference [25, 32].

In practice, the gap between the geometric and arithmetic means depends on the spread of the per-event speedup distribution. For tightly concentrated distributions the two agree closely; for wide or right-skewed distributions, the arithmetic mean is pulled upward by the upper tail. A toy example with two events makes the scale concrete: speedups S_1 and S_2 give an arithmetic mean $(S_1 + S_2)/2$ and a geometric mean $\sqrt{S_1 S_2}$. For a wide spread ($S_1 = 1.5$, $S_2 = 2.0$), the arithmetic mean is 1.750 and the geometric mean is $\sqrt{3.0} \approx 1.732$, a relative difference of roughly 1 %. For a tighter spread ($S_1 = 1.60$, $S_2 = 1.70$), the arithmetic mean is 1.650 and the geometric mean is $\sqrt{2.72} \approx 1.6492$, a relative difference below 0.05 %.

The per-event speedup distributions in this work fall closer to the second case: the bootstrap intervals reported in later chapters are typically within ±2 % of the point estimate. The choice between the two means therefore changes no reported number at the displayed precision. The geometric mean is nevertheless used throughout because it is the appropriate estimator for multiplicative quantities, is invariant under inversion, and is the standard choice in benchmarking.

Per-event time and throughput use the arithmetic mean. This directly answers the practical question: what does a single event cost, on average, under the observed workload? The geometric mean would be defensible under a log-normal timing assumption, and for the tight distributions seen here the two agree to within 1–3 %, a consequence of Jensen's inequality, $\langle 1/t \rangle \geq 1/\langle t \rangle$, which also means the arithmetic means of time and throughput are not exact reciprocals. The arithmetic mean wins out here because it is the conventional choice for absolute timing benchmarks, it is directly interpretable as expected cost, and it was used when the final results were generated. Speedup is a ratio and uses the geometric mean throughout, as described above.

Bootstrap Confidence Intervals

Confidence intervals for the geometric-mean speedup are constructed using a non-parametric bootstrap over events [22, 24]. The bootstrap operates on the per-event log-speedups $\ell_i = \log S_i$, where each S_i is the speedup for event i computed from the average timing over the $r = 5$ repeated runs:

$$\log S_i \;=\; \log \bar{t}^{(i)}_{\text{baseline}} - \log \bar{t}^{(i)}_{\text{optimised}}.$$

These log-speedups are computed once in additive form, which is mathematically equivalent to $\log(\bar{t}^{(i)}_{\text{baseline}}/\bar{t}^{(i)}_{\text{optimised}})$, and then passed to the bootstrap loop.

1. A pseudorandom number generator is initialised with a fixed seed (`seed` = 42) so the reported confidence interval bounds are reproducible across re-runs.
2. For each of $B = 10{,}000$ bootstrap iterations:
   - N events are sampled with replacement from $\{1, \ldots, N\}$, giving a resampled index set $\mathcal{I}_b$;
   - the mean log-speedup over those events is computed,

$$\mu_b \;=\; \frac{1}{N} \sum_{i \in \mathcal{I}_b} \log S_i,$$

     and transformed back to the speedup scale, $S_b = \exp(\mu_b)$.

3. The lower and upper bounds of the 95 % confidence interval are the 2.5th and 97.5th percentiles of the empirical bootstrap distribution $\{S_1, \ldots, S_B\}$:

$$\mathrm{CI}_{\mathrm{lower}} = Q_{0.025}(\{S_b\}), \qquad \mathrm{CI}_{\mathrm{upper}} = Q_{0.975}(\{S_b\}).$$

The closed interval $[\mathrm{CI}_{\mathrm{lower}}, \mathrm{CI}_{\mathrm{upper}}]$ is the reported 95 % confidence interval for the geometric-mean speedup. An optimised implementation is taken to be faster than the baseline when $\mathrm{CI}_{\mathrm{lower}} > 1.0$ and a regression when $\mathrm{CI}_{\mathrm{upper}} < 1.0$. Intervals containing 1.0 correspond to no detectable difference at the confidence level of the interval.

Assumptions and Limitations

Hardware and dataset dependence. All measurements are run on HEPP03 (Section 5.1.1) over the 50-event profiling dataset. Absolute timings depend on the microarchitecture, and the distribution of event-level runtimes depends on the occupancy and geometry of the sample. Reported speedups are therefore relative improvements under one fixed hardware and dataset configuration, not absolute performance numbers.

Statistical resolution. Three quantities affect the reported confidence intervals. The number of events N sets the underlying statistical precision, with the bootstrap interval width scaling approximately as $1/\sqrt{N}$. The number of repeated runs per event r reduces noise in each per-event timing estimate, but has a smaller effect once the within-event variance is already small, as is the case here. The number of bootstrap iterations B controls a different source of uncertainty: the numerical precision with which the bootstrap percentiles are estimated.

The bootstrap uses $B = 10{,}000$ resamples. This is large enough that the Monte-Carlo noise from the bootstrap procedure itself is negligible compared with the statistical uncertainty from the finite event sample, and the resulting percentile estimates are stable at the precision used for the reported confidence interval bounds. Increasing B further would therefore only refine the numerical percentile estimates, not improve the underlying experimental resolution. With $N = 50$ events and $r = 5$ repeated runs, the limiting factor is the amount of independent event information; resolving smaller effects would require additional events rather than additional bootstrap iterations.

Timing overhead. Manual timers introduce a small measurement overhead, applied in the same way to baseline and optimised configurations. Because the timed regions are coarse compared with the timer-call cost, this overhead is expected to be negligible relative to the measured runtimes. Any remaining additive overhead would tend to bias speedup ratios toward unity rather than create artificial improvements, since the measured ratio would approach $(t_{\mathrm{baseline}}+h)/(t_{\mathrm{optimised}}+h)$ for an additive overhead h.

Bootstrap independence. The bootstrap assumes independence between resampling units. The resampling unit here is the event. The absence of an execution-order trend in Section 5.2.1 supports treating events as approximately independent resampling units with respect to warm-up and drift effects. The r runs of a single event share the same input and are not treated as independent observations; they are averaged into a single per-event estimate before the bootstrap [31].

Aggregation loss. The reported geometric mean summarises a typical multiplicative improvement across events. Events with unusually large or small per-event speedups are compressed into this single number, so the aggregate result should

be interpreted together with the reported confidence interval and, where relevant, the per-event distributions shown in the results chapter.

5.3 Physics Validation Methodology

The aim of this work is to improve computational performance while either preserving reconstruction quality or trading it off against speed within pre-defined limits. This section describes how physics performance is validated for both categories of optimisation. Unlike execution time, the physics outputs are deterministic for identical inputs and configurations, so repeated execution of the same configuration does not produce statistical fluctuations in the validation metrics. Validation therefore reduces to direct comparison between the baseline and optimised implementations. Any observed difference is attributable to the implemented change, subject only to floating-point rounding and the finite representativeness of the validation dataset.

5.3.1 Optimisation Categories and Validation Strategy

Two categories of optimisation are considered:

- **Code-level optimisations** improve computational efficiency without intentionally altering the underlying algorithmic behaviour. Validation verifies that physics performance is preserved. Any residual differences are expected to be negligible and attributable to floating-point rounding or equivalent low-level numerical effects.
- **Algorithm-level modifications** intentionally change reconstruction behaviour to achieve improved performance or a different physics–throughput trade-off. Differences in physics output are expected and form part of the evaluation. Validation therefore focuses on quantifying and interpreting these changes in the context of the achieved computational speedup.

5.3.2 Evaluation Metrics

Physics performance is evaluated using the metrics defined in Section 2.5.2, computed by the ACTS validation framework under its standard reconstructability criteria and truth-matching association rules. This section specifies which metrics are reported and under which conditions.

Reported aggregates. Efficiency ε, fake ratio f, and duplicate ratio d are reported as event-population aggregates for every optimisation evaluated in this thesis, following the seed-level definitions given in Section 2.5.2.

Kinematic binning. To ensure sensitivity to localised effects, four quantities are also reported as differential distributions: the seeding efficiency ε, and the mean per-event counts of reconstructed, fake, and duplicated truth-matched objects, denoted $\langle n_{\text{reco}} \rangle$, $\langle n_{\text{fake}} \rangle$, and $\langle n_{\text{dup}} \rangle$. These quantities are plotted against the truth-particle kinematic variables transverse momentum p_T, pseudorapidity η, and azimuthal angle ϕ. Reporting the metrics differentially makes it possible to identify regions of phase space in which an optimisation disproportionately gains or loses performance, rather than only observing the population-averaged effect.

5.3.3 Code-Level Optimisation Validation

For each metric x, the absolute difference between the optimised and baseline implementations is computed as

$$\Delta x = x_{\text{opt}} - x_{\text{base}}. \tag{5.4}$$

Results are reported in tabular form alongside the baseline and optimised values.

A code-level optimisation is considered to preserve physics performance when the observed differences are negligible at the precision with which the metrics are reported and when no systematic deviation from the baseline is visible in the differential distributions. Because the validation outputs are deterministic for a fixed input sample, no statistical tests or confidence intervals are applied to repeated executions of the same configuration. The decision rests on direct numerical comparison between the baseline and optimised outputs.

5.3.4 Algorithm-Level Performance Evaluation

For algorithm-level modifications, absolute differences in the primary metrics are computed using the same definition as in the code-level case. To ensure consistent interpretation, a signed absolute change s_x is defined for each metric such that positive values indicate improved physics performance and negative values indicate degradation:

$$\begin{aligned} s_\varepsilon &= +\Delta\varepsilon && \text{(efficiency: higher is better)}, \\ s_f &= -\Delta f && \text{(fake ratio: lower is better)}, \\ s_d &= -\Delta d && \text{(duplicate ratio: lower is better)}. \end{aligned}$$

The signed changes s_ε, s_f, and s_d are reported in tabular form for every algorithm-level modification.

5.3.5 Interpretation

Since the physics outputs are deterministic for fixed inputs and configurations, differences between baseline and optimised configurations are not interpreted in terms of repeated-run statistical significance. Instead, each change is interpreted by its magnitude, direction, and consistency across the reported metrics and kinematic distributions. A change that improves or preserves efficiency, fake ratio, and duplicate ratio, without introducing a localised degradation in p_T, η, or ϕ, is considered a physics-preserving optimisation.

Optimisations that achieve a measurable speedup while preserving the physics metrics at the reported precision are adopted as the default configuration. Optimisations that introduce a non-negligible regression in one or more metrics are not adopted as the default, but are reported in the results chapters when their trade-off may still be relevant for use cases willing to exchange some physics quality for throughput.

5.3.6 Parameter Selection Methodology

Algorithm-level optimisations introduce parameters whose values affect both physics performance and computational cost. To select operating points, the seeding algorithm is executed on the physics-validation dataset under identical conditions across configurations, and both reconstruction metrics and wall-clock execution time are

recorded. This subsection describes the procedure used throughout Section 6.4; the parameters and selected values for each optimisation are reported in the corresponding results section.

The procedure is organised into four stages:

1. **Single-axis characterisation.** Each candidate parameter is varied in turn along its own axis, with all other parameters held at a shared anchor configuration corresponding to the runner defaults. These scans are run at a reduced event count of 5 events to keep the per-axis cost low.
2. **Derivation of joint-search intervals.** For each axis, the per-value response from stage 1 is inspected to identify the range over which the parameter has an observable effect. Saturated values, defined as values beyond which further change in the parameter produces no meaningful change in either physics performance or runtime, are excluded. The surviving values define the per-axis interval used in the joint search.
3. **Joint grid search at reduced event count.** The Cartesian product of the per-axis intervals is evaluated at 5 events, the same event count as the single-axis scans. The grid is intentionally coarse: its purpose is to rank configurations and identify promising operating points, not to deliver final metric values.
4. **Confirmation at full event count.** The top-ranked configurations under each of the selection rules defined below are re-evaluated on the full 50-event dataset. The resulting high-statistics metrics are the values to which the selection rules are ultimately applied, and they are the values reported in the results sections.

Single-axis characterisation

The first stage is a set of single-axis scans. In each scan, one parameter is varied across a plausible range while all other parameters are held fixed at the shared anchor configuration. These scans are not used to select final operating points directly. Their purpose is to identify which parameters have an observable effect, to locate the range over which each parameter is active, and to exclude values that have already saturated.

For each scanned value, three quantities are recorded: the physics metrics, the fake and duplicate ratios, and the wall-clock time per event. The corresponding scan figures are provided in Section B.3. Each figure shows efficiency, fake and duplicate ratios, and normalised wall-clock time as a function of the scanned parameter, with the runner default marked by a vertical dashed line.

The single-axis scans are useful for bounding the joint search space, but they do not expose interactions between parameters. A value that is optimal when all other parameters are fixed at their defaults need not remain optimal once another parameter is changed. The scans are therefore used only to define the candidate intervals for the joint grid search; final operating points are selected from the confirmed multi-parameter configurations described below.

Joint grid search

When several parameters interact, an operating point is chosen by a grid search. The grid intervals are derived from the single-axis characterisations. For each axis, the retained values are those that produced a meaningful response in the single-axis scan, typically a small number of values bracketing the useful range, with saturated values excluded as defined above. The Cartesian product of the per-axis intervals is evaluated at 5 events to keep the grid cost tractable.

At the propose stage, the top candidates under each selection rule are extracted from the 5-event grid. Because 5-event measurements are less representative than the 50-event confirmation measurements, the efficiency thresholds used to identify candidates at the propose stage are tightened by a small conservative margin. This reduces the chance that a configuration is promoted to the confirmation stage only because the reduced event sample overestimates its efficiency. The strict selection rules defined below are then applied without modification at the confirmation stage.

The candidates surviving the propose stage are re-evaluated on the full 50-event dataset, and the resulting set of configurations is reported alongside the cylindrical baseline.

Selection rules

Three operating points are reported for each joint optimisation, corresponding to three distinct trade-offs against the cylindrical baseline. Let $t(c)$ denote the per-event wall-clock time of configuration c, and let $\Delta\varepsilon(c)$, $\Delta f(c)$, and $\Delta d(c)$ denote the changes in efficiency, fake ratio, and duplicate ratio relative to the cylindrical baseline.

max_efficiency The configuration in the search space with the highest efficiency, regardless of timing. Ties on efficiency are broken in favour of the minimum time. This rule defines the physics-performance ceiling reachable within the search space.

best The fastest configuration that uses the cylindrical efficiency as a fixed working point while improving fake rejection and throughput; formally, the fastest configuration that does not regress on any primary metric relative to the cylindrical baseline:

$$best = \arg\min_{c} t(c) \quad \text{subject to} \quad \Delta\varepsilon(c) \geq 0,\ \Delta f(c) \leq 0,\ \Delta d(c) \leq 0.$$

Inequalities are non-strict, so a configuration that matches the cylindrical baseline on a metric is admissible; they are evaluated against exact zero so *best* is a strict no-regression rule. Ties on time are broken in favour of the highest efficiency. This is the default operating point unless stated otherwise: a drop-in replacement that does not degrade any metric.

fastest The fastest configuration above a hard efficiency floor $\varepsilon \geq 0.95$, with no constraint on f or d. The floor is set to permit trade-offs that the stricter *best* rule excludes while keeping efficiency within an acceptable range; the precise value is a working choice for this thesis and could be tightened or loosened without changing the structure of the rule. Ties on time are broken in favour of the highest efficiency. This rule gives a throughput-oriented operating point that permits movement in fake and duplicate ratio in exchange for speed.

The three rules are not guaranteed to select distinct configurations; when they coincide, the shared point is reported once. The full pipeline (configuration files for the shared anchor, the per-axis scans, the joint-search grid, the confirmation event count, and the selection rules; together with the runner scripts that execute each stage) lives in the project repository under `configs/parameter_optimization/`. The directory's `README.md` documents each file. The layout is designed so that the optimisation can be reproduced or extended to additional parameters by editing a single configuration file per axis.

Chapter 6

Results

This chapter presents performance results in five stages:

1. **Baseline comparison** (Section 6.1): head-to-head comparison of `Seeding` and `Seeding2`.
2. **Bottleneck analysis** (Section 6.2): profiling of `Seeding2` to identify the dominant hotspots.
3. **Code-level optimisations** (Section 6.3): implementation and impact of optimisations that preserve algorithmic behaviour.
4. **Algorithm-level optimisations** (Section 6.4): implementation and impact of optimisations that modify reconstruction behaviour.
5. **Seeding3** (Section 6.5): design and evaluation of a new seeding implementation incorporating the findings from the preceding stages.

6.1 Baseline Performance Comparison: `Seeding` vs. `Seeding2`

`Seeding2` is a refactored implementation of the ACTS seeding algorithm, developed within the ACTS collaboration prior to this work [34]. It preserves the reconstruction behaviour of the original `Seeding` implementation while reorganising the code along three structural axes:

Grid externalisation. Grid construction and management are separated from the seeding algorithm. Space points are pre-binned, and the core logic operates on abstract collections. This decouples data structures from algorithmic logic, allowing binning strategies to be exchanged or optimised independently of the seeding code.

Runtime dispatch. In `Seeding`, the grid type is fixed at compile time via a C++ template parameter, so each compiled binary is locked to one specific grid representation. `Seeding2` replaces this with runtime polymorphism: the grid type is chosen at run time through a virtual interface, a mechanism that lets the algorithm call grid methods without knowing the concrete grid type [34]. The same compiled binary now works with any grid implementation. The cost is a small per-call overhead on grid accesses, because each call goes through a function pointer rather than being inlined; this is the *indirect-dispatch* overhead.

Component modularisation. Large monolithic functions are decomposed into smaller, single-purpose components. This improves readability, testability, and profiling granularity, and enables targeted optimisation without major refactoring.

The merge-request description cites maintainability and extensibility as the motivation for the refactor [34]; on performance, the PR notes only that the refactored implementation "shows similar CPU performance", treating performance as a constraint to preserve rather than a target to improve. Whether the architectural changes have in fact produced measurable performance differences, and if so in which direction, is the subject of this section. `Seeding2` is the most thoroughly modernised version of the seeding algorithm in the current codebase, and is therefore also the foundation on which the code-level and algorithm-level optimisations of Sections 6.3 and 6.4 build.

Two analyses are presented. The first establishes the overall performance difference between the two implementations on both detector geometries. Speedup can come from two sources: doing less work, or doing the same work faster. The second analysis decomposes the measured speedup into these two contributions and shows that they contribute in different proportions on pixel and strip.

All measurements are performed with single-threaded execution on the 50-event profiling dataset described in Section 4.1, following the methodology outlined in Section 5.2.

Overall Performance

`Seeding2` achieves consistent and statistically significant speedups over `Seeding` across both detector geometries, as summarised in Table 6.1. On pixel data, `Seeding2` is approximately 1.11× faster, reducing per-event wall-clock time from 651 ms to 584 ms. On strip data, the speedup is approximately 1.05×, reducing per-event time from 195 ms to 185 ms.

The 95 % bootstrap confidence intervals on both speedups are narrower than ±0.005 ([1.112, 1.115] on pixel; [1.048, 1.054] on strip), indicating highly consistent performance across the five independent runs that underlie each measurement. This consistency confirms that the reported speedups reflect a stable property of the implementation rather than a favourable run.

Table 6.1: Overall performance comparison between `Seeding` and `Seeding2`, reported as time per event and throughput (events per second). Each configuration is run 5 times over the 50-event profiling dataset; the per-event time used in the comparison is the average across the 5 runs. Time per event and throughput are arithmetic means; speedup is the geometric mean of event-level ratios with 95 % bootstrap confidence intervals (10,000 iterations), see Section 5.2.2 for the aggregation rationale. Wall-clock and CPU times agreed to within 0.5 % across all configurations, confirming negligible I/O overhead and consistent single-threaded execution; only wall-clock measurements are reported.

Detector	**Time [ms/event]**		**Throughput [events/s]**		**Speedup [95% CI]**
	`Seeding`	`Seeding2`	`Seeding`	`Seeding2`	
Pixel	650.9 ± 32.0	584.3 ± 27.6	1.59 ± 0.08	1.77 ± 0.09	1.113 [1.112, 1.115]
Strip	194.8 ± 11.8	185.5 ± 11.4	5.40 ± 0.34	5.68 ± 0.37	1.051 [1.048, 1.054]

Decomposing the Speedup: Workload Reduction vs. Implementation Efficiency

The measured speedup can come from two sources: less work performed (fewer candidate pairs evaluated, fewer seeds produced), or the same work performed more efficiently

(better cache behaviour, tighter code paths, less overhead). The distinction matters for interpreting the speedup: workload reduction requires an algorithmic change and can affect the physics output, while implementation efficiency does not. Table 6.2 reports data-volume metrics at each pipeline stage for both implementations.

The two geometries tell qualitatively different stories.

On **pixel** data, the refactor left the grid mechanics essentially unchanged (total bins, occupied bins, empty-bin fraction, and mean space points per occupied bin all agree to the displayed precision) and produced a small reduction in the downstream workload: 2.2 % fewer bottom doublet candidates, 2.7 % fewer top doublet candidates, 2.0 % fewer triplets, and 1.0 % fewer output seeds. These small reductions are likely incidental consequences of minor changes in compatibility cuts or candidate filtering introduced during the refactor. Even taken together they would produce at most a $\sim 2\,\%$ speedup, well below the measured 11.3 %; the remaining ∼9 percentage points are attributable to implementation efficiency rather than reduced work.

Table 6.2: Workload comparison between `Seeding` and `Seeding2` for pixel and strip detector geometries. Workload metrics are deterministic given fixed input, so values are taken from a single 50-event run. Metrics are grouped by pipeline stage: grid construction, doublet finding, triplet finding, and seed output. The relative difference is (`Seeding2` − `Seeding`)/`Seeding`. The strip +4.9 % on total bins reflects an ACTS configuration asymmetry: `addStandardSeeding` honours a separate `rMaxGridConfig` for grid layout, while `GridTripletSeedingAlgorithm` takes a single `rMax` from the seed-finder configuration; for the ATLAS ITk strip detector these differ by 200 mm, producing a slightly larger outer radius for the `Seeding2` strip grid.

Stage	Metric	`Seeding`	`Seeding2`	Rel. diff.
Pixel detector				
Grid	Total bins	2,626	2,626	0.0%
	Occupied bins	2,600	2,600	0.0%
	Empty bins [%]	0.99	0.99	0.0%
	Mean SPs per occupied bin	71.26	71.26	0.0%
Doublets	Bottom candidates	1,802,982	1,763,590	−2.2%
	Top candidates	1,844,628	1,795,388	−2.7%
Triplets	Created triplets	142,999	140,198	−2.0%
Output	Total seeds	7,507	7,431	−1.0%
Strip detector				
Grid	Total bins	1,066	1,118	+4.9%
	Occupied bins	1,040	1,092	+5.0%
	Empty bins [%]	2.44	2.33	−4.7%
	Mean SPs per occupied bin	84.38	82.83	−1.8%
Doublets	Bottom candidates	907,357	880,403	−3.0%
	Top candidates	675,172	644,143	−4.6%
Triplets	Created triplets	13,290	12,575	−5.4%
Output	Total seeds	7,917	7,418	−6.3%

On **strip** data, the picture is different. The two implementations construct meaningfully different grids: `Seeding2` uses about 5 % more bins, with 5 % more of them occupied and slightly lower mean occupancy per bin (−1.8 %).

This traces to a configuration-surface difference rather than an algorithmic one: `addStandardSeeding` honours the separate `rMaxGridConfig` parameter while the newer `GridTripletSeedingAlgorithm` takes only a single `rMax` from the seed-finder configuration, producing a slightly larger strip grid for `Seeding2`.

The downstream workload is correspondingly lower in `Seeding2`: 3.0 % fewer bottom doublet candidates, 4.6 % fewer top doublet candidates, 5.4 % fewer triplets, and 6.3 % fewer output seeds. The grid difference plausibly contributes through the geometric cuts that depend on bin-resolved neighbour searches, though no attempt is made here to disentangle it quantitatively from other small refactoring effects. The speedup decomposes multiplicatively into a workload factor and an implementation-efficiency factor. The workload factor is the inverse of the triplet-count ratio, $1/(1-0.054) = 1.057$, since the triplet stage dominates the seeding runtime. The measured speedup factor is 1.051. The implementation-efficiency factor is therefore $1.051/1.057 \approx 0.994$, consistent with 1.000 within the bootstrap interval. The strip speedup is therefore workload-driven, with the workload reduction itself likely rooted in the upstream grid configuration.

Taken together, the pixel speedup isolates the implementation efficiency of the refactor almost cleanly, while the strip speedup reflects workload reduction with no detectable implementation-efficiency contribution. The stage-level decomposition presented next localises the pixel implementation gains within specific pipeline stages.

Stage-Level Efficiency: Waterfall Analysis

Figure 6.1 presents a waterfall decomposition of the total runtime change across eight functional stages, isolating where the implementation-efficiency gains arise. Per-event execution times in this decomposition include fine-grained timer instrumentation around each stage boundary. The instrumented totals agree with the uninstrumented per-event measurements of Table 6.1 to within 1.6 % across all four configurations, well below the per-stage contributions discussed below, so the stage-level attribution is read off directly as absolute contributions.

On **pixel** data, three inner-loop stages account for nearly the entire observed speedup: `MiddleBottoms` contributes −9.0 % of the `Seeding` baseline total (−58.9 ms per event), `TripletEvaluation` −3.1 % (−20.2 ms), and `MiddleTops` −1.9 % (−12.6 ms). Together they reduce per-event time by approximately 92 ms. The remaining stages are net slower in aggregate (+2.9 % of the baseline total, ∼18 ms per event), accounting for the gap between this gross gain and the ∼ 74 ms net speedup. The setup phase (the `Seeding(other)` residual, `GridSetup`, and `createSeedsGroup`) contributes +1.4 %, +1.0 %, and +0.5 % respectively, reflecting the cost of `Seeding2`'s more granular timer structure and heavier grid bookkeeping. `SeedFiltering` runs marginally faster (−0.4 %).

On **strip** data, the same inner-loop stages dominate the improvement but with a different ordering and substantially smaller absolute magnitudes: `TripletEvaluation` now leads at −4.9 % of the baseline total (−9.6 ms), `MiddleBottoms` −4.6 % (−9.2 ms), and `MiddleTops` only −0.7 % (−1.3 ms). Together these three give ∼ 20 ms of gross stage-level savings, but the net reduction is only ∼ 12 ms: setup adds back ∼ 8 ms (the `Seeding(other)` residual +1.2 %, `GridSetup` +1.8 %, `createSeedsGroup` +0.5 %, and `SeedFiltering` +0.5 %, which is net-faster on pixel; together +4.0 % of the baseline total). The setup overhead is proportionally much larger on strip than on pixel (eating ∼ 40 % of the gross inner-loop gain versus ∼ 20 % on pixel), because the setup costs are similar in absolute terms on the two geometries while the strip inner-loop gains are much smaller.

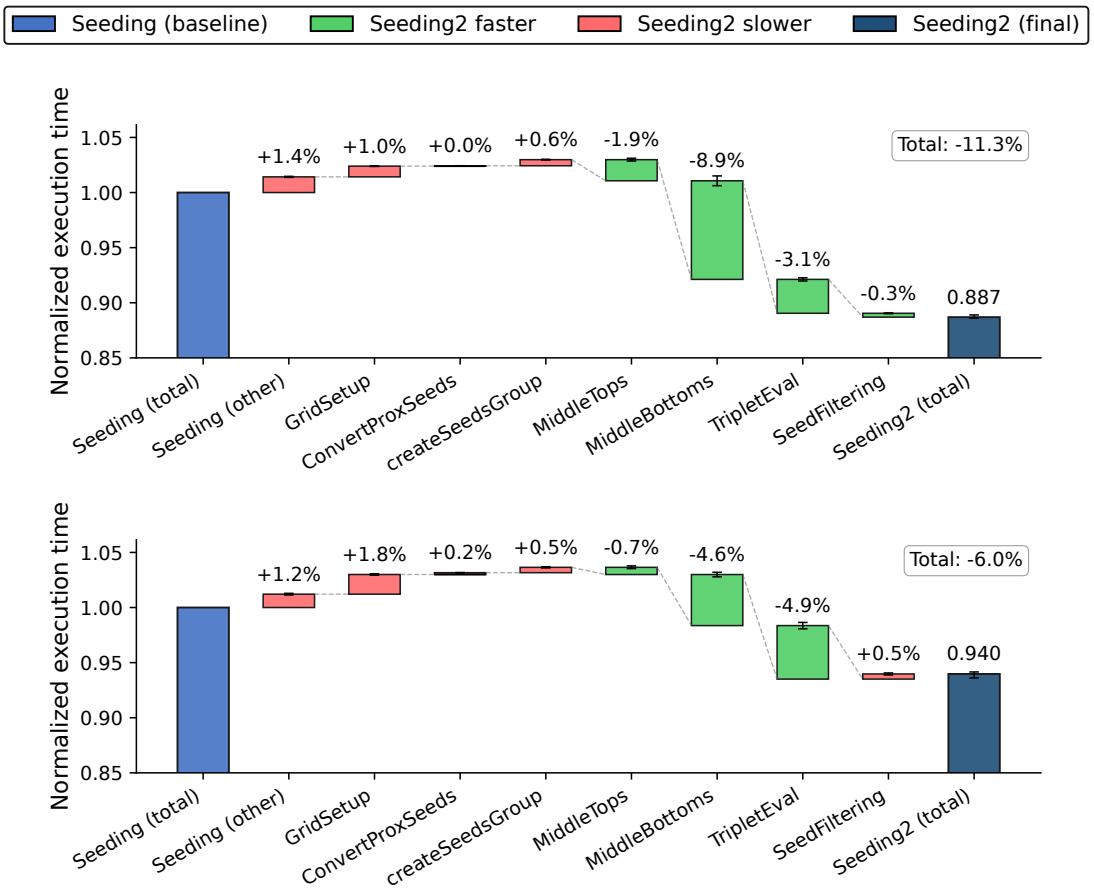


Figure 6.1: Waterfall decomposition of per-event execution time for `Seeding` and `Seeding2` on pixel (top) and strip (bottom) data. The leftmost (blue) bar is the normalised `Seeding` baseline; intermediate bars show the per-stage contribution to the overall change, with green for stages where `Seeding2` is faster and red for stages where it is slower. The rightmost (dark blue) bar is the final normalised `Seeding2` time, and the total relative change is annotated in the upper-right corner of each panel. Error bars show 95 % bootstrap confidence intervals (10,000 iterations). Stage-level instrumentation adds at most 1.6 % overhead to the uninstrumented per-event totals of Table 6.1; the per-stage values therefore represent absolute contributions to the total.

Across both geometries, the largest gains concentrate in the stages that dominate baseline runtime: `MiddleBottoms`, `MiddleTops`, and `TripletEvaluation` together account for approximately 95 % of total wall time on pixel and 92 % on strip in `Seeding`. The improvement therefore comes from where time is actually spent, not from isolated changes to minor code paths.

This pattern is consistent with the modularisation hypothesis from earlier in this section: splitting large functions into smaller ones gives the compiler more aggressive inlining opportunities and better register allocation in the inner loops, reducing per-doublet and per-triplet overhead in the stages that dominate. The trade-off is the cost of `Seeding2`'s more granular timer structure and its setup-phase work, which offsets a meaningful fraction of the inner-loop gains on both geometries, around a fifth on pixel and roughly half on strip. The strip speedup is smaller than the pixel speedup not primarily because of this overhead, however, but because the strip inner-loop gains are themselves much smaller in absolute terms. Combined with the configuration-driven workload reduction discussed earlier, the strip case shows essentially zero implementation-efficiency contribution from the refactor.

Function-Level Breakdown

Tables 6.3 and 6.4 list the 10 most time-consuming functions for each implementation on pixel and strip data, showing how the refactor redistributes work.

Table 6.3: Top 10 functions by self-time for `Seeding` and `Seeding2` on pixel data. Flat % = function self-time (fraction of total runtime spent in the function's own instructions); Cum % = cumulative time including callees (self plus all descendants in the call graph).

#	Pixel Detector	Flat %	Cum %
Seeding			
	`SeedFinder::getCompatibleDoublets`	57.10	70.24
	`SeedFinder::filterCandidates`	7.95	24.46
	`std::vector::operator[] (inline)`	4.52	4.53
	`std::__unguarded_linear_insert (inline)`	3.40	5.09
	`std::__invoke_impl (inline)`	2.74	4.41
	`std::__unguarded_partition (inline)`	2.48	5.87
	`__gnu_cxx::__normal_iterator::operator- (inline)`	1.75	1.75
	`SpacePointData::z (inline)`	1.58	1.58
	`{lambda(float, float, float)#1}::operator() (inline)`	1.53	1.53
	`std::vector::emplace_back (partial-inline)`	1.17	2.17
Seeding2			
	`createDoubletsImpl (inline)`	56.83	63.44
	`createPixelTripletTopCandidates (inline)`	6.29	8.30
	`std::__unguarded_linear_insert (inline)`	5.53	5.53
	`Impl::createDoublets (partial-inline)`	3.96	66.34
	`std::__unguarded_partition (inline)`	3.17	4.80
	`{lambda(Acts::DoubletsForMiddleSp::IndexAndCotTheta con...`	1.22	1.22
	`std::vector::emplace_back (inline)`	1.22	1.44
	`BroadTripletSeedFilter::filterTripletTopCandidates`	1.17	1.66
	`DoubletsForMiddleSp::emplace_back`	0.93	2.04
	`f64xsubf128`	0.91	0.91

In `Seeding`, `getCompatibleDoublets` dominates (57.1 % pixel, 45.5 % strip of flat self-time) with `filterCandidates` adding 8.0 % and 15.2 %. Together these two routines hold the bulk of the doublet search and candidate filtering logic, accounting for $\sim 65\,\%$ of total pixel runtime and $\sim 61\,\%$ of total strip runtime.

In Seeding2, createDoubletsImpl retains a comparable share of flat self-time (56.8 % pixel, 45.1 % strip): the innermost doublet evaluation logic is structurally similar between implementations. Two changes in the surrounding profile stand out.

First, filterCandidates no longer appears at the top level. Its work is now split between createPixelTripletTopCandidates (6.3 % pixel, 10.3 % strip) and BroadTripletSeedFilter::filterTripletTopCandidates (1.2 % pixel, 2.4 % strip), separating top-candidate construction from triplet-level filtering. On pixel the total filtering cost is broadly preserved (8.0 % in Seeding versus 7.5 % in Seeding2); on strip it drops by about 2.5 percentage points (15.2 % to 12.7 %), consistent with the smaller strip workload feeding fewer candidates into the filter. A new Impl::createDoublets entry (4.0 % pixel, 4.3 % strip) also appears, from the dispatch layer introduced by the doublet-construction modularisation.

Table 6.4: Top 10 functions by self-time for Seeding and Seeding2 on strip data. Column definitions as in Table 6.3.

