Performance optimization of the online data processing software of CERN's LHCb experiment

Thesis report

Péter Kardos

2018-2019

1 Abstract

write at the end
100-200 words
- what's the problem
- how was it solved
- what are the results
- conclusion: what it means for the future
must be understandable without extra info

Don't read this, it's just a placeholder. So this abstract should be about 100-200 words so I'm just writing some natural text to act as a placeholder. By looping this text a few times, I can probably make a 150 word section. So this abstract should be about 100-200 words so I'm just writing some natural text to act as a placeholder. By looping this text a few times, I can probably make a 150 word section. So this abstract should be about 100-200 words so I'm just writing some natural text to act as a placeholder. By looping this text a few times, I can probably make a 150 word section. So this abstract should be about 100-200 words so I'm just writing some natural text to act as a placeholder. By looping this text a few times, I can probably make a 150 word section. So this abstract should be about 100-200 words so I'm just writing some natural text to act as a placeholder. By looping this text a few times, I can probably make a 150 word section.

2 Introduction

describe the problem in detail
specific to my thesis:
environment:
- CERN's goals/activity
- CERN's hardware infrastructure (accelerators, experiments)
- LHCb's hardware infrastructure
- LHCb's software reconstruction system
problem:
- event rate from detector
- slow trigger → loss of physics (ACTUAL PROBLEM)
- by optimizing individual algorithms (in this thesis)

2.1 About CERN

CERN (European Organization for Nuclear Research) is an international high energy experimental physics research organization situated near Geneva, on the Franco-Swiss border. CERN is host to the world's largest particle accelerator and numerous experiments which aim to provide a better understanding of the universe. The goals of the experiments, among others, are to verify the standard model of particles. TODO: list more concrete goals. [1]

2.2 What are particle accelerators

2.2.1 Idea and purpose of accelerators

Particle accelerators, as the name suggests, accelerate charged particles to extremely high velocities. Generally, the accelerated particles are elementary particles, such as protons or electrons, or ions. CERN mainly operates with protons and lead ions, though we can see other particles as well. The velocity of the particles often approaches that of the speed of light in vacuum. CERN's first particle accelerator, the Synchrocyclotron, reached about 80% of the speed of light, whereas CERN's largest accelerator's, the LHC, accelerates its protons to 99.9999991% [8] of the speed of light.

The high-speed particles are made to collide with a stationary target or each other (particles of opposing directions having a frontal collision), which results in a shower of new-born particles flying away from the collision point. In high-energy collisions, exotic particles that are normally not seen are born, and their properties can be examined to advance the scientific field of particle physics.

2.2.2 Theory of operation

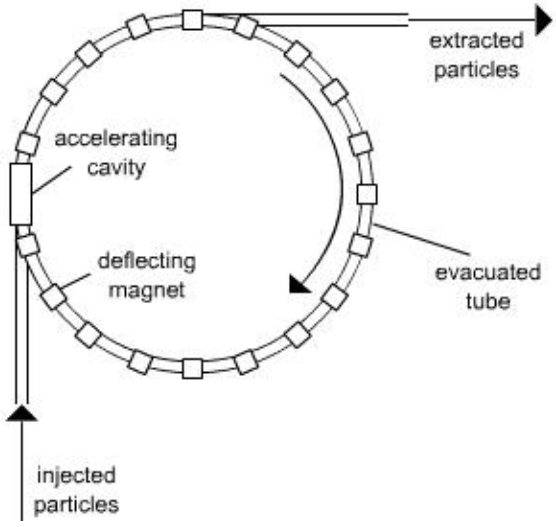


Figure 1: Simplified schematic of a circular particle accelerator with the crucial functional parts labeled.

Particle accelerators can be circular or linear. Figure 1 depicts a circular accelerator, which consists of the following parts:

- Evacuated tube (beam pipe): a closed, circular tube in which the particles can travel. To avoid the particles colliding with air particles, it is strictly under vacuum.
- Particle source: injects the particles into the beam pipe. For proton accelerators, a bottle of hydrogen serves as the source. The hydrogen atoms are ionized, and then linearly accelerated by an electric field before entering the circular beam pipe.
- Accelerating cavity: uses oscillating electromagnetic fields which are timed correctly to provide a push to the charged particles via electric force, accelerating them.
- Deflecting magnet: using the Lorentz-force, a strong magnetic field steers the particles inwards to the center of the circle, keeping it on the circular trajectory.
- Extraction pipe: once the particles are fast enough, they are extracted from the circular beam pipe to collide with a stationary target.

The particle is at first injected to the beam pipe at a low energy. Thanks to the deflecting magnets, it keeps revolving in the beam pipe for thousand of revolutions. Each turn, particles get boosted by the accelerating cavities as they pass by, increasing their energy. As the energy increases, deflecting magnets have to work more with stronger fields to keep them on track. When particles reach their maximum energy, which is determined by the construction and size of the accelerator, they are extracted from the beam pipe to collide with a stationary target.

In fact, it is not individual particles that revolve around the accelerator, rather, it's a swarm of thousands or millions of particles distributed throughout the entire circle,

forming a so called beam. The distribution, however, is not continuous. The particles group into *bunches* which are equally spaced throughout the circle. The LHC contains 2808 bunches, each consisting of $1.15 \cdot 10^{11}$ protons.

In addition to deflecting magnets, accelerators also employ *focusing magnets*. Their purpose is to squeeze the bunches in the directions perpendicular to the particles' velocity, resulting in the beam having a smaller cross-section. Without focusing magnets, the bunches would slowly disintegrate and the particles would hit the wall of the beam pipe.

Linear accelerators contain the same functional elements as circular ones, except for the fact that they don't need deflecting magnets since the particles travel in a straight path.

2.2.3 Particle colliders

As opposed to particle accelerators, colliders operate with two beams at the same time. A circular collider, such as the LHC, has two beam pipes right next to each other in the circular tunnel. The two beams circle in the opposing directions. At some specific points, the two beam pipes are made to cross, and as the particle beams intersect, the bunches collide with each other as opposed to colliding with a fixed target.

2.3 The accelerator complex [2]

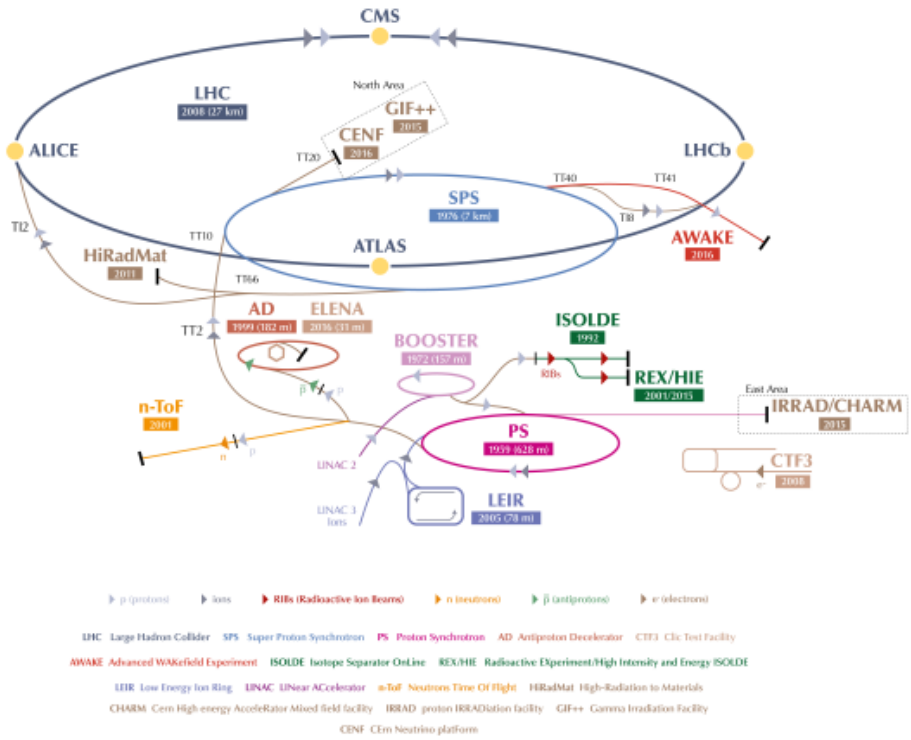


Figure 2: Schematic view of CERN's particle accelerators and experiments. LHC is shown on top by the largest circle. The four main experiments, CMS, ALICE, ATLAS and LHCb are marked with yellow dots along the LHC's circle.

