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 $\rightarrow$ 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 The accelerator complex [2]

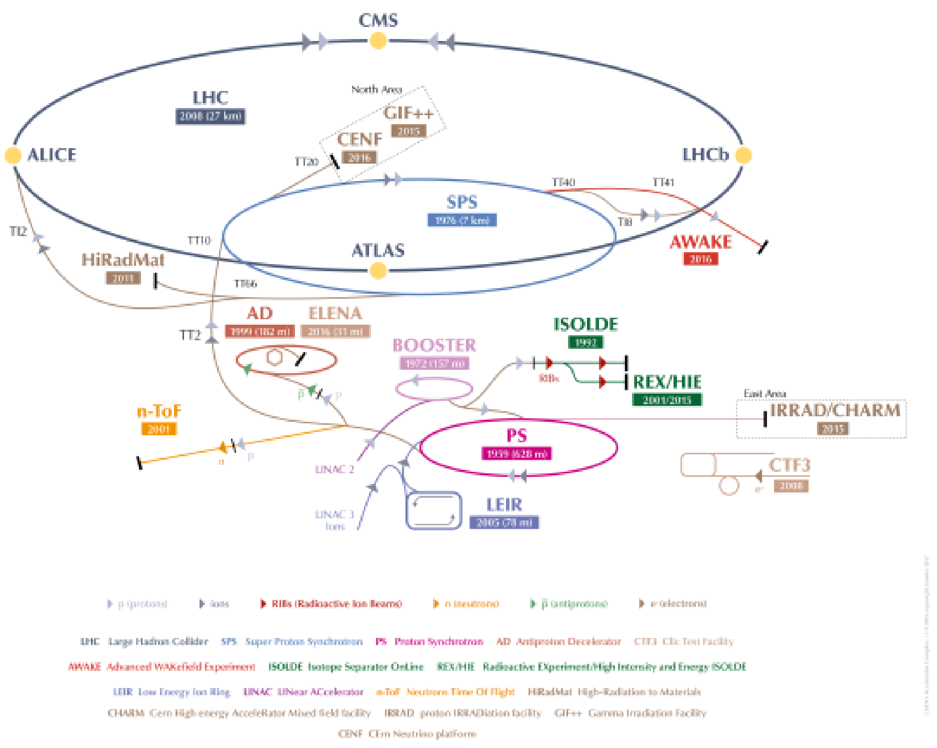

CMS
LHC
2008 (27 km)
North Area
GIF++
2015
CENF
2016
TT20
ALICE
LHCb
TT40
TT41
SPS
1976 (7 km)
TI8
TI2
TT10
ATLAS
AWAKE
2016
HiRadMat
2011
TT66
AD
1999 (182 m)
ELENA
2016 (31 m)
TT2
BOOSTER
1972 (157 m)
ISOLDE
1992
RIBs
REX/HIE
2001/2015
East Area
IRRAD/CHARM
2015
n-ToF
2001
PS
1959 (628 m)
CTF3
2008
LINAC 2
LINAC 3
Ions
LEIR
2005 (78 m)
p (protons)
ions
RIBs (Radioactive Ion Beams)
n (neutrons)
p̄ (antiprotons)
e⁻ (electrons)
LHC Large Hadron Collider
SPS Super Proton Synchrotron
PS Proton Synchrotron
AD Antiproton Decelerator
CTF3 Clic Test Facility
AWAKE Advanced WAKefield Experiment
ISOLDE Isotope Separator OnLine
REX/HIE Radioactive EXperiment/High Intensity and Energy ISOLDE
LEIR Low Energy Ion Ring
LINAC LINear ACcelerator
n-ToF Neutrons Time Of Flight
HiRadMat High-Radiation to Materials
CHARM Cern High energy AcceleRator Mixed field facility
IRRAD proton IRRADiation facility
GIF++ Gamma Irradiation Facility
CENF CErn Neutrino platForm

Figure 1: 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.

While CERN is mostly known for its Large Hadron Collider (LHC) which this thesis is concerned with, it is home to many more particle accelerators. These accelerators are useful on their own, but from the perspective of the LHC they are used to provide high energy protons that the LHC can further accelerate. Too low energy protons cannot be directly accepted into the LHC, so a sequence of progressively larger accelerators bump the energies up in steps. When a particular accelerator reached its top energy, its beam is simply transferred to a bigger one, finally getting injected into LHC.

The LHC is CERN's largest machine. It can be found inside a circular underground tunnel of a circumference of 27km. There are two accelerators inside the tunnel which accelerate protons so that there is one beam clockwise and another counter-clockwise. Protons are not equally distributed in the beams as they circle around, rather, they can be found in many equally spaced *bunches*.

At specific points along the circle of the LHC, the two beams of opposing directions are made to cross each other's path. As two bunches go through the crossing point at the same time and the individual protons collide[7]. Since each proton carries around 7 TeV of energy, the collision's yield is about 14 TeV. In the collision, other particles might be born, and that's exactly what scientists are looking forward to analyze.

