FAIR The Universe in the Laboratory

GNU Radio 4.0:
Use-Cases at the GSI/FAIR Accelerator Facility
& an Overview of New Features and Significant Enhancements

Ralph J. Steinhagen[1]

on behalf of the GR Architecture Team:
Josh Morman[2], Derek Kozel[2], John Sallay[2], Björn Balazs[3], Ivan Čukić[3],
Matthias Kretz[1], Alexander Krimm[1], Semën Lebedev[1], Frank Osterfeld[3], …

[1]GSI, Darmstadt, Germany, [2] GNU Radio 4.0 (lead),
[3]KDAB Berlin, Germany

- Who we are.
- Why we use and invested into GNU Radio.
- Why to move towards GNU Radio 4.0?
  - only quick overview of highlights, details and tutorials on GitHub, Indico & YouTube
- Quick Q&A
  - more ample time during session this afternoon – 1’30”

The Universe in the Laboratory

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

Federal Ministry of Education and Research

HESSEN

- **GSI Helmholtz Centre for Heavy Ion Research in Darmstadt, Germany** (est. since 1969)
- Shareholders: federal government (90%), Hesse (8%), Rhineland-Palatinate (1%), Thuringia (1%)
- Further locations (Helmholtz Institutes) in Mainz and Jena
- Hosts: **FAIR – Facility for Anti-Proton-and-Ion-Research** (est. since 2010)
- Employees: approx. 1,580

We explore the universe…

…in the lab.

Photo: J. Hosan/GSI/FAIR, “NASA, ESA, the Hubble Heritage Team (STScI/AURA), A. Nota (ESA/STScI), and the Westerlund 2 Science Team”

The Universe in the Laboratory

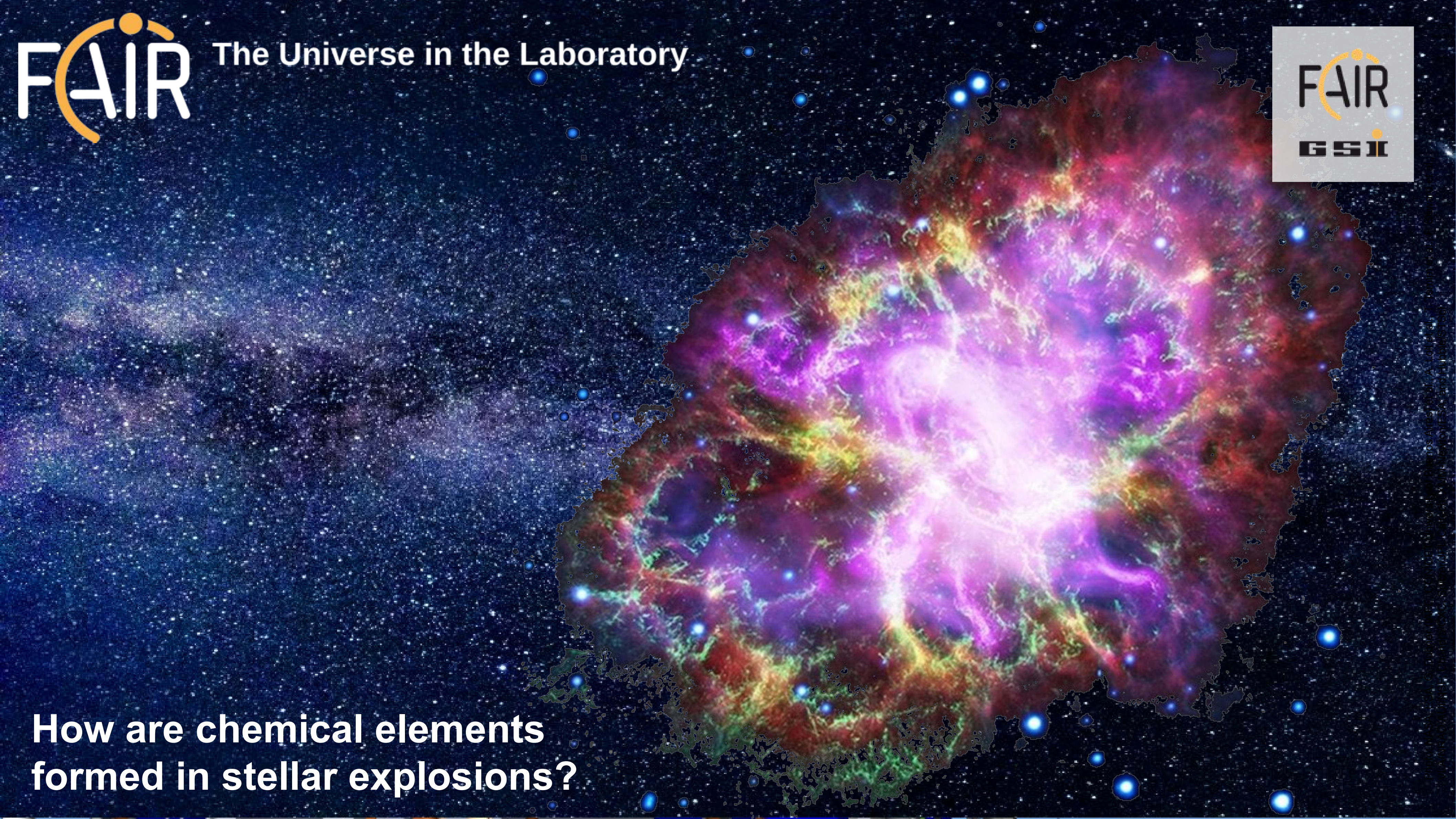
What are the smallest building blocks of matter?

How, Where
and When were they created?


Photo: NASA

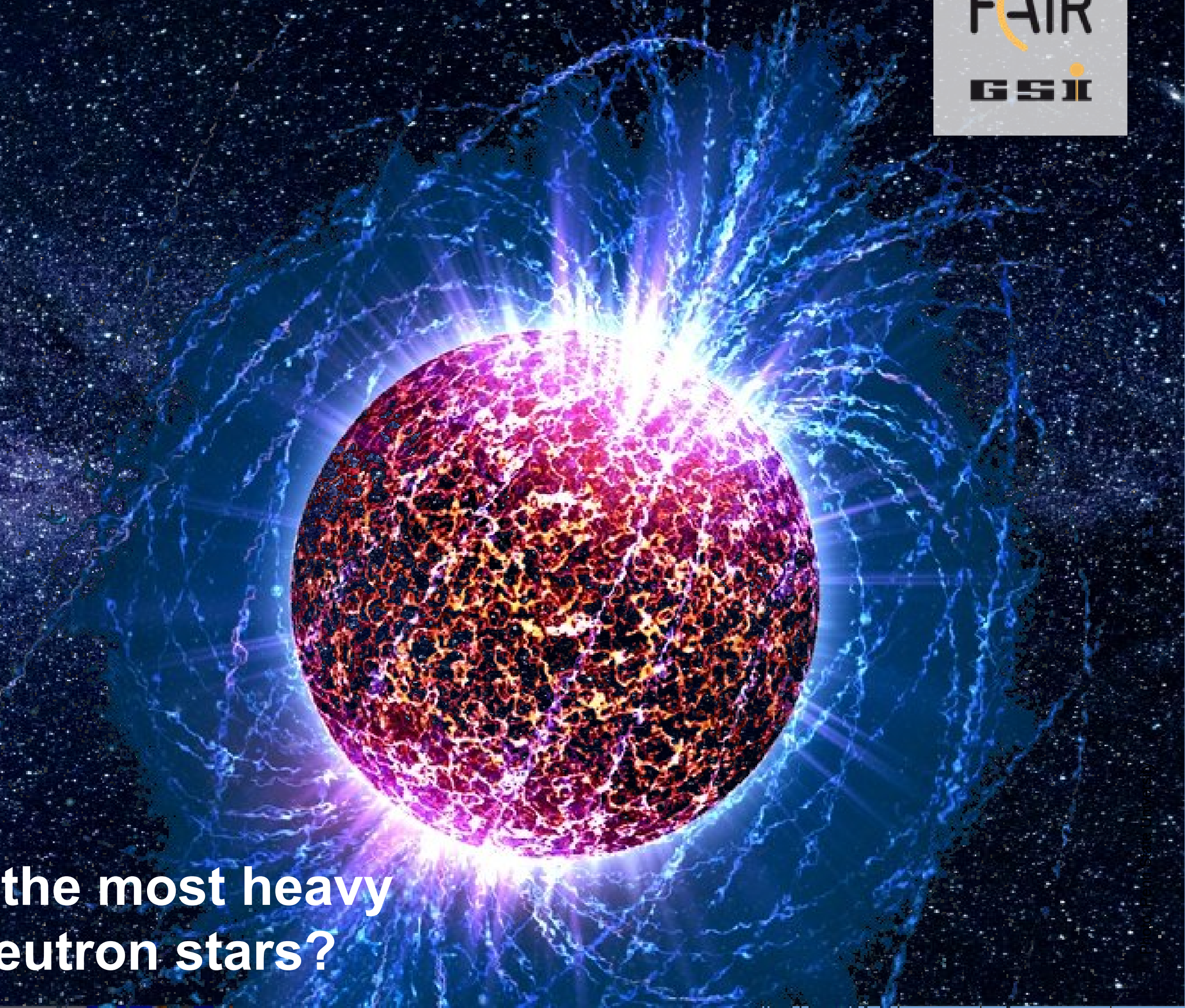
What happened to Anti-Matter?

The Universe in the Laboratory



How are chemical elements formed in stellar explosions?

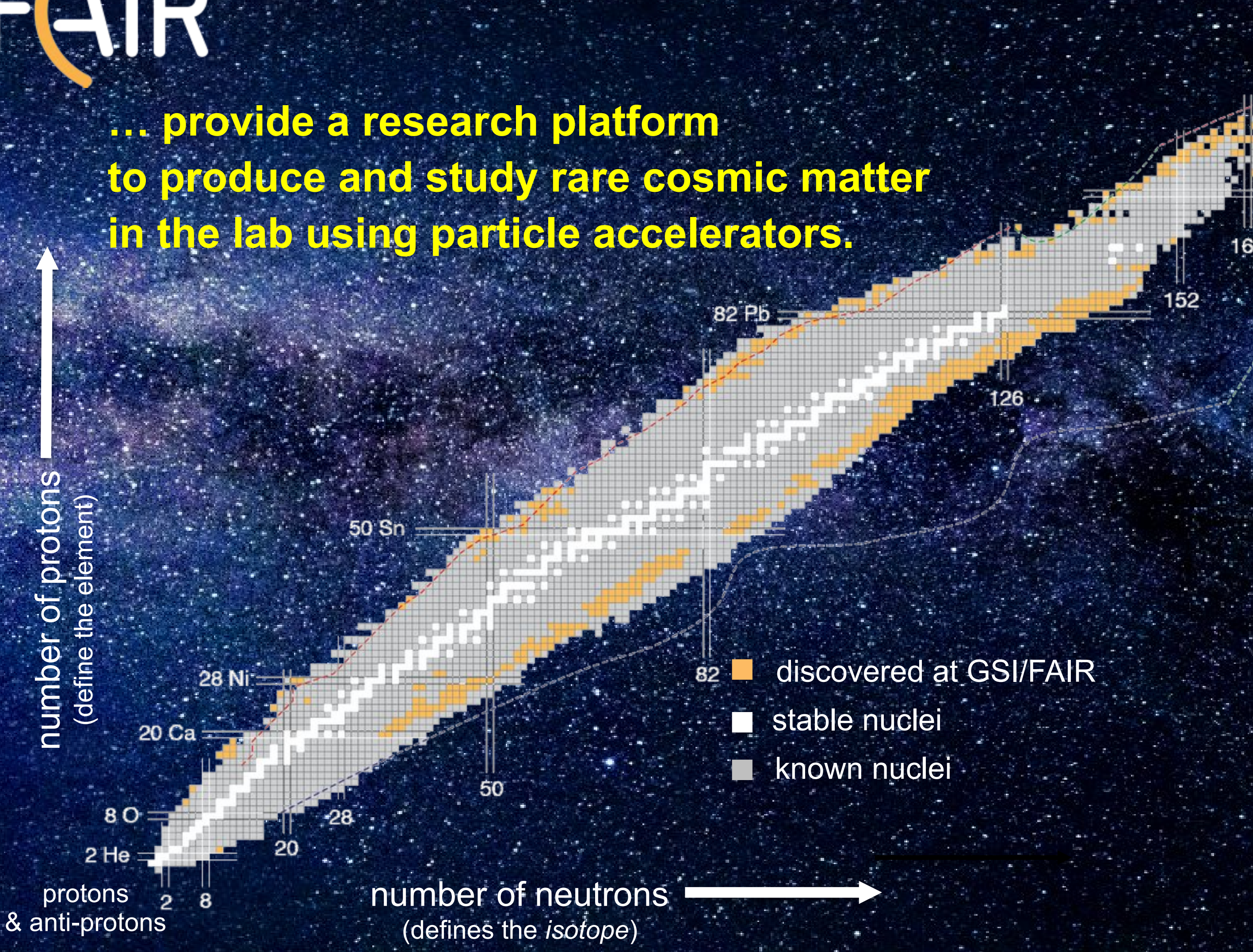
The Universe in the Laboratory



What does matter look like in the most heavy objects of our universe, the neutron stars?

The Universe in the Laboratory

**… provide a research platform
to produce and study rare cosmic matter
in the lab using particle accelerators.**



a la carte:
protons, anti-protons, (rare) isotopes ...

Applications: e.g. Cancer Therapy with Heavy Ions

- Precise, gentle and very successful!
- Treatment of 440 patients at GSI
- Established and in clinical operation in Heidelberg and Marburg
- At GSI: further R&D

Photo: A. Zschau/GSI

Photo: J. Hosan/HA Hessen Agentur

Photo: G. Otto/GSI

Photo: J. Hosan/GSI/FAIR

The Universe in the Laboratory

… will accelerate Particles with unprecedented:


Photo: G. Otto/GSI/FAIR

Intensity

10‘000 x
more particles

Output Power

10-100 x more

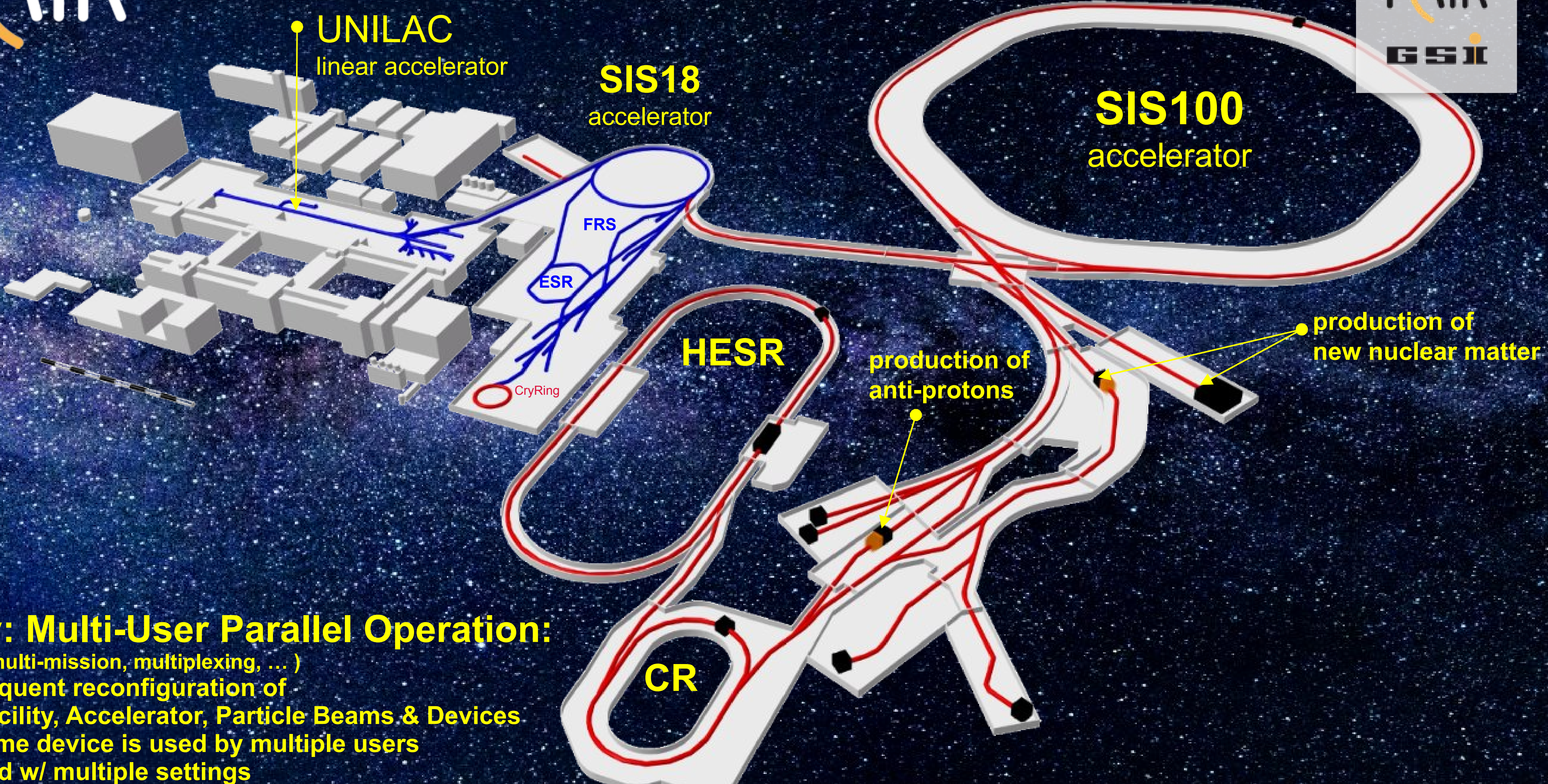
Quality

maximise precision/
selectivity

Versatility

all chemical elements/ions
and antiprotons

The Universe in the Laboratory



Key: Multi-User Parallel Operation:

(aka. multi-mission, multiplexing, …)

→ frequent reconfiguration of
Facility, Accelerator, Particle Beams & Devices

→ same device is used by multiple users
and w/ multiple settings

The Universe in the Laboratory

- **~ 3000 researchers**
- **~ 400 institutes**
- **50 countries &**
  11 share-holder (states)

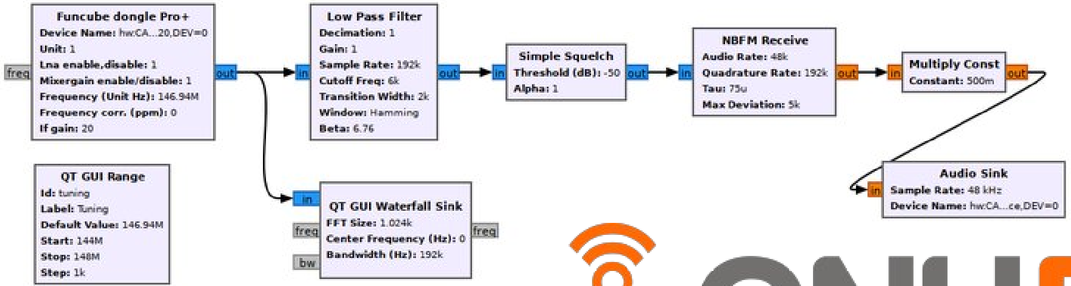
- **25 accelerator and experimental structures,**
  labs and other operation
  and supply structures
- **Underground accelerator ring**
  with a circumference
  of approx. 1,100 m
- **Around 150,000 m²** of total area

Picture: ion42, D. Fehrenz, GSI/FAIR

What is ...