As seen on 2, CERN has a quite complex system of particle accelerators. We can see several circular and linear accelerators, and also some deccelerators, all of which are running at the same time in a synchronized fashion so that they can interact with eachother.

The largest circle, in dark blue, is LHC, which is the world's largest and most powerful particle collider to date. The way protons reach their final energy in the LHC is a complex, multi-stage process. The protons are sourced from a bottle of hydrogen, and injected into LEIR via LINAC 3. When the energy of the protons reaches the maximum operating point of LEIR, they are transferred into PS for further boosting. When they reach PS's operating maximum, they are transferred into SPS, and finally into the LHC.

Inside the LHC, we can find two beams going in the opposite directions. The energy of each particle reaches about 7 TeVs at maximum. The two beams cross each other and particles collide at 4 points, marked CMS, ALICE, LHCb and ATLAS. At these points, complex particle detectors are installed to analyze the collisions. Detectors track the path of the particles, and make measurements on their properties such as momentum, charge and mass. From the collection of properties, particles can be identified. Measurements

provide valuable data to physicists who are trying to verify and extend the standard model of particles. In most cases, the raw data provided by the detectors is processed by software.

2.4 LHCb experiment's detector

2.4.1 Construction of the detector

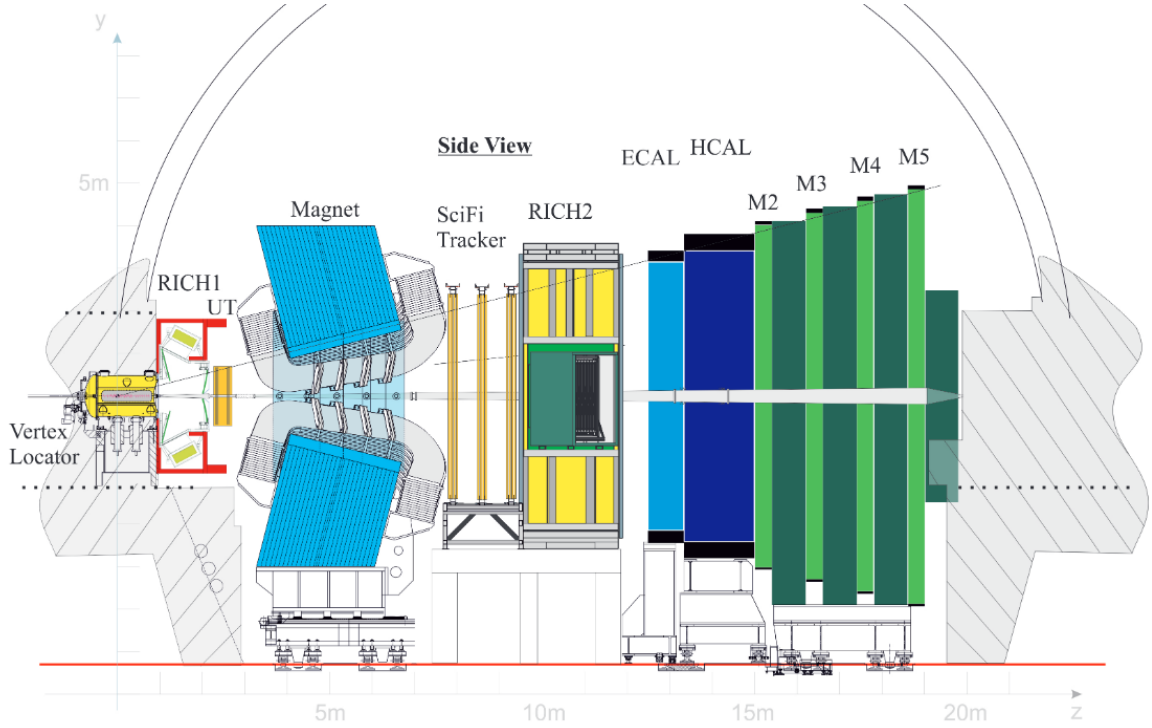


Figure 3: Side view of the LHCb detector.

Figure 3 shows the LHCb detector from the side, which means that the two beams of LHC are going in horizontal directions on the drawing through the middle of the detector. The beam pipe coincides with the horizontal axis of symmetry of the detector.

As seen on the labels, the detector consists of multiple layers of sub-detectors. Each layer has a hole in the center to let the beam pipes through. The two particle beams cross each other inside the Vertex Locator (VELO, at the right in yellow). As opposed to most other detectors, this one only analyzes the products of the collision in a narrow cone away from the VELO. The parts from RICH to M5 could be mirrored around the VELO to have two cones that touch each other by the tip, however it is not done for financial reasons.

The goal of the LHCb detector is the same as for all other detectors: reconstruct the paths and types of the particles. Even though full reconstruction uses all sub-detectors, we are only interested in partial reconstruction that can be done in real-time. This only involves the VELO, the UT (in orange, left of VELO) and the FT (SciFi Tracker, in the middle in orange).

2.4.2 Coordinate system

When doing calculatings, the detector must be placed in a 3 dimensional euclidean space. The coordinate axis are chosen so that Z down the beam pipe from the VELO towards the SciFi tracker, X points to the right when lookind down the Z axis, and Y points upwards.

TODO: add a picture

2.4.3 Operating principles

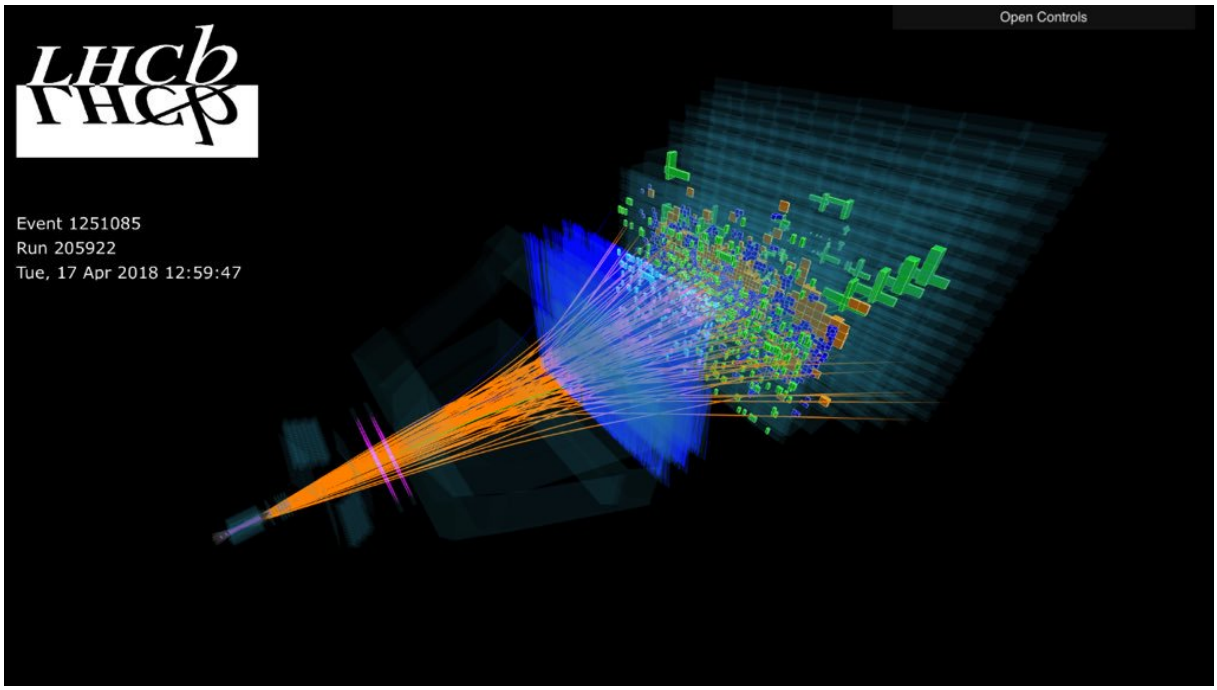


Figure 4: Particles created in a real collision and their interaction with the detector.

Figure 4 shows one particle collision event's results. The particles that were born in the collision are shown by the orange lines. The three aforementioned detectors, the VELO, UT and FT, can be identified by the origin of the orange particles, the pink cloud of lines and the bright blue cloud lines, respectively. The green and yellow cubic illustrations on the far-end of the detector belong to other sub-detectors, and are out of the scope of this paper.