2.3 Experiments on LHC

As seen on figure 1, the four main experiments dedicated to analyze LHC collisions are ATLAS, CMS, ALICE and LHCb. Consequently, these experiments have huge underground rooms around the collisions points, where they can fit their instruments.

The instruments are meant to track and identify particles created in the collisions, and are thus called particle *detectors*. The type and properties of particles created in the collisions 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, which does the tracking and identification.

2.4 LHCb's detector

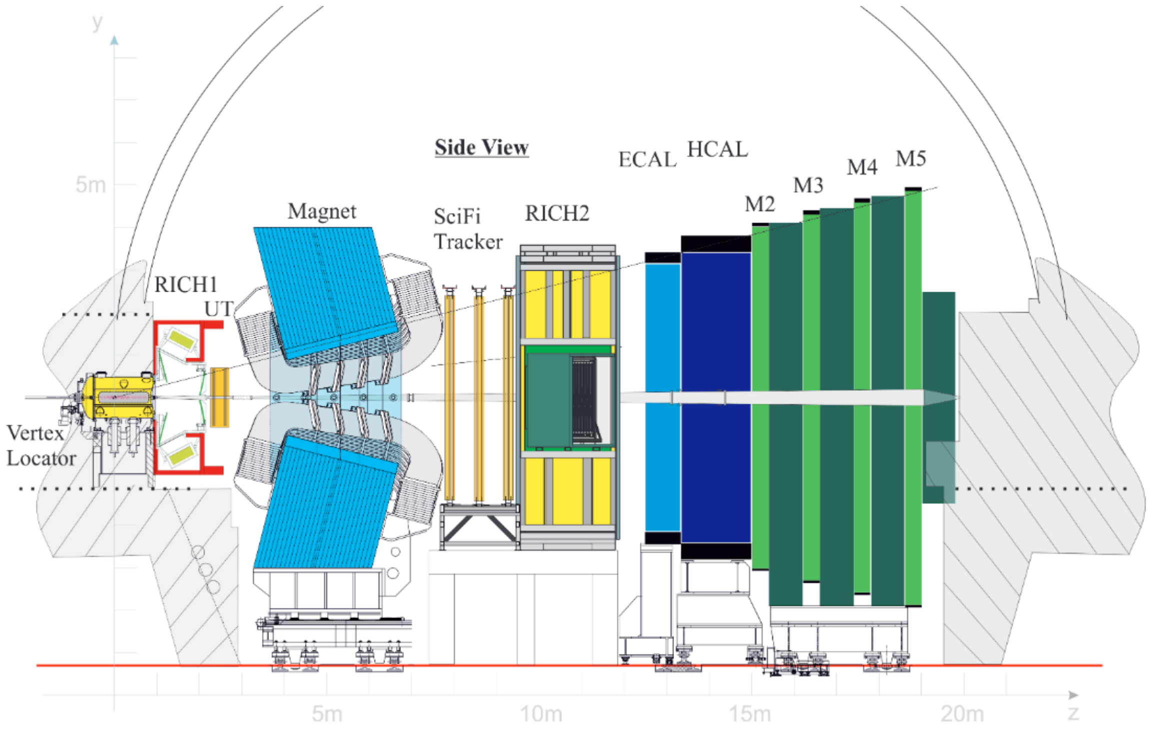


Figure 2: Side view of the LHCb detector.

Figure 2 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 middle of the detector coincides with the horizontal axis of symmetry.

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).

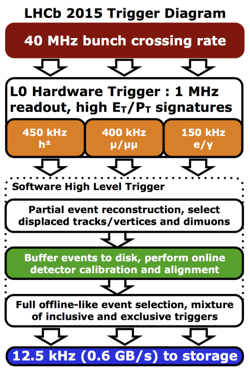
While full reconstruction uses all sub-detectors, real-time reconstruction only uses the VELO, the UT (in orange, left of VELO) and the FT (SciFi Tracker, in the middle in orange). Let's follow the life of a particle from its birth inside the VELO. While flying away from the collision point, it first crosses multiple layers of the VELO. Each layer consists of many pixels, and whenever the particle touches a pixel, the VELO forwards this information to the software. Eventually, the particle leaves the VELO and goes through the UT. The UT has a different mechanism compared to the VELO, but essentially it also records if a particle has gone through one of its layers. Leaving the UT, the particle goes through the large magnet, where its path is bent due to the Lorentz force by an amount dependent on its electric charge and momentum. The particle then keeps going straight through the FT, where its location is registered as usual. Note that for each bunch collision, hundreds of new particles go through the detector as described above. The reconstruction software tries to make sense out of the large number of registered particle positions by using knowledge about the magnetic field, and eventually identifies the individual particles and their tracks.

The purpose of real-time (sometimes called *online*) reconstruction is to determine what data to store. Most collision events are not interesting at all from a physics perspective, and only a small fraction is kept for long-term storage. Storing all data would be unfeasible because of its sheer amount. The software doing the online reconstruction and selection is called the *trigger*.