Software-Defined-Radio (SDR): shift of traditional radio-frequency (RF), signal- and event-processing implementations from Hard- to Software

… a flexible flow-graph-based signal/event processing toolkit


Funcube dongle Pro+
Device Name: hw:CA...20,DEV=0
Unit: 1
Lna enable,disable: 1
Mixergain enable/disable: 1
Frequency (Unit Hz): 146.94M
Frequency corr. (ppm): 0
If gain: 20

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 192k
Cutoff Freq: 6k
Transition Width: 2k
Window: Hamming
Beta: 6.76

Simple Squelch
Threshold (dB): -50
Alpha: 1

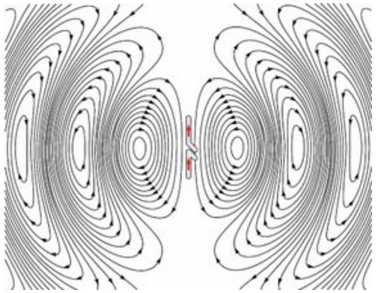
NBFM Receive
Audio Rate: 48k
Quadrature Rate: 192k
Tau: 75u
Max Deviation: 5k

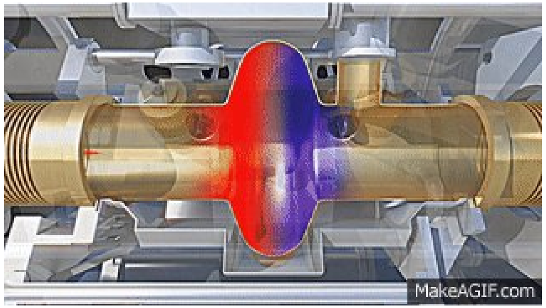
Multiply Const
Constant: 500m

Audio Sink
Sample Rate: 48 kHz
Device Name: hw:CA...ce,DEV=0

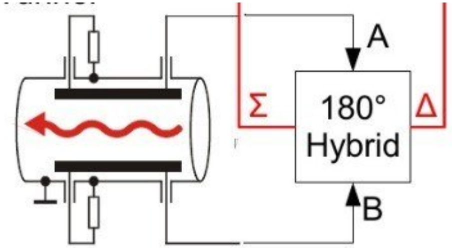
QT GUI Range
Id: tuning
Label: Tuning
Default Value: 146.94M
Start: 144M
Stop: 148M
Step: 1k

QT GUI Waterfall Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 192k





GNU Radio 4.0 beta
THE FREE & OPEN SOFTWARE RADIO ECOSYSTEM



free/libre open-source software

GPLv3 Free Software Free as in Freedom

& core: LGPLv3 Free Software Free as in Freedom

20+ Years of Expertise, Modernised for Today

- Community-Driven,
- Streamlined, and
- Ready for Tomorrow

CONNECT
SHARE
COLLABORATE

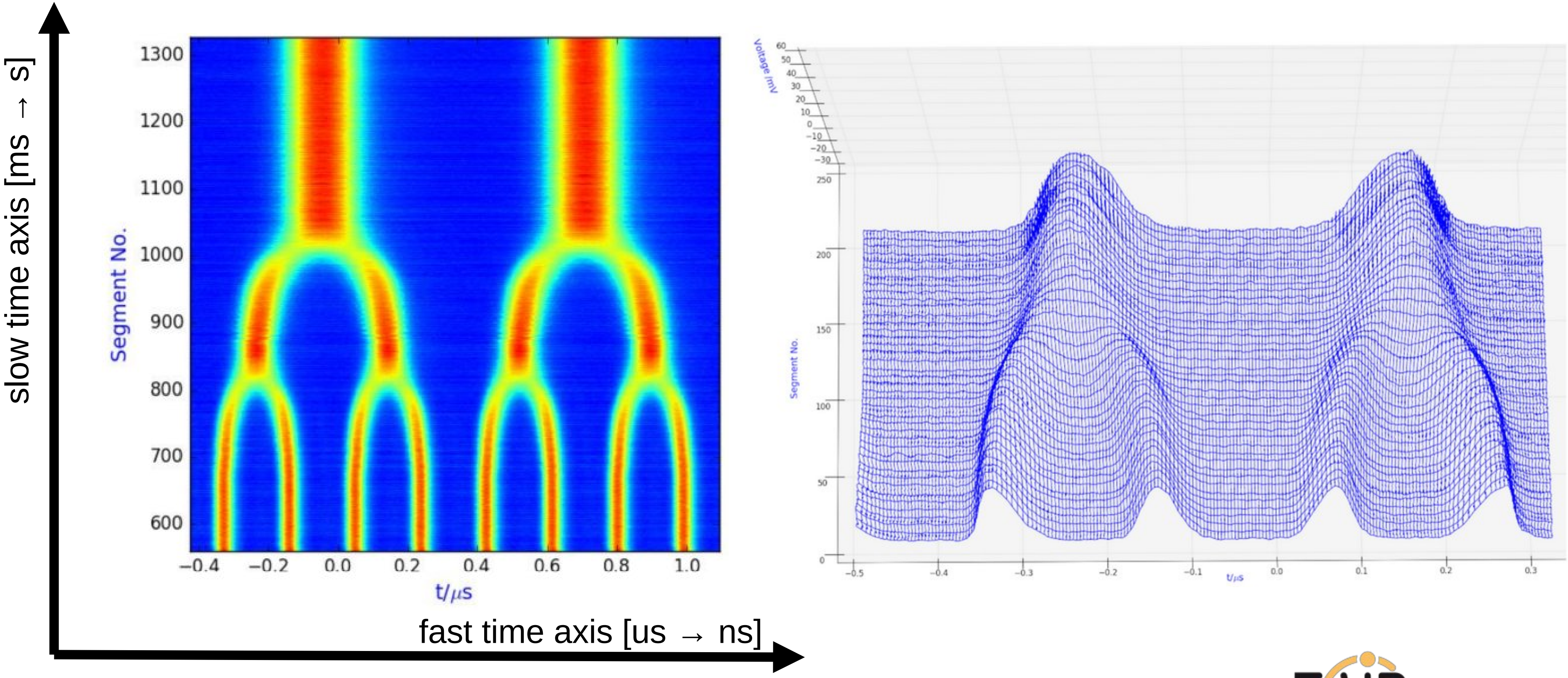
… a high-performance signal-processing toolkit.

std::simd

WA

AI

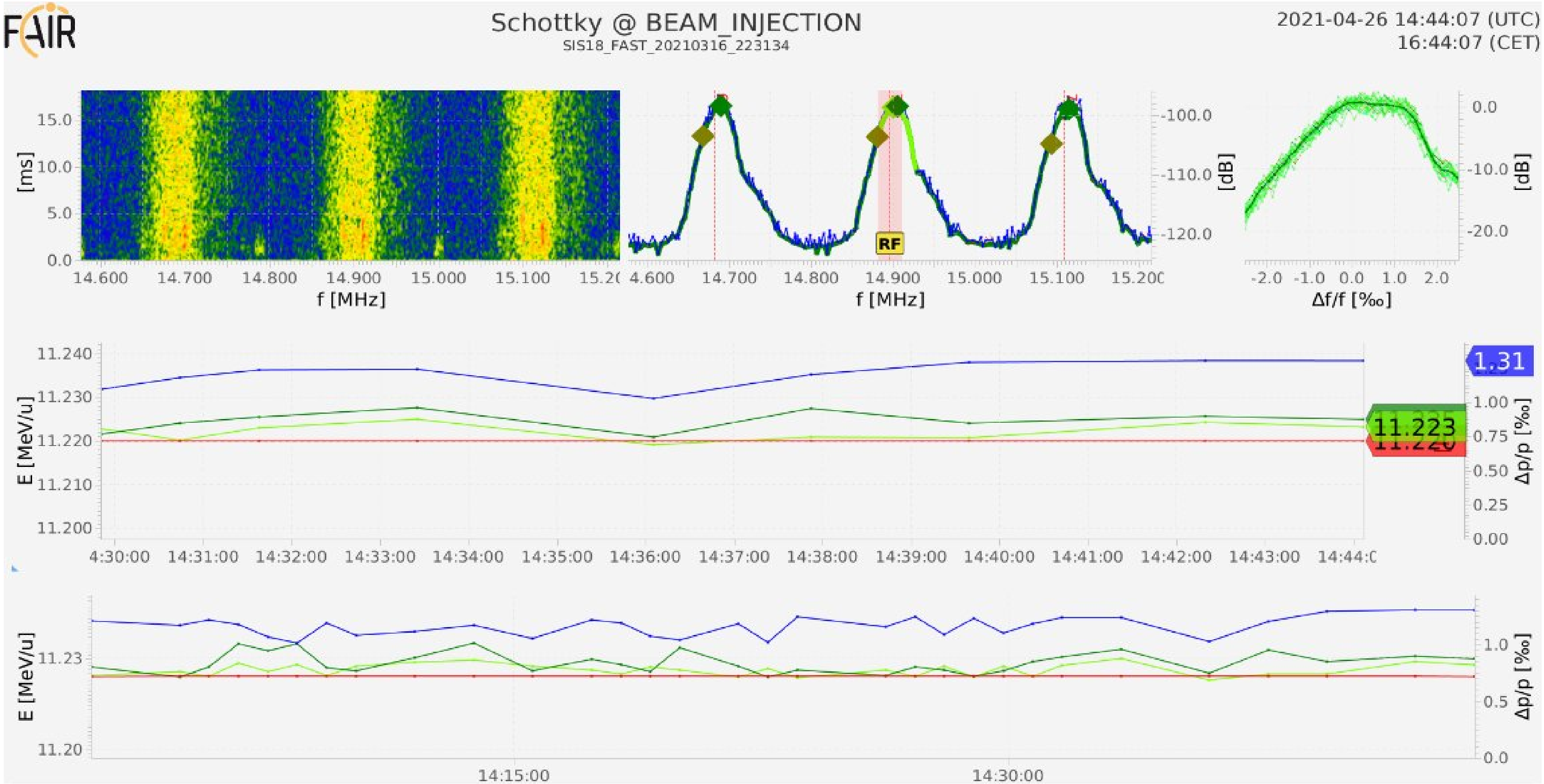
Bunch Merging Gymnastics @ SIS18 Dieter E. Lens et al.



fast event-processing (bursts): 1 us frame/segment every 5-10 us

RF Particle Beam Diagnostics – Frequency Domain

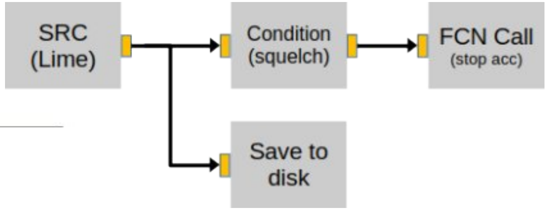
Energy and Momentum distribution tracking @ SIS18 Alexander Krimm et al.

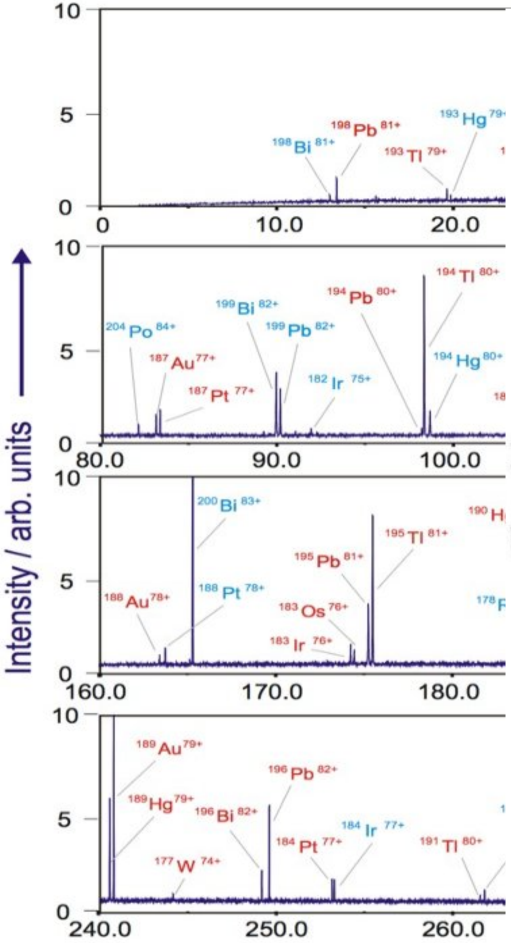


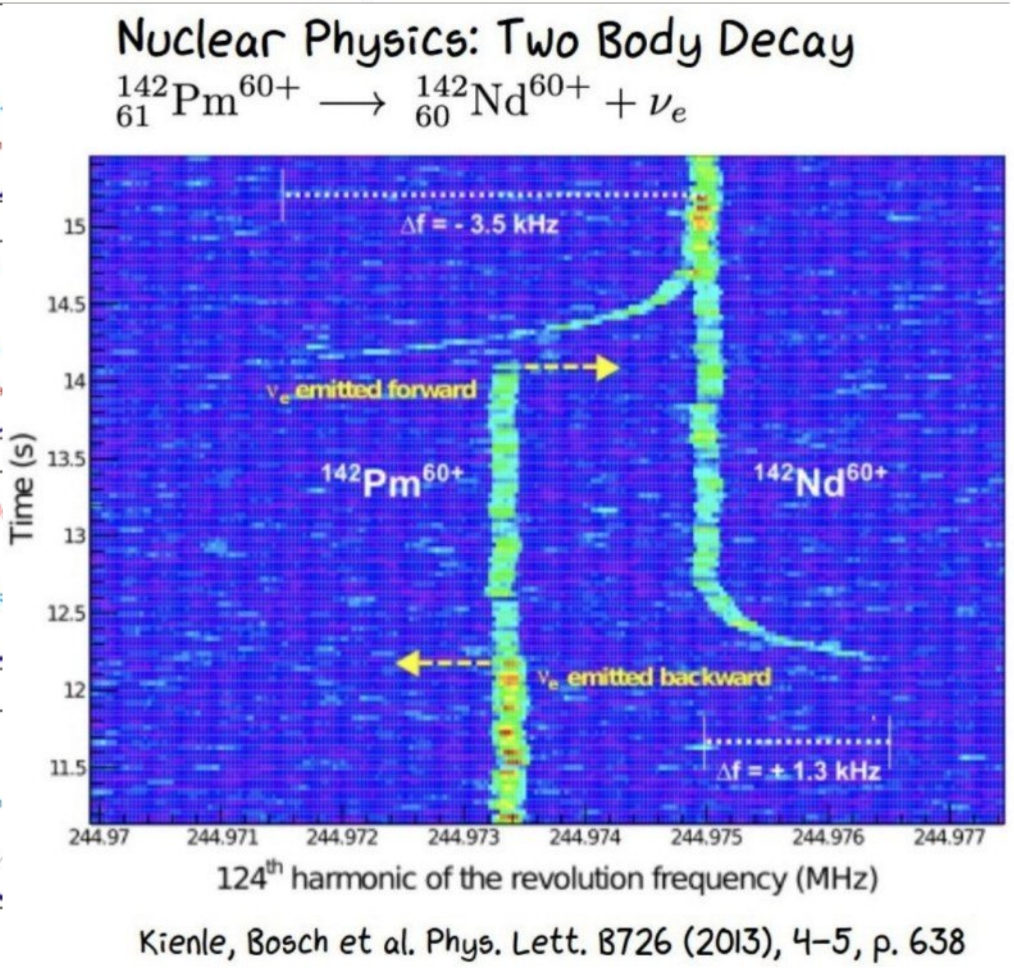
continuous tracking of momentum (carrier wave) offset/distribution

RF Particle Beam Diagnostics – Frequency Domain

Schottky-based Mass Spectroscopy @ ESR S. Sanjari et al.







Kienle, Bosch et al. Phys. Lett. B726 (2013), 4–5, p. 638

Upgrade goals:

- move offline
  → online processing
- stop accelerator
  on rare particle detection
  ↔ "squelch" analogy
- vendor neutrality,
  maintainability, …

N.B. link between event-driven data processing and packet-modem!!

GNU Radio & Beam-Based Services

driven by functional need for distributed middle-tier processing;

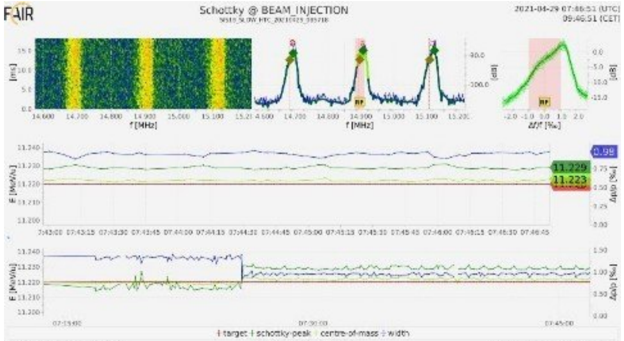
A) aggregation and sanitisation of source device data
(from multiple devices, experiments, TGA, archiving system, ...)

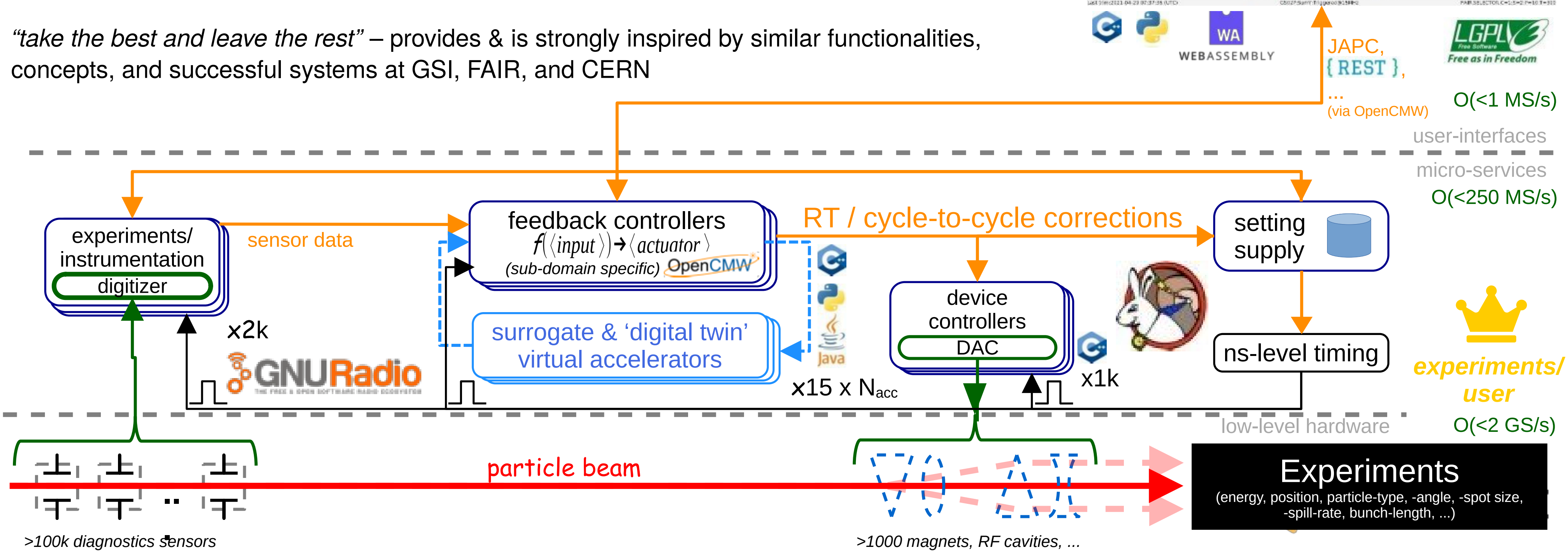
B) generic numerical signal-, data- and domain-logic- post-processing
(performance and online reconfiguration requirements)

C) output of feedback control signals to other systems and services

“take the best and leave the rest” – provides & is strongly inspired by similar functionalities, concepts, and successful systems at GSI, FAIR, and CERN

beam-based controls applications
(interactive, expert. & monitoring/web-type)





WEBASSEMBLY

JAPC,
{ REST },
...
(via OpenCMW)

LGPLv3 Free as in Freedom

O(<1 MS/s)

user-interfaces

micro-services

O(<250 MS/s)

experiments/
instrumentation
digitizer

sensor data

x2k

GNURadio

feedback controllers
$f(\langle input \rangle) \rightarrow \langle actuator \rangle$
(sub-domain specific) OpenCMW

surrogate & ‘digital twin’
virtual accelerators

RT / cycle-to-cycle corrections

Java

x15 x N_{acc}

device
controllers
DAC

x1k

setting
supply

ns-level timing

experiments/
user

O(<2 GS/s)

low-level hardware

particle beam

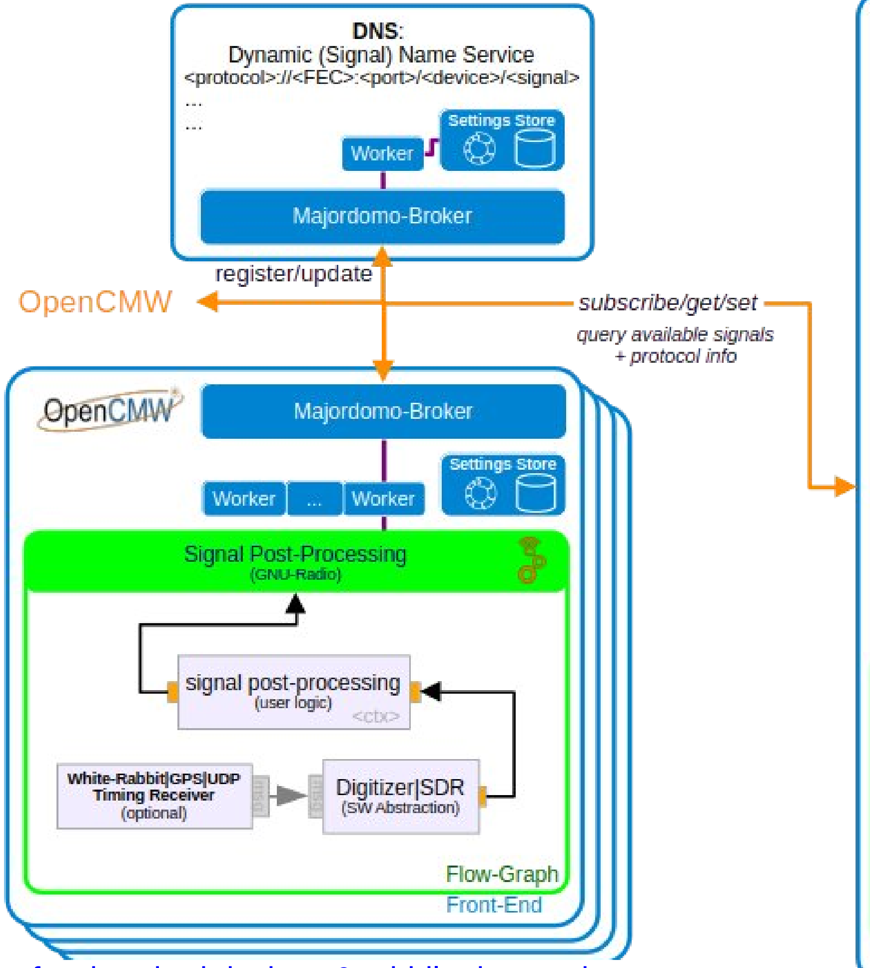
Experiments
(energy, position, particle-type, -angle, -spot size, -spill-rate, bunch-length, ...)

>100k diagnostics sensors

>1000 magnets, RF cavities, ...

Generic OpenDigitizer Integration @ GSI/FAIR

reimplementation: https://github.com/fair-acc/opendigitizer



DNS:
Dynamic (Signal) Name Service
<protocol>://<FEC>:<port>/<device>/<signal>
...
...