The VELO is a small detector, measures less than a meter in length. It is a silicon pixel detector, which looks much like a modern CCD camera: a rectangular array of pixels which detect light (or in this case, particles) that hit it. The difference is that particle detectors don't absorb the particle, rather, it passes through largely undisturbed. Additionally, the VELO consists of 26 such CCD-like rectangles, instead of just one.

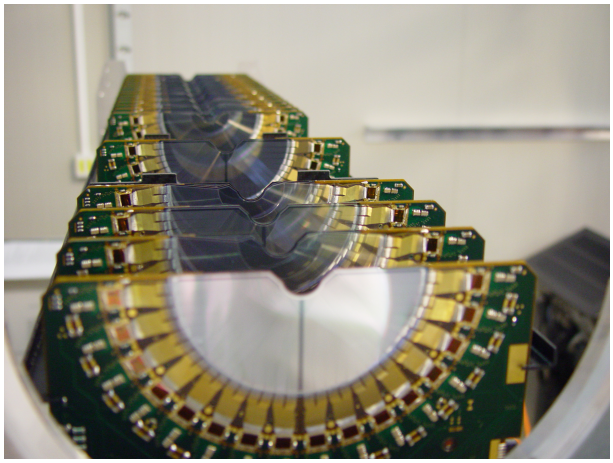


Figure 5: The 26 layers of the VELO. The pixels reside on the silver area, the PCB around contains the reading electronics, and the wedge at the top-center of the silver area corresponds to the beam pipe.

As opposed to the VELO, the UT is a silicon strip detector. Silicon strips detectors consist of not pixels, but from long fibers which act much like a pixel in the sense they also signal if a particle passes through them. A fiber of the UT is generally 10 centimeters long, and has a width of only 192 micrometers. This means that while on one axis, the particles position can be told with great accuracy, on the other axis the uncertainty if 10 centimeters. The entire detector measures about 1.7 meters in width and 1.4 meter in height.

Similarly to the UT, the FT is also a silicon strip detector, but the length of it's fibers is 2.5 meters. This means the 5 meter tall detector needs only two fibers to cover the full height.

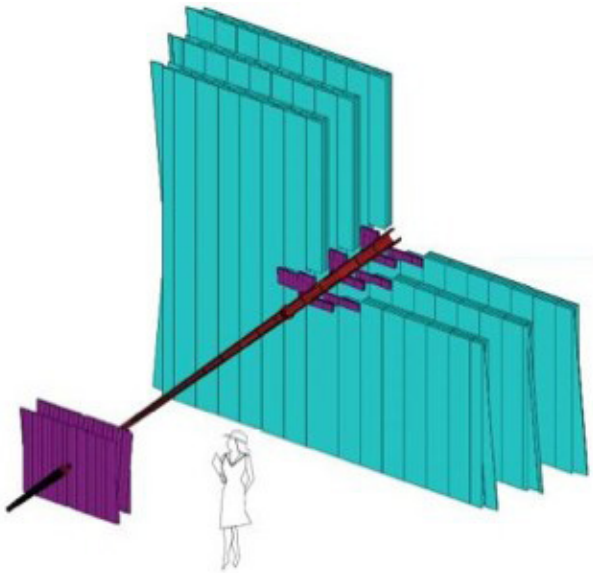


Figure 6: The UT in purple and the FT in blue, with a person next to them to illustrate the scale. The silicon strips are aligned vertically, leading to a good horizontal resolution but a poor vertical resolution.

In between the UT and the FT, a strong magnet is placed which bends the particles on the horizontal axis.

When a collision occurs, the particles follow a straight path and they are recorded passing through the pixels of the VELO. Initial particle trajectories can be reconstructed by finding hits in the VELO that align to form a straight line. These paths are then linearly extrapolated through the UT, and some of the silicon strips that were lit up by the particles are assigned to the initial trajectories acquired from the VELO. Inside the UT, particles paths are mostly straight, but they already experience a slight bending due to the magnet after the UT. From the amount of bending, the particle's momentum can be estimated. As the charged particles pass through the magnet, their trajectory bends, however, the amount of bending is a function of the particle's momentum. With a good momentum estimate, a guess can be given as to where the particle would hit the FT. In the suspected region, the silicon strips of the FT are searched and if the corresponding fibers are found, they are assigned to the particle.

At the end of the process, the all the pixels or fibers that were touched by a particle are known, which makes it possible to know the exact path of the particle. The amount of bending from the UT to the FT allows the calculation of the momentum, which, when paired with information from other detectors, such as energy and velocity, makes it possible to identify a particle. (Identification means knowing the name of the particle, such as electron or muon.)

2.5 Events and triggering

2.5.1 Collision events

As explained earlier, the LHC has 2808 bunches of protons circulating in both directions in the two beam pipes. We refer to the collision of the of these bunches as an *event*. Events are completely independent, that is, a bunch-bunch collision only happens after the previous collision's products have been analyzed and flushed from the detector. At nearly the speed of light, a particle takes 89 microseconds to do a revolution in the circular detector of 26 659 m circumference. The 2808 bunches are however spaced at a distance of 25 nanoseconds from each-other, giving way to at most 40 million collisions every second. Calculating with 2808 bunches and 89 microseconds, the bunch spacing should be 31.7 nanoseconds. There are, however, some longer gaps in the line of 2808 bunches, so two adjacent bunches are still 25 nanoseconds away. This is why, in practice, there are only roughly 30 million collisions a second, but the processing of that has to happen as fast as 40 million a second.

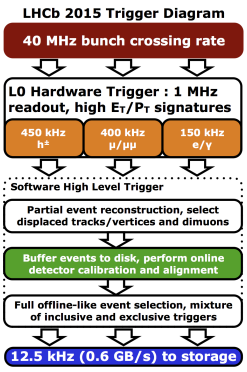
2.5.2 Real-time reconstruction and triggering

Most of the 30 million events that occur every second are absolutely uninteresting, with no exotic particles of interest being created. On the other hand, each event amounts to a significant amount of data which is a challenge to store. To reduce the amount of data to be stored, each event is processed in real-time to determine weather it is interesting or not. Uninteresting events are simply dropped, the interesting ones go to long-term storage for further analysis. The act of deciding if an event has to be stored is called *triggering*. The real-time processing of events is a simplified method that relies on the VELO, UT and FT detectors as described previously.

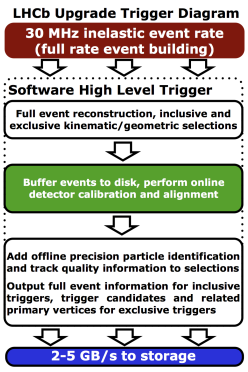
2.6 The 2019/20 upgrade of LHCb

The LHC shuts down for maintenance for two years from the end of 2018. The LHCb detector also undergoes maintenance and upgrade during that time, where the VELO is upgraded and the old TT is entirely replaced by the UT.

The pre-upgrade detector does the real-time event processing in two stages. The first stage employs FPGAs to do a preselection, which cuts down the 30 million events per second to roughly 1 million per second. The second stage uses a large farm of CPUs to do finer reconstruction and final trigger decision. During the upgrade, however, the FPGAs will be retired, and the CPU farm has to take the whole load of 30 million events per second. This puts a stress on the software stack that it was not written to handle, and needs a significant overhaul.