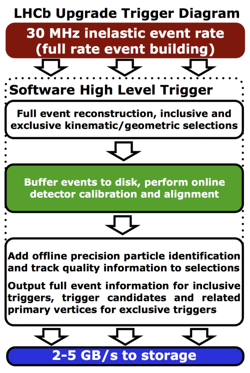
2.5 The 2018/19 upgrade of LHCb

The catch with the above described detector is that it does not really exist yet. The LHC will be shut down in 2018 december for maintenance, and that's when the LHCb collaboration will upgrade its detector to the one above. (The current detector is similar in construction, that's why it's referred to as *upgrade*.)

With the upgrade, software data processing will change significantly as well. Around 30 million bunch collision events occur every second, each of which go through the trigger. With the current detector, triggering is first done by hardware electronics, selecting only 1 million events per second, which are then further culled on a big server farm by software written in C++. After the upgrade, the hardware trigger is dropped and the entire trigger runs in software. This puts the high requirement on the software trigger to reconstruct and cull all the 30 million events every second.


LHCb 2015 Trigger Diagram
40 MHz bunch crossing rate
L0 Hardware Trigger : 1 MHz readout, high E_T/P_T signatures
450 kHz $h^\pm$
400 kHz $\mu/\mu\mu$
150 kHz e/γ
Software High Level Trigger
Partial event reconstruction, select displaced tracks/vertices and dimuons
Buffer events to disk, perform online detector calibration and alignment
Full offline-like event selection, mixture of inclusive and exclusive triggers
12.5 kHz (0.6 GB/s) to storage

(a) Trigger data flow scheme before the upgrade


LHCb Upgrade Trigger Diagram
30 MHz inelastic event rate (full rate event building)
Software High Level Trigger
Full event reconstruction, inclusive and exclusive kinematic/geometric selections
Buffer events to disk, perform online detector calibration and alignment
Add offline precision particle identification and track quality information to selections
Output full event information for inclusive triggers, trigger candidates and related primary vertices for exclusive triggers
2-5 GB/s to storage

(b) Trigger data flow scheme post-upgrade

Figure 3: Comparison of the two triggering solutions

2.6 Overview of the trigger software

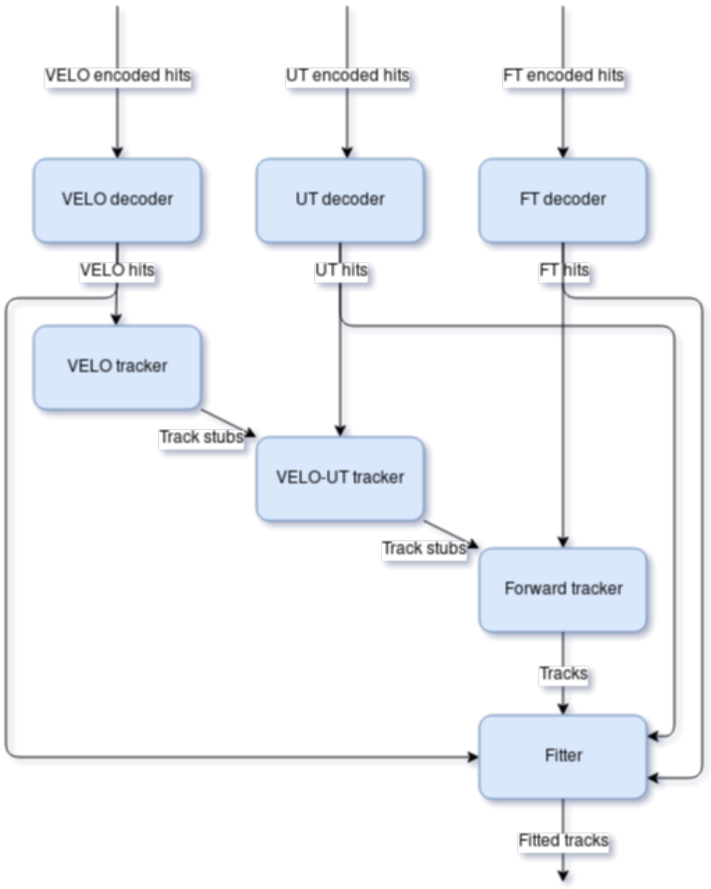


Figure 4: Simplified view of algorithms that perform online reconstruction.

The trigger software is basically the online reconstruction code and additional logic to select which events to keep. The online reconstruction software can be broken down to individual pieces referred to as *algorithms*, each of which serves a specific purpose. First of all, the highly compressed data that arrives from the detector has to be decoded to acquire hits that describe the position and measurement error of points where a particle has been seen. Second, the particles' tracks are progressively reconstructed by starting from the collision point in the VELO, and appending hits to an existing track. In the VELO and UT, we are looking for hits that form a straight line, and we try to match this straight track to another similarly straight track inside the FT. Finally, the track goes through fitting, which tries to modify the existing rough track to more closely line up with the hits it was produced from.

2.7 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 2.6, 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.