Worker

Settings Store

Majordomo-Broker

register/update

OpenCMW

subscribe/get/set

query available signals
+ protocol info

OpenCMW

Majordomo-Broker

Worker | ... | Worker

Settings Store

Signal Post-Processing
(GNU-Radio)

signal post-processing
(user logic)
<ctx>

White-Rabbit|GPS|UDP
Timing Receiver
(optional)

Digitizer|SDR
(SW Abstraction)

Flow-Graph

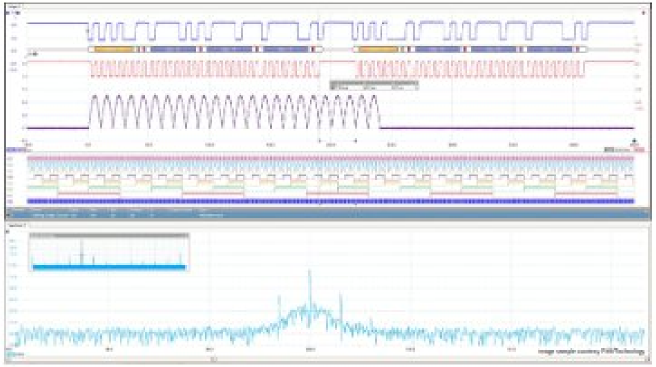
Front-End

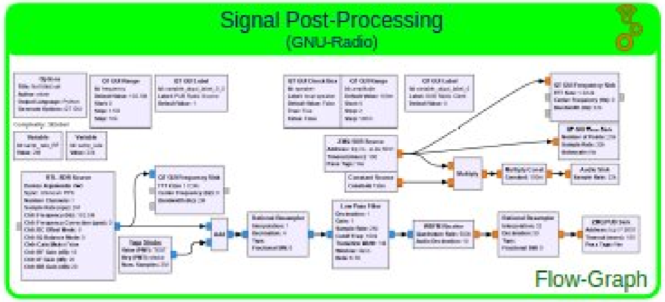
few hundred devices & middle-tier services

Digitizer Expert Application

(native & WASM-based)

- based on best-practices in existing test- and Instrumentation lab-type equipment
- modular composition
  - generic application (simple/pre-defined layout)
  - re-usable components → for custom apps/special layouts
- composition of signals from different digitizer FECs
  - configurable numerical post-/pre-processing





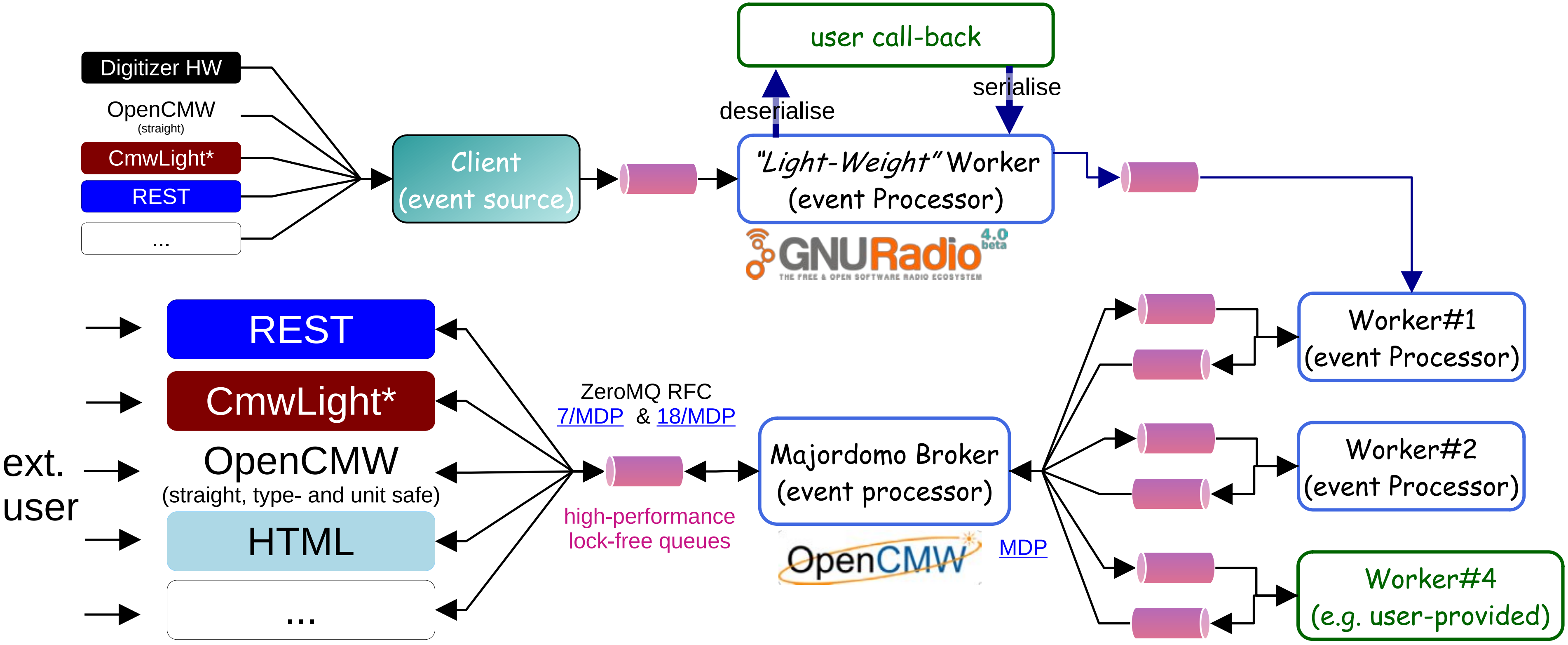
Signal Post-Processing
(GNU-Radio)

Flow-Graph

Expert GUI

Open Common Middle Ware – a RPC/Majordomo Implementation

https://github.com/fair-acc/opencmw-cpp



Digitizer HW
OpenCMW
(straight)
CmwLight*
REST
...

Client
(event source)

user call-back

deserialise

serialise

"Light-Weight" Worker
(event Processor)

REST
CmwLight*
OpenCMW
(straight, type- and unit safe)
HTML
...

ext.
user

ZeroMQ RFC
7/MDP & 18/MDP

high-performance
lock-free queues

Majordomo Broker
(event processor)

MDP

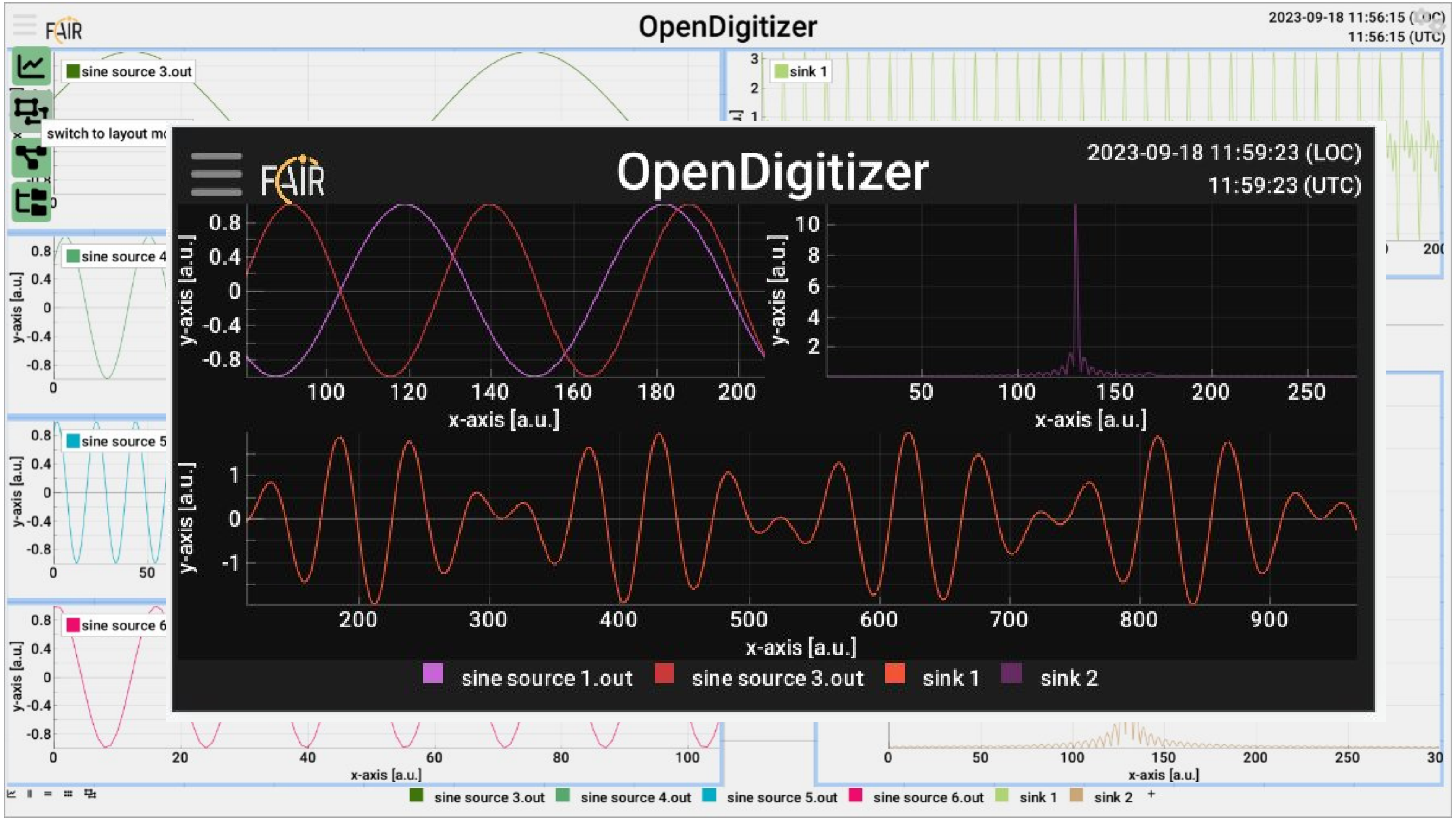
Worker#1
(event Processor)

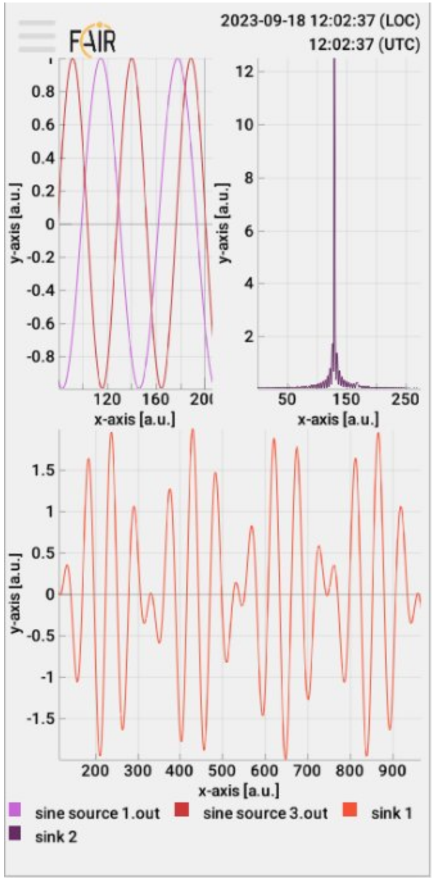
Worker#2
(event Processor)

Worker#4
(e.g. user-provided)

Generic OpenDigitizer Impressions

https://fair-acc.github.io/opendigitizer/





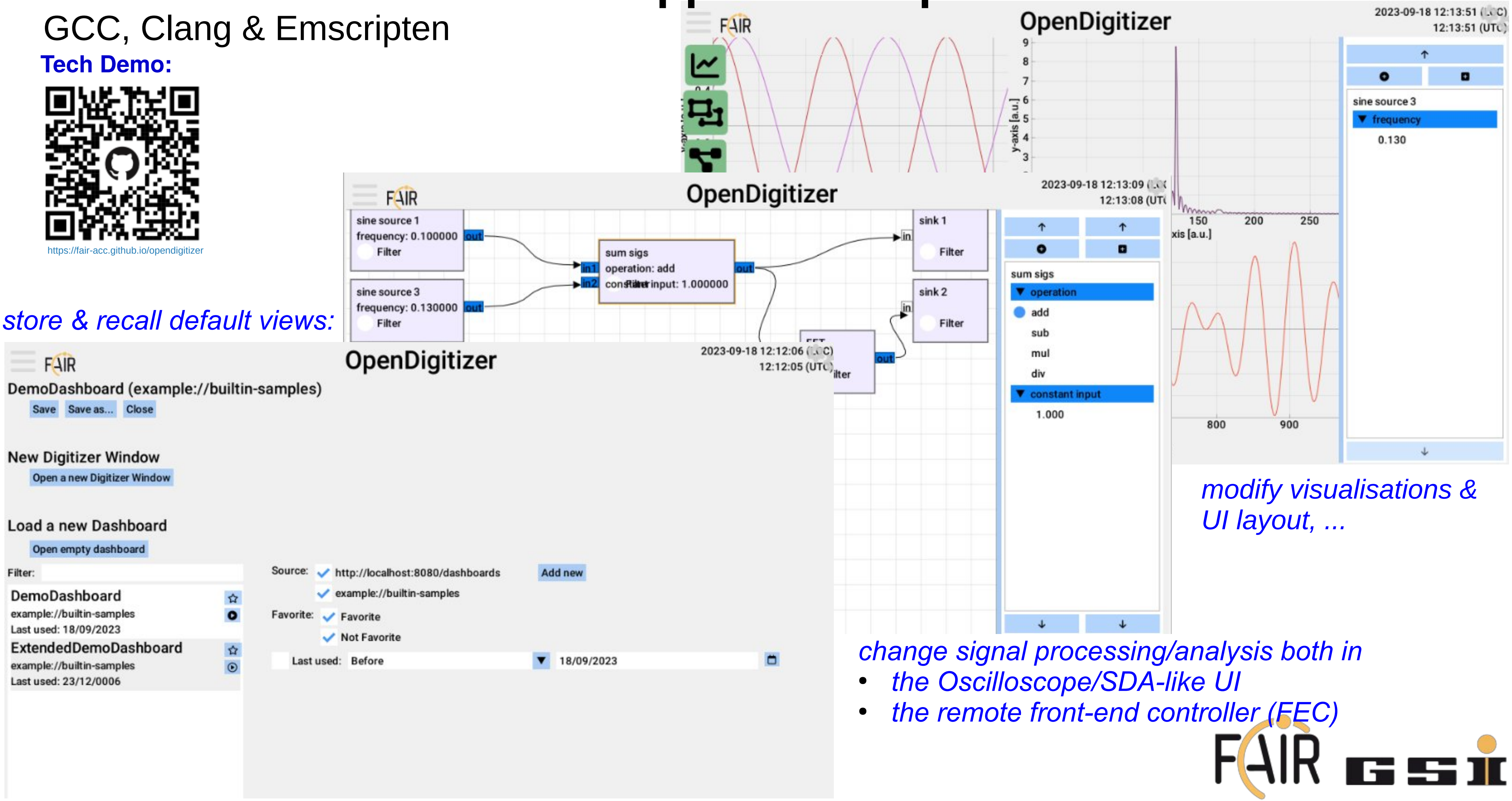
Broaden Cross-Platform Support: UI Impressions WIP

GCC, Clang & Emscripten

Tech Demo:

https://fair-acc.github.io/opendigitizer

store & recall default views:



OpenDigitizer

DemoDashboard (example://builtin-samples)

New Digitizer Window

Load a new Dashboard

Source: ✓ http://localhost:8080/dashboards
✓ example://builtin-samples

Favorite: ✓ Favorite
✓ Not Favorite

Last used: Before 18/09/2023

DemoDashboard
example://builtin-samples
Last used: 18/09/2023

ExtendedDemoDashboard
example://builtin-samples
Last used: 23/12/0006

sine source 1
frequency: 0.100000

sine source 3
frequency: 0.130000

sum sigs
operation: add
constant input: 1.000000

sum sigs
▼ operation
● add
sub
mul
div
▼ constant input
1.000

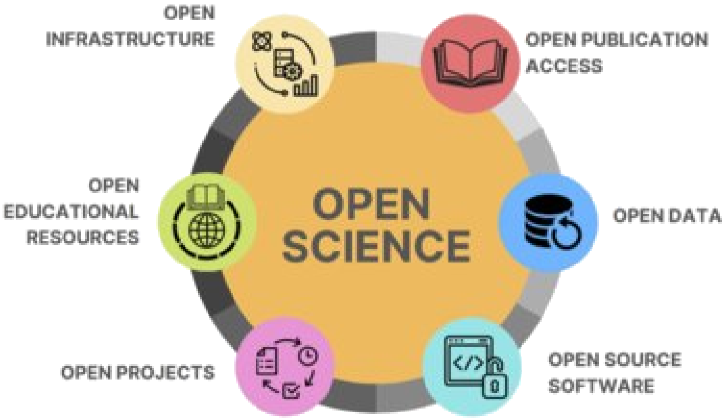
sine source 3
▼ frequency
0.130

modify visualisations & UI layout, ...

change signal processing/analysis both in
- *the Oscilloscope/SDA-like UI*
- *the remote front-end controller (FEC)*

Why we invest into GNU Radio 4.0$^{\text{beta}}$

https://www.gsi.de/en/work/forschung/open-science



- GSI/FAIR uses, supports, and promotes F.A.I.R. standards, Open-Science and FLOSS

- contributions tackle our specific in-house use-case:
  - high-performance, continuous & event-based signal- and data-processing enabling high-level beam-based diagnostics and feedback control loops
  - multi-user, multi-mission parallel operation in a large physically distributed environment
  - easier onboarding & more flexibility during commissioning and operation
  - visual flow-graphs: bridging the technology gap between experts with the required domain- and those with the necessary RSE-expertise (C++, 24h/7 operation, …)
    - notably: enables more staff & users to meaningful contribute to GSI/FAIR

- share technology with community, public organisations, and industry
  - avoid ‘re-inventing the wheel’ – co-invest and share maintenance efforts
  - invest into open-standards, vendor neutral interfaces, common infrastructure
  - building up competencies ↔ helps retaining/hiring/attracting new talents

Public Money
Public Code
fsfe.org

GNU Radio Timeline

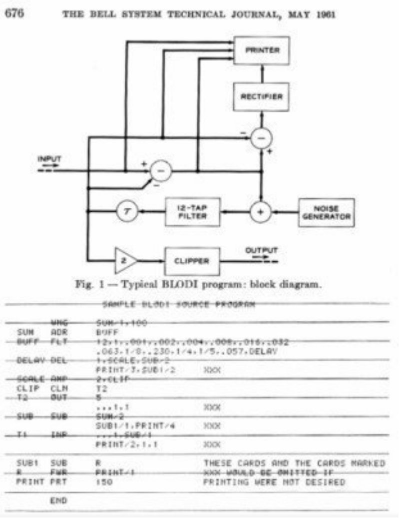
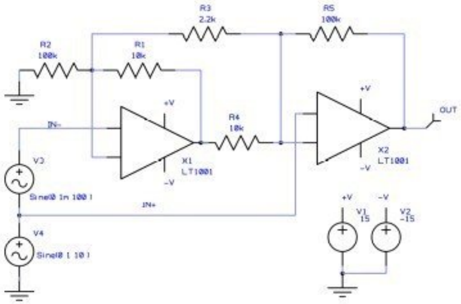
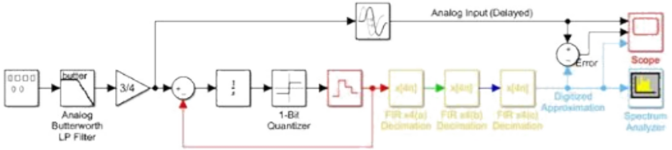
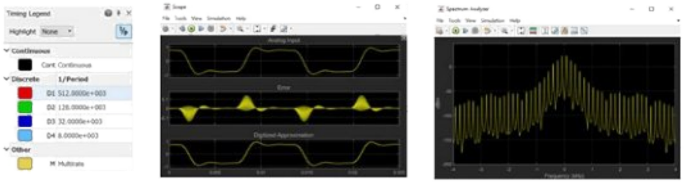

676 THE BELL SYSTEM TECHNICAL JOURNAL, MAY 1961

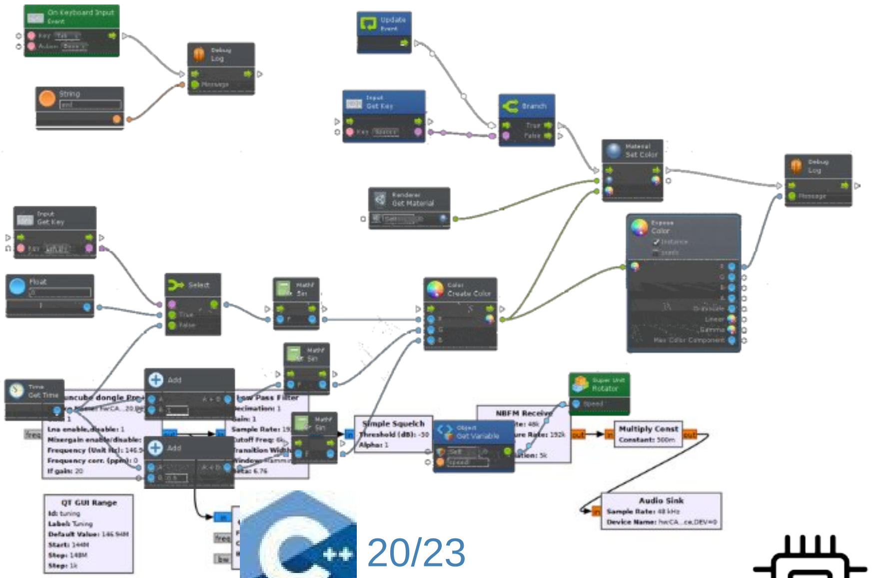
Fig. 1 — Typical BLODI program: block diagram.

```
                 SAMPLE BLODI SOURCE PROGRAM

        WNG     SUM-1,100
SUM     ADR     BUFF
BUFF    FLT     12,1,.001,.002,.004,.008,.016,.032
                .063,1/8,.230,1/4,1/5,.057,DELAY
DELAY   DEL     1,SCALE,SUB/2
                PRINT/3,SUB1/2        XXX
SCALE   AMP     2,CLIP
CLIP    CLM     T2
T2      OUT     5
                ...1,1                XXX
SUB     SUB     SUM/2
                SUB1/1,PRINT/4        XXX
T1      INP     ...1,SUB/1
                PRINT/2,1,1           XXX

SUB1    SUB     R                     THESE CARDS AND THE CARDS MARKED
R       FWR     PRINT/1               XXX WOULD BE OMITTED IF
PRINT   PRT     150                   PRINTING WERE NOT DESIRED

        END
```









20/23

performance, safety & design improvements

BLODI
(BLockDIagram)

1961

BELL SYSTEM

SPICE

1973

UC Berkeley

SIMULINK®

1984

MathWorks®

LabVIEW

1986

NATIONAL INSTRUMENTS™

SIMATIC S7 GRAPH
(PLCs)

1990ies

SIEMENS

GNU Radio
THE FREE & OPEN SOFTWARE RADIO ECOSYSTEM

2000

NSF

GPLv3 Free Software
Free as in Freedom

DARPA

&
vendor-neutral

~2006

unity

4.0^{beta}

2024

GPLv3 Free Software
Free as in Freedom

LGPLv3 Free Software
Free as in Freedom

Graph-Based Signal-Flow Description – Mechanical Sympathy

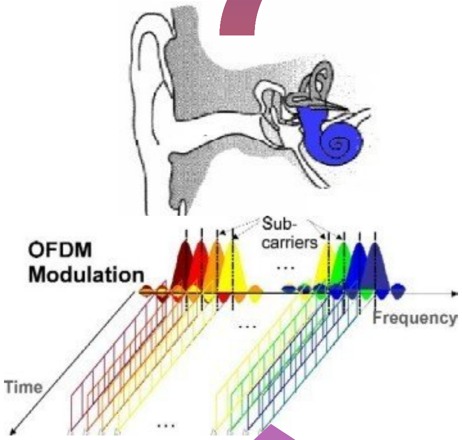
Intuitive Common Denominator for Education ...