(a) Trigger data flow scheme before the upgrade



(b) Trigger data flow scheme post-upgrade

Figure 7: Comparison of the two triggering solutions

2.7 Overview of the real-time processing software

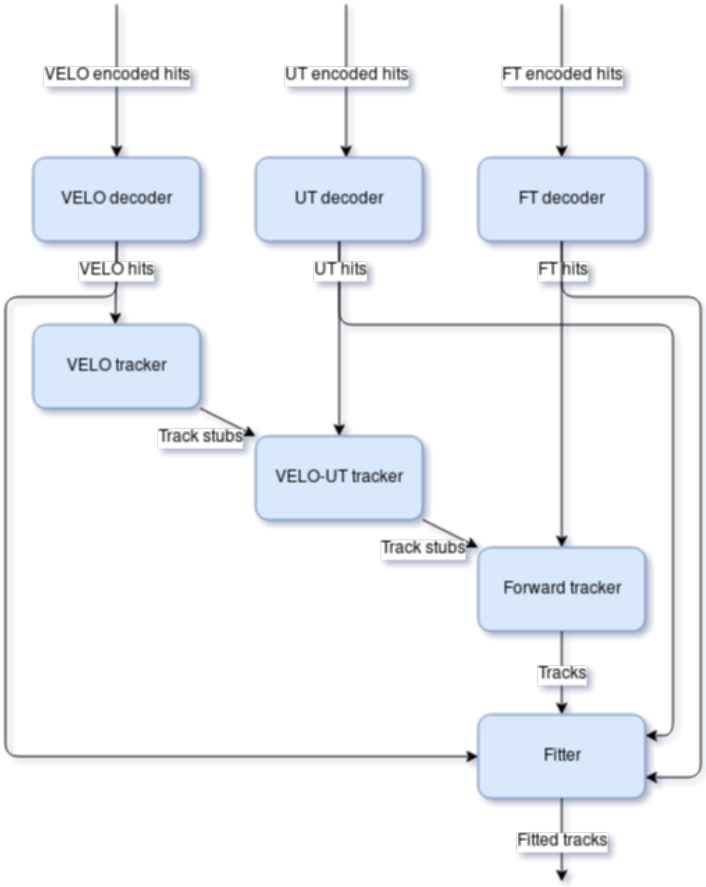


Figure 8: Simplified view of algorithms that perform online reconstruction.

The real-time processing software consists of algorithms which do a specific piece of the reconstruction of the particle paths. In addition to the reconstruction, there are other algorithms which do the selection of interesting events. Since the selection algorithms are generally a lot faster than reconstruction algorithms, they are excluded them from this discussion.

The general principle behind reconstruction is that a list of hits (where a pixel or silicon strip was hit) are given, and regular alignment of the hits indicate the path of the particle: hence the name *pattern recognition*. Once all particles have been found by gathering the hits they created, they can be identified and fed into the selection decision making.

The information on which pixels and strips were hit comes straight from the detectors, in a heavily compressed binary format. It has to be first decoded to give the position of the individual strips, a process done by the **VELO decoder**, **UT decoder** and **FT decoder**.

Once pixel hit positions are known by their global X,Y,Z coordinates, the **VELO tracker** find hits that align to form a straight line. It estimates the starting position and the slope of the line. The line is extended into the UT, where the **VELO-UT tracker** searches silicon strip hits that are very close to the extended line. From the slight offset of the strip hits from the extended line, the VELO-UT tracker can estimate the amount of bending in the magnetic space, thus the momentum of the particle. The last step of tracking is done by the **forward tracker**, which uses the momentum estimate to extrapolate the bent path of the particle through the magnet, to the layers of the FT. It then finds silicon strips of the FT which align into a straight line (no magnetic field in the FT either), the line having the right direction and position to be a possible path the particle took after going through the magnet.

Finally, the **fitter** uses all the hits to align the particles path more closely with the position of the hits. In principle, it works similarly to a curve fitting solution, but uses a Kalman filter internally. Having the most accurate path, the decision is made to keep or drop the event.

2.8 The aim of this thesis project

As mentioned, the abandoning of the hardware trigger stage highly increases the load on the software trigger. The main goal of this project is to optimize the current software trigger to make it about 3 times as fast. Failure to do so will result in valuable events being dropped, thus reducing the physics potential of the experiment.

Current computing hardware has changed significantly from the ones the software trigger was originally made for. The even larger gap between memory and CPU speeds demands a more efficient use of CPU caches. Additionally, CPU instruction sets now include SIMD operations, which can, for example, do 4 floating point operations in place of one in the same amount of time. Furthermore, modern CPUs have a complex logic for branch prediction and instruction pipelining, which require code to be tailored to serve them.

To exploit the full capability of current hardware, not only individual pieces of the trigger software need to be changed, but the global data flow also has to be rethought and optimized.

During this thesis project, I will be helping the LHCb collaboration to reach its optimization goals for the software trigger.

3 Choosing optimization targets

Explain the choice of initial choice of algorithms, based on the pie chart diagram and logical reasoning of our goals (i.e. what's needed).

As mentioned in 2.7, the reconstruction consists of individual algorithms which account for the bulk of the computation. (Scheduling the algorithms and culling decisions account for a much smaller CPU load.) It is straightforward to first start optimizing the algorithms which take the largest chunk of available computing power.

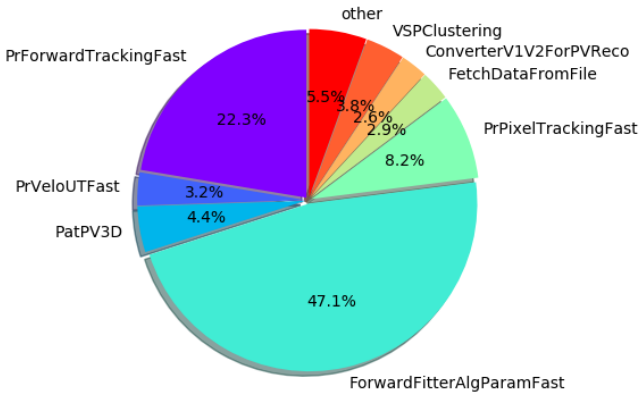


Figure 9: Workload split among HLT1 algorithms.

Looking at figure 9, we can see that the parametrized Kalman fitter takes nearly half the CPU budget, followed by the forward tracking which takes roughly a quarter. Based on this and initial performance profiling of the algorithms for hotspots, I decided to first examine and optimize the Kalman fitter.

4 Parametrized Kalman Fitter

As described in 2.7, the track is reconstructed incrementally, start with velo hits, extended by UT hits and finally adding the FT hits. This process, however, is not so accurate. This manifests itself in the creation of *ghost tracks* and missed tracks, and generally, tracks are only roughly aligned with the hits they were made from. Ghost tracks are tracks that did not exist in the real collision, they are merely artifacts of the reconstruction algorithms. As such, ghost tracks are highly undesirable, but this is where the Kalman fitter comes into play. The Kalman fitter basically refines the rough tracks that are spit out by preceding algorithms. The state of a particle can be described by its position, direction, and the quotient of its charge and momentum. The Kalman fitter first estimates the particle's state at its birth position based on the Velo hits alone. After that, it extrapolates the state of the particle to the next hit, or in other words, simulates the particle's travel until the next hit using the laws of physics. The new, *predicted* state will have some deviation to the

observed state (that is, the hit), however, the Kalman fitter can make a mathematically optimal estimate for the true state based on the prediction and observation. The very new optimal state estimate will then be extrapolated to the next hit again, and this repeats for all the hits of the track. As a result, the estimated state or path of the particle aligns more closely with the observed hits. In the case of ghost tracks, we can expect to have large deviations between the optimal estimated states and the observed hits, which could slipped through initial reconstruction algorithms but show up for the fitter. Such tracks are removed from the list of tracks, and that's why fitting is important.

4.1 Performance profiling for hotspots in the Kalman fitter

Callees	CPU Time: Total ▼	CPU Time: Self
ParameterizedKalmanFit::fit	100.0%	2.950s
ParKalman::LoadHits	49.4%	8.786s
ParameterizedKalmanFit::PredictState	24.4%	7.760s
ParKalman::ExtrapolateToVertex	9.4%	0.400s
ParKalman::UpdateState	8.5%	4.691s
ParKalman::AverageState	6.6%	7.560s
ParKalman::addInfoToTrack	0.9%	0.370s
ParKalman::DoOutlierRemoval	0.2%	1.250s
ParKalman::CreateVeloSeedState	0.1%	0.310s
StatusCode::StatusCode<StatusCode::ErrorCode, voi	0.0%	0.150s

Figure 10: Hotspots, or which parts of the Kalman fitter takes most of the time. Measured by Intel VTune Amplifier XE. MAYBE add appendix explaining profiling and vtune.

Figure 10 shows what fraction of the CPU time is spent in each individual function of the code. We can nicely see how the theoretical steps of the Kalman fitting map to the functions:

- LoadHits: acquires position and measurement error of hits
- PredictState: extrapolates the state to the next hit
- UpdateState: makes an optimal estimate for the true state using the predicted state and the measured hit
- AverageState, ExtrapolateToVertex, etc.: various operations

There is a major and obvious problem however: just acquiring the data on which the computation is done should not take over 50% of the Kalman fitting, but more like 1%.