#	Strip Detector	Flat %	Cum %
Seeding			
	SeedFinder::getCompatibleDoublets	45.52	57.47
	SeedFinder::filterCandidates	15.15	34.60
	std::__unguarded_linear_insert (inline)	4.64	6.75
	std::vector::operator[] (inline)	4.01	4.03
	std::__invoke_impl (inline)	2.42	3.34
	std::__unguarded_partition (inline)	2.03	4.74
	__gnu_cxx::__normal_iterator::operator- (inline)	1.71	1.71
	SpacePointData::z (inline)	1.46	1.46
	{lambda(float, float, float)#1}::operator() (inline)	1.38	1.38
	LinCircle::LinCircle (inline)	1.30	1.30
Seeding2			
	createDoubletsImpl (inline)	45.14	52.24
	createPixelTripletTopCandidates (inline)	10.31	12.78
	std::__unguarded_linear_insert (inline)	6.17	6.17
	Impl::createDoublets (partial-inline)	4.29	55.69
	operator() (inline)	2.77	3.90
	BroadTripletSeedFilter::filterTripletTopCandidates	2.42	2.98
	std::__unguarded_partition (inline)	2.42	3.41
	std::vector::emplace_back (inline)	1.09	1.46
	Impl::createTripletTopCandidates	1.07	13.85
	DoubletsForMiddleSp::emplace_back	1.07	1.91

Second, several low-level access functions prominent in Seeding drop out of Seeding2's top-10 list entirely. SpacePointData::z (1.6 %/1.5 % in Seeding) and std::vector::operator[] (4.5 %/4.0 %) are no longer visible as named hotspots. std::vector::emplace_back appears in Seeding2 at 1.2 %/1.1 %, along with a DoubletsForMiddleSp::emplace_back entry at 0.9 %/1.1 %. The disappearance of the accessor entries is consistent with more aggressive inlining of small accessor functions by the compiler, the work still happens but is now attributed to the calling routine rather than to the accessor itself, and the appearance of emplace_back entries points to a shift in the dominant access pattern from indexed reads of an existing container toward in-place construction of new entries.

The STL sort helpers (__unguarded_linear_insert and __unguarded_partition) take a larger share of total runtime in Seeding2: on pixel their combined flat share rises

from 5.9 % to 8.7 %, and on strip from 6.7 % to 8.6 %. Whether their absolute cost has changed is harder to read off the relative numbers alone, but the relative rise makes sorting a more attractive optimisation target than it was in `Seeding`, as the doublet stages no longer dominate the picture. This target is addressed in Section 6.3.3.

Full `gperftools` call-tree dumps for both implementations on both geometries are provided in Sections B.1 and B.2. They corroborate the flat-time redistributions discussed here and additionally document the parent/child relationships between functions.

Flamegraphs

Figure 6.2 shows the CPU flamegraphs for `Seeding` and `Seeding2` on pixel data. Three structural changes are visible in the comparison (marked ①–③), corroborated by the function-level profiles already discussed.

First, the dominant plateau (①) is essentially the same width but sits one frame deeper. In `Seeding`, `getCompatibleDoublets` sits directly beneath `createSeedsForGroup`. In `Seeding2`, `createDoubletsImpl` has comparable flat time (56.8 % vs. 57.1 % for `getCompatibleDoublets`, Table 6.3) but is reached through additional anonymous-namespace frames introduced by the modular structure. The call tree is deeper, but the extra frames carry essentially no work, the dominant share still sits in the innermost function.

Second, the small `filterCandidates` tower (②) on the left of the `Seeding` panel is replaced by named structure attributable to `createPixelTripletTopCandidates` and `filterTripletTopCandidates` in `Seeding2`. The visible tower in `Seeding` represents only one of the call paths through `filterCandidates`; the function-level breakdown (Table 6.3) sums these to 8.0 %. In `Seeding2` this aggregate is split between `createPixelTripletTopCandidates` (6.3 %) and `filterTripletTopCandidates` (1.2 %). The work has moved, not disappeared.

Third, the sort cluster (③) on the right of the profile is proportionally more prominent in `Seeding2` than in `Seeding`. The combined flat share of the STL sort helpers (`__unguarded_linear_insert` and `__unguarded_partition`) rises from 5.9 % to 8.7 % on pixel, partly because the doublet stages no longer dominate the total runtime so that any remaining work takes a larger share of what is left.

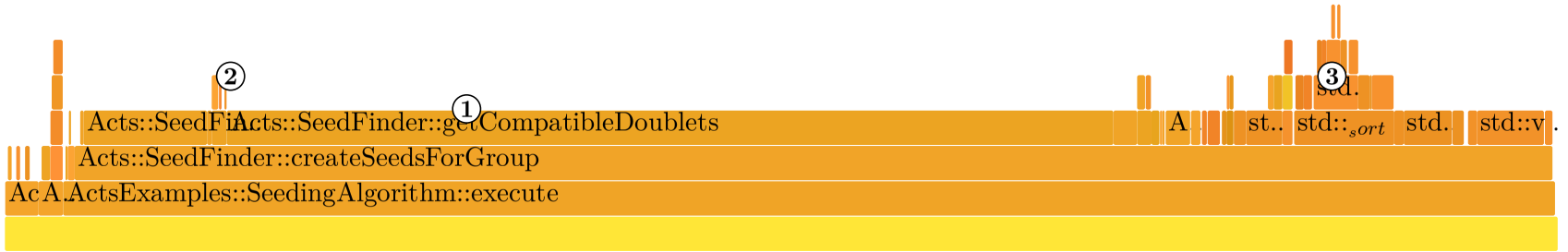
The strip flamegraphs (Sections B.1 and B.2) show the same three features.

Performance Scaling with Space-Point Multiplicity

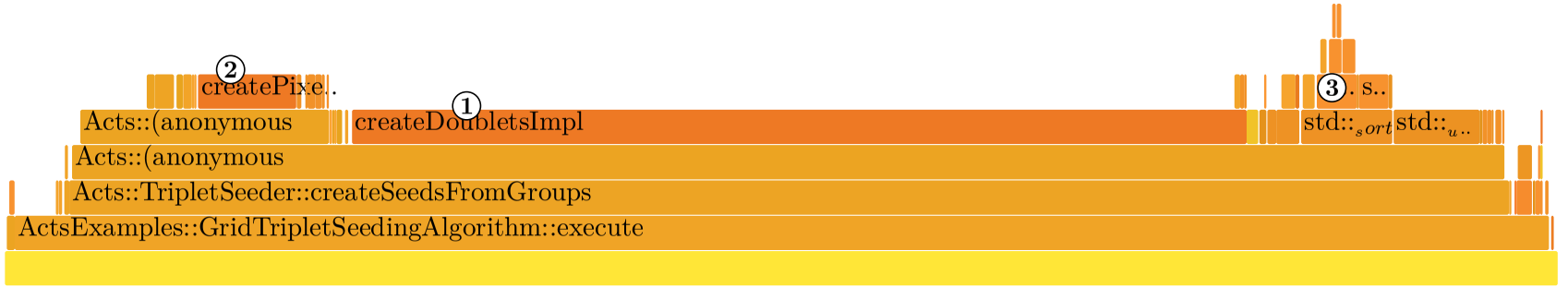
The grid-based algorithm design of Section 2.4.1 expects approximately linear scaling of execution time with space-point multiplicity N_{SP}, provided mean bin occupancy remains bounded. Figure 6.3 shows per-event execution time as a function of N_{SP}, used here to verify the regime and to check whether the speedup generalises beyond the profiling-dataset occupancy range.

On **pixel**, the slope drops from 5.085×10^{-3} to 4.436×10^{-3} ms per space point, a 12.8 % reduction in marginal cost. Within the measured range of $N_{\mathrm{SP}} \in [196{,}141,\ 289{,}892]$, the `Seeding2` speedup runs from $\sim$ 39 ms ($\sim$ 9.5 %) at the lowest multiplicity to $\sim$ 100 ms ($\sim$ 11.3 %) at the highest. The relative speedup grows modestly with N_{SP}, consistent with a slope-driven improvement: a per-space-point cost reduction compounds with the number of space points processed.

On **strip**, the slope difference is much smaller: 3.832×10^{-3} vs. 3.963×10^{-3} ms per space point, a 3.3 % reduction. Within the measured range of $N_{\mathrm{SP}} \in [68{,}480,\ 114{,}546]$,



(a) `Seeding` (baseline).



(b) `Seeding2`.

Figure 6.2: CPU flamegraphs for `Seeding` (top) and `Seeding2` (bottom) on pixel data. Frame width represents the fraction of on-CPU samples; colour encodes relative time spent, with redder frames running longer. Three structural changes are marked and discussed in the surrounding text: ① the dominant doublet-finding plateau, ② the filter region (top-candidate construction and triplet filtering), and ③ the sort cluster (STL sort helpers). Strip-data flamegraphs are shown in Sections B.1 and B.2.

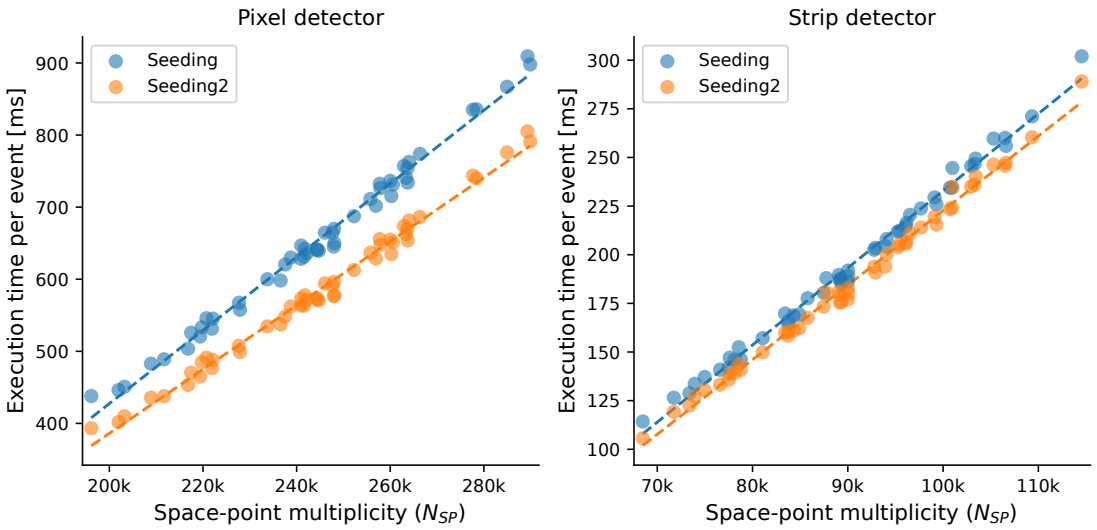


Figure 6.3: Per-event execution time as a function of space-point multiplicity (N_{SP}) for `Seeding` and `Seeding2` on pixel (left) and strip (right) data. Colour encodes implementation: blue for `Seeding`, orange for `Seeding2`. Each point is the per-event mean across the 5 runs. Dashed lines show linear fits, with the fitted equations in the legend; full coefficients are in Table 6.5.

the absolute speedup grows from ~ 6 to ~ 12 ms but the relative speedup shrinks from $\sim 5.7\,\%$ to $\sim 4.2\,\%$ as N_{SP} increases. The decreasing relative speedup with multiplicity is consistent with the workload-driven nature of the strip result: the small slope reduction is a smaller fraction of total time at high multiplicity than at low.

Linear models $t = a\,N_{\mathrm{SP}} + b$ were fitted to each of the four implementation–detector combinations; coefficients are in Table 6.5. The fits describe the data well within the observed range, with $R^2 \geq 0.989$ in all four cases.

Table 6.5: Linear fit coefficients ($t = a\,N_{\mathrm{SP}} + b$) for per-event execution time as a function of space-point multiplicity. The fitted intercepts b have no physical meaning: they arise from extrapolating a model fitted on a restricted, non-zero occupancy range back to $N_{\mathrm{SP}} = 0$, which is outside the data. The model is valid only within the sampled N_{SP} interval.

Detector	Algorithm	a	b	R^2
Pixel	`Seeding`	5.085×10^{-3}	-5.895×10^{2}	0.9898
	`Seeding2`	4.436×10^{-3}	-5.010×10^{2}	0.9894
Strip	`Seeding`	3.963×10^{-3}	-1.634×10^{2}	0.9930
	`Seeding2`	3.832×10^{-3}	-1.606×10^{2}	0.9931

Summary

`Seeding2` achieves measurable speedups of 1.11× on pixel and 1.05× on strip over `Seeding`. Decomposing these speedups shows the dominant source differs between the two geometries.

On pixel, `Seeding2` processes a marginally smaller workload (within a few percent across all pipeline stages) yet runs 11 % faster. Workload reduction alone caps at $\sim 2\,\%$, so the remaining ~ 9 percentage points of speedup are attributable to implementation efficiency. The stage-level decomposition localises these gains to `MiddleBottoms` ($-9.0\,\%$

of baseline total), `TripletEvaluation` (−3.1 %), and `MiddleTops` (−1.9 %). The gross inner-loop savings of ∼ 92 ms are partly absorbed by ∼ 19 ms of setup-phase work (`Seeding(other)` residual, `GridSetup`, `createSeedsGroup`), giving the observed ∼ 73 ms net. The pattern is consistent with the modularisation hypothesis that smaller, well-scoped functions enable more effective compiler optimisation of the innermost loops.

On strip, the picture is different. Workload reduction (6.3 % fewer seeds, 5.4 % fewer triplets) is comparable to the measured speedup (5.1 %), leaving no detectable implementation-efficiency contribution: the residual factor 1.051/1.057 ≈ 0.994 is consistent with 1.000 within the bootstrap interval. The same three inner-loop stages save ∼ 20 ms in aggregate, but ∼ 8 ms is absorbed by setup overhead, leaving the smaller ∼ 12 ms net speedup. The strip workload reduction itself traces to a configuration-surface asymmetry between the `addStandardSeeding` and `GridTripletSeedingAlgorithm` interfaces (`rMaxGridConfig` vs `rMax`) rather than to algorithmic improvements in `Seeding2`. Notably, the setup overhead consumes a larger fraction of gross inner-loop gains on strip (∼ 40 %) than on pixel (∼ 20 %), contributing to the smaller proportional net speedup.

The function-level analysis additionally shows that the pre-existing STL sort helpers (`__unguarded_linear_insert` + `__unguarded_partition`) rise from 5.9 % to 8.7 % of flat share on pixel, identifying sorting as a concrete target for subsequent optimisation. The scaling analysis shows that `Seeding2`'s marginal per-space-point cost is 12.8 % lower than `Seeding`'s on pixel; on strip, the slope is 3.3 % lower, consistent with the workload-driven nature of the strip result.

With the architectural baseline characterised, Section 6.2 profiles `Seeding2` in detail to identify the remaining computational bottlenecks below the level of architectural restructuring, which the code-level optimisations of Section 6.3 target.

6.2 Detailed Profiling of Seeding2

This section profiles `Seeding2` to find its dominant bottlenecks. Phase 1 uses stage-level timing and function-level sampling to find where runtime is spent; Phase 2 uses `gperftools` line profiling to annotate each hotspot at the source level. The findings motivate the code-level optimisations of Section 6.3.

6.2.1 Phase 1: Hotspot Identification

Stage-level timing

Table 6.6 reports the exclusive per-stage wall-clock timing over 50 events for both detector geometries. Exclusive timing attributes each stage's cost only to instructions executed directly within that scope; time in child stages is counted under those children, not under the parent.

The per-stage timings include the same fine-grained timer instrumentation used in the waterfall decomposition of Section 6.1, whose overhead was found to be at most 1.6 % of the uninstrumented per-event totals. The fractional shares below can therefore be read as absolute contributions.

Three inner-loop hotspots — `MiddleTops`, `MiddleBottoms`, and `TripletEvaluation` — account for 91.4 % of total runtime on pixel and 87.5 % on strip. Each is invoked ∼ 10^4–10^5 times per event, so any meaningful speedup must originate from within the doublet- and triplet-construction loop nest. Top-level dispatch stages (grid setup, proxy-space-point construction, and the outer-loop overhead of `createSeedsGroup` and

`Seeding` itself) together contribute under 9 % on pixel and under 13 % on strip, offering limited room for improvement.

Table 6.6: Exclusive per-stage timing breakdown for the pixel and strip seeding algorithms, reported as a fraction of total `Seeding2` wall time, as absolute mean wall-clock time per event, and as the mean number of invocations per processed event. Values are mean ± SD over 5 runs on 50 events. The ≈ symbol marks invocation counts that vary with event-level track multiplicity; exact counts indicate stages invoked once per event or constant across events. Stages are grouped into top-level dispatch (per-event setup and outer-loop control) and inner-loop hotspots (invoked many times per event within `createSeedsGroup`). Absolute times include the same fine-grained timer instrumentation used in Section 6.1; that section measured the overhead to be at most 1.6 % of the uninstrumented per-event totals reported in Table 6.1.

	Pixel detector			**Strip detector**		
Stage	**Excl. time** [%]	**Time/evt** [ms]	**Calls/evt**	**Excl. time** [%]	**Time/evt** [ms]	**Calls/evt**
Top-level dispatch						
Seeding	3.5 ± 0.0	20.6 ± 0.2	1	3.7 ± 0.1	6.91 ± 0.09	1
GridSetup	3.5 ± 0.0	20.3 ± 0.1	1	5.4 ± 0.1	10.0 ± 0.1	1
ConvertProxSeeds	0.1 ± 0.0	0.61 ± 0.02	1	0.3 ± 0.0	0.47 ± 0.02	1
createSeedsGroup	1.0 ± 0.0	5.56 ± 0.03	1800	1.5 ± 0.0	2.83 ± 0.01	924
Inner-loop hotspots						
MiddleTops	34.4 ± 0.0	200.4 ± 0.5	≈ 103,523	26.3 ± 0.2	49.1 ± 0.1	≈ 49,385
MiddleBottoms	32.2 ± 0.0	187.8 ± 0.4	≈ 82,434	29.7 ± 0.6	55.6 ± 1.7	≈ 43,771
TripletEval	24.8 ± 0.0	144.5 ± 0.3	≈ 77,850	31.5 ± 0.3	58.9 ± 0.1	≈ 42,017
SeedFiltering	0.6 ± 0.0	3.53 ± 0.02	≈ 77,850	1.7 ± 0.0	3.12 ± 0.05	≈ 42,017

Within the three hotspots, the relative ordering of `MiddleTops` and `TripletEvaluation` reverses between geometries: 34.4 % vs. 24.8 % on pixel and 26.3 % vs. 31.5 % on strip. The reversal is consistent with strip data producing more candidate triplets per middle space point than pixel: the ratio of `TripletEvaluation` invocations to `MiddleTops` invocations (Table 6.6, Calls/evt column) is ∼ 85 % on strip (42,017/49,385) and ∼ 75 % on pixel (77,850/103,523). Despite this reordering, the same three stages dominate in both configurations, so optimisations targeting these stages should benefit both geometries to the extent that they affect geometry-independent inner-loop work.

Function-level sampling

Tables 6.7 and 6.8 list the most time-consuming functions in `Seeding2` for pixel and strip data, obtained at 100 Hz sampling over five independent runs. The analysis addresses three questions: how concentrated the dominant cost is, which secondary routines are the next optimisation targets, and whether the profile reveals specific overhead patterns suitable for code-level intervention.

`createDoubletsImpl` dominates flat time on both detectors (56.83 % pixel, 45.14 % strip). The flat-to-cumulative gap is 6.6 pp on pixel (56.83 % to 63.44 %) and 7.1 pp on strip (45.14 % to 52.24 %), so roughly 90 % of `createDoubletsImpl`'s total cost is in its body rather than in routines it calls. A partial-inline remainder `(anonymous namespace)::Impl::createDoublets` adds a further 3.96 % on pixel and 4.29 % on

Table 6.7: Most time-consuming functions for `Seeding2` on pixel data (`gperftools`, $N = 5$ runs, 100 Hz sampling). Flat % = function self-time; Cum % = cumulative time including callees (self plus all descendants in the call graph). Function names are abbreviated.

#	Pixel Seeding2 (top 20)	Flat %	Cum %
1	`createDoubletsImpl (inline)`	56.83	63.44
2	`createPixelTripletTopCandidates (inline)`	6.29	8.30
3	`std::__unguarded_linear_insert (inline)`	5.53	5.53
4	`Impl::createDoublets (partial-inline)`	3.96	66.34
5	`std::__unguarded_partition (inline)`	3.17	4.80
6	`{lambda(Acts::DoubletsForMiddleSp::IndexAndCotTheta con...`	1.22	1.22
7	`std::vector::emplace_back (inline)`	1.22	1.44
8	`BroadTripletSeedFilter::filterTripletTopCandidates`	1.17	1.66
9	`DoubletsForMiddleSp::emplace_back`	0.93	2.04
10	`f64xsubf128`	0.91	0.91
11	`{lambda(void const*, Acts::SpacePointProxy2 const&, Act...`	0.91	1.15
12	`operator() (partial-inline)`	0.74	1.28
13	`std::vector::push_back (inline)`	0.58	1.54
14	`std::__insertion_sort (inline)`	0.57	3.04
15	`std::__introsort_loop`	0.53	6.23
16	`(anonymous namespace)::createSeedsFromGroupsImpl`	0.52	92.53
17	`Impl::createTripletTopCandidates`	0.51	8.81
18	`DoubletsForMiddleSp::sortByCotTheta`	0.36	13.57
19	`std::__invoke_impl (inline)`	0.33	2.81
20	`(anonymous namespace)::createAndFilterTriplets`	0.31	10.92

strip, so doublet creation in total accounts for roughly 61 % of pixel runtime and 49 % of strip runtime. Speedups targeting this stage must therefore focus on the body of `createDoubletsImpl` itself, not on the routines it invokes.

After doublet creation, the two largest optimisation targets are sorting and `createPixelTripletTopCandidates`. The STL insertion-sort internals (`__unguarded_linear_insert`, `__unguarded_partition`, `__insertion_sort`) together account for ~ 9.3 % of flat time on both pixel and strip, making sorting the second-largest aggregate hotspot after doublet creation. The corresponding `sortByCotTheta` projection lambda appears as a named entry (1.22 % pixel, 0.69 % strip), suggesting the projection is at least partly retained as a separate symbol rather than being fully inlined into the sort algorithm. `createPixelTripletTopCandidates` is the third-largest contributor at 6.29 % pixel and 10.31 % strip; its higher share on strip is consistent with the waterfall finding that triplet-related cost weighs more heavily under strip occupancy.

Two further entries point to concrete overhead patterns. The delegate-target lambda for the `Acts::Delegate`-dispatched per-space-point predicate (`itkFastTrackingSPselect`) appears at 0.91 % on pixel but does not reach the strip top-20, reflecting non-inlined delegate dispatch that becomes visible only on pixel, where the predicate is called more often. Container-growth allocations appear as `std::vector::emplace_back` and `DoubletsForMiddleSp::emplace_back` combined (2.15 % pixel, 2.16 % strip), consistent with repeated allocations during container growth and amenable to preallocation strategies.

The full call-tree structure produced by `gperftools`, documenting the parent/child relationships between the functions listed here, is provided in Section B.2. Together

Table 6.8: Most time-consuming functions for `Seeding2` on strip data (`gperftools`, $N = 5$ runs, 100 Hz sampling). Flat % = function self-time; Cum % = cumulative time including callees (self plus all descendants in the call graph). Function names are abbreviated.

#	Strip Seeding2 (top 20)	Flat %	Cum %
1	`createDoubletsImpl (inline)`	45.14	52.24
2	`createPixelTripletTopCandidates (inline)`	10.31	12.78
3	`std::__unguarded_linear_insert (inline)`	6.17	6.17
4	`Impl::createDoublets (partial-inline)`	4.29	55.69
5	`operator() (inline)`	2.77	3.90
6	`BroadTripletSeedFilter::filterTripletTopCandidates`	2.42	2.98
7	`std::__unguarded_partition (inline)`	2.42	3.41
8	`std::vector::emplace_back (inline)`	1.09	1.46
9	`Impl::createTripletTopCandidates`	1.07	13.85
10	`DoubletsForMiddleSp::emplace_back`	1.07	1.91
11	`(anonymous namespace)::createSeedsFromGroupsImpl`	0.86	89.99
12	`f64xsubf128`	0.81	0.81
13	`std::__insertion_sort (inline)`	0.73	4.80
14	`{lambda(Acts::DoubletsForMiddleSp::IndexAndCotTheta con...`	0.69	0.69
15	`SpacePointContainer2::varianceR (inline)`	0.64	0.64
16	`std::vector::operator[] (inline)`	0.64	0.64
17	`Proxy::u (inline)`	0.58	0.62
18	`_int_malloc`	0.51	0.56
19	`std::__do_uninit_copy (inline)`	0.47	0.58
20	`std::vector::push_back (inline)`	0.47	1.44

with the stage-level timing, these findings identify four classes of optimisation target for the line-level analysis in Phase 2: the body of `createDoubletsImpl`, the body of `createPixelTripletTopCandidates`, the indirect-call overhead of delegate dispatch, and the allocation pressure from container growth. Sorting also contributes significantly to the profile but is treated separately in Section 6.3.3.

Flamegraphs

Figure 6.4 shows the flamegraph for `Seeding2` on pixel data. Each horizontal bar represents a stack frame; width encodes the fraction of sampled CPU time spent in that frame and its callees, and colour is hash-based to help distinguish adjacent frames. Three sibling branches under `createSeedsFromGroups` are marked ①–③: the dominant `createDoubletsImpl` plateau ①; `sortByCotTheta` as a narrow tower ②; and the triplet-construction path leading to `createPixelTripletTopCandidates` ③. The three branches correspond to the three main phases of seed-forming within the inner loop — doublet creation, sorting, and triplet construction — at the same call-tree level rather than nested inside one another.

The strip-data flamegraph (Section B.2) shows the same three-branch structure.

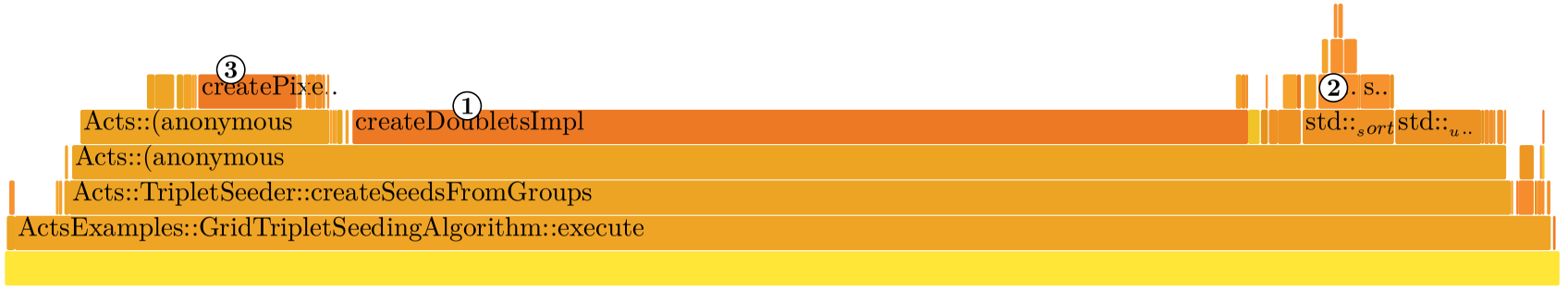


Figure 6.4: CPU flamegraph for `Seeding2` on pixel data. Frame width represents the fraction of on-CPU samples; colour is hash-based and carries no quantitative meaning. Three sibling branches discussed in the surrounding text are marked: ① the `createDoubletsImpl` plateau, ② the `sortByCotTheta` tower, and ③ the `createPixelTripletTopCandidates` branch.

6.2.2 Phase 2: Source-level Investigation

Phase 2 applies `gperftools` line-level annotation to the four in-scope hotspots identified in Phase 1: `createDoubletsImpl`, `createPixelTripletTopCandidates`, the indirect-call overhead of delegate dispatch, and the allocation pressure from container growth.

Before turning to these, one further profile contribution warrants comment. The insertion-sort internals (`__unguarded_linear_insert`, `__unguarded_partition`, `__insertion_sort`) together account for approximately 9.3 % of flat time on both pixel and strip, making them a large profile contribution. They are not addressed here for two related reasons. First, the hottest entries live in `libstdc++`'s introsort implementation rather than in ACTS code, so modifying them is outside the scope of a thesis on the ACTS seeding algorithm. Second, `std::sort` is close to optimal as a general-purpose sort: the observed cost is a function of the volume of doublets flowing into the sort, so its reduction is an algorithmic question about how sort and filter stages are structured rather than a line-level one. The algorithm-level optimisation in Section 6.3.3 addresses sorting through this structural route.

The line-level annotation attributes sampled CPU time to individual source lines, which lets us identify which expressions in each hotspot carry the most cost and distinguish between cost that is intrinsic to a single expression and cost that is spread across an entire loop body. Each hotspot subsection below reports the key findings from this annotation, which source-level constructs account for the largest shares of per-line sampled time and discusses what kind of change might reduce that cost. All per-line times underlying this discussion are totals over 5 runs of 50 events each (250 events total) at 100 Hz sampling. The specific implementations and their measured effects are presented in Section 6.3. The full annotation listings for the four hotspots on both pixel and strip are reproduced in Section B.2; the discussion below draws on those listings without reproducing the per-line cost figures in the text.

H1 — `createDoubletsImpl`

Line annotation shows that cost within `createDoubletsImpl` is distributed across the entire inner loop body on both detector geometries, consistent with a large candidate set where individual checks are cheap but are executed many times.

The single most expensive line within `createDoubletsImpl` is the collision-region origin check, which tests whether the estimated z-origin of a candidate pair falls within the allowed collision region. The implementation avoids division by comparing $z_M \Delta r - r_M \Delta z$ against the collision-region bounds multiplied by Δr, mathematically equivalent to the direct z_{origin} check:

$$z_M \Delta r - r_M \Delta z \;\in\; [\,\text{collisionRegionMin} \cdot \Delta r,\; \text{collisionRegionMax} \cdot \Delta r\,]. \tag{6.1}$$

The origin check is preceded by the Δz range check `outsideRangeCheck(deltaZ, deltaZMin, deltaZMax)`, which filters some candidates before the origin check is reached, but the origin check still dominates per-line cost on both pixel and strip.

On the pixel path, three further predicates are visible in the annotation but do not appear among the strip hotspots: the interaction-point cut `std::abs(rM * yNewFrame) > impactMax * xNewFrame`, the helix-diameter cut `(bCoef * bCoef) * minHelixDiameter2 > 1 + aCoef * aCoef`, and the `Delegate`-dispatched `experimentCuts` call. The strip path skips these via the negative branch of `interactionPointCut`.

Each candidate evaluates a sequence of predicates and is rejected at one of them; because the rejection points vary across candidates, the cost is spread across the whole predicate sequence rather than concentrated at any single expression.

Optimisation hypothesis. The doublet-search inner loop iterates over space-point pairs, and its iteration count scales quadratically with the number of space points per bin; any per-iteration cost is therefore amplified substantially in total runtime. Because the cost in H1 is spread across several predicates in the loop body rather than concentrated in any single one, optimisations that target these predicates (hoisting loop-invariant sub-expressions, replacing arithmetic with cheaper algebraically equivalent forms, or merging adjacent conditional checks) are multiplied by the full iteration count. This is implemented as **O1** and evaluated in Section 6.3.1.

H2 — `createPixelTripletTopCandidates`

The compatibility cut in `createPixelTripletTopCandidates` tests whether the difference in $\cot\theta$ between a bottom doublet and a candidate top doublet is consistent with their combined track-direction uncertainty, rejecting top candidates whose $\cot\theta$ differs too much from the bottom doublet's value. Line annotation shows an ordering inefficiency in this loop.

The space-point error term `error2`, formed by combining the per-doublet errors with a correlation term scaled by $1 + \cot\theta_B \cot\theta_T$, is computed before the squared $\cot\theta$ difference `deltaCotTheta2 = (cotThetaB - cotThetaT)` ^{2} that is the quantity actually compared against `error2` in the cut `deltaCotTheta2 > error2 + scatteringInRegion2`. On both detector geometries, the `error2` computation dominates the per-line samples in this loop, while `deltaCotTheta2` itself consumes negligible per-line cost. Every top doublet pays the full `error2` cost before the cut is evaluated, including the candidates that `deltaCotTheta2` alone would have been sufficient to reject. The subsequent early-exit branch `if (cotThetaB < cotThetaT) break;` operates downstream of the cut and does not change this.

Optimisation hypothesis. When an inner loop evaluates multiple rejection criteria in a fixed order, candidates that will ultimately be rejected by a later criterion still pay the cost of all earlier criteria. A cheap but discriminating criterion evaluated after an expensive one is a missed early-exit opportunity, which is the pattern H2 presents: `deltaCotTheta2` is a subtraction and a square, while `error2` is a multi-line error-propagation calculation. Reordering the checks so that the cheaper criterion executes first is expected to reduce the per-iteration cost for rejected candidates. Whether the reordering pays off overall depends on how discriminating `deltaCotTheta2` is in isolation: if it rejects enough candidates that the saved `error2` computations outweigh the cost of evaluating it on every candidate, the reordering wins. This is tested empirically as **O2** in Section 6.3.2.

H3 — Indirect call overhead

Annotation reveals two indirect-call sites worth attention, with different status as optimisation targets.

The first is the `Acts::Delegate` path used for the per-space-point pre-filter `itkFastTrackingSPselect`: the predicate body appears in its own profile frame (0.91 % of pixel runtime, negligible on strip), with non-trivial flat time on pixel

despite the predicate's triviality. This is consistent with the runtime-resolved function pointer underlying `Acts::Delegate` preventing the compiler from inlining the call at the observed sampling point. However, the analogous case for the `experimentCuts` delegate inside the doublet-finding inner loop is already addressed at compile time in `Seeding2`: `if constexpr (experimentCuts)` inside the 16-way template instantiation of `DoubletSeedFinder::Impl` eliminates the call site entirely when no experiment cuts are connected, or retains it as a single direct call otherwise. The pre-filter case has not received the same treatment, and remains a profile observation rather than a remaining optimisation target.

The second is the sort comparator path. Annotation of `DoubletsForMiddleSp::sortByCotTheta` shows the projection lambda `[](const IndexAndCotTheta& item) { return item.cotTheta; }` appearing as a distinct named routine in the profile, separately from the sort algorithm's internal frames. The wrapper `sortByCotTheta` itself carries negligible flat time on both geometries. The lambda's appearance as its own profile entry indicates that each sort comparison incurs an out-of-line call to extract the sort key. This is a statically-typed lambda whose inlining is defeated by `std::ranges::sort`'s projection interface, which adds an indirection layer between the sort algorithm and the key-extraction function.