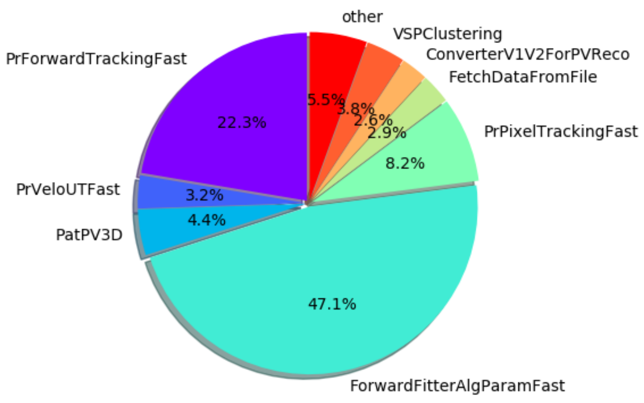


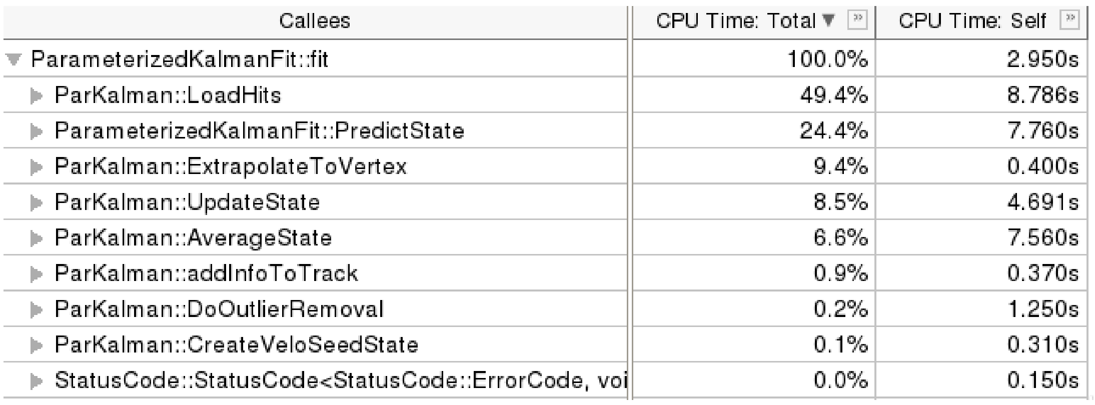
Figure 5: Workload split among HLT1 algorithms.

Looking at figure 5, 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 2.6, 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 6: 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 6 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*.

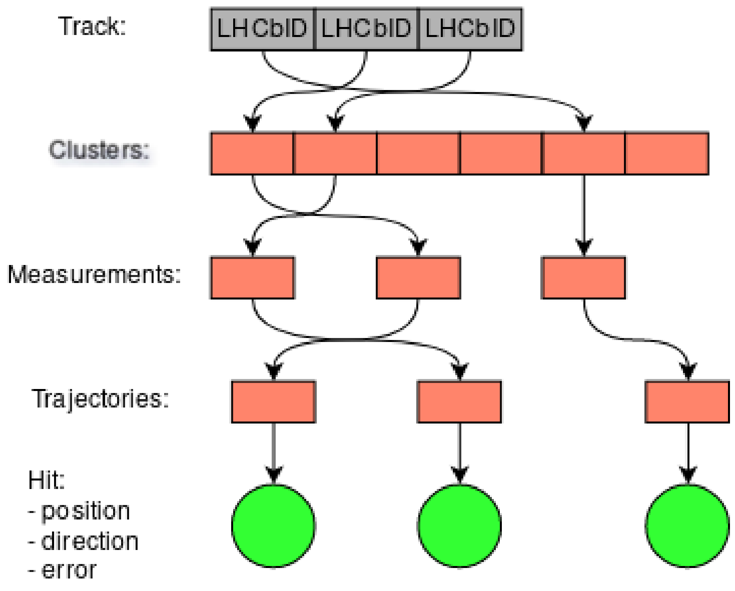
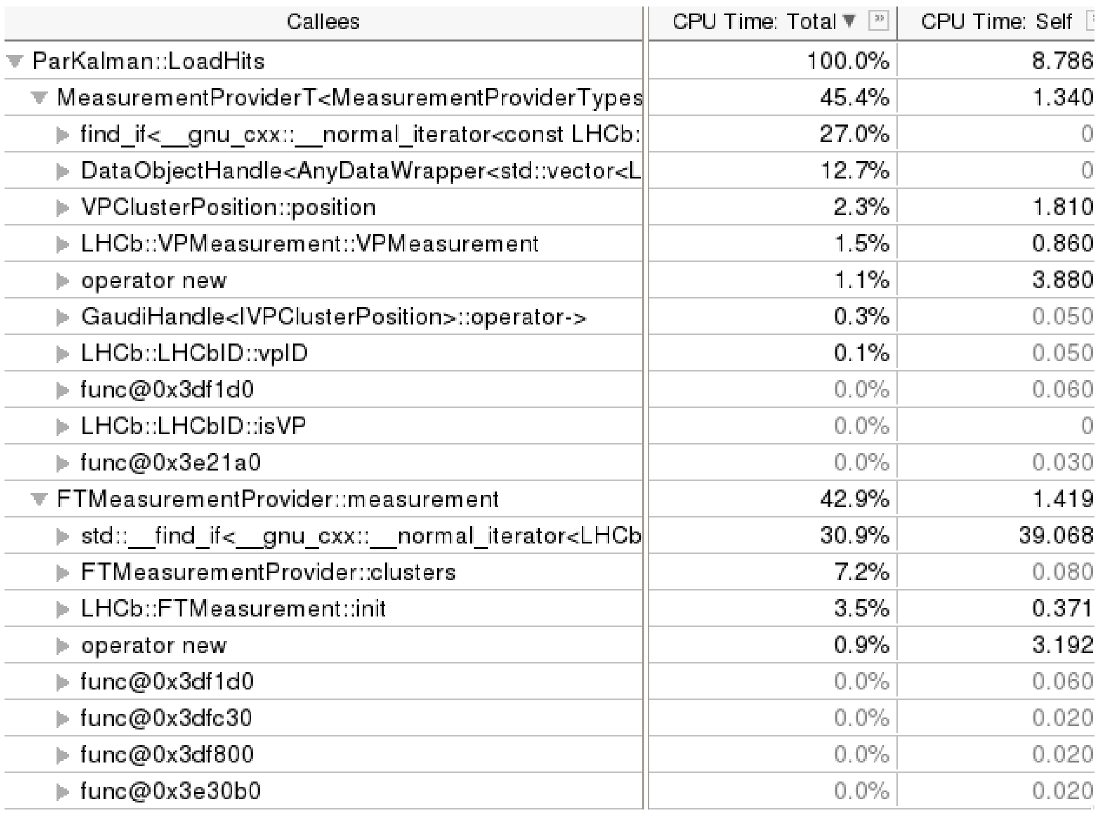