4.2 Loading hits in detail

Careful examination reveals the way hits are loaded through the so-called *Measurement providers*.

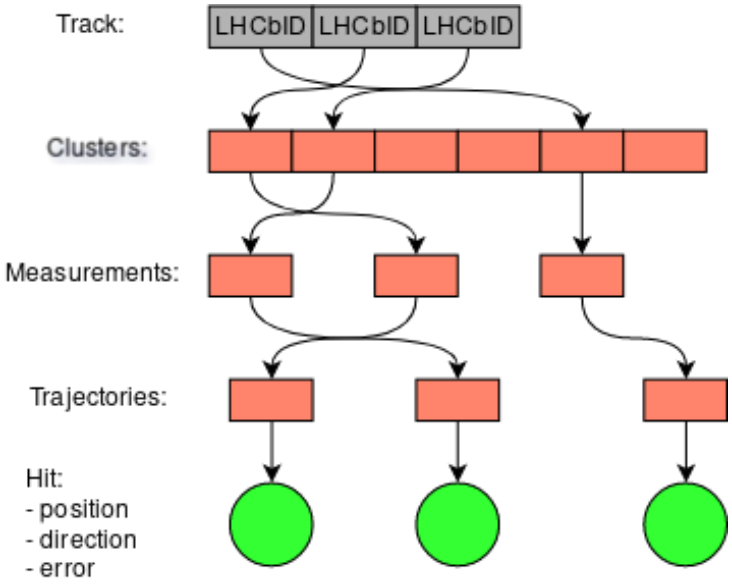


Figure 11: Illustration of how hit information is acquired from the array of LHCbIDs stored inside the Tracks. Contiguous array of clusters correspond to contiguous DRAM memory regions, while distinct objects, i.e. measurements have no spatial locality.

When a particle hits a detector, the identifier of the element of the detector that was hit is recorded. (Detector elements are analogous to the pixels of a digital CCD camera.) These elements are basically unambiguously identified by the so-called *LHCbIDs*, so it is enough to store the IDs inside the Track object and all information (such as location of the hit, measurement error) can be recovered.

Over the years however, this system grow unnecessarily complex resulting in a dramatic slowdown. Clusters, containing some basic information about the hit, such as its location, are stored inside measurement providers as a large array. In order to find the cluster that corresponds to the ID, this whole array is searched linearly. Once the cluster is found, a *Measurement* object is allocated on the heap and initialized from it. Finally, another object, called a *Trajectory*, is queried from the measurement, from which the data actually required can be extracted. The storage of clusters and creation of measurements is handled by *MeasurementProviders*. Additionally, we can distinguish separate measurement objects for the Velo, UT and FT hits.

As seen, this is a convoluted process, involving an asymptotically unacceptable linear search and a lot of dynamic memory allocation. Dynamic allocation is not only slow, it highly suffers from thread contention at the operating system level in our multi-threaded software. Additionally, the individually allocated objects are scattered around in memory, resulting in poor CPU cache performance MAYBE add appendix explaining caches.

Callees	CPU Time: Total ▼	CPU Time: Self
ParKalman::LoadHits	100.0%	8.786
MeasurementProviderT<MeasurementProviderTypes	45.4%	1.340
find_if<__gnu_cxx::__normal_iterator<const LHCb:	27.0%	0
DataObjectHandle<AnyDataWrapper<std::vector<L	12.7%	0
VPClusterPosition::position	2.3%	1.810
LHCb::VPMeasurement::VPMeasurement	1.5%	0.860
operator new	1.1%	3.880
GaudiHandle<IVPClusterPosition>::operator->	0.3%	0.050
LHCb::LHCbID::vpID	0.1%	0.050
func@0x3df1d0	0.0%	0.060
LHCb::LHCbID::isVP	0.0%	0
func@0x3e21a0	0.0%	0.030
FTMeasurementProvider::measurement	42.9%	1.419
std::__find_if<__gnu_cxx::__normal_iterator<LHCb	30.9%	39.068
FTMeasurementProvider::clusters	7.2%	0.080
LHCb::FTMeasurement::init	3.5%	0.371
operator new	0.9%	3.192
func@0x3df1d0	0.0%	0.060
func@0x3dfc30	0.0%	0.020
func@0x3df800	0.0%	0.020
func@0x3e30b0	0.0%	0.020

Figure 12: Breakdown of CPU usage of the LoadHits function

Figure 12 clearly shows an excerpt from the CPU profiler and helps to understand where LoadHits spends its time. The most obvious thing is the std::find_ifs that take nearly 60% of the entire time of LoadHits. This corresponds to the linear search among clusters. The rest of the overhead comes from various boilerplate code, clear trends cannot be understood, but the volume of the overhead is seen to be significant.

4.3 Simplifying the data loading

To avoid this long chain to acquire the required data, the hits should be directly stored inside the Track rather than only by their IDs. Ideally, this would not incur any performance penalty, since the algorithms preceding the Kalman fitter all use the position and error information associated with a hit, so the detector element identifier is fully decoded anyway.

As described, the Track object has the following content (largely simplified):

```
struct Track {
        std::vector<LHCbID> ids;
};
```

In the new model, the following structure is used:

```
struct TrackHit {
        Vector3D beginPosition;
        Vector3D endPosition;
```

```
        float errorX;
        float errorY;
};

struct Track {
        std::vector<LHCbID> ids;
        std::vector<TrackHit> veloHits;
        std::vector<TrackHit> utHits;
        std::vector<TrackHit> ftHits;
};
```

Notice how the IDs are kept: the unfortunate reason for this is that other algorithms rely on these, and they cannot be removed in this first iteration. This structure, however, completely eliminates clusters, measurement and trajectories from the chain, and the Kalman fitter reads the contiguously stored information straight out of the track. This does not stress the memory allocator and is friendly for the caches.

4.4 Performance profiling of the simplified model

Callees	CPU Time: Total ▼	CPU Time: Self
▼ ParameterizedKalmanFit::fit	100.0%	2.340s
▶ ParameterizedKalmanFit::PredictState	45.3%	6.420s
▶ ParKalman::ExtrapolateToVertex	23.7%	0.400s
▶ ParKalman::UpdateState	13.2%	4.020s
▶ ParKalman::AverageState	12.9%	5.950s
▶ ParKalman::addInfoToTrack	2.4%	0.400s
▶ ParKalman::LoadHits	1.2%	2.091s
▶ ParKalman::DoOutlierRemoval	0.2%	0.791s
▶ ParKalman::CreateVeloSeedState	0.1%	0.231s

Figure 13: Breakdown of the Kalman fitter after the simplified data loading

Figure 13 shows that with the new data model, the previous CPU hog, LoadHits, has completely disappeared, now accounting only for 2% of the fitting.

As the parametrized Kalman fitter takes about 47% of the entire reconstruction sequence, and about 50% of the fitter's computing load was removed by the above described code changes, we would expect an overall speedup of 31%. When measured, throughput increases from 4450 events processed per second to about 4850 events/second, or about 9.2%. As this is way less than the predicted 31%, the question arises as to where the performance is gone. First of all, three new data members were added to the Track to store the new TrackHits, and nothing has been removed. As Tracks are copied in the code, the three std::vectors also have to be copied, which involves dynamic memory allocation (a well-known performance drag) and memory copying. Second, part of the code that produces the TrackHits from other objects was not removed, but merely moved out of the fitter to other algorithms. The data conversion to TrackHits, along with adding

the TrackHits to the vectors and allocating the memory of the vectors adds additional overhead. In order to achieve the projected performance improvements, these issues have to be fixed and optimized.

4.5 High level optimizations

Besides the TrackHits (or IDs), the Track contained three additional dynamically allocated std::vectors, which were filled with valid data but were not necessary from a computing point of view. Removing these data members confirmed the hypothesis by which the additional data members in the track slowed down the algorithm sequence: I observed an increase of 17% in throughput (on top of the 9.2%) when removing these members. While this can be regarded as an optimization independent to the fitter itself, it gains back the speed lost with the additional members required by the fitter.

4.6 Micro-optimizations

To trade physics quality for performance, event culling decisions can be made earlier in the reconstruction sequence, before fitting. This results in fitting taking a lot smaller part of the entire sequence while other algorithms become more prevalent. My work, although sped the fitting up, slightly slowed other algorithms down. Consequently, the *best physics* case experienced a large increase in throughput, but the *best throughput* case got slightly slowed down. In an attempt to restore the performance of the other algorithms, I had to further analyze performance.