Optimisation hypothesis. Small callables invoked many times within a hot path are standard targets for compiler inlining: inlining eliminates the per-call overhead and can expose the call site to further optimisations such as register allocation, constant propagation, and vectorisation. The sort projection lambda is a clear candidate, since it executes many times per event and is the only remaining indirect-call site that template dispatch cannot address: `std::ranges::sort` is a library function whose interface introduces the indirection. Replacing the projection-based sort call with one that accepts the comparator as a direct template argument is expected to allow the compiler to inline the comparison body into the sort loop, eliminating the per-comparison call overhead. This approach is implemented as **O3** and evaluated in Section 6.3.3.

H4 — Allocation pressure

Line annotation identifies two independent sources of *heap allocation pressure* in the seeding pipeline: frequent allocator activity on the hot path that strains the allocator and triggers copying overhead from container reallocations.

Grid bin vectors. Annotation of `CylindricalSpacePointGrid2::insert` shows the per-bin `bin.push_back` carrying measurable flat time on both geometries. Each bin vector starts empty and grows on demand; when a bin's capacity is exceeded, its contents are copied to a new buffer. Because this happens across all occupied bins for every event, the allocation cost accumulates over the full insertion loop. The dominant cost within `insert` is the `binIndex` arithmetic rather than the `push_back` itself, but the latter is the relevant cost for H4 since it reflects allocator activity rather than coordinate computation.

Doublet container vectors. The doublet container `DoubletsForMiddleSp` uses a struct-of-arrays layout with five independent `std::vector` members (`m_spacePoints`, `m_cotTheta`, `m_er_iDeltaR`, `m_uv`, `m_xy`). Annotation of `emplace_back` shows that each accepted doublet triggers five separate `push_back` calls, with sampled cost spread across

the five member assignments rather than concentrated in any one. The combined contribution of `std::vector::emplace_back` and `DoubletsForMiddleSp::emplace_back` is 2.15 % on pixel and 2.16 % on strip, roughly symmetric in relative terms, reflecting that allocation overhead scales similarly with the rest of the seeding work on both geometries despite their different absolute doublet volumes. Each of the five vectors grows independently, so their reallocation schedules stagger across doublets, compounding allocator activity rather than amortising it.

Optimisation hypothesis. Both patterns share the same structural cause: vectors that start empty and are not sized before the hot path, forcing the allocator to intervene repeatedly during normal operation. Successive `push_back` calls on an initially empty `std::vector` produce a logarithmic number of reallocations, each copying all existing elements to a new buffer, for $O(N)$ total copying work. When the approximate final size is known or bounded in advance, a single upfront `reserve` replaces these intermediate allocations with one and eliminates the copy overhead. Here, both quantities are knowable before the hot path runs: the number of grid bins is fixed by configuration, so an average per-bin occupancy can be precomputed from N_{SP}; and the number of candidate top space points entering the doublet loop provides an upper bound on the doublet container's final size. This approach is implemented as **O4** and evaluated in Section 6.3.4.

6.2.3 Summary of Optimisation Targets

Table 6.9: Summary of identified hotspots and their Phase 2 observations. Flat-time fractions are pixel values relative to total `Seeding2` runtime; strip values are quoted alongside where they differ meaningfully. Per-line annotation figures are reproduced in Section B.2.

Hotspot	Function	Flat	Phase 2 observation
H1	`createDoubletsImpl`	56.83 %	Cost is distributed across the inner loop body; the collision-region origin check is the single largest line-level contributor on both pixel and strip.
H2	`createPixelTripletTopCandidates`	6.29 %	The expensive `error2` computation is performed before the cheap `deltaCotTheta2` cut, so candidates that could be rejected cheaply still pay the full error cost first.
H3	Sort projection lambda	1.22 %	The lambda (0.69 % on strip) executes in its own profile frame, confirming missed inlining at the sort comparator call site.
H4	`insert` / `emplace_back`	—	Grid bin vectors grow on demand during insertion. Each accepted doublet triggers five separate `push_back` calls (one per member of the container's struct-of-arrays), with reallocations occurring at each vector's growth points.

The four hotspots identified in Phase 1 and investigated in Phase 2 each admit a distinct type of targeted change: H1 points toward inner-loop restructuring, H2 toward early-exit reordering, H3 toward indirection removal at the sort comparator path, and H4 toward preallocation. Table 6.9 summarises the key observations from each hotspot. The corresponding code-level changes, O1 through O4, are implemented and evaluated in Section 6.3. Pixel and strip observations are qualitatively similar, with differences noted in the H1 and H3 discussions above where they affect the optimisation argument.

6.3 Code-Level Optimisations

This section presents four optimisations addressing the hotspots identified in Section 6.2. Each subsection follows the same structure: a short reminder of the hotspot examined, a description of the implemented change, a quantification of the timing impact, and verification that physics outputs are unchanged. Wall-clock timing uses stage timers over 50 events with bootstrap 95 % confidence intervals. Callgrind instruction counts provide a deterministic, machine-independent corroboration of the wall-clock results. For O4, Heaptrack allocation metrics are additionally reported as the primary evidence. Output equivalence is verified by comparing physics performance metrics (efficiency, fake ratio, duplicate ratio) against the `Baseline`.

6.3.1 O1 — `createDoubletsImpl` Inner-Loop Reformulation

H1 (Section 6.2.2) identified the collision-region origin check and the Δz range check as the dominant per-line costs in `createDoubletsImpl`'s inner loop. Both checks constrain the same underlying quantity $\cot\theta = \Delta z/\Delta r$ but are evaluated as two independent guarded branches per candidate pair. O1 replaces these with a single pre-computed bound on $\cot\theta$, eliminating one branch and two arithmetic operations per iteration.

Implementation

The collision-region condition,

$$c_{\min} \cdot \Delta r \;\le\; z_M \Delta r - r_M \Delta z \;\le\; c_{\max} \cdot \Delta r, \tag{6.2}$$

can be rearranged (dividing by $r_M \Delta r > 0$) into a bound on $\cot\theta$:

$$\frac{z_M - c_{\max}}{r_M} \;\le\; \cot\theta \;\le\; \frac{z_M - c_{\min}}{r_M}. \tag{6.3}$$

Intersecting this with the explicit $[-\cot\theta_{\max}, \cot\theta_{\max}]$ constraint yields a single tighter bound that subsumes both original checks. The two effective limits,

$$\cot\theta_{\text{eff,min}} = \max\Big(-\cot\theta_{\max},\ \tfrac{z_M - c_{\max}}{r_M}\Big), \tag{6.4}$$

$$\cot\theta_{\text{eff,max}} = \min\Big(\ \ \cot\theta_{\max},\ \tfrac{z_M - c_{\min}}{r_M}\Big), \tag{6.5}$$

depend only on middle-SP coordinates and configuration constants, so they are computed once per middle SP, outside the candidate loop. Inside the loop the two original branches are replaced by the single test

$$\Delta z \;\notin\; \big[\cot\theta_{\text{eff,min}} \cdot \Delta r,\ \cot\theta_{\text{eff,max}} \cdot \Delta r\big], \tag{6.6}$$

which avoids the division and retains the integer-`OR` branch idiom already used throughout `createDoubletsImpl` for branch prediction. The intermediate quantity `zOriginTimesDeltaR` and its two associated multiplications are eliminated entirely.

Per candidate iteration, the change amounts to two comparisons reduced to one, one subtraction and one multiplication removed, and one branch eliminated.

Performance

Table 6.10 reports wall-clock timing, Callgrind instruction counts, and branch mispredictions for the `Baseline` and O1. On pixel data, O1 produces a 1.041× speedup in total seeding time per event (1.040–1.042 at 95 %); on strip data, 1.035× (1.034–1.036). Callgrind confirms the underlying work reduction: total instruction count decreases by 8.94 % on pixel and 9.05 % on strip, with the local `createDoubletsImpl` instruction count showing a larger reduction of 12.84 % and 14.94 % respectively. The larger local reduction reflects the fact that the change operates entirely within `createDoubletsImpl`; the proportionally smaller global reduction shows that stages outside the modified function are unaffected.

The local wall-clock measurements do not resolve a corresponding reduction (1.009× pixel, 0.994× strip, both consistent with no change within their confidence intervals). This is a local-timer overhead effect: the per-iteration cost of `createDoubletsImpl` is small enough that the wall-clock timer's per-call overhead dominates the local measurement, so work reductions inside the function body do not translate cleanly into the local timer reading. The Callgrind instruction counts, which are not subject to timer overhead, are the authoritative local-scope measurement; the global wall-clock timer, which sits above the inner loop and accumulates over many iterations, confirms that the work reduction translates into visible runtime savings at the algorithm level.

Table 6.10: Wall-clock timing, Callgrind instruction counts, and branch mispredictions for `Baseline` and O1. Wall-clock values are mean per-event times with bootstrap 95 % CIs; speedups are geometric means of event-level ratios. Callgrind counts are totals over the profiling dataset; relative change is (O1 − Baseline)/Baseline.

Detector	Metric	Baseline	O1	Speedup / Rel. change
Pixel detector				
	Global time/event [ms]	581.76 ± 27.57	558.76 ± 26.26	1.041 [1.040, 1.042]
	`createDoubletsImpl` [ns]	226.09 ± 3.68	224.11 ± 0.76	1.009 [0.998, 1.026]
	Global instructions (Ir)	10.233×10^9	9.318×10^9	−8.94%
	`createDoubletsImpl` (Ir)	7.084×10^9	6.175×10^9	−12.84%
	Global branch mispred. (Bcm)	82.97×10^6	81.74×10^6	−1.48%
	`createDoubletsImpl` mispred. (Bcm)	49.26×10^6	48.79×10^6	−0.95%
Strip detector				
	Global time/event [ms]	185.56 ± 11.43	179.14 ± 10.84	1.035 [1.034, 1.036]
	`createDoubletsImpl` [ns]	116.79 ± 0.28	117.48 ± 0.75	0.994 [0.989, 0.999]
	Global instructions (Ir)	3.615×10^9	3.288×10^9	−9.05%
	`createDoubletsImpl` (Ir)	2.193×10^9	1.865×10^9	−14.94%
	Global branch mispred. (Bcm)	27.71×10^6	27.98×10^6	+0.97%
	`createDoubletsImpl` mispred. (Bcm)	15.14×10^6	15.70×10^6	+3.71%

The branch-misprediction pattern differs between geometries. On pixel, both global and local mispredictions decrease (−1.48 % global, −0.95 % local), consistent with the elimination of one branch per inner-loop iteration. On strip, the global misprediction count increases marginally (+0.97 %) and the local count rises by 3.71 % despite the same branch being eliminated. This suggests that the resulting compiler-generated code on strip contains branches that are individually less predictable than those in the `Baseline`, even though the source-level branch count is lower. Despite this different branch-level response, both detectors deliver comparable wall-clock speedups: the $\sim 9\,\%$ Ir reduction more than offsets the small strip misprediction increase.

Physics validation

All three metrics (efficiency, fake ratio, and duplicate ratio) are identical between `Baseline` and O1 to five decimal places; see Table 6.11. The two formulations of the $\cot\theta$ bound are mathematically equivalent, so the optimisation is exact at the precision of double-precision floating-point arithmetic.

Table 6.11: Physics validation for O1 (mean over 50 events). All differences are consistent with zero at five-decimal precision, confirming output equivalence with `Baseline`.

Metric	Baseline	O1	Δ
Efficiency	0.97350	0.97350	+0.00000
Fake ratio	0.93536	0.93536	+0.00000
Duplicate ratio	0.94241	0.94241	+0.00000

6.3.2 O2 — Early-Exit in `createPixelTripletTopCandidates`

H2 (Section 6.2.2) identified a producer-consumer ordering problem in the compatibility-cut loop of `createPixelTripletTopCandidates`: the expensive `error2` computation is performed before the cheap `deltaCotTheta2` check, even though `deltaCotTheta2` provides a one-sided bound that can reject candidates without ever needing `error2`. O2 reorders the checks so that `deltaCotTheta2` is evaluated first, and `error2` is computed only for candidates that survive the cheap pre-filter. Although the function is named `createPixelTripletTopCandidates`, the same implementation is invoked on the strip path; the reordering therefore applies to both detectors.

Implementation

Because `error2` ≥ 0 by construction, the compatibility condition

$$\delta \cot^2\theta \;>\; \texttt{error2} \;+\; \sigma^2_{\text{scatter}} \tag{6.7}$$

cannot be satisfied unless

$$\delta \cot^2\theta \;>\; \sigma^2_{\text{scatter}}. \tag{6.8}$$

This cheaper inequality only involves quantities already available at the top of the loop body: cotThetaB, which is fixed per bottom doublet, and cotThetaT, which is one load. It is therefore evaluated first, and only candidates that pass the cheap pre-filter go on to the full error2 computation:

```
const float deltaCotTheta  = cotThetaB - cotThetaT;
const float deltaCotTheta2 = deltaCotTheta * deltaCotTheta;
if (deltaCotTheta2 > scatteringInRegion2) {
    const float error2 = topDoublet.er() + erB
                       + 2 * (cotThetaAvg2 * varianceRM + varianceZM)
                           * iDeltaRB * topDoublet.iDeltaR();
    if (deltaCotTheta2 > error2 + scatteringInRegion2) {
        ...
        continue;
    }
}
```

For candidates that pass the pre-filter via the fast path ($\delta \cot^2\theta \leq \sigma^2_{\text{scatter}}$), `error2` is deferred to the second compatibility cut further down the loop body, where it is needed for the p_T-scaled scattering term. The set of accepted triplet candidates is unchanged.

Performance

Table 6.12 reports wall-clock timing, Callgrind instruction counts, and branch mispredictions for `Baseline` and O2. The optimisation produces a clear local speedup on both detectors: the targeted function `createPixelTripletTopCandidates` runs 1.079× faster per call on pixel (1.072–1.087) and 1.120× faster on strip (1.114–1.126). The local wall-clock timer resolves these speedups cleanly, in contrast to O1's local timer, because the per-call cost of `createPixelTripletTopCandidates` ($\sim$ 33 ns baseline) is large enough relative to the per-call timer overhead that the speedup is visible.

Callgrind's view of the same function is more subtle. On strip, the local instruction count decreases marginally (−0.60 %), consistent with the early-exit pattern eliminating `error2`-computation work for candidates rejected by the cheap pre-filter. On pixel, the local instruction count actually *increases* by 2.02 % even though the wall-clock cost falls by 7.9 % per call. The pixel wall-clock improvement is therefore not explained by instruction-count elimination, and likely reflects a microarchitectural effect of the rearranged loop body, such as better instruction-level parallelism, register usage, or memory-access overlap. Identifying the specific mechanism would require performance-counter measurements beyond the Callgrind and wall-clock scope of this work; the wall-clock measurement is consistent across runs (CI 1.072–1.087), so the effect is real, but the explanation remains an empirical observation rather than a predicted result.

At the global scope the effect is substantially smaller: 1.005× on pixel and 1.018× on strip. This is expected. `createPixelTripletTopCandidates` contributes 6.3 % of total `Seeding2` runtime on pixel and 10.3 % on strip (see Tables 6.7 and 6.8), so any optimisation here is bounded by the function's share of total runtime. A $\sim$ 8 % local speedup on a 6 % component predicts a global speedup of $\sim$ 0.5 %, which matches the pixel result. The strip global speedup of 1.8 % somewhat exceeds the $\sim$ 1.2 % prediction from 12 % × 10 %, suggesting the optimisation may also reduce downstream work or that the local-times-share decomposition slightly underestimates its reach. The smaller global effect is a function of the optimisation target's share of runtime, not a shortcoming of the optimisation itself.

Branch mispredictions increase on both detectors at the local scope (+3.93 % pixel, +10.79 % strip), despite the intent of the early-exit pattern to avoid the expensive `error2` computation on rejected candidates. The `deltaCotTheta2` pre-filter rejects roughly half the candidates by design, giving a more balanced branch-taken/not-taken ratio than the original less-frequently-triggered cut; branches with balanced ratios are harder for the

branch predictor to model than highly skewed ones, so the misprediction count rises despite the source-level branch elimination. At the global scope these local increases are small enough that pixel global Bcm even drops slightly (−0.60 %). The pattern mirrors that observed for O1 on strip: branch-elimination at the source level does not always produce a clean misprediction reduction at the microarchitectural level, but the wall-clock speedup absorbs the local misprediction cost because it operates on a path that is short relative to total runtime.

Table 6.12: Wall-clock timing, Callgrind instruction counts, and branch mispredictions for Baseline and O2. Wall-clock values are mean per-event times with bootstrap 95 % CIs; speedups are geometric means of event-level ratios. Callgrind counts are totals over the profiling dataset; relative change is (O2 − Baseline)/Baseline. `createPixelTripletTopCandidates` is invoked on both pixel and strip paths, so the optimisation applies to both detectors.

Detector	Metric	Baseline	O2	Speedup / Rel. change
Pixel detector				
	Global time/event [ms]	581.76 ± 27.57	578.86 ± 27.26	1.005 [1.005, 1.005]
	`createPixelTripletTopCandidates` [ns]	33.54 ± 0.16	31.08 ± 0.10	1.079 [1.072, 1.087]
	Global instructions (Ir)	10.233×10^9	10.249×10^9	+0.15%
	`createPixelTripletTopCandidates` (Ir)	709.41×10^6	723.75×10^6	+2.02%
	Global branch mispred. (Bcm)	82.97×10^6	82.47×10^6	−0.60%
	`createPixelTripletTopCandidates` mispred. (Bcm)	5.54×10^6	5.76×10^6	+3.93%
Strip detector				
	Global time/event [ms]	185.56 ± 11.43	182.22 ± 11.00	1.018 [1.017, 1.019]
	`createPixelTripletTopCandidates` [ns]	35.95 ± 0.14	32.11 ± 0.21	1.120 [1.114, 1.126]
	Global instructions (Ir)	3.615×10^9	3.613×10^9	−0.05%
	`createPixelTripletTopCandidates` (Ir)	382.86×10^6	380.57×10^6	−0.60%
	Global branch mispred. (Bcm)	27.71×10^6	27.88×10^6	+0.61%
	`createPixelTripletTopCandidates` mispred. (Bcm)	3.14×10^6	3.48×10^6	+10.79%

Physics validation

All three metrics (efficiency, fake ratio, and duplicate ratio) are identical between `Baseline` and O2 to five decimal places; see Table 6.13. O2 reorders the evaluation of two existing compatibility conditions but does not change which candidates are accepted, so the set of produced triplets is identical to the `Baseline`.

Table 6.13: Physics validation for O2 (mean over 50 events). All differences are consistent with zero at five-decimal precision, confirming output equivalence with `Baseline`.

Metric	Baseline	O2	Δ
Efficiency	0.97350	0.97350	+0.00000
Fake ratio	0.93536	0.93536	+0.00000
Duplicate ratio	0.94241	0.94241	+0.00000

6.3.3 O3 — Removing Sort-Path Indirection for Inlining

H3 (Section 6.2.2) identified two independent sources of missed inlining in `Seeding2`: the `Acts::Delegate` path and the `sortByCotTheta` projection lambda. The `Delegate` case is already addressed at compile time in `Seeding2`: `if constexpr (experimentCuts)` inside the 16-way template instantiation of `DoubletSeedFinder::Impl` eliminates the call site entirely when no experiment cuts are connected, and retains it as a single direct call otherwise. O3 therefore focuses on the remaining sort-path indirection, which is a runtime issue that template dispatch cannot solve.

Implementation

In the `Baseline`, doublets are sorted by $\cot\theta$ through a method that wraps `std::ranges::sort` with a projection on the `cotTheta` member of `IndexAndCotTheta`:

```
std::ranges::sort(range, comp, &IndexAndCotTheta::cotTheta);
```

The projection interface introduces an indirection layer the compiler does not see through here, so the projection callable appears as a separate named routine in the profile rather than being inlined into the comparator.

O3 adds a new template method `sortByCotThetaDirect` to `DoubletsForMiddleSp` that accepts the comparator as a direct template argument:

```
template <typename Comparator>
void sortByCotThetaDirect(
    const IndexRange& range,
    std::vector<IndexAndCotTheta>& indexAndCotTheta,
    Comparator&& comp) const {
  // ... populate indexAndCotTheta ...
  std::sort(indexAndCotTheta.begin(), indexAndCotTheta.end(),
            std::forward<Comparator>(comp));
}
```

Accepting the comparator as a direct template argument instead of routing it through `std::ranges::sort`'s projection mechanism allows the compiler to instantiate a dedicated sort specialisation with the comparator body fully inlined into the sort loop. A file-scope `constexpr` lambda is defined once in `TripletSeeder.cpp` and passed at the two call sites that benefit from inlining:

```
static constexpr auto kCotThetaComp =
    [](const auto& a, const auto& b) {
        return a.cotTheta < b.cotTheta;
    };

cache.bottomDoublets.sortByCotThetaDirect(
    {0, cache.bottomDoublets.size()},
    cache.sortedBottoms, kCotThetaComp);
cache.topDoublets.sortByCotThetaDirect(
    {0, cache.topDoublets.size()},
    cache.sortedTops, kCotThetaComp);
```

The original `sortByCotTheta` method is retained unchanged for call sites that do not require inlining; O3 is an additive change rather than a replacement.

Performance

Table 6.14 reports wall-clock timing, Callgrind instruction counts, and branch mispredictions for `Baseline` and O3. The optimisation produces a substantial effect inside the sort cluster: local instruction count decreases by 29.03 % on pixel and 22.41 % on strip, and local branch mispredictions decrease by 46.44 % on pixel and 31.89 % on strip. Both changes are consistent with inlining having taken effect: the compiler now sees a contiguous sort loop body rather than a call into an opaque projection, which eliminates the per-comparison dispatch overhead and produces a more predictable branch layout.

The misprediction reduction is the cleanest branch-prediction improvement observed across the four code-level optimisations. Unlike O1 and O2, where a source-level branch change did not translate into a reliable misprediction decrease, the inlining performed by O3 gives the compiler enough local context to restructure the branch layout itself. The local misprediction reduction exceeds the local instruction-count reduction in relative terms on both detectors (46 % vs. 29 % on pixel, 32 % vs. 22 % on strip), confirming that inlining benefits branch prediction beyond the simple arithmetic of "fewer instructions executed means fewer branches executed".

Table 6.14: Wall-clock timing, Callgrind instruction counts, and branch mispredictions for `Baseline` and O3. Wall-clock values are mean per-event times with bootstrap 95 % CIs; speedups are geometric means of event-level ratios. Callgrind counts are totals over the profiling dataset; relative change is (O3 − Baseline)/Baseline. The local scope is `sortByCotTheta`, aggregating the comparator, the `std::sort` template instantiations, and the vector operations invoked during the sort.

Detector	Metric	Baseline	O3	Speedup / Rel. change
Pixel detector				
	Global time/event [ms]	581.76 ± 27.57	585.44 ± 27.65	0.994 [0.993, 0.994]
	Global instructions (Ir)	10.233×10^9	10.220×10^9	−0.13%
	`sortByCotTheta` (Ir)	614.93×10^6	436.43×10^6	−29.03%
	Global branch mispred. (Bcm)	82.97×10^6	82.52×10^6	−0.54%
	`sortByCotTheta` mispred. (Bcm)	20.96×10^6	11.22×10^6	−46.44%
Strip detector				
	Global time/event [ms]	185.56 ± 11.43	183.98 ± 11.22	1.009 [1.008, 1.009]
	Global instructions (Ir)	3.615×10^9	3.611×10^9	−0.11%
	`sortByCotTheta` (Ir)	256.10×10^6	198.71×10^6	−22.41%
	Global branch mispred. (Bcm)	27.71×10^6	27.85×10^6	+0.51%
	`sortByCotTheta` mispred. (Bcm)	7.22×10^6	4.92×10^6	−31.89%

At the global scope the wall-clock effect is small and differs in direction between detectors. On pixel, O3 produces a marginal slowdown (0.994×, CI 0.993–0.994); on strip, a small speedup (1.009×, CI 0.978–1.030). Both are within 1 % of unity.

The global Callgrind picture explains why the large local reductions do not propagate cleanly. Global Ir falls by only 0.13 % (-1.3×10^7) on pixel and 0.11 % (-4×10^6) on strip, well below what the local 29 % (-1.78×10^8) and 22 % (-5.74×10^7) reductions would predict if the savings were genuinely eliminated work. Roughly 7 % of the local reduction reaches the global Ir total on both detectors; the other 93 % is an attribution effect. The new `sortByCotThetaDirect` template, designed to expose the sort loop to

inlining, is in fact inlined at its call sites in `createSeedsFromGroupsImpl`, so instructions previously named under `sortByCotTheta` are still being executed, they are now counted under the caller's scope rather than under the sort cluster. The same pattern explains the large local Bcm reductions ($-46\,\%$ pixel, $-32\,\%$ strip) accompanied by near-zero global Bcm change.

The Phase 1 analysis identified the sort cluster only as a secondary optimisation target, and the global wall-clock outcome is consistent with that expectation. The substantial local reductions confirm that the inlining mechanism worked as designed, the compiler can now see through the sort comparator and restructure the branch layout, but most of the saved "sort-cluster" work has been re-attributed to the surrounding scope rather than eliminated outright. What genuine net work the inlining removes is the $\sim$ 13–4 $\times\, 10^6$ Ir difference visible at the global scope, an order of magnitude too small to drive a meaningful wall-clock speedup.

Physics validation

All three metrics (efficiency, fake ratio, and duplicate ratio) are identical between `Baseline` and O3 to five decimal places; see Table 6.15. O3 substitutes one sort implementation for another with the same ordering semantics, so the sequence of doublets produced is identical to the `Baseline` and downstream triplet construction is unaffected.

Table 6.15: Physics validation for O3 (mean over 50 events). All differences are consistent with zero at five-decimal precision, confirming output equivalence with `Baseline`.

Metric	Baseline	O3	Δ
Efficiency	0.97350	0.97350	+0.00000
Fake ratio	0.93536	0.93536	+0.00000
Duplicate ratio	0.94241	0.94241	+0.00000

6.3.4 O4 — Upfront Vector Reservation

H4 (Section 6.2.2) identified two independent allocation patterns in which `std::vector` containers grow on demand during the hot path: grid bin vectors during `CylindricalSpacePointGrid2::insert`, and the five struct-of-arrays members of `DoubletsForMiddleSp` during `emplace_back`. In both cases the amortised $\mathcal{O}(1)$ cost of `push_back` is accompanied by $\mathcal{O}(\log N)$ reallocations that copy existing elements to a larger buffer. O4 replaces both patterns with upfront `reserve` calls sized to an estimate or upper bound on the final vector size.

Implementation

O4 modifies three files, grouped into two independent allocation patterns.

Grid bin vectors (`GridTripletSeedingAlgorithm.cpp`). The grid bin structure is fully determined at construction time by the configured ϕ and z bin edges, so `numberOfBins()` is available before any space points are inserted. A reservation block is added immediately after grid construction and before the insertion loop:

```
const std::size_t nBins = grid.numberOfBins();
const std::size_t avgBinOccupancy =
    (spacePoints.size() + nBins - 1) / nBins;
for (std::size_t i = 0; i < nBins; ++i) {
  grid.at(i).reserve(avgBinOccupancy);
}
```

The ceiling division ensures that bins receiving above-average occupancy require at most one reallocation rather than several. Bin occupancy is not uniform across ϕ–z, so the estimate is conservative: low-occupancy bins carry some unused capacity, but the total memory overhead is bounded by one additional allocation's worth per bin.

Doublet container vectors (`DoubletsForMiddleSp.hpp`, `DoubletSeedFinder.cpp`). `DoubletsForMiddleSp::emplace_back` appends to five independent `std::vector` members simultaneously. Without prior reservation each vector follows its own independent growth schedule, producing up to five separate heap reallocations per doubling event. A `reserve` method is added to `DoubletsForMiddleSp` that forwards a single capacity request to all five members:

```
void reserve(std::size_t capacity) {
  m_spacePoints.reserve(capacity);
  m_cotTheta.reserve(capacity);
  m_er_iDeltaR.reserve(capacity);
  m_uv.reserve(capacity);
  m_xy.reserve(capacity);
}
```

This is called in `createDoubletsImpl` immediately after the `sortedByR` offset adjustment and before the candidate zip loop:

```
compatibleDoublets.reserve(
    compatibleDoublets.size() + candidateSps.size());
```

`candidateSps.size()` is $\mathcal{O}(1)$ on both `ConstRange` and `ConstSubset` and provides a tight upper bound, since at most one doublet can be produced per candidate. The `+ compatibleDoublets.size()` term ensures correctness when `createDoublets` is called multiple times on the same output container across different space-point groups.

Mechanical effect on allocations

Heaptrack (Table 6.16) confirms that both reservation strategies work as designed. Total allocation calls in `GridTripletSeedingAlgorithm::execute` fall to 0.81× `Baseline` on pixel and 0.90× on strip (−19.3 % and −9.6 %). The bulk of this reduction comes from the grid-bin reservations: allocation calls inside `CylindricalSpacePointGrid2::insert` drop to 0.22× `Baseline` on pixel and 0.28× on strip, confirming that the per-bin `reserve` call eliminates the majority of reallocation events during grid filling.

Peak heap usage increases slightly (1.05× pixel, 1.04× strip at the algorithm level). This is the expected trade-off of pre-sizing: low-occupancy bins carry unused capacity and the doublet containers are sized to their worst-case upper bound. The memory overhead is modest and bounded.

Table 6.16: Heaptrack summary for O4. Ratios are O4/Baseline; values below 1.00× indicate reduced allocation count or peak memory. The table shows both the algorithm-level totals and the targeted insertion site.

Function	Metric	Baseline	O4	Ratio O4 / Baseline
Pixel detector				
`GridTripletSeedingAlgorithm::execute`				
	Peak memory	14.5 MB	15.2 MB	1.05×
	Allocations	86 373	69 679	0.81×
`CylindricalSpacePointGrid2::insert`				
	Peak memory	1.2 MB	1.1 MB	0.92×
	Allocations	56 425	12 151	0.22×
Strip detector				
`GridTripletSeedingAlgorithm::execute`				
	Peak memory	8.2 MB	8.5 MB	1.04×
	Allocations	65 518	59 224	0.90×
`CylindricalSpacePointGrid2::insert`				
	Peak memory	585.7 kB	586.2 kB	≈1.0×
	Allocations	25 075	7 041	0.28×

Performance

Although the allocation-count reduction is substantial, the wall-clock result is net-negative on both detectors. O4 runs at 0.986× on pixel (1.4 % slower than `Baseline`, CI 0.985–0.987) and 0.982× on strip (1.8 % slower, CI 0.980–0.983). The local `createDoubletsImpl` wall-clock timer shows a larger per-call slowdown on both detectors (0.907× pixel, 0.881× strip). Callgrind (Table 6.17) corroborates: global instruction count increases by 1.83 % on pixel and 2.77 % on strip, with the local `createDoubletsImpl` Ir rising by 2.66 % and 4.50 % respectively.

The additional work enters through the reservation calls themselves. Each call to `DoubletsForMiddleSp::reserve` forwards the requested capacity to five independent `std::vector::reserve` calls and issues five allocator requests when those capacities exceed the current allocation. This `reserve` is invoked once per middle space point inside `createDoubletsImpl`, where the middle-SP loop executes $\approx 10^5$ times per event (Table 6.7). The grid-bin reservation in `CylindricalSpacePointGrid2::insert` adds a further per-bin loop at grid construction time, issuing one `reserve` call per bin regardless of whether that bin will receive any space points.

The result is a clear but instructive null finding: O4 eliminates the allocation pattern identified in H4, but the overhead of issuing the upfront reservations exceeds the cost savings from the eliminated reallocations. The allocator as actually implemented is fast enough, and the reallocated buffers small enough, that the `Baseline`'s on-demand growth is cheaper in total than the reserved pattern. This is a reminder that profiling identifies where cost appears in the profile, but whether a specific remediation is net-beneficial depends on the relative cost of the fix itself, which only the measured result can settle.

Branch mispredictions are essentially unchanged across global and local scopes (all four Bcm rows within ±2 % of `Baseline`), consistent with the reservation strategy introducing no substantial new branch structure.

Table 6.17: Wall-clock timing, Callgrind instruction counts, and branch mispredictions for `Baseline` and O4. Wall-clock values are mean per-event times with bootstrap 95 % CIs; ratios below 1.00× indicate O4 is slower than `Baseline`. Callgrind counts are totals over the profiling dataset; relative change is (O4 − Baseline)/Baseline.

Detector	Metric	Baseline	O4	Speedup / Rel. change
Pixel detector				
	Global time/event [ms]	581.76 ± 27.57	589.89 ± 27.66	0.986 [0.985, 0.987]
	`createDoubletsImpl` [ns]	226.09 ± 3.68	249.32 ± 6.24	0.907 [0.880, 0.934]
	Global instructions (Ir)	10.233×10^9	10.421×10^9	+1.83%
	`createDoubletsImpl` (Ir)	7.084×10^9	7.273×10^9	+2.66%
	Global branch mispred. (Bcm)	82.97×10^6	82.67×10^6	−0.36%
	`createDoubletsImpl` mispred. (Bcm)	49.26×10^6	49.44×10^6	+0.36%
Strip detector				
	Global time/event [ms]	185.56 ± 11.43	188.90 ± 11.40	0.982 [0.980, 0.983]
	`createDoubletsImpl` [ns]	116.79 ± 0.28	132.51 ± 0.59	0.881 [0.878, 0.884]
	Global instructions (Ir)	3.615×10^9	3.715×10^9	+2.77%
	`createDoubletsImpl` (Ir)	2.193×10^9	2.291×10^9	+4.50%
	Global branch mispred. (Bcm)	27.71×10^6	27.63×10^6	−0.29%
	`createDoubletsImpl` mispred. (Bcm)	15.14×10^6	15.44×10^6	+2.00%

Physics validation

All three metrics (efficiency, fake ratio, and duplicate ratio) are identical between `Baseline` and O4 to five decimal places; see Table 6.18. Pre-reserving container capacity does not change what is stored in the containers, so the set of produced triplets is identical to the `Baseline`.

Table 6.18: Physics validation for O4 (mean over 50 events). All differences are consistent with zero at five-decimal precision, confirming output equivalence with `Baseline`.

Metric	Baseline	O4	Δ
Efficiency	0.97350	0.97350	+0.00000
Fake ratio	0.93536	0.93536	+0.00000
Duplicate ratio	0.94241	0.94241	+0.00000

6.3.5 Cumulative Impact

O1 and O2 target non-overlapping functions in different translation units (`createDoubletsImpl` in `DoubletSeedFinder.cpp`; the compatibility-cut loop in `createPixelTripletTopCandidates` within `TripletSeedFinder.cpp`) and do not modify any shared data path, so their effects should be independent. This section reports their combined effect, which is what gets carried forward to the algorithm-level evaluation in Section 6.4.