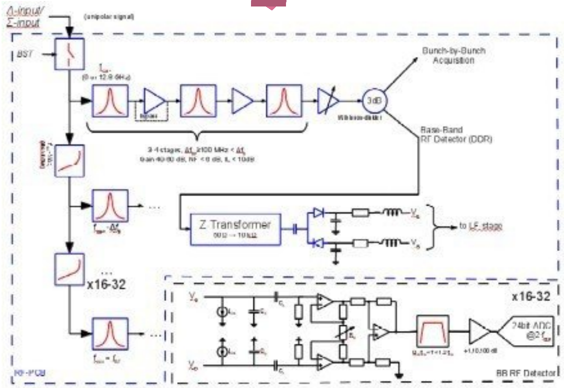
GNU Radio 4.0 goal:
remove barriers for wider public adoption

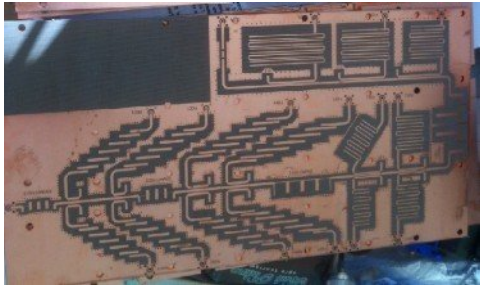
- better performance & functionalities
- dependable 24h/7 operation capabilities

Design + Simulate

HW/SW Design







keep concept
redesign SW

validate/
improve

Idea

Concept

Prototype

Industrial/Operational
Deployment

Software-Defined-Radio (SDR) trend: flexibility improved by shifting more-and-more functionality from HW → compile-time SW → runtime SW

Why we invest into GNU Radio 4.0$^{\text{beta}}$

Implementing Lean-, Clean- and Secure Coding Practices

… should be self-evident … but often isn't.

Challenges and Risks:

- **Total Cost of Ownership:**
  - large, non-secure codebases increase costs and risks.
  - challenges with flexibility, onboarding, and adapting to changes.
- **Meeting Security and Compliance Requirements:**
  - technical debt, negligence, and non-compliance leads to increased attack surfaces, performance loss, missing requirements, and public funding ↔ regulatory compliance with W.H./EU/national Cybersecurity Acts… is key!

Mitigation Strategies & Opportunities:

1. continuous improvement & keeping it 'lean' and 'clean': collectively refine code to minimise and address new vulnerabilities.
2. improve the 'bus factor' through extending usability, sharing maintenance across a wider community, public organisation and industry
3. promote GNU Radio as a Free and Open Industry Standard → LGPLv3

https://spectrum.ieee.org/lean-software-development

… for GR to be used in critical/public infrastructure or any real-world applications, it must meet higher standards for safety, cybersecurity, and product liability.


Derek Kozel


Josh Mormon

CONNECT
SHARE
COLLABORATE

In mid-2002 … gr-digitizer

- … GNU Radio 3.7 @ GSI/FAIR – Why? How?
  - Instantiating GR flow-graphs using C++-only?
  - Moving event-based processing RxCpp → GR3.X?
    - <discussion on buffer limitations, ...>
- *<… more discussions, inserting each other’s expertises)…>*
- **Proposal to address the core GR3.X ‘pain-points’ together:**
  - A) buffer- and compute-related performance improvements (SIMD)
  - B) making asynchronous event-based signal processing a first-class citizen of GR (↔ packet-radios)
  - C) enabling user-customisable schedulers that can be optimised for e.g. throughput, latency, hardware resources, ...)
  - D) easier integration of vendor-neutral heterogeneous, distributed and embedded computing, and
  - E) helping making GNU Radio safer, leaner, cleaner, and easier to use for core- and out-of-tree developers using modern C++ standards.
    → focus on: industrial 24h/7 deployment, safety, cybersecurity, long-term maintainability, compliance, ...

Community & Adoption

Technical Relevance & Functional Performance

Reliance on ‘true’ industry standards

Reduce Cognitive Complexity

Reduce Learning Curve

while promoting …

CONNECT
SHARE
COLLABORATE

GNU Radio 4.0 Modernisation Goals

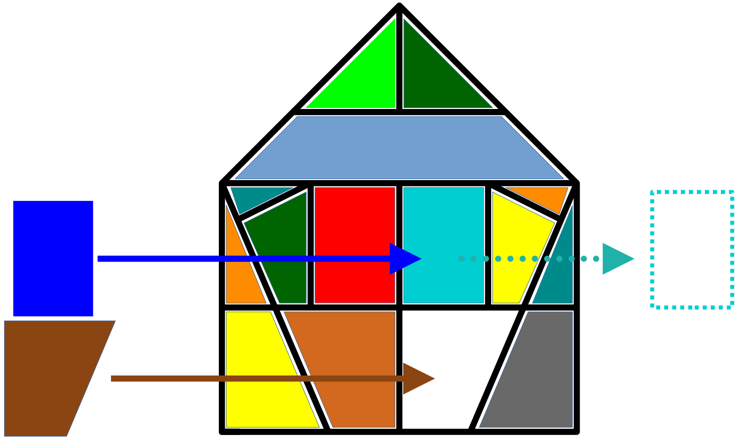
simplify onboarding for new contributors to participate/contribute more effectively

Outline: addressing the core GR3.X ‘pain-points’ together

A) buffer- and compute-related performance improvements (SIMD)
B) making asynchronous event-based signal processing a first-class citizen of GR (↔ packet-radios)
C) enabling user-customisable schedulers that can be optimised for e.g. throughput, latency, hardware resources, ...)
D) easier integration of vendor-neutral heterogeneous, distributed and embedded computing, and
E) helping making GNU Radio safer, leaner, cleaner, and easier to use for core- and out-of-tree developers using modern C++ standards.
→ focus on: industrial 24h/7 deployment, safety, cybersecurity, long-term maintainability, compliance, ...

GNU Radio 4: clean- and lean C++ code-base redesign

favours ‘composition’ over ‘inheritance’



traditional (prescriptive) frameworks:
- user implements stubs
- limited options to exchange or to extend

modular library:
- **performance & functionality focused!**
- C++’s only-pay-for-what-you-use paradigm
- free to extend, modify, synthesis new ideas
- compatible with public infrastructure, safety, security, and industrial use.

Anatomy of a GR4 Block

sharpened interfaces & stream-lined compositional implementation

(opt) **Streaming Input `PortIn<T>`**

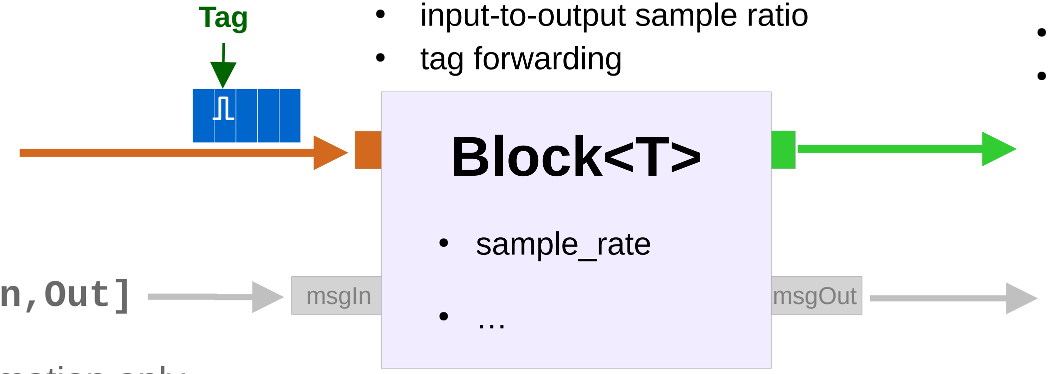
- always synchronous, with type T:
  - fundamental (integers, floats)
  - structured data (structs, classes)
- may have '**Tag**'s
  - map-type information
    - trigger,
    - new settings changes
    - …

Block<T>

- pure or stateful signal/data processing function
- settings expressed as
  - class members, and/or
  - map-type information
    I.e.{{"key1", value1}, {"key2", value2}, … }
- ensures constraints
  - input-to-output sample ratio
  - tag forwarding

(opt) **Streaming `PortOut<T>`**

- mirrors `PortIn<T>`
- **not necessarily 1:1 sample ration**
  - decimation, interpolation, …
  - also dynamic ratio, <u>including</u>
    - do not consume input
    - do not produce output
    - produce output w/o input



(opt) **`MsgPort[In,Out]`**

- asynchronous
- map-type information only
  i.e. {{"key1", value1}, {"key2", value2}, … }
- can be used to update settings
  (e.g. via RPC/UI)
- ↔ GR'3 aka. 'control port'

Modern and much Simpler C++ User-API

Code is single Source of Truth – easier to reason & maintain

```cpp


struct BasicMultiplier : public Block<BasicMultiplier> {
  InPort<float>  in;
  OutPort<float> out;
  float          scaling_factor = static_cast<float>(1); // comment



  constexpr float processOne(const float &a) const noexcept {
      return a * scaling_factor;
  }
};
```

Key Take-Aways:

- **Simplified Block Development**: stand-alone creation is more intuitive. Feedback? Let's discuss!
- **Efficient Functional Unit Testing**: directly test blocks without embedding in flow-graphs
  - offer three basic (optional) API variants: sample-by-sample, chunked, or arbitrary processing (i.e. ‘`work(…)`’) function
- **Compiler-Optimised Interface**: Type-strictness and constraints help w.r.t. efficient compiler optimisations
- **Early Error Detection**: most issues caught during compile time, reducing errors and debugging during run-time

Modern and much Simpler C++ User-API

Intrinsic SIMD support using `processOne(…)` API

```cpp
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public Block<BasicMultiplier<T>> {
  InPort<float>  in;
  OutPort<float> out;
  float          scaling_factor = static_cast<float>(1); // comment


  template<t_or_simd<T> V> // → intrinsic SIMD support
  constexpr V processOne(const V &a) const noexcept {
      return a * scaling_factor;
  }
};
```

Key Take-Aways:

- **Simplified Block Development**: stand-alone creation is more intuitive. Feedback? Let's discuss!
- **Efficient Functional Unit Testing**: directly test blocks without embedding in flow-graphs
  - offer three basic (optional) API variants: sample-by-sample, chunked, or arbitrary processing (i.e. ‘`work(…)`’) function
- **Compiler-Optimised Interface**: Type-strictness and constraints help w.r.t. efficient compiler optimisations
- **Early Error Detection**: most issues caught during compile time, reducing errors and debugging during run-time

Modern and much Simpler C++ User-API

Complementary `processBulk(…)` API I/II

```cpp
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public Block<BasicMultiplier<T>> {
  InPort<float>  in;
  OutPort<float> out;
  float          scaling_factor = static_cast<float>(1); // comment


  // complementary interface, e.g. for first-class resampling
  gr::work::Status processBulk(std::span<const T> in, std::span<T> out) {
       std::ranges::transform(in, out.begin(), [sf = scaling_factor](const T& val) {
          return val * sf;
       });
       return gr::work::Status::OK;
  }
};
```

Fun Fact (aka. beware of ‘premature optimisations’):

Benchmarking proved that using ‘`processOne(…)`’ is numerically more performant than ‘`processBulk(…)`’

rationale: locality, reduced scope that can be better exploited by the compiler and L1/L2/L3 CPU cache.

Modern and much Simpler C++ User-API

Complementary `processBulk(…)` API I/II

```
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public Block<BasicMultiplier<T>, Resampling</*N_IN, M_OUT*/>>> {
  InPort<float>  in;
  OutPort<float> out;
  float          scaling_factor = static_cast<float>(1); // comment


  // complementary interface, e.g. for first-class (arbitrary) resampling
  gr::work::Status processBulk(ConsumableSpan auto& in, PublishableSpan auto& out) const noexcept {
      // [..] user-defined processing logic [..]
      in.consume(3UL);  // consume N_IN  = 3 samples
      out.publish(2UL); // publish M_OUT = 2 samples → effectively a 3:2 re-sampler
      return gr::work::Status::OK;
  }
};
```

Fun Fact (aka. beware of ‘premature optimisations’):

Benchmarking proved that using ‘`processOne(…)`’ is numerically more performant than ‘`processBulk(…)`’

rationale: locality, reduced scope that can be better exploited by the compiler and L1/L2/L3 CPU cache.

Modern and much Simpler C++ User-API

runtime polymorphism: built-in GRC/YAML-style flow-graph support → enables UI/UX, Python, …

```yaml
# GRC/YAML-style graph definition
blocks:
  - name: main_source
    id: good::fixed_source
    parameters:
      event_count: 100
      unknown_property: 42
  - name: multiplier
    id: good::multiply
  - name: counter
    id: builtin_counter
  - name: sink
    id: good::cout_sink
    parameters:
      total_count: 100
      unknown_property: 42
connections:
  - [main_source, 0, multiplier, 0]
  - [multiplier, 0, counter, 0]
  - [counter, 0, sink, 0]
```

```cpp
using namespace gr;

try {
  // load plugin library (stored as .so)
  std::vector<std::filesystem::path> paths = /*...*/;
  gr::PluginLoader plugins(gr::globalBlockRegistry(),
                std::move(paths))

  // load/save GRC-style graph descriptions
  std::string grcGraph = /* … grc-style yaml file */
  gr::Graph   graph = gr::loadGrc(plugins, grcGraph);
  std::string savedGrcData = gr::saveGrc(graph);

  gr::scheduler::Simple scheduler(std::move(graph));
     expect(scheduler.runAndWait().has_value());
} catch (const gr::exception& e) {
  fmt::println(std::cerr, "unexpected exception: {}", e);
  // handle grc load/save error
}
```

→ UI/UX & Python-Bindings need community driven-integration

Modern and much Simpler C++ User-API

C++(Python)-Block PoC Prototype

vs.

this PoC

eg. GRC-like

```python
import time;
counter = 0

def process_bulk(ins, outs):
    global counter

    # Print current settings
    settings = this_block.getSettings()
    print("Current settings:", settings)

    # tag handling
    if this_block.tagAvailable():
        tag = this_block.getTag()
        print('Tag:', tag)

    counter += 1
    # process the input->output samples
    for i in range(len(ins)):
        outs[i][:] = ins[i] * 2

    # Update settings with the counter
    settings["counter"] = str(counter)
    this_block.setSettings(settings)
```

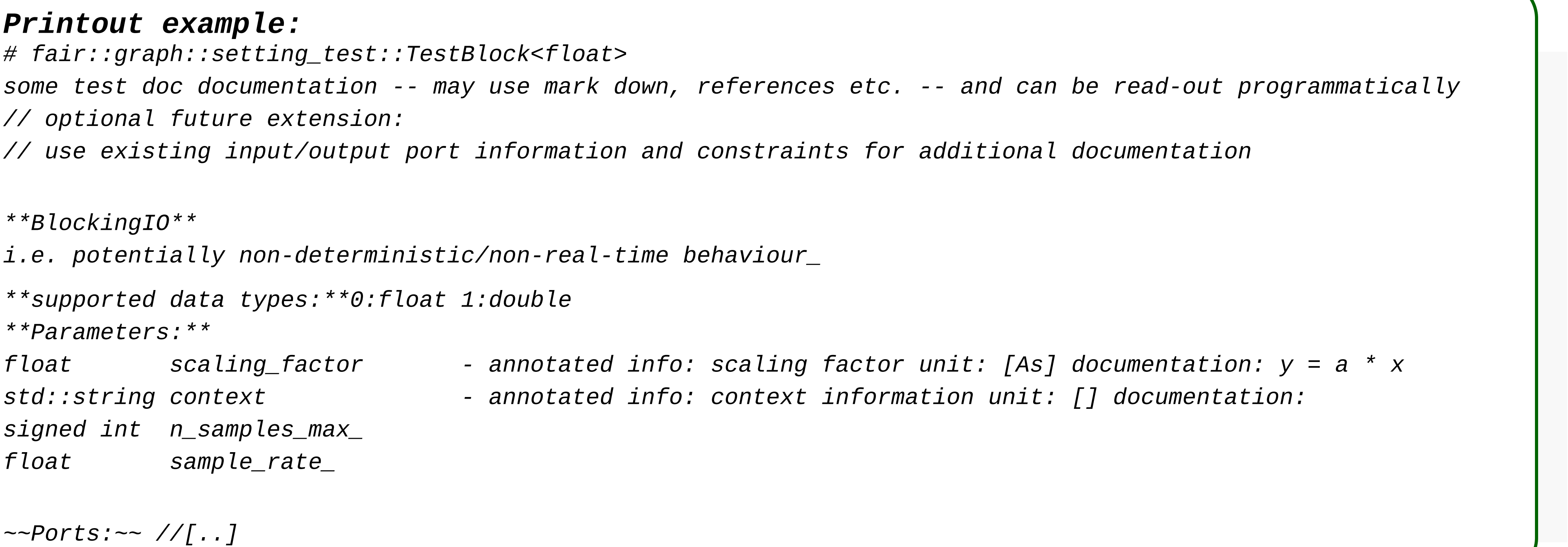
```cpp
using namespace gr;

Graph graph;
auto& src    = graph.emplaceBlock<TagSource<float>>(…);
auto& block = graph.emplaceBlock<PythonBlock<float>>({
                     { "n_inputs", 1U},
                     {"n_outputs", 1U},
                     { "pythonScript", pythonScript}});
auto& sink  = graph.emplaceBlock<TagSink<float>(…);

graph.connect(src, "out"s, block, "inputs#0"s);
graph.connect(block, "outputs#0"s, sink, "in"s);

scheduler::Simple sched{std::move(graph)};

try {
  expect(scheduler.runAndWait().has_value());
} catch (const gr::exception& e) {
  // handle grc load/save error
}
```

→ not feature-complete, but could be expanded upon by community

Modern and much Simpler C++ User-API

C++ compile-time reflection: Code is single Source of Truth

… can be used to generate Python bindings, code & UI documentation, provide UI meta information, further static reflection options, etc.



```
Printout example:
# fair::graph::setting_test::TestBlock<float>
some test doc documentation -- may use mark down, references etc. -- and can be read-out programmatically
// optional future extension:
// use existing input/output port information and constraints for additional documentation


**BlockingIO**
i.e. potentially non-deterministic/non-real-time behaviour_
**supported data types:**0:float 1:double
**Parameters:**
float       scaling_factor       - annotated info: scaling factor unit: [As] documentation: y = a * x
std::string context              - annotated info: context information unit: [] documentation:
signed int  n_samples_max_
float       sample_rate_


~~Ports:~~ //[..]
```

No additional DSL to learn for users!

FAIR The Universe in the Laboratory

European GNURadio Days 2024
27th - 30th August
Darmstadt, Germany +


Jean-Michel Friedt

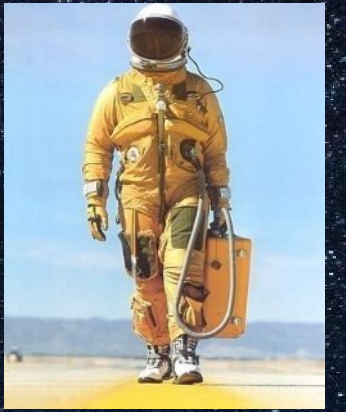

Dr. Ivan Čukić


Dr. Matthias Kretz


Alexander Krimm


Dr. Semën Lebedev

std::simd



Tutorials on Indico & YouTube

femto-st xlim HELMHOLTZ citi lab AKDAB GSI FAIR

Outline: addressing the core GR3.X 'pain-points' together

A) buffer- and compute-related performance improvements (SIMD)

B) making asynchronous event-based signal processing a first-class citizen of GR (↔ packet-radios)

C) enabling user-customisable schedulers that can be optimised for e.g. throughput, latency, hardware resources, ...)