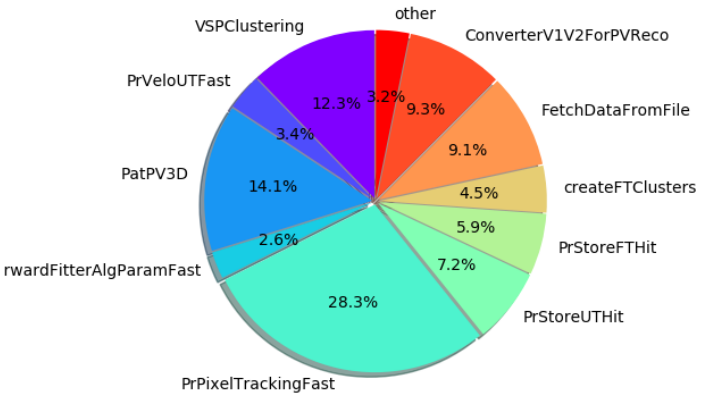


Figure 14: Distribution of CPU time among algorithms in the *best throughput* case with early event culling

Figure 14 shows the in the best throughput case, the pixel tracking algorithm takes the most amount of time. This algorithm is responsible for finding particle track stubs from only Velo hits, and was negatively affected by my fitter optimizations.

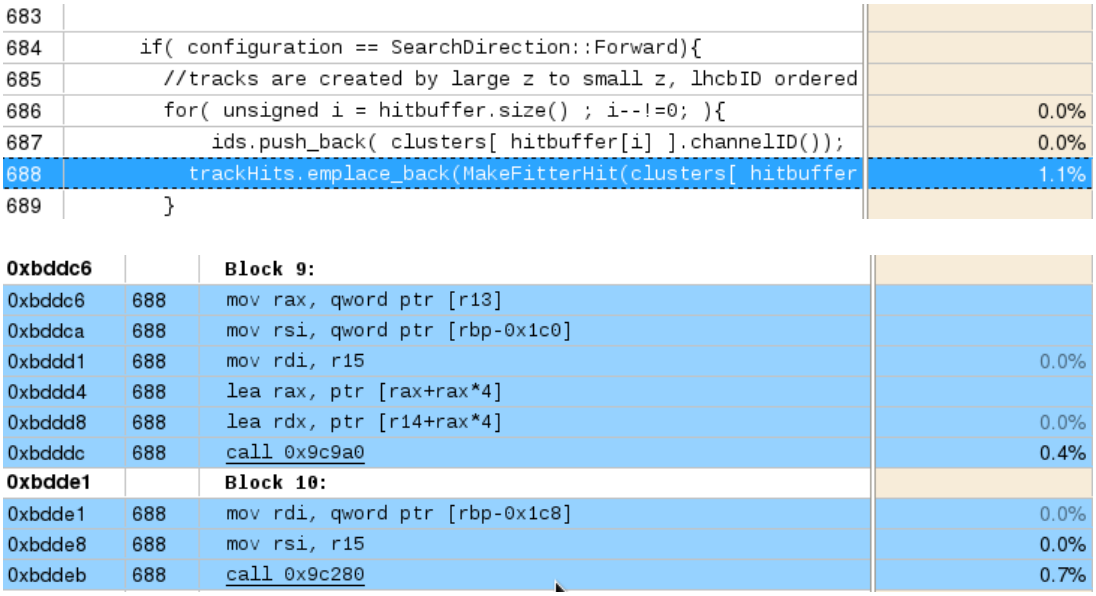


Figure 15: Code snippet from the profiler which shows the code I added to the pixel tracking algorithms in blue highlight. The upper image shows the C++ source code, the lower image shows the corresponding x86-64 disassembly.

Looking at the disassembly, we can see two *CALL* instructions, which correspond to the two function calls *MakeFitterHit* and *emplace_back*. This means that the functions haven't been inlined. Inlining[ref] is a complex topic, because it can make code faster by removing function prologues[ref], but excessive inlining can also make code slower by polluting the instruction caches with two many repeated code snippets. In this case, the latter is unlikely, since these function are only present at this location, so inlining would be preferable.

```
684       if( configuration == SearchDirection::Forward){
685         //tracks are created by large z to small z, lhcbID ordered
686         for( unsigned i = hitbuffer.size() ; i--!=0; ){                    0.0%
687           const auto cluster = clusters[ hitbuffer[i] ];                   0.0%
688             ids.push_back( cluster.channelID());
689
690           //LHCb::TrackHit hit;
691           Gaudi::XYZPointF beginPoint = { cluster.x(), cluster.y()
692
693           // Get the sensor and calculate error.
694           const LHCb::VPChannelID channel = cluster.channelID();
695           const DeVPSensor* sensor = m_vp->sensorOfChannel(channel
696           const unsigned int sensorNumber = sensor->sensorNumber()
697
698           bool isLong = sensor->isLong(channel);                           0.2%
699           float errorx = isLong ? m_errorXLong[sensorNumber] : m_e          0.0%
700           float errory = isLong ? m_errorYLong[sensorNumber] : m_e          0.0%
701
702           trackHits.emplace_back(beginPoint, beginPoint, errorx, e
703         }
704       }
```

```
0xbdc30          Block 9:
0xbdc30   702    mov rbx, qword ptr [rbp-0xd8]
0xbdc37   702    cmp rbx, qword ptr [rbp-0xd0]                              0.0%
0xbdc3e   702    mov rsi, rbx                                               0.0%
0xbdc41   702    jz 0xbdee8 <Block 35>                                      0.0%
0xbdc47          Block 10:
0xbdc47   702    pxor xmm0, xmm0
0xbdc4b   702    mov dword ptr [rbx+0x40], 0x0                              0.0%
0xbdc52   702    add rbx, 0x48                                              0.0%
0xbdc56   702    pxor xmm2, xmm2                                            0.0%
0xbdc5a   702    pxor xmm1, xmm1
0xbdc5e   702    cvtss2sd xmm0, dword ptr [rbp-0x1b0]                       0.0%
0xbdc66   702    cvtss2sd xmm2, dword ptr [rbp-0x1b8]                       0.0%
0xbdc6e   702    movsd qword ptr [rbx-0x38], xmm0                           0.3%
0xbdc73   702    movsd qword ptr [rbx-0x20], xmm0                           0.0%
0xbdc78   702    pxor xmm0, xmm0                                            0.0%
0xbdc7c   702    cvtss2sd xmm1, dword ptr [rbp-0x1a8]                       0.0%
0xbdc84   702    movsd qword ptr [rbx-0x48], xmm2                           0.0%
0xbdc89   702    cvtss2sd xmm0, dword ptr [rbp-0x1c8]                       0.0%
0xbdc91   702    movsd qword ptr [rbx-0x40], xmm1                           0.1%
0xbdc96   702    movsd qword ptr [rbx-0x18], xmm0                           0.0%
0xbdc9b   702    pxor xmm0, xmm0                                            0.0%
0xbdc9f   702    movsd qword ptr [rbx-0x30], xmm2
0xbdca4   702    movsd qword ptr [rbx-0x28], xmm1                           0.0%
0xbdca9   702    cvtss2sd xmm0, dword ptr [rbp-0x1c0]                       0.0%
0xbdcb1   702    movsd qword ptr [rbx-0x10], xmm0                           0.0%
0xbdcb6   686    cmp dword ptr [rbp-0x1a4], 0xffffffff                      0.0%
0xbdcbd   702    mov qword ptr [rbp-0xd8], rbx                              0.0%
0xbdcc4   686    jz 0xbe180 <Block 54>                                      0.0%
```

Figure 16: Source code and disassembly after manually inlining *MakeFitterHit* inside the *for* loop.

As can be seen on figure 16, both *CALL* instruction have disappeared. The first one for *MakeFitterHit* due to the manual inlining, and the second for emplace_back because

of the compiler automatically inlining it. Note that the automatic inlining was enabled by passing the constructor arguments of *TrackHit* to *emplace_back* instead of the ready object, exactly as *emplace_back* was meant to be used. Now, theoretically, the compiler could optimize out both cases as their semantics are equivalent, but it is apparently not capable of doing so.