O3 and O4 are excluded from the cumulative version. O3's global wall-clock effect is small in magnitude and inconsistent in sign between the two detectors, with a 0.6 % slowdown on pixel (0.994×) and a 0.9 % speedup on strip (1.009×, Section 6.3.3), so the net contribution averaged over both geometries is roughly zero. Its substantial local improvements (a 29 % instruction-count and 46 % misprediction reduction inside the sort cluster on pixel) do not propagate to the global wall-clock at the scale required to justify inclusion. O4 is net-negative on wall-clock time on both detectors despite its clean allocation-count reduction (Section 6.3.4) and is therefore not included. The cumulative version reported below is O1 + O2.

Recommendation

Of the four code-level optimisations evaluated in this section, only **O1** and **O2** are recommended for inclusion in the actual ACTS repository. They each deliver a measurable global wall-clock speedup with no detectable physics impact, and their effects stack independently. **O3** is not recommended: its local effect inside the sort cluster is striking (−29 % Ir, −46 % Bcm on pixel), but most of that reduction is an attribution shift caused by inlining the sort body into its caller, so the global wall-clock effect is small and inconsistent in sign between detectors. **O4** is actively counter-recommended: although it reduces allocation calls by 9–20 % as designed, the per-iteration overhead of the upfront `reserve` calls exceeds the savings from avoided reallocations, leaving a net 1–2 % wall-clock regression. The cumulative O1+O2 implementation is therefore the code-level configuration carried forward to the algorithm-level evaluation in Section 6.4.

Performance

Table 6.19 reports wall-clock timing, Callgrind instruction counts, and branch mispredictions for the cumulative O1+O2 configuration alongside the `Baseline`. Global wall-clock speedup is 1.048× on pixel (1.047–1.048 at 95 %) and 1.057× on strip (1.055–1.058). Naive multiplication of the individual speedups from Sections 6.3.1 and 6.3.2 predicts $1.041 \times 1.005 \approx 1.046\times$ on pixel and $1.035 \times 1.018 \approx 1.054\times$ on strip; the measured cumulative values are within measurement tolerance of these predictions, slightly on the favourable side. The optimisations stack independently, as expected from the fact that they act on different functions with no shared state or call path.

Callgrind corroborates the independence. On pixel, the cumulative configuration shows a 12.84 % reduction in local `createDoubletsImpl` instructions (matching O1 alone at 12.84 %) and a +2.02 % change in local `createPixelTripletTopCandidates` instructions (matching O2 alone at +2.02 %). The strip-detector pattern is identical: −14.94 % in `createDoubletsImpl` (matching O1 strip alone) and −0.60 % in `createPixelTripletTopCandidates` (matching O2 strip alone). Each optimisation's Callgrind signature appears in the cumulative configuration essentially unchanged, confirming that neither optimisation interferes with the other's effect at the instruction-count level.

The local wall-clock timer for `createDoubletsImpl` shows the same apparent null reading as for O1 alone (timer-overhead effect, discussed in Section 6.3.1); the local `createPixelTripletTopCandidates` timer resolves its expected $\sim$ 7–12 % per-call speedup cleanly, consistent with its per-call cost being large enough to outweigh the per-call timer overhead.

Branch mispredictions on the global counter decrease slightly on pixel (−0.49 %) and increase slightly on strip (+2.59 %), continuing the pattern observed for O1 and O2 individually. The net wall-clock speedup absorbs the strip-side misprediction increase because the instruction-count savings dominate the misprediction cost.

Table 6.19: Wall-clock timing, Callgrind instruction counts, and branch mispredictions for `Baseline` and the cumulative O1+O2 configuration. Wall-clock values are mean per-event times with bootstrap 95 % CIs; the speedup column is O1+O2 relative to `Baseline`. Callgrind counts are shown for both the function-local scopes targeted by O1 and O2 separately, so that the cumulative configuration's local signatures can be checked against the individual contributions.

Detector	Metric	Baseline	O1+O2	Speedup / Rel. change
Pixel detector				
	Global time/event [ms]	581.76 ± 27.57	555.21 ± 26.06	1.048 [1.047, 1.048]
	`createDoubletsImpl` [ns]	226.09 ± 3.68	223.29 ± 0.28	1.012 [1.000, 1.033]
	`createPixelTripletTopCandidates` [ns]	33.54 ± 0.16	31.33 ± 0.04	1.070 [1.067, 1.076]
	Global instructions (Ir)	10.233×10^9	9.333×10^9	−8.80%
	`createDoubletsImpl` (Ir)	7.084×10^9	6.175×10^9	−12.84%
	`createPixelTripletTopCandidates` (Ir)	709.41×10^6	723.75×10^6	+2.02%
	Global branch mispred. (Bcm)	82.97×10^6	82.57×10^6	−0.49%
	`createDoubletsImpl` mispred. (Bcm)	49.26×10^6	49.03×10^6	−0.47%
	`createPixelTripletTopCandidates` mispred. (Bcm)	5.54×10^6	6.09×10^6	+9.92%
Strip detector				
	Global time/event [ms]	185.56 ± 11.43	175.53 ± 10.61	1.057 [1.055, 1.058]
	`createDoubletsImpl` [ns]	116.79 ± 0.28	121.62 ± 4.94	0.961 [0.915, 0.991]
	`createPixelTripletTopCandidates` [ns]	35.95 ± 0.14	32.18 ± 0.03	1.117 [1.114, 1.122]
	Global instructions (Ir)	3.615×10^9	3.286×10^9	−9.11%
	`createDoubletsImpl` (Ir)	2.193×10^9	1.865×10^9	−14.94%
	`createPixelTripletTopCandidates` (Ir)	382.86×10^6	380.57×10^6	−0.60%
	Global branch mispred. (Bcm)	27.71×10^6	28.43×10^6	+2.59%
	`createDoubletsImpl` mispred. (Bcm)	15.14×10^6	15.72×10^6	+3.88%
	`createPixelTripletTopCandidates` mispred. (Bcm)	3.14×10^6	3.49×10^6	+11.06%

Physics validation

All three metrics are identical between `Baseline` and O1+O2 to five decimal places; see Table 6.20. Both O1 and O2 preserve output equivalence individually, and combining them introduces no interaction that would alter the produced triplet set.

Table 6.20: Physics validation for the cumulative O1+O2 configuration (mean over 50 events). All differences are consistent with zero at five-decimal precision, confirming output equivalence with `Baseline`.

Metric	Baseline	O1-2	Δ
Efficiency	0.97350	0.97350	+0.00000
Fake ratio	0.93536	0.93536	+0.00000
Duplicate ratio	0.94241	0.94241	+0.00000

6.4 Algorithm-Level Optimisations

The code-level optimisations of Section 6.3 targeted redundant computation and memory-access patterns within the existing seeding logic. This optimisation operates at a different level: it changes the algorithm's spatial data structure by replacing the cylindrical (ϕ, z, r) binning grid inside `GridTripletSeedingAlgorithm` with a spherical (ϕ, η, r) grid. Unlike O1 through O4, this optimisation is not motivated by Phase 2 hotspot identification but by a physics argument about how track densities distribute in the detector.

Spherical (ϕ, η, r) Grid Binning

In the cylindrical grid, uniform z-binning produces non-uniform coverage in pseudorapidity. Because

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

equal Δz intervals correspond to increasingly wide $\Delta\eta$ bins toward the forward region. This has two consequences.

First, particle production in high-energy proton-proton collisions is approximately uniform in rapidity, and therefore approximately uniform in (ϕ, η) for the relativistic particles considered here, but not in (ϕ, z). Under cylindrical binning the central bins become heavily populated while the forward bins remain sparse. The doublet-search inner loop has $O(N^2)$ cost in each bin's occupancy, so this imbalance concentrates the workload on a small number of heavily-populated central bins.

Second, charged-particle tracks originating from the interaction region propagate through the detector along paths of approximately constant η (up to corrections from magnetic bending and multiple scattering). Two space points on the same track therefore lie in approximately the same η bin regardless of the track's polar angle, whereas in z-binning they can lie in quite different z bins for forward tracks. Binning in η rather than z aligns the grid's neighbour-search structure with the natural geometry of tracks in the detector, reducing the number of bins a query must visit to find compatible doublet candidates.

Together, these arguments suggest that a (ϕ, η, r) grid should both balance occupancy more uniformly across bins and produce tighter neighbour queries. Whether these structural advantages translate into a measurable wall-clock speedup, and by how much, depends on implementation details that only measurement can settle: specifically, the cost of computing η from (z, r) (which involves a transcendental function), the grid's cache behaviour, and the overhead of the coordinate transformation at bin assignment.

Implementation

The optimisation replaces the z-axis of the seeding grid with pseudorapidity η, computed from each space point's coordinates as

$$\eta \;=\; \operatorname{asinh}\!\left(\frac{z}{\rho}\right), \qquad \rho = \sqrt{x^2 + y^2}.$$

The implementation uses a *dual-column* space-point layout that keeps the existing geometry calculations unchanged while enabling the spherical-grid neighbour search. The original (z, r) coordinates are retained for all geometric computations ($\cot\theta$, z-origin, impact parameter), while an (η, r) representation is used solely for bin assignment and neighbour queries. Column selection is resolved at compile time via a `useSphericalCoords` template flag in `DoubletSeedFinder`, so the downstream triplet-construction components (`TripletSeedFinder`, `TripletSeeder`, `BroadTripletSeedFilter`) are unchanged.

The following files were created or modified:

- `SphericalGridTripletSeedingAlgorithm.*` — new top-level algorithm class exposing the spherical grid to the ACTS framework.
- `SphericalSpacePointGrid2.*` — grid implementation with (ϕ, η, r) axes in place of (ϕ, z, r).
- `SpacePointContainer2.*`, `SpacePointProxy2.*` — extended with `Eta` and `EtaR` accessors to support the dual-column layout.
- `DoubletSeedFinder.cpp` — compile-time column selection via `if constexpr (useSphericalCoords)`.
- `TrackFindingPython.cpp`, `itk.py`, `reconstruction.py` — Python bindings and pipeline integration.

Parameter selection

The spherical grid introduces several η-specific parameters that have no counterpart in the cylindrical configuration. A parameter scan was performed over the principal parameters listed in Table 6.21 to identify operating points that balance physics performance against wall-clock time; full per-parameter scan plots are in Section B.3. Three configurations are retained: *Max Efficiency*, which maximises seeding efficiency; *Best*, which uses the cylindrical efficiency as a fixed working point while improving fake rejection and throughput; and *Fastest*, which maximises throughput at the cost of an efficiency loss in exchange for substantially improved fake rejection.

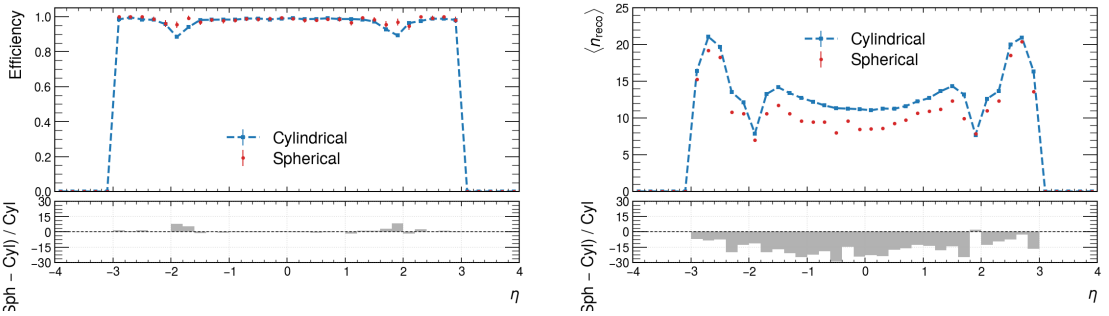
The three configurations deliver wall-clock speedups of 8.66×, 13.25×, and 11.12× over the cylindrical baseline, with different physics profiles. *Max Efficiency* reaches the highest efficiency (98.1 %, up 0.8 pp from cylindrical's 97.3 %) at the cost of a slight worsening of the duplicate ratio. *Best* preserves cylindrical efficiency (97.5 %, +0.2 pp) while improving both fake and duplicate ratios. *Fastest* delivers the largest fake-rate reduction (from 93.5 % to 33.0 %, a 60.5 pp drop) and the highest throughput, but its efficiency drops to 95.2 % (−2.1 pp below cylindrical), making it the only configuration that does not exceed the cylindrical efficiency baseline. *Best* is adopted as the spherical default version and is used for all spherical measurements reported in the remainder of this section, since it preserves the baseline efficiency margin while still delivering an order-of-magnitude wall-clock speedup.

Table 6.21: Selected spherical optimized parameters compared to the cylindrical baseline. The *Best* configuration is used in the physics validation that follows; *Max Efficiency* and *Fastest* are included to indicate the achievable envelope when one metric is prioritised. Parameter overrides are given relative to the runner defaults; metrics are means over 50 ODD events; speedup intervals are 95 % confidence intervals.

		Spherical configurations		
Quantity	**Cylindrical**	**Max Efficiency**	**Fastest**	**Best**
Parameter overrides (vs. runner defaults)				
$\Delta\eta_{\max}$	–	0.7	0.4	0.3
phiBinDeflectionCoverage	–	8	8	8
etaBinNeighborsBottom	–	$[-3,3]\times11$	$[-1,1]\times20$	$[-2,2]\times26$
etaBinNeighborsTop	–	$[-1,1]\times11$	$[-1,1]\times20$	$[-1,1]\times26$
maxSeedsPerSpM	–	4	1	4
impactMax	–	3	0.5	3
Efficiency	97.3%	**98.1%** (+0.8)	95.2% (-2.1)	97.5% (+0.2)
Fake ratio	93.5%	90.7% (-2.8)	**33.0%** (-60.6)	86.8% (-6.7)
Duplicate ratio	94.2%	95.0% (+0.8)	**85.7%** (-8.6)	90.5% (-3.8)
Wall-clock per event				
Time [ms]	607.0 [577.3, 637.4]	69.8 [67.0, 72.6]	45.5 [43.8, 47.2]	54.2 [52.2, 56.2]
Speedup vs. Cylindrical	–	8.66 [8.58, 8.73]	13.25 [13.08, 13.41]	11.12 [10.98, 11.26]

Physics validation

Figure 6.5 shows seeding efficiency and reconstructed track counts as a function of pseudorapidity η for the *Best* configuration alongside the cylindrical baseline. Seeding efficiency is preserved across the full η range: the relative difference (Spherical − Cylindrical)/Cylindrical lies within ±1 % almost everywhere, with larger fluctuations confined to $|\eta| \gtrsim 3$ where low statistics amplify bin-to-bin variation. Reconstructed track counts are essentially unchanged across the full η range.



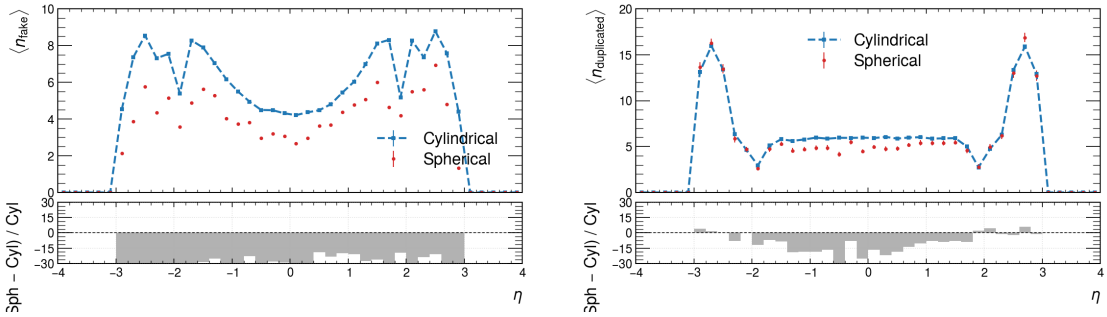
(a) Seeding efficiency.

(b) Reconstructed tracks.

Figure 6.5: Seeding efficiency and mean reconstructed track count per event as a function of pseudorapidity η for the cylindrical baseline and the spherical *Best* configuration. Each panel shows the absolute value above and the relative difference (Spherical − Cylindrical)/Cylindrical below. Efficiency is preserved across the full range, and reconstructed track counts are essentially unchanged.

The fake and duplicated track counts (Figure 6.6) show a more structured pattern. The spherical grid reduces both counts throughout the central region ($|\eta| \lesssim 2$), where

most tracks are produced. Integrated across η, these reductions translate into the improvements in the fake and duplicate ratios reported in the *Best* column of Table 6.21.



(a) Fake tracks.

(b) Duplicated tracks.

Figure 6.6: Mean fake and duplicated track counts per event as a function of pseudorapidity η for the cylindrical baseline and the spherical *Best* configuration. Each panel shows the absolute value above and the relative difference (Spherical − Cylindrical)/Cylindrical below. Both counts are reduced in the central region.

The central-region improvement matches the motivation for the spherical grid: more uniform bin occupancy in η reduces the number of candidate pairs evaluated in the densely populated central bins, which in turn reduces the fake and duplicated track counts there. The slight increases in fake and duplicated counts at forward η are small, within the variability expected in the low-statistics tails, and do not produce an integrated regression against the cylindrical baseline in any of the metrics.

Extended per-p_T, per-ϕ, per-z_0 breakdowns are provided in Section B.3.

Grid metrics comparison

Table 6.22 compares grid-level statistics between the cylindrical baseline and the spherical configuration. The structural differences predicted in the motivation are realised in the measured grid behaviour.

The input is unchanged: space points per event are identical between configurations (25,973), confirming that the spherical grid does not alter the upstream data. What changes is how that input is organised. Replacing the 2-bin z-axis with a 26-bin η-axis contributes a factor of 13× to the bin-count increase, and a finer ϕ-resolution contributes a further 7.2× (from 28 to 202 ϕ-bins). The r-axis is unchanged. The product is a 94× rise in grid size, from 56 to 5,252 cells; the number of occupied cells rises by 77× (from 52 to 3,983).

The finer cell structure reduces mean per-bin occupancy from 499.5 to 6.5 space points per occupied bin, a 77× drop. This is the key figure of merit for the doublet-search inner loop: candidate generation is approximately quadratic in occupancy within each bin neighbourhood, so the 77× reduction in mean occupancy translates into a substantial reduction in combinatorial work, with no change to the underlying space-point set. The spherical grid therefore achieves what the motivation argued it should: it redistributes the same input across a finer angular partition that matches the natural (ϕ, η) clustering of track space points.

A direct consequence of the finer cells is a higher empty-cell fraction: 24.2 % under spherical versus 7.1 % under cylindrical. This is not a regression. With a 94× finer grid partitioning a fixed set of $\sim$ 26,000 space points, some cells will be empty by chance, and they carry no runtime cost because the `binnedGroup` iteration of the seeding loop skips

empty cells without entering the inner compatibility-cut loop. The density of occupied cells matters more than the empty-cell fraction (−98.7 %, 499.5 → 6.5). The cylindrical grid's lower empty fraction reflects a coarse partitioning that places many kinematically unrelated space points in each cell; the spherical grid's higher empty fraction reflects having enough resolution to separate them.

The propagation of these grid-level changes through the doublet, triplet, and seed-output stages of the seeding pipeline is taken up in the Seeding3 workload analysis (Section 6.5); it suffices here to note that the predicted structural change is quantitatively realised in the measured grid behaviour, which is the precondition for the downstream speedup analysed in the rest of this section.

Table 6.22: Grid statistics comparing the cylindrical baseline and spherical configurations, reported per event as means over 50 events. The spherical grid raises grid cardinality by 94× (from 56 to 5,252 cells) and lowers mean per-bin occupancy by 77× (from 499.5 to 6.5 space points per occupied bin). The accompanying rise in empty-cell fraction (7.1 % → 24.2 %) is an expected consequence of the finer partitioning at fixed input size; empty cells carry no runtime cost because the seeding loop skips them. Downstream counts (doublets, triplets, seeds) are reported separately in Table 6.27.

Metric (per event)	**Cylindrical**	**Spherical**	**Abs. diff.**	**% diff.**
Total space points	25,973	25,973	+0	+0.0%
n_ϕ bins	28	202	+174	+621.4%
$n_{z/\eta}$ bins	2	26	+24	+1200.0%
n_r bins	1	1	+0	+0.0%
Total bins	56	5,252	+5,196	+9278.6%
Occupied bins	52	3,983	+3,931	+7559.2%
Empty bins [%]	7.14	24.17	+17.02	+238.3%
Mean SPs per occupied bin	499.49	6.52	−492.97	−98.7%

Callgrind

Table 6.23 summarises Callgrind performance counters for the cylindrical baseline and the spherical configuration. Because Callgrind instrumentation adds roughly 20–30× runtime overhead, measurements are taken on a 3-event subset rather than the full 50-event dataset; Table 6.24 is drawn from the same data.

The instruction count (Ir) reduces to 0.09× the baseline, an eleven-fold reduction in total computational work. Data accesses (Dr+Dw) reduce proportionally, to 0.11×, consistent with each unit of computation carrying roughly the same memory-operation intensity under both configurations.

Two of the four secondary metrics track the iteration-structure change implied by the grid-stats comparison (Table 6.22): level-one instruction-cache misses (I1mr, 6.50×) and the absolute branch-misprediction count (Bcm, 0.10×). Under the cylindrical baseline, 52 middle bin groups are processed per event, each containing on average 499.5 space points; the inner compatibility-check loops therefore execute many iterations per bin visit, with a hot instruction footprint that stays resident in the instruction cache across iterations and a branch predictor that accumulates confidence from repeated

exposure to the same branch structure. Under the spherical grid, 3,983 middle bin groups are processed per event, each containing only 6.5 space points. The inner loops execute briefly and return to the bin-entry code at every transition, so the same hot code is re-entered far more often. This explains the 6.5× increase in I1mr: the instruction cache's working set is the same size but is evicted and reloaded at each bin boundary. Bcm, by contrast, falls to 0.10× baseline, close to the Ir ratio of 0.09×, so branch mispredictions scale almost linearly with total instructions executed and the per-instruction misprediction rate is essentially unchanged.

The fourth secondary metric, LLC data misses, is essentially unchanged ($\approx 1.0\times$, $1.41 \times 10^4 \rightarrow 1.44 \times 10^4$). The absolute counts are of order 10^4 against 10^{10} instructions, too small in both configurations for the ratio to carry meaningful information; the working set of the seeding loop fits comfortably in the last-level cache on both grids.

These secondary effects are side effects of the spherical grid's many-short-bin-visits iteration pattern rather than regressions. The factor-of-eleven Ir reduction sets the runtime expectation for the optimisation, and the wall-clock measurement (Table 6.21) shows that the predicted reduction is achieved in practice.

Table 6.23: Callgrind performance counters for the cylindrical baseline and spherical configurations, aggregated over a 3-event subset of the physics-validation dataset. Counts are program-wide totals; the Ratio column is Spherical/Cylindrical. Measurements are instrumentation-based, so the ratios remain interpretable even though absolute wall-clock time under Callgrind exceeds native execution by more than an order of magnitude.

Metric	Cylindrical	Spherical	Spherical / Cylindrical
Computational work			
Instructions (Ir)	1.02×10^{10}	9.37×10^8	0.09×
Memory access behaviour			
Data accesses (Dr + Dw)	3.95×10^9	4.52×10^8	0.11×
LLC data misses (DLmr + DLmw)	1.41×10^4	1.44×10^4	≈1.0×
Instruction locality			
Instruction-cache misses (I1mr)	9.7×10^3	6.31×10^4	6.50×
Control-flow efficiency			
Branch mispredictions (Bcm)	9.16×10^7	8.96×10^6	0.10×

Table 6.24 breaks the reduction down by major seeding function. All three functions reduce by close to an order of magnitude in Ir: `createPixelTripletTopCandidates` drops to 0.05× its cylindrical instruction count (the largest reduction), `createDoubletsImpl` to 0.07×, and `filterTripletTopCandidates` to 0.09×. Branch mispredictions follow the same trend, with `createPixelTripletTopCandidates` showing the largest per-function Bcm reduction (0.08×) and the other two functions at 0.13×.

Two supporting routines included in Table 6.24 respond in the opposite direction. `TripletSeeder::createSeedsFromGroups`, the orchestration loop that iterates over middle bin groups, shows an Ir ratio of 2.72×, attributable to the increased number of bin groups iterated over (3,983 versus 52). `SpacePointGrid2::insert` shows an Ir ratio of 2.43×, attributable to insertion into 5,252 rather than 56 bins. Both increases are small in absolute terms (3.3 % and 2.2 % of total spherical program Ir respectively, by dividing the per-function spherical Ir in Table 6.24 by the program total in Table 6.23),

Table 6.24: Callgrind per-function breakdown for the major seeding routines, comparing the cylindrical baseline and spherical. All entries aggregate over template instantiations of the named function. The Ratio column is Spherical/Cylindrical; values below 1× indicate a reduction. The data is from the same 3-event subset as Table 6.23.

Function	Metric	Cylindrical	Spherical	Spherical / Cylindrical
`createDoubletsImpl`				
	Instructions (Ir)	3.48×10^9	2.27×10^8	0.07×
	Data accesses (Dr + Dw)	9.73×10^8	9.31×10^7	0.10×
	LLC data misses (DLmr + DLmw)	3×10^0	1×10^0	0.33×
	Instr. cache misses (I1mr)	2.19×10^2	2.78×10^3	12.71×
	Branch mispredictions (Bcm)	1.78×10^7	2.32×10^6	0.13×
`createPixelTripletTopCandidates`				
	Instructions (Ir)	3.08×10^9	1.45×10^8	0.05×
	Data accesses (Dr + Dw)	1.1×10^9	6.52×10^7	0.06×
	LLC data misses (DLmr + DLmw)	4×10^0	0	0.00×
	Instr. cache misses (I1mr)	6.5×10^1	6.1×10^1	0.94×
	Branch mispredictions (Bcm)	1.75×10^7	1.42×10^6	0.08×
`filterTripletTopCandidates`				
	Instructions (Ir)	8.2×10^8	7.65×10^7	0.09×
	Data accesses (Dr + Dw)	4.52×10^8	4.15×10^7	0.09×
	LLC data misses (DLmr + DLmw)	3.3×10^1	3×10^0	0.09×
	Instr. cache misses (I1mr)	3.85×10^2	4.86×10^3	12.63×
	Branch mispredictions (Bcm)	5.84×10^6	7.32×10^5	0.13×
`TripletSeeder::createSeedsFromGroups`				
	Instructions (Ir)	1.13×10^7	3.08×10^7	2.72×
	Data accesses (Dr + Dw)	4.61×10^6	1.29×10^7	2.79×
	LLC data misses (DLmr + DLmw)	1.5×10^1	9×10^0	0.60×
	Instr. cache misses (I1mr)	6.3×10^1	3.36×10^2	5.33×
	Branch mispredictions (Bcm)	1.06×10^5	4.7×10^5	4.42×
`SpacePointGrid2::insert`				
	Instructions (Ir)	8.38×10^6	2.04×10^7	2.43×
	Data accesses (Dr + Dw)	1.84×10^6	5.45×10^6	2.95×
	LLC data misses (DLmr + DLmw)	5×10^0	3×10^0	0.60×
	Instr. cache misses (I1mr)	2.4×10^1	6.6×10^1	2.75×
	Branch mispredictions (Bcm)	4.17×10^3	1.23×10^5	29.46×

and are far outweighed by the order-of-magnitude reductions in the dominant inner-loop hotspots; but the per-function reduction is not uniform across every routine affected by the change.

The instruction-cache behaviour is not uniform across functions either. `createDoubletsImpl` and `filterTripletTopCandidates` show elevated I1mr ratios (12.7× and 12.6× respectively), consistent with the short-bin-visit pattern repeatedly evicting and reloading their inner-loop code. `createPixelTripletTopCandidates`, by contrast, is essentially unchanged in I1mr (0.94×): its inner loop is short enough that the bin-boundary churn does not cross instruction-cache lines. The named functions in Table 6.24 account for 8 % of total I1mr on cylindrical and 13 % on spherical, so named-function I1mr grows somewhat faster than total I1mr. The global 6.5× I1mr ratio decomposes as $\sim 10.7\times$ for the named functions and $\sim 6.1\times$ for the residual framework, sort, and dispatch code. Both reflect the same short-bin-visit iteration pattern, not a property of the algorithm change itself.

The order-of-magnitude reduction across the dominant inner-loop functions is the signature of a data-structure-level change rather than a code-level tuning. The spherical grid does not optimise any single routine; it replaces the grid that every downstream routine queries for neighbour candidates, and the inner-loop savings propagate consistently across the three hotspots. The secondary routines (grid construction, orchestration) move less uniformly, and a few small ones scale unfavourably with the finer grid for reasons noted above, but the inner-loop savings dominate the totals: the program-wide instruction count drops by a factor of eleven (Table 6.23) despite the few routines that scale the other way.

Heaptrack

Table 6.25 reports peak heap memory and allocation counts for the three seeding functions most affected by the grid coordinate change, as measured by Heaptrack.

Peak memory decreases across all functions. The largest reduction is in `TripletSeeder::createSeedsFromGroups`: from 60.8 kB to 10.3 kB (0.17×), reflecting that the spherical grid's finer binning produces smaller bin-group data structures resident in memory during each seeding iteration. At the top level, `GridTripletSeedingAlgorithm::execute` peak memory falls from 2.2 MB to 1.6 MB (0.73×): the spherical grid still needs to hold the grid's bin structure in memory, and although each bin is smaller, there are many more of them.

Allocation counts show a trade-off characteristic of finer binning. `SpacePointGrid2::insert` issues 26.9× more allocation calls under spherical (1,622 → 43,674), a direct consequence of the bin count rising from 56 to 5,252 (Table 6.22): each bin's vector is allocated independently, so more bins means more allocations. This propagates to the top level, where `execute` issues 1.48× more allocations overall (77,381 → 114,310).

The higher allocation count has no practical impact on the measured runtime. The allocations are short-lived (`insert` peak memory remains at 0 B, indicating the allocated buffers are released before the function returns); their absolute count is small relative to total per-event work ($\sim 10^5$ allocations against $\sim 10^9$ instructions); and the overall computational cost is still dominated by the factor-of-eleven Ir reduction established by Callgrind. The allocator overhead that proved limiting for O4 in the code-level optimisations (Section 6.3.4) was driven by `reserve` calls in the middle-SP loop, called $\sim 10^5$ times per event. The spherical grid's additional allocations occur only at grid construction time and are amortised across all subsequent seeding work.

Table 6.25: Heaptrack peak heap memory and allocation counts for the major seeding functions, comparing the cylindrical baseline and spherical configurations. Peak memory is the maximum simultaneously live heap during the function call; allocations is the total number of heap allocation calls. The Ratio column is Spherical/Cylindrical.

Function	Metric	Cylindrical	Spherical	Ratio Spherical / Cylindrical
`GridTripletSeedingAlgorithm::execute`				
	Peak memory	2.2 MB	1.6 MB	0.73×
	Allocations	77 381	114 310	1.48×
`TripletSeeder::createSeedsFromGroups`				
	Peak memory	60.8 kB	10.3 kB	0.17×
	Allocations	75 571	70 436	0.93×
`SpacePointGrid2::insert`				
	Peak memory	0 B	0 B	≈1.0×
	Allocations	1 622	43 674	26.93×

Summary

The spherical grid's three lines of evidence converge on a consistent picture. The grid-stats comparison (Table 6.22) establishes the mechanical change: mean per-bin occupancy drops from 499.5 to 6.5 space points per occupied bin (77×), while the number of occupied middle bin groups rises by the same factor (from 52 to 3,983). Callgrind (Tables 6.23 and 6.24) shows the computational consequence: the three dominant inner-loop hot spots reduce to 0.05–0.09× their cylindrical instruction count, an order-of-magnitude reduction that signals a data-structure-level change rather than localised tuning. A few supporting routines (grid construction, orchestration) scale the other way as the cost of finer binning, but their share of program Ir is too low to offset the inner-loop savings. Wall-clock measurement (Table 6.21) confirms the runtime effect: the *Best* configuration runs at 13.25× the cylindrical speed, tracking the Callgrind global Ir ratio of $\sim 11\times$ with a small excess from favourable secondary effects (lower Bcm, no D-cache regression) that the Ir count alone does not capture. Physics metrics (Figures 6.5 and 6.6) show no regression and improve marginally in the central pseudorapidity region, matching the prediction that better bin-occupancy balance would reduce candidate-pair work where most tracks are produced. The secondary effects identified in the Callgrind breakdown (an elevated I-cache miss rate on the two larger inner loops, a higher allocation count, and an essentially unchanged LLC-miss rate) arise from the spherical grid's many-short-bin-visits iteration pattern, but are too small to offset the primary gain. The wall-clock speedup tracking the Callgrind instruction ratio closely confirms that none of them meaningfully degrades runtime. The spherical grid is an algorithm-level change acting on the spatial data structure, while O1 and O2 are code-level optimisations that modify inner-loop compatibility checks within the existing cylindrical grid. Because they operate on different layers, they should compose naturally. The Seeding3 configuration evaluated in Section 6.5 applies O1+O2 within the spherical grid; whether the combined effect stacks cleanly is the question addressed there.

6.5 The `Seeding3` Implementation

Seeding3 is a derivative of Seeding2 that combines the optimisations whose individual evaluations preserved physics performance and produced positive wall-clock gains: O1 and O2 from the code-level set (Sections 6.3.1 and 6.3.2) and the algorithm-level Spherical (Section 6.4). O3 is excluded because its wall-clock effect is small in magnitude and inconsistent in sign between detectors, with a $\sim 0.6\,\%$ slowdown on pixel and a $\sim 0.9\,\%$ speedup on strip (Section 6.3.3), giving no clear net benefit. O4 is excluded because its wall-clock result is net-negative on both detectors despite a clean allocation-count reduction (Section 6.3.4). Seeding3 is therefore Seeding2 plus O1, O2, and Spherical: three changes to the inner-loop arithmetic and the spatial data structure that together aim to reduce redundant computation, defer expensive sub-expressions to smaller candidate subsets, and align the bin geometry with the approximate uniformity of particle production in pseudorapidity.

The reference for all Seeding3 measurements in this section is the unmodified Seeding2 implementation, the same baseline used throughout Chapter 6. All Seeding3 measurements are performed on the 50-event ODD dataset described in Section 4.2, single-threaded, with the bootstrap methodology described in Chapter 5. The individual contributions of O1, O2, and Spherical were characterised separately in Sections 6.3.1, 6.3.2 and 6.4; the focus here is on the integrated Seeding2-to-Seeding3 wall-clock comparison rather than re-deriving the individual contributions inside the composite. Because O1, O2, and Spherical individually preserve physics output relative to Seeding2, the physics-preservation claim carries forward for Seeding3; this section therefore focuses on runtime performance.