Figure 7: Illustration of how hit information is acquired from the array of LHCbIDs stored inside the Tracks. Contiguous array of clusters correspond to contiguous DRAM memory regions, while distinct objects, i.e. measurements have no spatial locality.

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 8: Breakdown of CPU usage of the LoadHits function

Figure 8 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;
};
```

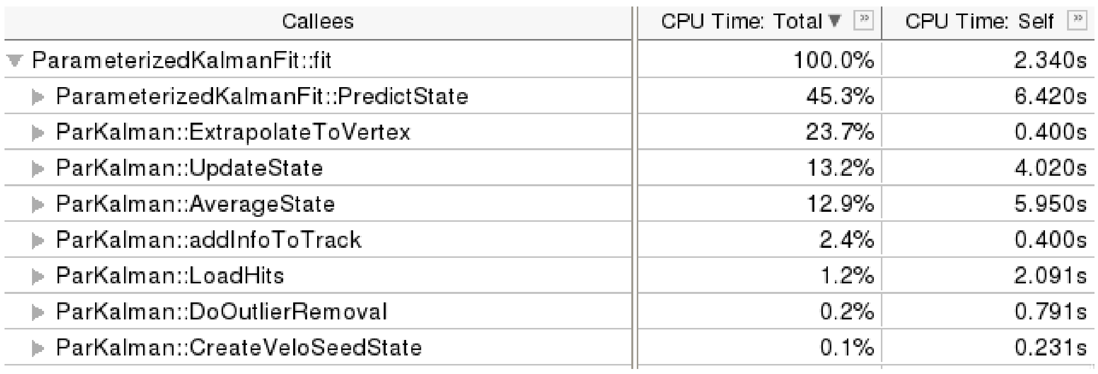
In the new model, the following structure is used:

```
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 9: Breakdown of the Kalman fitter after the simplified data loading

Figure 9 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.

