

Online Transverse Beam Instability Detection in the LHC

- High-Throughput Real-Time Parallel Data Analysis

Martin Söderén

Supervisor : Daniel Valuch
Examiner : Christoph Kessler
Academical Supervisor : Lu Li



Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

Abstract

This thesis presents the ADT transverse instability detection system, the next generation of instability detection in the LHC at CERN, Geneva. The system is presented after a thorough study of underlying causes for instabilities in high energy particle accelerators, current parallel programming paradigms, the available hardware and software at CERN and possible instability detection techniques. The requirements for the system involve handling vast amounts of data which need to be analyzed in real-time and in this data detect rapid amplitude growth while limiting the computational resources required to a minimum. The result of this thesis was a system that could generate a trigger when an instability was detected, which was used to save data from observation instruments around the LHC. A fixed display in the CERN control centre was also created which allows scientists and operators at CERN to monitor the oscillation amplitude of all particle bunches. The conclusion is that the complete system will be a valuable asset at CERN to help further develop the LHC.

Acknowledgments

"First and foremost, I would like to thank my supervisor at CERN, Dr. Daniel Valuch for his endless support, he always had time to sit down and take a coffee and explain anything unclear. My professor at Linköping University, Prof. Dr. Christoph Kessler together with my academical advisor Lu Li for their guidance and advice during this project. I would also like to thank all my colleagues in the BE-RF-FB group."

Martin Söderén
Geneva, October 2017

Contents

Abstract	iii
Acknowledgments	iv
Contents	v
List of Figures	viii
List of Listings	ix
List of Abbreviations	ix
List of Symbols	xi
1 Introduction	1
1.1 Motivation	1
1.2 Aim	2
1.3 Delimitations	3
1.4 Thesis Structure	3
2 Background	4
2.1 Definition of Real-Time in the Context of Particle Accelerators	4
2.2 Particle Accelerator Technology	4
2.2.1 The Purpose and Design of a High Energy Particle Accelerator	5
2.2.2 Dipole Magnets	6
2.2.3 Quadrupole Magnets	7
2.2.4 Chromaticity	9
2.2.5 Sextupole	9
2.2.6 Instabilities in the LHC	10
2.2.7 LHC Transverse Feedback System (ADT)	10
2.2.8 The ObsBox System	13
2.2.9 From a Gas Bottle to the LHC	16
2.3 Parallel Computing Technology	18
2.3.1 MIMD Architectures	19
2.3.2 Race Conditions and Synchronization Problems When Programming for a Parallel System	19
2.3.3 How Deadlocks Can Occur When Programming for a Parallel System . .	22
2.3.4 How Strangled Scaling and Lack of Locality Can Affect Performance . .	22
2.3.5 SIMD Instructions in x86-64 Processors	23
2.3.6 Industry Standards for Parallel Programming	24
2.3.7 The Parallel Pipeline Programming Pattern	25
2.3.8 Skeleton Programming	26
2.3.9 Tools for Analyzing Performance and Function	27

2.3.10	Advances in Compiler Technologies	27
3	Infrastructure at CERN	29
3.1	FESA	29
3.2	LHC Instability Trigger Network (LIST)	31
3.3	The LHC Logging System	32
4	Instantaneous Amplitude Calculation	33
4.1	Fast 16 bit Signed Integer to Single Precision Floating Point Conversion Using Intel Intrinsics	33
4.2	Transverse Oscillation Amplitude Calculation Using the Hilbert Transform . .	34
4.2.1	Optimizing the Hilbert Transformer for the LHC	35
5	Methodology	37
5.1	Pre-Study	37
5.2	Implementation	37
5.3	Evaluation	38
5.4	Presentation of Results	38
6	Implementation and Architecture	39
6.1	Architecture of the ADT Instability Detection System	39
6.1.1	Verifying That FESA Can Handle the High Bandwidth Data Streams . .	39
6.1.2	Proposed System Design	41
6.1.3	Potential Limitations in ObsBoxBuffer's Capacity	42
6.1.4	Proposed Structure for Exploiting the Algorithm Level Parallelism . . .	42
6.1.5	Exploiting Data-Level Parallelism	45
6.1.6	Proposed Algorithm to Detect Instabilities	45
6.2	Implementation of the ADT Instability Detection System	46
6.2.1	Retrieve the Data and Triggering a Real-Time Event	46
6.2.2	Serializing the Data and Converting It from Signed Integer to Single-Precision Floating-Point in the Real-Time Action	46
6.2.3	Injection Oscillation Triggering Prevention	47
6.2.4	Notch Filter	47
6.2.5	The Hilbert Transform Stage	47
6.2.6	The Amplitude Stage	47
6.2.7	The Maximum Stage	48
6.2.8	Moving Average / Instability Detection Stage	48
6.2.9	Transverse Activity Monitor Stage	48
7	Results and Discussion	49
7.1	Performance Evaluation	49
7.1.1	Optimizing the Pipeline	49
7.1.2	Performance Comparison Between Different Compilers	51
7.2	Functional Results	54
7.2.1	Testing the Algorithm in an Offline Environment	54
7.2.2	Automated Tuning of the Moving Average Threshold	56
7.2.3	Setting up the System	57
7.2.4	Setting the Thresholds	58
7.2.5	Tools To help Analyze the Collected Data	59
7.2.6	Results From Online Testing	60
7.2.7	Usage of the real-time Transverse Activity Monitor in the CCC	62
7.2.8	Real Life Example on How the System Helped Scientists at CERN . . .	63
7.2.9	An Example of How the System Detects Instabilities	65
7.3	Method Discussion	66

8	Related Work	68
8.1	The LHC Head-Tail Monitor	68
8.2	The LHC Base-Band Tune System (BBQ)	69
8.3	The Multiband-Instability-Monitor (MIM)	70
8.4	Algorithms for Instability Detection	70
8.4.1	Moving-Average	71
8.4.2	Three-Averages Algorithm	72
8.4.3	Increase-Subsequence Algorithm	72
8.4.4	Exponential Curve Fitting Using the Least Square Method	73
9	Conclusion	75
9.1	Relevance	75
9.2	Future Work	76
	Bibliography	77
	Appendices	80
A	Real Time Action	81
B	Constructing the Pipeline	88
C	Serializing Data and Pushing It to the Pipeline	90
D	Injection Oscillation Triggering Prevention Stage	93
E	Notch Filter Stage	95
F	Hilbert Transform Stage	97
G	Amplitude Calculation Stage	100
H	Maximum Stage	102
I	Instability Detection Stage	104
J	Transverse Activity Monitor Stage	110
K	Hilbert Filter Analysis in Matlab	112

List of Figures

1.1	The CCC	2
2.1	Overview of the CERN complex (courtesy of CERN)	5
2.2	The magnetic field in bending magnet in the LHC (courtesy of CERN)	6
2.3	Model of two focusing magnets (courtesy of CERN)	6
2.4	Frenet-Serret coordinate system[49]	7
2.5	Phase space of a particle in the accelerator [49]	8
2.6	Head-tail oscillation with mode=1 (courtesy of CERN)	10
2.7	Bunch injection with transverse damper off	11
2.8	Bunch injection with the transverse damper on	11
2.9	ADT overview (courtesy of the BE-RF-FB section at CERN)	12
2.10	The first bunch ever to circulate in the LHC	13
2.11	How the data acquisition capabilities of the ADT has increased	13
2.12	Installation in SR4 with two ADTObsBoxes	14
2.13	System overview	15
2.14	Part of Linac2	17
2.15	The four superimposed synchrotrons in the PSB (courtesy of CERN)	17
2.16	The author in the LHC tunnel inspecting the pickup connections for the ADT system	18
2.17	Principle of SIMD instructions	24
2.18	Example of output from Intel VTune	27
3.1	FESA workflow (Courtesy of CERN)	30
3.2	FESA navigator	31
4.1	Comparison of amplitude ripple between two Hilbert filters	35
4.2	Frequency response of two different filters	35
4.3	Comparison of the vector reconstructions using the two different filters	35
6.1	Diagram of data transfer times from ObsBoxBuffer	41
6.2	Block diagram of the system design	41
6.3	Block diagram of the pipeline design	42
6.4	Block diagram of instability detection part of the pipeline	46
7.1	Throughput of each stage in the pipeline and the complete pipeline	50
7.2	Activity in each stage when the pipeline is not saturated	51
7.3	Activity in each stage when the pipeline is saturated	51
7.4	Comparison of throughput using GCC and ICC	53
7.5	Comparison between different compilers	54
7.6	Test environment	55
7.7	Unstable beam	55
7.8	Stable beam with orbit drift	56
7.9	How the ADTBufferSaver fits in the system	58
7.10	Glitch in the data stream for HB2	59

7.11	Beam perturbed by injection	60
7.12	Excitation for coupling measurement during fill 6200 using the ADT as exciter . .	61
7.13	Amplitude growth during fill 6200	61
7.14	Short excitation for tune measurement during fill 6221 using the ADT as exciter . .	61
7.15	Low frequency orbit drift during fill 6221	61
7.16	The amplitude of bunch 731 during fill 6266 with slow rise time	62
7.17	Rapid amplitude growth during fill 6227	62
7.18	Entry in the LHC operator logbook	62
7.19	Instability shown in the ADT transverse activity monitor	63
7.20	Multiple bunches were unstable and this was visible thanks to the fixed display . .	63
7.21	Screenshot of the ADT transverse Activity Monitor	64
7.22	Some unstable bunches	64
7.23	Some unstable bunches	64
7.24	Some unstable bunches	64
7.25	Bunches 699, 732, 2949 and 3356 were deemed unstable by the system at 19:29:21 .	64
7.26	Part of the slide from the LHC morning meeting 4th October	65
7.27	Raw positional data for bunch 735 during the squeeze of fill 6266 in the LHC . . .	65
7.28	Data after notch filter	66
7.29	Instantaneous oscillation amplitude calculated using the Hilbert transform	66
7.30	Moving average of the instantaneous amplitude	66
8.1	Overview of the head-tail monitor system (courtesy of CERN)	69
8.2	A mode 4 instability captured by the head-tail monitor (courtesy of CERN)	69
8.3	Overview of the LHC BBQ system (courtesy of CERN)	70
8.4	Transverse position of an unstable bunch in the LHC	71
8.5	Moving average over windows with $W=1024$	71
8.6	Instability detection pipeline using exponential curve fitting	74

List of Listings

2.1	Example of a race condition	20
2.2	Solving synchronization with a mutex	21
2.3	Solving synchronization with a condition variable	21
2.4	Deadlock example	22
2.5	Spatial locality example	23
2.6	Example of Intel AVX intrinsics	23
2.7	OpenMP example	25
2.8	SkePU example	26
4.1	Fast 16 bit signed to float conversion	33
4.2	Hilbert transform using Intel Intrinsics	36
4.3	Instantaneous amplitude calculation using Intel intrinsics	36
6.1	Multi-threaded event producer	40
6.2	Real-time event	40
6.3	QueueElement used in pipeline	42
6.4	BlockingQueue used in pipeline	43
6.5	Status structure which is used to communicate with the pipeline	44
7.1	Driver for testing each stage	50
7.2	ICC Maximum assembly	53
7.3	GCC 4.4.7 Maximum stage assembly	53
7.4	GCC 5.4.0 Maximum stage assembly	54
7.5	Filename convention	59
7.6	Plotting all unstable bunches from all planes during a specific time	63
8.1	Exponential Curve Fitting	73
A.1	The real-time action which is triggered from a subscription to ObsBoxBuffer . .	81
B.1	This is the constructor of the pipeline which shows how the stages is created . .	88
C.1	Float conversion	90
D.1	Injection trigger prevention	93
E.1	Notch filter	95
F.1	Hilbert transform	97
G.1	Amplitude	100
H.1	Maximum amplitude	102
I.1	Instability detection	104
J.1	ADT activity monitor	110
K.1	Matlab filter analysis	112

List of Abbreviations

ADT	LHC Transverse Feedback System
API	Application Programming Interface
AVX	Advanced Vector Extensions
BBQ	Diode-Based Base-Band-Tune
BPM	Beam Position Module
BSRT	Transverse synchrotron light monitors
CAS	Compare And Swap
CCC	CERN Control Centre
CERN	European Organization for Nuclear Research
CMW	Controls MiddleWare
DMA	Direct Memory Access
DSP	Digital Signal Processing
FESA	Front-End Software Architecture
FIR	Finite Impulse Response
FIR	Finite Impulse Response
GPGPU	General-PURpose Graphics Processing Unit
GPU	Graphical Processing Unit
IDE	Integrated Development Environment
JAPC	JAVA API for Parameter Control
LHC	Large Hadron Collider
LIST	LHC Instability Trigger Distribution
MIM	Multiband Instability Monitor
MIMD	Multiple Instruction-streams Multiple Data-streams
MISD	Multiple Instruction-streams Single Data-stream
PRAM	Parallel Random Access Memory
PS	Proton Synchrotron
PSB	Proton Synchrotron Booster
PTP	Precision Time Protocol
SIMD	Single Instruction-stream Multiple Data-streams
SISD	Single Instruction-stream Single Data-stream
SNR	Signal Noise Ratio
SPEC	Simple PCIe FMC Carrier
SPS	Super Proton Synchrotron
SSE	Streaming SIMD Extensions
UMA	Uniform Memory Access
VME	Versa Module Europa
mDSPU	Digital Signal Processing Unit
mmap	Memory MAPped I/O

List of Symbols

$\alpha(s), \gamma(s)$	Twiss parameters	-
$A[n]$	Instantaneous amplitude	-
$\mathbf{B}$	Vector of magnetic field	T
B	Magnitude of magnetic field	T
β	Betatron function/twiss parameter	-
c	The speed of light	ms^{-1}
$\mathbf{E}$	Vector of electric field	Vm^{-1}
e	Elementary charge	C
ϵ	Geometric emittance	m^2
f_{rev}	revolution frequency	Hz
$\mathbf{F}_c$	Vector of centrifugal force	N
$\mathbf{F}_L$	Vector of Lorentz force	N
F_L	magnitude of Lorentz force	N
γ	Relativistic factor	-
$\mathcal{H}$	Hilbert transform	-
$K_u(s)$	Focusing effect	-
m_0	Relativistic mass of the proton	eV
MA_m	Moving average window	-
p	relativistic momentum	$kg \cdot ms^{-1}$
$\phi[n]$	Instantaneous frequency	-
q	Charge	C
Q_u	Betatron tune	-
ρ	Bending radius	m
σ	Standard deviation	-
$u(s)$	Solution to Hills's equation	-
$\mathbf{v}$	Velocity vector	ms^{-1}
v	Velocity scalar	ms^{-1}
W	Window length	-
$\hat{x}, \hat{y}, \hat{s}$	Basis in the Frenet-Serret coordinate system	-
$x_c[t]$	Analytic function signal	-
$x_i[t]$	Hilbert transform of a real one dimensional discrete signal	-
$x_r[t]$	Real discrete signal	-



1 Introduction

This chapter provides an introduction to the thesis project. It contains a motivation for the project in Sec. 1.1 followed by the aims and delimitations which are considered. For an overview of the report see Sec. 1.4.

1.1 Motivation

Particle accelerators have been in development for the last hundred years and are being operated for a wide variety of applications. High energy accelerators are used to explore the structure of matter. There are also several kinds of medical particle accelerators whose usage ranges from treating tumors to radioisotopes production. They are also found in industries where they are being used for geology, ion implantation in integrated circuits, lithography or sterilization. To further develop particle accelerators, knowledge and expertise from many fields are required and it is not uncommon that the latest techniques from these fields are applied. These fields include mathematics, physics, electronics, computer science, cryogenics, vacuum technology, material design, mechanical engineering or civil engineering to name a few. The particle accelerators of today are among the biggest and most complex machines in the world [59].

Computer technology is a crucial part of any modern high-energy particle accelerator. Partly because they generate vast amounts of data that need to be analyzed properly within a reasonable time. The latest technology in computer science is also required for the control and diagnostic systems of the machine. The beam, which is circulating close to the speed of light and contains extensive amounts of energy, needs to be precisely controlled. High energy particle physics is still a field with much to explore and every aspect of a high-intensity beam is still not fully understood. To widen the understanding of the dynamics of high-intensity beams multiple parameters need to be extracted using the accelerator diagnostics systems and compared with the available beam dynamic models. The main challenge is to do online analysis and extract valuable beam parameters since vast amounts of data are generated.

One important limiting factor for creating accelerators with higher energies are transverse beam instabilities that can occur sporadically. These can be very hard to detect in time and a lot of work is put into creating systems for reliable transverse instability detection. These

instabilities are not an uncommon phenomenon in the largest accelerator in the world, the LHC (*Large Hadron Collider*) located at CERN (*European Organization for Nuclear Research*), Switzerland. The first particle beam in the LHC was circulating in 2008 and one year later it set the world record in achieving a center of mass beam energy at 2.36 TeV. Today the beam is circulating most of the year at 6.5 TeV. This mass beam energy is planned to be increased in the future. There are theories that transverse instabilities will become more frequent when the energy is increased and in order to prevent this, a better understanding of the background for these instabilities is required.

1.2 Aim

This thesis aims to develop and put in production a system to detect transverse instabilities in real-time in the LHC. The purpose of this system is to alert other devices so a snapshot of the LHC can be stored for later analysis. This means:

- Design an algorithm that can detect instabilities using the available hardware and software
- Implement the algorithm efficiently so it can handle the immense throughput required with low latency
- Create an environment for testing the system for verification of its function
- Interface the system with already existing CERN infrastructure
- Verify functionality with real beams in the LHC
- Study if the current software and hardware can be used for further high-performance computing options and more sophisticated data analysis

When this system is deployed accelerator physicists at CERN will be able to access machine parameters that were stored after an instability was detected. This will help to get a better understanding of instabilities in the LHC and help understand how these can be avoided in future experiments. There will also be a fixed display in the CCC (*CERN Control Centre*) where the operators can monitor the transverse activity in the LHC.



Figure 1.1: The LHC island in the CCC during the restart after the long shutdown 2016/2017 (courtesy of CERN)

1.3 Delimitations

As mentioned in Sec. 1.1 instability detection in the LHC is important for gathering data to compare models with real data. This thesis only covers the case of high-throughput instability detection in the LHC using CERN's infrastructure and the limitations which it puts on the project regarding available software, hardware, and practices. The results can be applied to instability detection in any time series but there can also be other solutions more promising when implemented outside the CERN infrastructure.

There are difficulties in assessing the quality of a possible instability detection system since one does not necessarily know how many instabilities went unnoticed. However positive triggers can easily be assessed since a trigger will result in beam parameters being saved to a long time storage and from this data it is easy to verify the function of the system.

1.4 Thesis Structure

Chapter 1 gives a introduction to the whole thesis regarding motivation, aim and delimitations. Chapter 2 gives a theoretical background to the two major areas in this report, high-energy particle colliders in Sec. 2.2 and high-performance parallel computing in Sec. 2.3. Ch. 3 gives more information regarding the infrastructure at CERN and how it will be helpful when implementing the new transverse instability detection system. Ch. 4 describes how to calculate the instantaneous amplitude from a discrete time series. This is followed by Ch. 8 which describes related work, most of which had been done at CERN. Ch. 5 covers the methodology used in the implementation and how it was evaluated. Finally, the discussion and results are presented in Ch. 7 and the conclusion in Ch. 9.



2 Background

This chapter starts by defining real-time in the high-energy particle accelerator domain and then continues to give a theoretical basis for the two major concepts in this thesis, high-energy particle accelerators, and high-performance parallel computing. Section 2.2 gives a theoretical basis on high-energy particle accelerators. It only covers the concepts very briefly since going in depths on the function of particle accelerators is out of this thesis's scope. Sec. 2.3 covers parallel computer architectures and different ways of parallelizing software and the difficulties which come with it.

2.1 Definition of Real-Time in the Context of Particle Accelerators

The phrase “high-throughput real-time parallel data analysis” have different meanings in different scientific domains. The domain for this thesis is high energy particle accelerators. In this domain, the term “real-time” simply means that the analysis must be performed fast enough to not saturate the available computing resources. The experiments in the LHC such as ATLAS and CMS generate too much data so it is unfeasible to store and analyze all of it. To overcome this, they have multiple stages of real-time analysis with finer granularity in each stage to filter out irrelevant events. The result is a manageable amount of data which can be shared with scientist all over the world. In this project, however, the data can't be filtered or decimated so the system must have a high throughput. To achieve this using the hardware available the system must do the analysis in parallel.[20]

2.2 Particle Accelerator Technology

To perform fundamental particle research, high energy particle colliders are required to get the resolution required for studying the interactions between particles. These machines are extremely complex and expensive so only a few laboratories in the world exercise this research. In a collider such as the LHC, two beams are circulating in opposite directions and the collisions take place in several interaction points where the beams cross paths. When two particles from the two beams collide they result in subatomic particles which can be detected by the experiments in the interaction points. To achieve these collisions, many problems need to be overcome. The accelerated particles must originate from somewhere, they need

to stay in the accelerator, they need to be accelerated to the desired energy and they need to be packed together to increase the probability of collisions. To explain how all of this is done examples from the CERN complex will be used. An overview of the complex can be seen in Fig. 2.1.[27]

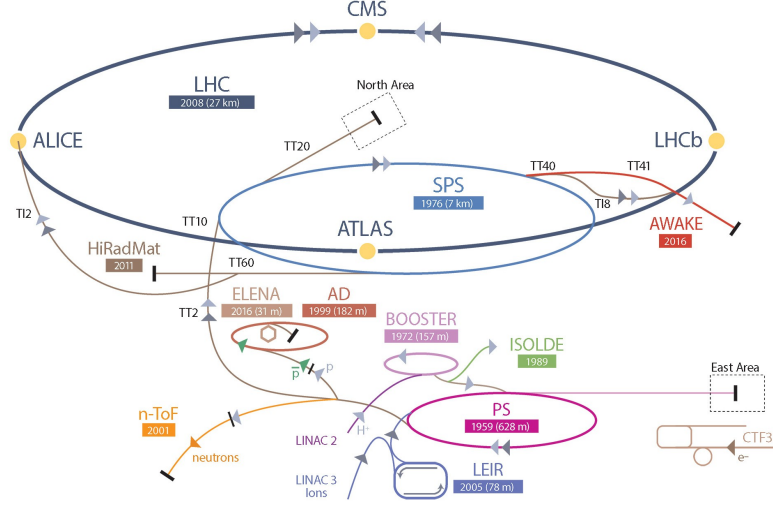


Figure 2.1: Overview of the CERN complex (courtesy of CERN)

2.2.1 The Purpose and Design of a High Energy Particle Accelerator

The purpose of a high energy particle accelerator can be summarized by a single equation[23]:

$$L = \frac{N_1 N_2 f_{rev} N_b}{4\pi\sigma_x\sigma_y} \quad (2.1)$$

Where $N_{1,2}$ represent the number of particles per bunch, f_{ref} the revolution frequency, N_b the number of bunches in the beams, and $\sigma_{x,y}$ the transverse and longitudinal beam size. L is the luminosity of the machine which describes the number of events registered by the detectors and the purpose of the LHC is to deliver a high-quality beam to all experiments. The energy of the particles in the beam is also an important parameter for the experiments since high energy is required to achieve the resolution required in the detectors but that is fixed at 6.5 TeV and is limited by the strength of the bending (dipole) magnets and the radius of the accelerators. The luminosity, however, can be improved by reducing the beam size.

To control the beam two physical effects are used, magnetic fields and electromagnetic fields. Electromagnetic fields are used to accelerate the beam using super conductive RF cavities which are designed so the electromagnetic waves become resonant and build up inside the cavity. When a charged particle travels through the cavity it experiences the field and is accelerated. Magnets of different orders (multipoles) are used to control the properties of the particles in the beam and the effect of the magnets on the particles can be described by a single equation (Lorentz force):

$$\mathbf{F}_L = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \quad (2.2)$$

Where $\mathbf{B}$ is the vector of the magnetic field, q is the charge of a particle, $\mathbf{E}$ is the vector of the electric field and $\mathbf{v}$ is the velocity of the particles. Different multipoles have a specific effect on the beam, the simplest magnets are the dipoles which are used to bend the beam.

2.2.2 Dipole Magnets

To bend the particles around the circular design orbit dipole magnets are used which generates a vertical magnetic field. In the LHC, the velocity v for all particles is close to the speed of light c and the charge of the particles is e , if it is only filled with protons, so Eq. 2.2 can be simplified to Eq. 2.3, if we ignore any potential electric field:

$$F_L \approx evB \quad (2.3)$$

The centrifugal force on a particle in its circular path is:

$$F_c = \frac{\gamma m_0 v^2}{\rho} \quad (2.4)$$

Putting Eq.2.3=Eq.2.4 with $v = c$ yields:

$$e \cdot B = \frac{\gamma m_0 c}{\rho} \quad (2.5)$$

$$p = \gamma m_0 c \quad (2.6)$$

Where Eq. 2.5 and Eq. 2.6 yields:

$$B = \frac{p}{e\rho} \quad (2.7)$$

With:

- $p = 7 \cdot 10^{12} / c [eV \cdot s \cdot m^{-1}] = 1.1215 \cdot 10^{-6} / c [J \cdot s \cdot m^{-1}]$ Particle momentum in LHC
- $\rho = \frac{15 \cdot 1232}{2\pi} [m]$ Since there are 1232 bending magnets in the LHC of length 15 m

$$B = 8.33 \text{ T} \quad (2.8)$$

This is exactly the maximum magnetic strength of the bending magnets in the LHC. In Fig. 2.2 the magnetic field lines of a bending magnet in the LHC is shown. The direction of the magnetic fields in the two beam pipes are opposite each other in the horizontal plane since one beam circulates clockwise and the other one circulates anticlockwise[45].

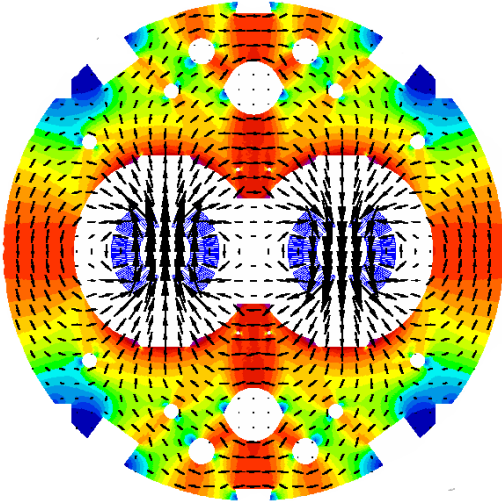


Figure 2.2: The magnetic field in bending magnet in the LHC (courtesy of CERN)

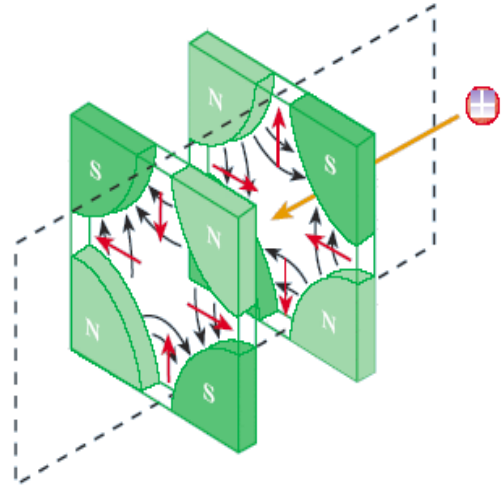


Figure 2.3: Model of two focusing magnets (courtesy of CERN)

2.2.3 Quadrupole Magnets

Because of the alternating electric fields in the RF cavities, the particles in the accelerators will be packed longitudinally into packages, called bunches. In one LHC bunch, the average number of particles is $\approx 1.15 \cdot 10^{11}$. All the particles in one bunch have the same charge so they will repel each other and over time the density of a bunch will decrease in all three dimensions. This will not only lower the probability of collisions in the interaction points but also lead to the beam hitting the machine aperture. To keep the particle density high, the particles need to be focused, just as with an optical lens. This is done using quadrupole magnets which have four poles, see Fig. 2.3. When a particle bunch experiences the magnetic field of a quadrupole magnet it is focused in one transverse plane and defocused in the other. This is why focusing magnets are always paired together where the second one is rotated 90° around the axis of beam travel.[45]

To describe the particle motion in the accelerator the Frenet-Serret coordinate system is normally used which describes the particle motion in reference to the design orbit in a right-handed orthogonal coordinate system moving along with the reference particle and with basis $\hat{x}$, $\hat{y}$ and $\hat{s}$.

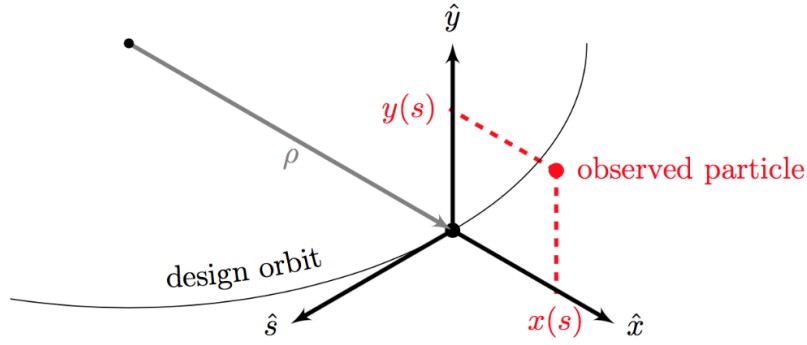


Figure 2.4: Frenet-Serret coordinate system[49]

The transverse motion of a single particle in the LHC can be described by Hill's equation:

$$u'' + K_u(s)u = 0 \quad (u = x, y) \quad (2.9)$$

Where K_u represents the focusing effect of the dipoles and the quadrupoles, according to:

$$K_x(s) = \frac{1}{\rho^2} - k(s) \quad (2.10)$$

$$K_y(s) = k(s) \quad (2.11)$$

where $k(s)$ is the normalized quadrupole strength and $\frac{1}{\rho^2}$ describes the geometric contribution to the weak focusing of the dipoles which only arises in the horizontal plane. The homogeneous solution to the Hill equation can be found with *Floquet's theorem* using an amplitude-modulation ansatz [50] and describes the transverse oscillation around the reference orbit, called the betatron oscillation:

$$u(s) = \sqrt{\beta(s)}\sqrt{\epsilon}\cos(\psi(s) + \phi) \quad (2.12)$$

$$u'(s) = \frac{du}{ds} = -\frac{\sqrt{\epsilon}}{\beta(s)}[\alpha(s)\cos(\psi(s) + \phi) + \sin(\psi(s) + \phi)] \quad (2.13)$$

The full derivation of this solution can be found in [58]. ϵ is the geometric emittance and is proportional to the area $A = \epsilon\pi$ of the phase space ellipse formed by the particle's potential states of motion at a specific point in the accelerator, as illustration in Fig. 2.5. The shape of the ellipse is defined by the Twiss parameters α , β and γ as follows:

$$\gamma(s)u'^2(s) + \alpha\alpha'(s)u'(s)u(s) + \beta(s)u^2(s) = \epsilon \quad (2.14)$$

where $\beta(s)$ is the amplitude function of the solution to the Hill's equation. It depends on the lattice (all quadrupoles) of the machine. The α parameter is defined as $\alpha(s) = -\frac{\beta'(s)}{2}$ and the γ parameter can be expressed as:

$$\gamma(s) = \frac{1 + \alpha^2(s)}{\beta(s)} \quad (2.15)$$

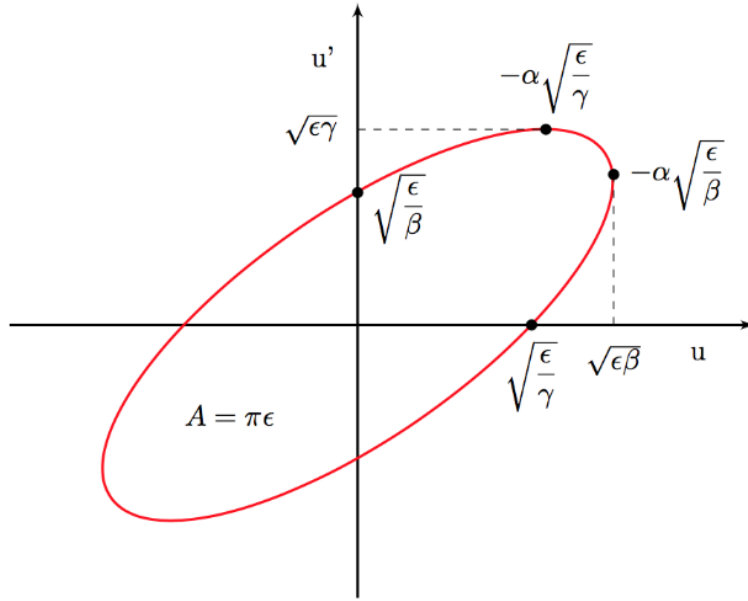


Figure 2.5: Phase space of a particle in the accelerator [49]

The betatron tune is defined as the total phase over a full circumference divided by 2π and represents the number of betatron oscillations done by the particle, per turns:

$$Q_u = \frac{1}{2\pi} \oint \frac{ds}{\beta_u(s)} \quad (u = x, y) \quad (2.16)$$

The tune of the transverse planes defines the working point of the machine and has to be chosen with caution. If there is an error in the optical system which periodically kicks the particle at a specific point in the machine it can induce optical resonance which could increase the amplitude of the betatron oscillation and lead to particle loss. Optical resonance can occur if the following condition is satisfied:

$$lQ = p, \quad (l, p \in \mathbb{Z}) \quad (2.17)$$

The lattice (all optical elements combined) of the machine can also result in coupled resonance between the transverse planes if the following condition is satisfied:

$$xQ_x + nQ_y = p, \quad (m, n, p \in \mathbb{Z}) \quad (2.18)$$

The order of resonance is $|m| + |n|$. As a rule of thumb, lower order of resonance normally have faster rise times so if the rise time of an instability can be measured, then the order of resonance can be deducted.