The three component optimisations are summarised below; full discussion is in Sections 6.3.1, 6.3.2 and 6.4.

O1 — `createDoubletsImpl` inner-loop reformulation. The collision-region origin check and the Δz range check constrain the same underlying quantity ($\cot\theta = \Delta z/\Delta r$) yet are evaluated as two independent branches for every candidate pair. O1 intersects the two constraints algebraically into a single tighter bound on $\cot\theta$, computes its limits once per middle space point outside the candidate loop, and replaces the two original branches with a single range test inside the loop (Section 6.3.1).

O2 — Loop reordering in `createPixelTripletTopCandidates`. In the baseline compatibility-cut loop, the full `error2` computation is paid for every top-doublet candidate before the cheaper $\delta\cot^2\theta$ discriminant is evaluated. O2 exploits the fact that `error2` ≥ 0 by construction to introduce a cheap pre-filter on $\delta\cot^2\theta$, deferring the `error2` computation to the subset of candidates that survive it (Section 6.3.2).

Spherical — (ϕ, η, r) grid binning. The cylindrical grid bins uniformly in z, producing non-uniform $\Delta\eta$ bin widths and resulting in heavily populated central bins alongside sparse forward bins. Spherical replaces the z-axis with pseudorapidity $\eta = \operatorname{asinh}(z/r)$, using (η, r) exclusively for bin assignment while retaining (z, r) for all physics calculations. The finer, more uniform binning reduces mean bin occupancy and the combinatorial growth of doublet and triplet candidates (Section 6.4).

Overall Performance

Table 6.26 reports the mean wall-clock time per event for Seeding2 (the thesis baseline) and Seeding3 (the composite proposed in this chapter), both measured on the ODD 50-event profiling dataset.

Seeding3 delivers an 11.571× wall-clock speedup over Seeding2 ([11.417, 11.723] at 95 %), reducing per-event wall-clock time from 613.1 ms to 52.6 ms and lifting single-thread event-processing throughput from 1.68 to 19.34 events per second. The CI half-width of ~ 0.15 (~ 1.3 % of the speedup) reflects a high-precision measurement.

The speedup combines contributions from two layers: the algorithm-level Spherical change to the spatial grid, which restructures the candidate pool the algorithm evaluates, and the code-level O1+O2 inner-loop reformulations, which reduce per-candidate compatibility-check cost. Because the two layers act on different stages of the algorithm — the spatial data structure versus the per-candidate compatibility checks — their contributions partially compose, with the spherical change being the dominant driver. The decomposition that follows examines how much of the overall speedup is attributable to workload reduction (the population of candidate pairs the algorithm has to evaluate) and how much to implementation efficiency (the per-candidate cost of evaluation).

Table 6.26: Overall wall-clock performance comparison between Seeding2 (the thesis baseline) and Seeding3 (the composite proposed in this chapter) on the ODD profiling dataset. Each configuration is run 5 times over the 50-event profiling dataset; the per-event time used in the comparison is the average across the 5 runs. Time per event and throughput are arithmetic means; speedup is the geometric mean of event-level ratios with a 95 % bootstrap confidence interval (10,000 iterations) — see Section 5.2.2 for the aggregation rationale. Wall-clock and CPU times agreed to within 0.5 % across both configurations, confirming negligible I/O overhead and consistent single-threaded execution; only wall-clock measurements are reported.

Detector	**Time [ms/event]**		**Throughput [events/s]**		**Speedup [95% CI]**
	Seeding2	Seeding3	Seeding2	Seeding3	
ODD	613.1 ± 29.9	52.6 ± 1.9	1.68 ± 0.08	19.34 ± 0.69	11.571 [11.417, 11.723]

Decomposing the Speedup: Workload Reduction vs. Implementation Efficiency

The 11.6× wall-clock speedup of Seeding3 over Seeding2 (Table 6.26) could come from two distinct sources: Seeding3 might do the same work as Seeding2 but faster (implementation efficiency), or it might do substantially less work per event (workload reduction), or both. Table 6.27 decomposes the improvement by pipeline stage to isolate how much of the speedup is attributable to each mechanism.

The grid-level change has already been established in the spherical-grid evaluation (Section 6.4): the spherical grid raises grid size by 94× (56 → 5,252 cells) and lowers mean per-bin occupancy by 77× (499.5 → 6.5 space points per occupied bin) without altering the underlying space-point set. What this section traces is how that structural change propagates downstream through the doublet, triplet, and seed-output stages.

Bottom doublet candidates drop by 90.6 % (from 3,257,721 to 304,769 per event) and top doublet candidates by 88.0 % (from 1,370,351 to 164,444). Triplet candidate generation falls by 90.3 % (from 781,208 to 75,505), while seed output falls by only 29.5 % (from 61,673 to 43,476). The asymmetry between the $\sim$ 90 % reduction in

candidate-pair work and the $\sim 30\,\%$ reduction in seed output is the key observation: Seeding3 eliminates roughly three times as much combinatorial work as it removes seed output, meaning the eliminated candidate pairs were disproportionately pairs that never produced valid seeds. The doublet-candidates-per-output- seed ratio drops from ~ 75 (4,628,072/61,673) under `Seeding2` to ~ 11 (469,213/43,476) under `Seeding3`, a $7\times$ gain in candidate-construction selectivity. The $30\,\%$ drop in seed output represents fakes and duplicates that the spherical grid never forms, consistent with the fake- and duplicate-rate improvements established by the Spherical physics validation (Section 6.4).

Table 6.27: Workload comparison between `Seeding2` and the `Seeding3` composite, reported per event as means over 50 events. Metrics are grouped by pipeline stage, from grid construction through seed output. Relative difference is computed as (Seeding3 − Seeding2)/Seeding2 and expressed as a percentage; for metrics already expressed as percentages (Empty bins), this is the ratio of percentages, not a percentage-point change.

Stage	**Metric**	`Seeding2`	`Seeding3`	**Rel. diff.**
Grid	Total bins	56	5 252	+9278.6%
	n_ϕ bins	28	202	+621.4%
	$n_{z/\eta}$ bins	2	26	+1200.0%
	n_r bins	1	1	0.0%
	Occupied bins	52	3 983	+7559.2%
	Empty bins [%]	7.14	24.17	+238.3%
	Mean SPs per occupied bin	499.49	6.52	−98.7%
Doublets	Bottom doublet candidates	3 257 721	304 769	−90.6%
	Top doublet candidates	1 370 351	164 444	−88.0%
Triplets	Created triplets	781 208	75 505	−90.3%
Output	Total seeds	61 673	43 476	−29.5%

Workload reduction dominates the wall-clock speedup. The $\sim 90\,\%$ drop in candidate counts at both the doublet and triplet stages implies a workload-driven speedup of roughly $10\times$ if per-candidate cost is unchanged. The measured $11.571\times$ exceeds this by $\sim 16\,\%$, attributable to the per-candidate cost reductions from O1 and O2 in the inner-loop compatibility checks. Spherical therefore provides the dominant workload reduction, while O1+O2 provide a smaller multiplicative implementation-efficiency gain on top.

Stage-Level Breakdown of the Speedup

The overall speedup (Section 6.5) and the workload decomposition (Section 6.5) establish *how much* faster Seeding3 is and *what* is being eliminated. The remaining question is *where* in the pipeline the savings materialise. Figure 6.7 decomposes per-event execution time into its constituent stages, expressed as fractions of the Seeding2 baseline total.

Nearly the entire speedup comes from three stages. `TripletEvaluation` alone

accounts for a 63.2 % reduction of the baseline total, with `MiddleBottoms` and `MiddleTops` contributing a further 17.0 % and 11.6 % respectively, together 91.8 % of the baseline time. These are the stages that perform per-candidate compatibility cuts inside the doublet and triplet search, and are the direct targets of both the spherical grid change (which reduces the candidate population) and the O1 and O2 inner-loop reformulations (which reduce per-candidate cost). The savings concentrate where the optimisations were targeted.

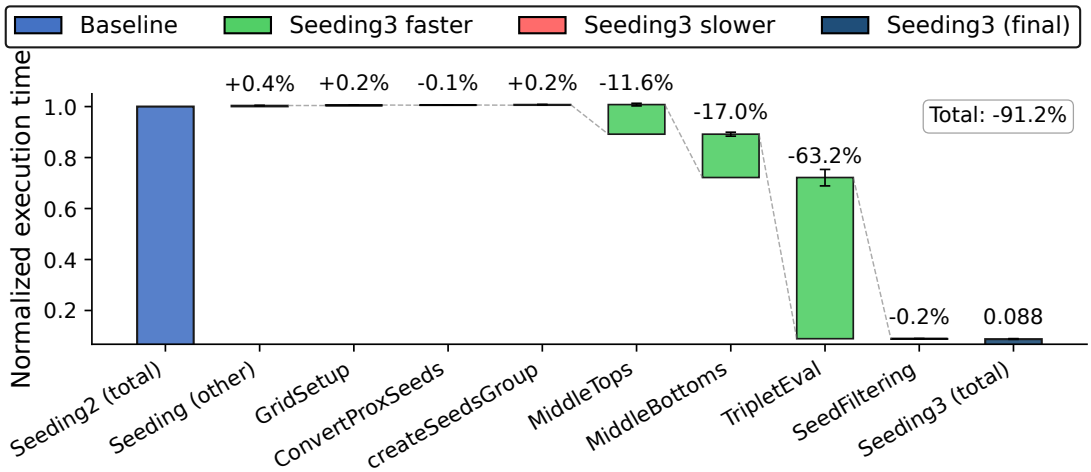


Figure 6.7: Stage-level waterfall decomposition of per-event execution time, comparing `Seeding2` (the thesis baseline) with `Seeding3`. Each bar gives the signed contribution of one instrumented stage to the total speedup, normalised to the Seeding2 baseline total time; green bars indicate stages where Seeding3 runs faster, red where it runs slower, and the final green bar gives the resulting Seeding3 total as a fraction of the baseline. Error bars are 95 % bootstrap confidence intervals (10,000 iterations). The total speedup measured from the instrumented build is 11.345× ([11.160, 11.519] at 95 %), slightly lower than the non-instrumented wall-clock speedup of 11.571× reported in Table 6.26; the instrumentation adds approximately the same absolute time to both configurations, which compresses the speedup ratio (roughly 2 % relative to Seeding3's native time).

Stages outside the inner-loop hot path contribute negligibly. Three of them (`Seeding(other)` residual, `GridSetup`, and `createSeedsGroup`) are marginally slower under Seeding3 (+0.4 %, +0.2 %, and +0.2 % of the baseline total respectively), while `ConvProxSeeds` and `SeedFiltering` each get marginally faster (−0.1 % and −0.2 %). The regression in `GridSetup` is expected: the spherical grid contains 94× more bins than the cylindrical baseline (Section 6.5), so grid construction has correspondingly more work to do. The regression is well under one percent of the baseline total, so the cost of the finer grid is amortised across the inner-loop savings, consistent with the spherical-grid analysis (Section 6.4), which found the additional allocations and elevated per-function instruction counts in `GridTripletSeedingAlgorithm::execute` and `SpacePointGrid2::insert` to be small relative to the inner-loop savings.

Performance Scaling with Space-Point Multiplicity

The optimisations in Seeding3 target inner-loop costs inside the doublet and triplet searches. Whether they also affect how the algorithm's runtime scales with input size is a separate question. This subsection examines per-event execution time as a function

of space-point multiplicity N_{SP} across both configurations, to test whether Seeding3's savings change the scaling behaviour relative to Seeding2.

Figure 6.8 plots per-event execution time against N_{SP} for each algorithm across the natural event-to-event spread of the profiling dataset. Both configurations show approximately linear scaling in this range, and a degree-1 polynomial fit is a good description of each (Table 6.28, with $R^2 \geq 0.987$ for both). Linearity in N_{SP} over this range is empirical rather than asymptotic: the inner-loop combinatorial work per bin is quadratic in bin occupancy, but bin occupancy stays approximately constant as N_{SP} grows, so the total cost scales with the number of occupied bins rather than with N_{SP} directly. This is the same mechanism that underlies the grid-based design in both the cylindrical baseline and the spherical grid.

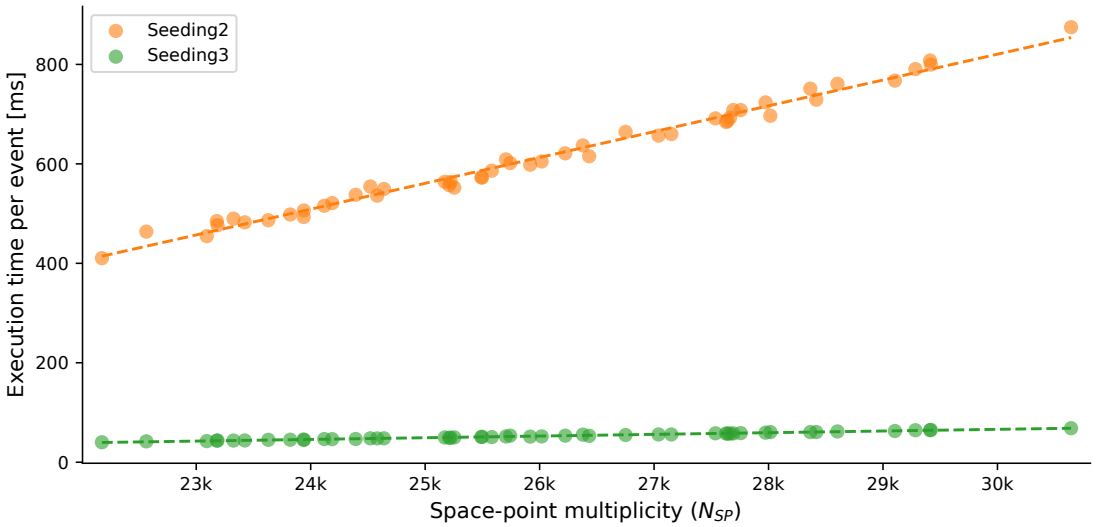


Figure 6.8: Mean per-event execution time as a function of space-point multiplicity N_{SP} for Seeding2 and Seeding3, averaged over 5 runs per configuration. Dashed lines show degree-1 polynomial fits; fit parameters are given in Table 6.28. The N_{SP} range shown corresponds to the natural event-to-event variation of the profiling dataset.

Table 6.28: Degree-1 polynomial fit parameters for per-event execution time as a function of space-point multiplicity, $y = a\,N_{\mathrm{SP}} + b$, with y in milliseconds. The slope a quantifies per-space-point cost in the linear regime; the intercept b has no physical interpretation (it is obtained by extrapolating the linear fit back to $N_{\mathrm{SP}} = 0$, outside the sampled range). The linear model is valid only within the observed N_{SP} interval.

Algorithm	a	b	R^2
`Seeding2`	5.193×10^{-2}	-7.373×10^{2}	0.9879
`Seeding3`	3.378×10^{-3}	-3.520×10^{1}	0.9930

The substantive comparison is in the slopes. Seeding3's per-space-point slope (3.378×10^{-3} ms) is 15× smaller than Seeding2's (5.193×10^{-2} ms), consistent with the inner-loop savings established in Section 6.5: a reduction in candidate work per space point reduces the rate at which runtime grows with N_{SP}. The slope reduction is the signature of a per-candidate work reduction rather than a fixed-cost change: a fixed-cost optimisation would shift the line down without changing its slope. Within the measured range, the per-event speedup is 11.57×; the larger slope ratio of 15×

means that, if the linear scaling continues at higher N_{SP}, the Seeding3 advantage would approach the slope ratio rather than the currently observed wall-clock ratio. This is the desired behaviour for HL-LHC conditions, where N_{SP} is expected to scale further with pile-up.

Summary

The four lines of evidence presented in this section converge on a consistent picture. Seeding3 achieves a wall-clock speedup of 11.571× over Seeding2 ([11.417, 11.723] at 95 %, Table 6.26), reducing per-event runtime from 613.1 ms to 52.6 ms and lifting single-thread throughput from 1.68 to 19.34 events per second. The contributions of the three modifications partially compose: the algorithm-level spherical grid is the dominant driver, with the code-level O1 and O2 reformulations adding a smaller multiplicative gain on top.

The workload decomposition (Section 6.5) shows that the spherical grid reduces mean doublet candidates by $\sim$ 90 % and triplet candidates by $\sim$ 90 %, while the number of output seeds falls by only $\sim$ 30 %. The finer binning therefore eliminates combinatorial candidates that would have produced fakes or duplicates rather than additional real tracks: efficiency is preserved while fake and duplicate ratios fall. Workload reduction alone accounts for approximately 10× of the speedup; the remaining $\sim$ 16 % multiplicative factor comes from O1 and O2, which reduce the per-candidate cost of the work that remains.

The waterfall decomposition (Section 6.5) localises the savings to three stages of the seeding pipeline: `TripletEvaluation`, `MiddleBottoms`, and `MiddleTops` together account for 91.8 % of the Seeding2 baseline total time, the stages where per-candidate compatibility cuts are evaluated and therefore the targets of the optimisation design. Stages outside the inner-loop hot path are largely unchanged, and the minor regression in `GridSetup` (+0.2 % of baseline total) is the expected cost of the spherical grid's 94× finer bin structure, small enough to be amortised many times over by the downstream savings.

The scaling analysis (Section 6.5) shows that Seeding3's slope against space-point multiplicity is 15× smaller than Seeding2's. The speedup is therefore not a fixed-cost reduction but a reduction in per-space-point work, and the Seeding3 advantage would grow further with N_{SP} if the linear scaling continues, which is the behaviour desired for HL-LHC conditions, where space-point counts scale with pile-up.

Taken together, these measurements establish that combining targeted code-level inner-loop reformulations (O1, O2) with an algorithm-level change to the spatial grid (Spherical) yields a composite implementation that is an order of magnitude faster than the thesis baseline, preserves physics output within measurement precision, and scales more favourably with input size. The principal methodological finding is that performance work at distinct abstraction levels (inner-loop arithmetic and spatial binning) can be developed and evaluated independently and combined to compound effects, though the contributions do not stack cleanly multiplicatively: the spatial-grid change dominates, with the inner-loop reformulations adding a smaller multiplicative gain inside the improved grid.

Chapter 7

Conclusion

This thesis presented a systematic profiling study and optimisation effort targeting the CPU performance of the ACTS track seeding algorithm in preparation for HL-LHC-scale data processing. Starting from a detailed characterisation of the baseline `Seeding2` implementation, a two-phase profiling methodology was used to identify dominant computational bottlenecks, motivate targeted changes, and quantify their impact through wall-clock timing, Callgrind instruction counts, Heaptrack allocation metrics, and physics validation. The culminating contribution is `Seeding3`, a composite implementation combining the optimisations that individually satisfied both runtime and physics-preservation criteria. This chapter summarises the findings in relation to the research questions posed in Section 1.3, discusses their limitations, and outlines directions for future work.

7.1 Summary of Findings

RQ1 — Profiling the original implementation. Where does the seeding algorithm spend most of its runtime, and what are the main computational bottlenecks?

Lightweight `gperftools` sampling identified four dominant hotspots in `Seeding2` (H1–H4). The innermost doublet-construction loop in `createDoubletsImpl` is the single largest contributor, at 56.83 % flat time on pixel and 45.14 % on strip, with cost distributed across the entire function rather than localised to a single expression. Three further hotspots were identified: an ordering inefficiency in `createPixelTripletTopCandidates`, where an expensive `error2` computation preceded a cheaper rejection criterion; indirect call overhead in the sort path via the `sortByCotTheta` projection lambda; and repeated heap allocation in two independent vector-growth paths, namely grid bin vectors and the five-member `DoubletsForMiddleSp` struct-of-arrays. Stage-level timing confirmed that `MiddleBottoms`, `MiddleTops`, and `TripletEvaluation` together account for the large majority of pixel and strip runtime, establishing that any meaningful speedup must originate within the doublet–triplet loop nest.

RQ2 — Developing optimisations. Can code-level or algorithmic optimisations improve runtime without degrading seeding quality metrics, such as efficiency and fake rate?

Four code-level optimisations (O1–O4) and one algorithm-level optimisation (Spherical) were implemented and evaluated independently. All five preserved physics output within measurement precision, confirming that the underlying optimisation

hypotheses were compatible with the algorithm's reconstruction requirements. However, only three delivered a positive runtime contribution.

O1 reformulated the `createDoubletsImpl` inner loop, merging the collision-region origin check and the Δz range check into a single tighter bound on $\cot\theta$ computed once per middle space point. Local instruction count was reduced by 12.84 % on pixel and 14.94 % on strip, with a global wall-clock speedup of 1.041× on pixel and 1.035× on strip.

O2 reordered the compatibility-cut computation in `createPixelTripletTopCandidates` so that the cheap $\delta\cot^2\theta$ pre-filter ran before the expensive `error2` computation, deferring the expensive path to the subset of candidates that survive the cheap one. Per-call wall-clock cost of the targeted function fell by 7.9 % on pixel and 12.0 % on strip; the small share of the function in total runtime limits the global wall-clock contribution to 0.5–1.8 %.

O3 eliminated runtime indirection in the sort path by replacing a type-erased `std::ranges::sort` projection with a direct-template comparator, recovering compiler inlining. Local branch mispredictions inside the sort cluster fell by 46 % on pixel and 32 % on strip, the cleanest branch-prediction improvement observed across the four code-level optimisations. The global wall-clock effect was small and inconsistent in sign between detectors, with a 0.6 % slowdown on pixel and a 0.9 % speedup on strip. This reflects both the sort cluster's modest share of total runtime and the fact that much of the local reduction is an attribution shift: the inlined sort body is now counted under the calling function's scope rather than under the sort cluster.

O4 added upfront `reserve` calls to the grid bin vectors and the doublet container. Although the intervention successfully reduced allocation call counts by 72–78 %, the per-`reserve` overhead introduced in the hot inner loop exceeded the savings from avoided reallocations, yielding a net wall-clock regression of 1.4 % on pixel and 1.8 % on strip. O4 is therefore reported as a methodologically instructive null result: profiling identifies where cost is incurred, but the cost of the intervention itself must also be accounted for.

Spherical replaced the cylindrical (ϕ, z, r) seeding grid with a spherical (ϕ, η, r) grid, aligning the bin geometry with the approximate conservation of pseudorapidity along particle tracks. Mean bin occupancy dropped by 98.7 %, from 499.5 to 6.5 space points per occupied bin, while the number of occupied middle bin groups rose by a factor of 77, from 52 to 3,983. Callgrind confirmed an order-of-magnitude instruction-count reduction across every dominant inner-loop function: `createDoubletsImpl`, `createPixelTripletTopCandidates`, and `filterTripletTopCandidates` all reduced to 0.05–0.09× their cylindrical counts, with a few small supporting routines moving in the opposite direction as the price of the finer binning. Wall-clock time per event for the *Best* operating point dropped from 607.0 ms to 54.2 ms, an 11.12× speedup, almost exactly matching the global Callgrind instruction-ratio reduction ($1/0.09 \approx 11.1\times$). Physics output was preserved, with the fake ratio and duplicate ratio marginally improved in the central pseudorapidity region.

RQ3 — Comparing implementations. How do the combined optimisations compare quantitatively to the baseline?

The `Seeding3` composite incorporates the three optimisations that achieved meaningful runtime reduction with preserved physics output: O1, O2, and Spherical. O3 is excluded because its global wall-clock effect is small in magnitude and inconsistent in sign between detectors, a $\sim$0.6 % pixel slowdown and a $\sim$0.9 % strip speedup, so its net contribution averaged over both geometries is approximately zero. O4 is excluded

because its wall-clock result is net-negative on both detectors despite the clean allocation-count reduction.

Measured on the ODD profiling dataset, `Seeding3` reduces per-event wall-clock time from 613.1 ms to 52.6 ms, a 11.571× speedup over `Seeding2` ([11.417, 11.723] at 95 %), lifting single-thread throughput from 1.68 to 19.34 events per second. The magnitude is consistent with the multiplicative composition of the three modifications, which act on different layers of the implementation, spatial data structure for Spherical, and inner-loop arithmetic for O1 and O2, and therefore appear to stack without significant interference. Workload decomposition shows that the Spherical grid is the dominant contributor, accounting for roughly $\sim 10\times$ of the speedup through a 90 % reduction in triplet candidates; the remaining ~ 16 % multiplicative factor comes from the O1+O2 inner-loop reformulations.

RQ4 — Evaluating practical implications. What are the implications for HL-LHC-scale data processing?

The HL-LHC will subject the reconstruction chain to an unprecedented rate of high-pile-up events at $\langle\mu\rangle = 200$, at which track seeding has been projected to become one of the dominant contributors to reconstruction time. To place the magnitude of the `Seeding3` speedup in context, published ATLAS profiling of the Run 3 reconstruction chain found that inner-detector track reconstruction accounted for around 68 % of total ATLAS event-reconstruction CPU time in the legacy software release at $\langle\mu\rangle = 50$ [15], making it the single largest event-level consumer of reconstruction CPU. Within track reconstruction itself, the track-finding step (of which seed formation is a direct input) is reported as "by far the largest CPU consumer for track and total ATLAS reconstruction" and the component that scales worst with $\langle\mu\rangle$ [15]. The same paper notes that seed formation specifically "contributes a significant fraction of the overall track reconstruction run time due to the combinatorial complexity of the process" [15].

The ATLAS Phase-II tracking-software effort has explicitly targeted this regime: a recent ACTS-based ITk track-finding optimisation campaign reports a 5.4× reduction in single-thread CPU time on $t\bar{t}$ events at $\langle\mu\rangle = 200$ relative to its initial integration, narrowing the gap to the legacy implementation to about 8.5 % [11]. An order-of-magnitude reduction in seeding wall-clock cost, achieved without compromise to physics output and accompanied by improved scaling behaviour with N_{SP} (Section 6.5), therefore feeds into the same time budget that the wider ATLAS optimisation campaign is working to recover. It also provides potential headroom either for tighter seeding criteria or for expanded reconstruction scope. The specific implications depend on the integrated reconstruction configuration. What this thesis establishes is that optimisations to the seeding stage can be implemented and tested under systematic profiling, and that such optimisations can deliver large local gains without degrading reconstruction quality.

The combination of findings answers the central methodological question posed in the introduction. Systematic profiling reveals specific, targetable bottlenecks in the ACTS seeding algorithm, and addressing them at the appropriate layer of abstraction (inner-loop arithmetic for O1 and O2, spatial data structure for Spherical) yields substantial runtime reductions with no degradation in physics performance.

7.2 Limitations

Several limitations qualify the generalisability of the reported results.

First, all measurements were performed on a single computing node (HEPP03, AMD EPYC 7713) under a fixed software stack (LCG 105, GCC 13). HEPP03 is a shared interactive node rather than a quiescent, dedicated benchmarking machine, and is not held in a quiet state during measurements; co-resident jobs therefore introduce a small but non-zero level of system noise. This noise is absorbed into the run-to-run variability captured by the bootstrap confidence intervals, typically at the few-percent level, and is far smaller than the headline speedups reported here. Nevertheless, individual absolute timings should not be over-interpreted. Absolute timing figures are hardware-specific and should not be extrapolated directly to other architectures without re-profiling. The relative speedups and optimisation decomposition are expected to be more robust to architectural variation, but this has not been verified.

Second, the profiling dataset uses precomputed space points from a single fixed ROOT file. This ensures reproducibility and isolates the seeding stage from upstream simulation cost, but does not capture the full event-to-event variability encountered in an online or production reconstruction pipeline. The physics validation dataset partially addresses this with geometry-aware events simulated at $\langle\mu\rangle = 200$, but it does not fully reproduce end-to-end production conditions.

Third, the physics validation uses the Open Data Detector (ODD), a simplified geometry that does not reproduce all features of the ATLAS ITk layout. Validation on a full ITk geometry with realistic pile-up remains outside the scope of this work.

Fourth, a broader scan of Spherical's parameter space, and a dedicated study of the sensitivity of physics metrics to parameter choices, would strengthen confidence in the selected configuration. The Spherical implementation in this thesis should therefore be regarded as a strong example baseline for further studies rather than as a globally optimised configuration.

Fifth, the O4 null result demonstrates that the act of applying a seemingly correct optimisation can itself be costly. The other optimisations included in `Seeding3` were not subject to a similarly rigorous per-intervention overhead analysis, and their individual contributions are measured only at the level of end-to-end wall-clock time. Finer decomposition of the per-intervention overheads would require additional targeted microbenchmarks.

Sixth, the optimisations O1–O4 were validated on the GNN4ITk pixel/strip dataset, while Spherical and `Seeding3` were measured on the ODD dataset. The cross-dataset comparison was addressed by reporting integrated wall-clock and Callgrind measurements on each dataset separately, rather than attempting multiplicative stacking across datasets. However, the absence of an ablation run that applies O1 and O2 standalone on the ODD geometry leaves their individual contribution inside the `Seeding3` composite quantified only through the overall integrated speedup.

7.3 Future Work

Several natural directions for future investigation follow from the results and limitations described above.

Parameter optimisation for the spherical grid. The Spherical configuration used in `Seeding3` was selected from a limited manual scan. An automated multi-objective

search balancing runtime speedup against physics quality could identify parameter configurations that offer better physics–runtime trade-offs.

Validation on ITk geometry. Repeating the physics validation on the full ATLAS ITk detector geometry, with realistic simulation, pile-up at $\langle\mu\rangle = 200$, and a substantially larger event count than the validation sample used here, would provide a more realistic test of the preserved reconstruction quality and expose any geometry-specific edge cases that the ODD cannot.

Multi-threaded and GPU execution. All results in this thesis target single-threaded CPU execution. Extending the profiling methodology to multi-threaded configurations and to GPU implementations such as `traccc` [26] would determine whether the identified bottlenecks persist under parallel execution and whether the spherical binning strategy transfers to GPU memory hierarchies.

Further inner-loop refinement. The waterfall analysis (Section 6.5) shows that under `Seeding3`, `TripletEvaluation` is still the largest remaining time contributor. With the grid-level work reduced by Spherical and the inner-loop predicates simplified by O1 and O2, further gains would require attacking either the residual arithmetic inside the triplet filter or the data layout used to hold triplet candidates. The same Callgrind-and-annotation methodology developed in this thesis is directly applicable to these remaining hotspots.

Extending the methodology to other reconstruction stages. The profiling methodology showcased here is not specific to seeding and can be applied to other stages of the ACTS reconstruction chain. It combines lightweight sampling for hotspot identification, source-level annotation for root-cause analysis, and instrumentation-based tools for confirmatory verification, and would apply particularly to the Combinatorial Kalman Filter (CKF) track-finding stage that follows seeding and may become the new bottleneck once seeding is no longer dominant.

7.4 Closing Remarks

The High-Luminosity LHC will produce collision data at an unprecedented rate, placing extreme demands on every stage of the reconstruction chain. This thesis has demonstrated that the ACTS track seeding algorithm contains addressable inefficiencies at both the code and algorithm level, that these inefficiencies can be identified through systematic profiling, and that addressing them can yield reductions in seeding wall-clock cost with no degradation in physics performance. Equally, the negative result of O4 shows that optimisation is not guaranteed: even apparently well-motivated interventions must be evaluated in terms of their end-to-end cost. The composite `Seeding3` implementation, and the methodology that produced it, together represent a step toward a reconstruction pipeline capable of sustaining high-quality tracking at HL-LHC occupancies, and provide a template that can be applied systematically to other stages of the chain as the framework continues to evolve.

Appendix A

Build Instructions

The build steps below reproduce the ACTS environment used in this thesis. The same procedure is automated end-to-end by the `environment/setupACTS.sh` and `environment/setup.sh` scripts in the project repository (see Section 5.1.3); the manual recipe is included here for reference and to document the exact CMake configuration applied.

Prerequisites

- An Apptainer (Singularity) shell with CVMFS mounted at `/cvmfs/sft.cern.ch/`.
- A local `gperftools` installation. The instructions below assume the install prefix `$HOME/usr/local`, i.e. headers in `$HOME/usr/local/include` and libraries in `$HOME/usr/local/lib`; adjust accordingly if a different prefix is used.
- `git-lfs` available on `PATH`. LCG 105 does not ship it; either install it manually (the repository's `environment/setupACTS.sh` pulls a static `git-lfs` binary into `$HOME/usr/local/bin/` and registers the filters automatically) or use a system-provided version. `git-lfs` is required because the OpenDataDetector submodule stores its duplicate-seed classifier weights as LFS blobs.

Clone the ACTS repository

```
mkdir -p ACTS
cd ACTS
mkdir -p source build
git clone --recursive https://github.com/acts-project/acts.git source/
cd source
```

Check out the ACTS release

This thesis uses ACTS `v44.0.0`.

```
git fetch --tags
git checkout v44.0.0
git submodule update --init --recursive
git submodule foreach --recursive 'git lfs pull || true'
```

Source the LCG view

```
source /cvmfs/sft.cern.ch/lcg/views/<lcg_release>/<lcg_platform>/setup.sh
```

This thesis uses `lcg_release = LCG_105` and `lcg_platform = x86_64-el9-gcc13-opt`.

Configure the ACTS build

Run from inside the ACTS source directory. The build tree is placed at `source/build/`.

```
cmake -B build -S . \
  -DCMAKE_BUILD_TYPE=Release \
  -DACTS_BUILD_EXAMPLES=ON \
  -DACTS_BUILD_EXAMPLES_PYTHON_BINDINGS=ON \
  -DACTS_BUILD_EXAMPLES_PYTHIA8=ON \
  -DACTS_BUILD_EXAMPLES_ROOT=ON \
  -DACTS_BUILD_EXAMPLES_DD4HEP=ON \
  -DACTS_BUILD_PLUGIN_DD4HEP=ON \
  -DACTS_BUILD_PLUGIN_JSON=ON \
  -DACTS_BUILD_PLUGIN_ROOT=ON \
  -DACTS_BUILD_PLUGIN_ONNX=ON \
  -DACTS_BUILD_FATRAS=ON \
  -DACTS_BUILD_ODD=ON \
  -DACTS_ENABLE_CPU_PROFILING=ON \
  -DACTS_GPERF_INSTALL_DIR=$HOME/usr/local
```

CMake options used:

`CMAKE_BUILD_TYPE=Release` Build with `-O3 -DNDEBUG` to match production deployment conditions.

`ACTS_BUILD_EXAMPLES=ON` Build the example applications used as drivers for the seeding pipeline.

`ACTS_BUILD_EXAMPLES_PYTHON_BINDINGS=ON` Build the Python bindings for the examples.

`ACTS_BUILD_EXAMPLES_PYTHIA8=ON` Build the Pythia8-based event generation example used by the ODD pipeline.

`ACTS_BUILD_EXAMPLES_ROOT=ON` Build the ROOT readers/writers used for input and output.