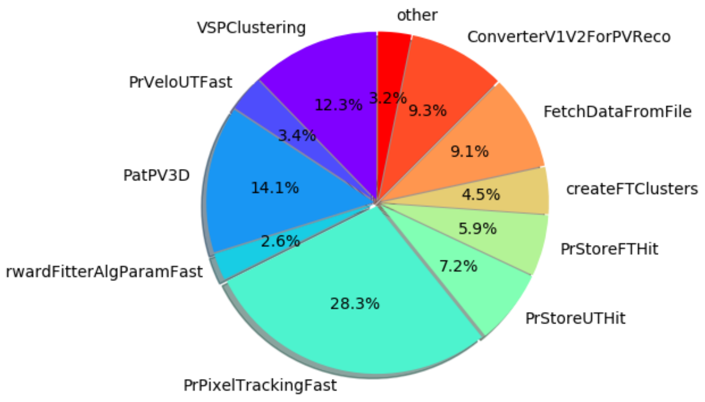
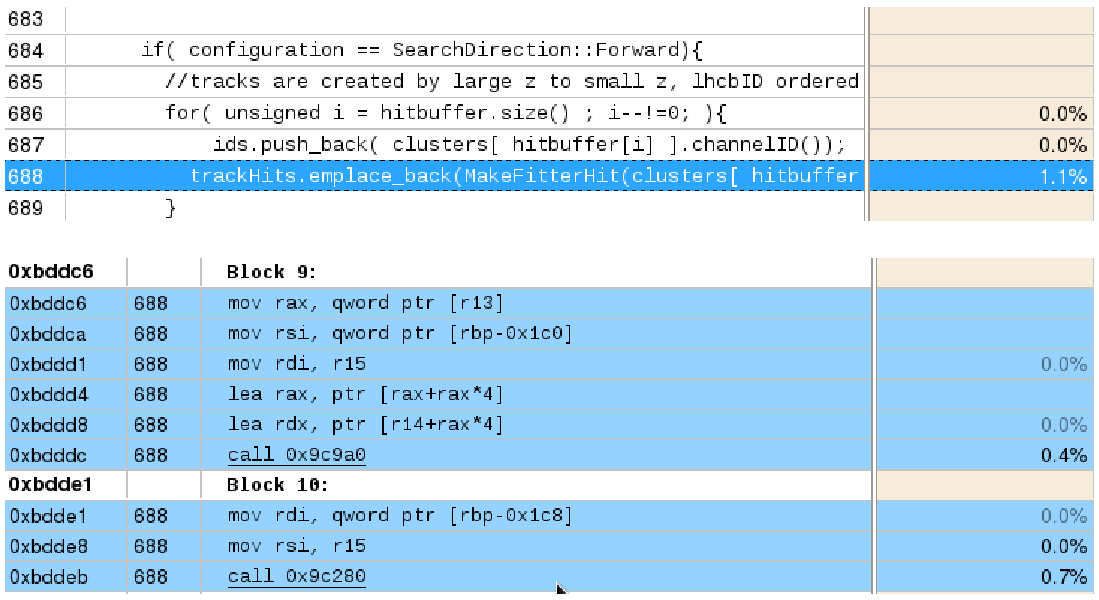


Figure 10: Distribution of CPU time among algorithms in the *best throughput* case with early event culling

Figure 10 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.

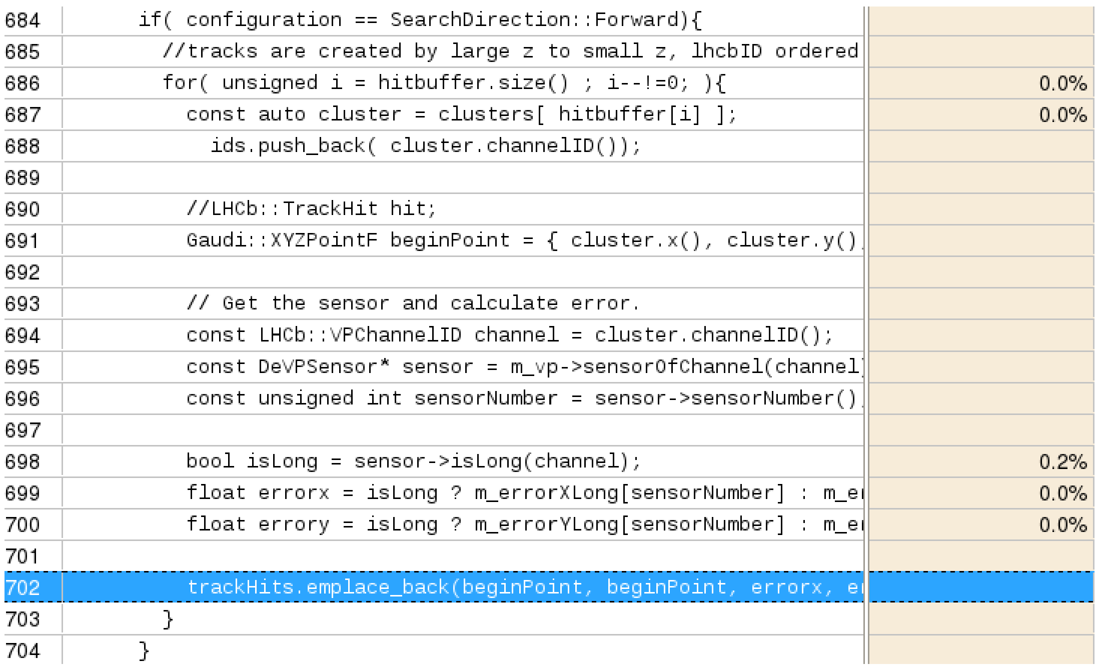


```
683
684        if( configuration == SearchDirection::Forward){
685          //tracks are created by large z to small z, lhcbID ordered
686          for( unsigned i = hitbuffer.size() ; i--!=0; ){                0.0%
687              ids.push_back( clusters[ hitbuffer[i] ].channelID());      0.0%
688            trackHits.emplace_back(MakeFitterHit(clusters[ hitbuffer      1.1%
689          }
```