2.2.4 Chromaticity

From Eq. 2.2 it is obvious that the deflection of a charged particle in a magnetic field depends on the particle's energy. This effect causes a dispersion in the dipole magnets but also modifies the effective focusing strength of the quadrupoles which is inversely proportional to the momentum of the particle. The change of focusing strength due to energy deviation is:

$$\Delta k = 1 \frac{1}{p^2} \frac{dB_y}{dx} \Delta p = k\delta \quad (2.19)$$

With $\delta = \frac{\Delta p}{p}$, the relative momentum error. This quadrupole error results in a tune shift proportional to the energy offset:

$$\Delta Q = \frac{1}{a\pi} \int \beta(s) \Delta k(s) ds = \left[-\frac{1}{a\pi} \int \beta(s) k(s) ds \right] \delta \quad (2.20)$$

So if the tune shift can be measured, then the quadrupole error can be deduced. The derivation of the betatron tune with respect to the momentum deviation includes the effect of all quadrupoles in the lattice and is called the natural chromaticity of the machine:

$$Q' = \frac{dQ}{d\delta} = -\frac{1}{4\pi} \int \beta(s) k(s) ds \quad (2.21)$$

The natural chromaticity depends on the quadrupole magnets and the focusing strength of the machine and it is typically negative. $d\delta$ from Eq. 2.21 is normally in the order of 10^{-4} . Combined with the usually large chromaticity of a machine, the tune spread is in the order of 10^{-2} [50]. This large tune spread will inevitably force some particles to oscillate on some higher-order resonances, leading to amplitude growth and particle loss. To control this, the chromaticity must be adjusted and is normally set around zero or slightly positive which means that the particle momentum spread barely affects the particle tune spread. To achieve this, sextupoles with non-linear dispersions functions are used to sort the particles depending on momentum.

2.2.5 Sextupole

A sextupole is a higher order magnet with a larger focusing effect on particles that are displaced further from the axis compared to a quadrupole. The driving terms of a quadrupole can be expressed as:

$$f_x(s) = m(s) \cdot (x^2 - y^2) \quad (2.22)$$

$$f_y(s) = m(s) \cdot xy \quad (2.23)$$

with the sextupole strength:

$$m(s) = \frac{q}{p} \frac{\partial^2 B}{\partial^2 x} \quad (2.24)$$

By matching the sextupoles with the quadrupoles:

$$m(s) \cdot D(s) \approx k(s) \quad (2.25)$$

By adding the effect of the sextupoles to the natural chromaticity:

$$Q' = \frac{dQ}{d\delta} = -\frac{1}{4\pi} \int \beta(s) [k(s) + m(s)D(s)] ds \quad (2.26)$$

The chromaticity can be controlled together with the stability of the beam.

2.2.6 Instabilities in the LHC

Even though the tune and chromaticity are controlled, there are several other reasons for instabilities in a high-energy particle accelerator. For example beam-beam interaction, electron clouding, and wake fields [4]. It is important to measure beam properties before, during, and after an instability to understand the source and make corrections. Normal symptoms of instabilities are emittance growth and losses on specific bunches. This can happen at any time during operation and it is normally very unpredictable. The emittance growth can be seen in the BSRT (*Beam Synchrotron Radiation Telescope*) but by that time it is too late to react since there is a long latency in the BSRT operation. The BSRT is a camera that records the radiation from the beam pipe [5]. There is a big need to detect instabilities as they occur to measure relevant beam parameters for future corrections. By implementing the ADT instability detection system, the tune shift and the rise time of the instability can be measured and the possible causes can be limited.

Beam-beam interactions happen in one of the four interaction points. Both beams affect one another since they generate an electromagnetic field. A simple model of this is the strong-weak beam interaction model where one strong beam is considered unperturbed by the other beam and at the same time, it acts as a non-linear focal lens. This means that the weak beam is periodically perturbed by the strong beam which can cause an instability [10].

Electron Clouding occurs when charged particles disturb stray electrons in the particle accelerator which makes the electrons hit the beam pipe. This will result in more electrons being emitted because of secondary emissions. This will generate an electric field which could perturb the accelerated particles in the accelerator [2].

Wakefields are electromagnetic fields which are created when the charged particles in a particle accelerator interact with the vacuum chamber. The coupling can exist because of irregularities in the beam pipe material or geometric features in the beam pipe. This creates a coupled system of particles and electromagnetic fields that may become unstable. Since we have causality in the accelerator, the fields exist after the bunch that is coupled with the machine. These fields affect trailing bunches and can be seen as a dynamic magnetic field which can cause the trailing bunches to oscillate [38]. When bunches are affected by the wakefield there is a coupled bunch instability and most of these can be dampened by the LHC transverse feedback system (ADT).

Beam-beam interactions, electron clouding, and wakefields can cause head-tail resonance [3]. During a head-tail instability, the bunch oscillates internally with higher modes. For example, if the bunch is experiencing a mode 1 head-tail oscillation the head and tail are oscillating in a counter-phase, see Fig. 2.6.



Figure 2.6: Head-tail oscillation with mode=1 (courtesy of CERN)

2.2.7 LHC Transverse Feedback System (ADT)

When a bunch is injected into the LHC there is an injection error and a sudden change of direction when the particles are exposed to the bending magnetic field. This causes injection

oscillations and emittance growth. To dampen these oscillations, the LHC has the ADT. The ADT is, in theory, a simple proportional feedback system which measures the position for each bunch once per revolution and during the following turn applies an electrostatic kick proportional to the error from the design orbit. In Fig. 2.7 the injection oscillation is clear when compared to Fig. 2.8, where the damper is turned on. It also dampens potential coupled bunches instabilities.

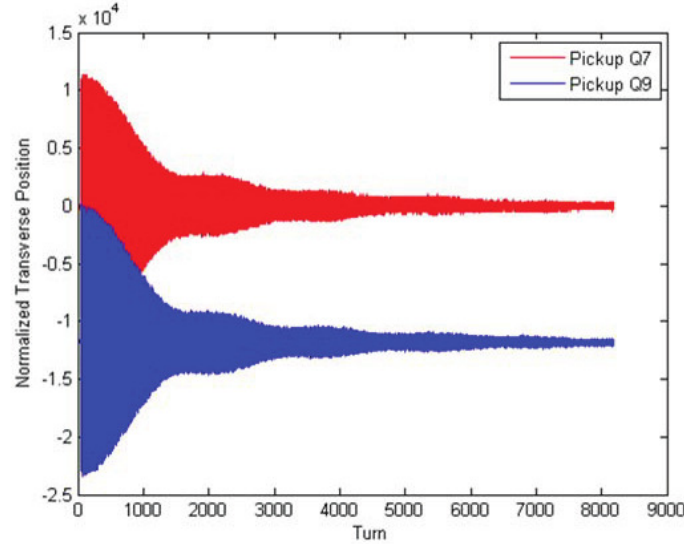


Figure 2.7: Bunch injection with transverse damper off. The transverse oscillation is damped naturally in thousands of turns (courtesy of the BE-RF-FB section at CERN)

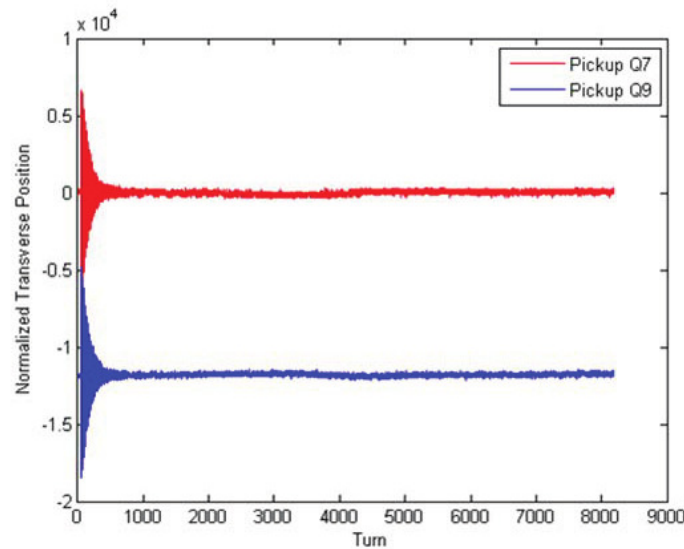


Figure 2.8: Bunch injection with the transverse damper on. The transverse oscillation is actively damped by ADT in hundred turns (courtesy of the BE-RF-FB section at CERN)

The main task of the ADT is to actively force the oscillatory part of the beam transverse position to zero. It serves several other needs such as beam exciter which is useful for extracting beam parameters. This system is also valuable because it is the only place in the LHC where full-rate bunch-by-bunch positional data with submicron resolution is available. This means

that every turn, each bunch position in the machine is measured and buffered for a couple of minutes. Using the FESA framework described later in Sec. 3.1 this data can be accessed by users all over CERN to analyze the dynamics of the particle beam.

There are four independent systems in the LHC, one per beam and the transverse plane. Each of these uses two dedicated pickups out of the four available that provide the transverse position of each bunch every turn. This data is digitally processed to calculate a correction drive signal to the power amplifiers which are feeding the electrostatic kickers [25].

Each pickup is an electromagnetic coupling device installed in the vacuum chamber (beam pipe) which is very similar to an RF directional coupler. Two stripline lines parallel to the chamber are installed on both sides of the chamber. The beam represents the center conductor. When the beam is centered, the induced voltage to both electrodes is the same. When displaced, more voltage is induced into the electrode closer to the beam. The voltages from both electrodes are fed into a hybrid filter which generates a difference and sum signal. These signals are fed to the surface through long coaxial cables where they are fed into a BPM (*Beam Position Module*) which calculates the transverse position of each bunch. This data is then fed over a fiber-optic link to a DSPU (*Digital Signal Processing Unit*) which calculates the output signal for the power amplifiers which drive the electrostatic kickers [25].

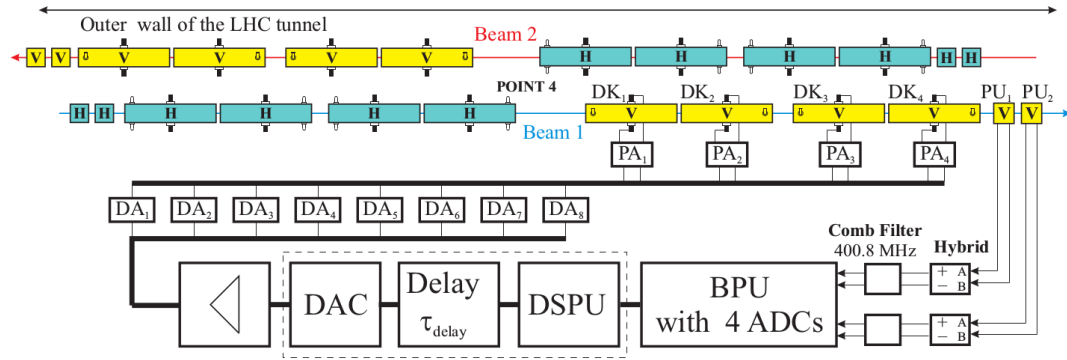


Figure 2.9: ADT overview (courtesy of the BE-RF-FB section at CERN)

The ADT system is the only place in the LHC where a full-rate bunch-by-bunch transverse position data is available for an unlimited length of time. This was the only system that was capable of observing the first particle bunch in the LHC when it was started in 2008 as can be seen in Fig. 2.10.

The position is calculated with sub- μm resolution. Earlier, this information was only available in the DSPUs where the data could not be extracted at full-rate because of the limitations in the VME bus [39]. Because of these limitations, only the data for 8 bunches and 32768 turns could be extracted. This was increased by adding external memory to the DSPUs which allowed the positional data for all bunches during 144 turns to be captured. This was then increased significantly by introducing the ObsBox system. The evolution of the data capturing capabilities of the ADT system can be seen in Fig. 2.11.

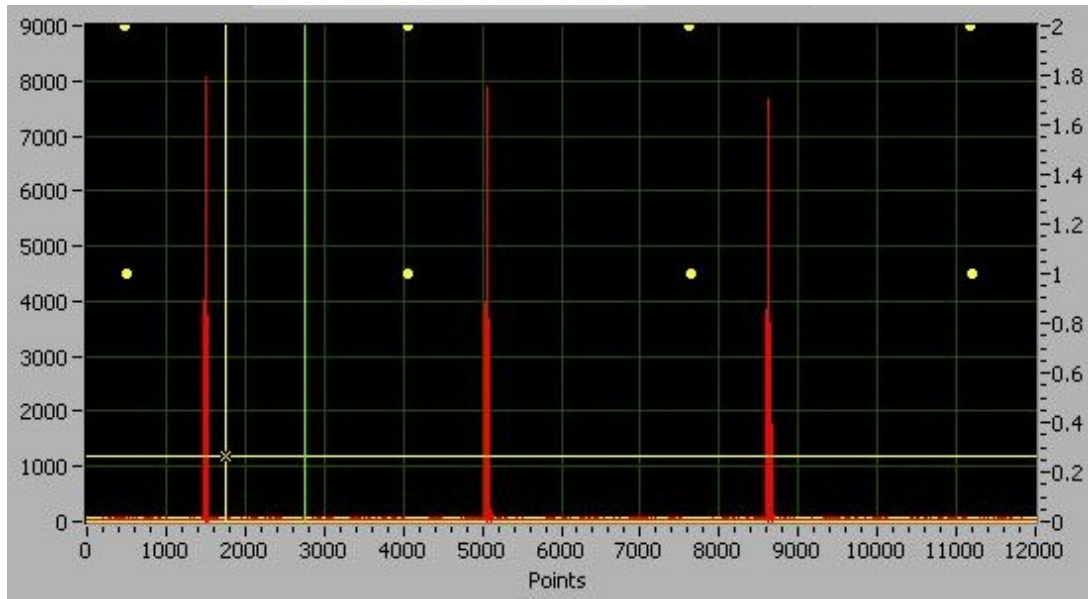


Figure 2.10: The first bunch ever to circulate in the LHC on the 10th September 2008 08:30:10 UTC captured by the ADT (courtesy of the BE-RF-FB section at CERN)

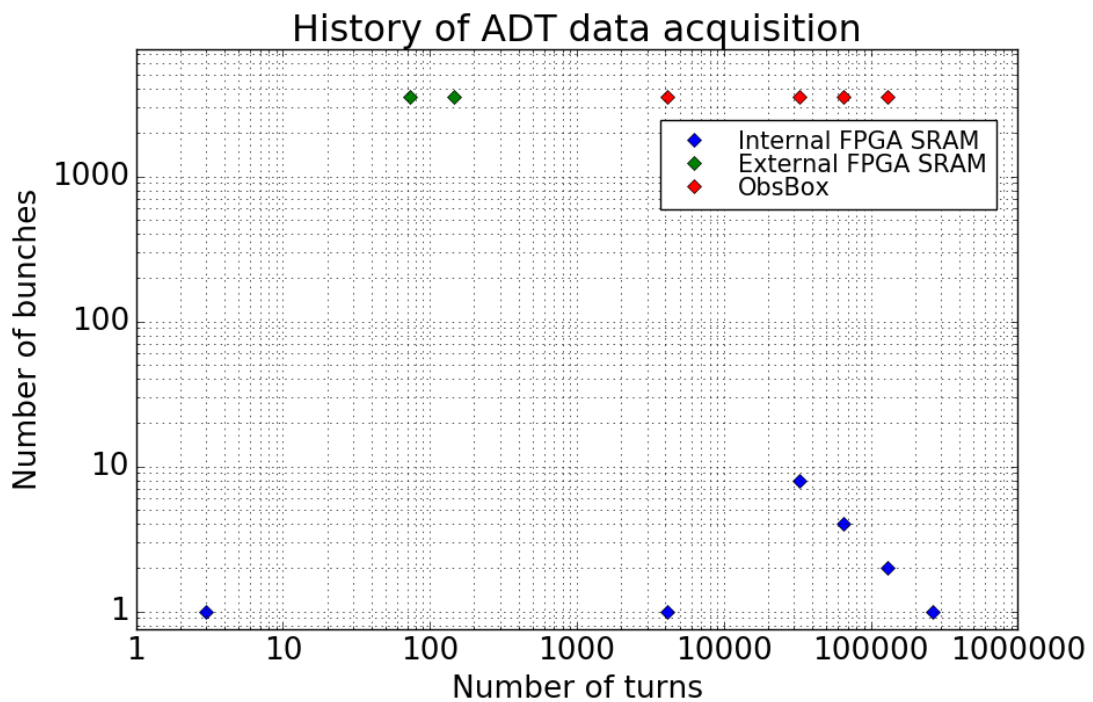


Figure 2.11: How the data acquisition capabilities of the ADT has increased

2.2.8 The ObsBox System

The so-called "ObsBox" system was designed to overcome the limitations of the VME bus and allow for an increase in the amount of data available for analysis without any prior filtering, decimation or disruption of the ADT function. The current hardware allows for six minutes of full-rate data buffering instead of milliseconds in the ADT. It is a very powerful computer system based on standard server hardware with custom PCI format Gb-link receivers. There

are four servers, one for each plane which receives data from each pickup by four fiber-optic links. Each channel transmits the bunch position as 16 bit signed integer, originating in the 16-bit fixed point digital signal processing in FPGAs. The revolution frequency of the LHC is 11245 Hz and the position is sampled once per turn. The maximum number of particle bunches in one LHC ring is 3564. This requires $6 \cdot 60 \cdot 11245 \cdot 3564 \cdot 4 \cdot 2 \text{ B} \approx 115 \text{ GB}$ of memory. Each fiber-optic channel needs its own PCIe card and there must also be a timing card which can receive timing events. So in total 5 PCIe slots are required and preferably one extra PCIe 16x for potential online analysis using a GPU. The SuperMicro 6028U-TR4+ was chosen which has 5 PCIe slots x8 slots and one x16 slot, all in a 2U rack-mounted format. It also supports two Intel Xeon E5-2600 v4/v3 processors and up to 1.5 TB of RAM. The ObsBox servers have dual Intel Xeon E5-2620 and 132 GB of RAM [39]. The cards which are used to receive the fiber-optic signal is made at CERN by the controls group and are called SPEC (*Simple PCIe FMC carrier*)[41]. The timing card is a simple five channel digital I/O card [40].

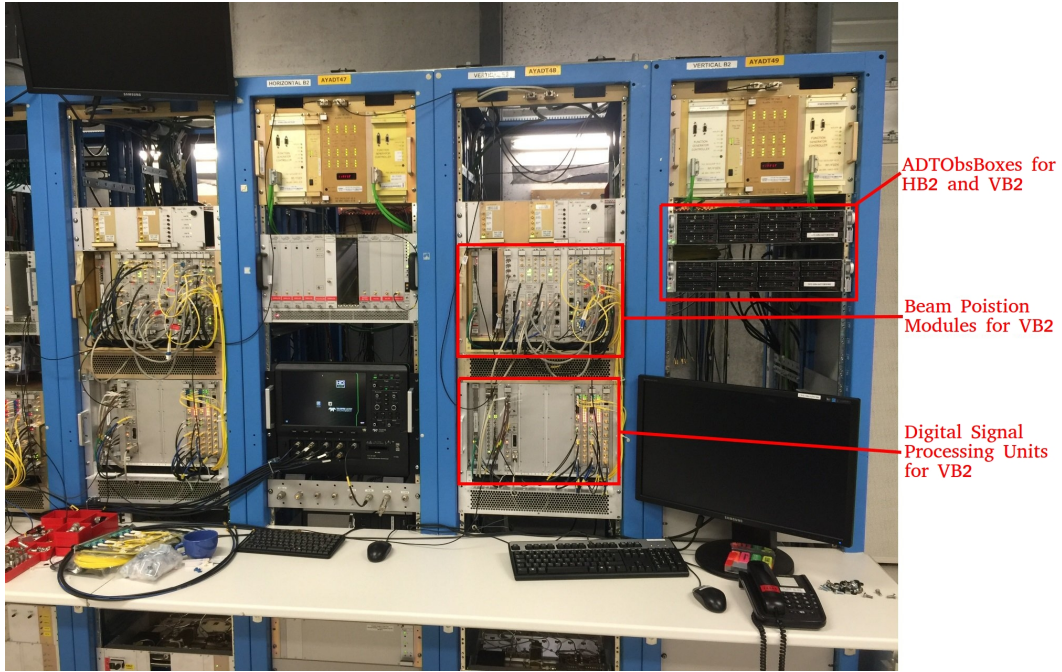


Figure 2.12: Installation in SR4 with two ADTObsBoxes in the rack to the far right (courtesy of the BE-RF-FB section at CERN)

These servers run Scientific Linux, which is being developed by Fermi National Accelerator Laboratory and is based on Red Hat Enterprise Linux, but has been patched with a real-time kernel [15]. The real-time patch guarantees that the latency between an interrupt and the process being called is usually below 10 microseconds. Fig. 2.13 shows an overview of the system.

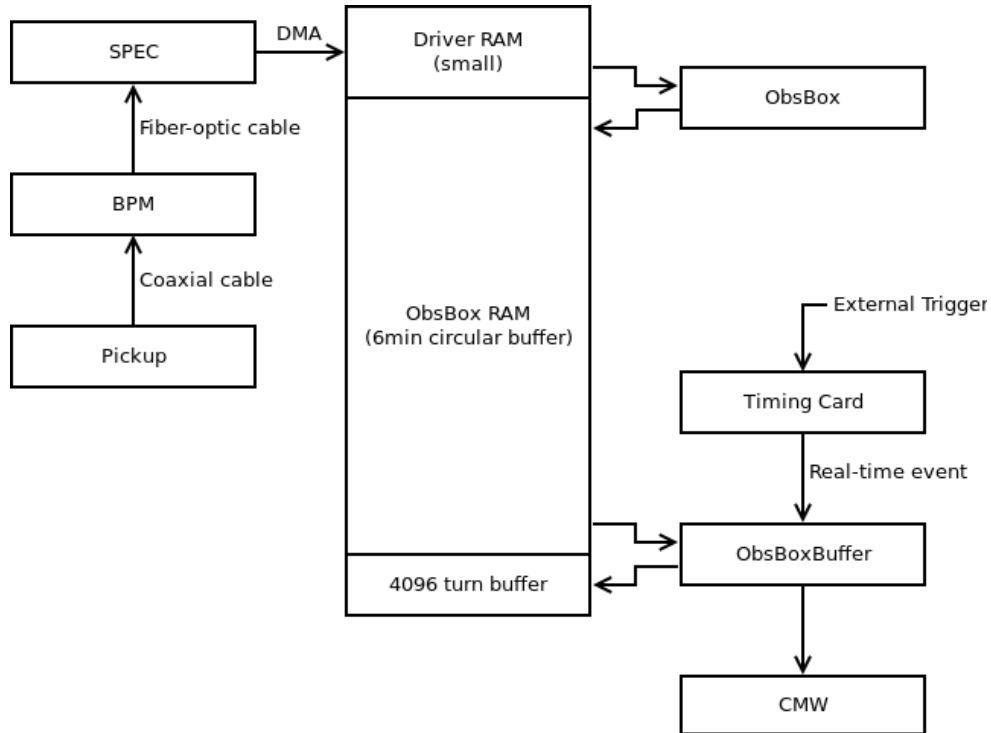


Figure 2.13: System overview

The SPEC card is a bridge between the fiber-optic channel and the RAM on the host computer. It has DMA (*Direct Memory Access*) which means that the CPU does not need to be involved in the data transfer. The driver for the SPEC is based on the ZIO framework which defines the input/output data flow and the user-space interface to access the data [42]. The framework makes it easy to set up buffers, triggers, and Linux DMA configurations. The data can be accessed through normal mmap (*Memory Mapped I/O*) which means that the data can be read just like a normal file. There was a need for an abstraction layer above this so the data could easily be acquired anywhere at CERN, with some extra requirements:

- The users must be able to request data at any time.
- The data must be frozen in different buffers of different lengths from external triggers.
- The data must be fetched from the driver memory often since the driver memory is small
- The application must use a standard protocol so it can be integrated into CERN infrastructure easily.

To fulfill all these requirements while keeping the software as simple as possible, the FESA (*Front-End software Architecture*) framework was used which is the standard framework for developing machine control software at CERN. See Sec. 3.1 for more information about FESA.

The software part of the ObsBox system consists of two FESA classes. There is the ObsBox class which communicates with the ZIO driver and handles the 6 minutes long main circular buffer. On top of this, there is the ObsBoxBuffer class which reads data from the circular buffer and transfers it to shorter buffers which can be acquired by users. The data collection task of the ObsBox class and the data distribution task of the ObsBoxBuffer class was split to provide freedom and flexibility. This allows for creating virtually unlimited numbers of user buffers to serve all kinds of purposes with different data fetching habits. There

are special buffers for injection observation, post-mortem analysis, and spectral analysis, to name a few. This data can be accessed in many ways:

- Through the FESA navigator, which is a simple generic Java-client application for testing FESA classes.
- Through JAPC (*JAVA API for Parameter Control*), which is a JAVA API used to interface with the middleware at CERN.
- Through PyJAPC, which is a Python interface to JAPC.
- Directly through CMW (*Control Middleware*), which is the control middleware at CERN.
- From any other FESA class that has an association relationship with ObsBoxBuffer.

There are many applications for the available data. It can be used for offline analysis where users are downloading the data for later analysis. It can be used for semi-offline analysis where data is being captured and downloaded on triggers and is being analyzed on the fly. It can also be used for an online analysis which is done on the server. Before this project, the only online analysis that was done was ADT performance analysis to monitor the function of the LHC feedback system [31] and it was only executed during injection and was not time critical.

For implementing an online instability detection system on the ADTObsBox servers, a solution would be to create a new FESA class which acquires the data from the shortest buffer available (4096 turns at the moment). These buffers can be automatically frozen periodically every 4096 turns so it would be a full-rate data stream with packages of 4096 turns corresponding to ≈ 350 ms worth of data. The new class would subscribe to this buffer and every time it receives a new package it can analyze the data to detect potential instabilities and notify appropriate devices in the LHC. However, the framework must be tested before the design decision is made, as it is not certain that it can handle the vast data transfers that are required.

2.2.9 From a Gas Bottle to the LHC

The protons which are circulating in the LHC start as hydrogen gas in a bottle in Linac 2 (*Linear accelerator 2*) which is a linear particle accelerator, there are also Linac 1, Linac 3 and Linac 4 at CERN. The gas is fed into Linac 2 where all electrons are stripped off so only the hydrogen nucleus remains, i.e. protons. It uses multiple radiofrequency fields to accelerate the particles to the desired energy of 50 MeV. This is equivalent to letting each particle be accelerated in an electric field with a potential difference of $5 \cdot 10^7$ V [8].

After Linac 2, they are transferred to the PSB (*Proton Synchrotron Booster*) which consists of four independent circular particle accelerators which accelerate the particles to 1.4 GeV, see Fig. 2.15. The PSB is a synchrotron just as PS (*Proton Synchrotron*), SPS (*Super Proton Synchrotron*) and LHC at CERN. It uses strong magnets to bend the particles around a circular design orbit using the Lorentz force and RF cavities to accelerate them. RF power generators supply an electromagnetic field in an RF cavity which is carefully designed so that the wave becomes resonant inside the cavity. Charged particles pass through the cavity and accelerate in the resulting field. The field in the cavity oscillates with a specific frequency which results in particles being packed into bunches [8].



Figure 2.14: Part of Linac2 which is the first step to having particles circulating in the LHC (courtesy of CERN)

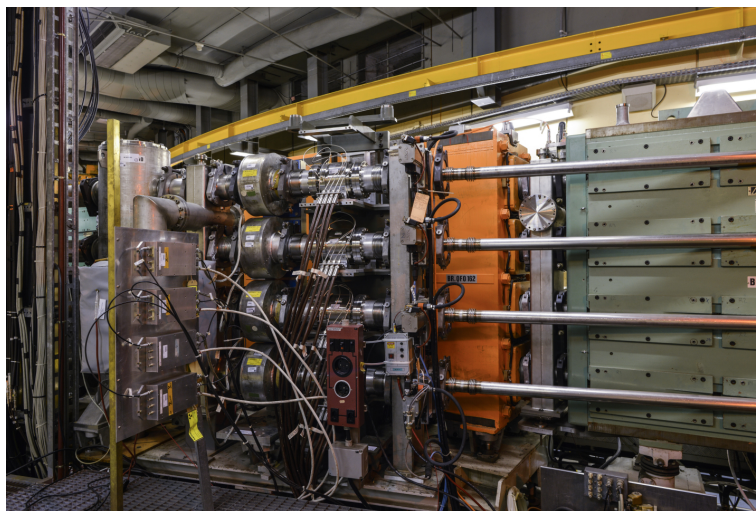


Figure 2.15: The four superimposed synchrotrons in the PSB (courtesy of CERN)

From the PSB, the particles are transferred to the PS which is one of CERN's oldest accelerators dating back to 1959. There, just as in the PSB, they are bent in a circular design orbit by bending magnets and accelerated by RF cavities resulting in an energy of 25 GeV. After the PS, the particles go through the same procedure in the SPS where they accelerated to 450 GeV, and then they are finally injected into the LHC where the beam is accelerated to 6.5 TeV. After the acceleration, the design orbit can be changed so the beams collide. A part of the 27 km long LHC tunnel can be seen in Fig. 2.16. After the LHC has been filled, the particles can circulate for many hours. Up to 12 hours of collisions is not uncommon but after a while, too many particles have collided and the number of particles left in the machine is sparse so the beam is dumped and then refilled [8].

At every interaction point, there is a different experiment which observes the collisions.

There is ATLAS, CMS, LHCb, and ALICE which are all designed to capture different events. ATLAS and CMS are general purpose detectors which were used for finding the Higgs boson among other things. ALICE is specialized in heavy-ion collisions and LHCb is specialized in analyzing properties of the bottom quark [7].



Figure 2.16: The author in the LHC tunnel inspecting the pickup connections for the ADT system

2.3 Parallel Computing Technology

Oh, a few programmers in love with the challenge have shown that most types of problems can be force-fit onto parallel computers, but general programmers, especially professional programmers who “have lives”, ignore parallel computers.

Timothy G. Mattson

Beverly A. Sanders

Berna L. Massingill

Parallel computers come in many forms and they surround us every day. There are multi-core computers, tablets, and cellphones which are used by most people for everyday use. They are still very complex and they contain a great deal of computing power which can go unused unless the programmers of the operating system and of the applications have knowledge about how to fully utilize the hardware. There are also more advanced, specialized parallel computers such as clusters that are composed of many computers which work together on one problem. Multiple clusters can be connected to form a computing grid such as the Worldwide LHC Computing Grid which is used to analyze the data from all experiments [9]. There are massively multi-core systems such as modern GPUs (Graphical Processing Unit) providing over a thousand cores and massively parallel specialized on-chip solutions. All of these can normally be classified into four categories as in Flynn’s taxonomy[17]:

- **SISD**: Single Instruction-stream, single data-stream
- **SIMD**: Single Instruction-stream, multiple data-streams
- **MISD**: Multiple Instruction-streams, single data-stream
- **MIMD**: Multiple Instruction-streams, Multiple data-streams

The most interesting categories are MIMD since this contains the normal multi-core super-scalar processor which is being used in most computers, and SIMD because most modern CPUs have special SIMD instructions which are being used to speed up, for example, vector operations. The SISD category contains the sequential computer which utilizes no parallelism. It fetches one instruction at a time from memory and executes it. This is where computers started once upon a time. The MISD category does not have any real implementations.

2.3.1 MIMD Architectures

In the MIMD architecture, multiple processor units execute instructions asynchronously, independently and concurrently. Depending on hardware and software support a MIMD architecture can either be used to execute multiple tasks concurrently, execute one task in parallel or a combination of them. Many tasks can be executed simultaneously because most normal CPUs are multitasking, meaning that they swap the context rapidly so all processes have access to the processor unit. Today there is almost an abundance of computing power and the biggest bottleneck is communication and synchronization.[26]

There are two different types of MIMD architectures and the difference is the memory layout. There are MIMD architectures where the data is distributed over multiple processing nodes. This architecture is called NUMA (*Non-Uniform Memory Access*). Nodes normally exchange data using some kind of message passing interface such as OpenMPI [43]. These nodes can consist of multiple cores which use a shared memory architecture. In the shared memory architecture, all processing units share the same memory address space but the physical memory may or may not be shared. If the memory is shared between processes, the system has a UMA architecture (*uniform memory access*). UMA is great for performance since multiple processing units can access the same memory at the same time. A problem with this is cache coherency, meaning that data which is in multiple processing units cache memory must be kept consistent and there are many different algorithms for this [19].

For actually implementing a shared memory program the norm is to use threads. A thread can be viewed as a sequence of instructions that can be managed by the operating system scheduler. In the case of Linux, which is the most used operating system kernel, each thread is its own process and multiple threads can communicate through inter-process communication methods. When a program creates a new thread, a child process is created which shares the text memory, stack, heap etc. The new process can start executing instructions that are different from the ones which the parent process is executing but they can access the same memory. This is useful since they can split the workload and execute their instructions on different processing units and fully utilize the available computing power. This is useful in for example many areas of computing such as dense linear algebra, spectral methods, unstructured grid analysis (finite element analysis), Monte Carlo methods, physics simulations, computer games to name a few. A parallel program can be implemented using several different methods, see Sec. 2.3.6 for more details on different methods [19].