`ACTS_BUILD_EXAMPLES_DD4HEP=ON` Build the DD4hep-based example applications; required for the ODD.

`ACTS_BUILD_PLUGIN_DD4HEP=ON` Build the DD4hep geometry plugin.

`ACTS_BUILD_PLUGIN_JSON=ON` Build the JSON plugin; required for digitisation and seeding configuration files.

`ACTS_BUILD_PLUGIN_ROOT=ON` Build the ROOT plugin used for magnetic-field I/O and output.

`ACTS_BUILD_PLUGIN_ONNX=ON` Build the ONNX inference plugin, used by the ODD duplicate-seed classifier.

`ACTS_BUILD_FATRAS=ON` Build the Fatras fast-tracking simulator used for particle simulation in `full_chain_odd.py`.

`ACTS_BUILD_ODD=ON` Build the OpenDataDetector used in `full_chain_odd.py`.

`ACTS_ENABLE_CPU_PROFILING=ON` Link `gperftools` into the build to enable CPU profiling.

`ACTS_GPERF_INSTALL_DIR` Path to the local `gperftools` install prefix (here: `$HOME/usr/local`).

Build ACTS

```
cmake --build build -j$(nproc)
```

The full build takes approximately 15–30 minutes on a modern multi-core machine; adjust the `-j` value to fit available cores and memory.

Per-shell runtime environment

The runtime environment below must be re-applied in every new shell before running ACTS Python examples. The `environment/setup.sh` script in the thesis repository automates all of these steps and additionally verifies that `import acts` succeeds.

```
# 1. LCG view (compiler toolchain, ROOT, Pythia8, etc.)
source /cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-el9-gcc13-opt/setup.sh

# 2. ACTS Python bindings (from inside the ACTS source dir)
source build/python/setup.sh

# 3. OpenDataDetector
export ODD_PATH=$PWD/thirdparty/OpenDataDetector
export LD_LIBRARY_PATH=$PWD/build/thirdparty/OpenDataDetector/factory:\
${LD_LIBRARY_PATH:-}

# 4. gperftools (so libprofiler.so is locatable at runtime)
export LD_LIBRARY_PATH=$HOME/usr/local/lib:${LD_LIBRARY_PATH:-}
export PATH=$HOME/usr/local/bin:${PATH:-}
```

Appendix B

Supplementary Figures and Data

B.1 Baseline Comparison: `Seeding` vs. `Seeding2`

This appendix collects supplementary `Seeding` profiling artefacts for the baseline comparison in Section 6.1. The corresponding `Seeding2` artefacts are given in Section B.2.

`Seeding` strip-data flamegraph

Compared with the `Seeding2` strip flamegraph in Section B.2, the main structural differences match those seen for pixel data: a broader dominant plateau, a visible `filterCandidates` tower, and a less prominent sort cluster.

`Seeding` callgraphs

The full `gperftools` callgraphs for `Seeding` are shown below. They provide the reference point for the `Seeding2` callgraphs in Section B.2.

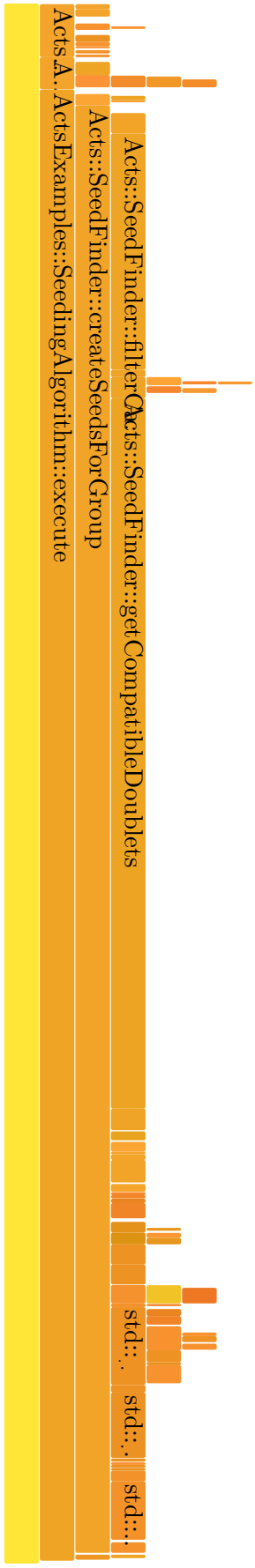


Figure B.1: CPU flamegraph for `Seeding` on strip data. Bar width denotes the fraction of on-CPU samples; colour is hash-based.

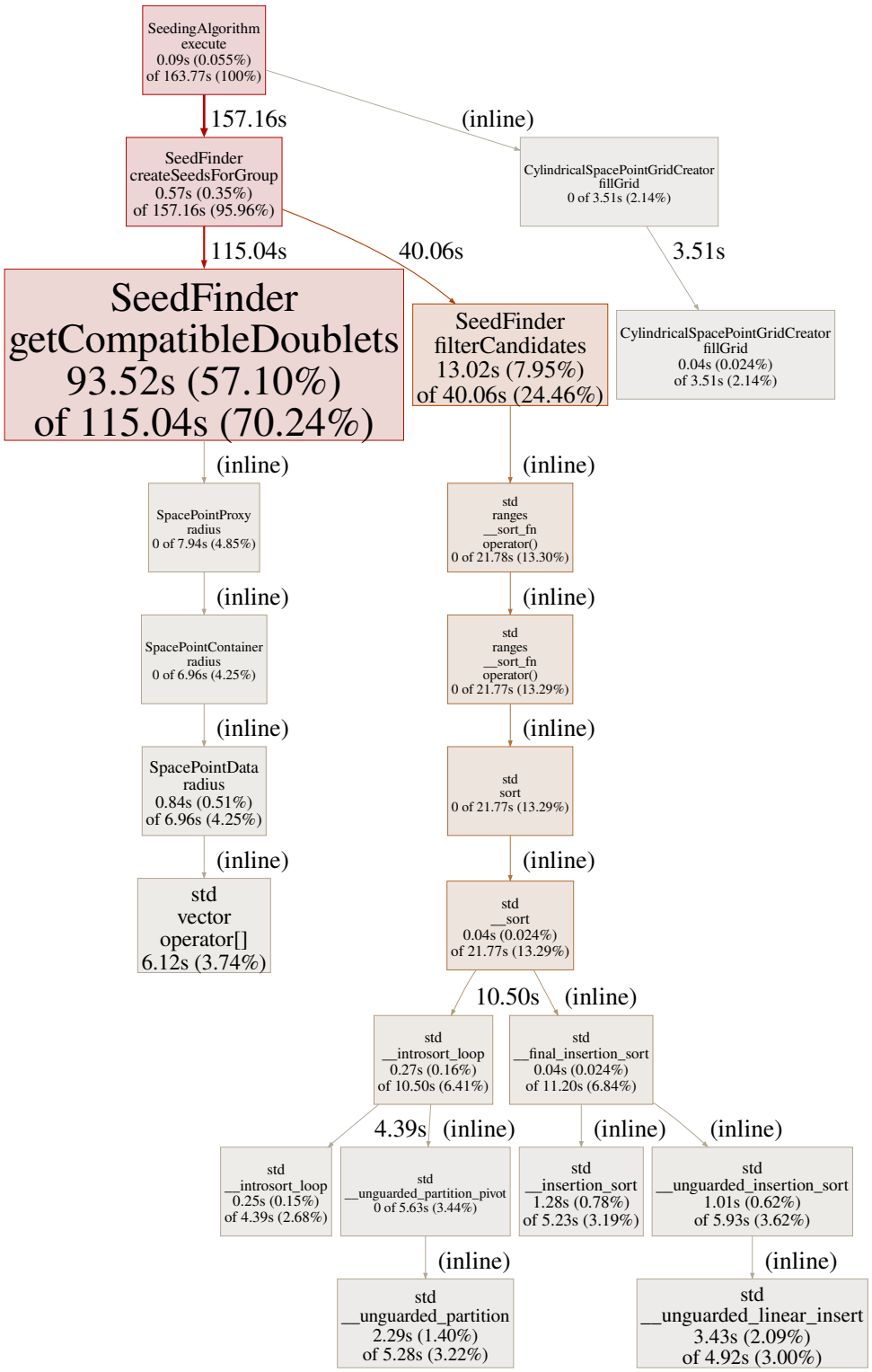


Figure B.2: Callgraph for Seeding on pixel data. Nodes show self-time and percentage of total runtime; edges show sampled call paths.

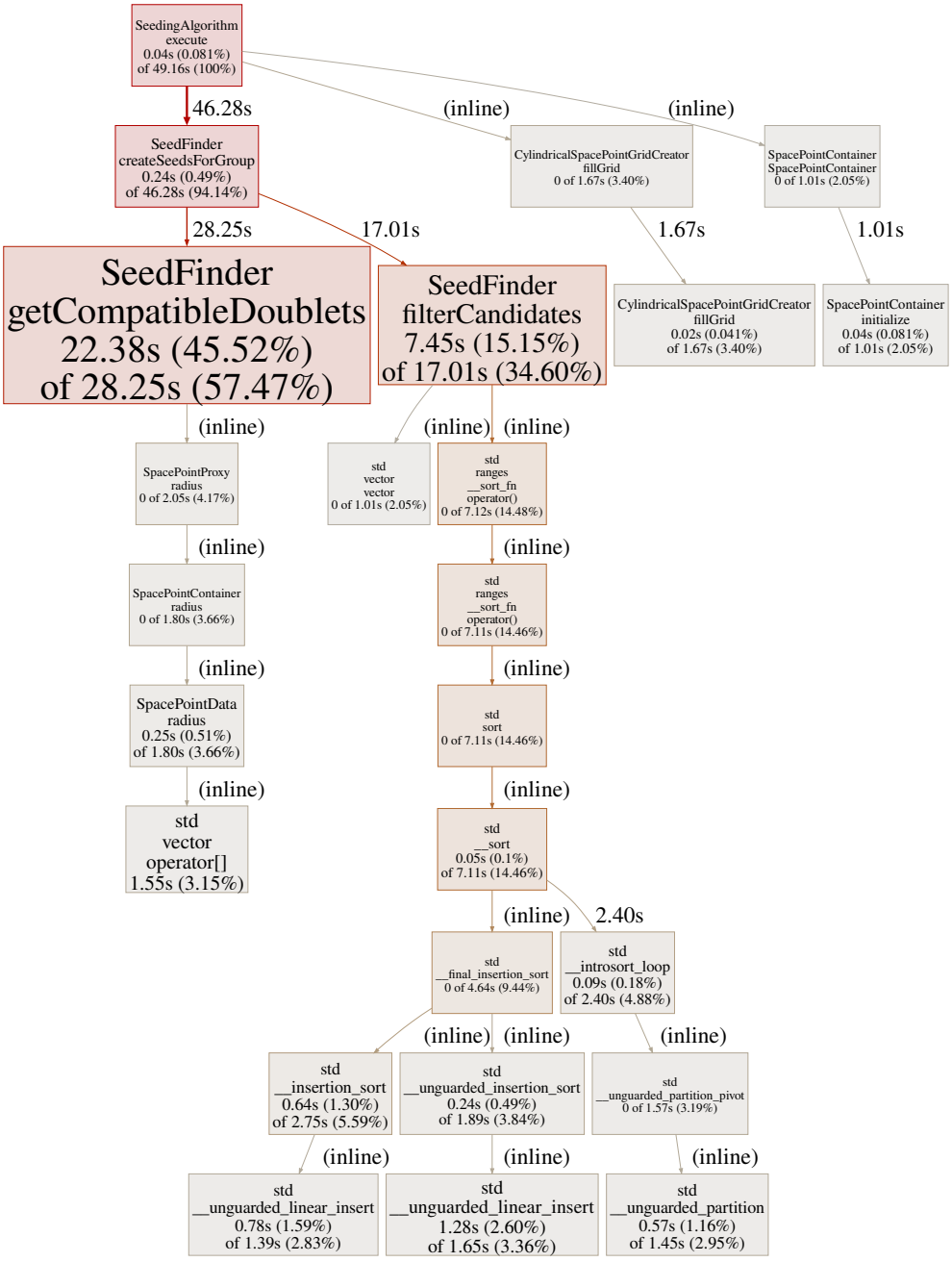


Figure B.3: Callgraph for `Seeding` on strip data.

B.2 Detailed Seeding2 Profiling

This appendix supplements the Seeding2 profiling results in Section 6.2 and collects the artefacts referenced in the baseline comparison.

Seeding2 strip-data flamegraph

The strip-data flamegraph shows the same structure as the pixel case: createDoubletsImpl dominates, while triplet construction and sorting appear as smaller clusters. The filterCandidates tower present in Seeding is absent.

Seeding2 callgraphs

The callgraphs show the refactored structure of Seeding2: the filterCandidates subtree is removed, and sorting appears as a separate branch under sortByCotTheta.

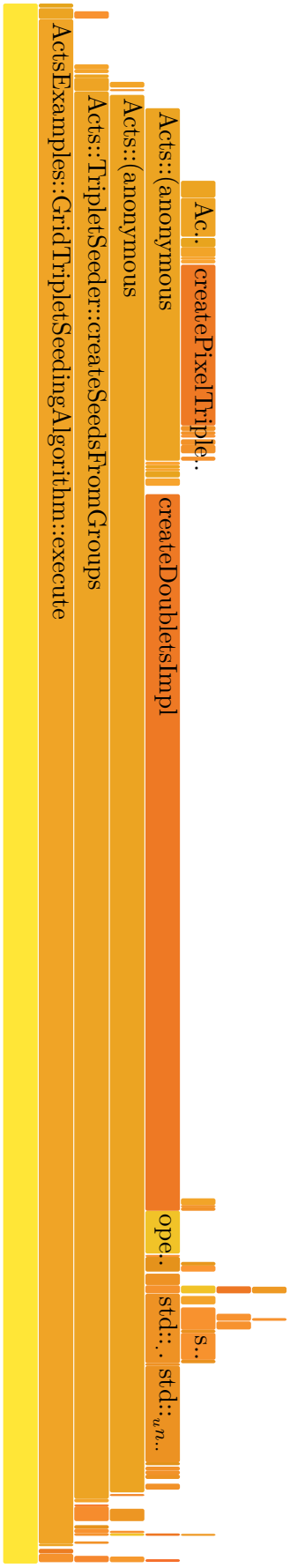


Figure B.4: CPU flamegraph for `Seeding2` on strip data. Bar width denotes the fraction of on-CPU samples; colour is hash-based.

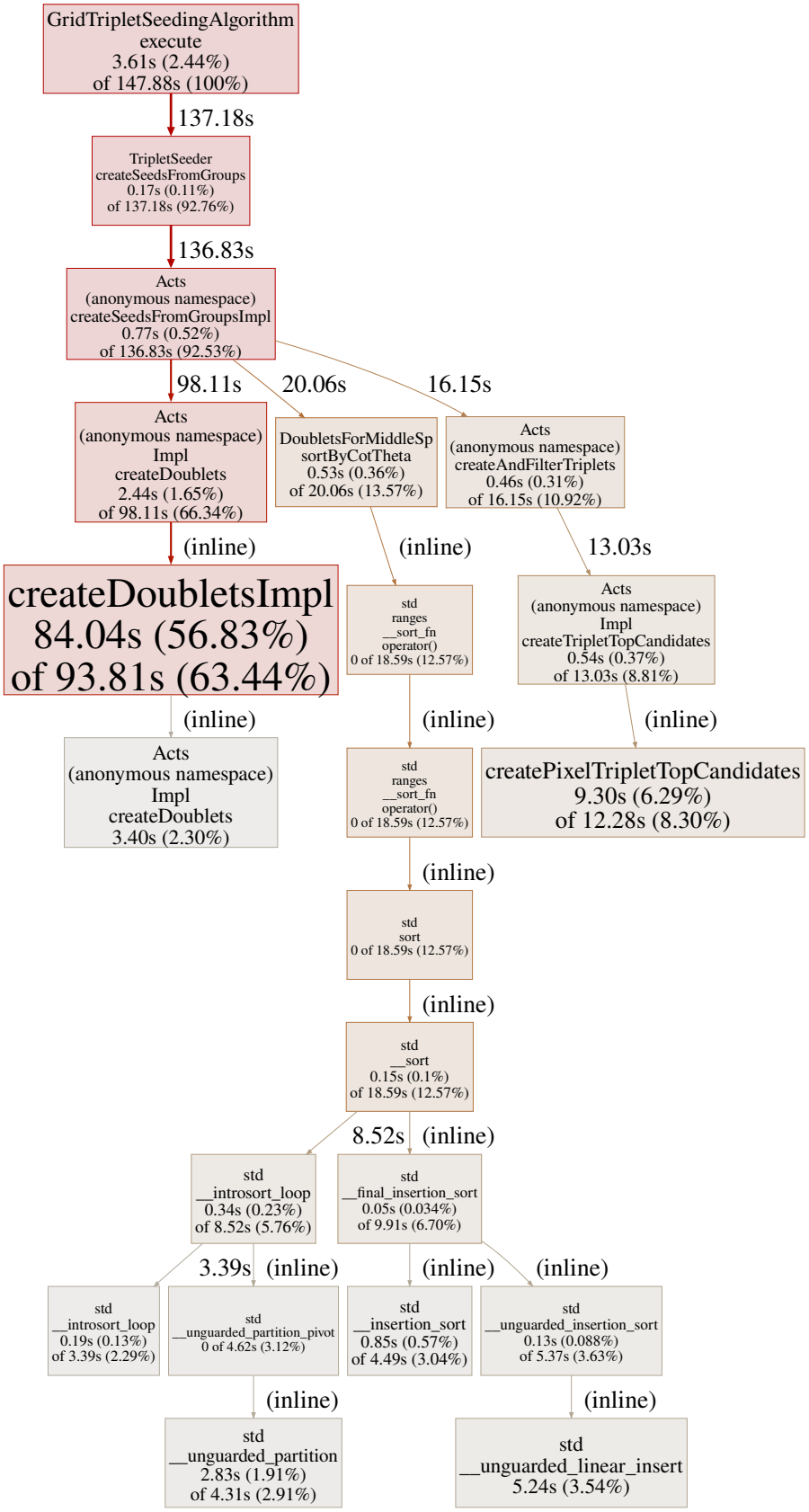


Figure B.5: Callgraph for Seeding2 on pixel data.

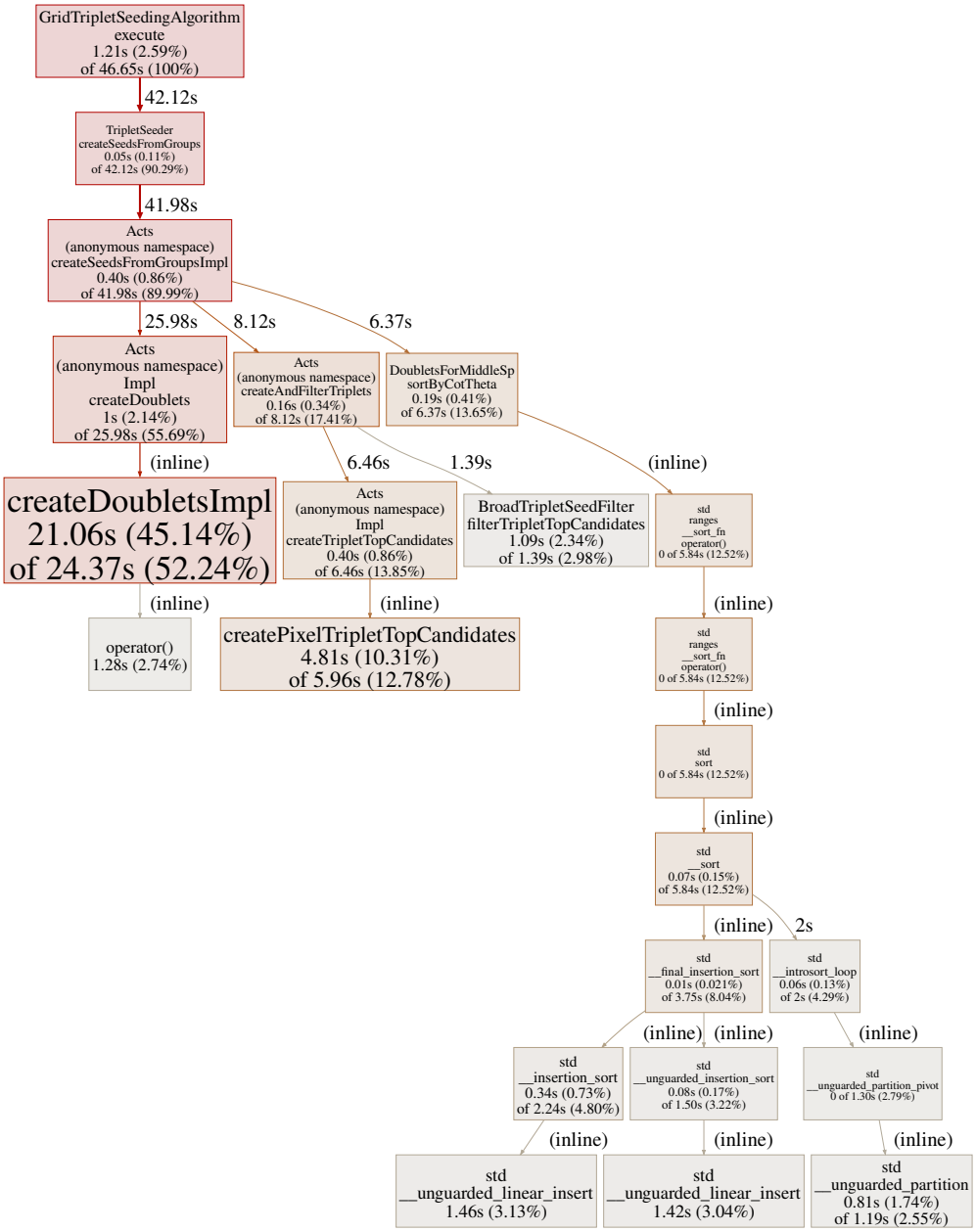


Figure B.6: Callgraph for `Seeding2` on strip data.

Seeding2 line-level annotations

The listings below give the line-level profiler annotations for the four hotspot hypotheses H1–H4 in both pixel and strip configurations. The first two columns report flat and cumulative time; dots indicate values below the rounding threshold.

H1 — createDoubletsImpl

Listing B.1: Pixel: createDoubletsImpl line-level annotation (total 1.55 s).

```
Total: 93.81s
ROUTINE ======================== createDoubletsImpl in /storage/thomaaks/acts-seeding-optimization
    ↪ /ACTS/source/Core/src/Seeding2/DoubletSeedFinder.cpp
    84.04s     93.81s (flat, cum)   100% of Total
         .          .     41:  template <typename CandidateSps>
         .          .     42:  void createDoubletsImpl(const ConstSpacePointProxy2& middleSp,
         .          .     43:                          const MiddleSpInfo& middleSpInfo,
         .          .     44:                          CandidateSps& candidateSps,
         .          .     45:                          DoubletsForMiddleSp& compatibleDoublets)
    ↪ const {
      20ms       20ms     46:    const float impactMax =
         .          .     47:        isBottomCandidate ? -m_cfg.impactMax : m_cfg.impactMax;
         .          .     48:
      20ms      450ms     49:    const float xM = middleSp.xy()[0];
      10ms       10ms     50:    const float yM = middleSp.xy()[1];
      70ms       70ms     51:    const float zM = middleSp.zr()[0];
     250ms      250ms     52:    const float rM = middleSp.zr()[1];
         .      140ms     53:    const float varianceZM = middleSp.varianceZ();
         .       60ms     54:    const float varianceRM = middleSp.varianceR();
         .          .     55:
         .          .     56:    // equivalent to impactMax / (rM * rM);
     210ms      210ms     57:    const float vIPAbs = impactMax * middleSpInfo.uIP2;
         .          .     58:
         .          .     59:    const auto outsideRangeCheck = [](const float value, const float
    ↪ min,
         .          .     60:                                      const float max) {
         .          .     61:      // intentionally using `|` after profiling. faster due to
    ↪ better branch
         .          .     62:      // prediction
         .          .     63:      return static_cast<bool>(static_cast<int>(value < min) |
         .          .     64:                               static_cast<int>(value > max));
         .          .     65:    };
         .          .     66:
         .          .     67:    const auto calculateError = [&](float varianceZ0, float
    ↪ varianceR0,
         .          .     68:                                    float iDeltaR2, float cotTheta) {
         .          .     69:      return iDeltaR2 * ((varianceZM + varianceZ0) +
         .          .     70:                         (cotTheta * cotTheta) * (varianceRM +
    ↪ varianceR0));
         .          .     71:    };
         .          .     72:
         .          .     73:    if constexpr (sortedByR) {
         .          .     74:      // find the first SP inside the radius region of interest and
    ↪ update
         .          .     75:      // the iterator so we don't need to look at the other SPs again
         .          .     76:      std::uint32_t offset = 0;
     260ms      350ms     77:      for (ConstSpacePointProxy2 otherSp : candidateSps) {
         .          .     78:        if constexpr (isBottomCandidate) {
         .          .     79:          // if r-distance is too big, try next SP in bin
     330ms      330ms     80:          if (rM - otherSp.zr()[1] <= m_cfg.deltaRMax) {
         .          .     81:            break;
         .          .     82:          }
         .          .     83:        } else {
         .          .     84:          // if r-distance is too small, try next SP in bin
     1.08s      1.08s     85:          if (otherSp.zr()[1] - rM >= m_cfg.deltaRMin) {
         .          .     86:            break;
         .          .     87:          }
         .          .     88:        }
```

```
     .          .     89:
   480ms      480ms     90:       ++offset;
     .          .     91:     }
     .      150ms     92:     candidateSps = candidateSps.subrange(offset);
     .          .     93:   }
     .          .     94:
     .          .     95:   const SpacePointContainer2& container = candidateSps.container();
  15.48s     16.33s     96:   for (auto [index0, xy0, zr0, varianceZ0, varianceR0] :
↪ candidateSps.zip(
     .          .     97:            container.xyColumn(), container.zrColumn(),
     .          .     98:            container.varianceZColumn(), container.varianceRColumn()
↪ )) {
   2.68s      2.68s     99:     const float x0 = xy0[0];
   240ms      240ms    100:     const float y0 = xy0[1];
   2.08s      2.08s    101:     const float z0 = zr0[0];
     .          .    102:     const float r0 = zr0[1];
     .          .    103:
     .          .    104:     float deltaR = 0;
     .          .    105:     if constexpr (isBottomCandidate) {
   230ms      230ms    106:       deltaR = rM - r0;
     .          .    107:
     .          .    108:       if constexpr (sortedByR) {
     .          .    109:         // if r-distance is too small we are done
   2.89s      2.89s    110:         if (deltaR < m_cfg.deltaRMin) {
     .          .    111:           break;
     .          .    112:         }
     .          .    113:       }
     .          .    114:     } else {
   360ms      360ms    115:       deltaR = r0 - rM;
     .          .    116:
     .          .    117:       if constexpr (sortedByR) {
     .          .    118:         // if r-distance is too big we are done
   3.13s      3.13s    119:         if (deltaR > m_cfg.deltaRMax) {
     .          .    120:           break;
     .          .    121:         }
     .          .    122:       }
     .          .    123:     }
     .          .    124:
     .          .    125:     if constexpr (!sortedByR) {
     .          .    126:       if (outsideRangeCheck(deltaR, m_cfg.deltaRMin, m_cfg.
↪ deltaRMax)) {
     .          .    127:         continue;
     .          .    128:       }
     .          .    129:     }
     .          .    130:
     .          .    131:     float deltaZ = 0;
     .          .    132:     if constexpr (isBottomCandidate) {
   540ms      540ms    133:       deltaZ = zM - z0;
     .          .    134:     } else {
   690ms      690ms    135:       deltaZ = z0 - zM;
     .          .    136:     }
     .          .    137:
   5.88s      6.12s    138:     if (outsideRangeCheck(deltaZ, m_cfg.deltaZMin, m_cfg.deltaZMax)
↪ ) {
     .          .    139:       continue;
     .          .    140:     }
     .          .    141:
     .          .    142:     // the longitudinal impact parameter zOrigin is defined as (zM
↪ - rM *
     .          .    143:     // cotTheta) where cotTheta is the ratio Z/R (forward angle) of
↪  space
     .          .    144:     // point duplet but instead we calculate (zOrigin * deltaR) and
↪  multiply
     .          .    145:     // collisionRegion by deltaR to avoid divisions
   2.91s      2.91s    146:     const float zOriginTimesDeltaR = zM * deltaR - rM * deltaZ;
     .          .    147:     // check if duplet origin on z axis within collision region
  17.99s     17.99s    148:     if (outsideRangeCheck(zOriginTimesDeltaR,
     .          .    149:                           m_cfg.collisionRegionMin * deltaR,
     .          .    150:                           m_cfg.collisionRegionMax * deltaR)) {
     .          .    151:       continue;
```

```
     .          .    152:      }
     .          .    153:
     .          .    154:      // if interactionPointCut is false we apply z cuts before
↪ coordinate
     .          .    155:      // transformation to avoid unnecessary calculations. If
     .          .    156:      // interactionPointCut is true we apply the curvature cut first
↪  because it
     .          .    157:      // is more frequent but requires the coordinate transformation
     .          .    158:      if constexpr (!interactionPointCut) {
     .          .    159:        // check if duplet cotTheta is within the region of interest
     .          .    160:        // cotTheta is defined as (deltaZ / deltaR) but instead we
↪ multiply
     .          .    161:        // cotThetaMax by deltaR to avoid division
     .          .    162:        if (outsideRangeCheck(deltaZ, -m_cfg.cotThetaMax * deltaR,
     .          .    163:                              m_cfg.cotThetaMax * deltaR)) {
     .          .    164:          continue;
     .          .    165:        }
     .          .    166:
     .          .    167:        // transform SP coordinates to the u-v reference frame
     .          .    168:        const float deltaX = xO - xM;
     .          .    169:        const float deltaY = yO - yM;
     .          .    170:
     .          .    171:        const float xNewFrame =
     .          .    172:            deltaX * middleSpInfo.cosPhiM + deltaY * middleSpInfo.
↪ sinPhiM;
     .          .    173:        const float yNewFrame =
     .          .    174:            deltaY * middleSpInfo.cosPhiM - deltaX * middleSpInfo.
↪ sinPhiM;
     .          .    175:
     .          .    176:        const float deltaR2 = deltaX * deltaX + deltaY * deltaY;
     .          .    177:        const float iDeltaR2 = 1 / deltaR2;
     .          .    178:
     .          .    179:        const float uT = xNewFrame * iDeltaR2;
     .          .    180:        const float vT = yNewFrame * iDeltaR2;
     .          .    181:
     .          .    182:        const float iDeltaR = std::sqrt(iDeltaR2);
     .          .    183:        const float cotTheta = deltaZ * iDeltaR;
     .          .    184:
     .          .    185:        const float er =
     .          .    186:            calculateError(varianceZO, varianceRO, iDeltaR2, cotTheta
↪ );
     .          .    187:
     .          .    188:        // fill output vectors
     .          .    189:        compatibleDoublets.emplace_back(indexO, cotTheta, iDeltaR, er
↪ , uT, vT,
     .          .    190:                                        xNewFrame, yNewFrame);
     .          .    191:        continue;
     .          .    192:      }
     .          .    193:
     .          .    194:      // transform SP coordinates to the u-v reference frame
 610ms      610ms    195:      const float deltaX = xO - xM;
 4.43s      4.43s    196:      const float deltaY = yO - yM;
     .          .    197:
  20ms       20ms    198:      const float xNewFrame =
 500ms      500ms    199:          deltaX * middleSpInfo.cosPhiM + deltaY * middleSpInfo.
↪ sinPhiM;
 370ms      370ms    200:      const float yNewFrame =
 130ms      130ms    201:          deltaY * middleSpInfo.cosPhiM - deltaX * middleSpInfo.
↪ sinPhiM;
     .          .    202:
 540ms      540ms    203:      const float deltaR2 = deltaX * deltaX + deltaY * deltaY;
 380ms      380ms    204:      const float iDeltaR2 = 1 / deltaR2;
     .          .    205:
 220ms      220ms    206:      const float uT = xNewFrame * iDeltaR2;
 1.38s      1.38s    207:      const float vT = yNewFrame * iDeltaR2;
     .          .    208:
     .          .    209:      // We check the interaction point by evaluating the minimal
↪ distance
     .          .    210:      // between the origin and the straight line connecting the two
↪ points in
```

```
     .          .    211:      // the doublets. Using a geometric similarity, the Im is given
↪ by
     .          .    212:      // yNewFrame * rM / deltaR > config.impactMax
     .          .    213:      // However, we make here an approximation of the impact
↪ parameter
     .          .    214:      // which is valid under the assumption yNewFrame / xNewFrame is
↪  small
     .          .    215:      // The correct computation would be:
     .          .    216:      // yNewFrame * yNewFrame * rM * rM > config.impactMax *
     .          .    217:      // config.impactMax * deltaR2
 2.94s      2.94s    218:      if (std::abs(rM * yNewFrame) > impactMax * xNewFrame) {
     .          .    219:        // in the rotated frame the interaction point is positioned
↪ at x = -rM
     .          .    220:        // and y ~= impactParam
 1.51s      1.51s    221:        const float vIP = (yNewFrame > 0) ? -vIPAbs : vIPAbs;
     .          .    222:
     .          .    223:        // we can obtain aCoef as the slope dv/du of the linear
↪ function,
     .          .    224:        // estimated using du and dv between the two SP bCoef is
↪ obtained by
     .          .    225:        // inserting aCoef into the linear equation
 340ms      340ms    226:        const float aCoef = (vT - vIP) / (uT - middleSpInfo.uIP);
 3.22s      3.22s    227:        const float bCoef = vIP - aCoef * middleSpInfo.uIP;
     .          .    228:        // the distance of the straight line from the origin (radius
↪ of the
     .          .    229:        // circle) is related to aCoef and bCoef by d^2 = bCoef^2 /
↪ (1 +
     .          .    230:        // aCoef^2) = 1 / (radius^2) and we can apply the cut on the
↪ curvature
 3.92s      3.92s    231:        if ((bCoef * bCoef) * m_cfg.minHelixDiameter2 > 1 + aCoef *
↪ aCoef) {
     .          .    232:          continue;
     .          .    233:        }
     .          .    234:      }
     .          .    235:
     .          .    236:      // check if duplet cotTheta is within the region of interest
     .          .    237:      // cotTheta is defined as (deltaZ / deltaR) but instead we
↪ multiply
     .          .    238:      // cotThetaMax by deltaR to avoid division
 900ms      3.42s    239:      if (outsideRangeCheck(deltaZ, -m_cfg.cotThetaMax * deltaR,
     .          .    240:                            m_cfg.cotThetaMax * deltaR)) {
     .          .    241:        continue;
     .          .    242:      }
     .          .    243:
     .      200ms    244:      const float iDeltaR = std::sqrt(iDeltaR2);
 2.32s      2.32s    245:      const float cotTheta = deltaZ * iDeltaR;
     .          .    246:
     .          .    247:      // discard doublets based on experiment specific cuts
     .          .    248:      if constexpr (experimentCuts) {
 690ms      2.88s    249:        if (!m_cfg.experimentCuts(middleSp, container[indexO],
↪ cotTheta,
     .          .    250:                                  isBottomCandidate)) {
     .          .    251:          continue;
     .          .    252:        }
     .          .    253:      }
     .          .    254:
     .          .    255:      const float er =
     .      1.74s    256:          calculateError(varianceZO, varianceRO, iDeltaR2, cotTheta);
     .          .    257:
     .          .    258:      // fill output vectors
 1.79s      2.95s    259:      compatibleDoublets.emplace_back(indexO, cotTheta, iDeltaR, er,
↪ uT, vT,
     .          .    260:                                      xNewFrame, yNewFrame);
     .          .    261:    }
     .          .    262:  }
     .          .    263:
     .          .    264:  void createDoublets(const ConstSpacePointProxy2& middleSp,
```