D) easier integration of vendor-neutral heterogeneous, distributed and embedded computing, and

E) helping making GNU Radio safer, leaner, cleaner, and easier to use for core- and out-of-tree developers using modern C++ standards.
→ focus on: industrial 24h/7 deployment, safety, cybersecurity, long-term maintainability, compliance, ...

Graph-Based Signal-Processing – ‘Mechanical Sympathy’

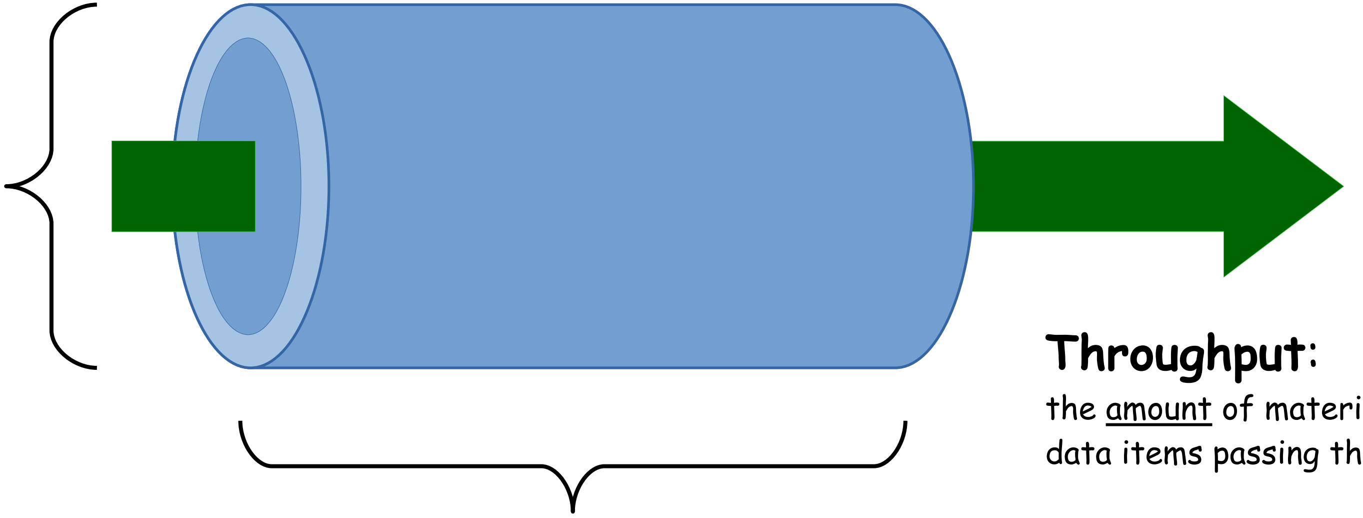
"You don't have to be an engineer to be be a racing driver, but you do have to have Mechanical Sympathy." Jackie Stewart*

more general: *“understand and care for how the machine you are working on itself works, to be able to get best performance out of the system”*

*'Mechanical Sympathy' by Jackie Stewart (aka. the 'Flying Scotsman') source AMD

img reference

Latency, Bandwidth & Throughput



Bandwidth:
maximum rate
of data transfer

Throughput:
the amount of material or
data items passing through

Latency:
delay between
the cause and the effect

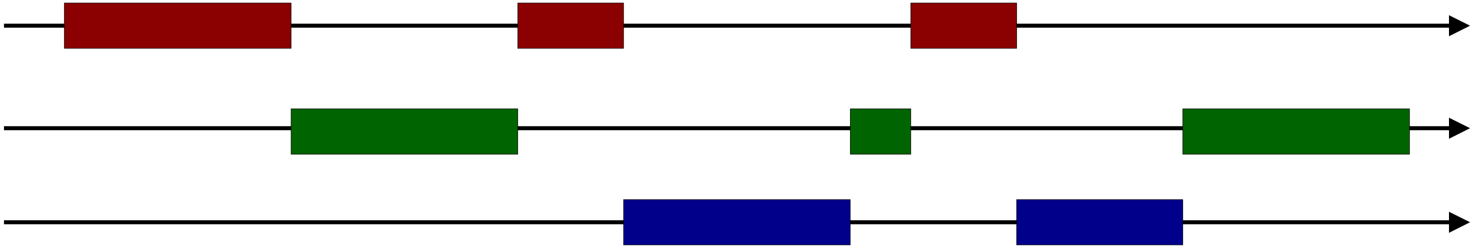
need lower latency →

a) maximise IO/memory bandwidth or compute efficiency and/or
b) minimise critical data & compute sections

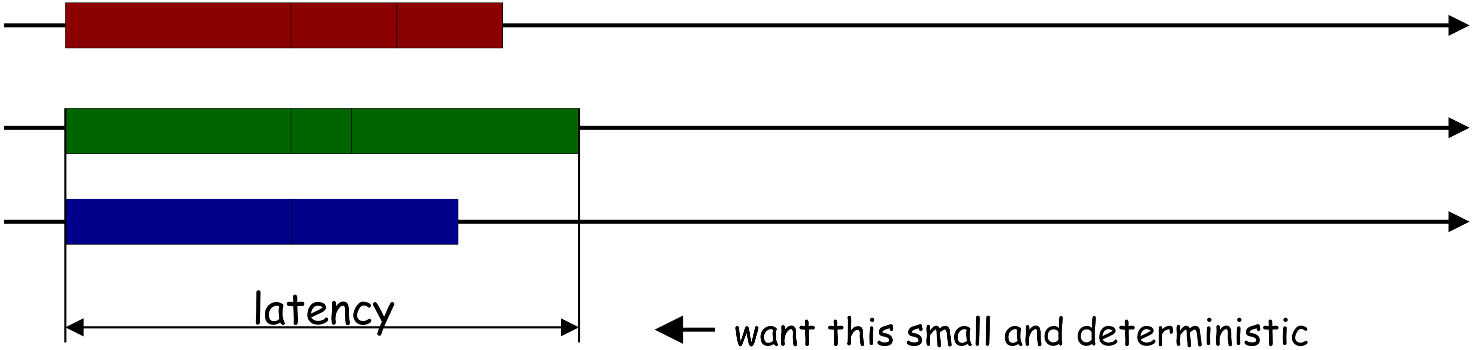
Concurrent & Parallel

Theory

- Concurrent, non-parallel execution:



- Concurrent, parallel execution:



← want this small and deterministic

excellent more in-depth talk by Gil Tene (Azul): "How NOT to Measure Latency", QCon 2012, pdf, video

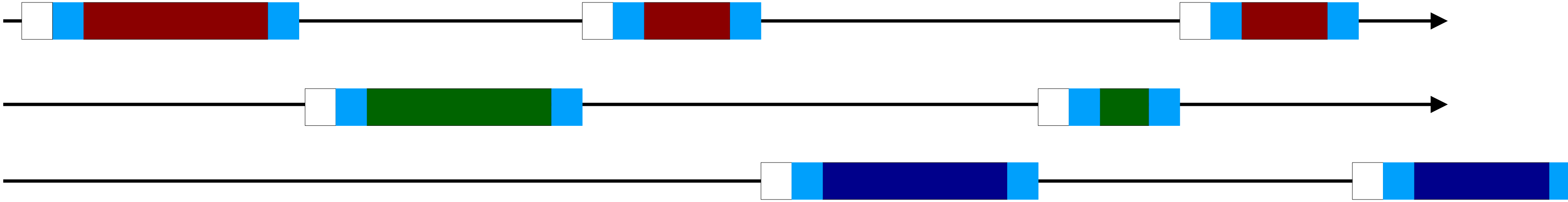
Concurrent & Parallel

Real-World Latencies

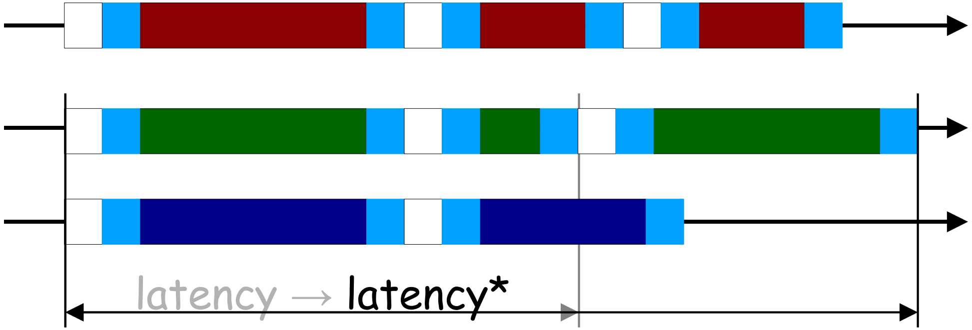
GR4 core goal – minimise these:

☐ context switch
🟦 systematic overhead

- Concurrent, non-parallel execution:



- Concurrent, parallel execution:



Examples for (Zen3 architecture):

1) **Context Switches**
    1) Thread: 1 → 10 µs
    2) Process: 5 → 20 µs
2) **Atomic Operations**:
    1) L1 cache: 10s of ns
    2) L1 to L2 Cache: 10 → 20 ns
    3) L1 to L3 Cache: 20 → 50 ns
    4) L1 to RAM: 60 → 100+ ns
3) **IO/ISR response**: 1 → 10 µs
4) **USB 3.x**: 100 µs → few ms
5) **Thread Core Migration**: 5 → 20 µs

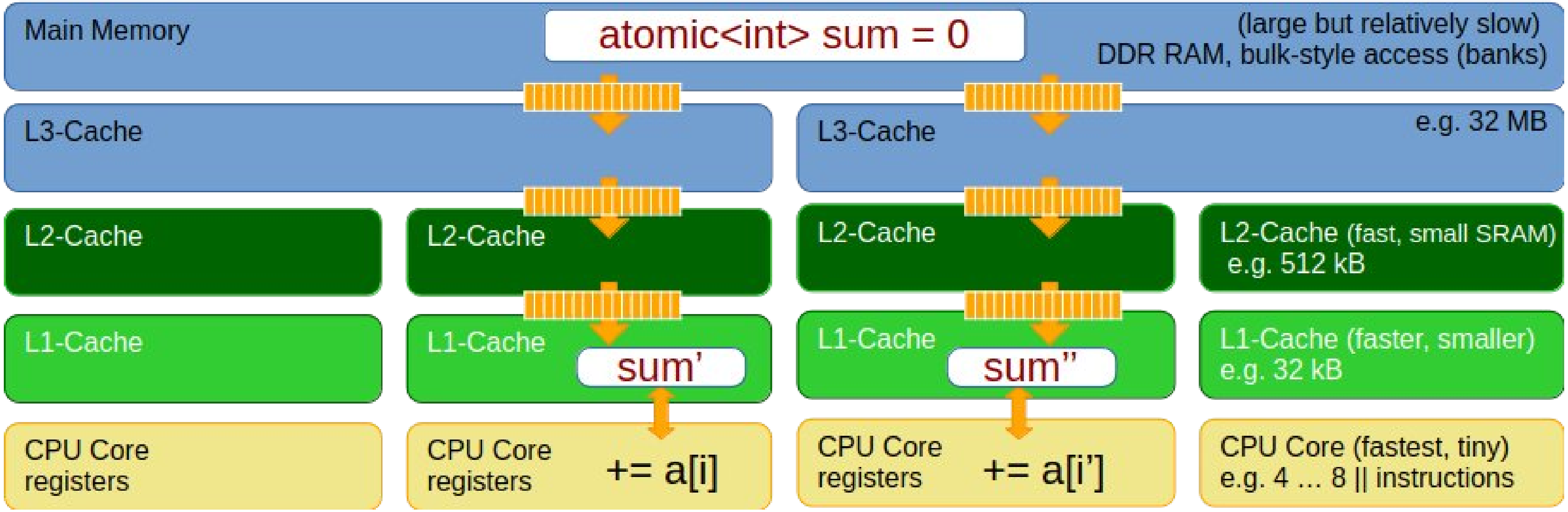
Classic Paradigm (GR3 et al):

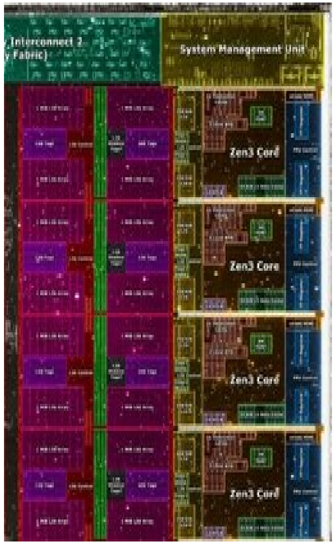
- **Pro**: maximise/increase actual useful work to diminish negative contributions due to latencies/context switches
- **Con: same strategy also increases latencies! :-(**

Performance through ‘Mechanical Sympathy’

Know your Hardware and your Programming Language

- synchronising across cores invalidates caches → slow reloads
  - keep memory/instructions local and small
  - avoiding *false-sharing* and *cache evictions*
  - modern C++: smaller more efficient code (minimise *virtual* calls)
    → cache evictions are less likely → better performance



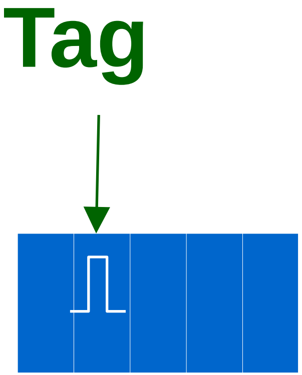


further details and discussions:

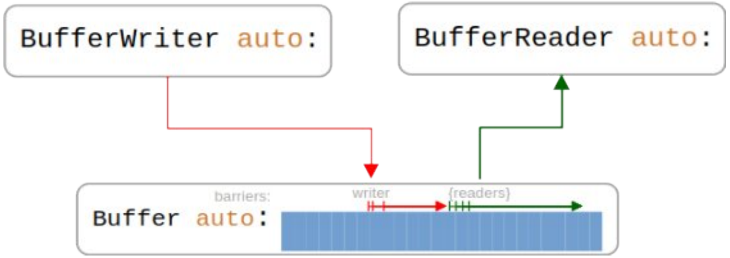
- 2022-03-09 C++ UG Meeting: https://indico.gsi.de/event/13919/

Performance through ‘Mechanical Sympathy’

Type-Strict High-Performance Lock-Free Circular Buffers I/II



- Follows classic reader-writer paradigm
  - `‘Buffer’` – conceptually as before, actual backing type-safe memory, RAII, ...

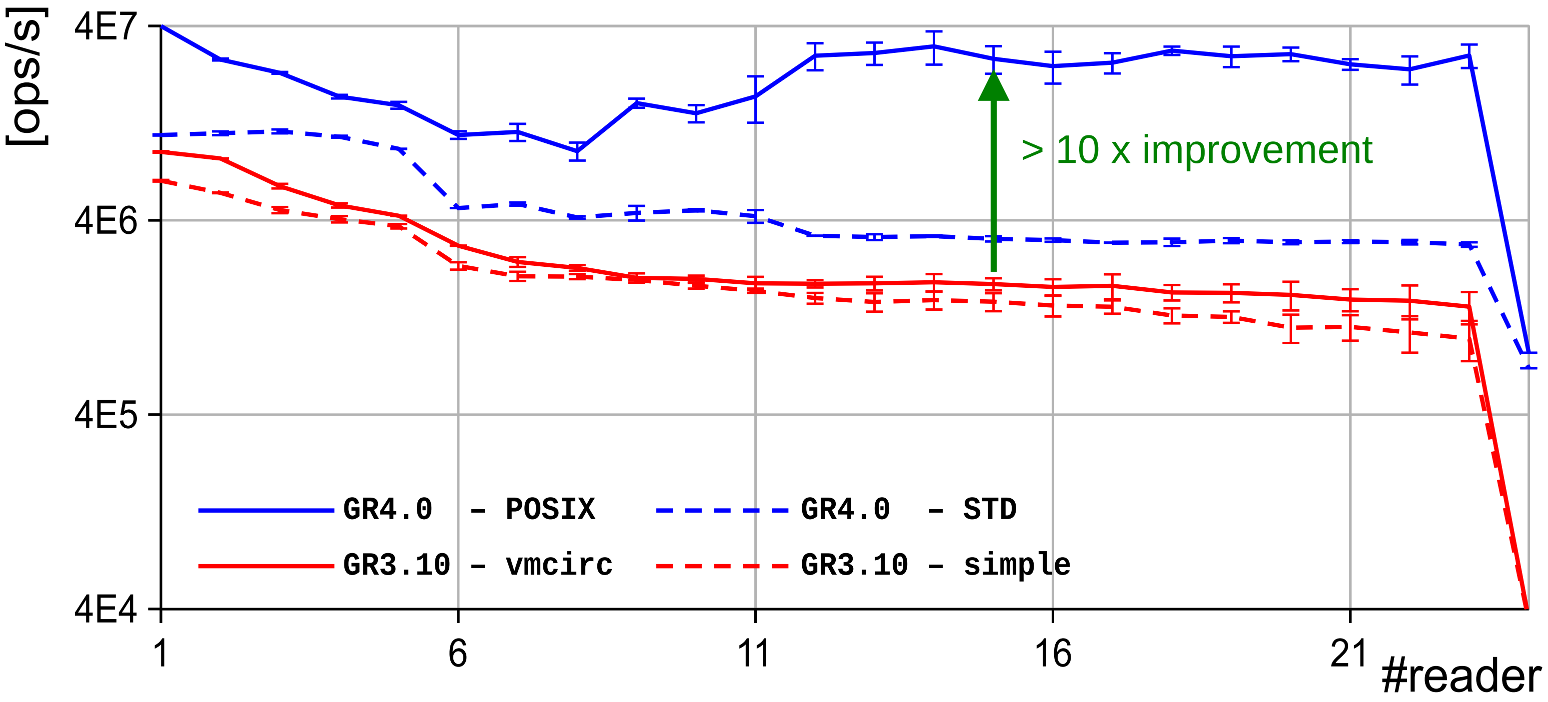


  - NEW FEATURE: can safely efficiently propagate both
    - unstructured safe data types, same as in GR3: `uint8_t, int16_t, …, float, double, std::complex<[float, double]>`
    - structured data types (aka. aggregates, structs, classes), e.g. `pmtv::map_t, std::vector<T>, …, Tag, Packet<T>, Tensor<T>, DataSet<T>,…`

LMAX Disruptor inspired: https://lmax-exchange.github.io/disruptor/
further details: 2022-03-09 C++ UG Meeting: https://indico.gsi.de/event/13919/

Performance through 'Mechanical Sympathy'

Type-Strict High-Performance Lock-Free Circular Buffers II/II



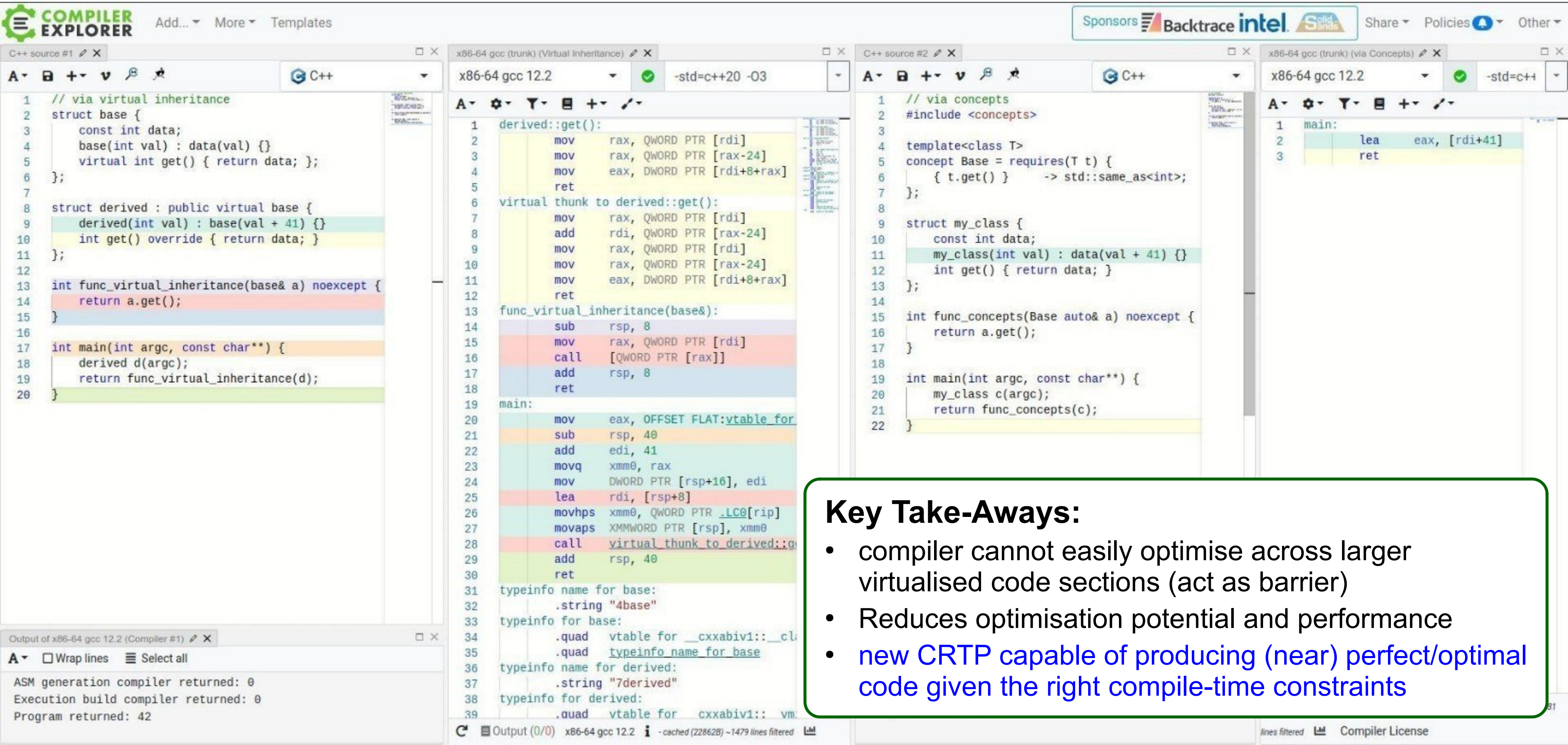
main key-ingredients:

- *made new CircularBuffer<T> lock-free (using atomic CAS paradigm)*
- *strict typing & `constexpr`*
  → enables better compiler optimisation and L1/L2/L3 cache locality

N.B. test scenario on equal footing
but absolute values could be improved
through better wait/scheduling strategies

SIMD & Merge-API

moving from virtual inheritance → strict typing, CRTP & concepts: https://compiler-explorer.com/z/fe5Khcxfv



```cpp
// via virtual inheritance
struct base {
    const int data;
    base(int val) : data(val) {}
    virtual int get() { return data; };
};

struct derived : public virtual base {
    derived(int val) : base(val + 41) {}
    int get() override { return data; }
};

int func_virtual_inheritance(base& a) noexcept {
    return a.get();
}

int main(int argc, const char**) {
    derived d(argc);
    return func_virtual_inheritance(d);
}
```

```asm
derived::get():
        mov     rax, QWORD PTR [rdi]
        mov     rax, QWORD PTR [rax-24]
        mov     eax, DWORD PTR [rdi+8+rax]
        ret
virtual thunk to derived::get():
        mov     rax, QWORD PTR [rdi]
        add     rdi, QWORD PTR [rax-24]
        mov     rax, QWORD PTR [rdi]
        mov     rax, QWORD PTR [rax-24]
        mov     eax, DWORD PTR [rdi+8+rax]
        ret
func_virtual_inheritance(base&):
        sub     rsp, 8
        mov     rax, QWORD PTR [rdi]
        call    [QWORD PTR [rax]]
        add     rsp, 8
        ret
main:
        mov     eax, OFFSET FLAT:vtable for
        sub     rsp, 40
        add     edi, 41
        movq    xmm0, rax
        mov     DWORD PTR [rsp+16], edi
        lea     rdi, [rsp+8]
        movhps  xmm0, QWORD PTR .LC0[rip]
        movaps  XMMWORD PTR [rsp], xmm0
        call    virtual_thunk_to_derived::g
        add     rsp, 40
        ret
typeinfo name for base:
        .string "4base"
typeinfo for base:
        .quad   vtable for __cxxabiv1::__cl
        .quad   typeinfo name for base
typeinfo name for derived:
        .string "7derived"
typeinfo for derived:
        .quad   vtable for __cxxabiv1::__vm
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 42
```

```cpp
// via concepts
#include <concepts>

template<class T>
concept Base = requires(T t) {
    { t.get() }     -> std::same_as<int>;
};

struct my_class {
    const int data;
    my_class(int val) : data(val + 41) {}
    int get() { return data; }
};

int func_concepts(Base auto& a) noexcept {
    return a.get();
}

int main(int argc, const char**) {
    my_class c(argc);
    return func_concepts(c);
}
```

```asm
main:
        lea     eax, [rdi+41]
        ret
```

Key Take-Aways:

- compiler cannot easily optimise across larger virtualised code sections (act as barrier)
- Reduces optimisation potential and performance
- new CRTP capable of producing (near) perfect/optimal code given the right compile-time constraints

SIMD & Merge-API

The best buffer performance is when no buffer is required

- `processOne(…)` (and later `processBulk(…)`) enable:
  - runtime merge:
    → omits `Buffer<T>` + atomics
  - compile-time merge:
    → omits `Buffer<T>` and facilitate larger scope compiler optimisations (i.e. `-O2` and `-O3`)
    → smaller memory footprint → more efficient use of
    - L1/L2/L3 caches
    - smaller code sizes → target to be able to run on a micro-processor (e.g. RP2040)
- Facilitates transition from flexible R&D prototype
  → production use that requires less flexibility and more performances

SIMD & Merge-API Performance Figures

out-of-the-box ‘Single Instruction, Multiple Data’ (SIMD) acceleration

```
┌────────────────────────────benchmark:─────────────────────────────┬──cache misses───┬──mean──┬─stddev─┬──max───┬─ops/s─┐
│ merged  src→sink                                                  │ 1.3k /   3k = 46% │ 626 ns │ 110 ns │ 952 ns │ 16.4G │
│ merged  src->copy->sink                                           │ 391 / 971  = 40% │ 957 ns │ 106 ns │   1 us │ 10.7G │
│ merged  src(N=1024)->b1(N≤128)->b2(N=1024)->b3(N=32...128)->sink  │ 398 / 960  = 41% │ 957 ns │ 103 ns │   1 us │ 10.7G │
│ merged src→mult(2.0)→divide(2.0)→add(-1)→sink                     │ 401 /   1k = 40% │   3 us │ 108 ns │   4 us │  3.0G │
│ merged  src->(mult(2.0)->div(2.0)->add(-1))^10->sink              │ 470 /   1k = 42% │  41 us │ 189 ns │  42 us │  248M │
│ runtime src->sink                                                 │  9k / 174k =  5% │  42 us │  98 us │ 336 us │  241M │
│ runtime src(N=1024)->b1(N≤128)->b2(N=1024)->b3(N=32...128)->sink  │ 20k / 648k =  3% │ 125 us │ 328 us │   1 ms │ 81.7M │
│ runtime src->mult(2.0)->div(2.0)->add(-1)->sink - processOne(..)  │ 24k / 663k =  4% │ 105 us │ 259 us │ 882 us │ 97.5M │
│ runtime src->mult(2.0)->div(2.0)->add(-1)->sink - processBulk(..) │ 24k / 664k =  4% │ 152 us │ 358 us │   1 ms │ 67.3M │
│ runtime src→(mult(2.0)→div(2.0)→add(-1))^10→sink                  │ 56k / 686k =  8% │ 127 us │  28 us │ 198 us │ 80.6M │
└───────────────────────────────────────────────────────────────────┴──────────────────┴────────┴────────┴────────┴───────┘
```

CPU: AMD Ryzen 9 5900X (Zen 3)

Key Take-Aways:

- SIMD provides another > 4 performance gain (or more depending on architecture)
- constexpr → enables block/graph compile-optimisation pushing performance to CPU hardware limit (& thermal throttling!!)
  - *side effect: code size can fit < few MBs → GR4 deployment on micro-controller, FPGAs, … (WIP)*
- beware of premature optimisations (PMOs)
  - e.g. `processOne(…)` vs. `processBulk(…)`

Outline: addressing the core GR3.X ‘pain-points’ together

A) buffer- and compute-related performance improvements (SIMD)

B) making asynchronous event-based signal processing a first-class citizen of GR (↔ packet-radios)

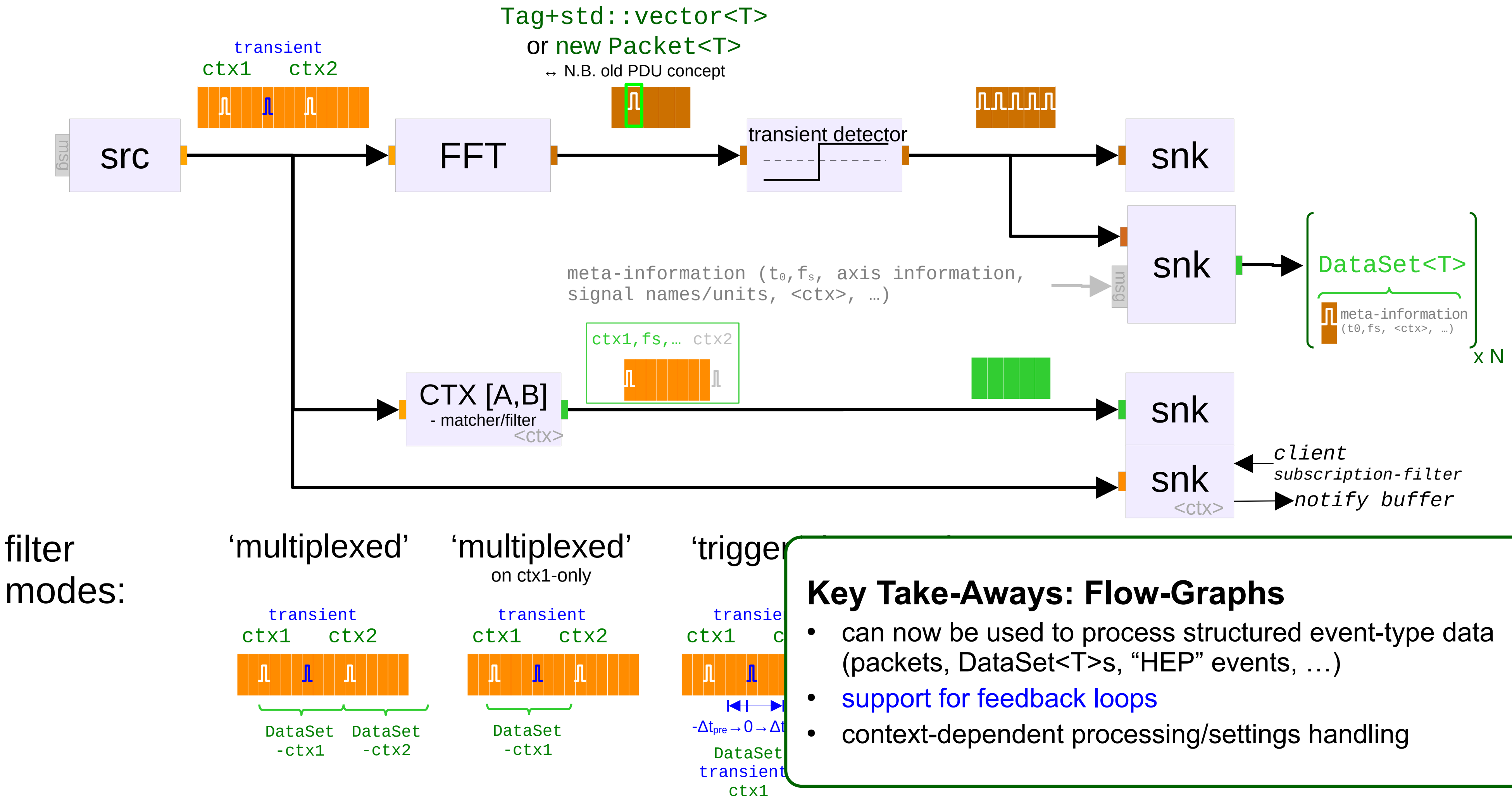
C) enabling user-customisable schedulers that can be optimised for e.g. throughput, latency, hardware resources, ...)

D) easier integration of vendor-neutral heterogeneous, distributed and embedded computing, and

E) helping making GNU Radio safer, leaner, cleaner, and easier to use for core- and out-of-tree developers using modern C++ standards.
→ focus on: industrial 24h/7 deployment, safety, cybersecurity, long-term maintainability, compliance, ...

Complex Data Types and Event-Style/Packet Processing

synchronous packet/event processing (HEP inspired) – new: `Packet<T>,Tensor<T>, DataSet<T>`

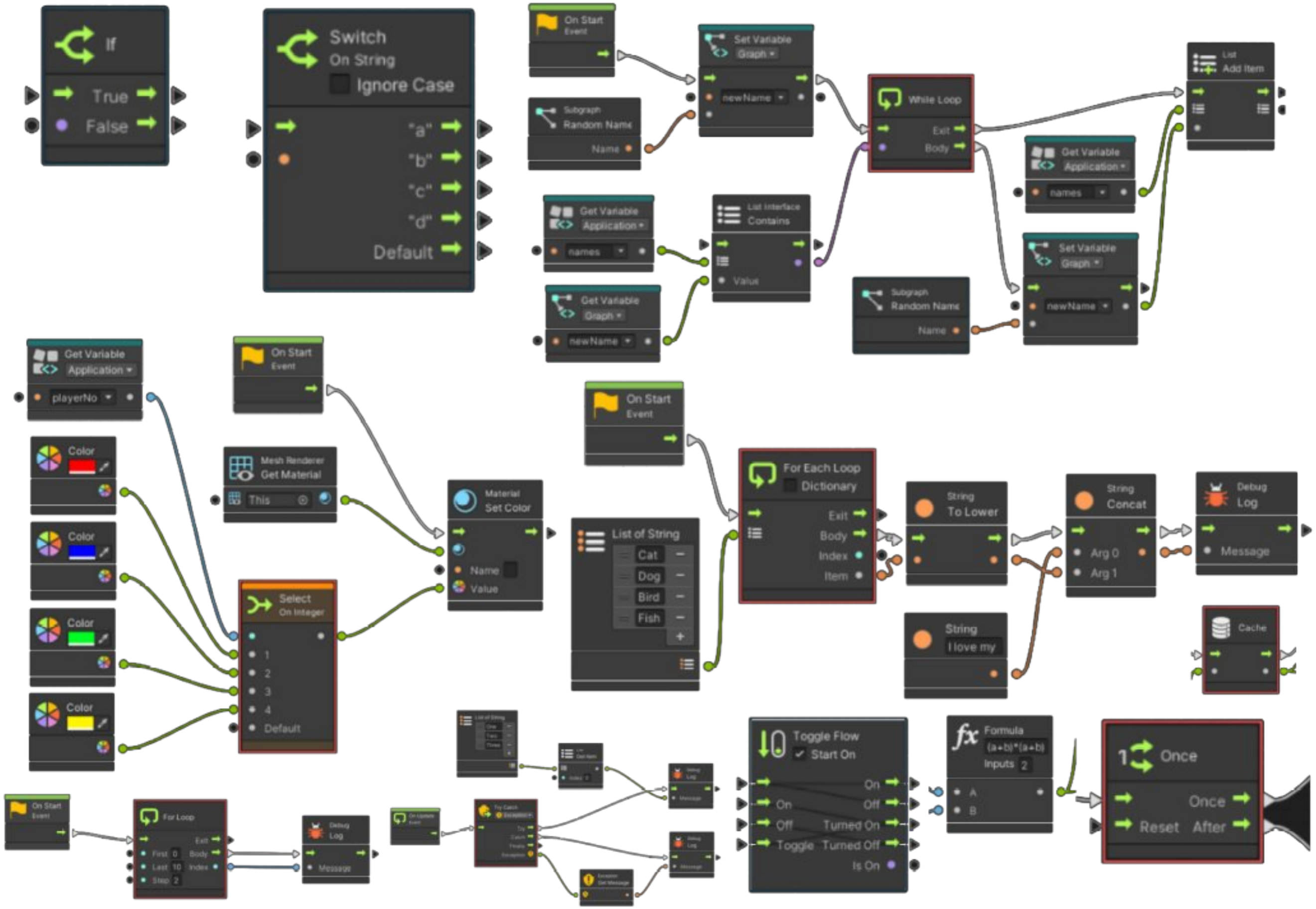


Key Take-Aways: Flow-Graphs

- can now be used to process structured event-type data (packets, DataSet<T>s, "HEP" events, …)
- support for feedback loops
- context-dependent processing/settings handling

Future Vision/Extension: Inspiration from the Gaming Industry

Basic Scripting of more complex signal flow/processing mechanisms

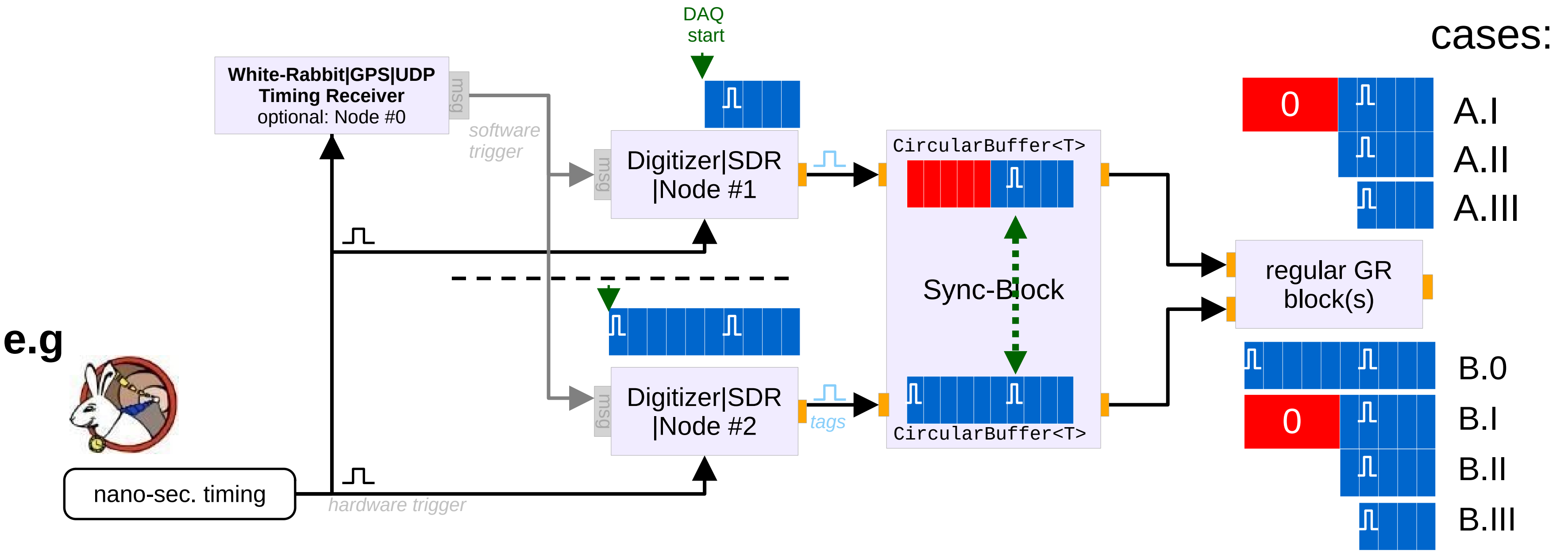


e.g. https://docs.unity3d.com/Packages/com.unity.visualscripting@1.8/manual/vs-control.html

Timing Synchronisation Across Multiple Nodes

aka. ‘The Two Clock Problem’

- MIMO signals – if possible – are usually synchronised via each RX channel being on the same DAQ system
- not always possible: limited #channel per device (↔costs), largely spacially distributed DAQs (e.g. FAIR: 4.5 km)
- real-world problem: (re-)synchronise physically/spacially distributed sources within the same flow-graph
  - failure cases to consider: ‘reconnecting/restarting SDRs/nodes’, ‘no data’ & time-outs, … clock-drifts, transmission delays, ...



solved through standardised ‘Tag’s;
TRIGGER_NAME, TRIGGER_TIME, TRIGGER_OFFSET

Outline: addressing the core GR3.X ‘pain-points’ together

A) buffer- and compute-related performance improvements (SIMD)

B) making asynchronous event-based signal processing a first-class citizen of GR (↔ packet-radios)

C) enabling user-customisable schedulers that can be optimised for e.g. throughput, latency, hardware resources, ...)

D) easier integration of vendor-neutral heterogeneous, distributed and embedded computing, and

E) helping making GNU Radio safer, leaner, cleaner, and easier to use for core- and out-of-tree developers using modern C++ standards.
→ focus on: industrial 24h/7 deployment, safety, cybersecurity, long-term maintainability, compliance, ...