2.3.2 Race Conditions and Synchronization Problems When Programming for a Parallel System

Section 2.3.1 describes the UMA architecture where all processing units have access to the same memory. This is great for performance but this can create non-deterministic programs if data is transformed in the wrong order. To solve this, access to the data must be synchronized in some way and race conditions must be avoided. Race conditions occur when two different concurrent threads access the same data at the same time which can cause unde-

defined behavior. See Listing 2.1 where two threads have access to a global array and assume the following scenario:

1. The first thread reads one value from the array, adds 1 to it and then it is preempted by the scheduler.
2. The second thread reads the same value from the array, adds 2 to it, writes it back to the array and then gets preempted.
3. The first thread continues and writes over the value from the second thread with 1.

Listing 2.1: Example of a race condition

```
int data[10]={0,0,0,0,0,0,0,0,0,0};
void threadFunc(int x){
    for(int i=0;i<10;i++){
        int temp=data[i];
        temp+=x;
        data[i]=temp;
    }
}
std::thread first (threadFunc,1);
std::thread second (threadFunc,2);
first.join();
second.join();
```

This is a simple example that illustrates a serious problem. There are many techniques for serializing data access such as:

- Barriers
- Semaphores
- Mutex
- Critical sections with conditional variables
- Atomics

To guarantee mutual exclusion (only one thread can enter a critical section) a mutex can be used [1]. With this the problem in Listing 2.1 can be solved, see Listing 2.2.

Barriers can be implemented as function calls which make a thread sleep until all spawned threads have called the function. Semaphores are signaling mechanisms which can allow one or more threads to access a critical section and it has a counter associated with it. When a thread enters the section the counter is decremented atomically and if a thread tries to access the section while the counter is zero it has to wait. A mutex is a lock which allows one single thread to access a critical section. When it enters the section, it locks the mutex and when it leaves it unlocks it. A thread that tries to lock it while another thread is in the critical section sleeps until the lock is unlocked. See Listing 2.2 on how this can solve the synchronization problem [36].

Listing 2.2: Solving synchronization with a mutex

```

int data[10]={0,0,0,0,0,0,0,0,0,0};
std::mutex mtx;
void threadFunc(int x){
    mtx.lock();
    for(int i=0;i<10;i++){
        int temp=data[i];
        temp+=x;
        data[i]=temp;
    }
    mtx.unlock();
}
std::thread first (threadFunc,1);
std::thread second (threadFunc,2);
first.join();
second.join();

```

Condition variables is a way of letting threads sleep until a criterion is met. For example, you have a consumer thread which waits for data to consume, this thread can wait for a condition variable and when another thread has created data for it to consume, it can notify that thread using the condition variable which wakes it up. An example is given in Listing 2.3. This is useful for many applications, for example if you have a pipeline pattern where data is passed through the pipeline then this can be used for notifying the next stage that data is available [57].

Listing 2.3: Solving synchronization with a condition variable

```

std::mutex mtx;
std::condition_variable cv;
std::string data;
bool ready=false;
bool processed=false;
void threadFunc(){
    std::unique_lock<std::mutex> lk(mtx);
    cv.wait(lk, []{return ready;});
    data += "world";
    processed = true;
    lk.unlock();
    cv.notify_one();
}
//Start thread which will wait until data is ready
std::thread first (threadFunc);
//Prepare data
data="Hello_"
ready = true;
//Notify thread
cv.notify_one();
//Wait for thread to finish
std::unique_lock<std::mutex> lk(mtx);
cv.wait(lk, []{return processed;});
first.join();

```

Choosing what way to synchronize data accesses completely depends on the application.

The programmer must have knowledge about how the data is accessed and choose the appropriate method.

2.3.3 How Deadlocks Can Occur When Programming for a Parallel System

A natural problem that every programmer that deals with parallel programs must take considerate care of is deadlocks. Deadlocks occur when two processing units are waiting for one another to release a resource. See Listing 2.4 and assume the following scenario:

1. The first thread calls transfer and locks down A's mutex
2. The second thread calls transfer and locks B's mutex
3. The first thread tries to lock B's mutex but it is locked by the second thread, it waits for the second thread to unlock it
4. The second thread tries to lock A's mutex but it is locked by the first thread, it waits for the first thread to unlock it.
5. They are deadlocked forever

Listing 2.4: Deadlock example

```
class Account {
    double balance;
    std::mutex mtx;
    void withdraw(double amount) {
        balance -= amount;
    }
    void deposit(double amount) {
        balance += amount;
    }
    void transfer(Account from, Account to, double amount) {
        from.mtx.lock();
        to.mtx.lock();
        from.withdraw(amount);
        to.deposit(amount);
        to.mtx.unlock();
        from.mtx.unlock();
    }
}

Account A;
Account B;
std::thread first= std::thread(transfer,A,B);
std::thread second= std::thread(transfer,B,A);
```

There are multiple ways of avoiding this behavior. For example, by avoiding locks as much as possible, never holding more than one lock at a time and acquiring multiple mutexes in the same order. But mutexes are useful and sometimes the best way of synchronizing access to critical sections [36].

2.3.4 How Strangled Scaling and Lack of Locality Can Affect Performance

Strangled scaling is a result of serializing access to data through some kind of synchronization mechanism. A part of the memory that is protected must have its state sent between threads which adds overhead. This shows when coarse-grained locking is being used (meaning that

a large part of the memory is protected by one lock). For example, if multiple threads are accessing a large matrix which is protected by a lock then most of the runtime threads will be waiting to get the lock. To prevent this, fine-grained locking can be done, for example, one lock per column/row of the matrix [36].

The locality has to do with memory access and there are two important concepts, temporal locality, and spatial locality. Temporal locality is when a processing unit will probably soon access the same memory location again. Spatial locality is when the processing unit will probably access memory location nearby. Both spatial and temporal locality is improved if the data is in the cache memory of the processing unit. Reading data from memory is time-consuming and when the CPU reads data from the memory it does not read in only that memory but it reads a complete cache line (the amount of data that fits in one line in the cache of the processing unit). If a transform is being executed element-wise on an array of variables sequentially, then there is a good chance that the next element in the array is already in the cache. See Listing 2.5 for a good and bad example of spatial locality. The best way to access memory is sequentially [36].

Listing 2.5: Spatial locality example

```
float data[256][256];
void bad_spatial_locality_init() {
    for(int i =0;i<256;i++){
        for(int j =0;j<256;j++){
            data[j][i]=1.0;
        }
    }
}
void good_spatial_locality_init() {
    for(int i =0;i<256;i++){
        for(int j =0;j<256;j++){
            data[i][j]=1.0;
        }
    }
}
```

2.3.5 SIMD Instructions in x86-64 Processors

SIMD instructions are multiprocessing elements which can be used to exploit data level parallelism. For example, if a vector of length 8 is added element-wise to another vector of same length. Without SIMD this takes eight scalar add operations while if there are two SIMD registers available of length 8, this can be done in one operation, see Fig. 2.17. SIMD instructions in x86-64 processors have been around since the first Pentium processors with the MMX registers which were 64-bit wide and allowed for 8, 16, 32 or 64-bit integer operations. This was later extended with SSE (*Streaming SIMD Extensions*). The first generation SSE introduced 8 128-bit long registers which allowed operations on 4 32-bit single precision floats in one cycle. In programs where the same operations are applied to many elements such as digital signal processing or graphics processing, the number of operations could be increased by 400% compared to scalar operations. SSE2 introduced SIMD operations for double precision floating points. There have been further upgrades in SSE3, SSSE3, SSE4 and SSE4.1 which all introduced new features. But the biggest change came with AVX (*Advanced Vector Extensions*). AVX was introduced in 2011 in the Intel Sandy Bridge platform and it extends the SIMD registers to 256-bit. This allows for simultaneous operations on 4 double precision floating point values or 8 single precision floating point values which can be executed using up to 32 registers, see Listing 2.6 on how it can be used [16].

Listing 2.6: Example of Intel AVX intrinsics

```
//Allocate 32byte aligned memory
float* data = (float*) _mm_malloc(8 * sizeof(float), 32);
//Init memory
for(int i =0;i<10;i++){
    *(data+i)=static_cast<float>(i);
}
//Load it into a register
__m256 data_vec = _mm256_load_ps(data);
//Add the vector to itself
data = _mm256_add_ps(data, data);
//Free the data
_mm_free(data);
```

The AVX registers can be accessed either through inline assembly, high-level intrinsics or by compiler auto-vectorization. There are problems with all of these. Both inline assembly and high-level intrinsics are error prone and not very readable compared with normal high-level languages such as C or C++. The maintainability of the code decreases since the numbers of programmers that can handle this is sparse. But at the same time, this gives full control over the program and one does not have to rely on the compiler. In high-performance applications and libraries, this might be suitable. This does sacrifice portability since some assembler instructions or intrinsics might not be available on some processors.

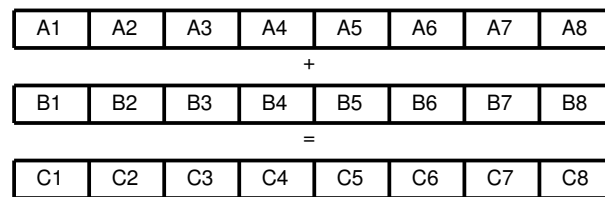


Figure 2.17: Principle of SIMD instructions

Auto-vectorization in compilers can be enabled by compiling with optimization enabled depending on compiler and version. In most cases, auto-vectorization only applies to the embarrassingly parallel problems such as for-loops without any dependencies and cannot replace an experienced programmer.

2.3.6 Industry Standards for Parallel Programming

There is no single standard for parallel programming today but many approaches on how to achieve concurrent execution. Different designs suit different platforms and application so the programmer has to choose the one that suits the specific problem the best.

Pthreads (POSIX Threads) is an execution model that allows a program multiple flows of execution. Each flow is referred to as a thread and control over this is achieved by using the POSIX Threads API. It is available on both UNIX-like systems and Windows so it is cross-platform. The API also gives access to synchronization tools between threads such as mutexes, condition variables, and barriers. See Sec. 2.3.2 for more information in synchronization [37].

OpenMP (*Open Multi-Processing*) is an open standard for shared memory parallel programming. It is supported in many compilers such as GCC and ICC and it consists of compiler **#pragma** directives. It is a very efficient way of generating concurrent code from sequential code where parts of the program can run in parallel. Listing 2.7 shows a simple example

where the loop will be executed in parallel by up to 10 threads. It is possible to define how many threads will be used, how the load will be shared and also conditional concurrency meaning that the loop will only run in parallel if the number of iterations is bigger than a certain value [36].

Listing 2.7: OpenMP example

```
int data[10]={0,0,0,0,0,0,0,0,0,0};
#pragma omp parallel for
for(int i=0;i<10;i++){
    data[i]++;
}
```

MPI (*Message Passing Interface*) is a standard for passing messages between processes. It is a versatile interface which can be used to pass messages in a distributed memory environment such as a cluster or just on a local machine. It is a standard with many implementations just as Pthreads but the most common implementation is OpenMPI [43].

CUDA (*Compute Unified Device Architecture*) is a framework for GPU/GPU (*General Purpose Graphical Processing Unit*) computing. It specifically targets NVIDIA GPUs and allows them to be used as accelerators for high-performance applications [19].

OpenCL (*Open Computing Language*) is a cross-platform framework for heterogeneous computing. It targets multiple platforms and can run on a normal x86 CPUs, GPUs, Cell processors and FPGAs to name a few [36].

TBB (*Threading Building Blocks*) is a C++ template library for building parallel programs on multi-core processors. It breaks down the program into tasks which can be executed in parallel. It works by generating a dependency graph of all tasks which are then executed and synchronized according to the dependency graph [36].

2.3.7 The Parallel Pipeline Programming Pattern

For online algorithms where the input is coming from real-time sources such as keyboards, pointing devices or any general I/O a pipeline is a good way of overlapping computation and I/O. This means that the computation can be executed on part of the available data while more data is being acquired. This is also natural in DSP (*Digital Signal Processing*) where data is being sampled continuously and flows through a signal pipeline where it can be downsampled, mixed and filters can be applied concurrently. The same practice can, of course, be applied in signal processing on MIMD architectures. There are already multiple data pipelines in a normal computer. There is an instruction pipeline in most modern CPU architectures which executes multiple instructions at the same time and there are SIMD registers which can be used for loop-level pipelining. There is a graphical pipeline in every graphics card which executes the tessellation, vertex processing, geometry processing etc. Then there is algorithm-level pipelining where the programmer formulates an algorithm like a pipeline [36].

A pipeline is a linear sequence of stages where data flows through the pipeline from the first stage to the last one. The data is partitioned into pieces that are called items and each stage performs a transform on the item. This is favorable for soft real-time and online applications since early items can flow through the pipeline before later items are available. The composition of a pipeline is also very straightforward and it maps very well to the serial I/O. Pipelines can also be used to limit the resources a program is allowed to use, meaning that it can process a vast amount of data using very little memory. It also makes it easy to avoid potential deadlocks, see Sec. 2.3.3, and each stage in the pipeline can be

analyzed, improved, and debugged separately. However, the throughput of the pipeline is still limited by the slowest stage which can be found by analyzing every stage separately [36].

To process each item, a worker is needed. A worker can, for example, be a thread. There are two different ways of implementing a pipeline, either the worker can be tied to a stage or a worker can be tied to an item and follow it through the pipeline. So either the item flows past the stages or the stages flow past the item. The major difference is the locality of reference, see Sec. 2.3.4. If the worker is tied to the data and the worker is tied to a processing unit, this could result in great locality assuming that the item can fit in the cache. However, if the data is passed around workers which are tied to different processing units the data will be passed around different cache memories which will result in cache misses, this, of course, depends on the cache size, size of the item, and how the data is accessed in the item. But a general rule could be big stage-small item should be bound to the stage and small stage-big item should be bound to the item. The TBB library which was mentioned in Sec. 2.3.6 has support for linear pipelines in its `TBB::parallel_pipeline` template and also support for more complex pipelines in `TBB::flow`.

2.3.8 Skeleton Programming

A major problem with generic programming for parallel systems is that the underlying hardware differs in many ways and even slight differences can affect performance immensely. There is no silver bullet for creating a parallel program that fully utilizes the hardware independent of what parallel platform it is executed on. Skeleton programming is a template method design pattern which enforces a structure which is efficiently parallelizable by the system [12]. It draws inspiration from functional programming languages where higher-order functions are accepting other functions as arguments which then transform the data. Typical higher-order functions are `map`, `scan`, `stencil`, and `reduction` which are quite common in functional programming languages such as Haskell. A `map` function applies a function to every element of a set and each application of the function is independent so the transform can be executed in parallel for each element in the set.

Skeleton programming is an added level of abstraction to get a more useful set of resources in exchange for a loss of performance by adding overhead compared with a low-level implementation. But the new resources will also result in portable performance and scalability. This means that an algorithm implemented today using skeleton programming can run on many parallel platforms, such as multi-core CPUs and GPUs, can be used for a long time since the skeleton programming framework takes care of the parallelization. By enforcing the restrictions to use a higher-order function with gaps for its own implementations and leaving the low-level optimization for the framework, portable performance can be achieved.

There are many research projects which are exploring the possibilities of skeleton programming such as Skell BE[48], SkelCL[52] and SkePU 2[14]. Listing 2.8 shows a simple example of how a dot product could be calculated using the SkePU framework.

Listing 2.8: SkePU example

```
BINARY_FUNC(mul , float , a, b,
    return a*b;
)
BINARY_FUNC(add , float , a, b,
    return a+b;
)
int main()
{
```

```

skepu::MapReduce <mul , add > dotProduct(new mul , new add );
skepu::Vector <float > v0(20, 2.0f);
skepu::Vector <float > v1(20, 5.0f);
float r = dotProduct(v0 , v1);
}

```

2.3.9 Tools for Analyzing Performance and Function

There are many tools available for analyzing the performance of software. There are profilers which are used to find potential bottlenecks in the code and general debuggers to find bugs which result in the software not generating correct results. The most commonly used debugger for Linux is GDB[22] which supports everything that is expected from a debugger and more. Breakpoints in the code can be defined to the program halts when they are reached, memory can be modified when the execution is halted and the call stack can be inspected. GDB is used purely from the terminal but if a graphical interface is required on top of that then DDD[21] can be used which is also a GNU project. To find memory leaks, Valgrind[56] is an excellent tool, it keeps track of the memory allocated by the program and the pointers to that memory. Any memory that does not have a pointer to it is defined lost.

To analyze the software after it produces the expected result, any of the available profilers might be used. They are used for dynamic program analysis that measures memory usage, the usage of certain instructions, frequency, and duration of function calls to name a few. When profiling multi-threaded software, it is valuable to analyze how much time is spent waiting for synchronization primitives, the load balancing between threads/cores and any potential bottlenecks in the code. For example Callgrind[55] is a free tool which is a part of Valgrind. Callgrind generates a call-graph which displays where most of the execution time is spent. Another great tool for profiling especial parallel programs is Intel VTune[28] which supports automatic hotspot analysis (bottlenecks), concurrency analysis and locks analysis. An example of the output which VTune can generate can be seen in Fig. 2.18.

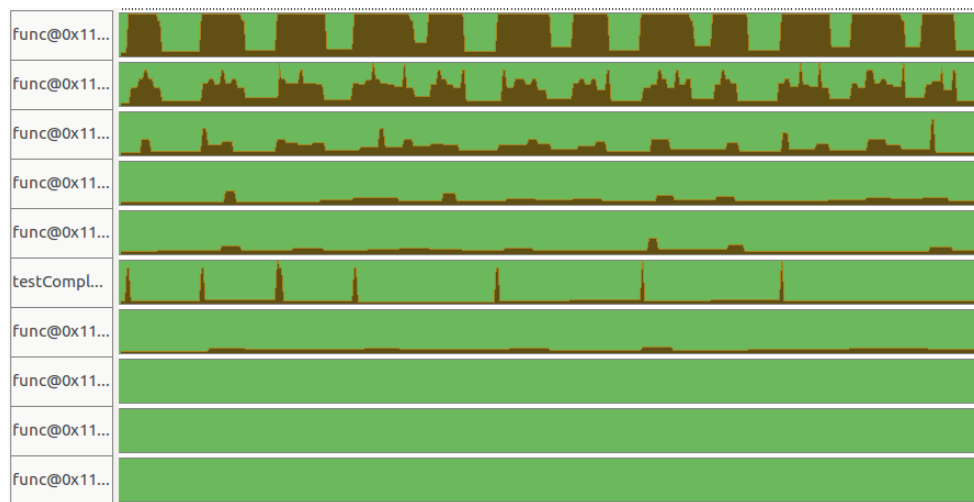


Figure 2.18: Example of output from Intel VTune

2.3.10 Advances in Compiler Technologies

A while ago, the processors started to hit the power wall, meaning that they could not increase the performance by increasing the clock frequency. To improve the performance of processors, ILP (*Instruction-Level Parallelism*) started to be explored. ILP is a collection of

techniques used to reduce run time of instructions. This can be done both in hardware and software. On the hardware level, there can be instruction pipelines where the execution of multiple instructions can overlap, or out-of-order execution where instructions are ordered in a more efficient manner while keeping data dependencies. Reordering can also be done by the compiler. Another technique which is used by the compiler is auto-vectorization where multiple scalar operations can be exchanged for fewer vector operations. To further improve performance, multi-core processors were introduced which could execute multiple streams of instructions concurrently. Unfortunately, they were still limited by the bandwidth of the memory bus, which means that the data could not reach the processing units as fast as they are executing them. To solve this, bigger cache memories were introduced which were very efficient if the instructions and text memory had spatial and temporal locality.

It is not an easy task for the compiler to find the vectorizable code, and verifying that the compiler is doing a good job is a daunting task. Research has shown that modern compilers can at most vectorize 45-71% of perfectly vectorizable loops and 18-30% of loops in normal applications [34]. To fully utilize the vector capabilities of modern processors the compilers must not only vectorize the loops, but they must change the memory layout, do code replacement and align the data. To do this they must do accurate interprocedural pointer disambiguation and interprocedural array dependencies analysis which can be extremely complicated [34]. There is no standard for auto-vectorization so different compilers behave differently and some compilers need more guidance than others to generate efficient code.

3

Infrastructure at CERN

To be able to support all physics experiments CERN needs an immense technical and computing infrastructure. The CERN data centre has 110000 processor cores divided over 10000 servers which process a petabyte of data every day. This results in 30 petabytes per year which must be stored permanently. It has state-of-the-art network equipment with over 35000km of optical fiber. The whole complex has Wi-Fi that is being used by over 4000 users simultaneously. But the infrastructure is not only hardware but also software to help the engineers to control the equipment [6]. There are three important parts of the accelerator control infrastructure for this project. The FESA framework allows for a fast standardized release of control equipment software. The LIST network allows for distributing instability triggers across the LHC. The LHC logging system is used to log parameters from all devices used in the LHC.

3.1 FESA

The FESA (*Front-End Software Architecture*) framework is a project launched in 2003 at CERN by the controls group [24]. It is a complete environment for equipment specialists to design, develop, deploy and test equipment software at CERN. This tool helps to standardize, speed-up and simplify the task of developing control software. The FESA infrastructure contains many interconnected components such as:

- Object-oriented Real-Time Framework: Defines the architecture of the software and lets the programmer focus on the functionality of the control software.
- Graphical Tools: Graphical applications for generating design, deployment, and instantiation of a new FESA class such as a plugin for the Eclipse IDE [13].
- FESA Design Schema: The complete meta-model of the FESA class. This forces the designer to construct the new class out of several predefined objects:
 - A public interface for controlling the class
 - A device-model which is an abstraction of the underlying hardware
 - A set of server actions which are triggered from the public interface
 - A set of real-time actions which are triggered by logical events

- A set of logical events which can be triggered by many kinds of sources
- Code generation: From the design, deployment, and instantiation XML documents efficient C++ code is generated.
- Test environment: From the design XML document a JAVA GUI is automatically generated for interfacing with the new class during testing, see Fig. 3.2.

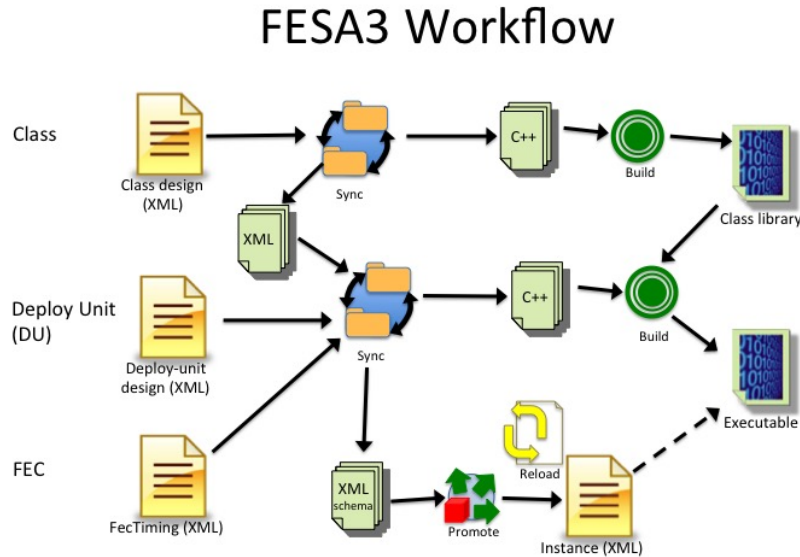


Figure 3.1: FESA workflow (Courtesy of CERN)

The usage of FESA across all CERN's accelerators has led to a standardized high-level language which is used for developing portable, maintainable, and efficient control equipment software. If the developers have used configuration fields in an appropriate manner then deploying new instances of a FESA class on new FECs (*Front-End Computer*) can be done directly in the CCDB (*Controls Configuration DataBase*) and it can be operational in a short timespan [24].

Here is an example of how one would implement a simple FESA class using the Eclipse IDE (*Integrated Development Environment*) which only reads a temperature from some control equipment:

1. In Eclipse one creates a new FESA class project which automatically has version control using SVN.
2. Create a data field.
3. Create an acquisition interface.
4. Add a field to it named Temp and link it to the data field.
5. Define a get-action and link it to the interface.
6. Generate code, this will generate code for the get-action in which the underlying hardware library can be called to acquire the temperature and set the data field.

7. Create a new FESA deploy unit project in the Eclipse IDE.
8. Add the class project to the deploy unit and add an FEC to it which it can run on and also define a class instance (a FEC can run multiple instances of the same FESA class).
9. Automatically deploy the class and the deploy unit from the Eclipse plugin.
10. Update the startup sequence for the FEC where it should run so it starts the new class.
11. Reboot the FEC.

This is a very simple example and FESA supports many features such as real-time actions which can be triggered by an internal timer or an external timing event.

The FESA framework is a possible platform for developing the new LHC instability detection system since it is well-known and integrated into the CERN computing infrastructure. It has full support for real-time computing in the Scientific Linux distribution which is used in all FECs at CERN. One limitation, for now, is that it only supports GCC 4.4.7 which means it has limited C++11 standard support. It must also be verified that the FESA framework can handle the transfer of the gigabit streams of data which will be required by the LHC instability detection system.

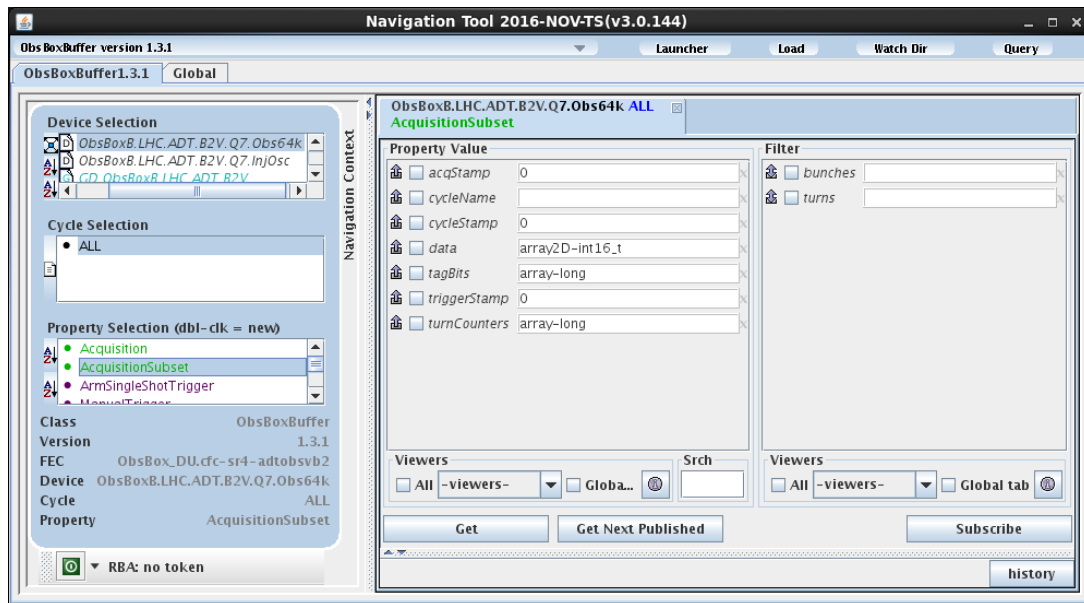


Figure 3.2: FESA navigator

3.2 LHC Instability Trigger Network (LIST)

The purpose of the LIST is to receive a trigger from a “cloud” of devices and distribute it to all relevant devices in a very deterministic manner, with low and known latency. It is powered by White Rabbit which is a multi-laboratory and multi-company collaboration tool for the development of Ethernet-based technology. It allows for sub-nanosecond synchronization and deterministic data transfer by extending the PTP (*Precision Time Protocol*). The main feature of a White Rabbit network is that the output pulse will be generated at the exact same time with sub-nanosecond scatter among all nodes distributed over a large distance covering over 10 km [60].

The LIST allows for bi-directional trigger distribution between equipment in the LHC that is

capable of detecting instabilities and devices containing relevant information to analyze the cause of the instability. If any device in the LHC ring detects an instability it can send out a trigger with a payload and any device connected to the network can take precaution, such as freezing their observation buffer and sending it for long time storage for later analysis [60].

3.3 The LHC Logging System

The LHC logging system is a collection of tools and frameworks which allows logging of the state of all equipment hardware. It logs data from all the hardware in the LHC. The result of all this is that anyone can go back in time and check the configuration of every device around the LHC at a specific time. This system will be used by the instability detection system to log when a bunch in the LHC became unstable. There are multiples ways of extracting the stored data but a simple way is to use TIMBER which is a generic Java application for extracting data which also has Python bindings [46].



4 Instantaneous Amplitude Calculation

The data that is received from the ObsBoxBuffer FESA class contains the transverse position for each bunch each turn as 16 bit signed integers. To simplify the detection of a potential instability, it makes sense to calculate the instantaneous amplitude for each bunch. This chapter describes first how a vector of 16 bit signed integers can be converted to a vector of single precision floating points very fast using Intel intrinsics in Sec. 4.1 and how the instantaneous amplitude can be calculated using the Hilbert transform in Sec. 4.2.

4.1 Fast 16 bit Signed Integer to Single Precision Floating Point Conversion Using Intel Intrinsics

The first step to calculate the instantaneous amplitude from the positional data is to convert the data to single precision floating points. There is no instruction for direct 16 bit signed integer to single precision floating-point conversion so there must be an intermediate step to convert it to 32 bit signed integer. This can be done very efficiently using AVX registers and Intel intrinsics in the following manner:

Listing 4.1: Fast 16 bit signed integer to single precision floating point conversion using Intel Intrinsics

```
//load 8 short int
__m128i a0 = _mm_loadu_si128(data);
//split into two registers
__m128i b0 = _mm_unpackhi_epi64(a0, a0);
//convert to 32 bit integers
a0 = _mm_cvtepi16_epi32(a0);
b0 = _mm_cvtepi16_epi32(b0);
//convert to 32 bit float
__m128 c0 = _mm_cvtepi32_ps(a0);
__m128 d0 = _mm_cvtepi32_ps(b0);
```

4.2 Transverse Oscillation Amplitude Calculation Using the Hilbert Transform

After the single precision floating-point conversion the instantaneous oscillation amplitude for each bunch can be calculated. To calculate the instantaneous amplitude, the Hilbert transform can be used [33][47][44]. The Hilbert transform can be implemented as a FIR (*Finite Impulse Response*) filter and can be tuned for the appropriate frequency band. Using the Hilbert transform, the analytic signal for the real signal is calculated from which the instantaneous amplitude can be calculated.

A real signal $x_r[n]$ is a one-dimensional array of real values over time. This signal normally has positive and negative frequency components with a symmetry around the zero-frequency point. This signal can be extended by a companion function $x_i[t]$ so that the resulting signal only has positive frequency components. The relation between $x_r[n]$ and $x_i[t]$ is related through the Hilbert transform [33][47].

$$x_c[n] = x_r[n] + jx_i[n] \quad (4.1)$$

Where $x_c[t]$ also can be expressed as

$$x_c[n] = A[n]e^{j\phi[n]} \quad (4.2)$$

where

$$A[n] = \sqrt{x_r^2[n] + x_i^2[n]} \quad (4.3)$$

The Hilbert relationship is

$$\mathcal{H}[x_r[n]] = x_i[n] \quad (4.4)$$

which can also be expressed as

$$x_i[n] = \sum_{m=-\infty}^{\infty} h[n-m]x_r[m] \quad (4.5)$$

where:

$$h[n] = \begin{cases} \frac{2}{\pi} \frac{\sin^2(\pi n/2)}{n}, & \text{if } n \neq 0 \\ 0, & n = 0 \end{cases} \quad (4.6)$$

However, Eq. 4.5 is not really helpful since it is not absolutely summable and also non-causal. That is when a discrete Hilbert transformer of order M can be created with a Kaiser window approximation which, instead of Eq. 4.6, uses:

$$h[n] = \begin{cases} \frac{2}{\pi} \frac{\sin^2[\pi(n-n_d)/2]}{n-n_d}, & \text{if } 0 \leq n \leq M \\ 0, & \text{otherwise} \end{cases} \quad (4.7)$$

where:

$$n_d = M/2 \quad (4.8)$$

So for example with $M = 6$:

$$h = [-0.2122 \quad 0 \quad -0.6366 \quad 0 \quad 0.6366 \quad 0 \quad 0.2122] \quad (4.9)$$

4.2.1 Optimizing the Hilbert Transformer for the LHC

Since the frequency of the transverse oscillation in the LHC is known [50] there is a potential for optimization. The FIR filter can be tuned so the magnitude response for the tune band can be flat. By using the Matlab Filter Designer [35] and creating a Hilbert FIR filter of order 6 tuned to the normalized band $[0.27 \ 0.32]$ the following coefficients were acquired:

$$h = [-0.0906 \quad -0.0198 \quad -0.5941 \quad 0 \quad 0.5941 \quad 0.0198 \quad 0.0906] \quad (4.10)$$

The two filters were compared by feeding both with a sinusoidal wave with a normalized frequency of 0.305 and with fixed amplitude of 10000 to simulate the transverse oscillation. For each signal the amplitude was calculated and the ripple was compared as can be seen in Fig. 4.1. The frequency response of the two filter can be seen in Fig. 4.2 where it is clear that the optimized filter has a much flatter magnitude response in the relevant band.