Besides the absence of function calls, there is another thing noticeable on the assembly instruction. There is a large number of instructions moving quad words (*MOVSD*), that is 64 bit double precision numbers. Furthermore, the *CVTSS2SD* instructions are converting 32 bit single precision numbers to 64 bit doubles. Looking at the source code, we can indeed notice that input data from which the *TrackHit* is made is stored as single precision floats, but the *TrackHits* themselves are double precision because the fitter is using double precision calculations. Changing *TrackHit* to store single floats as well, thus delaying the conversion, will hurt performance at another place where it has less of an impact.

0xbdc28		**Block 9:**	
0xbdc28	702	mov rbx, qword ptr [rbp-0xd8]	
0xbdc2f	702	cmp rbx, qword ptr [rbp-0xd0]	0.0%
0xbdc36	702	mov rsi, rbx	0.0%
0xbdc39	702	jz 0xbded8 <Block 35>	0.0%
0xbdc3f		**Block 10:**	
0xbdc3f	702	movss xmm4, dword ptr [rbp-0x1b8]	
0xbdc47	702	mov dword ptr [rbx+0x20], 0x0	0.0%
0xbdc4e	702	add rbx, 0x24	0.0%
0xbdc52	702	movss xmm5, dword ptr [rbp-0x1a8]	0.0%
0xbdc5a	702	movss xmm6, dword ptr [rbp-0x1b0]	
0xbdc62	702	movss xmm1, dword ptr [rbp-0x1c8]	0.0%
0xbdc6a	702	movss xmm7, dword ptr [rbp-0x1c0]	0.0%
0xbdc72	702	movss dword ptr [rbx-0x24], xmm4	0.0%
0xbdc77	702	movss dword ptr [rbx-0x20], xmm5	0.0%
0xbdc7c	702	movss dword ptr [rbx-0x1c], xmm6	0.2%
0xbdc81	702	movss dword ptr [rbx-0x18], xmm4	0.0%
0xbdc86	702	movss dword ptr [rbx-0x14], xmm5	0.0%
0xbdc8b	702	movss dword ptr [rbx-0x10], xmm6	0.0%
0xbdc90	702	movss dword ptr [rbx-0xc], xmm1	0.0%
0xbdc95	702	movss dword ptr [rbx-0x8], xmm7	0.0%
0xbdc9a	686	cmp dword ptr [rbp-0x1a4], 0xffffffff	0.0%
0xbdca1	702	mov qword ptr [rbp-0xd8], rbx	0.0%
0xbdca8	686	jz 0xbe168 <Block 54>	0.0%

Figure 17: Disassembly after changing *TrackHits* to store single floats.

The disassembly on figure 17 clearly shows that the single precision to double precision conversions are gone just like the *PXOR* instructions, and now it only moves double word memory units. With this little change, I managed to throw out lots of unnecessary instruction and the amount of memory moved around is also smaller.

Due to the complex interactions inside modern, pipelined CPUs and between the CPU, the DRAM and caches, it is hard to explain how and why the changes affected the performance. Nevertheless, inlining and trimming the assembly code has increased performance of the *best throughput* scenario with early event culling from 13300 events per second to around 13700. Notably, the basic case without my code changes has produced about 14500 events per second. (As measured with my development branch on our

performance test machines during development.)

4.7 Results, conclusion

I managed to significantly increase the throughput of the *best physics* case from 4450 events per second to 5870 events per second, or a 32% increase. Unfortunately, the *best throughput* case slowed down from 14500 events/sec to about 13700, or 5% decrease in throughput.

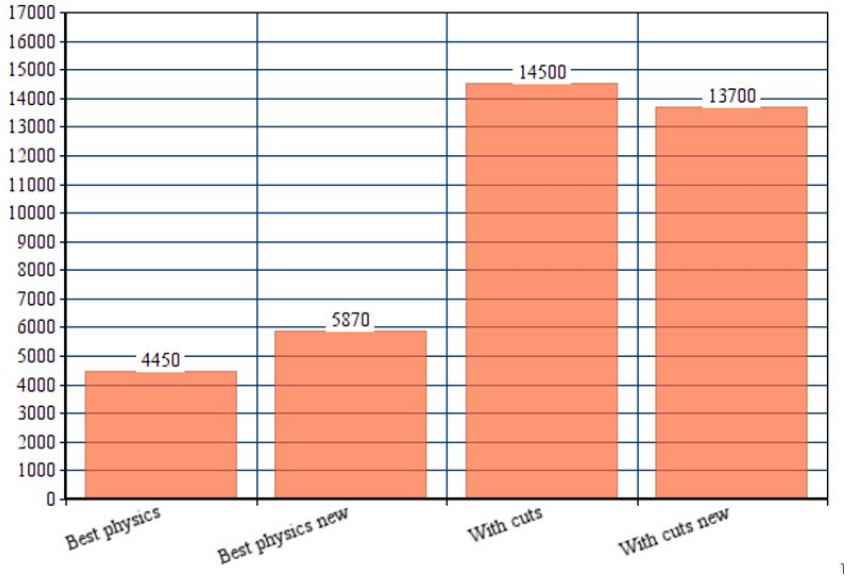


Figure 18: Throughput of the particle path reconstruction before and after my modifications, for the *best physics* and *best throughput* cases.

Another interesting figure to look at is how the weight of the parameterized Kalman fitter in the entire algorithm sequence has changed. Previously, it took 47% of the whole sequence, and now it only takes 22%. If in addition the fact that the whole sequence is significantly faster is factored in, the fitter is well above two times faster. (These type of measurements are to be taken with a grain of salt because of complex system interactions and the consequent inaccuracy of profiling, but are interesting and provide a good general view.)

(a) Before optimizations

(b) After optimizations

Figure 19: The distribution of processor time requirements of each algorithms. The parameterized Kalman fitter is identified by the label *ForwardFitterAlgParamFast*.

In light of the code changes and their effect on the overall performance, we can safely say that code should strive to do the data transformations in the simplest possible way. Adding extra layers on top, if not done carefully using zero-cost abstractions, will dramatically slow the code down. In performance critical applications, a good data oriented design can give far better benefits that assembly-level micro-optimizations. Additionally, a good data structures opens up the doors to more effective micro-optimizations, such as vectorization[ref].

5 Streamlining the computation's data model

6 Vectorizing and optimizing the Velo-UT algorithm

As highlighted in 2.7, the Velo-UT algorithm extends the straight line tracks created in the VELO tracking algorithm by assigning UT silicon strip hits to it. More importantly, the Velo-UT algorithm provides a course estimate for the momentum of the particle, allowing efficient tracking in the FT. The algorithm takes roughly 10 percent of the pipeline, so optimizing it is not expected to provide a high overall gain in performance, however its code is outdated and is in much need of an overhaul. My goal for the Velo-UT was to streamline the code to adhere to modern coding practices, and in the meantime, to make the algorithm play well with modern CPU architectures. In light of this, I was aiming at a highly SIMD-vectorizedTODO: add appendix code, which operates with SOATODO: add appendix data structures

6.1 Geometry of the UT detector

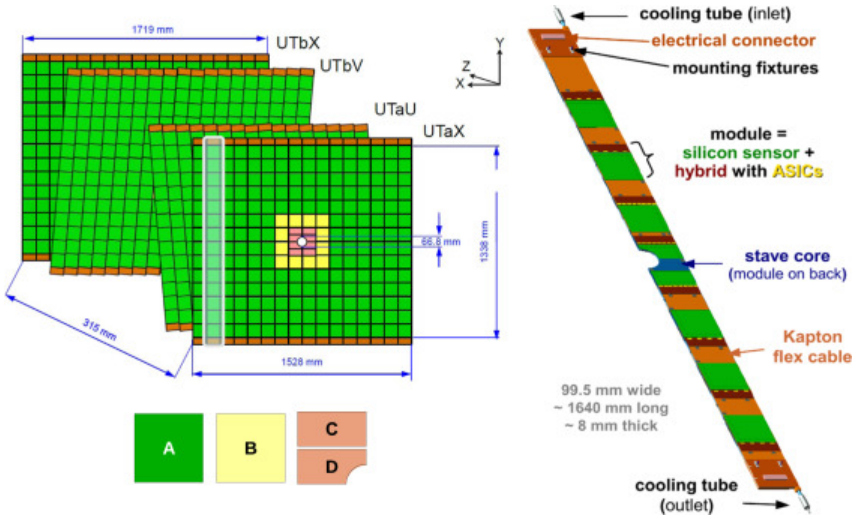


Figure 20: Illustration of the key elements of the UT hardware.