GR3 Scheduler Architecture

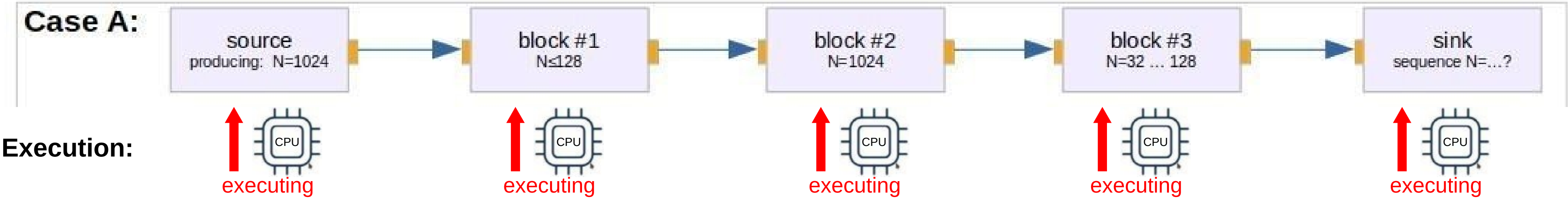
key rationale: ‘micro-service’-style limitations in GR3 vs. ‘Mechanical Sympathy’

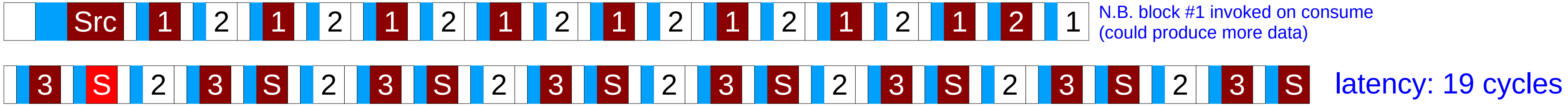
Two limiting factors in GR3 design (main ‘pain points’ and reason why GR4 development was started):

A) blocks are also `` `std::thread`s `` and their processing function managed by the systems scheduler (Linux: O(1)→ CFS)
→ blocks compete with each other and system tasks (starvation, non-determinism, non-embedded/u-processor friendly)
- notably: many more threads created than there are available computing resources/cores

B) max/min buffer data intake is not globally controlled by default
→ creates undesired latencies, larger memory footprint than necessary

Example (simplified):



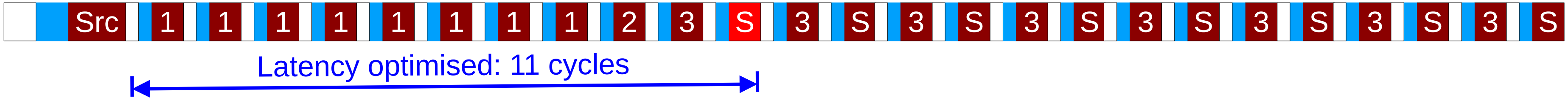


N.B. block #1 invoked on consume
(could produce more data)

latency: 19 cycles

Wouldn’t this be nicer (ignoring other optimisation, e.g. Merge-API)**:**

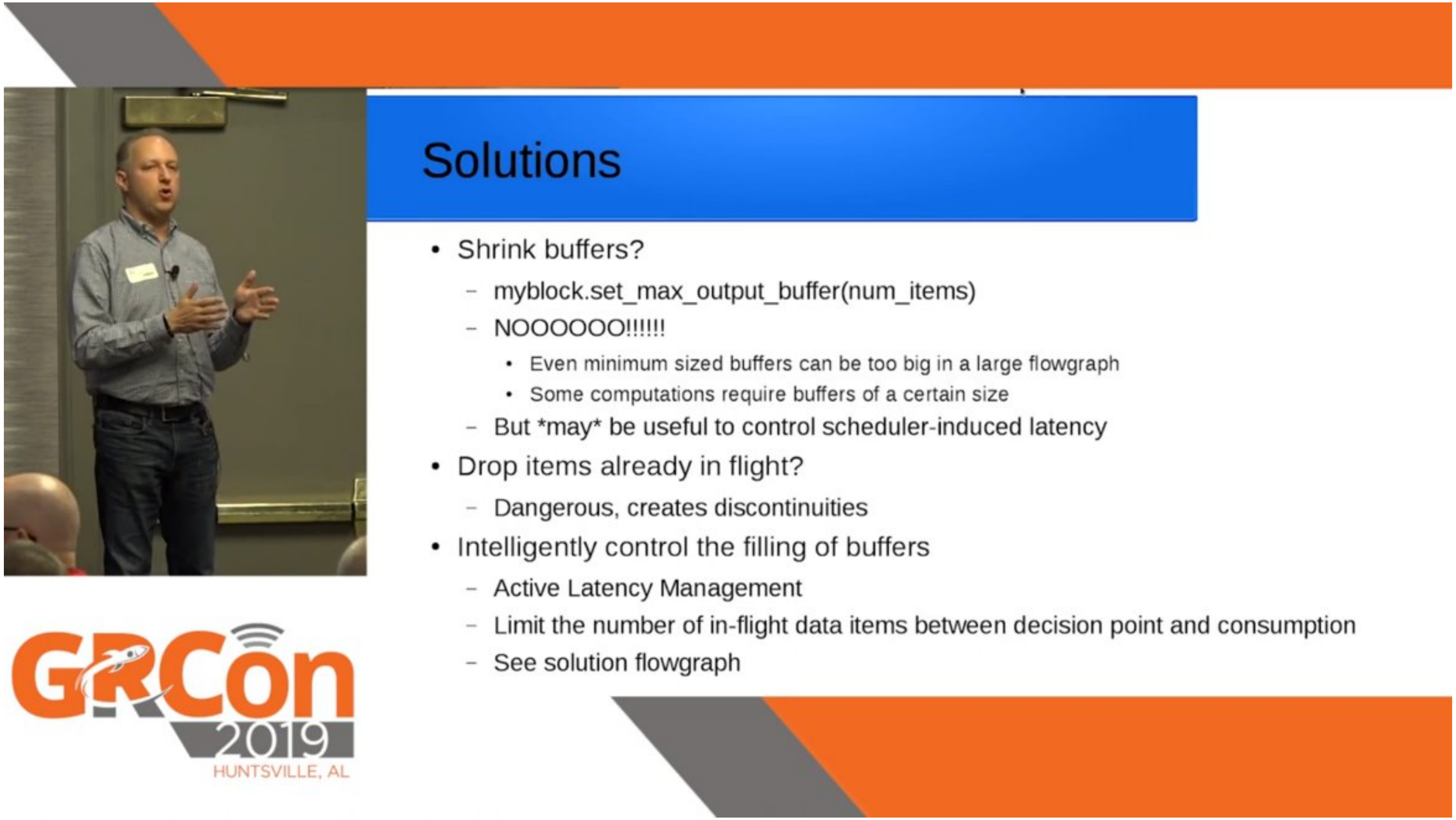
throughput optimised (18 cycles)



Latency optimised: 11 cycles

GR3 Scheduler Architecture

‘micro-service’-style limitations in GR3 → not a new but long-standing problem



Solutions

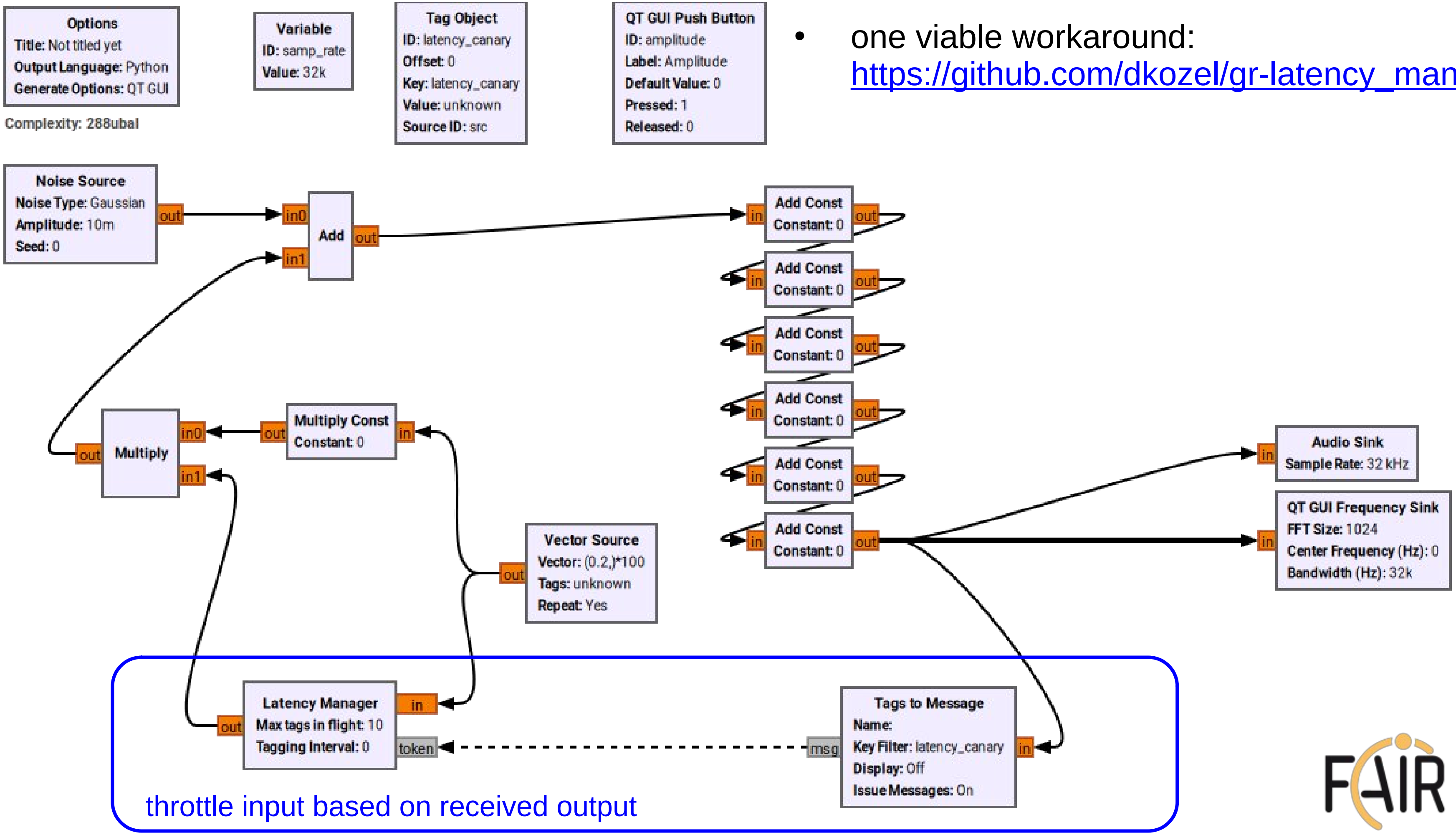
- Shrink buffers?
  - myblock.set_max_output_buffer(num_items)
  - NOOOOOO!!!!!!
    - Even minimum sized buffers can be too big in a large flowgraph
    - Some computations require buffers of a certain size
  - But *may* be useful to control scheduler-induced latency
- Drop items already in flight?
  - Dangerous, creates discontinuities
- Intelligently control the filling of buffers
  - Active Latency Management
  - Limit the number of in-flight data items between decision point and consumption
  - See solution flowgraph

Matt Ettus & Derek Kozel https://youtu.be/jq0RewceCwc

GR3 Scheduler Architecture

‘micro-service’-style limitations in GR3 → not a new but long-standing problem

- one viable workaround: https://github.com/dkozel/gr-latency_manager



Options
Title: Not titled yet
Output Language: Python
Generate Options: QT GUI

Complexity: 288ubal

Variable
ID: samp_rate
Value: 32k

Tag Object
ID: latency_canary
Offset: 0
Key: latency_canary
Value: unknown
Source ID: src

QT GUI Push Button
ID: amplitude
Label: Amplitude
Default Value: 0
Pressed: 1
Released: 0

Noise Source
Noise Type: Gaussian
Amplitude: 10m
Seed: 0

Add Const
Constant: 0

Multiply Const
Constant: 0

Vector Source
Vector: (0.2,)*100
Tags: unknown
Repeat: Yes

Audio Sink
Sample Rate: 32 kHz

QT GUI Frequency Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

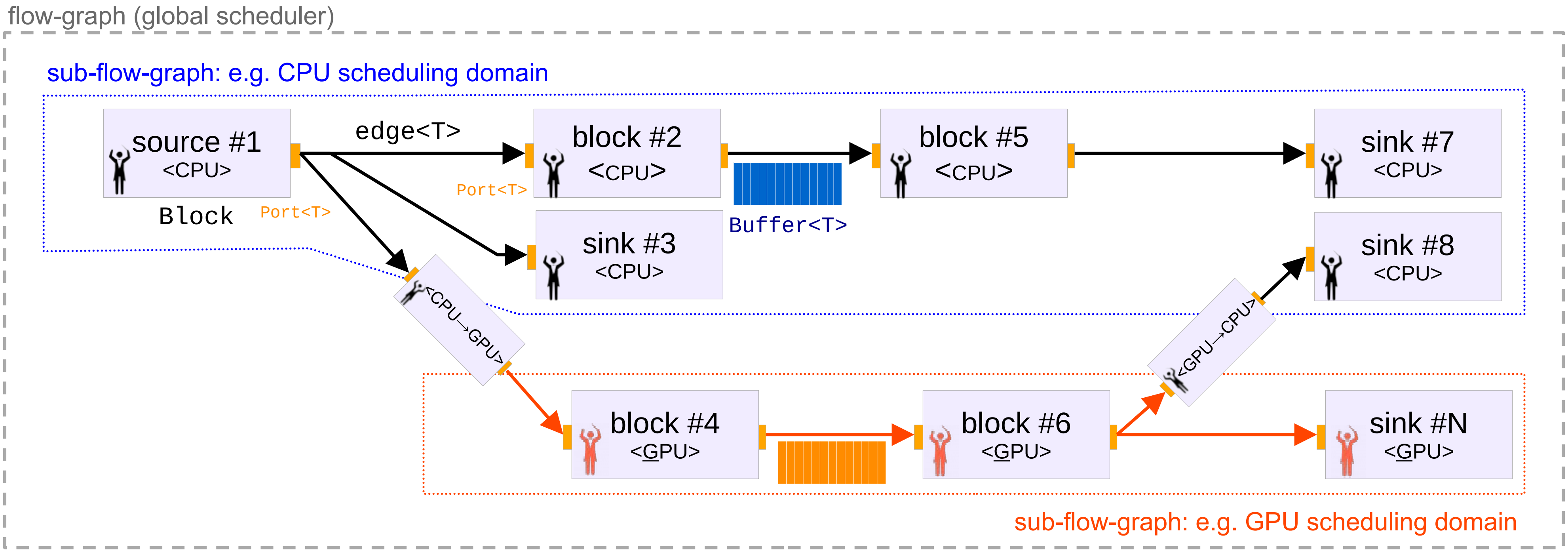
Latency Manager
Max tags in flight: 10
Tagging Interval: 0

Tags to Message
Name:
Key Filter: latency_canary
Display: Off
Issue Messages: On

throttle input based on received output

Graph-Based Signal-Flow Description

GR3.x→4: multiple compute domains & inverted scheduler paradigm `Block` → `Graph`



flow-graph (global scheduler)

sub-flow-graph: e.g. CPU scheduling domain

source #1 <CPU>

`Block`

`Port<T>`

`edge<T>`

block #2 <CPU>

`Port<T>`

`Buffer<T>`

block #5 <CPU>

sink #7 <CPU>

sink #3 <CPU>

sink #8 <CPU>

<CPU→GPU>

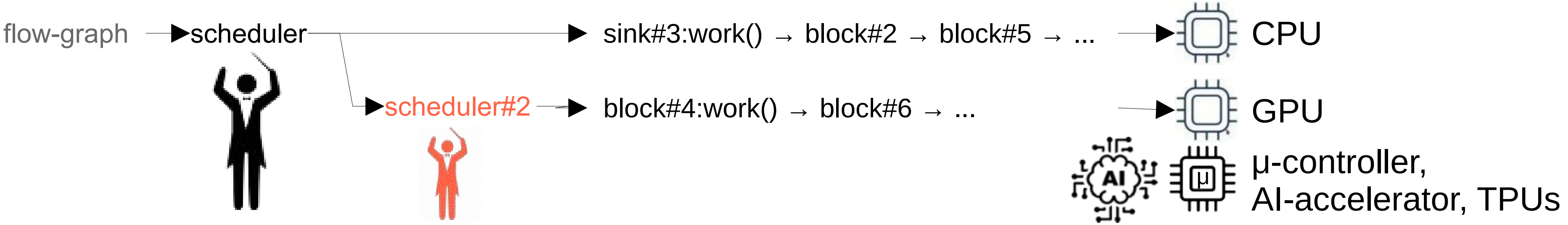
<GPU→CPU>

block #4 <GPU>

block #6 <GPU>

sink #N <GPU>

sub-flow-graph: e.g. GPU scheduling domain



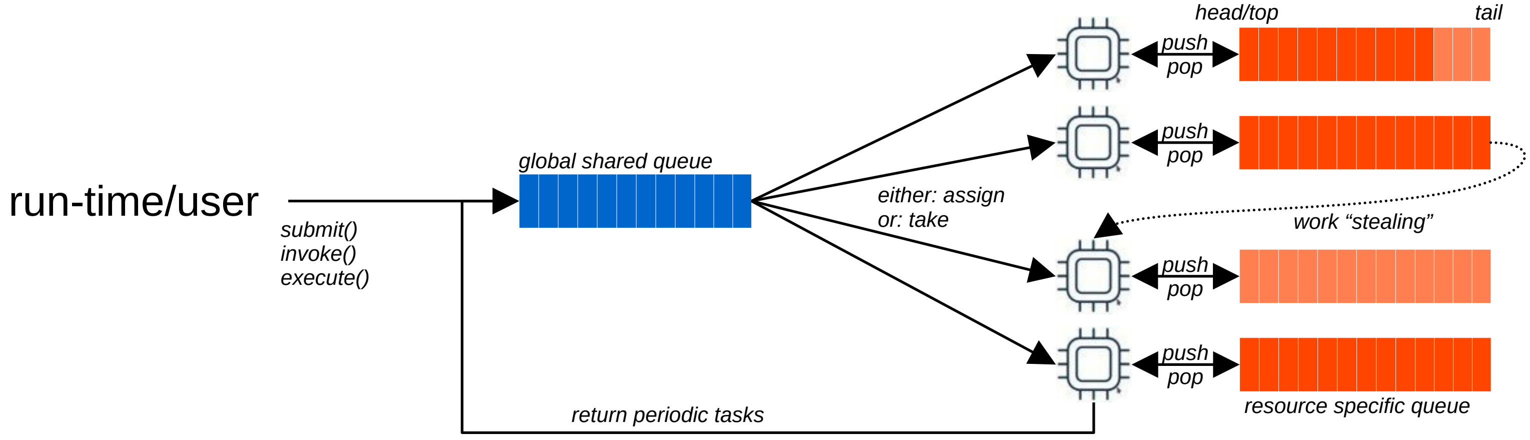
flow-graph ▶scheduler ▶ sink#3:work() → block#2 → block#5 → ... ▶ CPU

▶scheduler#2 ▶ block#4:work() → block#6 → ... ▶ GPU

µ-controller,
AI-accelerator, TPUs

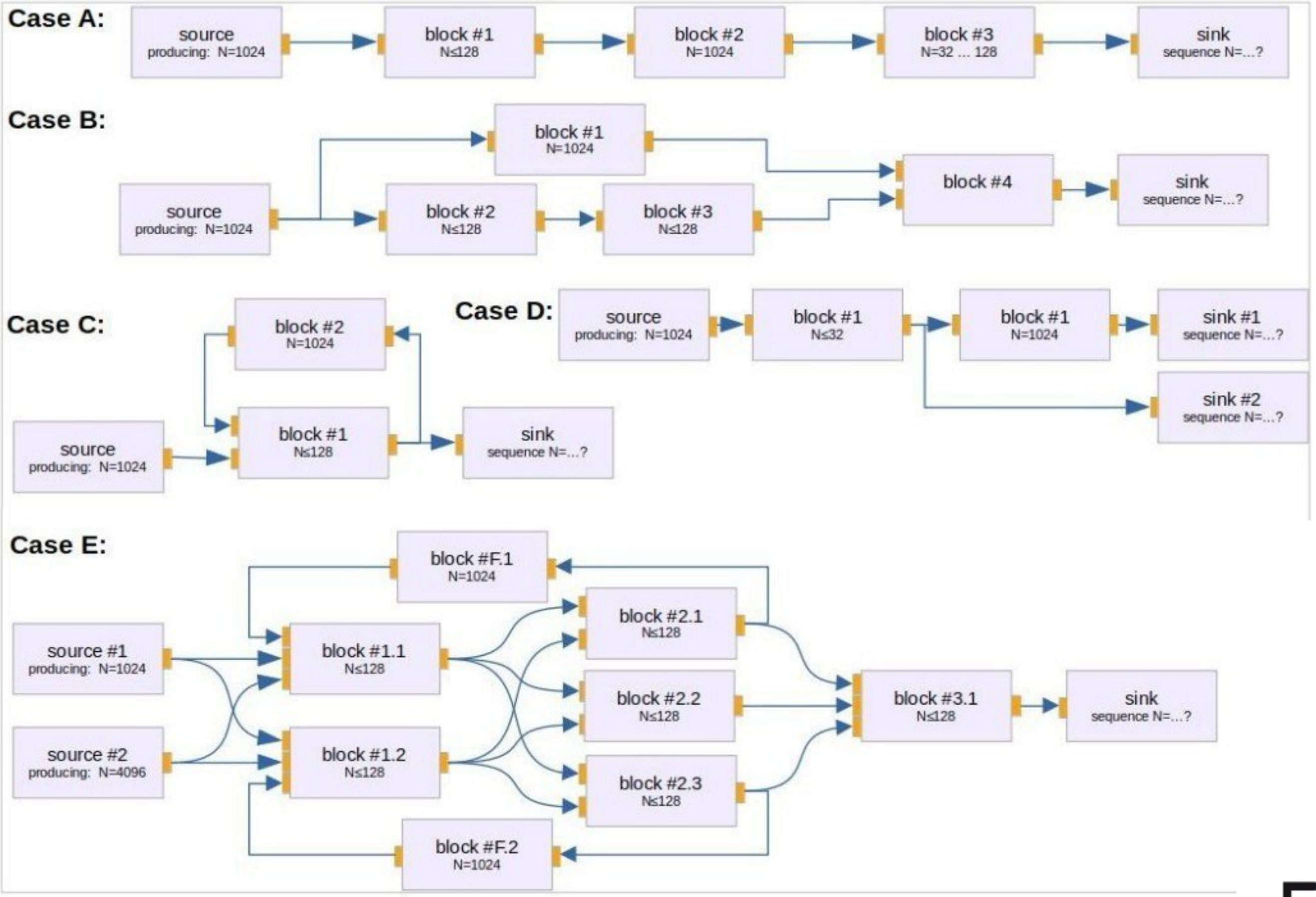
User-pluggable Work Scheduler API Paradigm

- simplified core API providing flow-graph topology, block & work constraints, ...
  - e.g. `` `work(requestedWork)’ `` → `` `process[One,Bulk](...)’ `` function → controls latency
- enables users to write their own custom schedulers that
  - optimise for their specific application: latency vs. throughput vs. execution order vs. …
  - assign and distribute block work functions across available compute resources (CPU|GPU|...)
  - choose your own high-level scheduler implementation specific design choices:
    - static scheduling (merge-API), round-robin vs. prioritised scheduling, dependent/pre-requisite flow-graphs first
    - CPU shielding/thread affinity, real-time vs. non-real-time sub-flow-graphs, …
    - data chunk-size based, ‘single global queue’ vs. ‘per-core queues & work stealing`, …



User-pluggable Work Scheduler API Paradigm

Example: Topologies specific designed to trip-up schedulers 😈 😇

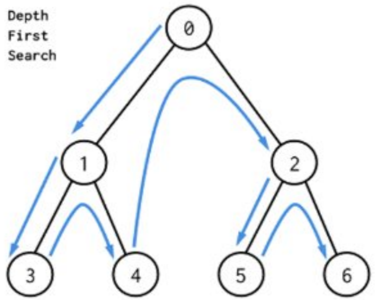


exercise:
what is the correct, best, and most efficient execution order?