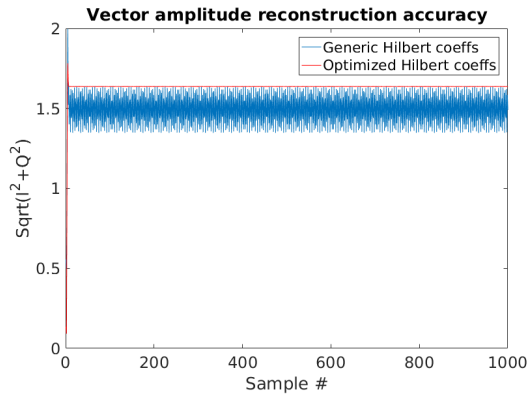


Figure 4.1: Comparison of amplitude ripple between two Hilbert filters

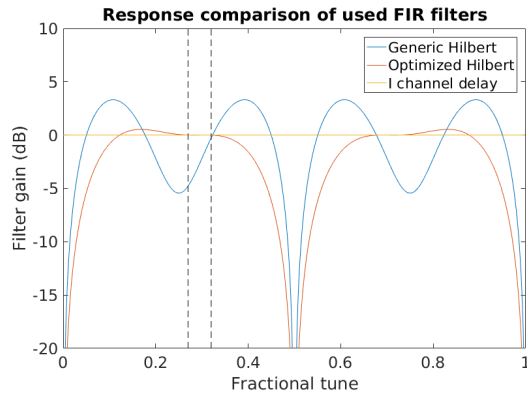


Figure 4.2: Frequency response of the two different filters. The optimized filter has a flat frequency response in the relevant band while the same band in the generic filter differs several dB

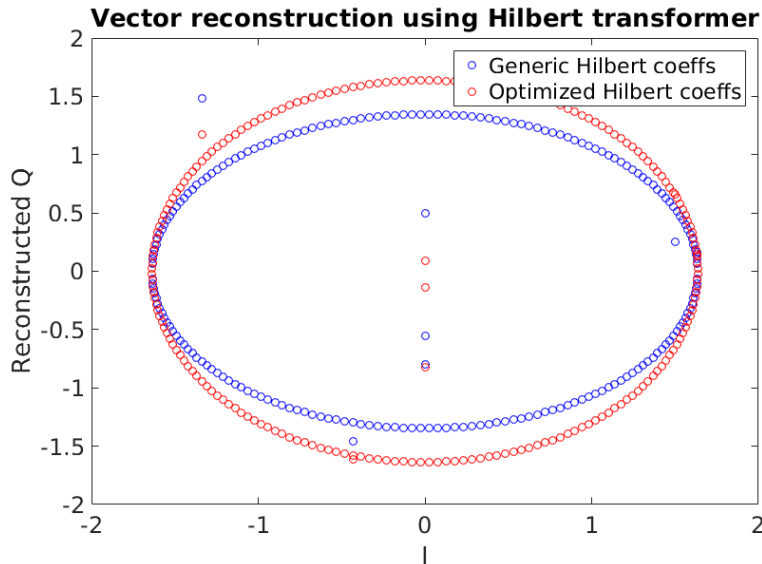


Figure 4.3: Comparison of the vector reconstructions using the two different filters

The optimized filter generated from Matlab requires 7 multiplications and the calculated filter requires 4 multiplications since the zero valued taps can be ignored so it is a question of computational performance versus signal performance.

Both of them can be efficiently calculated using AVX registers as mentioned in Sec. 2.3.5. If the calculated filter from Eq. 4.9 is used then the analytic signal for two turns can be calculated simultaneously since it only requires 4 multiplications and one AVX register can hold 8 single precision floating points. Listing 4.2 shows how this can be accomplished using Intel intrinsics as described in Sec. 2.3.5.

Listing 4.2: Hilbert transform using Intel Intrinsics

```
__m256 constants = _mm256_set_ps(-0.2122, -0.6366, 0.6366,
0.2122, -0.2122, -0.6366, 0.6366, 0.2122);

__m256 turn_data = _mm256_set_ps(turn[0], turn[2], turn[4],
turn[6], turn[1], turn[3], turn[5], turn[7]);

__m256 res = _mm256_mul_ps(constants, turn_data);
__m128 lower = _mm256_extractf128_ps(res, 1);
__m128 upper = _mm256_extractf128_ps(res, 0);
//Sum
lower = _mm_hadd_ps(lower, lower);
__m128 XI0 = _mm_hadd_ps(lower, lower);
upper = _mm_hadd_ps(upper, upper);
__m128 XI1 = _mm_hadd_ps(upper, upper);
```

Where XI0 and XI1 corresponds to $x_i[0]$ and $x_i[1]$ in Eq. 4.1. Because $h[n]$ satisfies the symmetry condition $h[n] = -h[M - n]$ for $0 \leq n \leq M$ the phase is exactly 90° plus a linear component corresponding to a delay of $n_d = 3$ turns. So to correct for this, the real signal $x_r[n]$ needs to be delayed 3 turns to be in phase with $x_i[n]$.

Listing 4.3: Instantaneous amplitude calculation using Intel intrinsics

```
//samples
float samples[16]={};
//Result from Hilbert transform
float hilbert[16]={};
__m256 Q = _mm256_load_ps(samples);
__m256 I = _mm256_load_ps(hilbert+3);
__m256 Qpow = _mm256_mul_ps(Q, Q);
__m256 Ipow = _mm256_mul_ps(I, I);
__m256 res = _mm256_add_ps(Qpow, Ipow);
//square contains the instantaneous amplitude
__m256 square = _mm256_sqrt_ps(res);
```

After this step the instantaneous amplitude is calculated as seen in Listing 4.3, and is ready to be analyzed by any potential instability detection algorithm.



5 Methodology

The method was split into three separate steps: pre-study in Sec. 5.1, implementation in Sec. 5.2 and evaluation in Sec. 5.3. The goal of the pre-study was to define the architecture of the project and verify the feasibility. The goal of the implementation part was to have a fully operational system that could be tested and verified in the evaluation part. Two parts of the project were evaluated, first the function of the system which meant analyzing if it could detect instabilities and then the performance which meant analyzing that the system could handle the throughput.

5.1 Pre-Study

This project required a great deal of knowledge in several fields such as particle accelerators technology to understand the cause of instabilities, knowledge in digital signal processing to properly process the data and also in parallel programming to handle the vast amount of information. This information was gathered by finding relevant source material in this field and get an overview on how this problem could be solved. The relevant source material included articles, conference papers, and books in all fields.

To implement this, knowledge about the available software and hardware at CERN had to be collected which among other things were done by a one-week course about the FESA framework. From all the information gathered an architecture of the system could be created and the feasibility of the project could be analyzed. The architecture design involved how the system would be implemented, which parallel programming paradigm should be used and how the data should be analyzed. The result of the pre-study was a well-defined architecture that could be implemented and evaluated.

5.2 Implementation

From the pre-study, it was clear that the data could be processed by a system which used the pipeline programming paradigm which would be a part of a FESA class. This came the fact that the data should be analyzed in stages and the data arrives as a stream, this is very similar to how DSP processors function which mostly implements pipelines. This new FESA class

would run on the ObsBox servers at point 4 in the LHC ring where they would acquire data from the ObsBoxBuffer FESA class. To handle the throughput and have complete control of the code without relying on the compiler, all the crucial parts of the computational pipeline would be coded using Intel Intrinsics. The reason for this was the old compiler which is used at CERN and it supports auto-vectorization rather poorly. The software needed to run in parallel was done using one POSIX thread for each stage of the pipeline. The result of the implementation was a complete deployed system that could be evaluated. The system was extended with the data saving features of the ADTBufferSaver FESA class and scripts for analyzing the collected data which made the evaluation easier. This allowed the physicists at CERN to analyze bunch-by-bunch data with recorded instabilities. A byproduct of this system was the ADT transverse activity monitor which was displayed in the CCC. This allowed the operators of the LHC to view the oscillation amplitude for each bunch in the two rings which made it easier to view rapid amplitude growth.

5.3 Evaluation

The two most important aspects of this project were to analyze the data generated by the ObsBox system without saturating the hardware and to properly detect instabilities in the data streams. The performance of the computational pipeline which was created during the implementation phase was evaluated by its maximum throughput. The difference in throughput when using manual optimization and auto-vectorization was evaluated together with the difference in auto-vectorization performance in different compilers. This was done by generating data and passing it to the different stages of the pipeline to find the bottlenecks and then evaluate the performance of the whole pipeline.

The functional performance of the system was analyzed by first simulating rapid amplitude growth in a testing environment to see that it properly detected it and by using stored data. This was done because most of the development took place when there was no beam in the LHC. When the LHC started up in May 2017 the system was tested by letting it run for a long time and collect data. The data collected was analyzed manually by creating plots and visually inspecting them. The ADT transverse activity monitor combined with the LHC operators logbook helped a lot during this process since potential instabilities were often logged.

5.4 Presentation of Results

The result from this thesis is a completely operational system, deployed at CERN, together with the source code for it along with the results and the discussion in this thesis report. The code is available at CERN gitlab. Preliminary results of this thesis were also presented in a paper which was presented at IPAC 2017 [54].



6 Implementation and Architecture

This chapter explains the architecture of the system in Sec. 6.1 and the implementation of the system in Sec. 6.2.

6.1 Architecture of the ADT Instability Detection System

This section explains the overall software architecture of the system and also results from tests that verified that the system could be implemented.

6.1.1 Verifying That FESA Can Handle the High Bandwidth Data Streams

FESA as described in Sec. 3.1 is the standard framework at CERN for developing equipment software. If it can be used in this project it would make the development more straightforward since that would require a more complicated deployment of the software together with a more complicated integration to the CERN infrastructure. There is no known project at CERN which uses the framework in this manner so the function must be verified before a design decision is made. One fact that makes the verification simpler is that the data will be transferred between two FESA classes which are running on the same physical machine. It also helps that the data from the ObsBoxBuffer FESA class as described in Sec. 2.2.8 have a time stamp which tells when the buffer was frozen. With this information, it is easy to calculate the time it takes for the data to reach its destination.

The ObsBoxBuffer FESA class and the new ADT transverse instability detection class will run on the same physical machine so the data will never reach the physical network. It will instead be routed through the internal loopback device. The bandwidth of the loopback device on a target machine was measured using iPerf [30] for a quick sanity check and the measured result was ≈ 9000 MB/s. The required bandwidth is $11245 \cdot 3564 \cdot 2$ B/s ≈ 80 MB/s so any potential limitation should lie in the FESA framework. To test this, a simple FESA class was created with an association relationship to the ObsBoxBuffer class. This means that the test class is aware of this class and can subscribe to properties of that class.

To make it as real as possible the test class used a multi-threaded event producer which

handled the subscription. When new data was available it attached the data as a payload to a real-time event and triggered a real-time action that could analyze the data.

Listing 6.1: Multi-threaded event producer

```
class MultiThreadedCESProducer
{
public:
MultiThreadedCESProducer(Device* device)
{
    proxyInterface.subscribe(device->BufferName.getAsString(),
        "Acquisition", "");
}
bool produceEvent(fesa::RTEvent& event)
{
    std::string deviceString;
    std::string propertyString;
    std::auto_ptr<PropertyData> data = proxyInterface.waitNotification(
        deviceString, propertyString);
    boost::shared_ptr<RTEEventPayload> payload(new OnSubscriptionRTEEventPayload(
        deviceString, propertyString, data));
    event.setPayload(payload);
    payload.reset();
    event.setMultiplexingContext(event.getMultiplexingContext());
    return true;
}

private:
fesa::ProxyInterface proxyInterface;
}
```

The function *produceEvent* in Listing 6.1 is called in a loop from the underlying FESA framework. This eventProducer triggers a real-time event which can extract the payload and do the analysis. To measure the time it takes for the data to reach the real-time event from the point the buffer is frozen, the difference between the trigger stamp in the payload and the system time is printed. The buffer is frozen periodically every 4096 turns.

Listing 6.2: Real-time event

```
class OnMultiThreadedCES : public OnMultiThreadedCESBase
{
public:
OnMultiThreadedCES(){};
void execute(fesa::RTEvent* pEvt)
{
    std::auto_ptr<const ObsBoxBuffer::AcquisitionPropertyData> data =
        OnSubscriptionRTEEventPayload::extract<ObsBoxBuffer
            ::AcquisitionPropertyData>(*pEvt);
    std::cout<<fesa::getSystemTime()-data.triggerStamp.get()<<std::endl;
    //This is a 4096 times 3564 matrix containing all the positional data
    fesa::ImmutableArray2D<int16_t> positional_data = data->getData();
}
};
```

Both the trigger stamp and the system time have nanosecond resolution and the time was measured 300 times. From these samples, the maximum time was 469 ms and the minimum

time was 429 ms with an average of 447 ms. By also measuring the time the producer waits for new data and the time it took for the real-time action to receive the data, Fig. 6.1 could be created. From Fig. 6.1 it is visible that the data transfers overlap. The setup from Listing 6.2 and Listing 6.1 was left running over 24 hours and during that time the skew between triggering and receiving the data was stable with an average of 447 ms.

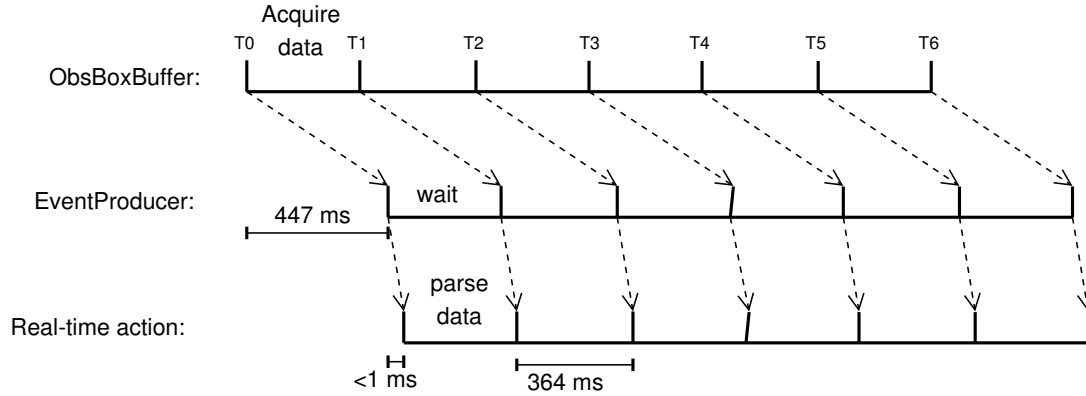


Figure 6.1: Diagram of data transfer times from ObsBoxBuffer

The data transfer was the only real concern regarding FESA because when the data has been received in the real-time event any potential parallel programming paradigm in the C++ language is available to be used for parsing the data. It will also make it easier to distribute the result from the analysis over CERN since it can be published through a FESA interface.

6.1.2 Proposed System Design

The proposed solution for the ADT transverse instability detection system is to create a FESA class with 4 instances, one per available server, which handles one transverse plane each. It is proposed that the data stream from the Q7 pickup is used since it has the best SNR. The new class will have an association relationship with the already existing ObsBoxBuffer class which allows it to start subscriptions to the properties of that class. The shortest 4096 turns buffers for the Q7 pickup will be configured to periodically freeze every 4096 turns so the new class receives a full-rate data stream. The new class will have a multi-threaded event producer which subscribes to these buffers and creates a real-time event which triggers a real-time action in which the data analysis can be performed. The data analysis in the real-time event must be able to analyze 4096 turns worth of data in 364 ms, preferably less to have a margin, as not to saturate the available hardware. If an instability is detected, a trigger will be sent through the LIST network. The LIST network has a FESA class for interfacing with it and in the first version it is simplest to have an association with that FESA class and send a trigger that way and in the future do it properly using dedicated hardware. A block diagram of the proposed structure can be seen in Fig. 6.2.

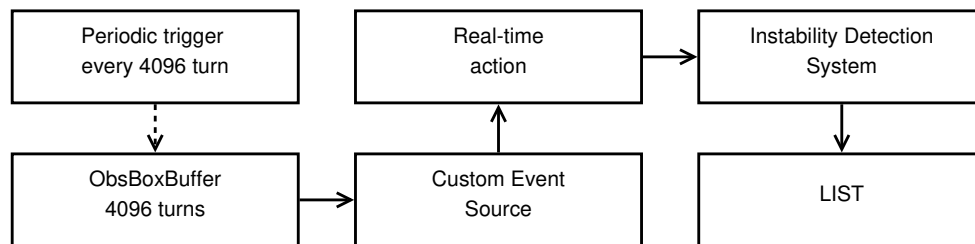


Figure 6.2: Block diagram of the system design

6.1.3 Potential Limitations in ObsBoxBuffer's Capacity

The ObsBoxBuffer class is used in multiple applications and is a vital tool for observing the state of the beam. It is used for diagnosing the ADT, long time frame spectrum analysis to analyze how seismic activity affects the beams, post-mortem analysis of the beams and injection drift observations to name a few. It is also used frequently for machine development which means that during some periods of time, a vast amount of data is generated and saved. There is a possibility that the ObsBoxBuffer class cannot handle this load. Stress tests must be performed after implementation to verify that the new ADT instability detection system does not interfere with the normal operations. It could be that the project interferes with the normal operations either through saturating ObsBoxBuffer with multiple requests at the same time or that the instability detection uses too much of the available computational resources. This could be solved by splitting the fibers from the BPMs to a new server dedicated to online analysis.

6.1.4 Proposed Structure for Exploiting the Algorithm Level Parallelism

From Sec. 6.1.1 it seems clear that the new ADT transverse instability system can be implemented as a FESA class, but it is still not decided how the data will be analyzed when it is available in the real-time event. The implementation does not necessarily need to be portable between platforms since the implementation will most likely only run on the ADTObsBoxes in SR4 so all optimizations can be done with these servers in mind. This means that inline assembly and Intel intrinsics are perfectly valid options, the only problem with this is maintainability. The pipeline pattern which was discussed in Sec. 2.3.7 fits the use case of the data very well since it will process a digital signal in multiple stages. This is a very specific application for a specific platform so the use of existing libraries which abstracts away the platform is not really necessary. To be able to handle the throughput required while keeping the latency as low as possible it makes sense to program as close to the hardware as possible and not introduce any abstraction layers that might cause overhead.

The idea is that the real-time action will serialize the data which is received from the ObsBoxBuffer to turn-by-turn data and push it into a pipeline which contains the appropriate stages to detect instabilities. The stages will be fairly small and the items will be pretty small ($3564 \cdot 4B \approx 14kB$) so how the worker/item relation is set up does not really matter.

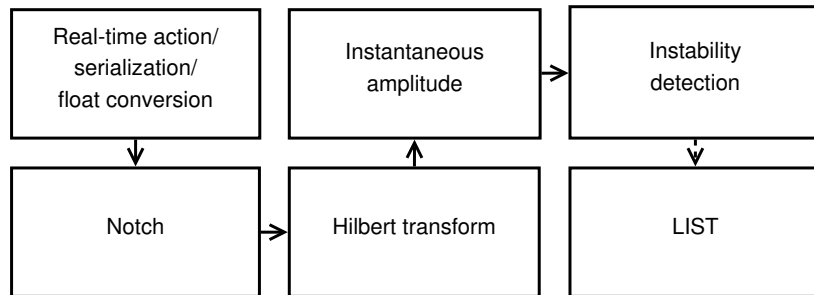


Figure 6.3: Block diagram of the pipeline design

The first step is done in the FESA real-time action where the data is serialized and converted to single precision floating points. What it does is to take each row in the matrix that is received from the ObsBoxBuffer class, convert it and store it in a structure that can be passed between stages in the pipeline. To speed up loading into SIMD registers it is best if the memory is 32 byte aligned.

Listing 6.3: QueueElement used in pipeline

```

class QueueElement {
public:
    QueueElement(std::size_t size);
    ~QueueElement();
    void InsertNewdata(float* input_data, std::size_t size);
    void InsertNewdataExtra(float* input_data, std::size_t size);
    void ChangeSize(std::size_t size);
    //used to store normal data
    float* data;
    //number of elements in data;
    std::size_t data_size;
    //used for extra data after computing I and Q
    float* data_extra;
    //number of elements in data_extra
    std::size_t data_extra_size;
    //internal counter of turns we have calculated
    unsigned long long turn;
    std::shared_ptr<unsigned> bunches;
private:
    //real size of data,
    std::size_t data_size_real;
    //real size of data_extra
    std::size_t data_extra_size_real;
protected:
};

```

The class that is shown in Listing 6.3 is used to pass data through the pipeline. It has two pointer members since it must contain both the real signal and the companion signal after the Hilbert transform, as described in Sec. 4.2. The system was designed so that the set of bunches which are being analyzed can change during runtime through the FESA interface and instances of QueueElement are being reused since it is expensive to allocate and deallocate memory all the time. That is why the class has members to distinguish between real size and data size because if the number of bunches to analyze decreases in size the same memory is used but not all of it.

To pass data between stages a simple blocking queue was implemented which used condition variables, see Sec. 2.3.2, to notify the next stage that data is available for consumption. This means that the stages which have no work to do will be sleeping and not waste computer cycles. This does create some overhead. This is a blocking configuration but it could also be implemented using a lock-free alternative. The reason for blocking is the fact that the data arrives in bursts and most of the time some stages in the pipeline will not have anything to do. With lock-free configuration these will wait for data and use CPU cycles while with a condition variables they will be suspended until data is available.

Listing 6.4: BlockingQueue used in pipeline

```

template<typename T>
class BlockingQueue{
public:
    BlockingQueue() :mtx(), full_(), empty_(), capacity_(MAX_CAPACITY) { }
    void Put(const T& task){
        std::unique_lock<std::mutex> lock(mtx);
        while(queue_.size() == capacity_){
            full_.wait(lock );

```

```

    }
    assert(queue_.size() < capacity_);
    queue_.push(task);
    empty_.notify_all();
}

T Take() {
    std::unique_lock<std::mutex> lock(mtx);
    while(queue_.empty()) {
        empty_.wait(lock);
    }
    assert(!queue_.empty());
    T front(queue_.front());
    queue_.pop();
    full_.notify_all();
    return front;
}
private:
mutable std::mutex mtx;
std::condition_variable full_;
std::condition_variable empty_;
std::queue<T> queue_;
size_t capacity_;
};

```

To communicate with the pipeline a data structure is shared between the real-time action and all the stages in the pipeline. This allows for changing settings in the pipeline during operation, extract debug data, extract performance data and also extraction of detected instabilities because only the real-time action has access to send triggers through the list. It would have been possible to do this in the pipeline but to make the pipeline easy to debug and do performance testing it was best to not have any dependencies on CERN software. This meant the pipeline could be tested on a local computer without access to the technical network.

Since all stages in the pipeline and the real-time action are executing concurrently the data in the structure must be protected against race conditions as mentioned in Sec. 2.3.2. When possible, it is best to use non-blocking synchronization such as atomic operations and only if it is really necessary then use mutexes. The structure can be seen in Listings 6.5 and two different techniques for synchronization are used. Integers and booleans supports atomic operations which solves the synchronization problem but for single precision floating points, a mutex was used.

Listing 6.5: Status structure which is used to communicate with the pipeline

```

class Status {
public:
    Status() {
        last_time_data_was_zero=
            (std::atomic_ullong*)malloc(3564*sizeof(std::atomic_ullong));
        for(std::size_t i=0;i<3564;i++){
            *(last_time_data_was_zero+i)=0;
        }
        transverseActivityMonitor=(float*)malloc(3564*sizeof(float));
    };
    ~Status() {

```

```

};
//used to keep track of the last time a datastream consisted of zeros
std::atomic_ullong* last_time_data_was_zero;
//used to send all instantaneous amplitudes to the device
//so they can be displayed in the CCC
float* transverseActivityMonitor;
//scales the transverseActivityMonitor
float scaling;
//used to configure new Hilbert coefficients
float* hilbert;
//Signal that there are new coefficients available
std::atomic_bool new_hilbert;
//Used to send detected instabilities to the real-time action
BlockingQueue<Unstable*>* winlqueue;
//for how long not to send triggers after injection
std::atomic_ullong prevent_injection;
//protect float arrays
std::mutex* mtx;
//used to analyze each stage in the pipeline for a single bunch
std::atomic_int* pipeline_analyzer;
//tells the pipeline which bunch to analyze
std::atomic_uint bunch_to_analyze;
//the current turn number
std::atomic_llong turns;
};

```

The stages are simple functions which are target functions for Posix threads as described in Sec. 2.3.6. All stage functions take an input queue, an output queue, and the status structure as input parameters. After creation and initialization, each stage waits for data to arrive in the input queue, do the data transformation, and put the item in the output queue.

6.1.5 Exploiting Data-Level Parallelism

Most computations performed in the pipeline stages will be vector operations performed on long vectors consisting of up to 3564 elements. To maximize throughput and fully utilize the available hardware the SIMD registers in the Intel Xeon processors should be utilized. This could be done either by relying on the compiler to optimize all the loops over the elements using auto-vectorization or by manually programming the operations. Since the standard compiler at CERN is GCC 4.4.7 which is an old version released 2012, one year after AVX was introduced, the choice fell on using Intel Intrinsics. Since this is a specific application for a specific platform where throughput is critical this was the best way to guarantee optimized usage of the hardware. The intrinsics were chosen using [29] which is a tool for choosing intrinsics. From the intel guide it is possible to filter out what operation is desirable. There are multiple combinations which can achieve the same results and they were chosen to mimic the normal C++ code as much as possible. No multiply-accumulate instructions were used but it could be used to further improve performance.

6.1.6 Proposed Algorithm to Detect Instabilities

The proposed algorithm is the moving average as described in Sec. 8.4.1 since it is simple and it can use multiple stages in a pipeline to have different window lengths for detecting a wide variety of instabilities.

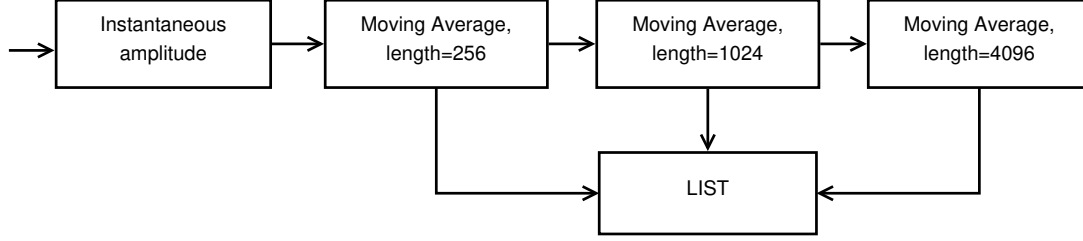


Figure 6.4: Block diagram of instability detection part of the pipeline

If the configuration in Fig. 6.4 is used then the first stage would receive the instantaneous amplitude at a frequency of 11245 Hz and every 256 samples it will compare the average of the latest 256 samples with the average of the 256 samples before that. If the new average is larger than the previous average times a threshold value, a trigger is sent through the LIST network, see Sec. 3.2. This is done per bunch in the LHC so it is known which bunch is unstable and which window detected it. The first stage of length 256 passes the average to the next stage of length 1024 which does the same analysis but on every fourth sample. The second moving average stage passes the average to the third stage which does the same comparison on every fourth sample. It is also possible to calculate the rise time from the averages which can give information about the underlying cause of the instability as discussed in Sec. 2.2.3.

6.2 Implementation of the ADT Instability Detection System

This section explains the implementation of each stage in the computational pipeline.

6.2.1 Retrieve the Data and Triggering a Real-Time Event

The data is retrieved just as discussed in Sec. 6.1.1. There is a simpler way to set this up using the FESA framework but this makes the subscription static and it was preferable to be able to change the subscription during operation. The ObsBoxBuffer allows subscription to a subset of bunches/turns and through the interface of the ADT instability detection system it should be possible to modify which bunches in the machine are being analyzed. When a user modifies the selected filter, the multi-threaded event producer will change the subscription so only the bunches defined by the filter are received. This can be used to lower the stress on the servers.

6.2.2 Serializing the Data and Converting It from Signed Integer to Single-Precision Floating-Point in the Real-Time Action

The procedure for serializing the data is quite simple. For every row in the matrix received from ObsBoxBuffer as described in Sec. 6.2.1 a new QueueElement is created, see Listing 6.3. By using the technique in Sec. 4.1 each row is converted to single precision floating points and passed to the next stage. Here, some optimizations could be made; there is a BlockingQueue as described in Listing 6.4 which passes back old QueueElements from the last stage which has already been used to reduce the time needed for allocating and deallocating memory each time. The complete implementation of this can be seen in Appendix C. The reason for the float conversion and why all data the manipulation is done using floating-point numbers is is the complexity of the analysis. This could be implemented using only fixed-point numbers and it would probably be faster but it would require a much greater care to compensate for quantization noise.

6.2.3 Injection Oscillation Triggering Prevention

The first stage in the pipeline which runs on a separate thread is the stage that prevents sending triggers on injection oscillation. When a bunch is injected into the LHC, the data stream will first consist of zeros and then contain heavy oscillations until it is dampened by the ADT. This oscillation will result in triggers from the instability detection stages unless precautions are taken. In the Status structure, there is an array of 64 bit unsigned integers which describes the latest turn where each bunch had a data stream of zeros. This stage checks if five consecutive turns have zeros in the data stream. If that is the case the probability of it being an empty bunch is pretty big. When this is the case the value for that bunch in the array in Status is updated to the turn number of the latest turn. This value is later used in the instability detection stages to verify that it is not an injection oscillation. The complete implementation can be seen in Appendix D.

6.2.4 Notch Filter

The second stage in the pipeline is the notch filter which centers the transverse position around zero. It is a FIR filter with coefficients:

$$h = [1 \quad -1] \quad (6.1)$$

This is implemented using a circular buffer of length two. When the stage is created it fills the buffer with two elements and after that it enters a while loop which runs forever. In the while loop, it applies the notch filter on the two elements in the circular buffer and stores the result in the oldest element. That element is then pushed to the next stage in the pipeline, a new element is pushed to the circular buffer and the procedure repeats. The whole implementation can be viewed in Appendix E.

6.2.5 The Hilbert Transform Stage

After the notch, the data is passed through the blocking queue to the stage which calculates the analytic signal for each bunch. This is implemented with a circular buffer of length 7 since we are using a 7 tap FIR filter to compute the transform.

$$h = [-0.0906 \quad -0.0198 \quad -0.5941 \quad 0 \quad 0.5941 \quad 0.0198 \quad 0.0906] \quad (6.2)$$

When the stage is created it fills the circular buffer and enters a forever loop. Every time a new item is available it calculates the analytic signal for each bunch and puts the result in the oldest item which is passed to the next stage. It also copies the data from the newest item to the item which is three turns “older” to adjust the phase as described in Sec. 4.2. It was discussed in Sec. 4.2.1 which coefficients should be used, either the ones which required 4 multiplications or the ones which required 6 multiplications. The first implementation used 4 multiplications which meant that 2 bunches could be calculated simultaneously. But it was discovered and will be discussed in Sec. 7.1.2 that this stage was not limiting the pipeline so in the end 6 multiplications were used for a better signal. The whole implementation can be seen in Appendix F.

6.2.6 The Amplitude Stage

The amplitude stage is very simple. Every time an item is available, this stage takes it and computes the instantaneous amplitude of the real signal and the analytic signal as described in Eq. 4.3 in Sec. 4.2. The implementation can be seen in Appendix G.

6.2.7 The Maximum Stage

This is an afterthought because the transverse activity that was being displayed in the CCC showed the average transverse activity from the last 4096 turns. It meant that higher frequency variations in the transverse activity could be smeared out and not be visible in the CCC display. To overcome this limitation, this stage which keeps track of the largest observed instantaneous amplitude from the latest 4096 turns was added. It has an array of floats which contains the maximum value. Every time an item is available it compares the values in the array with the new values and updates the array. Every 4096 turns this array is copied to the Status structure and then reset to zero. The whole implementation can be seen in Appendix H.

6.2.8 Moving Average / Instability Detection Stage

The reason why this project was created was to detect amplitude growth in the bunch-by-bunch positional data streams from the ADTObsBox and this is done in this stage. The process used is described in Sec. 8.4.1 and the length of the windows is passed as a parameter when the stage is created. Multiple of these stages can be used one after another to cover amplitude growth with different rise times. It contains a circular buffer which is of the same length as the window, and two arrays which contain the current summation for each bunch and an old summation for each bunch. When a new item is available the values are added to the current summation array and the values from the oldest item in the circular buffer is removed from the summation and then the new item is pushed to the circular buffer.

A counter keeps tracks of the number of items acquired and when it is equal to the window length, the values in the array containing the old summation are multiplied with the threshold and compared with the new summation. If this is bigger for any bunch, a structure is passed to the real-time event through a blocking queue. After this, the new summation is copied to the array containing the old summation and to the oldest element in the circular buffer, after being divided by the window length, which is passed to the next stage and the counter is reset. The whole implementation can be seen in Appendix I.

6.2.9 Transverse Activity Monitor Stage

The last stage is a simple stage which copies the data after the last average window to the Status structure so it can be copied to a FESA field and displayed in the CCC. The whole implementation can be seen in Appendix J.

7

Results and Discussion

This chapter explains how the system was tested and presents the results from these tests together with a discussion regarding the results.

7.1 Performance Evaluation

This section describes how the pipeline was optimized together with the evaluation of the computational performance of the data pipeline after the optimization.

7.1.1 Optimizing the Pipeline

To analyze the performance of each stage separately, appropriate drivers were created which generated data that could be sent to the stage without any other part of the pipeline interfering. A stub was also created which just cleared the output queue when data was available. To analyze potential bottlenecks in each stage multiple different techniques were used. First Callgrind was used, see Sec. 2.3.9, to see where most time was spent in each stage. Some optimizations could be done here, such as replacing some vectors with arrays.