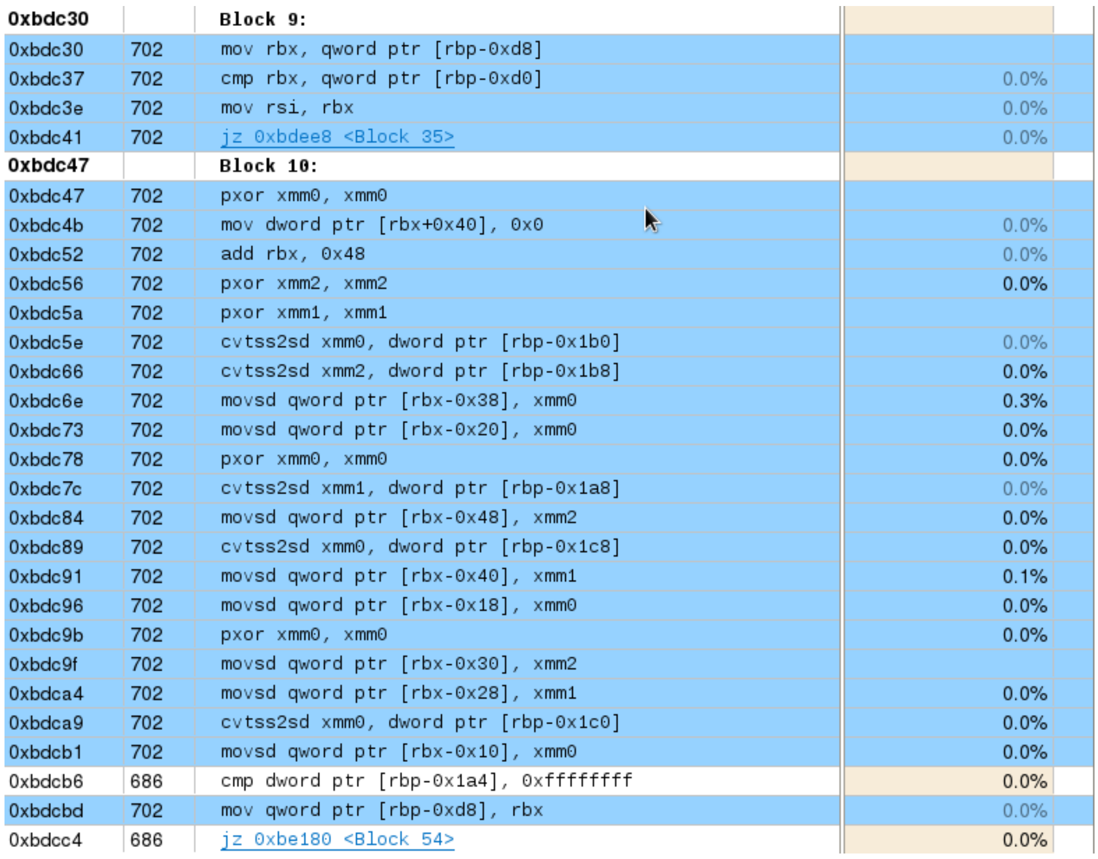
```
0xbddc6            Block 9:
0xbddc6    688     mov rax, qword ptr [r13]
0xbddca    688     mov rsi, qword ptr [rbp-0x1c0]
0xbddd1    688     mov rdi, r15                                0.0%
0xbddd4    688     lea rax, ptr [rax+rax*4]
0xbddd8    688     lea rdx, ptr [r14+rax*4]                    0.0%
0xbdddc    688     call 0x9c9a0                                0.4%
0xbdde1            Block 10:
0xbdde1    688     mov rdi, qword ptr [rbp-0x1c8]              0.0%
0xbdde8    688     mov rsi, r15                                0.0%
0xbddeb    688     call 0x9c280                                0.7%
```