Figure 20 shows the rectangular panels of the UT as looking down the detector from the VELO. The UT detector consists of four panels, their placement in the entire detector can be seen on figure 3. Each of the four panels is made up 10 cm by 10 cm sensors. Though both the front two and rear two panels contain 14 rows of sensors, the rear panels are wider. The middle two panels have their sensors tilted by +5 and -5 degrees, which helps to mitigate the poor vertical resolution resulting from using silicon strips instead of pixel detectors. Each sensor contains silicon strips that run the full length of 10 cm of the sensor. There are 512 strips next to each other on every sensor. The exceptions are the sensor near the center of the panels, since the ones marked in yellow contain 1024 strips, and the ones marked in red contain 1024 half-length strips. The higher density area in the middle is required to deal with increased luminousity in the axis of the beam pipe. For each collision event, the software receives a list of the silicon strips that were hit by a particle. The Technical Design Report for the LHCb tracker[9] explains the geometry in more detail.

6.2 Decoding of the raw data from the detector

Each silicon strip has a unique numeric identifier. During a collision event, the electronics of the UT detector compile a list of identifiers of the strips that were hit by a particle, which the software receives in a heaavily compressed format. With a description of the exact geometry of the detector, that is, the location and dimension of sensors and strips in the global 3D space, the software is able to represent the strips that were hit by a line segment in the 3D space instead of just an ID. The UTHits resulting from the decoding process are gathered into a HitHandler structure, which groups the hits by which sensor they belong to. The Velo-UT algorithm consumes the HitHandler alongside the VELO tracks to produce *upstream* tracks.

6.3 The original Velo-UT algorithm

The main idea of the Velo-UT algorithm is to consume the VELO tracks and the UT hits, and extend the VELO tracks with hits from the UT to create so called *upstream* tracks. The algorithm handles each VELO track separately. The process for a single VELO tracks can be broken down into the following steps:

1. Skip track if unstuitable
2. Extrapolate VELO track to the UT detector
3. Gather UT hits that are close to the extrapolated track
4. Find 3 or 4 UT hits that align in a line
5. Analyze the *track candidates* that consist of the VELO track + 3-4 UT hits
6. Select the best candidate (if any) to extend the VELO track

6.3.1 Filtering tracks

Any kind of tracks can be fed into the algorithm from the VELO tracking. Some tracks point to the wrong direction, away from the UT, other tracks go very close to the beamline and hit the central hole in the UT's panels, and some tracks may also simply miss the UT's panels as they go too much to the sides. These tracks are not worth trying to extend because there will be no solution, so they are dropped.

6.3.2 Extrapolating tracks

Tracks from the VELO are described by the *state* of the particle. The state is a tuple of x,y and z coordinates, and the slopes tx=dx/dz and ty=dy/dz. In other words, the particle has a position [x,y,z] and has a velocity vector in the direction [tx,ty,1]. The usual coordinate system (see 2.4.2) is used for the states as well. To gather UT hits, one has to know where a track crosses a particular panel of the UT, which can be calculated by extrapolating the x,y,z position of the state to the z position of the panel along to line given by the slopes.

6.3.3 Gathering UT hits

Once the state is extrapolated to the z coordinate of a panel, one can collect hits that are close to the x and y position of the extrapolated state. The implementation of the algorithm first collects the list of the 10 by 10 centimeter sensors that fall close enough to the extrapolated state. In case the track passes through the middle of one sensor, only one sensor will be tagged, however if the track hits the corner between four sensors, all the four may be tagged. Afterwards, the algorithm iterates through all the hits that belong to the tagged sensors, and checks for each hit if they are within tolerance to the extrapolated state. This, including the extrapolation, is repeated for all four panels, thus the final result is a set of close-by hits for each of the four panels.

6.3.4 Finding aligned hits

There are on average 2-3 hits returned for each panel. Among these hits, the algorithm tries to find 3 or 4 hits that form a fairly straight line. These are called track candidates, and since there may be multiple sets of hits that form a straight line, the algorithm must decide on which is the best candidate.

6.3.5 Analyzing candidates

Each candidate that belongs to a track goes through a fitting stage. This is much like curve fitting or linear regression, the algorithm find a mathematically optimal set of parameters that best describe the path of the particle that have created the UT hits of the candidate.

6.4 Selecting the best candidate

The error of the fitting of a candidate can be calculated by determining how far the parameterized path of the particle lies from the position of the UT hits it was fitted from. If any of the candidates have a sufficiently low error, the one with the lowest is picked to form the newly created upstream track.

6.5 Implementation and its pitfalls for the original Velo-UT algorithm

6.5.1 Gathering hits

TODO: add picture from profiler

The largest part of the Velo-UT algorithm is collecting the hits that belong to a track, which can be seen on figure ref to figure. Upon closely inspecting the code, we can see a complex flow control logic and several calls to distant services.

The algorithm loops over the four panels, extrapolates to track to them, and finds the sensors that fall close to the extrapolated track. In the next step, the algorithm loops over the found sensors, and collects each hit that falls within a certain distance of the extrapolated track.

One issue with this is that there is no need to base the lookup of hits close to a track on sensors, any kind of space partitioning can be used to make it more efficient. Second, the grouping of hits by sensors that is done by the HitHandler, requires the HitHandler to either perform a complete sorting of the hits by their sensor or to receive the hits in a particular order. The former option takes O(nlogn) time, while the latter make the implementation error-prone and fragile to change. Groouping by sensors could be implemented in a robust and efficient manner, but migrating to uniform grid space partitioning seemed like a better solution.

6.5.2 Lack of vectorization

The Velo-UT algorithm processes tracks one by one. If, instead, the steps6.3 were executed on all tracks at once, vectorization opportunities could be exploited. Some parts, such as the extrapolation of tracks can be perfectly vectorized, other parts, like gathering the hits are more difficult, while the finding of aligned hits is not feasible at all. With SIMD, 4 to

16 (depending on the CPU ISA) tracks can be extrapolated at the same time instead of just one.

6.5.3 Overcomplicated flow control

do a microarch exploration and attach branching inefficiency profiling The implementation heavily relies on early exit conditions, that is, the code detects early on if the results will be unsatisfactory, and skips the execution of the remaining code. While this seems intuitive to cut computation times back, it may achieve the opposite effect on modern CPUs. Each time a CPU hits a branching point in the code (e.g, if statement), it tries to guess which branch will be chosen, and starts executing the predicted branch before the condition of the if statement is evaluated. If the CPU guessed wrong, it has to unroll the entire branch, and restart the correct one. While the CPU generally makes very accurate predictions, the cost of a misprediction is high, which gives motivation to structure code in a branchless fashion. Additionally, branchless code is easier to vectorize.

6.5.4 Flawed code design

The code violates several coding best practices, which is not strictly an optimization problem, but is still an important concern as it is hard to find optimization opportunities in code that is hard to reason about. The most severe breaches are the single responsibility and dependency inversion priciples. An example for the former is formClusters, which not only find aligned hits but also parametrically fits them, an example for the latter is simpleFit, which modifies the flow control of its caller (formClusters) via modification of in-out (quasi-global) parameters. When the problems are alleviated, the search for aligned hits and the fitting can be done completely separately, which provides a massive gain thanks to the perfect vectorization of the fitting process.

7 Conclusion

- summarize my own contributions
- summarize achieved results
- make conclusions about them
- how it affects the future
BRIEFLY

8 References

[1] About CERN:
https://home.cern/about

[2] The accelerator complex:
https://home.cern/about/accelerators

[3] About the Large Hadron Collider:
https://home.cern/topics/large-hadron-collider

[4] About the Large Hadron Collider beauty experiment:
https://home.cern/about/experiments/lhcb

[5] Why collide lead ions:
http://alicematters.web.cern.ch/?q=FAQ-why-lead-ions

[6] Energy of the LHC:
https://home.cern/about/engineering/restarting-lhc-why-13-tev

[7] LHC collisions:
https://lhc-machine-outreach.web.cern.ch/lhc-machine-outreach/collisions.htm

[8] LHC facts and figures:
https://public-archive.web.cern.ch/en/LHC/Facts-en.html

[9] Tracker (UT and FT) Technical Design Report:
https://cds.cern.ch/record/1647400?ln=en