After profiling each stage, the throughput of each stage was measured. The driver which generated data waited a certain period before sending new data and this time was decreased every 32768 “turns”. If the size of the input queue was larger than 4096 then that stage was saturated. The normal operations for making a thread sleep such as `usleep` and `std::this_thread::sleep_for` could not be used because the time to make a thread sleep and wake up was greater than the shortest time required to saturate some stages. To overcome this, an unoptimized loop was used where the number of iterations decreased over time and the time between sending new data to the stage was measured with the `high_resolution_clock`. The implementation of the driver can be seen in Listing 7.1. The setup

for testing the complete pipeline was similar but a complete matrix was generated instead.

Listing 7.1: Driver for testing each stage

```
signed long timeToWait = 100000;
std::chrono::high_resolution_clock::time_point lastInsert =
std::chrono::high_resolution_clock::now();
unsigned long counter = 0;
while (true) {
    for (std::size_t i = 0; i < timeToWait; i++) {
        asm("");
    }
    QueueElement* temp = new QueueElement(3564);
    temp->data_size = 3564;
    temp->bunches = bunches_pointer_shared;
    stl2->Put(temp);
    if (stl2->Size() > 4096) {
        std::cout << "failed_at_" << static_cast<unsigned long long>
        (std::chrono::duration_cast< std::chrono::nanoseconds >
        (std::chrono::high_resolution_clock::now() - lastInsert)
        .count()) << std::endl;
    }
    lastInsert = std::chrono::high_resolution_clock::now();
    counter++;
    if (counter == 32768) {
        counter = 0;
        if (timeToWait > 1000) {
            timeToWait -= 1000;
        } else {
            timeToWait -= 10;
        }
    }
    std::cout << "time_reduced_to_" << timeToWait << std::endl;
}
}
```

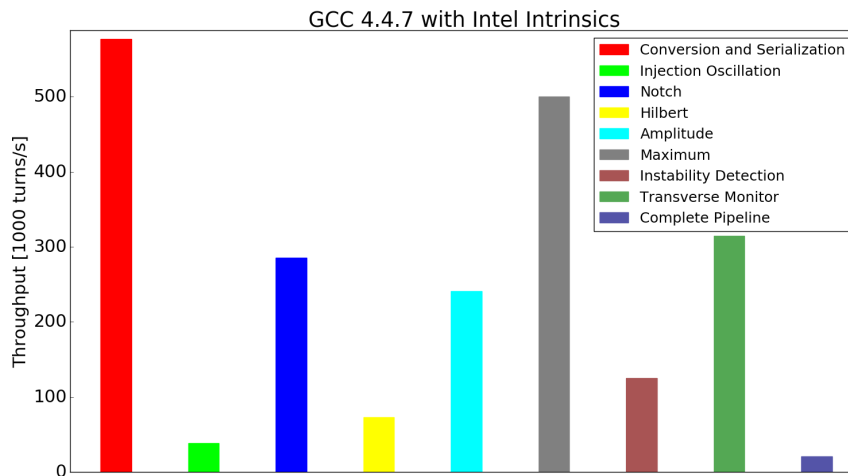


Figure 7.1: Throughput of each stage in the pipeline and the complete pipeline

Figure 7.1 shows the throughput for the every stage in the pipeline (a higher bar is better) and also for the complete pipeline. It is clearly visible that the stage which prevents injection oscillation triggering is the limiting factor. The average measured throughput of that stage was 38275 turns per second which is approximately 240 % better than required (11245 which is the revolution frequency of the LHC). The average throughput of the complete pipeline was 20820 turns per second which is around 85 % better than required which is a reasonable safety margin. The reason why this is the limiting stage is that every time a new item is available this stage has to go through the item and look up which bunch is in which place since the set of bunches being analyzed can change. If it is an empty bunch it also has to access the memory in the Status structure and update it. If greater performance is required then this is where optimization is needed. The implementation could be improved if the conversion and serialization stage pads the data so data items with 3564 bunches are always passed through the pipeline.

To visualize the activity in each stage Intel VTune was used, see Sec. 2.3.9. In Fig. 7.2 it is visible that each stage has time to recover before new data is available while in Fig. 7.3 it is visible that the injection prevention is becoming saturated and cannot handle any higher throughput. This data was captured during the throughput tests mentioned earlier.

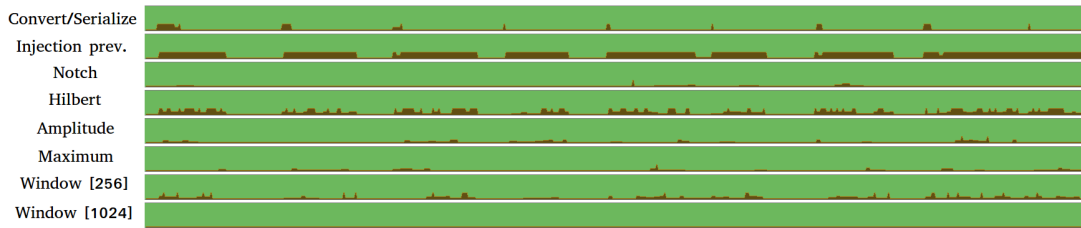


Figure 7.2: Activity in each stage when the pipeline is not saturated

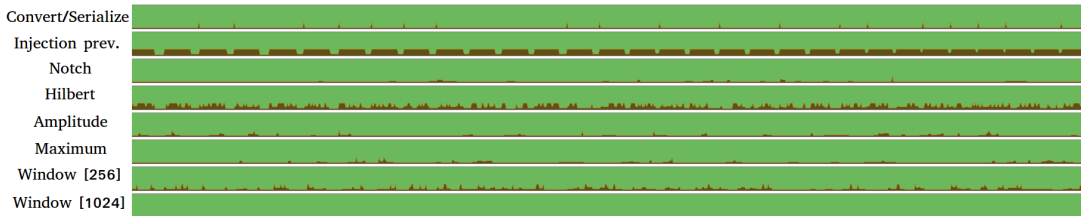


Figure 7.3: Activity in each stage when the pipeline is saturated. The Injection prevention stage could not handle any higher throughput

7.1.2 Performance Comparison Between Different Compilers

To see how auto-vectorization performance has increased over the last years the same tests as described in Sec. 7.1.1 were compiled with three different compilers and executed. The compilers were GCC 4.4.7 since it is the standard compiler at CERN, GCC 5.4.0, and ICC 17.0.0. ICC 17.0.0 was chosen because it is the latest Intel compiler and GCC 5.4.0 because it was released around the same time as ICC 17.

To see how the compilers handled auto-vectorization all stages had two implementations, one with manual optimizations using Intel intrinsics and one normal implementation using standard operations. Both of them were compiled with maximum optimization enabled.

```
-std=c++0x -O3 -msse4.1 -mavx -march=native
```

The results from all tests can be seen in Fig. 7.4. There is a clear difference between manual vectorization and auto-vectorization when using the older GCC version, every stage is faster. All loops were perfectly vectorizable and the memory was already aligned for AVX registers but in most cases, it still used scalar operations when examining the generated assembly code. The results for the old and the new GCC compiler when using intrinsics are pretty close but when normal operations are used the newer one has improved or remained the same in all stages except the maximum stage. The only thing that is done at that stage is a comparison and an assignment. When examining the assembly code it is clear that both GCC compilers use scalar operations as can be seen in Listings 7.3 and 7.4. The major difference is that the older GCC iterates over three scalar instructions so it does one comparison each iteration and then branches. The newer version, however, does 8 comparisons in each iteration and with each comparison, there is a possible branching. The reason for the decrease in throughput could be bad branch prediction in the pipeline. The ICC compiler, however, generates perfect vector code, see Listings. 7.2. It even uses 8 registers so the instructions can be independent in the instruction pipeline.

The ICC was the only compiler that generated more efficient executables from unoptimized code than from the manually optimized code. Most stages could handle a higher throughput when the normal code was used and this reflected in a higher throughput through the whole pipeline. Figure 7.5 shows the throughput through the complete pipeline when compiled with the different compilers. Unoptimized code compiled with ICC could handle approximately 42000 turns per second while the manually optimized code could only handle 32000 turns per second.

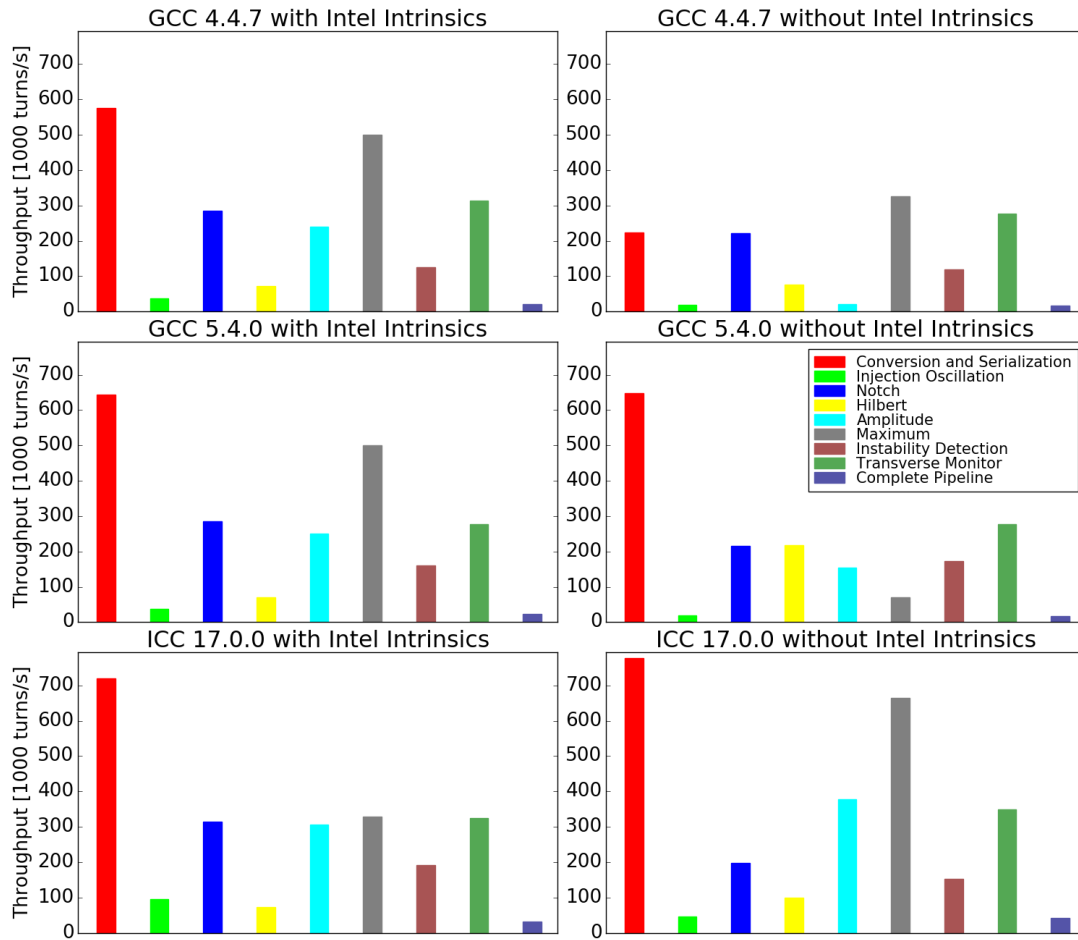


Figure 7.4: Comparison of throughput using GCC and ICC

Listing 7.2: ICC Maximum assembly

```

406a50: c5 fc 10 04 8e      vmovups (%rsi,%rcx,4),%ymm0
406a55: c4 c1 7c 5d 0c 88    vminps (%r8,%rcx,4),%ymm0,%ymm1
406a5b: c4 c1 7c 11 0c 88    vmovups %ymm1, (%r8,%rcx,4)
406a61: c5 fc 10 54 8e 20    vmovups 0x20(%rsi,%rcx,4),%ymm2
406a67: c4 c1 6c 5d 5c 88 20 vminps 0x20(%r8,%rcx,4),%ymm2,%ymm3
406a6e: c4 c1 7c 11 5c 88 20 vmovups %ymm3,0x20(%r8,%rcx,4)
406a75: c5 fc 10 64 8e 40    vmovups 0x40(%rsi,%rcx,4),%ymm4
406a7b: c4 c1 5c 5d 6c 88 40 vminps 0x40(%r8,%rcx,4),%ymm4,%ymm5
406a82: c4 c1 7c 11 6c 88 40 vmovups %ymm5,0x40(%r8,%rcx,4)
406a89: c5 fc 10 74 8e 60    vmovups 0x60(%rsi,%rcx,4),%ymm6
406a8f: c4 c1 4c 5d 7c 88 60 vminps 0x60(%r8,%rcx,4),%ymm6,%ymm7
406a96: c4 c1 7c 11 7c 88 60 vmovups %ymm7,0x60(%r8,%rcx,4)

```

Listing 7.3: GCC 4.4.7 Maximum stage assembly

```

407260: c5 fa 10 04 86      vmovss (%rsi,%rax,4),%xmm0
407265: c5 fa 10 0c 82      vmovss (%rdx,%rax,4),%xmm1
40726a: c5 f8 2e c8         vucomiss %xmm0,%xmm1
40726e: 76 09              jbe 407279 <removed address>

```

Listing 7.4: GCC 5.4.0 Maximum stage assembly

```

40882f: c5 fa 10 41 18      vmovss 0x18(%rcx), %xmm0
408834: c5 fa 10 4a 18      vmovss 0x18(%rdx), %xmm1
408839: c5 f8 2e c8         vucomiss %xmm0, %xmm1
40883d: 76 05              jbe     408844 <removed address>

```

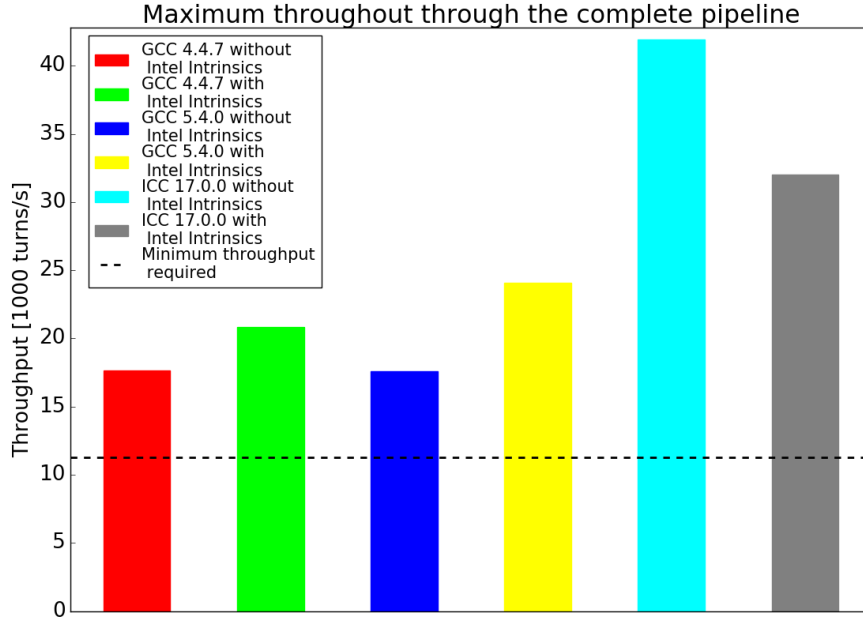


Figure 7.5: Comparison of throughput through the complete pipeline using GCC and ICC, with and without manual optimization

These results are not very general and only apply to this specific problem but it seems that compilers are getting better at auto-vectorization and in the future, data-level optimizations might be limiting the compiler to fully optimize the executable. It is, of course, possible to use the ICC compiler in a FESA project by compiling the code into a static library and then link it to the project.

7.2 Functional Results

This section describes how well the system could detect beam instabilities.

7.2.1 Testing the Algorithm in an Offline Environment

A test environment was created to visualize how the ADT instability detection system behaved when fed with different signals. It had a simple graphical interface which was created using Qt5. A simple signal generator was implemented which generated a sinusoid with variable amplitude and oscillation frequency:

$$f[t] = A \cdot \sin(2\pi ft) \quad (7.1)$$

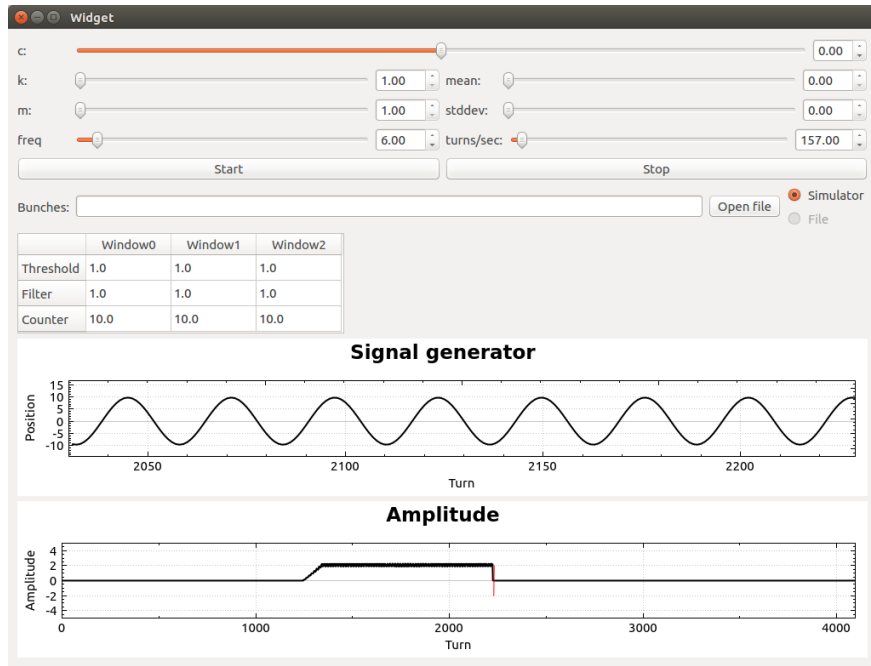


Figure 7.6: Test environment

The test environment also allowed for loading stored HDF files which had been saved during machine development runs. These are normally 32768 turns long because they were captured using the 32k buffer in the ObsBoxBuffer FESA class and contained data from both stable and unstable beams. An example of an unstable beam can be seen in Fig. 7.7 and a stable beam can be seen in Fig. 7.8. This test environment was in the end mostly used to see where the system detected instabilities after it had been tuned using the automated tuning process described in Sec. 7.2.2.

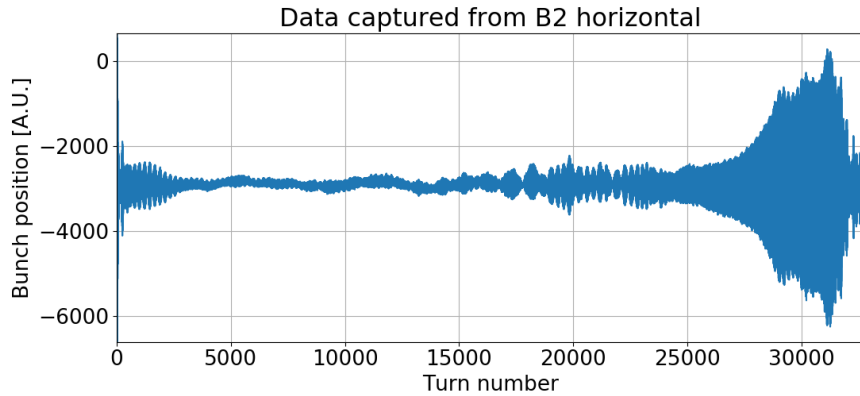


Figure 7.7: Unstable beam

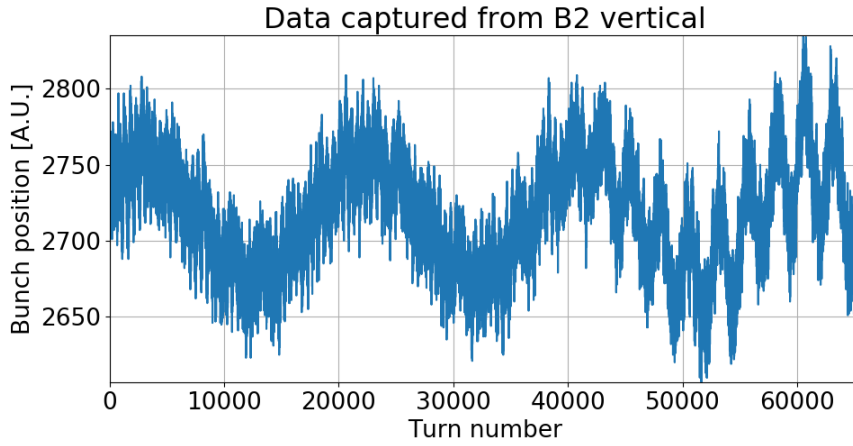


Figure 7.8: Stable beam with orbit drift

7.2.2 Automated Tuning of the Moving Average Threshold

The first step towards automatic tuning was to find relevant data. There was plenty of data stored but there was no metadata which described what each file contained. To solve this plots for all files with all bunches were generated to manually filter out interesting files. Patterns like the one in Fig. 7.7 were coveted. When enough files had been selected, each bunch in each file was plotted and from that separate bunches were selected. The data for the bunches which contained instabilities were extracted into separate HDF files with added metadata which described where the instability began and ended.

The only available tuning is to configure the threshold for each window. In the current configuration there are three windows of length 256, 1024, and 4096 turns which were chosen because of the normal rise-times of instabilities observed in the LHC. The automatic tuning was done using a Python script which tested evenly spaced thresholds for each window and then recorded which instabilities were detected and any false triggers. From this data, the configurations of the different thresholds for each window which detected most instabilities and as few false triggers as possible were found. This was done according to Algorithm 1

```

hits=[];
misses=[];
for Every window do
    for Threshold=0;Threshold<2;Threshold+=0.1 do
        for Every File do
            Run the data in the file through the pipeline and record where it trigger;
            if Trigger was in an instability then
                add it to hits;
            end
            else
                add it to misses;
            end
        end
    end
end
end

```

Find the configuration of different thresholds for each window which achieves the maximum amount of hits at possible and as few misses as possible;

Algorithm 1: How to automatically tune the threshold

The rating score for different configurations was calculated with +1 for any instability detection and -5 for every false trigger. The scores were achieved from testing, the value for the false trigger was decreased until a low amount of false triggers were achieved. The data which was used to test the algorithm contained as much normal data with a stable beam as data with instabilities. The result of the tuning was a threshold of 2 for the window with length 256 samples and 2.22 for the other two windows. This means that the first window of length 256 will detect instabilities with a rise time shorter than 22 ms, the second window with length 1024 will detect rise times shorter than 100 ms and the last will detect rise times shorter than 400 ms. With this configuration, the ADT instability detection system detected 95 % of all instabilities with 5 % percent of all triggers being false when tested on the extracted data.

7.2.3 Setting up the System

Most of the development took place during the extended year-end stop of LHC 2016/2017 which meant that testing could only be done with stored or simulated data as described in Sec. 7.2.1. After the startup in May 2017, it was possible to test the ADT instability detection system with real data. The system was deployed and ready to be tested before the first beam in the machine and all the data that was generated was logged in the LHC logging system. The data could be accessed through the TIMBER interface, see Sec. 2.3.9.

The ADT instability detection system publishes data about the detected instabilities every time it receives new data from the `ObsBoxBuffer` class, which is approximately every 364 ms. The data which is published consists of several different data fields:

- A boolean vector of length 3564 which describes which bunches are unstable.
- A single boolean which is the result of all the values from previous vector OR:ed.
- An integer vector of length 3564 which describes which window detected an instability. The windows are represented as 1,2 and 4. If the first bunch is deemed unstable by the second and third window then the value in first position in the vector is 6.
- A floating point vector of length 3564 which contains the average instantaneous oscillation amplitude for each bunch from the last 4096 turns.
- A floating point vector of length 3564 which contains the maximum instantaneous oscillation amplitude for each bunch from the last 4096 turns.
- A floating point number which contains the average of all non zero average oscillation amplitudes.
- A floating point number which contains the maximum of all maximum oscillation amplitudes.

It was also desirable to introduce a proper service to save the actual bunch-by-bunch positional data from the `ObsBoxBuffer`, not only for this project but for several other projects as well. To support this a new FESA class called *ADTBufferSaver* was developed. It was a simple class that could be configured to subscribe to up to four buffers. When any of these buffers was updated, the data would be saved as an HDF file. It is now used for several different purposes. It saves all injection buffers on every injection in the LHC for long term drift observation. It saves the new instability buffers when instabilities are detected and also post-mortem buffers which are frozen when the machine dumps the beam. When any of this freeze, the *ADTBufferSaver* would save the data onto the network file system. An overview of how the system works together can be seen in Fig. 7.9. The new instability buffers are

65536 turns long and there are four of them (one for each transverse plane) which receives data from the Q7 pickup since the instability detection uses that pickup.

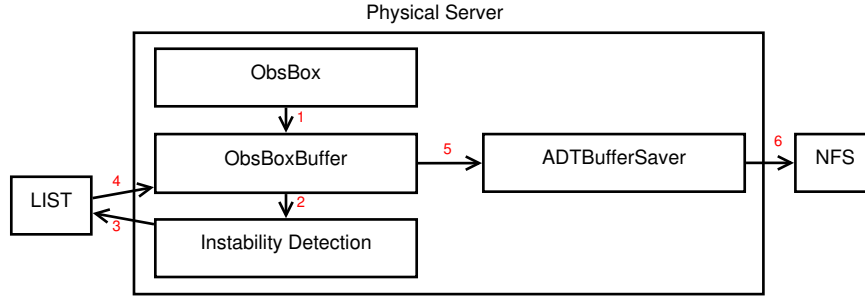


Figure 7.9: How the ADTBufferSaver fits in the system

The following list describes how the different systems work together, as visualized in Fig. 7.9.

1. The ObsBox class collects data which is stored in a large circular buffer, a periodic trigger saves a subset of the data into a 4096 turns long buffer
2. When the 4096 turns long buffer is updated, the instability detection analyzes the data and detects a potential instability
3. It sends a hardware trigger to the LIST network
4. The new 65536 turns buffer for instability is frozen by the LIST trigger
5. The ADTBufferSaver class receives the data from the 65536 turns long instability buffer
6. The data is saved on the network file system for later analysis

7.2.4 Setting the Thresholds

In the first setup for testing the system, the thresholds which were calculated from the automatic tuning were used but it turned out that they were too high since there were no triggers being generated. Not even during injection with injection prevention disabled. The conclusion from this was that the stored data did not represent the normal state of the LHC very well and a new procedure for setting the thresholds was thought up:

```

for Every window do
    Disable all other windows by setting a very high threshold (>10);
    Set a high enough threshold (>4) on current window so there are no triggers;
    while No triggers do
        | lower threshold by 0.1;
    end
    Increase threshold for current window by 0.2 to have a margin;
end

```

Algorithm 2: How to set the threshold for the trigger

This was done for all four transverse planes during flat top with stable beams. They were configured using the FESA navigator and the triggers from the system could be observed using the FESA navigator as well. The margin was chosen arbitrary to be 0.2 because a margin of 0.1 still generated some sporadic triggers, the result from this was (VB1= vertical beam 1, HB2= Horizontal beam 2):

	VB1	VB2	HB1	HB2
256	1.8	1.8	2.0	3.0
1024	1.5	1.5	1.8	2.5
4096	1.4	1.4	1.5	2.0

Table 7.1: Threshold after applying Algorithm 2

The reason for the high thresholds in the horizontal plane in beam two turned out to be caused by a glitch randomly appearing in the data stream which caused the bunch position to be registered as 0 for one turn, as can be seen in Fig. 7.10. This caused the instantaneous amplitude to momentarily increase rapidly due to the notch filter, which triggered the instability detection. The cause has not been investigated yet at the time of writing so the instability detection for that plane is more or less disabled. It could possibly still be useful for triggers from the longest window where the short amplitude increase is less noticeable.

The last step was to configure for how long the injection oscillation triggering prevention would be enabled. This was done by using the thresholds that were produced experimentally and observing the generated triggers during injection. The number of turns which it was disabled was incremented by steps of 1024 until triggers were no longer generated during injection. The result from this was that no triggers were sent for bunches before 12288 turns had passed after they were injected. After this configuration, the system was completely autonomous and could collect data from instabilities.

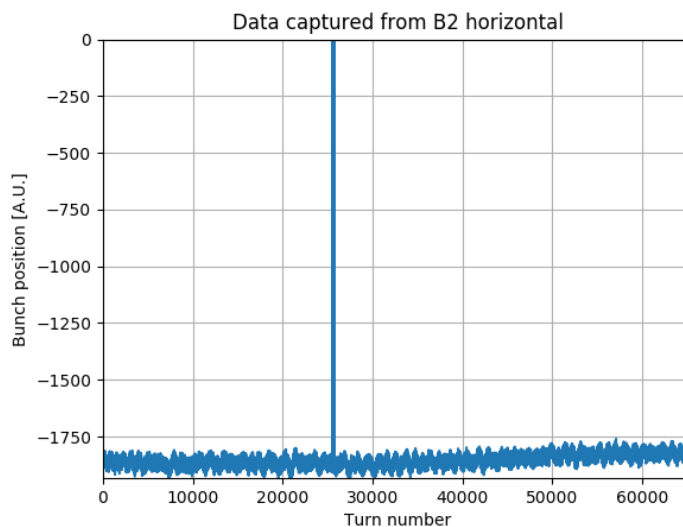


Figure 7.10: Glitch in the data stream for HB2

7.2.5 Tools To help Analyze the Collected Data

Every file that was collected with the help of the ADT instability detection system was saved on CERN's network file system (NFS) where the ADT had 6 TB of space to store data. This space was used for post-mortem, injection, and instability data. All files had the same naming convention:

Listing 7.5: Filename convention

```
06177_Inst_B2H_Q7_20170908_13h18m42s.h5
```

Where the first part describes during which LHC fill it was generated, the second describes what generated the file (Inst=Instability, Inj=Injection, Pos=Post-mortem), the third which transverse plane, the fourth which pickup and the last two which time it was frozen. They were organized in the following file structure:

```
nfs
├── cs-ccr-bqhtnfs
│   ├── fillnr
│   │   ├── postmortem_data
│   │   ├── instability_data
│   │   └── injection_data
│   ├── plots
│   │   └── fillnr
│   └── scripts
```

A Python script was created which used the time in the filename to extract which bunches were unstable that time from the LHC Logging System by using PyTimber. After that, the relevant bunches could be plotted to see if they showed signs of amplitude growth. Another Python script ran that script for every file in a specific fill folder. This meant that after a fill in the LHC, the script could be started and when done, the plots could be inspected manually.

7.2.6 Results From Online Testing

The complete system with the data acquisition using ADTBufferSaver was fully implemented the 5th of September. By the 26th of September 2017, 253 GB of data divided over 76 LHC fills had been collected by buffers freezing because of triggers sent by the ADT instability detection system. By using the Python scripts mentioned in Sec. 7.2.5 plots could be generated after every fill to analyze the performance of the instability detection. From the 580 TB which was analyzed during this period, only 253 GB was saved and, when manually analyzing the data only about 10% contained interesting activities. There was still a lot of triggers generated during injection which generates a lot of unnecessary data. Injection oscillation in the injected bunches was not the only problem; when they were injected, they perturb and change the orbit of bunches in the other beam, which caused a rise in amplitude, see Fig. 7.11.

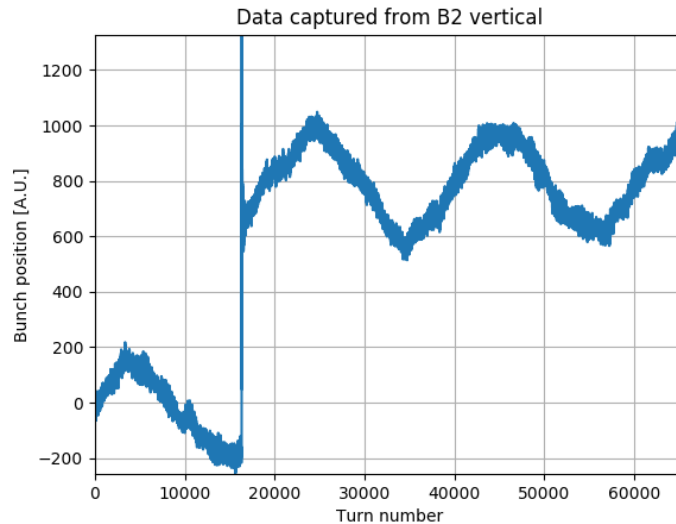


Figure 7.11: Beam perturbed by injection

To be able to completely ignore triggers during injection a new feature was added which disabled the system when the energy in the LHC was lower than a specified value. The value could, for example, be set to 6.5 TeV to only detect instabilities when the machine was at flat-top (fully accelerated). The thresholds could also be increased to further limit the amount of data generated but it was at a manageable level.

Examples of activities that were captured can be seen in Figures 7.12 to 7.17. To analyze the data which was stored required quite a lot of manual work and it would be possible to use this system as a first level data filter and then use more advanced techniques on the filtered data. In the end, the system worked well with only a few bugs that had to be fixed, such as making sure that the queue with instability data was cleared before more data was pushed to the pipeline. If this was not done there was a chance for a potential deadlock.