Figure 11: 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       }
```

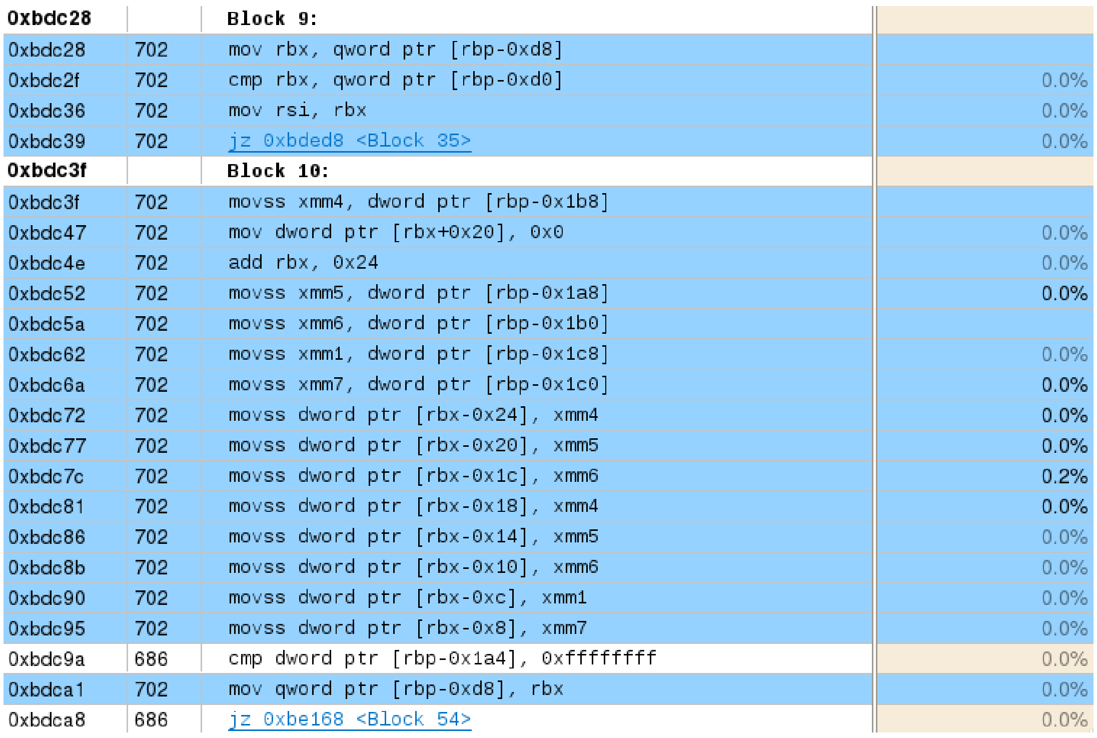


```
0xbdc30           Block 9:
0xbdc30   702     mov rbx, qword ptr [rbp-0xd8]
0xbdc37   702     cmp rbx, qword ptr [rbp-0xd0]                    0.0%
0xbdc3e   702     mov rsi, rbx                                     0.0%
0xbdc41   702     jz 0xbdee8 <Block 35>                            0.0%
0xbdc47           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 12: Source code and disassembly after manually inlining *MakeFitterHit* inside the *for* loop.

As can be seen on figure 12, 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 13: Disassembly after changing *TrackHits* to store single floats.

The disassembly on figure 13 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.

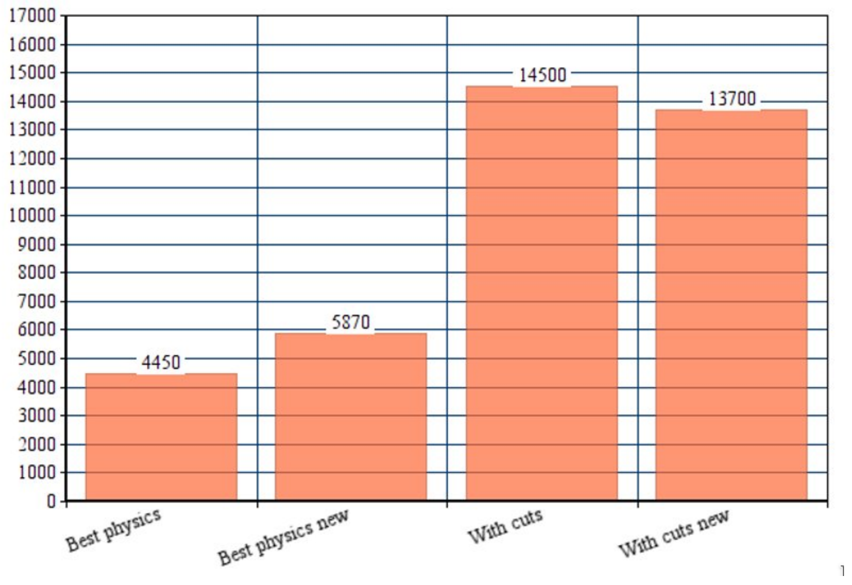
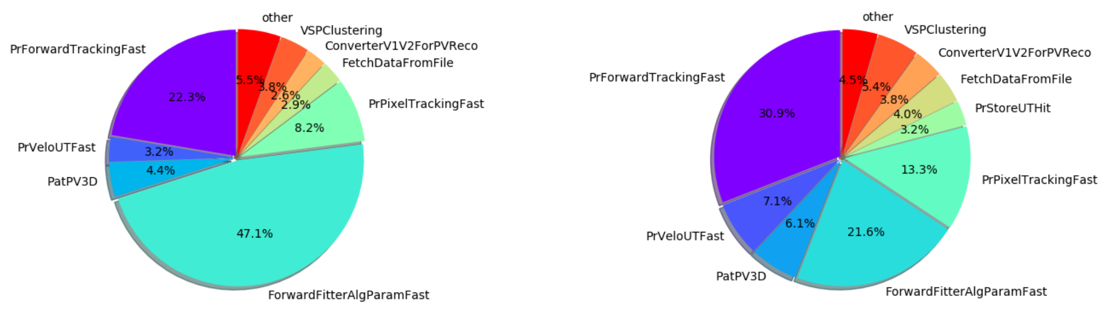


Figure 14: Throughput of the particle path reconstruction before and after my modifications, for the *best physics* and *best throughput* cases.

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 15: 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 TBD

6 Conclusion

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

BRIEFLY

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