Listing B.2: Strip: `createDoubletsImpl` line-level annotation (total 420 ms).

```
Total: 24.37s
ROUTINE ======================== createDoubletsImpl in /storage/thomaaks/acts-seeding-optimization
    ↪ /ACTS/source/Core/src/Seeding2/DoubletSeedFinder.cpp
    21.06s     24.37s (flat, cum)   100% of Total
         .          .     44:                          CandidateSps& candidateSps,
         .          .     45:                          DoubletsForMiddleSp& compatibleDoublets)
    ↪ const {
         .          .     46:    const float impactMax =
         .          .     47:        isBottomCandidate ? -m_cfg.impactMax : m_cfg.impactMax;
         .          .     48:
         .       60ms     49:    const float xM = middleSp.xy()[0];
         .          .     50:    const float yM = middleSp.xy()[1];
      20ms       30ms     51:    const float zM = middleSp.zr()[0];
     150ms      150ms     52:    const float rM = middleSp.zr()[1];
         .       40ms     53:    const float varianceZM = middleSp.varianceZ();
         .      220ms     54:    const float varianceRM = middleSp.varianceR();
         .          .     55:
         .          .     56:    // equivalent to impactMax / (rM * rM);
         .          .     57:    const float vIPAbs = impactMax * middleSpInfo.uIP2;
         .          .     58:
         .          .     59:    const auto outsideRangeCheck = [](const float value, const float
    ↪ min,
         .          .     60:                                      const float max) {
         .          .     61:      // intentionally using `|` after profiling. faster due to
    ↪ better branch
         .          .     62:      // prediction
         .          .     63:      return static_cast<bool>(static_cast<int>(value < min) |
         .          .     64:                               static_cast<int>(value > max));
         .          .     65:    };
         .          .     66:
         .          .     67:    const auto calculateError = [&](float varianceZO, float
    ↪ varianceRO,
         .          .     68:                                    float iDeltaR2, float cotTheta) {
         .          .     69:      return iDeltaR2 * ((varianceZM + varianceZO) +
         .          .     70:                         (cotTheta * cotTheta) * (varianceRM +
    ↪ varianceRO));
         .          .     71:    };
         .          .     72:
         .          .     73:    if constexpr (sortedByR) {
         .          .     74:      // find the first SP inside the radius region of interest and
    ↪ update
         .          .     75:      // the iterator so we don't need to look at the other SPs again
         .          .     76:      std::uint32_t offset = 0;
      80ms      130ms     77:      for (ConstSpacePointProxy2 otherSp : candidateSps) {
         .          .     78:        if constexpr (isBottomCandidate) {
         .          .     79:          // if r-distance is too big, try next SP in bin
     130ms      130ms     80:          if (rM - otherSp.zr()[1] <= m_cfg.deltaRMax) {
         .          .     81:            break;
         .          .     82:          }
         .          .     83:        } else {
         .          .     84:          // if r-distance is too small, try next SP in bin
     360ms      360ms     85:          if (otherSp.zr()[1] - rM >= m_cfg.deltaRMin) {
         .          .     86:            break;
         .          .     87:          }
         .          .     88:        }
         .          .     89:
     140ms      140ms     90:        ++offset;
         .          .     91:      }
         .       40ms     92:      candidateSps = candidateSps.subrange(offset);
         .          .     93:    }
         .          .     94:
         .          .     95:    const SpacePointContainer2& container = candidateSps.container();
     5.22s      5.42s     96:    for (auto [indexO, xyO, zrO, varianceZO, varianceRO] :
    ↪ candidateSps.zip(
         .          .     97:             container.xyColumn(), container.zrColumn(),
         .          .     98:             container.varianceZColumn(), container.varianceRColumn()
    ↪ )) {
         .          .     99:      const float xO = xyO[0];
         .          .    100:      const float yO = xyO[1];
     380ms      380ms    101:      const float zO = zrO[0];
```

```
      .          .    102:      const float r0 = zr0[1];
      .          .    103:
      .          .    104:      float deltaR = 0;
      .          .    105:      if constexpr (isBottomCandidate) {
  190ms      190ms    106:        deltaR = rM - r0;
      .          .    107:
      .          .    108:        if constexpr (sortedByR) {
      .          .    109:          // if r-distance is too small we are done
  1.16s      1.16s    110:          if (deltaR < m_cfg.deltaRMin) {
      .          .    111:            break;
      .          .    112:          }
      .          .    113:        }
      .          .    114:      } else {
   30ms       30ms    115:        deltaR = r0 - rM;
      .          .    116:
      .          .    117:        if constexpr (sortedByR) {
      .          .    118:          // if r-distance is too big we are done
  1.22s      1.22s    119:          if (deltaR > m_cfg.deltaRMax) {
      .          .    120:            break;
      .          .    121:          }
      .          .    122:        }
      .          .    123:      }
      .          .    124:
      .          .    125:      if constexpr (!sortedByR) {
      .          .    126:        if (outsideRangeCheck(deltaR, m_cfg.deltaRMin, m_cfg.
↪ deltaRMax)) {
      .          .    127:          continue;
      .          .    128:        }
      .          .    129:      }
      .          .    130:
      .          .    131:      float deltaZ = 0;
      .          .    132:      if constexpr (isBottomCandidate) {
  230ms      230ms    133:        deltaZ = zM - z0;
      .          .    134:      } else {
  140ms      140ms    135:        deltaZ = z0 - zM;
      .          .    136:      }
      .          .    137:
   2.42s      2.58s   138:      if (outsideRangeCheck(deltaZ, m_cfg.deltaZMin, m_cfg.deltaZMax)
↪ ) {
      .          .    139:        continue;
      .          .    140:      }
      .          .    141:
      .          .    142:      // the longitudinal impact parameter zOrigin is defined as (zM
↪ - rM *
      .          .    143:      // cotTheta) where cotTheta is the ratio Z/R (forward angle) of
↪  space
      .          .    144:      // point duplet but instead we calculate (zOrigin * deltaR) and
↪  multiply
      .          .    145:      // collisionRegion by deltaR to avoid divisions
  400ms      400ms    146:      const float zOriginTimesDeltaR = zM * deltaR - rM * deltaZ;
      .          .    147:      // check if duplet origin on z axis within collision region
   5.41s      5.41s   148:      if (outsideRangeCheck(zOriginTimesDeltaR,
      .          .    149:                            m_cfg.collisionRegionMin * deltaR,
      .          .    150:                            m_cfg.collisionRegionMax * deltaR)) {
      .          .    151:        continue;
      .          .    152:      }
      .          .    153:
      .          .    154:      // if interactionPointCut is false we apply z cuts before
↪ coordinate
      .          .    155:      // transformation to avoid unnecessary calculations. If
      .          .    156:      // interactionPointCut is true we apply the curvature cut first
↪  because it
      .          .    157:      // is more frequent but requires the coordinate transformation
      .          .    158:      if constexpr (!interactionPointCut) {
      .          .    159:        // check if duplet cotTheta is within the region of interest
      .          .    160:        // cotTheta is defined as (deltaZ / deltaR) but instead we
↪ multiply
      .          .    161:        // cotThetaMax by deltaR to avoid division
  360ms      1.27s    162:        if (outsideRangeCheck(deltaZ, -m_cfg.cotThetaMax * deltaR,
      .          .    163:                              m_cfg.cotThetaMax * deltaR)) {
```

```
        .          .    164:          continue;
        .          .    165:        }
        .          .    166:
        .          .    167:        // transform SP coordinates to the u-v reference frame
     20ms       20ms    168:        const float deltaX = x0 - xM;
    140ms      140ms    169:        const float deltaY = y0 - yM;
        .          .    170:
     10ms       10ms    171:        const float xNewFrame =
    100ms      100ms    172:            deltaX * middleSpInfo.cosPhiM + deltaY * middleSpInfo.
    ↪ sinPhiM;
    130ms      130ms    173:        const float yNewFrame =
     50ms       50ms    174:            deltaY * middleSpInfo.cosPhiM - deltaX * middleSpInfo.
    ↪ sinPhiM;
        .          .    175:
    160ms      160ms    176:        const float deltaR2 = deltaX * deltaX + deltaY * deltaY;
     60ms       60ms    177:        const float iDeltaR2 = 1 / deltaR2;
        .          .    178:
    680ms      680ms    179:        const float uT = xNewFrame * iDeltaR2;
     70ms       70ms    180:        const float vT = yNewFrame * iDeltaR2;
        .          .    181:
        .      130ms    182:        const float iDeltaR = std::sqrt(iDeltaR2);
    920ms      920ms    183:        const float cotTheta = deltaZ * iDeltaR;
        .          .    184:
        .          .    185:        const float er =
        .      1.12s    186:            calculateError(varianceZ0, varianceR0, iDeltaR2, cotTheta
    ↪ );
        .          .    187:
        .          .    188:        // fill output vectors
    680ms      1.05s    189:        compatibleDoublets.emplace_back(index0, cotTheta, iDeltaR, er
    ↪ , uT, vT,
        .          .    190:                                        xNewFrame, yNewFrame);
        .          .    191:        continue;
        .          .    192:      }
        .          .    193:
        .          .    194:      // transform SP coordinates to the u-v reference frame
```

H2 — `createPixelTripletTopCandidates`

Listing B.3: Pixel: `createPixelTripletTopCandidates` line-level annotation (total 250 ms).

```
Total: 12.28s
ROUTINE ======================== createPixelTripletTopCandidates in /storage/thomaaks/acts-seeding
    ↪ -optimization/ACTS/source/Core/src/Seeding2/TripletSeedFinder.cpp
     9.30s     12.28s (flat, cum)   100% of Total
        .          .     94:  template <typename TopDoublets>
        .          .     95:  void createPixelTripletTopCandidates(
        .          .     96:      const ConstSpacePointProxy2& spM,
        .          .     97:      const DoubletsForMiddleSp::Proxy& bottomDoublet, TopDoublets&
    ↪ topDoublets,
        .          .     98:      TripletTopCandidates& tripletTopCandidates) const {
     20ms       20ms     99:    const float rM = spM.zr()[1];
        .      220ms    100:    const float varianceZM = spM.varianceZ();
        .      210ms    101:    const float varianceRM = spM.varianceR();
        .          .    102:
        .          .    103:    // Reserve enough space, in case current capacity is too little
        .      440ms    104:    tripletTopCandidates.reserve(tripletTopCandidates.size() +
        .          .    105:                                 topDoublets.size());
        .          .    106:
        .       90ms    107:    const float cotThetaB = bottomDoublet.cotTheta();
        .      200ms    108:    const float erB = bottomDoublet.er();
        .          .    109:    const float iDeltaRB = bottomDoublet.iDeltaR();
        .      600ms    110:    const float Ub = bottomDoublet.u();
        .       30ms    111:    const float Vb = bottomDoublet.v();
        .          .    112:
        .          .    113:    // 1+(cot^2(theta)) = 1/sin^2(theta)
        .          .    114:    const float iSinTheta2 = 1 + cotThetaB * cotThetaB;
        .          .    115:    const float sigmaSquaredPtDependent = iSinTheta2 * m_cfg.
    ↪ sigmapT2perRadius;
```

```
     .          .    116:    // calculate max scattering for min momentum at the seed's theta
↪ angle
     .          .    117:    // scaling scatteringAngle^2 by sin^2(theta) to convert pT^2 to p
↪ ^2
     .          .    118:    // accurate would be taking 1/atan(thetaBottom)-1/atan(thetaTop)
↪ <
     .          .    119:    // scattering
     .          .    120:    // but to avoid trig functions we approximate cot by scaling by
     .          .    121:    // 1/sin^4(theta)
     .          .    122:    // resolving with pT to p scaling --> only divide by sin^2(theta)
     .          .    123:    // max approximation error for allowed scattering angles of 0.04
↪ rad at
     .          .    124:    // eta=infinity: ~8.5%
 310ms      310ms    125:    const float scatteringInRegion2 = m_cfg.multipleScattering2 *
↪ iSinTheta2;
     .          .    126:
     .          .    127:    std::size_t topDoubletOffset = 0;
 250ms      260ms    128:    for (auto [topDoublet, topDoubletIndex] :
     .      440ms    129:         zip(topDoublets, std::ranges::iota_view<std::size_t, std::
↪ size_t>(
     .          .    130:                                 0, topDoublets.size()))) {
     .      140ms    131:      const SpacePointIndex2 spT = topDoublet.spacePointIndex();
     .          .    132:      const float cotThetaT = topDoublet.cotTheta();
     .          .    133:
     .          .    134:      // use geometric average
 250ms      250ms    135:      const float cotThetaAvg2 = cotThetaB * cotThetaT;
     .          .    136:
     .          .    137:      // add errors of spB-spM and spM-spT pairs and add the
↪ correlation term
     .          .    138:      // for errors on spM
 470ms      850ms    139:      const float error2 = topDoublet.er() + erB +
 470ms      470ms    140:                           2 * (cotThetaAvg2 * varianceRM +
↪ varianceZM) *
 500ms      500ms    141:                                iDeltaRB * topDoublet.iDeltaR();
     .          .    142:
  60ms       60ms    143:      const float deltaCotTheta = cotThetaB - cotThetaT;
  20ms       20ms    144:      const float deltaCotTheta2 = deltaCotTheta * deltaCotTheta;
     .          .    145:
     .          .    146:      // Apply a cut on the compatibility between the r-z slope of
↪ the two
     .          .    147:      // seed segments. This is done by comparing the squared
↪ difference
     .          .    148:      // between slopes, and comparing to the squared uncertainty in
↪ this
     .          .    149:      // difference - we keep a seed if the difference is compatible
↪ within
     .          .    150:      // the assumed uncertainties. The uncertainties get
↪ contribution from
     .          .    151:      // the  space-point-related squared error (error2) and a
↪ scattering term
     .          .    152:      // calculated assuming the minimum pt we expect to reconstruct
     .          .    153:      // (scatteringInRegion2). This assumes gaussian error
↪ propagation which
     .          .    154:      // allows just adding the two errors if they are uncorrelated (
↪ which is
     .          .    155:      // fair for scattering and measurement uncertainties)
 1.91s      1.91s    156:      if (deltaCotTheta2 > error2 + scatteringInRegion2) {
     .          .    157:        if constexpr (sortedByCotTheta) {
     .          .    158:          // skip top SPs based on cotTheta sorting when producing
↪ triplets
     .          .    159:          // break if cotTheta from bottom SP < cotTheta from top SP
↪ because
     .          .    160:          // the SP are sorted by cotTheta
 810ms      810ms    161:          if (cotThetaB < cotThetaT) {
     .          .    162:            break;
     .          .    163:          }
  50ms       50ms    164:          topDoubletOffset = topDoubletIndex + 1;
     .          .    165:        }
     .          .    166:        continue;
     .          .    167:      }
```

```
     .          .    168:
  1.22s      1.24s    169:      const float dU = topDoublet.u() - Ub;
     .          .    170:      // protects against division by 0
  460ms      460ms    171:      if (dU == 0) {
     .          .    172:        continue;
     .          .    173:      }
     .          .    174:      // A and B are evaluated as a function of the circumference
↪ parameters
     .          .    175:      // x_0 and y_0
     .          .    176:      const float A = (topDoublet.v() - Vb) / dU;
  170ms      170ms    177:      const float S2 = 1 + A * A;
  400ms      400ms    178:      const float B = Vb - A * Ub;
     .          .    179:      const float B2 = B * B;
     .          .    180:
     .          .    181:      // sqrt(S2)/B = 2 * helixradius
     .          .    182:      // calculated radius must not be smaller than minimum radius
  440ms      440ms    183:      if (S2 < B2 * m_cfg.minHelixDiameter2) {
     .          .    184:        continue;
     .          .    185:      }
     .          .    186:
     .          .    187:      // refinement of the cut on the compatibility between the r-z
↪ slope of
     .          .    188:      // the two seed segments using a scattering term scaled by the
↪ actual
     .          .    189:      // measured pT (p2scatterSigma)
   10ms       10ms    190:      const float iHelixDiameter2 = B2 / S2;
     .          .    191:      // convert p(T) to p scaling by sin^2(theta) AND scale by 1/sin
↪ ^4(theta)
     .          .    192:      // from rad to deltaCotTheta
  430ms      430ms    193:      const float p2scatterSigma = iHelixDiameter2 *
↪ sigmaSquaredPtDependent;
     .          .    194:      // if deltaTheta larger than allowed scattering for calculated
↪ pT, skip
  300ms      300ms    195:      if (deltaCotTheta2 > error2 + p2scatterSigma) {
     .          .    196:        if constexpr (sortedByCotTheta) {
  220ms      220ms    197:          if (cotThetaB < cotThetaT) {
     .          .    198:            break;
     .          .    199:          }
     .          .    200:          topDoubletOffset = topDoubletIndex;
     .          .    201:        }
     .          .    202:        continue;
     .          .    203:      }
     .          .    204:
     .          .    205:      // A and B allow calculation of impact params in U/V plane with
↪  linear
     .          .    206:      // function
     .          .    207:      // (in contrast to having to solve a quadratic function in x/y
↪ plane)
  210ms      230ms    208:      const float im = std::abs((A - B * rM) * rM);
   20ms       20ms    209:      if (im > m_cfg.impactMax) {
     .          .    210:        continue;
     .          .    211:      }
     .          .    212:
     .          .    213:      // inverse diameter is signed depending on if the curvature is
     .          .    214:      // positive/negative in phi
  200ms      250ms    215:      tripletTopCandidates.emplace_back(spT, B / std::sqrt(S2), im);
     .          .    216:    }  // loop on tops
     .          .    217:
     .          .    218:    if constexpr (sortedByCotTheta) {
     .          .    219:      // remove the top doublets that were skipped due to cotTheta
↪ sorting
  100ms      230ms    220:      topDoublets = topDoublets.subrange(topDoubletOffset);
     .          .    221:    }
     .          .    222:  }
     .          .    223:
     .          .    224:  template <typename TopDoublets>
     .          .    225:  void createStripTripletTopCandidates(
```

Listing B.4: Strip: `createPixelTripletTopCandidates` line-level annotation (total 60 ms).

```
Total: 5.96s
ROUTINE ======================== createPixelTripletTopCandidates in /storage/thomaaks/acts-seeding
    ↪ -optimization/ACTS/source/Core/src/Seeding2/TripletSeedFinder.cpp
     4.81s      5.96s (flat, cum)   100% of Total
         .          .     94:  template <typename TopDoublets>
         .          .     95:  void createPixelTripletTopCandidates(
         .          .     96:      const ConstSpacePointProxy2& spM,
         .          .     97:      const DoubletsForMiddleSp::Proxy& bottomDoublet, TopDoublets&
    ↪ topDoublets,
         .          .     98:      TripletTopCandidates& tripletTopCandidates) const {
      10ms       10ms     99:    const float rM = spM.zr()[1];
         .      110ms    100:    const float varianceZM = spM.varianceZ();
         .       80ms    101:    const float varianceRM = spM.varianceR();
         .          .    102:
         .          .    103:    // Reserve enough space, in case current capacity is too little
         .      160ms    104:    tripletTopCandidates.reserve(tripletTopCandidates.size() +
         .          .    105:                                 topDoublets.size());
         .          .    106:
         .       10ms    107:    const float cotThetaB = bottomDoublet.cotTheta();
         .      120ms    108:    const float erB = bottomDoublet.er();
         .          .    109:    const float iDeltaRB = bottomDoublet.iDeltaR();
         .      270ms    110:    const float Ub = bottomDoublet.u();
         .          .    111:    const float Vb = bottomDoublet.v();
         .          .    112:
         .          .    113:    // 1+(cot^2(theta)) = 1/sin^2(theta)
         .          .    114:    const float iSinTheta2 = 1 + cotThetaB * cotThetaB;
         .          .    115:    const float sigmaSquaredPtDependent = iSinTheta2 * m_cfg.
    ↪ sigmapT2perRadius;
         .          .    116:    // calculate max scattering for min momentum at the seed's theta
    ↪ angle
         .          .    117:    // scaling scatteringAngle^2 by sin^2(theta) to convert pT^2 to p
    ↪ ^2
         .          .    118:    // accurate would be taking 1/atan(thetaBottom)-1/atan(thetaTop)
    ↪ <
         .          .    119:    // scattering
         .          .    120:    // but to avoid trig functions we approximate cot by scaling by
         .          .    121:    // 1/sin^4(theta)
         .          .    122:    // resolving with pT to p scaling --> only divide by sin^2(theta)
         .          .    123:    // max approximation error for allowed scattering angles of 0.04
    ↪ rad at
         .          .    124:    // eta=infinity: ~8.5%
     140ms      140ms    125:    const float scatteringInRegion2 = m_cfg.multipleScattering2 *
    ↪ iSinTheta2;
         .          .    126:
         .          .    127:    std::size_t topDoubletOffset = 0;
     170ms      170ms    128:    for (auto [topDoublet, topDoubletIndex] :
         .      150ms    129:         zip(topDoublets, std::ranges::iota_view<std::size_t, std::
    ↪ size_t>(
         .          .    130:                                 0, topDoublets.size()))) {
         .       20ms    131:      const SpacePointIndex2 spT = topDoublet.spacePointIndex();
         .          .    132:      const float cotThetaT = topDoublet.cotTheta();
         .          .    133:
         .          .    134:      // use geometric average
     110ms      110ms    135:      const float cotThetaAvg2 = cotThetaB * cotThetaT;
         .          .    136:
         .          .    137:      // add errors of spB-spM and spM-spT pairs and add the
    ↪ correlation term
         .          .    138:      // for errors on spM
     330ms      440ms    139:      const float error2 = topDoublet.er() + erB +
     220ms      220ms    140:                           2 * (cotThetaAvg2 * varianceRM +
    ↪ varianceZM) *
     300ms      300ms    141:                               iDeltaRB * topDoublet.iDeltaR();
         .          .    142:
      10ms       10ms    143:      const float deltaCotTheta = cotThetaB - cotThetaT;
      10ms       10ms    144:      const float deltaCotTheta2 = deltaCotTheta * deltaCotTheta;
         .          .    145:
         .          .    146:      // Apply a cut on the compatibility between the r-z slope of
    ↪ the two
         .          .    147:      // seed segments. This is done by comparing the squared
    ↪ difference
```

```
     .          .    148:      // between slopes, and comparing to the squared uncertainty in
↪ this
     .          .    149:      // difference - we keep a seed if the difference is compatible
↪ within
     .          .    150:      // the assumed uncertainties. The uncertainties get
↪ contribution from
     .          .    151:      // the  space-point-related squared error (error2) and a
↪ scattering term
     .          .    152:      // calculated assuming the minimum pt we expect to reconstruct
     .          .    153:      // (scatteringInRegion2). This assumes gaussian error
↪ propagation which
     .          .    154:      // allows just adding the two errors if they are uncorrelated (
↪ which is
     .          .    155:      // fair for scattering and measurement uncertainties)
 850ms      850ms    156:      if (deltaCotTheta2 > error2 + scatteringInRegion2) {
     .          .    157:        if constexpr (sortedByCotTheta) {
     .          .    158:          // skip top SPs based on cotTheta sorting when producing
↪ triplets
     .          .    159:          // break if cotTheta from bottom SP < cotTheta from top SP
↪ because
     .          .    160:          // the SP are sorted by cotTheta
 420ms      420ms    161:          if (cotThetaB < cotThetaT) {
     .          .    162:            break;
     .          .    163:          }
     .          .    164:          topDoubletOffset = topDoubletIndex + 1;
     .          .    165:        }
     .          .    166:        continue;
     .          .    167:      }
     .          .    168:
 790ms      810ms    169:      const float dU = topDoublet.u() - Ub;
     .          .    170:      // protects against division by 0
 200ms      200ms    171:      if (dU == 0) {
     .          .    172:        continue;
     .          .    173:      }
     .          .    174:      // A and B are evaluated as a function of the circumference
↪ parameters
     .          .    175:      // x_0 and y_0
     .          .    176:      const float A = (topDoublet.v() - Vb) / dU;
  50ms       50ms    177:      const float S2 = 1 + A * A;
 140ms      140ms    178:      const float B = Vb - A * Ub;
     .          .    179:      const float B2 = B * B;
     .          .    180:
     .          .    181:      // sqrt(S2)/B = 2 * helixradius
     .          .    182:      // calculated radius must not be smaller than minimum radius
 320ms      320ms    183:      if (S2 < B2 * m_cfg.minHelixDiameter2) {
     .          .    184:        continue;
     .          .    185:      }
     .          .    186:
     .          .    187:      // refinement of the cut on the compatibility between the r-z
↪ slope of
     .          .    188:      // the two seed segments using a scattering term scaled by the
↪ actual
     .          .    189:      // measured pT (p2scatterSigma)
     .          .    190:      const float iHelixDiameter2 = B2 / S2;
     .          .    191:      // convert p(T) to p scaling by sin^2(theta) AND scale by 1/sin
↪ ^4(theta)
     .          .    192:      // from rad to deltaCotTheta
 370ms      370ms    193:      const float p2scatterSigma = iHelixDiameter2 *
↪ sigmaSquaredPtDependent;
     .          .    194:      // if deltaTheta larger than allowed scattering for calculated
↪ pT, skip
  70ms       70ms    195:      if (deltaCotTheta2 > error2 + p2scatterSigma) {
     .          .    196:        if constexpr (sortedByCotTheta) {
 110ms      110ms    197:          if (cotThetaB < cotThetaT) {
     .          .    198:            break;
     .          .    199:          }
     .          .    200:          topDoubletOffset = topDoubletIndex;
     .          .    201:        }
     .          .    202:        continue;
     .          .    203:      }
```

```
         .          .    204:
         .          .    205:      // A and B allow calculation of impact params in U/V plane with
    ↪  linear
         .          .    206:      // function
         .          .    207:      // (in contrast to having to solve a quadratic function in x/y
    ↪ plane)
      90ms       90ms    208:      const float im = std::abs((A - B * rM) * rM);
      10ms       10ms    209:      if (im > m_cfg.impactMax) {
         .          .    210:        continue;
         .          .    211:      }
         .          .    212:
         .          .    213:      // inverse diameter is signed depending on if the curvature is
         .          .    214:      // positive/negative in phi
      30ms       30ms    215:      tripletTopCandidates.emplace_back(spT, B / std::sqrt(S2), im);
         .          .    216:    }  // loop on tops
         .          .    217:
         .          .    218:    if constexpr (sortedByCotTheta) {
         .          .    219:      // remove the top doublets that were skipped due to cotTheta
    ↪ sorting
      60ms      160ms    220:      topDoublets = topDoublets.subrange(topDoubletOffset);
         .          .    221:    }
         .          .    222:  }
         .          .    223:
         .          .    224:  template <typename TopDoublets>
         .          .    225:  void createStripTripletTopCandidates(
```

H3 — Indirect call in `sortByCotTheta`

Listing B.5: Pixel: `DoubletsForMiddleSp::sortByCotTheta` and comparator lambda annotation (total 350 ms).

```
Total: 20.06s
ROUTINE ======================== Acts::DoubletsForMiddleSp::sortByCotTheta in /storage/thomaaks/
    ↪ acts-seeding-optimization/ACTS/source/Core/include/Acts/Seeding2/DoubletSeedFinder.hpp
     530ms     20.06s (flat, cum)   100% of Total
         .          .     86:  using IndexAndCotThetaSubset = std::span<const IndexAndCotTheta>;
         .          .     87:
         .          .     88:  /// Sort doublets by cotTheta within given range
         .          .     89:  /// @param range Index range to sort within
         .          .     90:  /// @param indexAndCotTheta Output vector containing sorted index
    ↪ and cotTheta pairs
      30ms       30ms     91:  void sortByCotTheta(const IndexRange& range,
         .          .     92:                       std::vector<IndexAndCotTheta>& indexAndCotTheta
    ↪ ) const {
         .       30ms     93:    indexAndCotTheta.clear();
      20ms       70ms     94:    indexAndCotTheta.reserve(range.second - range.first);
     240ms      240ms     95:    for (Index i = range.first; i < range.second; ++i) {
         .      860ms     96:      indexAndCotTheta.emplace_back(i, m_cotTheta[i]);
         .          .     97:    }
         .     18.59s     98:    std::ranges::sort(indexAndCotTheta, {}, [](const IndexAndCotTheta
    ↪ & item) {
         .          .     99:      return item.cotTheta;
         .          .    100:    });
     240ms      240ms    101:  }
         .          .    102:
         .          .    103:  class Proxy {
         .          .    104:   public:
         .          .    105:    Proxy(const DoubletsForMiddleSp* container, Index index)
         .          .    106:        : m_container(container), m_index(index) {}
ROUTINE ======================== Acts::DoubletsForMiddleSp::sortByCotTheta() const::{lambda(Acts::
    ↪ DoubletsForMiddleSp::IndexAndCotTheta const&)#1}::operator() in /storage/thomaaks/acts-
    ↪ seeding-optimization/ACTS/source/Core/include/Acts/Seeding2/DoubletSeedFinder.hpp
     1.81s      1.81s (flat, cum)  9.02% of Total
         .          .     94:    indexAndCotTheta.reserve(range.second - range.first);
         .          .     95:    for (Index i = range.first; i < range.second; ++i) {
         .          .     96:      indexAndCotTheta.emplace_back(i, m_cotTheta[i]);
         .          .     97:    }
```

```
         .          .     98:    std::ranges::sort(indexAndCotTheta, {}, [](const IndexAndCotTheta
    ↪ & item) {
      1.81s      1.81s     99:      return item.cotTheta;
         .          .    100:    });
         .          .    101:  }
         .          .    102:
         .          .    103:  class Proxy {
         .          .    104:   public:
```

Listing B.6: Strip: `DoubletsForMiddleSp::sortByCotTheta` and comparator lambda annotation (total 140 ms).

```
Total: 6.37s
ROUTINE ======================== Acts::DoubletsForMiddleSp::sortByCotTheta in /storage/thomaaks/
    ↪ acts-seeding-optimization/ACTS/source/Core/include/Acts/Seeding2/DoubletSeedFinder.hpp
     190ms      6.37s (flat, cum)   100% of Total
         .          .     86:  using IndexAndCotThetaSubset = std::span<const IndexAndCotTheta>;
         .          .     87:
         .          .     88:  /// Sort doublets by cotTheta within given range
         .          .     89:  /// @param range Index range to sort within
         .          .     90:  /// @param indexAndCotTheta Output vector containing sorted index
    ↪ and cotTheta pairs
      20ms       20ms     91:  void sortByCotTheta(const IndexRange& range,
         .          .     92:                       std::vector<IndexAndCotTheta>& indexAndCotTheta
    ↪ ) const {
         .       20ms     93:    indexAndCotTheta.clear();
         .       10ms     94:    indexAndCotTheta.reserve(range.second - range.first);
      50ms       50ms     95:    for (Index i = range.first; i < range.second; ++i) {
         .      310ms     96:      indexAndCotTheta.emplace_back(i, m_cotTheta[i]);
         .          .     97:    }
         .      5.84s     98:    std::ranges::sort(indexAndCotTheta, {}, [](const IndexAndCotTheta
    ↪ & item) {
         .          .     99:      return item.cotTheta;
         .          .    100:    });
     120ms      120ms    101:  }
         .          .    102:
         .          .    103:  class Proxy {
         .          .    104:   public:
         .          .    105:    Proxy(const DoubletsForMiddleSp* container, Index index)
         .          .    106:        : m_container(container), m_index(index) {}
ROUTINE ======================== Acts::DoubletsForMiddleSp::sortByCotTheta() const::{lambda(Acts::
    ↪ DoubletsForMiddleSp::IndexAndCotTheta const&)#1}::operator() in /storage/thomaaks/acts-
    ↪ seeding-optimization/ACTS/source/Core/include/Acts/Seeding2/DoubletSeedFinder.hpp
     320ms      320ms (flat, cum)  5.02% of Total
         .          .     94:    indexAndCotTheta.reserve(range.second - range.first);
         .          .     95:    for (Index i = range.first; i < range.second; ++i) {
         .          .     96:      indexAndCotTheta.emplace_back(i, m_cotTheta[i]);
         .          .     97:    }
         .          .     98:    std::ranges::sort(indexAndCotTheta, {}, [](const IndexAndCotTheta
    ↪ & item) {
     320ms      320ms     99:      return item.cotTheta;
         .          .    100:    });
         .          .    101:  }
         .          .    102:
         .          .    103:  class Proxy {
         .          .    104:   public:
```

H4 — Allocation pressure in `emplace_back`

Listing B.7: Pixel: `DoubletsForMiddleSp::emplace_back` annotation (total 50 ms).

```
Total: 3.01s
ROUTINE ======================== Acts::DoubletsForMiddleSp::emplace_back in /storage/thomaaks/acts
    ↪ -seeding-optimization/ACTS/source/Core/include/Acts/Seeding2/DoubletSeedFinder.hpp
     1.37s      3.01s (flat, cum)   100% of Total
         .          .     57:  /// @param er Error in R coordinate
         .          .     58:  /// @param u U coordinate parameter
```

```
         .          .     59:  /// @param v V coordinate parameter
         .          .     60:  /// @param x X coordinate
         .          .     61:  /// @param y Y coordinate
     610ms      610ms     62:  void emplace_back(SpacePointIndex2 sp, float cotTheta, float
    ↪ iDeltaR,
         .          .     63:                    float er, float u, float v, float x, float y) {
         .      330ms     64:    m_spacePoints.push_back(sp);
         .      210ms     65:    m_cotTheta.push_back(cotTheta);
         .      270ms     66:    m_er_iDeltaR.push_back({er, iDeltaR});
         .       10ms     67:    m_uv.push_back({u, v});
         .      820ms     68:    m_xy.push_back({x, y});
     760ms      760ms     69:  }
         .          .     70:
         .          .     71:  /// Get reference to space point indices container
         .          .     72:  /// @return Const reference to space point indices vector
         .          .     73:  const std::vector<SpacePointIndex2>& spacePoints() const {
         .          .     74:    return m_spacePoints;
```

Listing B.8: Strip: `DoubletsForMiddleSp::emplace_back` annotation (total 20 ms).

```
Total: 890ms
ROUTINE ======================== Acts::DoubletsForMiddleSp::emplace_back in /storage/thomaaks/acts
    ↪ -seeding-optimization/ACTS/source/Core/include/Acts/Seeding2/DoubletSeedFinder.hpp
     500ms      890ms (flat, cum)   100% of Total
         .          .     57:  /// @param er Error in R coordinate
         .          .     58:  /// @param u U coordinate parameter
         .          .     59:  /// @param v V coordinate parameter
         .          .     60:  /// @param x X coordinate
         .          .     61:  /// @param y Y coordinate
     340ms      340ms     62:  void emplace_back(SpacePointIndex2 sp, float cotTheta, float
    ↪ iDeltaR,
         .          .     63:                    float er, float u, float v, float x, float y) {
         .       70ms     64:    m_spacePoints.push_back(sp);
         .       80ms     65:    m_cotTheta.push_back(cotTheta);
         .      130ms     66:    m_er_iDeltaR.push_back({er, iDeltaR});
         .          .     67:    m_uv.push_back({u, v});
         .      110ms     68:    m_xy.push_back({x, y});
     160ms      160ms     69:  }
         .          .     70:
         .          .     71:  /// Get reference to space point indices container
         .          .     72:  /// @return Const reference to space point indices vector
         .          .     73:  const std::vector<SpacePointIndex2>& spacePoints() const {
         .          .     74:    return m_spacePoints;
```

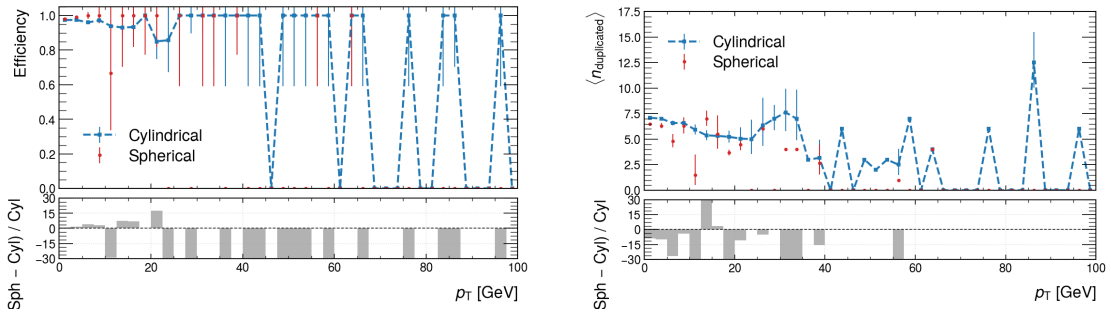
B.3 Algorithm-Level Optimisations

This appendix supplements the spherical grid-binning results in Section 6.4.