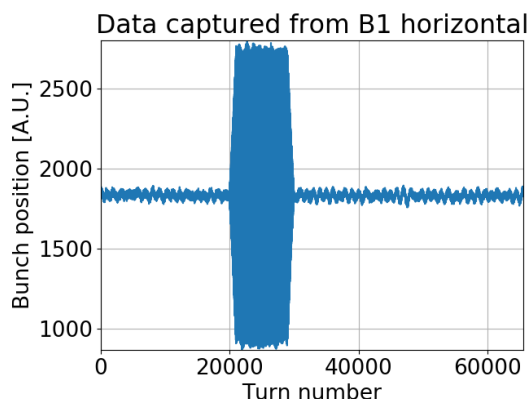


Figure 7.12: Excitation for coupling measurement during fill 6200 using the ADT as exciter

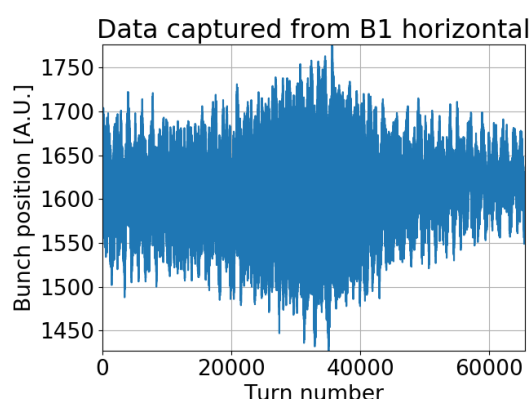


Figure 7.13: Amplitude growth during fill 6200

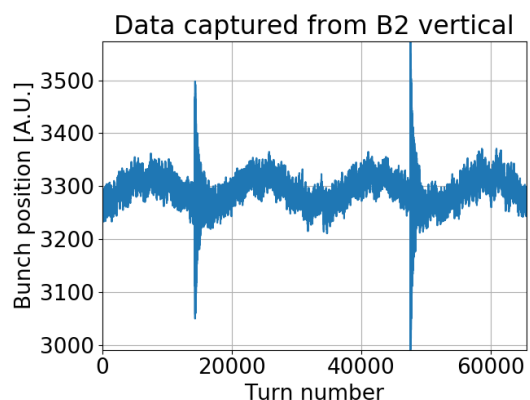


Figure 7.14: Short excitation for tune measurement during fill 6221 using the ADT as exciter

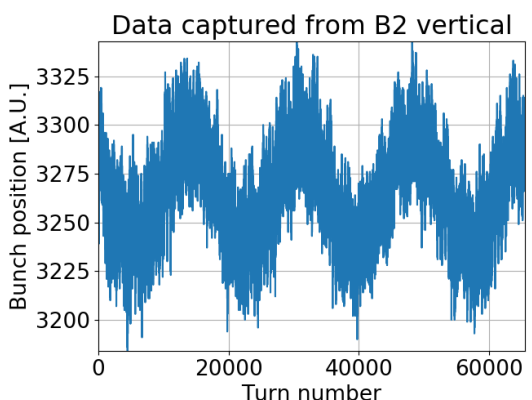


Figure 7.15: Low frequency orbit drift during fill 6221

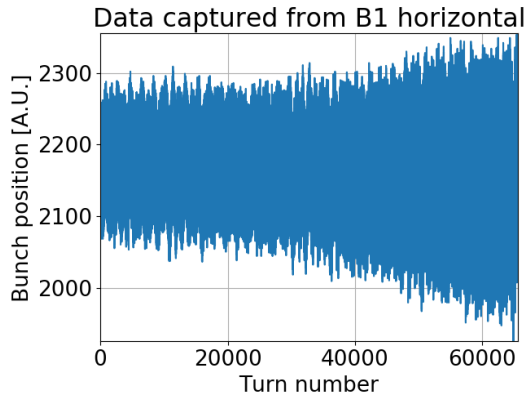


Figure 7.16: The amplitude of bunch 731 during fill 6266 with slow rise time

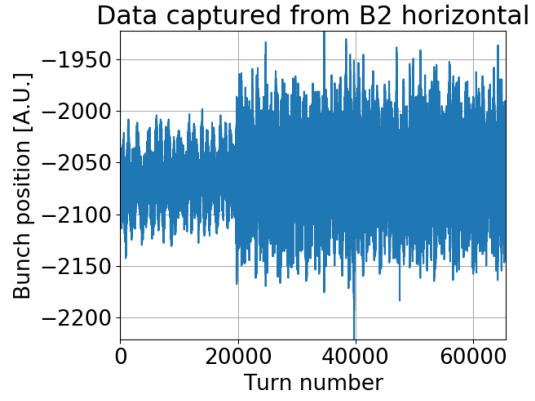


Figure 7.17: Rapid amplitude growth during fill 6227

7.2.7 Usage of the real-time Transverse Activity Monitor in the CCC

The display in the CCC has become an important operational tool to observe the real-time bunch-by-bunch transverse activity. It is used as an early warning for bunch instability or detection of unwanted transverse excitation (i.e. by injection cleaning or injection kicker edge displacement). It allows the operators of the LHC in the CCC to monitor the peak activity over time, slow activity (average over 4096 turns), fast activity (maximum over 4096 turns) and the evolution of activity over time. The calculated activity is calibrated in micrometers, so the impression of how severe the activity is immediately visible. Everything is displayed per bunch so it is always possible to determine which bunch is becoming unstable. Examples of logbooks entries from the LHC operators can be seen in Figures 7.18 to 7.20.

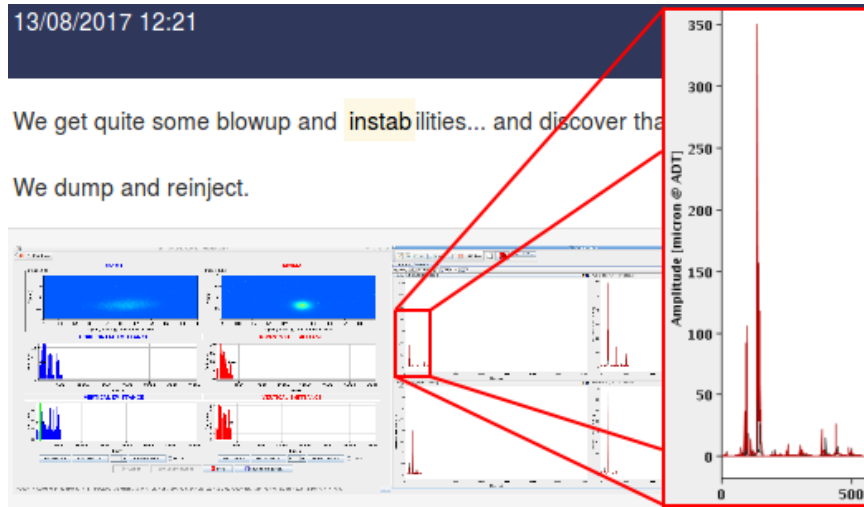


Figure 7.18: Entry in the LHC operator logbook after an instability occurred because the Landau damping was disabled

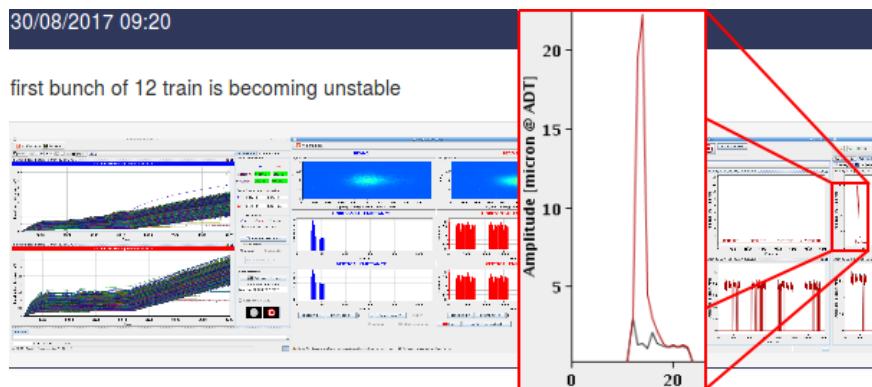


Figure 7.19: Instability shown in the ADT transverse activity monitor. The red line is the maximum oscillation amplitude achieved and the black line is the current amplitude which is a average from the last 4096 turns

Easy availability of the real-time transverse activity immediately triggered many questions about the performance of various subsystems and helped to explain some long observed features during the beam injection cycle.

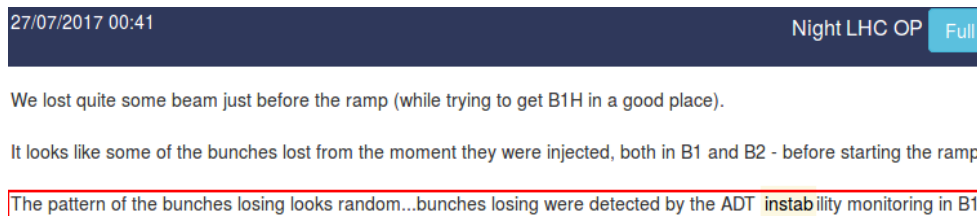


Figure 7.20: Multiple bunches were unstable and this was visible thanks to the fixed display

7.2.8 Real Life Example on How the System Helped Scientists at CERN

On the morning of the 3rd October 2017, the logbook showed an entry from the day before (02/10/2017 20:18) which contained a capture from the ADT Transverse Activity Monitor, the capture can be seen in Fig. 7.21. This showed a clear increase in transverse activity during fill 6266, when squeezing the beam. By using the TIMBER application it was clear that the system had triggered multiple times during this period and there were multiple stored HDF files thanks to ADTBufferSaver. By using the Python script which extracts data about the unstable bunches from TIMBER and by using some Bash scripting (see Listing 7.6), plots for the relevant period could be easily created. The result from this can be seen in Figures 7.22 to 7.25. This information was sent to relevant people in the ABP (*Accelerator and Beam Physics*) group at CERN so they could analyze the data stored by the ADTBufferSaver and figure out the cause of the instability. These findings were presented the following day (4th October) at the morning LHC operations meeting. The cause was quickly found and fixed for the next injection.

Listing 7.6: Plotting all unstable bunches from all planes during a specific time

```
for file in 6266/instability_data/06266_Inst_B*_19*; do
    python scripts/plotinstabilityfile.py -notch -f "$file" -max 10 ;
done
```

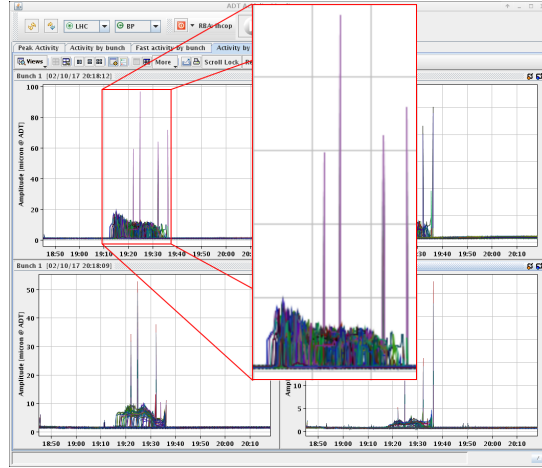


Figure 7.21: Screenshot of the ADT transverse Activity Monitor in a entry in the LHC OP log-book the 2nd October 2017 20:18 which shows increased bunch-by-bunch transverse activity between 19:10 and 19:40 the 2nd October

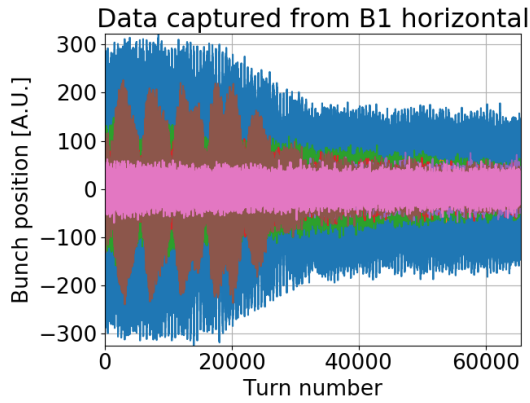


Figure 7.22: Bunches 1239, 1240, 1314, 1811, 2502 and 2922 were deemed unstable during fill 6266 by the new system at 19:22:25

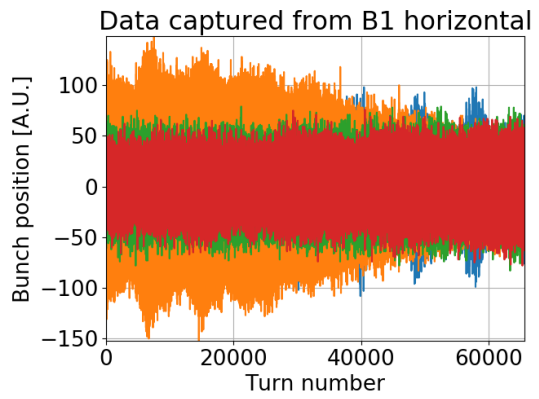


Figure 7.23: Bunches 917, 2032, 3204 and 3239 were deemed unstable by the new system at 19:24:05

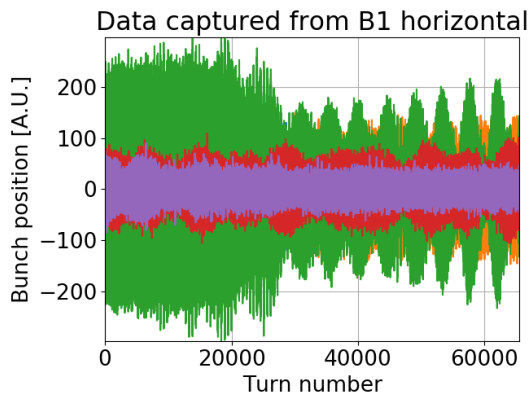


Figure 7.24: Bunches 522, 731, 735, 1679 and 2761 were deemed unstable by the new system at 19:28:10

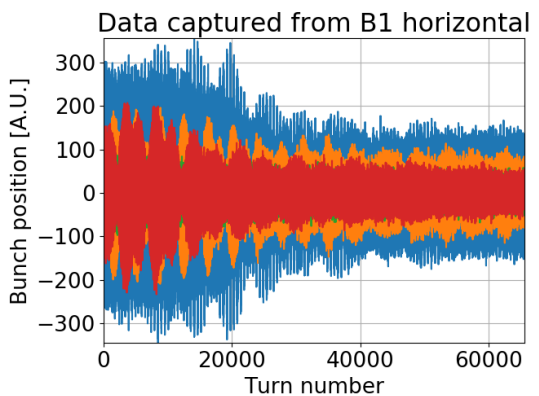


Figure 7.25: Bunches 699, 732, 2949 and 3356 were deemed unstable by the system at 19:29:21

There was a lot of interest regarding the instability during the squeeze in the LHC morning meeting and part of one slide can be seen in Fig. 7.26

ADT activity data stream (bunch-by-bunch) by M. Soderen and D. Valuch proved to be an essential diagnostics for this situation

Figure 7.26: Part of the slide from the LHC morning meeting 4th October

7.2.9 An Example of How the System Detects Instabilities

This section describes how the data is processed in the pipeline. The data in this example was captured during the instability that occurred during the squeeze of fill 6266 as already described in Sec. 7.2.8.

Figure 7.27 shows the raw data captured by the ADTBufferSaver FESA class after it received a trigger from the LIST network. The trigger was generated by the ADT instability detection system. There is, of course, a delay from the point that the 4096 turns buffer has been sent to the system to when an instability has been detected and the longer 65536 turn buffer has been frozen. This delay is however compensated by an offset in the instability buffer of 32768 turns which means that the ADTBufferSaver gets data which is 32768 turns old.

The data is first sent through a notch filter and the result can be seen in Fig. 7.28. The notch filter is there for closed orbit suppression, which means that it centers the data around 0. After the notch filter, the analytic signal is calculated and from this, the instantaneous oscillation amplitude can be calculated as described in Sec. 4.2 and shown in Fig. 7.29. The last step in the pipeline is to calculate the moving averages and compare consecutive values to detect any rapid amplitude increase. The moving averages together with the generated triggers can be seen in Fig. 7.30.

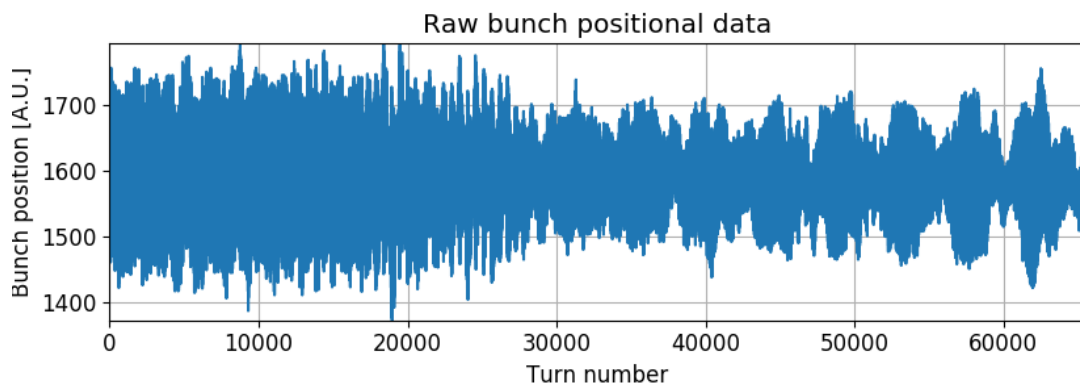


Figure 7.27: Raw positional data for bunch 735 during the squeeze of fill 6266 in the LHC

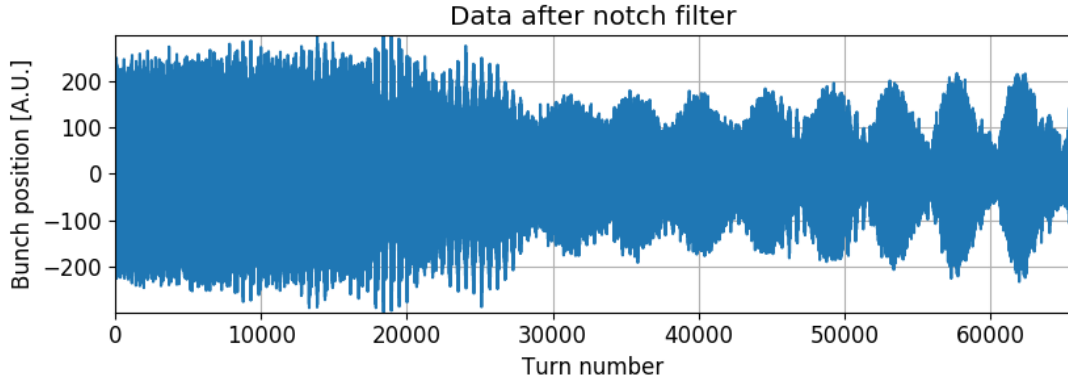


Figure 7.28: Data after notch filter

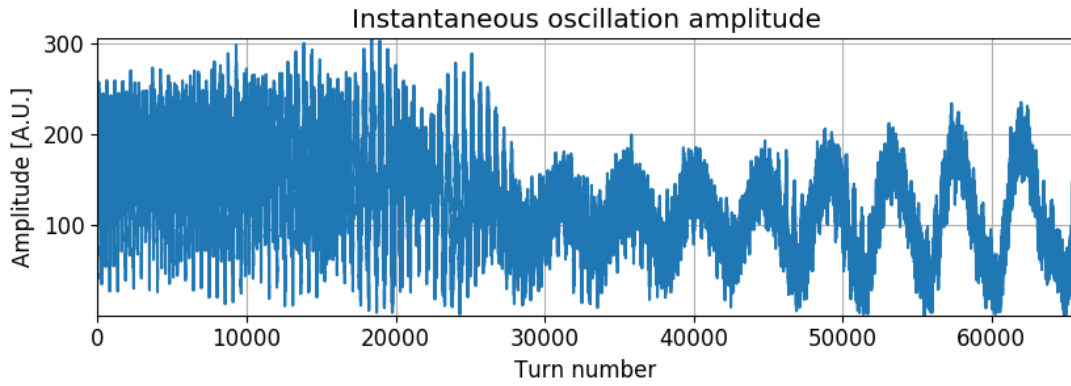


Figure 7.29: Instantaneous oscillation amplitude calculated using the Hilbert transform

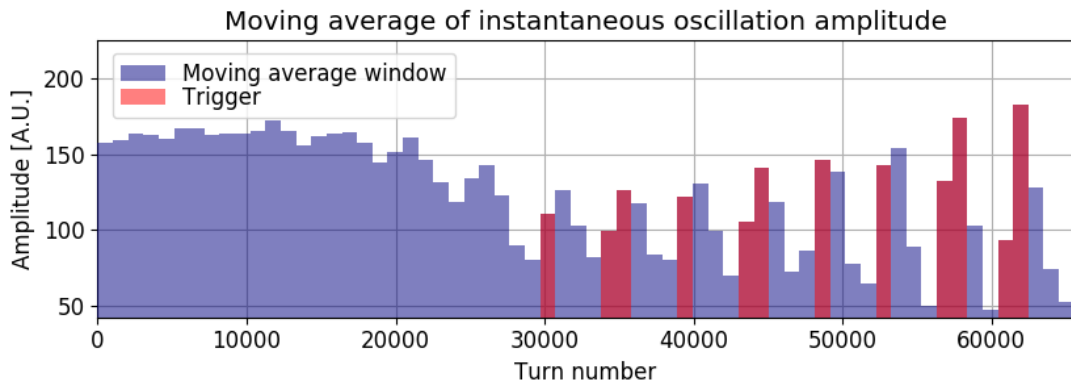


Figure 7.30: Moving average of the instantaneous amplitude and the generated triggers, the window length is 1024 and the trigger threshold was 1.4

7.3 Method Discussion

The architecture of the system was defined fairly early in the process for simplicity and practical reasons. Only the FESA framework was considered for implementation and only the pipeline pattern was considered for extracting algorithm level parallelism. It would have been interesting to test different implementations using different parallel programming paradigms, not only the pipeline pattern, but the time for this project was limited. It would

also have been interesting to test different instability detection algorithms, especially exponential curve fitting. The method itself was appropriate for the scope of this thesis, and future evaluation of online instability detection in high energy particle accelerators are left for future studies.



8 Related Work

There are multiple systems implemented for detecting transverse oscillation amplitude growth. For example the BBQ, Head-Tail monitor, MIM and soon the ADT transverse instability detection system. All of them have their advantages and disadvantages. The biggest challenge is sampling the signal from the pickup because the length of a bunch in the LHC is 1.5 ns which means that the sampling frequency must be in the order of 6-12 GHz [51]. At these speeds, the resolution is limited even with state-of-the-art technology. The different systems tackle this problem in different ways.

8.1 The LHC Head-Tail Monitor

The Head-Tail monitor was originally designed for machine parameter extraction but it can also be used for analyzing the beam stability [11][32]. It uses a stripline pickup just as the ADT and the signal are passed through a hybrid filter which generates a sum and difference signal just as in the ADT. This signal is however not passed to a BPM but to a very fast digitizer which samples the signal at 10 GSPS ($10 \cdot 10^9$ samples per second) with 8bit resolution. This high sampling rate means that it can see intra-bunch motion, meaning that it can see the shape of the bunch compared with the ADT that just sees the centre of charge for the whole bunch. The problem with this high sampling rate is the amount of data that is created. For now, it can only store 11 turns (1 ms) of data and this data takes 10 seconds to download from the digitizer. This means that a full-rate bunch-by-bunch extraction is not possible, but more modern digitizers are being tested that will allow for longer acquisitions. Figure 8.1 shows an overview of the system and Fig. 8.2 shows an instability that was detected during 2015.

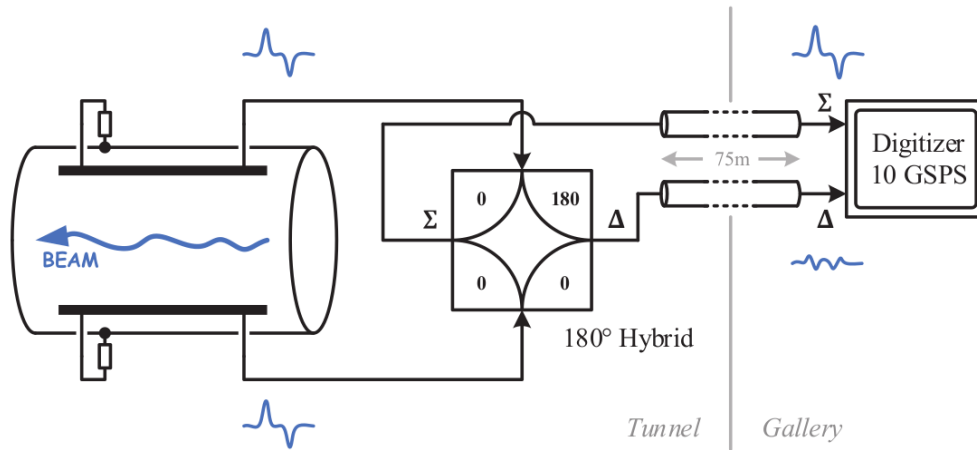


Figure 8.1: Overview of the head-tail monitor system (courtesy of CERN)

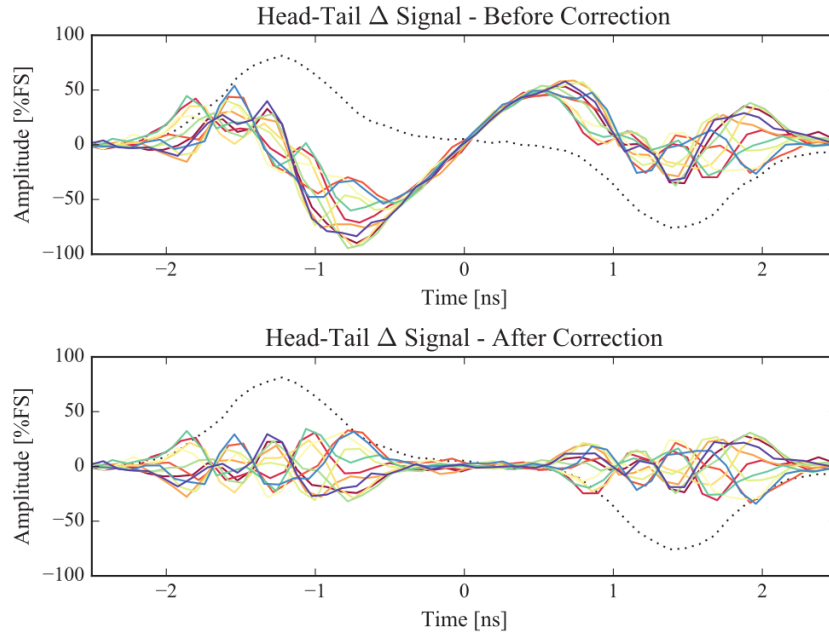


Figure 8.2: A mode 4 instability captured by the head-tail monitor (courtesy of CERN)

8.2 The LHC Base-Band Tune System (BBQ)

The BBQ is a diode peak detector which converts a high-frequency signal from a pickup to a low-frequency signal that can be sampled with high-resolution audio ADCs [18]. An overview of the system can be seen in Fig. 8.3. It is used for nonintrusive beam parameter extraction, meaning that it can extract beam parameters without exciting the beam.

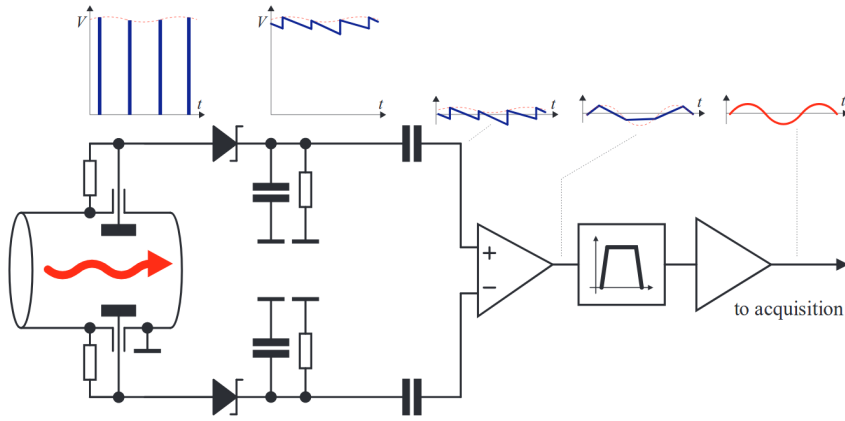


Figure 8.3: Overview of the LHC BBQ system (courtesy of CERN)

This system allows for extremely high turn-by-turn resolution up to 30nm compared to 1 μm in the ADT and 50 μm with a normal button pickup. It also has excellent signal-to-noise ratio but this comes at a price. The BBQ signal which is sampled is an average of all transverse oscillations of all bunches in the LHC. So it cannot detect transverse bunch-by-bunch instabilities. But it can detect that there is an ongoing instability on some bunch if that instability is large enough so it becomes the dominant component of the signal. There is ongoing research regarding instability detection using the BBQ which has shown great promise but all instability detection is FPGA based [32].

8.3 The Multiband-Instability-Monitor (MIM)

The MIM is a prototype being developed and tested at CERN, it is currently being tested in both SPS and LHC. The MIM tackles the required high resolution in another way than the BBQ. Just as with the ADT and the BBQ, it uses a stripline pickup but afterward the signal is split into several frequency bands using an RF filter bank and with narrower frequency bands the signal can be sampled with much higher resolution ADCs and also in parallel. However, with the current strip-line pickups which only have an effective bandwidth of up to 6GHz, the number of effective channels are 16 spaced by 400MHz. Better pick-ups are being developed to support up to 12GHz which would allow up to 32 channels. This system is, however, mainly designed to measure intra-bunch motion and it is still in its development phase [51].

8.4 Algorithms for Instability Detection

Detecting an outlier in a time series is not straightforward and there has been a lot of statistical research done in, for example, trend analysis to predict stock prices. The problem here, however, is fairly simple because during normal operation, the bunch transverse oscillation amplitude will be stable and the only thing that needs to be detected is an exponential growth increase. See Fig. 8.4, it shows real data for one bunch in the LHC which started to become unstable after injection.

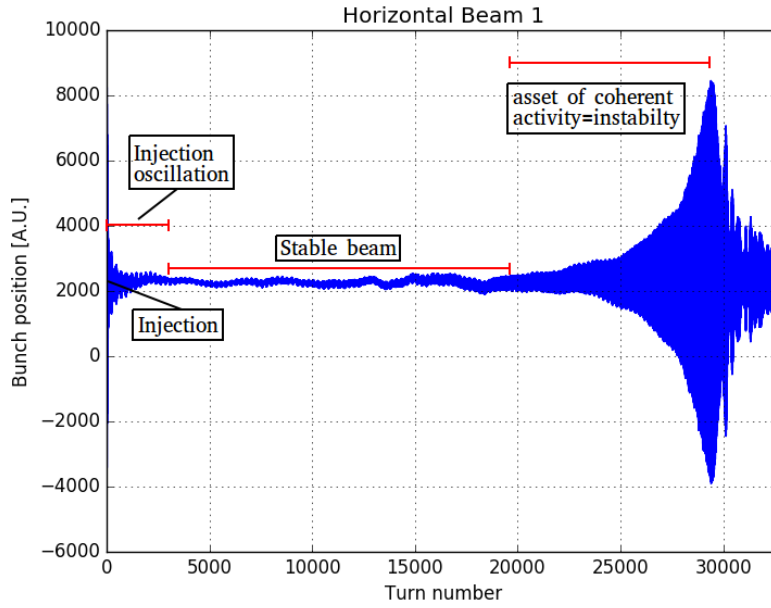


Figure 8.4: Transverse position of an unstable bunch in the LHC. The bunch was injected and the injection oscillation was quickly damped by the ADT but after 20000 turns in the LHC, (≈ 2 s) the amplitude of the transverse oscillation increased rapidly which lead to an unstable bunch.

8.4.1 Moving-Average

A simple and robust way of detecting an exponential amplitude growth in a time series is to calculate the moving average over a window with length W and compare that with the next window.

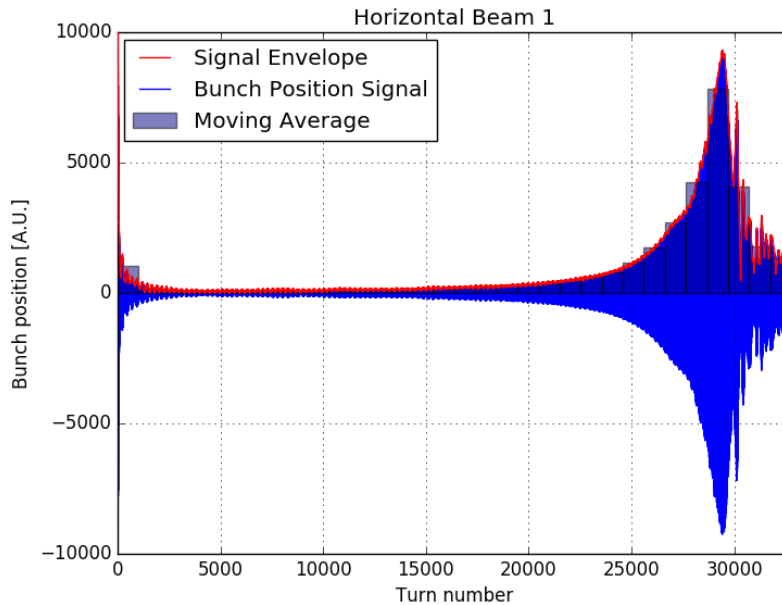


Figure 8.5: Moving average over windows with $W=1024$

Calculating the average for the m :th window is straight forward:

$$MA_m = \frac{1}{W} \sum_{i=mW}^{(m+1)W} A_i \quad (8.1)$$

Where A is from Eq. 4.3. And during normal operations it is assumed that:

$$MA_m \approx MA_{m+1} \quad (8.2)$$

And during an exponential amplitude growth is it assumed that:

$$MA_m \ll MA_{m+1} \quad (8.3)$$

To adjust the threshold for a trigger:

$$MA_m \cdot T < MA_{m+1} \quad (8.4)$$

If $T = 1.5$ is used, this requires the next window average to be 50% higher than the previous window to send a trigger. To be able to detect instabilities with a variety of rise times, multiple moving averages with different window length can be combined. To reduce the number of calculations needed, the different averages can be stages in a pipeline where the average from the first window is passed to the second stage.

8.4.2 Three-Averages Algorithm

The three-average algorithm has been tested at CERN both in the MIM and the BBQ. It computes the average of the standard deviation about the mean of the signal over three different time windows. The different window lengths are arranged such that $W_{short} < W_{med} < W_{long}$. From these windows σ_{short} , σ_{med} and σ_{long} are calculated and it is assumed that during normal beam $\sigma_{short} \approx \sigma_{med} \approx \sigma_{long}$ holds. During an exponential amplitude growth, the average of a longer window will grow slower than the one of the short windows. The following inequalities should hold during an instability:

$$\begin{aligned} \sigma_{short} - \alpha \sigma_{med} &> 0 \\ \sigma_{med} - \beta \sigma_{long} &> 0 \end{aligned} \quad (8.5)$$

Where $\alpha, \beta > 0$ are coefficients chosen to set the trigger threshold. For a higher level of confidence the numbers of turns in the machine where the inequalities in Eq. 8.5 are true are counted and when the counter reaches a threshold it sends a trigger. This was implemented on FPGAs in both the MIM and the BBQ. During tests, it was discovered that this algorithm does not perform well when the rise time is slow since the averages follow the slow increase and the difference is never big enough to generate a trigger [32].

8.4.3 Increase-Subsequence Algorithm

The Increase-Subsequence algorithm has also been tested in the BBQ and the MIM. It keeps track of the latest W samples and every time a new sample is acquired it checks if the new sample is larger than the maximum value from the subset. If that is the case, a counter is incremented or if the oldest value in the subset is the largest, the counter is decremented. When the counter reaches a certain threshold a trigger is generated.

During tests, it was discovered that it handles long rise times better compared to the Three-Averages algorithm. However, when there are many bunches in the LHC the amplitude change is less prominent and the Three-Averages performs better since it is more sensitive [32].

8.4.4 Exponential Curve Fitting Using the Least Square Method

After low-pass filtering the signal to remove high-frequency noise, the signal can be approximated to fit the functional form:

$$A[n] = e^a e^{bn} \quad (8.6)$$

Where $A[n]$ is the amplitude from Eq. 4.3 and n is the turn-number. The least-square fitting can be found using (with A_n instead of $A[n]$ and over W samples):

$$a = \frac{\sum_{n=1}^W (n^2 A_n) \sum_{n=1}^W (A_n \ln(A_n)) - \sum_{n=1}^W (n A_n) \sum_{n=1}^W (n A_n \ln(A_n))}{\sum_{n=1}^W A_n \sum_{n=1}^W (n^2 A_n) - (\sum_{n=1}^W n A_n)^2} \quad (8.7)$$

$$b = \frac{\sum_{n=1}^W A_n \sum_{n=1}^W (n A_n \ln(A_n)) - \sum_{n=1}^W (n A_n) \sum_{n=1}^W (A_n \ln(A_n))}{\sum_{n=1}^W A_n \sum_{n=1}^W (n^2 A_n) - (\sum_{n=1}^W n A_n)^2} \quad (8.8)$$

Where a is proportional to the linear growth in the amplitude and b is proportional to the exponential growth. During normal operation it is assumed that $b \approx 0$ and during an instability with exponential amplitude growth $b > 0$. This is interesting since the rise time of the instability is automatically acquired [53].

Listing 8.1: Exponential Curve Fitting Using the Least Square Method implemented using Intel intrinsics

```
float sum(const __m256& x1) {
    __m256 temp= _mm256_hadd_ps(x1,x1);
    temp=_mm256_hadd_ps(temp,temp);
    return temp[0]+temp[4];
}
__m256 x;
__m256 y;
float Y=0.0, XY=0.0, X2Y=0.0, YLNY=0.0, XYLNY=0.0;

Y+=sum(y);
__m256 XYv=_mm256_mul_ps(x,y);
XY+=sum(XYv);
X2Y+=sum(_mm256_mul_ps(XYv,x));
__m256 LNYv= _mm256_clog_ps(y);
XYLNY+=sum(_mm256_mul_ps(XYv,LNYv));
YLNY+=sum(_mm256_mul_ps(y,LNYv));
float temp=(Y*X2Y-pow(XY,2.0));
float a=(X2Y*YLNY-XY*XYLNY)/temp;
float b=(Y*XYLNY-XY*YLNY)/temp;
```

In this example, the number of samples was only 8 but this can easily be extended to any number of samples. It is important to be able to detect instabilities of different rise times and the range of instabilities that can be detected depends on the number of samples used for the curve fitting. Fig. 8.6 shows an example of how this can be implemented. The instantaneous amplitude is calculated and passed to one block which does a curve fitting on 128 samples and then decimates the signal by a factor 4. The signal is then passed through a chain of identical blocks where the last block receives a signal which has been downsampled by a factor of 1024. If this was applied to the bunch-by-bunch transverse positional data in the ObsBox which has a sampling frequency of 11245 Hz, then the last block would do a curve fitting on ≈ 11.65 s of downsampled data.

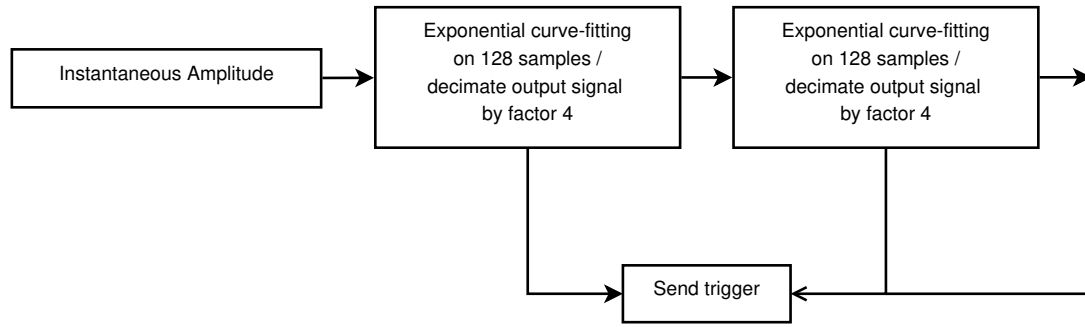


Figure 8.6: Instability detection pipeline using exponential curve fitting

This can be done without downsampling but that would require the server to keep $\approx 1\text{GB}$ of data in memory and every time a new sample is received a new curve-fitting on 131072 samples would have to be done for every bunch which is quite computationally intensive. This is interesting since we are looking for exponential amplitude growth and this would give the rise-time of the instability directly, it could also be more error proof than the current algorithm. No tests were done with this algorithm because of time-constraints.



9 Conclusion

A real-time transverse instability detection system for the LHC which used the bunch-by-bunch transverse positional data available from the ADT has been developed which sends out a trigger through the LIST network when an instability is detected. This trigger is used by observation equipment around the LHC to freeze their buffers for later analysis. The system also allows to detect which bunches in the LHC are becoming unstable. This data is logged in the LHC logging system which can later be used to filter the data that all observation equipment generated. The system also calculates the transverse oscillation amplitude in real-time for each bunch in the machine which is both logged and displayed on a fixed display in the CCC.

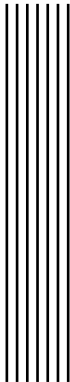
The new system has been tested so it can handle the throughput required. To verify that the system could detect instabilities, simulated data, stored data, and data from real beams have been used. The system has so far proved helpful and will continue to deliver information about the transverse activity in the LHC which can be used for further research in beam dynamics. It showed that the FESA framework can be used for online analysis of high bandwidth data stream. It presented some results on how different compilers can generate executables with different performance even though this is nothing new but it also showed how manual optimizations can limit the compiler to really optimize the generated binaries. When looking back at the aim of this project, all that was required to be accomplished has been achieved. The first system capable of detecting bunch-by-bunch transverse instability in real-time has been developed. This could be implemented on a FPGA or a GPU but the goal of this project was to implement it using the available hardware.

9.1 Relevance

The results are not only relevant for high energy particle accelerators research but also in high-performance computing. This system is not only limited to being used at CERN but at other research institutes around the world with particle accelerators. It has already proven itself useful for scientists and operators at CERN. This can lead to similar systems being developed at other particle physics laboratories around the world.

9.2 Future Work

There are many possibilities for future work. There should be more research in more advanced algorithms for instability detection, this could be an interesting application for machine learning. There have already been discussions here at CERN to create a new system that would not rely on the ObsBox system or the FESA framework but a system dedicated for online analysis. This could resolve in a low latency instability detection system which relies on more modern compilers, software, and hardware. The online analysis is not limited to instability detection, it could also be bunch-by-bunch tune extraction, coupling analysis, singular-value decomposition for head-tail motion extraction if the resolution of the BMPs is increased. There is a lot of interesting information that could be extracted from the bunch-by-bunch positional data available in the ADT.



Bibliography

- [1] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1990. (Cited on page 20.)
- [2] G. Arduini, K. Cornelis, W. Hofle, G. Rumolo, and F. Zimmermann. The electron cloud instability of the LHC beam in the CERN SPS. In *LHC-Project-Report-637. CERN-LHC-Project-Report-637*, 2003. (Cited on page 10.)
- [3] V. Balbekov. Single bunch transverse instability in a circular accelerator with chromaticity and space charge. *Journal of Instrumentation*, 10(10), 2015. (Cited on page 10.)
- [4] X. Buffat. Transverse beams stability studies at the Large Hadron Collider, 2015. (Cited on page 10.)
- [5] F. Caspers, A. Goldblatt, A. Nosych, F. Roncarolo, G. Trad, C. Völlinger, and M. Wendt. The LHC Synchrotron Light Monitor (BSRT). (CERN-ACC-SLIDES-2014-0049), Apr 2014. (Cited on page 10.)
- [6] CERN. CERN Data Centre. <http://information-technology.web.cern.ch/about/computer-centre>. [Online; Accessed: 2017-09-01]. (Cited on page 29.)
- [7] CERN. Experiments. <https://home.cern/about/experiments>. [Online; Accessed: 2017-08-10]. (Cited on page 18.)
- [8] CERN. The accelerator complex. <https://home.cern/about/accelerators>. [Online; Accessed: 2017-08-10]. (Cited on pages 16 and 17.)
- [9] CERN. Worldwide LHC Computing Grid. <http://wlcg.web.cern.ch/>. [Online; Accessed: 2017-08-13]. (Cited on page 18.)
- [10] A. Chao and R. Ruth. Coherent beam-beam instability in colliding beam storage rings. *Part. Accel.*, 16(SLAC-AP-37. SLAC-PUB-3400):201–216. 27 p, Aug 1984. (Cited on page 10.)
- [11] D. Cocq, R. Jones, and H. Schmickler. The Measurement of Chromaticity via a Head-Tail Phase Shift. (CERN-SL-98-062-BI):9, Nov 1998. (Cited on page 68.)
- [12] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991. (Cited on page 26.)

- [13] Eclipse. Eclipse. <https://eclipse.org/ide/>. [Online; Accessed: 2017-09-11]. (Cited on page 29.)
- [14] A. Ernstsson. SkePU 2: Language embedding and compiler support for flexible and type-safe skeleton programming. Master's thesis, 2016. (Cited on page 26.)
- [15] FermiLab. Scientific Linux. <https://www.scientificlinux.org/>. [Online; Accessed: 2017-09-11]. (Cited on page 14.)
- [16] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 19, 2008. (Cited on page 23.)
- [17] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972. (Cited on page 18.)
- [18] M. Gasior. FARADAY CUP AWARD: High Sensitivity Tune Measurement using Direct Diode Detection. *Conf. Proc.*, C1204151(CERN-ATS-2012-246):7, Apr 2012. (Cited on page 69.)
- [19] F. Gebali. *Algorithms and parallel computing*. John Wiley & Sons, 2011. (Cited on pages 19 and 25.)
- [20] V. Gligorov. Real-time data analysis at the lhc: present and future. 2015. (Cited on page 4.)
- [21] GNU. Data Display Logger. <https://www.gnu.org/software/ddd/>. [Online; Accessed: 2017-09-20]. (Cited on page 27.)
- [22] GNU. GDB: The GNU Project Debugger. <https://www.gnu.org/software/gdb/>. [Online; Accessed: 2017-09-20]. (Cited on page 27.)
- [23] W. Herr and B. Muratori. Concept of luminosity. 2006. (Cited on page 5.)
- [24] T. Hoffman. Fesa - the front-end software architecture. In *Proceedings of ICALEPCS2003*, 2008. (Cited on pages 29 and 30.)
- [25] W. Hofle, G. Kotzian, M. Schokker, and D. Valuch. LHC damper beam commissioning in 2010. In *2nd International Particle Accelerator Conference*, 2011. (Cited on page 12.)
- [26] C. Holt, J. Singh, and J. Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. 1996. (Cited on page 19.)
- [27] B. Holzer. Introduction to particle accelerators and their limitations. 2017. (Cited on page 5.)
- [28] Intel. Getting Started with Intel® VTune™ Amplifier 2018. <https://software.intel.com/en-us/get-started-with-vtune>. [Online; Accessed: 2017-09-20]. (Cited on page 27.)
- [29] Intel. Intel intrinsics guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>, note = [Online; Accessed: 2017-11-23]. (Cited on page 45.)
- [30] Iperf. iperf. <https://iperf.fr/>. [Online; Accessed: 2017-09-13]. (Cited on page 39.)
- [31] G. Kotzian. Transverse feedback parameter extraction from excitation data. In *8th International Particle Accelerator Conference*, 2017. (Cited on page 16.)

- [32] T. Levens, K. Lasocha, and T. Lefevre. Recent developments for instability monitoring at the LHC. In *Proceedings of IBIC2016*, 2016. (Cited on pages 68, 70, and 72.)
- [33] Y. Liu. Hilbert transform and applications. InTech, 2012. (Cited on page 34.)
- [34] S. Maleki, Y. Gao, M. Garzarán, T. Wong, and D. Padua. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 28.)
- [35] Mathworks. Getting Started with Filter Designer. <https://ch.mathworks.com/help/signal/ug/getting-started-with-filter-designer.html>. [Online; Accessed: 2017-09-12]. (Cited on page 35.)
- [36] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. (Cited on pages 20, 22, 23, and 25.)
- [37] B. Nichols, D. Buttler, and J. Farrell. *Pthreads Programming. A POSIX Standard For Better Multiprocessing*. O'Reilly Media, 1998. (Cited on page 24.)
- [38] M. Nicolas. *The LHC Transverse Coupled-Bunch Instability*. PhD thesis, 2012. (Cited on page 10.)
- [39] M. Ojeda, P. Baudrenghien, and A. Butterworth. Processing high-bandwidth bunch-per-bunch observation data from the RF and transverse damper system of the LHC. In *Proceedings of ICALEPCS2015*, 2015. (Cited on pages 12 and 14.)
- [40] Open Hardware Group. fmc-dio-5chttla FMC 5-channel Digital I/O module. <https://www.ohwr.org/projects/fmc-dio-5chttla/wiki>. [Online; Accessed: 2017-08-16]. (Cited on page 14.)
- [41] Open Hardware Group. Simple PCIe FMC carrier (SPEC). <https://www.ohwr.org/projects/spec/wiki>. [Online; Accessed: 2017-08-16]. (Cited on page 14.)
- [42] Open Hardware Group. Zio – the Ultimate I/O framework. <https://www.ohwr.org/projects/zio/wiki>. [Online; Accessed: 2017-08-17]. (Cited on page 15.)
- [43] Open MPI. Open Source High Performance Computing. <https://www.open-mpi.org/>. [Online; Accessed: 2017-08-13]. (Cited on pages 19 and 25.)
- [44] A. Oppenheim, W. Ronald, and R. John. Discrete hilbert transforms. In *Discrete-Time Signal Processing*. Pearson, 1989. (Cited on page 34.)
- [45] T. Pettersson and P. Lefèvre. The Large Hadron Collider: conceptual design. Technical report, 1995. (Cited on pages 6 and 7.)
- [46] C. Roderick, R. Billen, R. Aparicio, E. Grancher, A. Khodabandeh, and N. Chinchilla. The LHC Logging Service : Handling terabytes of on-line data. In *Proceedings of ICALEPCS2009*, 2009. (Cited on page 32.)
- [47] D. Romero and G. Dolecek. Digital fir hilbert transformers: Fundamentals and efficient design methods. InTech, 2012. (Cited on page 34.)
- [48] T. Saidani, C. Tadonki, L. Lacassagne, J. Falcou, and D. Etiemble. Algorithmic Skeletons within an Embedded Domain Specific Language for the CELL Processor. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 67–76, Sep 2009. (Cited on page 26.)

- [49] F. Soubelet. *Towards an LHC octupole optics model for PyHEADTAIL*. PhD thesis, 2017. (Cited on pages 7 and 8.)
- [50] R. Steinhagen. Tune and chromaticity diagnostics. 2009. (Cited on pages 7, 9, and 35.)
- [51] R. Steinhagen, M. Boland, and T. Lucas. A multiband-instability-monitor for high-frequency intra-bunch beam diagnostics. 2013. (Cited on pages 68 and 70.)
- [52] M. Steuwer and S. Gorlatch. *SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems*, pages 258–272. 2013. (Cited on page 26.)
- [53] T. Strutz. *A practical introduction to weighted least squares and beyond*. Springer Vieweg, 2016. (Cited on page 73.)
- [54] M. Söderén, G. Kotzian, M. Ojeda Sandońs, and D. Valuch. Online bunch by bunch transverse instability detection in lh. In *Proceedings, 8th International Particle Accelerator Conference (IPAC2017)*, 2017. (Cited on page 38.)
- [55] Valgrind. Callgrind: a call-graph generating cache and branch prediction profiler. <http://valgrind.org/docs/manual/cl-manual.html>. [Online; Accessed: 2017-09-20]. (Cited on page 27.)
- [56] Valgrind. Valgrind. <http://valgrind.org/>. [Online; Accessed: 2017-09-20]. (Cited on page 27.)
- [57] C. Wang, Y. Liu, and M. Spear. Transaction-friendly condition variables. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 198–207, 2014. (Cited on page 21.)
- [58] K. Wille. *The physics of particle accelerators*. Clarendon Press; 1 edition, 2000. (Cited on page 8.)
- [59] E. Wilson. *An introduction to particle accelerators*. Oxford University Press, 2001. (Cited on page 1.)
- [60] T. Włostowski, G. Daniluk, M. Lipinski, J. Serrano, and F. Vaga. Trigger and RF Distribution Using White Rabbit. 2015. (Cited on pages 31 and 32.)

A Real Time Action

Listing A.1: The real-time action which is triggered from a subscription to ObsBoxBuffer

```
//Constructor for Real-time event
OnMultiThreadedCES::OnMultiThreadedCES(fesa::RTActionConfig& rtActionConfig,
const fesa::AbstractServiceLocator* serviceLocator,
const std::map<std::string, const fesa::AbstractServiceLocator*>&
serviceLocatorRelatedClasses) :OnMultiThreadedCESBase(rtActionConfig,
serviceLocator, serviceLocatorRelatedClasses)
{
    std::size_t turns = 0;
    std::size_t window1 = 0;
    std::size_t window2 = 0;
    std::size_t window3 = 0;
    //For every device but there is only one on each machine
    for (auto it = ALLADTCopraServiceLocator_>getDeviceCollection().begin();
        it != ALLADTCopraServiceLocator_>getDeviceCollection().end(); it++)
    {
        //Get configuration from device
        turns = (std::size_t) (*it)->Turns.get();
        window1 = (std::size_t) (*it)->Window1Size.get();
        window2 = (std::size_t) (*it)->Window2Size.get();
        window3 = (std::size_t) (*it)->Window3Size.get();
    }
    //Init Unstable structs to keep track of instabilities
    instabilities = (Unstable**) malloc(3564 * sizeof(Unstable*));
    for (std::size_t i = 0; i < 3564; i++)
    {
        *(instabilities + i) = (Unstable*) malloc(sizeof(Unstable));
        (*(instabilities + i))->start = false;
        (*(instabilities + i))->bunch = (unsigned short) i;
        (*(instabilities + i))->turn = 0;
        (*(instabilities + i))->turnEnd = 0;
    }
}
```

```
(*(instabilities + i))->counter = 0;
*(instabilities + i))->unstable = false;
*(instabilities + i))->place = 0;
}
//Init the status struct which is the communication
//between the real-time action and the pipeline
status = new Status();
//Create a pipeline
pipeline = new Pipeline(status, turns, window1, window2, window3);
textCounter = 0;
simulatedAmplitude = 0.0f;
time = 0.0;
counter = 0;
}

bool compareUnstable(const Unstable* a, const Unstable* b)
{
    return a->turn < b->turn;
}

void OnMultiThreadedCES::execute(fesa::RTEvent* pEvt)
{
    //extract payload
    fesa::MultiplexingContext* context = pEvt->getMultiplexingContext();
    const OnSubscriptionRTEEventPayload* payload =
    dynamic_cast<const OnSubscriptionRTEEventPayload*>(pEvt->getPayload().get());
    //This is a hack, the payload devicename
    //contains both devicename and buffername
    std::string buffername = payload->getDeviceName()
    .substr(0, payload->getDeviceName().find(":"));
    std::string devicename = payload->getDeviceName()
    .substr(payload->getDeviceName().find(":") + 1,
    payload->getDeviceName().size() - 1);
    //To display a human readable text which describes which bunches
    //are unstable in the FESA interface
    std::vector<std::string> unstabletext;
    //get device from servicelocator
    Device* device = ALLADTCopraServiceLocator_->getDevice(devicename);
    //Reset the text in the interface
    device->UnstableText.set("", context);
    //Variables for getting the data from the payload
    const int16_t* array_pointer = NULL;
    std::size_t columns = 0;
    std::size_t rows = 0;
    std::size_t triggerstamp = 0;
    bool triggered=false;
    //If there are any changes in settings from the interface
    if (device->ChangesAvailable.get(context))
    {
        //Change the threshold settings
        fesa::ImmutableArray<float> thresholdfilters =
        device->ThresholdFilter.get(context);
```

```

fesa::ImmutableArray<float> thresholdpercentage =
device->ThresholdPercentage.get(context);
//Change the number of turns we ignore a instability after
//the data was zero to prevent triggering on
//injection oscillation
int64_t injection_oscillation =
device->InjectionOscillationTurns.get(context);
(status->prevent_injection) = injection_oscillation;
for (std::size_t i = 0; i < thresholdfilters.size(); i++)
{
    status->SetThresholdFilter(i, thresholdfilters[i]);
}
for (std::size_t i = 0; i < thresholdpercentage.size(); i++)
{
    status->SetThresHoldPercentage(i, thresholdpercentage[i]);
}
//In the interface you can analyze every stage in the pipeline
//for a specific bunch, which bunch is set here
status->bunch_to_analyze = device->bunch_to_analyze.get(context);
//Reset changes available
device->ChangesAvailable.set(false, context);
//If we got a reset from the interface, clear everything
if (device->Reset.get(context))
{
    for (std::size_t i = 0; i < 3564; i++)
    {
        (*(instabilities + i))->start = false;
        (*(instabilities + i))->turn = 0;
        (*(instabilities + i))->turnEnd = 0;
        (*(instabilities + i))->counter = 0;
        (*(instabilities + i))->unstable = false;
        (*(instabilities + i))->place = 0;
        device->Unstable.setCell(false, i, context);
        device->triggeredWindow.setCell(0, i, context);
    }
    device->Reset.set(false, context);
}
//Update coefficients for the hilbert transform
//these are floats so they can't be atomic
//So I used a lock to protect the data in the
//status struct
uint32_t size=0;
const float* hilbert= device->Hilbert.get(size,context);
status->mtx->lock();
memcpy(status->hilbert,hilbert,size*sizeof(float));
status->scaling=device->ScalingTransverseActivityMonitor.get(context);
status->mtx->unlock();
status->new_hilbert=true;
}
//We can get all bunches or a subset of them
if (payload->getPropertyName() == "Acquisition")
{
    std::auto_ptr<const ObsBoxBuffer::AcquisitionPropertyData> data

```

```
= OnSubscriptionRTEEventPayload::extract<
ObsBoxBuffer::AcquisitionPropertyData>(*pEvt);
try
{
    //Extract data
    fesa::ImmutableArray2D<int16_t> array = data->getData();
    device->triggerStamp.set(data->triggerStamp.get(), context);
    triggerstamp = data->triggerStamp.get();
    columns = array.getNumberOfColumns();
    rows = array.getNumberOfRows();
    array_pointer = array;
}
catch (const fesa::FesaException& exception)
{
    LOG_ERROR_IF(logger, exception.what());
}

}
else
{
    std::auto_ptr<const ObsBoxBuffer::AcquisitionSubsetPropertyData> data
    = OnSubscriptionRTEEventPayload::extract<
ObsBoxBuffer::AcquisitionSubsetPropertyData>(*pEvt);
    try
    {
        //Extract data
        fesa::ImmutableArray2D<int16_t> array = data->getData();
        device->triggerStamp.set(data->triggerStamp.get(), context);
        triggerstamp = data->triggerStamp.get();
        columns = array.getNumberOfColumns();
        rows = array.getNumberOfRows();
        array_pointer = array;
    }
    catch (const fesa::FesaException& exception)
    {
        LOG_ERROR_IF(logger, exception.what());
    }
}
//If something went wrong throw a exception
if (array_pointer == NULL || columns == 0 || rows == 0)
{
    throw std::runtime_error("something_went_wrong_during
dataextraction_in_OnMultiThrededCES::execute");
}
fesa::NoneContext noContext;
//Create a set with all bunches
std::string bunches = device->bunches_filter.getAsString(&noContext);
std::set<unsigned int> bunches_set = SubsetSelection::parse(bunches, 0, 3563);
//Copy the Averaged amplitude from the status to the interface
//this is displayed in the CCC
status->mtx->lock();
device->transverseActivityMonitor.set(status->transverseActivityMonitor, 3564,
context);
```

```

status->mtx->unlock();
//Pipeline analyzer is atomic ints to this does not need a lock
for (std::size_t i = 0; i < 10; i++)
{
    device->pipeline_analyzer.setCell(status->pipeline_analyzer[i], i, context);
}
//When a instability is detected it is visible in the interface
//until the next real-time action is triggered, the
//instabilities_delay queue contains instabilities
//detected last time
while (!instabilities_delay.empty())
{
    unsigned temp = instabilities_delay.front();
    instabilities_delay.pop();
    unstable* tempIn = *(instabilities + temp);
    device->Unstable.setCell(false, temp, context);
    std::ostringstream tempString;
    tempString << "Bunch_" << tempIn->bunch << "_Start:"
    << tempIn->turn << "_End:" << tempIn->turnEnd;
    unstabletext.push_back(tempString.str());
    LOG_INFO_IF(logger, tempString.str());
    device->triggeredWindow.setCell(0, tempIn->bunch, context);
}
//Extract new instabilities from the pipeline
if (!status->winlqueue->Empty())
{
    counter = 0;
    std::vector<Unstable*> tempInstabilities = status->winlqueue->TakeAll();
    for (auto it = tempInstabilities.begin(); it != tempInstabilities.end(); it++)
    {
        Unstable* tempIn = *(instabilities + (*it)->bunch);
        instabilities_delay.push((*it)->bunch);
        tempIn->turn = (*it)->turn;
        tempIn->turnEnd = (*it)->turnEnd;
        tempIn->counter = 1;
        device->Unstable.setCell(true, (*it)->bunch, context);
        std::ostringstream temp;
        temp << "Bunch_" << tempIn->bunch << "_Start:" << tempIn->turn;
        unstabletext.push_back(temp.str());
        LOG_INFO_IF(logger, temp.str());
        //We only send on trigger per real-time action
        if (device->Trigger.get(context) &&!triggered)
        {
            ProxyInterface proxyinterface = ProxyInterface();
            CGWRTD_IN::TrigAtTimePropertyData* triggerControl =
            new CGWRTD_IN::TrigAtTimePropertyData();
            triggerControl->setAtTimePicoSeconds(0);
            triggerControl->setAtTimeSeconds(0);
            triggerControl->setSequenceNumber(0);
            proxyinterface.setProperty(device->TriggerName.getAsString(context),
            "LHC.USER.ALL", *triggerControl);
            triggered=true;
        }
    }
}

```

```
uint32_t size;
const int8_t* triggeredWindows = device->triggeredWindow.get(size, context);
int8_t place = 0;
//This is for detecting which windows detected a instability
//can be more than one
if (tempIn->place == 0)
{
    place = 1;
}
else if (tempIn->place == 1)
{
    place = 2;
}
else
{
    place = 4;
}
device->triggeredWindow.setCell(*(triggeredWindows + tempIn->bunch) | place, t
free(*it);
}
}
else
{
    counter++;
}
//This is a failsafe, if we have not detected a instability
//the last 100 actions we reset everything
if (counter > 100)
{
    counter = 0;
    for (std::size_t i = 0; i < 3564; i++)
    {
        device->Unstable.setCell(false, i, context);
        device->triggeredWindow.setCell(0, i, context);
        (*(instabilities + i))->start = false;
        (*(instabilities + i))->turn = 0;
        (*(instabilities + i))->turnEnd = 0;
        (*(instabilities + i))->counter = 0;
        (*(instabilities + i))->unstable = false;
        (*(instabilities + i))->place = 0;
    }
}
//Create a string which describes which bunches were unstable
std::string tempstring = "";
for (auto it = unstabletext.begin(); it != unstabletext.end(); it++)
{
    if (tempstring.size() + (*it).size() < 254)
    {
        tempstring += *it;
        tempstring += ",";
    }
}
device->UnstableText.set(tempstring, context);
```

```

//A instability can be simulated through the interface
//This generates a sinusoid with increasing amplitude
if (device->simulate.get(context))
{
    int16_t* temp_data = (int16_t*) malloc(rows * columns * sizeof(int16_t));
    simulatedAmplitude += device->SimulatedAmplitudeIncrease.get(context);
    if (simulatedAmplitude < 0.0)
    {
        simulatedAmplitude = 0.0;
    }
    if (simulatedAmplitude > 10000.0)
    {
        simulatedAmplitude = 10000.0;
    }
    for (std::size_t i = 0; i < rows; i++)
    {
        float f_t = simulatedAmplitude * sin(2.0f * M_PI * 0.32f * 11000.0f * time);
        time += 1.0 / 11000.0;
        int16_t value = (int16_t) (f_t);
        for (std::size_t j = 0; j < columns; j++)
        {
            *(temp_data + i * columns + j) = value;
        }
    }
    //Push the simulated data to the pipeline
    pipeline->PushData(temp_data, rows, columns, triggerstamp, bunches_set);
    free(temp_data);
}
else
{
    //push the real data to the pipeline
    pipeline->PushData(array_pointer, rows, columns, triggerstamp, bunches_set);
}
}

```

B Constructing the Pipeline

Listing B.1: This is the constructor of the pipeline which shows how the stages is created

```
Pipeline::Pipeline(Status* input_status, std::size_t queue, std::size_t window1
, std::size_t window2, std::size_t window3)
{
    //used to send/receive data to and from the real-time action
    status = input_status;
    //Queue from pushdata to injection prevention stage
    st12 = new BlockingQueue<QueueElement*>();
    st12->SetCapacity(2 * queue);
    //queue from injection prevention stage to notch stage
    st23 = new BlockingQueue<QueueElement*>();
    st23->SetCapacity(2 * queue);
    //queue from notch stage to Hilbert stage
    st34 = new BlockingQueue<QueueElement*>();
    st34->SetCapacity(2 * queue);
    //queue from hilbert stage to amplitude stage
    st45 = new BlockingQueue<QueueElement*>();
    st45->SetCapacity(2 * queue);
    //queue from amplitude stage to first instability detection stage
    st56 = new BlockingQueue<QueueElement*>();
    st56->SetCapacity(2 * queue);
    //queue from first instability detection stage to second stage
    st67 = new BlockingQueue<QueueElement*>();
    st67->SetCapacity(2 * queue);
    //queue from second instability stage to third stage
    st78 = new BlockingQueue<QueueElement*>();
    st78->SetCapacity(2 * queue);
    //queue from third instability stage to transverseactivity stage
    st89 = new BlockingQueue<QueueElement*>();
    st89->SetCapacity(2 * queue);
    //queue which is used for reusing old QueueElements
```

```

reuse = new BlockingQueue<QueueElement*>();
reuse->SetCapacity(queue * 2);
//only used in pushdata, dont use anywhere else
temp_16_floats = (float*) _mm_malloc(16 * sizeof(float), 16);
//only used in pushdata, dont use anywhere else
temp_16_short = (short*) _mm_malloc(16 * sizeof(short), 16);
//injection prevention after the pushdata queue
threads.push_back(new boost::thread(boost::bind(&injectionOscillationPrevention
, st12, input_status, st23)));
//notch after the injection oscillation triggering prevention queue
threads.push_back(new boost::thread(boost::bind(&stage2, st23,
input_status, st34)));
//Hilbert after the notch
threads.push_back(new boost::thread(boost::bind(&stage3, st34,
input_status, st45)));
//amplitude after the Hilbert stage
threads.push_back(new boost::thread(boost::bind(&stage4, st45,
input_status, st56)));
//first window, does the most computations,
threads.push_back(new boost::thread(boost::bind(&window, st56,
input_status, st67, window1, 0, reuse)));
//second window, does very little computation
threads.push_back(new boost::thread(boost::bind(&window, st67,
input_status, st78, window2 / window1, 1, reuse)));
//third window, does very little computation
threads.push_back(new boost::thread(boost::bind(&window, st78,
input_status, st89, window3 / window2, 2, reuse)));
threads.push_back(new boost::thread(boost::bind(
&transverseActivityMonitor, st89, reuse, input_status)));
}