Spherical grid: physics-validation supplement

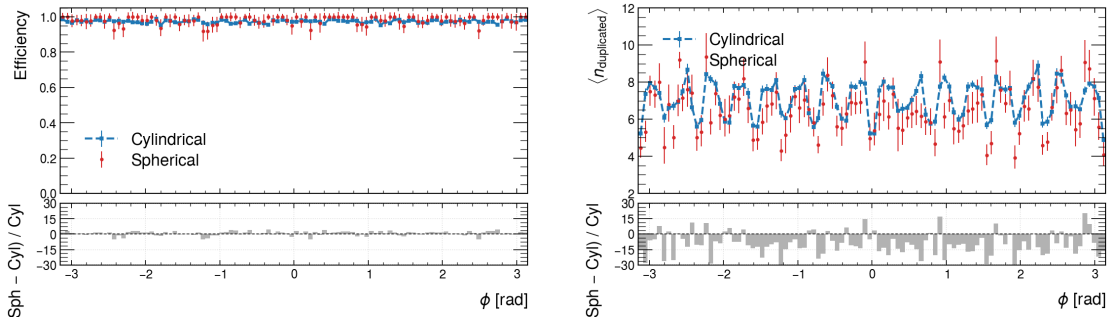
The figures below extend the validation of the *Best* spherical configuration from pseudorapidity to transverse momentum p_T and azimuthal angle ϕ. Ratio panels show (Spherical − Cylindrical)/Cylindrical.

The spherical configuration reproduces the cylindrical efficiency within statistical uncertainties where the bins are sufficiently populated. The largest fluctuations occur in the sparse high-p_T tail. In ϕ, the efficiency is flat and consistent with the cylindrical baseline, while the duplicated-track count follows the same detector-driven periodic structure in both configurations.



(a) Efficiency. (b) Duplicated tracks.

Figure B.7: Seeding performance as a function of transverse momentum p_T for the cylindrical and spherical *Best* configurations. The high-p_T tail is sparsely populated and therefore shows larger statistical fluctuations.



(a) Efficiency. (b) Duplicated tracks.

Figure B.8: Seeding performance as a function of azimuthal angle ϕ for the cylindrical and spherical *Best* configurations. The duplicated tracks show the same detector-driven periodic structure in both configurations.

Spherical grid: parameter-scan supplement

Each scan varies one parameter while keeping the others at the runner defaults. The vertical dashed line marks the default. The panels show efficiency, fake and duplicate ratios, and wall-clock time normalised to the cylindrical baseline. Across the one-dimensional scans, the spherical grid remains between approximately 7 % and 25 % of the cylindrical wall-clock time.

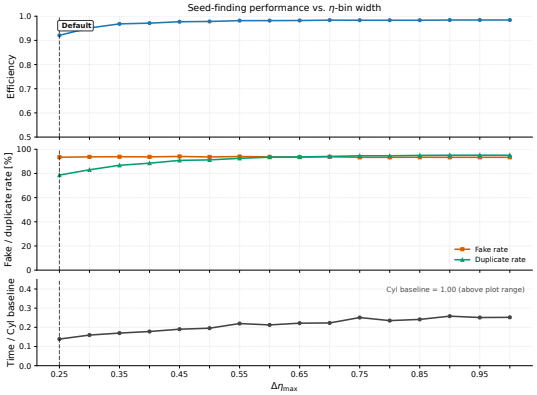


Figure B.9: Spherical scan of $\Delta\eta_{\max}$. Efficiency plateaus near $\Delta\eta_{\max} \gtrsim 0.35$; the selected value 0.3 lies near the start of the plateau.

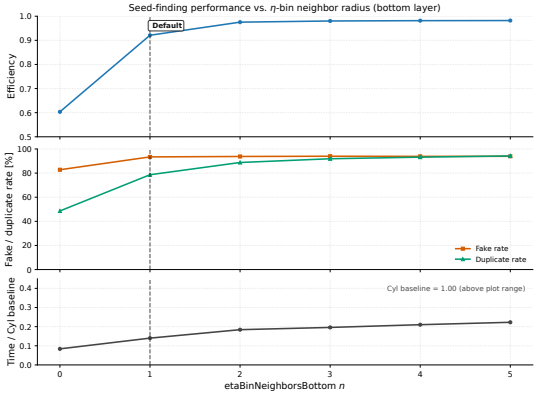


Figure B.10: Spherical scan of the bottom-layer η-bin neighbour radius. Efficiency saturates near $n = 2$, used by the *Best* configuration.

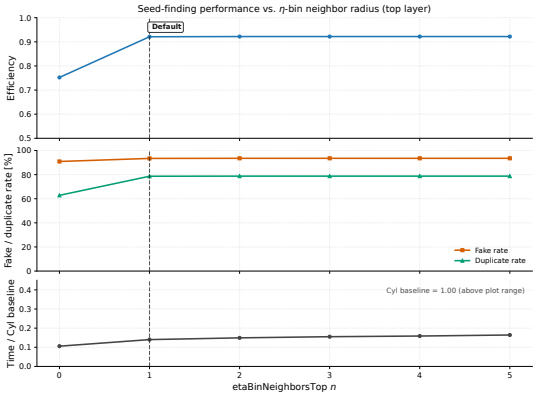


Figure B.11: Spherical scan of the top-layer η-bin neighbour radius. Efficiency already saturates at the default $n = 1$; larger values add cost without improving performance.

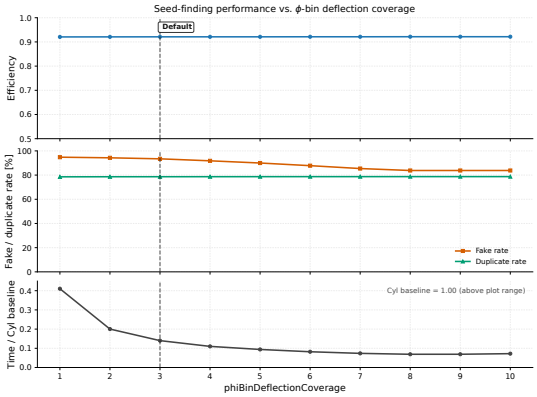


Figure B.12: Spherical scan of ϕ-bin deflection coverage. Physics metrics are nearly flat, while runtime decreases strongly; the selected configurations use 8.

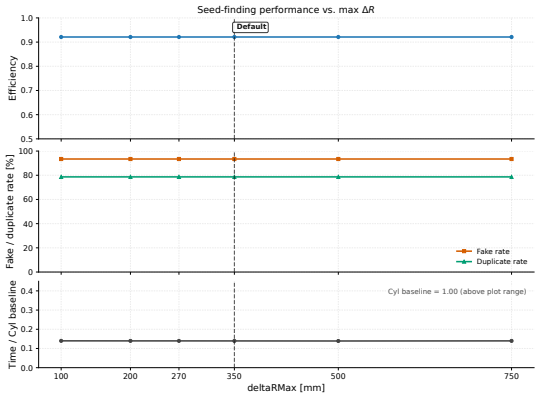


Figure B.13: Spherical scan of $\Delta R_{\max}$. No visible dependence is seen in the physics metrics or runtime; the default 350 mm is outside the active region.

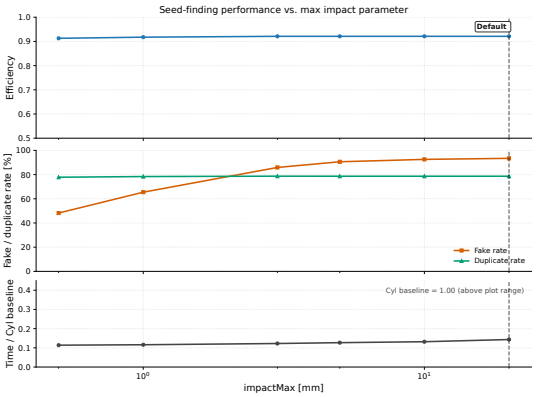


Figure B.14: Spherical scan of $d_{0,\max}$. Tightening the cut reduces the fake rate with only a small efficiency loss; the default is kept for cylindrical compatibility.

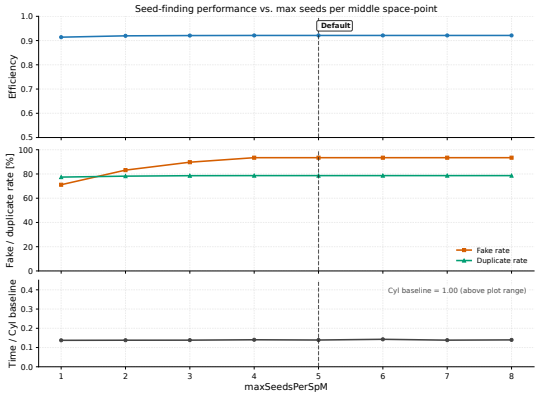


Figure B.15: Spherical scan of the maximum seeds per middle space point. Lowering the cap reduces fake seeds at a small efficiency cost; this lever is used by the *Fastest* configuration.

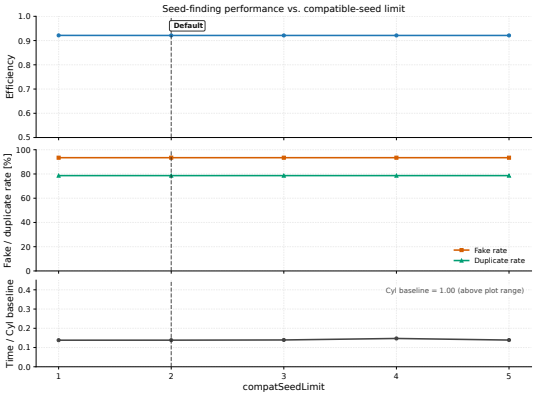


Figure B.16: Spherical scan of the compatible-seed limit. The scan is flat across the tested range, indicating that the default is already in the saturated regime.

B.4 `Seeding3` Proposal

This appendix collects the function-level profiles used to support the `Seeding3` evaluation in Section 6.5. They provide additional call-stack detail beyond the stage-level results in Chapter 6.

Top functions by flat CPU time

Table B.1 lists the ten most expensive functions for `Seeding2` and `Seeding3`, aggregated over five runs. Percentages are relative to each algorithm's top-level scope.

Table B.1: Top-10 functions by flat CPU time for `Seeding2` and `Seeding3`. Profiles are aggregated over five runs and filtered to each algorithm's top-level scope.

#	Function	Flat %	Cum %
Seeding2			
	`createDoubletsImpl (inline)`	24.70	30.28
	`createPixelTripletTopCandidates (inline)`	23.95	28.44
	`std::__unguarded_partition (inline)`	10.68	15.69
	`std::__unguarded_linear_insert (inline)`	8.80	8.80
	`{lambda(Acts::DoubletsForMiddleSp::IndexAndCotTheta con...`	3.75	3.75
	`BroadTripletSeedFilter::filterTripletTopCandidates (par...`	3.69	7.43
	`Impl::createDoublets (partial-inline)`	2.71	31.43
	`Impl::createTripletTopCandidates`	2.12	30.56
	`operator() (inline)`	1.58	2.25
	`std::vector::operator[] (inline)`	1.44	1.44
Seeding3			
	`createDoubletsImpl (inline)`	16.96	23.35
	`createPixelTripletTopCandidates (inline)`	16.45	19.90
	`std::__unguarded_linear_insert (inline)`	7.42	7.42
	`Impl::createDoublets (partial-inline)`	3.74	26.06
	`BroadTripletSeedFilter::filterTripletTopCandidates (par...`	3.74	9.03
	`std::__unguarded_partition (inline)`	3.30	4.48
	`f64xsubf128`	2.42	2.42
	`Impl::createTripletTopCandidates`	2.28	22.17
	`std::vector::emplace_back (inline)`	2.13	2.86
	`operator() (inline)`	2.06	4.70

Flamegraphs

The flamegraphs compare the on-CPU sample distribution before and after the `Seeding3` changes. The reductions reported in Section 6.5 appear mainly in `TripletEvaluation`, `MiddleBottoms`, and `MiddleTops`.

Callgraphs

The callgraphs provide the same comparison at call-path level. The compatibility-check functions that dominate `Seeding2` shrink in `Seeding3`, consistent with the stage-level timing breakdown.

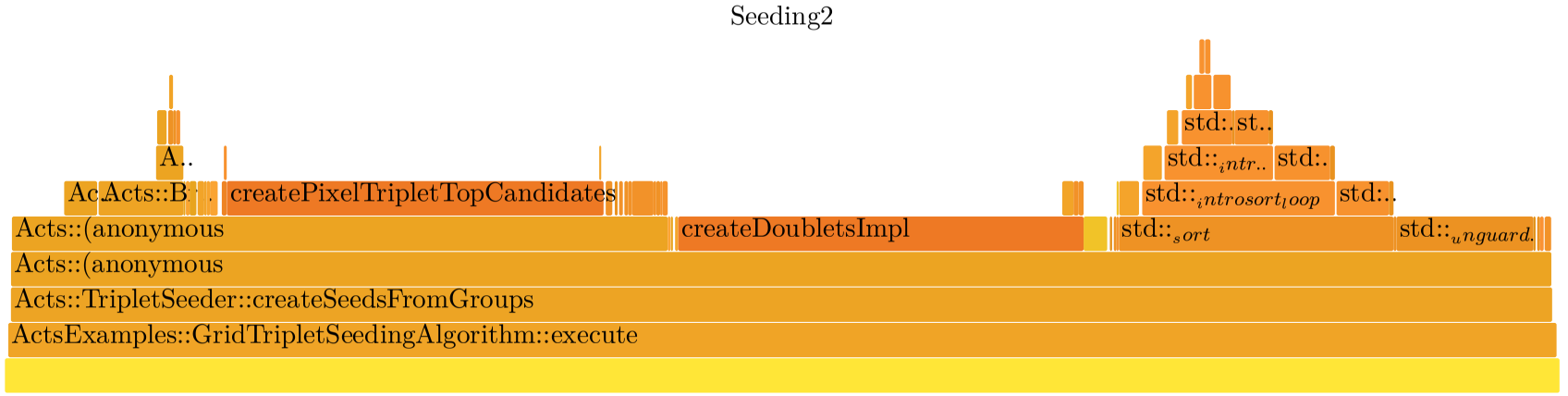


Figure B.17: CPU flamegraph for `Seeding2`, aggregated over five runs. Width denotes the fraction of on-CPU samples; colour is hash-based.

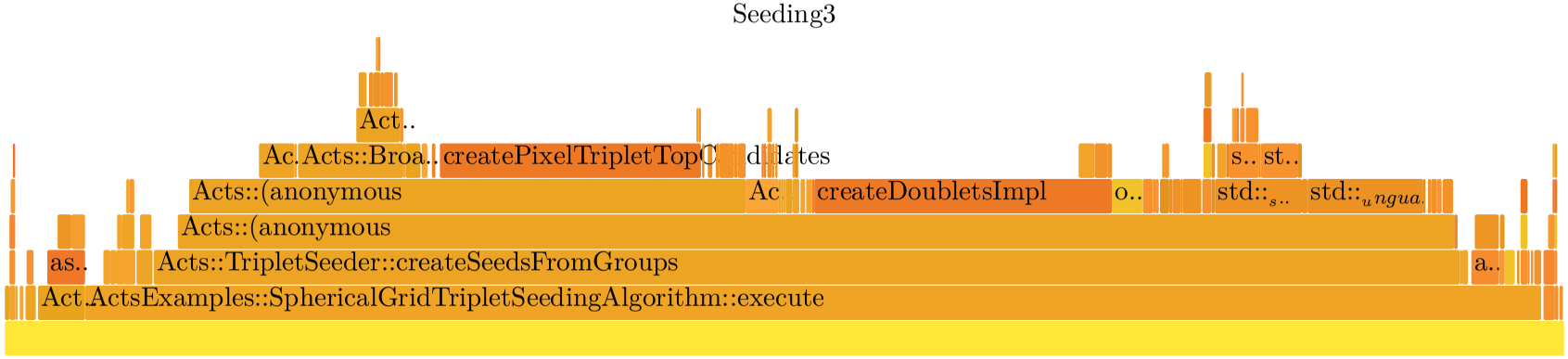


Figure B.18: CPU flamegraph for `Seeding3`, aggregated over five runs. Width denotes the fraction of on-CPU samples; colour is hash-based.

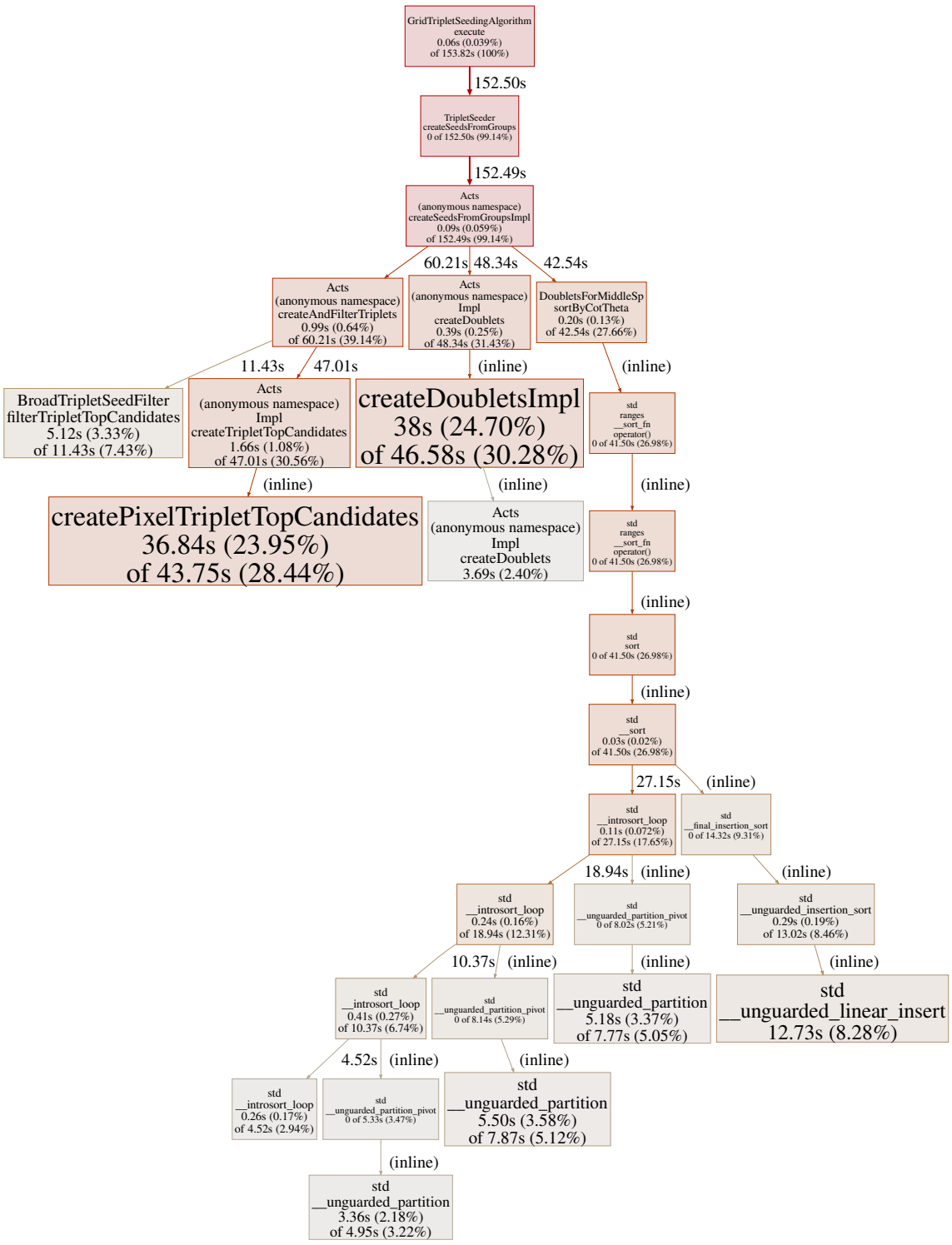


Figure B.19: Annotated `gperftools` callgraph for `Seeding2` aggregated over five runs; top 35 nodes retained.

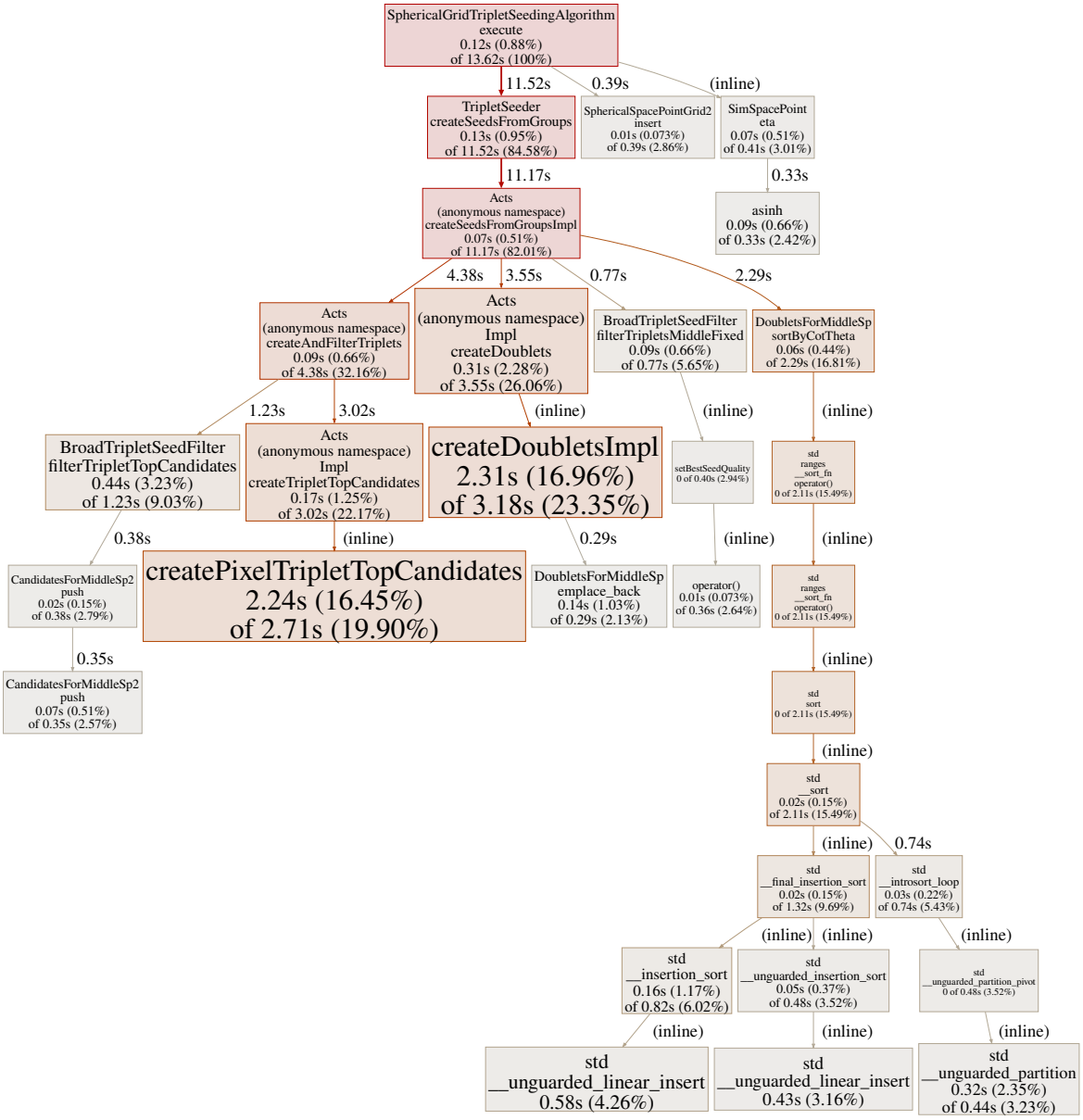


Figure B.20: Annotated `gperftools` callgraph for `Seeding3` aggregated over five runs; top 35 nodes retained.

Bibliography

[1] ACTS Collaboration. Private communication. 2026.

[2] ACTS Project. *ACTS — Tracking in a nutshell*. Accessed: 2025-10-07. 2025. URL: https://acts.readthedocs.io/en/latest/tracking.html.

[3] ACTS Project. *acts-project/acts*. GitHub repository. 2025. URL: https://github.com/acts-project/acts.

[4] ACTS Project. *Getting Started — ACTS Documentation*. Accessed: 2025-10-17. 2025. URL: https://acts.readthedocs.io/en/v28.1.0/getting_started.html.

[5] ACTS Project. *Seeding — ACTS Documentation*. Accessed: 2025-10-17. 2025. URL: https://acts.readthedocs.io/en/latest/core/reconstruction/pattern_recognition/seeding.html.

[6] X. Ai, C. Allaire, N. Calace, et al. "A Common Tracking Software Project". In: *Comput. Softw. Big Sci.* 6.1 (2022), p. 8. DOI: 10.1007/s41781-021-00078-8. URL: https://doi.org/10.1007/s41781-021-00078-8.

[7] C. Allaire et al. *Open Data Detector*. 2022. DOI: 10.5281/zenodo.6445359.

[8] ATLAS Collaboration. "ATLAS b-jet identification performance and efficiency measurement with $t\bar{t}$ events in pp collisions at $\sqrt{s} = 13$ TeV". In: *Eur. Phys. J. C* 79 (2019), p. 970. DOI: 10.1140/epjc/s10052-019-7450-8. arXiv: 1907.05120 [hep-ex].

[9] ATLAS Collaboration. *ATLAS Event Display: HL-LHC simulated event inside the ITk detector*. ATLAS-PHOTO-2023-042, CERN Document Server. Image dated 08-06-2023. 2023. URL: https://cds.cern.ch/record/2861288/.

[10] ATLAS Collaboration. "Electron and photon efficiencies in LHC Run 2 with the ATLAS experiment". In: *JHEP* 05 (2024), p. 162. DOI: 10.1007/JHEP05(2024)162. arXiv: 2308.13362 [hep-ex].

[11] ATLAS Collaboration. *Initial improvements to the CPU performance for the ACTS-based track reconstruction software for ATLAS during the HL-LHC*. ATLAS PUB Note ATL-PHYS-PUB-2024-017. CERN, Oct. 2024. URL: https://cds.cern.ch/record/2912217.

[12] ATLAS Collaboration. "Jet energy scale and resolution measured in proton–proton collisions at $\sqrt{s} = 13$ TeV with the ATLAS detector". In: *Eur. Phys. J. C* 81 (2021), p. 689. DOI: 10.1140/epjc/s10052-021-09402-3. arXiv: 2007.02645 [hep-ex].

[13] ATLAS Collaboration. *Measurement of the tau lepton reconstruction and identification performance in the ATLAS experiment using pp collisions at* $\sqrt{s} =$ 13 *TeV*. Tech. rep. ATLAS-CONF-2017-029. CERN, 2017. URL: https://cds.cern.ch/record/2261772.

[14] ATLAS Collaboration. "Software and computing for Run 3 of the ATLAS experiment at the LHC". In: *Eur. Phys. J. C* 85 (2025), p. 234. DOI: 10.1140/epjc/s10052-024-13701-w. URL: https://doi.org/10.1140/epjc/s10052-024-13701-w.

[15] ATLAS Collaboration. "Software Performance of the ATLAS Track Reconstruction for LHC Run 3". In: *Comput. Softw. Big Sci.* 8 (2024), p. 9. DOI: 10.1007/s41781-023-00111-y. arXiv: 2308.09471 [hep-ex].

[16] ATLAS Collaboration. "Studies of the muon momentum calibration and performance of the ATLAS detector with *pp* collisions at $\sqrt{s} = 13$ TeV". In: *Eur. Phys. J. C* 83 (2023), p. 686. DOI: 10.1140/epjc/s10052-023-11584-x. arXiv: 2212.07338 [hep-ex].

[17] ATLAS Collaboration. *Technical Design Report for the ATLAS ITk Pixel Detector*. Tech. rep. Geneva: CERN, 2018. URL: https://cds.cern.ch/record/2285585.

[18] I. Béjar Alonso et al., eds. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2020. DOI: 10.23731/CYRM-2020-0010. URL: https://cds.cern.ch/record/2749422.

[19] Christian Bierlich et al. "A comprehensive guide to the physics and usage of PYTHIA 8.3". In: *SciPost Phys. Codebases* (2022), p. 8. DOI: 10.21468/SciPostPhysCodeb.8. arXiv: 2203.11601 [hep-ph].

[20] Sylvain Caillou et al. "Physics Performance of the ATLAS GNN4ITk Track Reconstruction Chain". In: *EPJ Web Conf.* 295 (2024), p. 03030. DOI: 10.1051/epjconf/202429503030. URL: https://doi.org/10.1051/epjconf/202429503030.

[21] T. G. Cornelissen et al. *Updates of the ATLAS Tracking Event Data Model (Release 13)*. Tech. rep. ATL-SOFT-PUB-2007-003. ATLAS Collaboration, 2007. URL: https://cds.cern.ch/record/1038095.

[22] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*. Cambridge University Press, 1997.

[23] K. Edmonds et al. *The Fast ATLAS Track Simulation (FATRAS)*. Tech. rep. ATL-SOFT-PUB-2008-001. CERN, 2008. URL: https://cds.cern.ch/record/1091969.

[24] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. CRC Press, 1994.

[25] Philip J. Fleming and John J. Wallace. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results". In: *Communications of the ACM* 29.3 (1986), pp. 218–221. DOI: 10.1145/5666.5673.

[26] Paul Gessinger et al. *traccc: GPU track reconstruction library for HEP experiments*. 2025. arXiv: 2505.22822 [hep-ex]. URL: https://arxiv.org/abs/2505.22822.

[27] Paul Gessinger-Befurt, Andreas Salzburger, and Joana Niermann. "The Open Data Detector Tracking System". In: *J. Phys. Conf. Ser.* 2438.1 (2023), p. 012110. DOI: 10.1088/1742-6596/2438/1/012110. URL: https://cds.cern.ch/record/2869673.

[28] Laura Gonella. "The ATLAS ITk detector system for the Phase-II LHC upgrade". In: *Nucl. Instrum. Meth. A* 1045 (2023), p. 167597. ISSN: 0168-9002. DOI: 10.1016/j.nima.2022.167597. URL: https://www.sciencedirect.com/science/article/pii/S0168900222008890.

[29] Google. *gperftools: Fast, multi-threaded* `malloc()` *and performance analysis tools*. https://github.com/gperftools/gperftools. Accessed: 2026-03-16. 2024.

[30] Brendan Gregg. *Flame Graphs*. https://www.brendangregg.com/flamegraphs.html. Accessed: 2026-03-16. 2016.

[31] Stanley E. Lazic. "The problem of pseudoreplication in neuroscientific studies: is it affecting your analysis?" In: *BMC Neurosci.* 11 (2010), p. 5. DOI: 10.1186/1471-2202-11-5.

[32] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.

[33] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[34] Andi Wand. *feat: `BroadTripletSeedFinder` using the new space point and seed EDM*. ACTS Project, GitHub pull request #4324. Merged 2025-07-03. Introduces `DoubletSeedFinder`, `BroadTripletSeedFinder`, `BroadTripletSeedFilter`, `CylindricalSpacePointGr` and `GridTripletSeedingAlgorithm`; replaces the previous template-based `SeedFinder`/`SeedFilter` with grid-agnostic implementations. July 2025. URL: https://github.com/acts-project/acts/pull/4324.

[35] Milian Wolff and KDE Community. *Heaptrack: A Heap Memory Profiler for Linux*. https://github.com/KDE/heaptrack. Accessed: 2026-03-16. 2024.