```

C

Serializing Data and Pushing It to the Pipeline

Listing C.1: This function is called from the real-time action and it converts the data to floats, serializes the data into pipeline elements and then passes the elements to the next stage.

```
//converts int_16 to float and splits it up turn by turn
void Pipeline::PushData(const int16_t* array_pointer, std::size_t &rows
, std::size_t &columns, std::size_t &triggerStamp, std::set<unsigned int> &bunches)
{
    unsigned* bunches_pointer =
        (unsigned*) malloc(bunches.size() * sizeof(unsigned));
    std::shared_ptr<unsigned> bunches_pointer_shared(bunches_pointer, free);
    std::size_t i = 0;
    //create a bunchset which every QueueElement has a
    //shared pointer to
    for (auto it = bunches.begin(); it != bunches.end(); it++)
    {
        *(bunches_pointer + i) = *it;
        i++;
    }
    //for every row
    for (std::size_t i = 0; i < rows; i++)
    {
        QueueElement* output;
        //Check if we can reuse a old element
        //or if we have to create a new
        if (reuse->Empty())
        {
            output = new QueueElement(columns);
        }
        else
        {
            output = reuse->Take();
            output->ChangeSize(columns);
        }
    }
}
```

```

    output->bunches.reset();
}
//for every column in the row
for (std::size_t j = 0; j < columns; j += 16)
{
    if (j + 16 < columns)
    {
        const int16_t* current_position = array_pointer + i * columns + j;
        //load 8 16bit unsigned
        __m128i a0 = _mm_loadu_si128((const __m128i *) (current_position));
        //load another 8 into another register
        __m128i a1 = _mm_loadu_si128((const __m128i *) (current_position + 8));
        // Split into two registers
        __m128i b0 = _mm_unpackhi_epi64(a0, a0);
        __m128i b1 = _mm_unpackhi_epi64(a1, a1);
        // Convert to 32-bit integers
        a0 = _mm_cvtepi16_epi32(a0);
        b0 = _mm_cvtepi16_epi32(b0);
        a1 = _mm_cvtepi16_epi32(a1);
        b1 = _mm_cvtepi16_epi32(b1);
        // Convert to float
        __m128 c0 = _mm_cvtepi32_ps(a0);
        __m128 d0 = _mm_cvtepi32_ps(b0);
        __m128 c1 = _mm_cvtepi32_ps(a1);
        __m128 d1 = _mm_cvtepi32_ps(b1);
        //store result
        _mm_storeu_ps(output->data + j + 0, c0);
        _mm_storeu_ps(output->data + j + 4, d0);
        _mm_storeu_ps(output->data + j + 8, c1);
        _mm_storeu_ps(output->data + j + 12, d1);
    }
    //if there are less then 16 short int left
    else
    {
        const int16_t* current_position = array_pointer + i * columns + j;
        memcpy(temp_16_short, current_position, (columns - j) * sizeof(short));
        //load 8 16bit unsigned
        __m128i a0 = _mm_loadu_si128((const __m128i *) (temp_16_short));
        //load another 8 into another register
        __m128i a1 = _mm_loadu_si128((const __m128i *) (temp_16_short + 8));
        // Split into two registers
        __m128i b0 = _mm_unpackhi_epi64(a0, a0);
        __m128i b1 = _mm_unpackhi_epi64(a1, a1);
        // Convert to 32-bit integers
        a0 = _mm_cvtepi16_epi32(a0);
        b0 = _mm_cvtepi16_epi32(b0);
        a1 = _mm_cvtepi16_epi32(a1);
        b1 = _mm_cvtepi16_epi32(b1);
        // Convert to float
        __m128 c0 = _mm_cvtepi32_ps(a0);
        __m128 d0 = _mm_cvtepi32_ps(b0);
        __m128 c1 = _mm_cvtepi32_ps(a1);
        __m128 d1 = _mm_cvtepi32_ps(b1);
    }
}

```

```
    _mm_storeu_ps(temp_16_floats + 0, c0);
    _mm_storeu_ps(temp_16_floats + 4, d0);
    _mm_storeu_ps(temp_16_floats + 8, c1);
    _mm_storeu_ps(temp_16_floats + 12, d1);
    memcpy(output->data + j, temp_16_floats, (columns - j) * sizeof(float));
}
}
//Set the value in the pipeline analyzer
*(status->pipeline_analyzer + 0) = static_cast<unsigned>
    (*(output->data + status->bunch_to_analyze));
//set internal turncounter in QueueElement
output->turn = status->GetIncreaseCounter();
//Set the size of the QueueElement
output->data_size = columns;
//Set the set of bunches
output->bunches = bunches_pointer_shared;
//push it to the queue for the next stage
st12->Put(output);
}
bunches_pointer_shared.reset();
}
```

D

Injection Oscillation Triggering Prevention Stage

Listing D.1: This is the first proper stage in the pipeline, it checks if the data is zero and if it is, it sets a counter in the status struct which is used in the detection stages to not trigger

```
void injectionOscillationPrevention(BlockingQueue<QueueElement*>* input_queue,
Status* status,BlockingQueue<QueueElement*>* output_queue)
{
    //used for when there are less than 8 floats left
    float* data = (float*) _mm_malloc(8 * sizeof(float), 32);
    //run forever
    while (true)
    {
        //take an item from the queue
        QueueElement* temp = input_queue->Take();
        std::size_t number_chrunched = 0;
        std::size_t number_of_elements = temp->data_size;
        float* newest = temp->data;
        unsigned* bunches = temp->bunches.get();
        //compare 8 floats at a time
        while (number_chrunched + 8 <= number_of_elements)
        {
            __m256 newest_vector = _mm256_load_ps(newest + number_chrunched);
            //compare if it is less then 1, normal values are >400
            __m256 cmp = _mm256_cmp_ps(newest_vector, _mm256_set1_ps(1.0f), _CMP_LT_OQ);
            int cmp_mask = _mm256_movemask_ps(cmp);
            for (std::size_t i = 0; i < 8; i++)
            {
                //data in is zero for this bunch
                if (cmp_mask & (1 << i))
                {
                    //set the turn counter in the struct for this bunch
                    unsigned bunch = *(bunches + number_chrunched + i);
                    *(status->last_time_data_was_zero + bunch) = temp->turn;
```

```
    }
}
number_chrunched += 8;
}
//special case
if (number_chrunched < number_of_elements)
{
    std::size_t rest = number_of_elements - number_chrunched;
    memcpy(data, newest + number_chrunched, rest * sizeof(float));
    __m256 newest_vector = _mm256_load_ps(data);
    __m256 cmp = _mm256_cmp_ps(newest_vector, _mm256_set1_ps(1.0f), _CMP_LT_OQ);
    int cmp_mask = _mm256_movemask_ps(cmp);
    for (std::size_t i = 0; i < rest; i++)
    {
        //data in is zero for this bunch
        if (cmp_mask & (1 << i))
        {
            unsigned bunch = *(bunches + number_chrunched + i);
            *(status->last_time_data_was_zero + bunch) = temp->turn;
        }
    }
}
//send data to next stage
output_queue->Put(temp);
}
}
```

E Notch Filter Stage

Listing E.1: This stage does the notch filtering which means that it centers the data around zero, also referred to as closed orbit suppression

```
//notch filter
void stage2(BlockingQueue<QueueElement*>* input_queue, Status* status,
BlockingQueue<QueueElement*>* output_queue)
{
    //used for special case
    float* data = (float*) _mm_malloc(8 * sizeof(float), 32);
    float* data1 = (float*) _mm_malloc(8 * sizeof(float), 32);
    //circular buffer since we need delay one QueueElement
    boost::circular_buffer<QueueElement*>* buffer =
    new boost::circular_buffer<QueueElement*>(2);
    //Take two QueueElements
    buffer->push_back(input_queue->Take());
    buffer->push_back(input_queue->Take());
    //Sanity check so they dont have different sizes
    while ((*buffer)[0]->data_size != (*buffer)[1]->data_size)
    {
        delete (*buffer)[0];
        buffer->push_back(input_queue->Take());
    }
    //constants used
    __m256 minus = _mm256_set1_ps(-1.0f);
    __m256 zeros = _mm256_set1_ps(0.0f);
    //run forever
    while (true)
    {
        //grab the data from the QueueElements
        float* oldest = (*buffer)[0]->data;
        float* newest = (*buffer)[1]->data;
        std::size_t number_of_elements = (*buffer)[0]->data_size;
```

```
std::size_t number_chrunched = 0;
//Do the notch filter for 8 bunches at a time
while (number_chrunched + 8 <= number_of_elements)
{
    //load 8 floats from n-1
    __m256 oldest_vector = _mm256_load_ps(oldest + number_chrunched);
    //load 8 floats from n
    __m256 newest_vector = _mm256_load_ps(newest + number_chrunched);
    __m256 addition = _mm256_mul_ps(oldest_vector, minus);
    __m256 res = _mm256_add_ps(addition, newest_vector);
    _mm256_store_ps(oldest + number_chrunched, res);
    number_chrunched += 8;
}
if (number_chrunched < number_of_elements)
{
    std::size_t rest = number_of_elements - number_chrunched;
    memcpy(data, oldest + number_chrunched, rest * sizeof(float));
    memcpy(data1, newest + number_chrunched, rest * sizeof(float));
    //load 8 floats from n-1
    __m256 oldest_vector = _mm256_load_ps(data);
    //load 8 floats from n
    __m256 newest_vector = _mm256_load_ps(data1);
    __m256 addition = _mm256_mul_ps(oldest_vector, minus);
    __m256 res = _mm256_add_ps(addition, newest_vector);
    _mm256_store_ps(data, res);
    memcpy(oldest + number_chrunched, data, rest * sizeof(float));
}
//push oldest QueueElement to output queue
*(status->pipeline_analyzer + 1) =
static_cast<unsigned> ((*buffer)[0]->data + status->bunch_to_analyze));
output_queue->Put ((*buffer)[0]);
//push a new QueueElement to buffer
buffer->push_back(input_queue->Take());
//sanity check
while ((*buffer)[0]->data_size != (*buffer)[1]->data_size)
{
    delete (*buffer)[0];
    buffer->push_back(input_queue->Take());
}
}
```


F Hilbert Transform Stage

Listing F.1: This stage does the Hilbert transform which generates a companion signal and it applies the delay of the real signal so the two signals have the same phase

```
void stage3(BlockingQueue<QueueElement*>* input_queue, Status* status,
BlockingQueue<QueueElement*>* output_queue)
{
    //circular buffer since we must delay 7 QueueElements
    boost::circular_buffer<QueueElement*>* buffer =
    new boost::circular_buffer<QueueElement*>(7);
    //grab 7 QueueElements
    for (std::size_t i = 0; i < 7; i++)
    {
        buffer->push_back(input_queue->Take());
    }
    for (std::size_t i = 0; i < 7; i++)
    {
        (*buffer)[i]->data_extra_size = (*buffer)[i]->data_size;
    }
    //sanity check
    while ((*buffer)[0]->data_size != (*buffer)[1]->data_size ||
(*buffer)[1]->data_size != (*buffer)[2]->data_size
|| (*buffer)[2]->data_size != (*buffer)[3]->data_size
|| (*buffer)[3]->data_size != (*buffer)[4]->data_size
|| (*buffer)[4]->data_size != (*buffer)[5]->data_size
|| (*buffer)[5]->data_size != (*buffer)[6]->data_size)
    {
        delete (*buffer)[0];
        buffer->push_back(input_queue->Take());
        (*buffer)[6]->data_extra_size = (*buffer)[6]->data_size;
    }
    //shift the data 3 QueueElements
    for (std::size_t i = 0; i < 4; i++)
```

```
{
    memcpy ((*buffer)[i]->data_extra, (*buffer)[i + 3]->data,
            (*buffer)[i]->data_size * sizeof(float));
}
//load the hilbert coefficients from the status struct
status->mtx->lock();
__m256 constants = _mm256_load_ps(status->hilbert);
status->mtx->unlock();
//run forever
while (true)
{
    //check if coefficients have changed
    if (status->new_hilbert)
    {
        status->mtx->lock();
        constants = _mm256_load_ps(status->hilbert);
        status->mtx->unlock();
        status->new_hilbert = false;
    }
    std::size_t number_chrunched = 0;
    std::size_t number_of_elements = (*buffer)[0]->data_size;
    //get all pointers
    float* n_6 = (*buffer)[6]->data;
    float* n_5 = (*buffer)[5]->data;
    float* n_4 = (*buffer)[4]->data;
    float* n_3 = (*buffer)[3]->data;
    float* n_2 = (*buffer)[2]->data;
    float* n_1 = (*buffer)[1]->data;
    float* n = (*buffer)[0]->data;
    //for every bunch
    while (number_chrunched < number_of_elements)
    {
        //load values into a register
        __m256 values = _mm256_set_ps(*(n_6 + number_chrunched),
            *(n_5 + number_chrunched),
            *(n_4 + number_chrunched), *(n_3 + number_chrunched),
            *(n_2 + number_chrunched ),
            *(n_1 + number_chrunched), *(n + number_chrunched ), 0.0f);
        //multiply them with the constants
        __m256 res = _mm256_mul_ps(constants, values);
        //magic summation
        __m256 temp = _mm256_hadd_ps(res, res);
        temp = _mm256_hadd_ps(temp, temp);
        const U256f u = { temp };
        float sum = u.a[0] + u.a[4];
        *(n + number_chrunched) = sum;
        number_chrunched += 1;
    }
    (*buffer)[0]->data_extra_size = (*buffer)[0]->data_size;
    //set the data in the pipeline analyzer
    *(status->pipeline_analyzer + 2) = static_cast<unsigned>
    (*( (*buffer)[0]->data + status->bunch_to_analyze));
    *(status->pipeline_analyzer + 3) = static_cast<unsigned>
```

```

    ((*buffer)[0]->data_extra + status->bunch_to_analyze));
    //push oldest QueueElement to the output queue
    output_queue->Put ((*buffer)[0]);
    //push a new QueueElement to the circular buffer
    buffer->push_back(input_queue->Take());
    //sanity check
    while ((*buffer)[0]->data_size != (*buffer)[1]->data_size ||
        (*buffer)[1]->data_size != (*buffer)[2]->data_size
        || (*buffer)[2]->data_size != (*buffer)[3]->data_size
        || (*buffer)[3]->data_size != (*buffer)[4]->data_size
        || (*buffer)[4]->data_size != (*buffer)[5]->data_size
        || (*buffer)[5]->data_size != (*buffer)[6]->data_size)
    {
        delete (*buffer)[0];
        buffer->push_back(input_queue->Take());
        (*buffer)[6]->data_extra_size = (*buffer)[6]->data_size;
    }
    //shift the data 3 QueueElements
    memcpy((*buffer)[3]->data_extra, (*buffer)[6]->data,
        (*buffer)[6]->data_size * sizeof(float));
    (*buffer)[3]->data_extra_size = (*buffer)[6]->data_size;
}
}

```

G

Amplitude Calculation Stage

Listing G.1: This stage calculates the instantaneous amplitude from the companion signal and the real signal

```
void stage4(BlockingQueue<QueueElement*>* input_queue, Status* status,
BlockingQueue<QueueElement*>* output_queue)
{
    //used for special case
    float* data = (float*) _mm_malloc(8 * sizeof(float), 32);
    float* data1 = (float*) _mm_malloc(8 * sizeof(float), 32);
    //run forever
    while (true)
    {
        //Grab a QueueElement
        QueueElement* temp = input_queue->Take();
        std::size_t number_chrunched = 0;
        std::size_t number_of_elements = temp->data_size;
        //for every bunch
        while (number_chrunched + 8 <= number_of_elements)
        {
            //load 8 floats from data(real signal)
            __m256 Q = _mm256_load_ps(temp->data + number_chrunched);
            //load 8 floats from data_extra(companion signal)
            __m256 I = _mm256_load_ps(temp->data_extra + number_chrunched);
            //
            __m256 Qpow = _mm256_mul_ps(Q, Q);
            __m256 Ipow = _mm256_mul_ps(I, I);
            __m256 res = _mm256_add_ps(Qpow, Ipow);
            __m256 square = _mm256_sqrt_ps(res);
            _mm256_store_ps(temp->data + number_chrunched, square);
            number_chrunched += 8;
        }
        if (number_chrunched < number_of_elements)
```

```

{
    std::size_t rest = number_of_elements - number_chrunched;
    memcpy(data, temp->data + number_chrunched, rest * sizeof(float));
    memcpy(data1, temp->data_extra + number_chrunched, rest * sizeof(float));
    __m256 Q = _mm256_load_ps(data);
    __m256 I = _mm256_load_ps(data1);
    __m256 Qpow = _mm256_mul_ps(Q, Q);
    __m256 Ipow = _mm256_mul_ps(I, I);
    __m256 res = _mm256_add_ps(Qpow, Ipow);
    __m256 square = _mm256_sqrt_ps(res);
    _mm256_store_ps(data, square);
    memcpy(temp->data + number_chrunched, data, rest * sizeof(float));
}
*(status->pipeline_analyzer + 4) =
static_cast<unsigned>(*(temp->data + status->bunch_to_analyze));
output_queue->Put(temp);
}
}

```



H Maximum Stage

Listing H.1: This stage keep tracks of the maximum instantaneous amplitude for the latest 4096 turns

```
void Maximum(BlockingQueue<QueueElement*>* input_queue,
BlockingQueue<QueueElement*>* output_queue, Status* status) {
    //used to store the maximums
    QueueElement* maximums = new QueueElement(3564, false);
    float scaling = 1.0f;
    //set maximum to zero
    for (std::size_t i = 0; i < 3564; i++) {
        *(maximums->data + i) = 0.0;
    }
    std::size_t counter = 0;
    //run forever
    while (true) {
        //get queueelement
        QueueElement* temp = input_queue->Take();
        for (std::size_t i = 0; i < 3560; i += 8) {
            __m256 load = _mm256_load_ps(temp->data + i);
            __m256 load_max = _mm256_load_ps(maximums->data + i);
            __m256 cmp = _mm256_cmp_ps(load, load_max, _CMP_GE_OQ);
            U256IF un;
            un.v = cmp;
            _mm256_maskstore_ps(maximums->data + i, un.a, load);
        }
        for (std::size_t i = 0; i < 4; i++) {
            if (*(temp->data + 3560 + i) > *(maximums->data + 3560 + i)) {
                *(maximums->data + 3560 + i) = *(temp->data + 3560 + i);
            }
        }
        output_queue->Put(temp);
        if (counter == 4096) {
```

```

    for (std::size_t i = 0; i < 3560; i += 8) {
        __m256 load = _mm256_load_ps(maximums->data + i);
        __m256 scale = _mm256_set1_ps(scaling);
        __m256 res = _mm256_mul_ps(load, scale);
        _mm256_store_ps(maximums->data + i, res);
    }
    *(maximums->data + 3560) *= scaling;
    *(maximums->data + 3561) *= scaling;
    *(maximums->data + 3562) *= scaling;
    *(maximums->data + 3563) *= scaling;
    status->mtx->lock();
    memcpy(status->transverseActivityMonitorMaximum, maximums->data,
        3564 * sizeof(float));
    scaling = status->scaling;
    status->mtx->unlock();
    for (std::size_t i = 0; i < 3564; i++) {
        *(maximums->data + i) = 0.0;
    }
    counter = 0;
}
counter++;
}
}

```

I Instability Detection Stage

Listing I.1: This stage detects a potential instability in each separate bunch using moving averages

```
void window(BlockingQueue<QueueElement*>* input_queue, Status* status,
BlockingQueue<QueueElement*>* output_queue, std::size_t window_size,
std::size_t place, BlockingQueue<QueueElement*>* reuse)
{
    //used to keep track if the bunch is already defined unstable
    bool unstable[3564] ={ false };
    //used to send information about instabilites to the real-time action
    BlockingQueue<Unstable*>* unstaBlQueue=status->winlqueue;
    //constant vector used to divide the sum to get the average
    float window_size_float = (float) window_size;
    __m256 window_sizes = _mm256_set1_ps(window_size_float);
    //A circular buffer since we must delay the same number of QueueElements
    //as the window size
    boost::circular_buffer<QueueElement*>* buffer =
    new boost::circular_buffer<QueueElement*>(window_size);
    //Used in the special case for memory copying
    float* temp_storage = (float*) _mm_malloc(8 * sizeof(float), 32);
    //used to store the sum of all values
    float* total = (float*) _mm_malloc(3564 * sizeof(float), 32);
    float* old_total = (float*) _mm_malloc(3564 * sizeof(float), 32);
    //fill the circular buffer and add values to the sum
    for (std::size_t i = 0; i < window_size; i++)
    {
        QueueElement* temp = input_queue->Take();
        unsigned* bunches = temp->bunches.get();
        for (std::size_t j = 0; j < temp->data_size; j++)
        {
            //add value to old and new sum
            *(old_total + *(bunches + j)) += *(temp->data + j);
        }
    }
}
```

```

        *(total + *(bunches + j)) += *(temp->data + j);
    }
    //Push QueueElement to circular buffer
    buffer->push_back(temp);
}
//keep track so we know when we have pulled
//window_size number of QueueElement
std::size_t counter = 0;
//even though we do not receive all bunches we send all bunches to the next
//stage. For that we need this set to be consistent with the design
unsigned* bunches_pointer = (unsigned*) malloc(3564 * sizeof(unsigned));
std::shared_ptr<unsigned> bunches_pointer_shared(bunches_pointer, free);
for (std::size_t i = 0; i < 3564; i++)
{
    *(bunches_pointer + i) = (unsigned) i;
}
//run forever
while (true)
{
    //grab a QueueElement from the input queue
    QueueElement* temp = input_queue->Take();
    //this contains the indexes for which bunches are in this QueueElement
    //normally it should be a range from 0..3564
    unsigned* bunches = temp->bunches.get();
    //add the values to the sum
    for (std::size_t j = 0; j < temp->data_size; j++)
    {
        *(total + *(bunches + j)) += *(temp->data + j);
    }
    //grab the oldest QueueElement from the Circular buffer
    QueueElement* oldest = (*buffer)[0];
    unsigned* bunches_old = oldest->bunches.get();
    //remove values from oldest sum
    for (std::size_t j = 0; j < oldest->data_size; j++)
    {
        //add amplitude to total, TODO check if float is enough
        *(total + *(bunches_old + j)) -= *(oldest->data + j);
    }
    counter++;
    //time to check difference between new and old sum
    if (counter >= window_size)
    {
        //we use the oldest QueueElement to pass
        //the data from this window to the next stage
        oldest->ChangeSize(3564);
        oldest->data_size = 3564;
        oldest->data_extra_size = 3564;
        oldest->bunches = bunches_pointer_shared;
        for (std::size_t i = 0; i <= 3552; i += 8)
        {
            __m256 total_m256, old_total_m256;
            //load 8 floats from total
            total_m256 = _mm256_load_ps(total + i);

```

```
old_total_m256 = _mm256_load_ps(old_total + i);
//generate the new old_total
__m256 threshold_filter = _mm256_set1_ps(1.0f -
status->GetThresholdFilter(place));
__m256 threshold_filter_complement = _mm256_set1_ps(status->
GetThresholdFilter(place));
__m256 new_average = _mm256_mul_ps(total_m256, threshold_filter);
__m256 new_average_second =
mm256_mul_ps(old_total_m256, threshold_filter_complement);
new_average = _mm256_add_ps(new_average, new_average_second);
__mm256_store_ps(old_total + i, new_average);
//div the 8 floats in total with the size of the window
total_m256 = _mm256_div_ps(total_m256, window_sizes);
//check if value less than one but still
//not zero because of numerical errors
__m256 less_than_one =
_mm256_cmp_ps(total_m256, _mm256_set1_ps(1.0f), _CMP_LT_OQ);
__m256 average =
_mm256_blendv_ps(total_m256, _mm256_setzero_ps(), less_than_one);
__mm256_store_ps(oldest->data + i, average);
//div the 8 floats in total with the wize of the window
old_total_m256 = _mm256_div_ps(old_total_m256, window_sizes);
//check if value less than one but still
//not zero because of numerical errors
less_than_one = _mm256_cmp_ps(old_total_m256,
_mm256_set1_ps(1.0f), _CMP_LT_OQ);
__m256 old_average = _mm256_blendv_ps(old_total_m256,
_mm256_setzero_ps(), less_than_one);
__m256 threshold_percentage =
_mm256_set1_ps(status->GetThresHoldPercentage(place));
old_average = _mm256_mul_ps(old_average, threshold_percentage);
__m256 res = _mm256_cmp_ps(average, old_average, _CMP_GT_OQ);
//extract the sign bit from each
int res_bitmask = _mm256_movemask_ps(res);
//int used as mask
int shift_int = 1;
//for every bunch currently being evaulated
for (std::size_t j = 0; j < 8; j++)
{
    //if the new value is bigger than the old value times the threshold
    if (res_bitmask & shift_int)
    {
        //if it is not a injection
        if (*(status->last_time_data_was_zero + i + j) +
(status->prevent_injection) < status->GetCounter())
        {
            //Notify real-time action that it is unstable
            Unstable* tempUnstable = new Unstable();
            tempUnstable->bunch = (short unsigned int) (i + j);
            tempUnstable->turn = temp->turn;
            tempUnstable->start = true;
            tempUnstable->place = (unsigned) place;
            unstaQueue->Put(tempUnstable);
        }
    }
}
```

```

        unstable[i + j] = true;
    }
}
shift_int = shift_int << 1;
}
}
//special case
__m256 total_m256, old_total_m256;
total_m256 = _mm256_set_ps(*(total + 3560), *(total + 3561),
*(total + 3562), *(total + 3563), 0.0f, 0.0f, 0.0f, 0.0f);
old_total_m256 = _mm256_set_ps(*(old_total + 3560),
*(old_total + 3561), *(old_total + 3562),
*(old_total + 3563), 0.0f, 0.0f, 0.0f, 0.0f);
//generate the new old_total
__m256 threshold_filter =
_mm256_set1_ps(status->GetThresholdFilter(place));
__m256 threshold_filter_complement =
_mm256_set1_ps(1.0f - status->GetThresholdFilter(place));
__m256 new_average = _mm256_mul_ps(total_m256, threshold_filter);
__m256 new_average_second =
_mm256_mul_ps(old_total_m256, threshold_filter_complement);
new_average = _mm256_add_ps(new_average, new_average_second);
_mm256_store_ps(temp_storage, new_average);
*(old_total + 3560) = *(temp_storage + 0);
*(old_total + 3561) = *(temp_storage + 1);
*(old_total + 3562) = *(temp_storage + 2);
*(old_total + 3563) = *(temp_storage + 3);
//div the 8 floats in total with the wize of the window
total_m256 = _mm256_div_ps(total_m256, window_sizes);
//check if value less than one but still
//not zero because of numerical errors
__m256 less_than_one =
_mm256_cmp_ps(total_m256, _mm256_set1_ps(1.0f), _CMP_LT_OQ);
__m256 average =
_mm256_blendv_ps(total_m256, _mm256_setzero_ps(), less_than_one);
_mm256_store_ps(temp_storage, average);
*(oldest->data + 3560) = *(temp_storage + 7);
*(oldest->data + 3561) = *(temp_storage + 6);
*(oldest->data + 3562) = *(temp_storage + 5);
*(oldest->data + 3563) = *(temp_storage + 4);
//div the 8 floats in total with the wize of the window
old_total_m256 = _mm256_div_ps(old_total_m256, window_sizes);
//check if value less than one but still
//not zero because of numerical errors
less_than_one =
_mm256_cmp_ps(old_total_m256, _mm256_set1_ps(1.0f), _CMP_LT_OQ);
__m256 old_average =
_mm256_blendv_ps(old_total_m256, _mm256_setzero_ps(), less_than_one);
__m256 threshold_percentage =
_mm256_set1_ps(status->GetThresHoldPercentage(place));
average = _mm256_mul_ps(average, threshold_percentage);
__m256 res = _mm256_cmp_ps(average, old_average, _CMP_GT_OQ);
//extract the sign bit from each

```

```
int res_bitmask = _mm256_movemask_ps(res);
//int used as mask
int shift_int = 1;
//for the four last bunches
for (std::size_t j = 0; j < 4; j++)
{
    //if the new value is larger than the old value times the threshold
    if (res_bitmask & shift_int)
    {
        //if it is not a injection
        if (*(status->last_time_data_was_zero + 3560 + j) +
            (status->prevent_injection) < status->GetCounter())
        {
            //notify real-time action that the bunch is unstable
            Unstable* tempUnstable = new Unstable();
            tempUnstable->bunch = (short unsigned int) (3560 + j);
            tempUnstable->turn = oldest->turn;
            tempUnstable->start = true;
            tempUnstable->place = (unsigned) place;
            unstableQueue->Put(tempUnstable);
            unstable[3560 + j] = true;
        }
    }
    shift_int = shift_int << 1;
}
//reset counter
counter = 0;
oldest->turn = temp->turn;
//set the value in the pipeline analyzer
*(status->pipeline_analyzer + 5 + place) =
static_cast<unsigned>(*(oldest->data + status->bunch_to_analyze));
//Push the item to the queue for the next stage
output_queue->Put(oldest);
//Push a new element to the circular buffer
buffer->push_back(temp);
}
else
//if it is not time to check the values
{
    //if the reuse queue is full delete the item
    if (reuse->Full())
    {
        delete oldest;
    }
    //else push it to the queue for reuse
    else
    {
        reuse->Put(oldest);
    }
    //push a new value to the circular buffer
    buffer->push_back(temp);
}
}
```

}

J

Transverse Activity Monitor Stage

Listing J.1: This stage is the last stage after the last moving average window which just scales the data and sends it to the status struct so it can be displayed in the CCC

```
void transverseActivityMonitor(BlockingQueue<QueueElement*>* input_queue,
BlockingQueue<QueueElement*>* output_queue, Status* status)
{
    //used for special case
    float* data = (float*) _mm_malloc(8 * sizeof(float), 32);
    //initial scaling
    float scaling = 1.0f;
    while (true)
    {
        //grab one QueueElement from the input queue
        QueueElement* temp = input_queue->Take();
        std::size_t number_chrunched = 0;
        std::size_t number_of_elements = temp->data_size;
        //for every bunch, 8 bunches at a time
        while (number_chrunched + 8 <= number_of_elements)
        {
            //load 8 floats
            __m256 load = _mm256_load_ps(temp->data + number_chrunched);
            //load the scaling
            __m256 scale = _mm256_set1_ps(scaling);
            //multiply the floats with the scaling
            __m256 res = _mm256_mul_ps(load, scale);
            //store the result
            _mm256_store_ps(temp->data + number_chrunched, res);
            number_chrunched += 8;
        }
        //special case for the last bunches
        if (number_chrunched < number_of_elements)
        {
```

```

        std::size_t rest = number_of_elements - number_chrunched;
        memcpy(data, temp->data + number_chrunched, rest * sizeof(float));
        __m256 load = _mm256_load_ps(data);
        __m256 scale = _mm256_set1_ps(scaling);
        __m256 res = _mm256_mul_ps(load, scale);
        _mm256_store_ps(data, res);
        memcpy(temp->data + number_chrunched, data, rest * sizeof(float));
    }
    //send the values to the status struct, this must be protected by a lock
    status->mtx->lock();
    scaling = status->scaling;
    memcpy(status->transverseActivityMonitor,
        temp->data, sizeof(float) * temp->data_size);
    status->mtx->unlock();
    //set the value in the pipeline analyzer
    *(status->pipeline_analyzer + 8) =
    static_cast<unsigned>(*(temp->data + status->bunch_to_analyze));
    //push the QueueElement to the reusequeue
    output_queue->Put(temp);
}
}

```

K

Hilbert Filter Analysis in Matlab

Listing K.1: Analysis of the performance of the calculated Hilbert FIR filter and the Matlab generated filter

```
time = 0:1:1000; % in samples
Q = 0.305; % oscillation frequency
A = 10000; % oscillation amplitude
Offset = 5000; % closed orbit offset

Hilbert = 1.000*[-0.2122, 0, -0.6366, 0, 0.6366, 0 0.2122]; %
Hilbert_optimal = 1.000*[-0.090606105, -0.019801411, -0.594091993,
0.000000000, 0.594091993, 0.019801411, 0.090606105]; % coefficients

Notch = [1,-1];
Delay = [0 0 0 1];

pickupdata = Offset + A*cos(2*pi*Q*time);

afterNotch = filter(Notch,1,pickupdata);

Q = filter(Hilbert,1,afterNotch);
Qopt = filter(Hilbert_optimal,1,afterNotch);
I = filter(Delay,1,afterNotch);

A = sqrt(I.^2 + Q.^2);
Aopt = sqrt(I.^2 + Qopt.^2);

%plot(time,I,time,Q);
plot(I,Q,'r');
xlim([-2e4 2e4]);
ylim([-2e4 2e4]);
legend('Calculated_coeffs');
xlabel('I');
```

```
ylabel('Reconstructed_Q');

figure(2);
plot(I,Qopt,'b');
xlim([-2e4 2e4]);
ylim([-2e4 2e4]);
legend('Matlab_coeffs');
xlabel('I');
ylabel('Reconstructed_Q');

figure(3);
plot(1:1:length(A),A,1:1:length(Aopt),Aopt,'r');
legend('Current_coeffs','Optimized_coeffs');
xlabel('Sample_#');
ylabel('Sqrt(I^2+Q^2)');
```