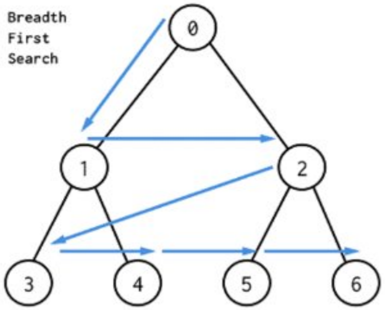
User-pluggable Work Scheduler Architecture

Implemented initially only the most basic scheduler strategies to test and verify new API

0. Busy-Looping → “Simple” naive implementation
1. Depth-first



2. Breadth first



Other possible Algorithms:

https://github.com/gnuradio/gnuradio4/blob/main/include/README.md

- *Topological Sort*
- *Critical Path Method (CPM) → minimizes total completion time*
- *A* → shortest path*
- *Wu Algorithm → minimal execution time*
- *Johnson’s Algorithm → CPM on multiple processor cores*
- *Program Evaluation and Review Technique (PERT)*
- *Belman-Ford Algorithm*
- *Dijkstra's Algorithm → shortest path*
- *A* → shortest path*
- *… combinations of the above and many more*

Next Step: GNU Radio competition to find the best ‘default’, ‘real-time’, ‘throughput’ optimising scheduler for given benchmark topologies.

Outline: addressing the core GR3.X ‘pain-points’ together

A) buffer- and compute-related performance improvements (SIMD)

B) making asynchronous event-based signal processing a first-class citizen of GR (↔ packet-radios)

C) enabling user-customisable schedulers that can be optimised for e.g. throughput, latency, hardware resources, ...)

D) easier integration of vendor-neutral heterogeneous, distributed and embedded computing, and

E) helping making GNU Radio safer, leaner, cleaner, and easier to use for core- and out-of-tree developers using modern C++ standards.
→ focus on: industrial 24h/7 deployment, safety, cybersecurity, long-term maintainability, compliance, ...

Next Steps:
Pushing GNU Radio 4.0$^{\text{beta}}$ towards wider production use

PUBLIC MONEY
PUBLIC CODE

- **Community Engagement**
  - **Foster Community, Increase Visibility, Educate & Document**
- **Core Library & User Experience**
  - **Complete Core-Lib Blocks.**
  - **Default official GRC-style UI Integration.**
- **Performance & Long-Term Vision**
  - **Optimise Compile-Time Performance (WIP)**
  - **Expand Utility, UX, & HW Integrations**:
    GPUs, FPGAs, micro-controller, AI-accelerators & -TPUs
  - **Expand Core Team:** increase the 'bus factor'
  - **Sustainable Growth.**
  - **Safety & Security Hardening:** meet regulatory standards
    → critical for industry and public infrastructure adoption → looking for partners

SYCL™

AdaptiveCpp

Open Questions:

- What would it take for your organisation, institute, or company to publicly adopt GR4.0?
- Is the present license still too restrictive for GR to be used in public organisations & industry?

FAIR The Universe in the Laboratory

CONNECT
SHARE
COLLABORATE

Looking forward to a technical dialogue and
building and strengthening cross-disciplinary partnerships …

Thank you!

GNU Radio 4.0 beta
THE FREE & OPEN SOFTWARE RADIO ECOSYSTEM

Stay in touch with us!

Instagram:
@universeinthelab

Facebook:
@GSIHelmholtzzentrum
@FAIRAccelerator

LinkedIn:
GSI Helmholtz Centre
for Heavy Ion Research

Mastodon:
@FAIR_GSI_de
@helmholtz.social

YouTube:
FAIR/GSI –
The Universe in the Lab

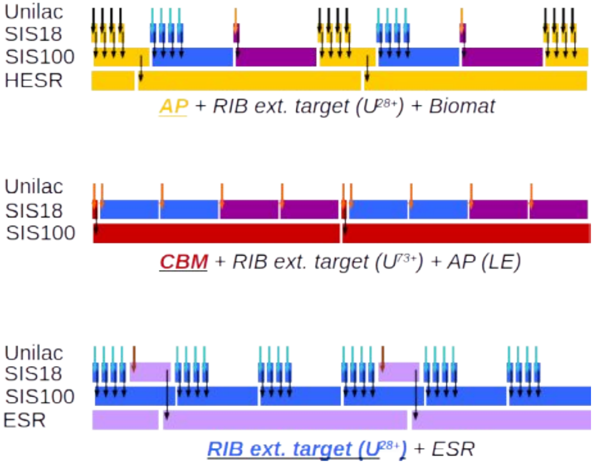
Appendix

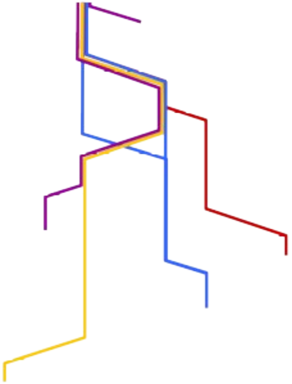
Modern and much Simpler C++ User-API

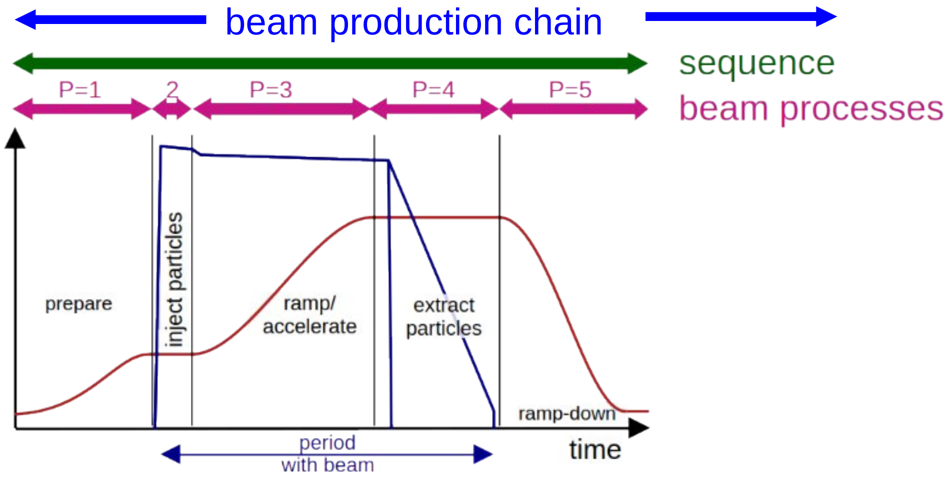
Transactional and Multiplexed Settings Interface

Large industrial setting: SDR HW device are often not exclusively used by one user or analysis

A) w/o feedback → operate multiple parallel signal-processing pipelines
B) w/ feedback → reconfigure HW/algorithms on-the-fly (e.g. adaptive gain scheduling)







- (optionally) multiplexed block settings changes via special context ‘`ctx`’ Tag
  → facilitate flexible multi-mission/multi-user operations

Modern and much Simpler C++ User-API

Transactional and Multiplexed Settings Interface

```cpp
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public Block<BasicMultiplier<T>> {
  InPort<float>  in;
  OutPort<float> out;
  float          scaling_factor = static_cast<float>(1);
  std::string    ctx;     // ↔ multiplexing settings context (optional info)

  template<t_or_simd<T> V>
  constexpr V processOne(const V &a) const noexcept {
      return a * scaling_factor;
  }
};
ENABLE_REFLECTION_FOR_TEMPLATE(BasicMultiplier, in, out, scaling_factor, ctx);
```

Essentially three settings APIs:

A) default: via pmt-messages and `msgIn` ↔ `msgOut` port cascades (scheduler ↔ graph ↔ block)
B) via Tag: propagation of e.g. sample_rate to down-stream blocks
C) via Block (primarily unit-tests): `block.setting.set({“scaling_factor”, T(42)});`
(via John Sallay’s pmt library)

Generic OpenDigitizer Scope – since 2017 (ACO+SYS)

19” Hardware Integration & Deployment

Device type	#Systems
Magnet power converters	~180
RF systems (Master DDS, cavities)	~70
Fast pulsed devices (Kickers, choppers, mag. horn)	~40
Beam exciters (KO, TFS, BTF, stoch. cooling)	~15
Beam signals (Schottky, FCT/RF, phase probes)	~10
HV devices (Septa, e-cooler HV)	~25
Miscellaneous (Pulse power, MPS, Testing)	~10
#Systems Total	**~350**
#Digitizers Total	**~300**

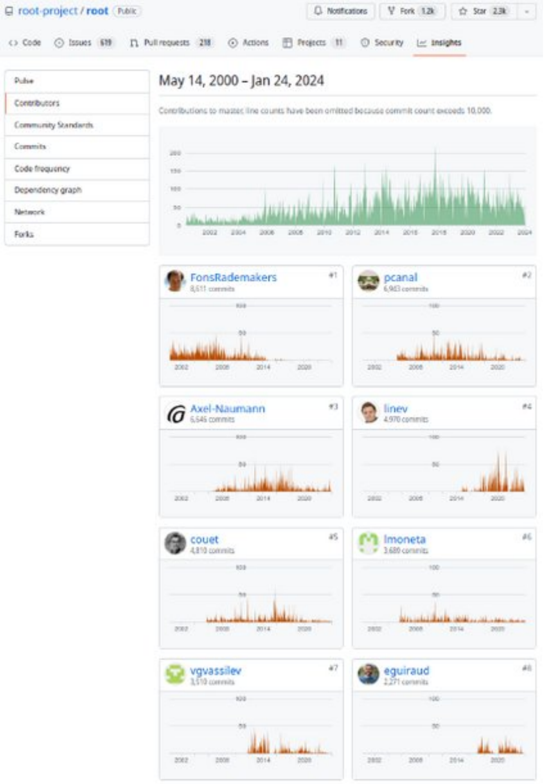
- i.e. system w/o pre-existing solution (e.g. those provided by BEA)
- presently deployed ~200 systems (mostly SIS18)
  → ~300+ systems @FAIR
  (many different internal and external groups involved)

Metric: Bus Factor & Reusability

taken from the OSWG/C++-UG software guideline discussions @GSI/FAIR

> … *"number of team members that have to disappear from a project before the project stalls due to lack of knowledgeable or competent personnel"*, Wikipedia

GitHub Tooling Example:



root-project / root Public

May 14, 2000 – Jan 24, 2024

Contributions to master, line counts have been omitted because commit count exceeds 10,000.

FonsRademakers #1 8,611 commits

pcanal #2 6,963 commits

Axel-Naumann #3 6,546 commits

linev #4 4,970 commits

couet #5 4,810 commits

lmoneta #6 3,680 commits

vgvassilev #7 3,510 commits

eguiraud #8 2,271 commits

GitHub Tooling Example (by JetBrains):

Only pattern matches are shown

Using this mode, we can highlight if the bus factor changes when one or more authors leave

Name: python/cpython

Bus Factor: 13

Author Contribution

Colors can be picked by clicking on the squares below

☐ OK [5+]

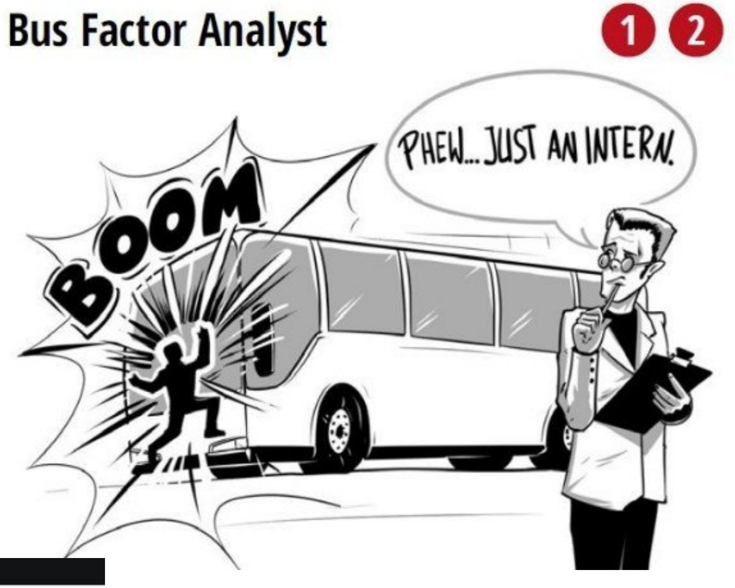
🟨 Low [2-4]

🟥 Dangerous [0-1]

▨ Not Applicable

Choose the values at which the colors switchover

1 — 20



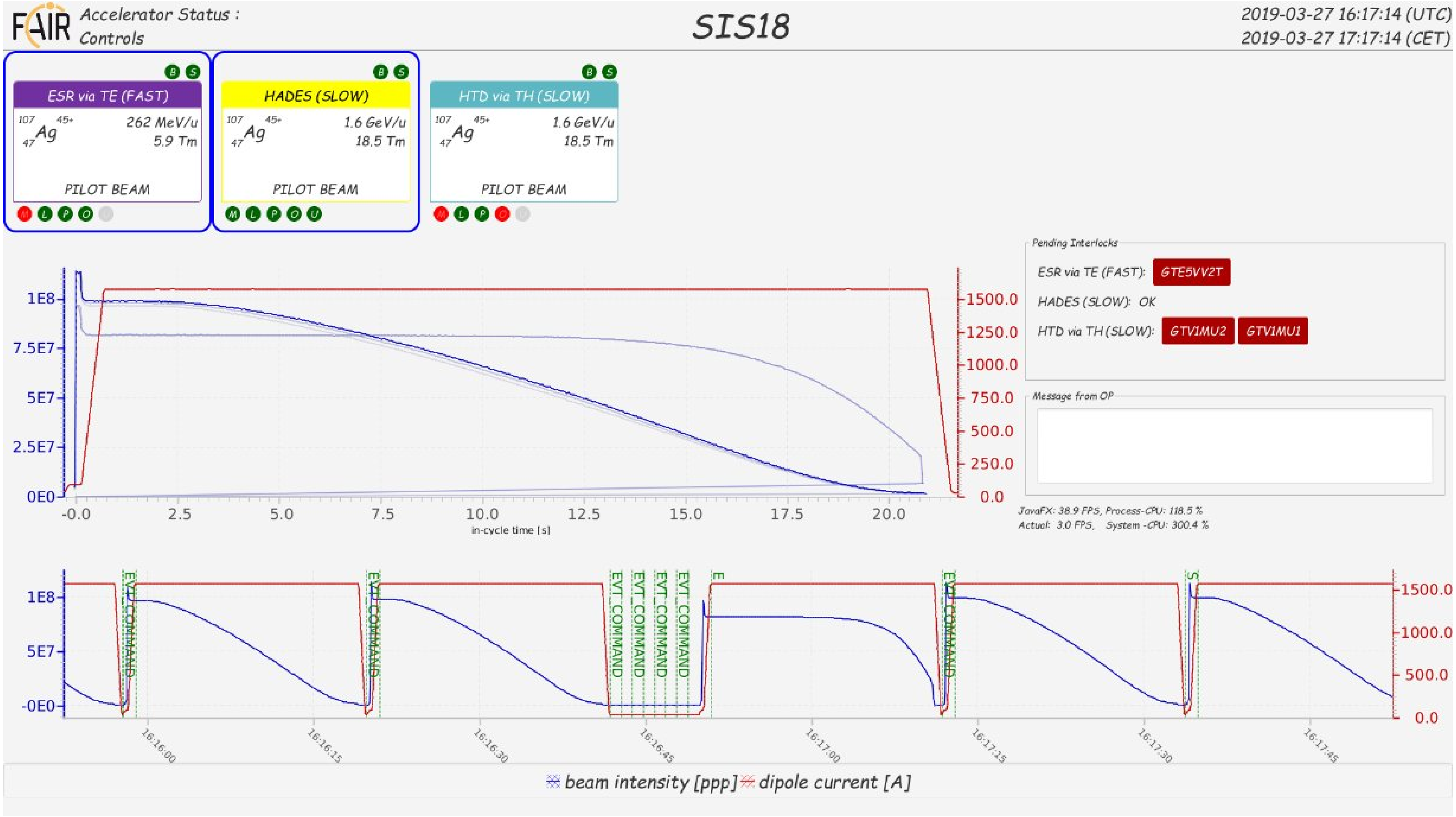
https://www.playitstartup.com/

Goal: bus factor ≥ 3

(for L2, L3 applications)

Generic <u>Open</u>Digitizer Facility Monitoring – Early Fixed Displays

https://fair-acc.github.io/opendigitizer/



FAIR Accelerator Status : Controls

SIS18

2019-03-27 16:17:14 (UTC)
2019-03-27 17:17:14 (CET)

ESR via TE (FAST)
$^{107}_{47}Ag^{45+}$ 262 MeV/u 5.9 Tm
PILOT BEAM

HADES (SLOW)
$^{107}_{47}Ag^{45+}$ 1.6 GeV/u 18.5 Tm
PILOT BEAM

HTD via TH (SLOW)
$^{107}_{47}Ag^{45+}$ 1.6 GeV/u 18.5 Tm
PILOT BEAM

Pending Interlocks
ESR via TE (FAST): GTE5VV2T
HADES (SLOW): OK
HTD via TH (SLOW): GTV1MU2 GTV1MU1

Message from OP

JavaFX: 38.9 FPS, Process-CPU: 118.5 %
Actual: 3.0 FPS, System -CPU: 300.4 %

beam intensity [ppp] dipole current [A]

GNU Radio Timeline

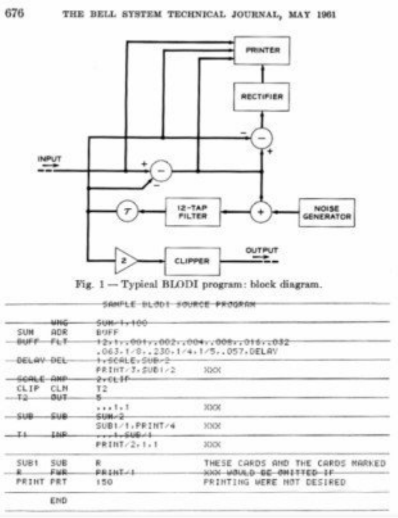
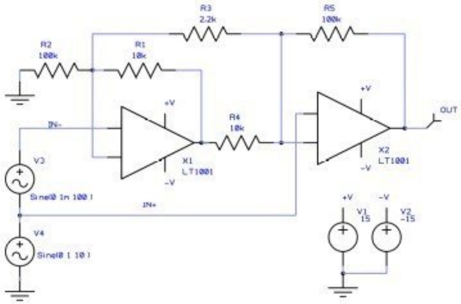
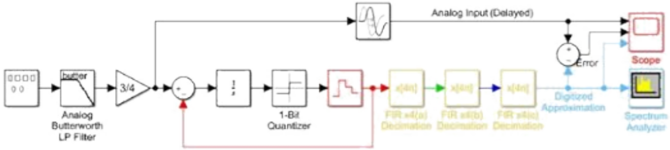
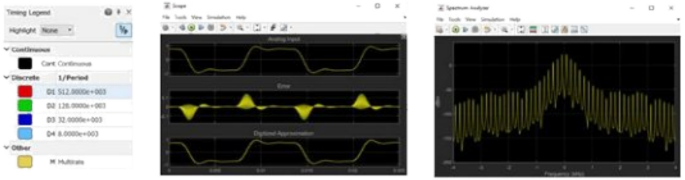

676 THE BELL SYSTEM TECHNICAL JOURNAL, MAY 1961

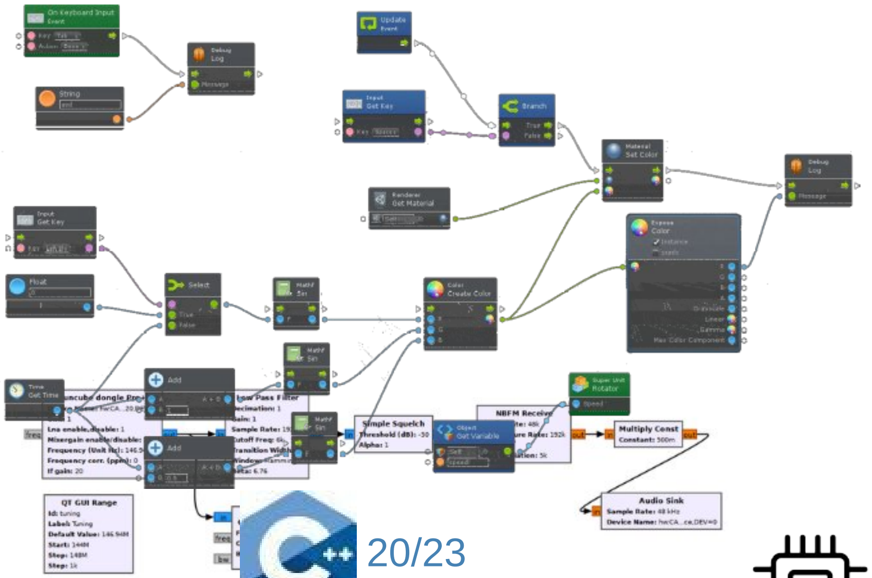
Fig. 1 — Typical BLODI program: block diagram.

```
            SAMPLE BLODI SOURCE PROGRAM
      WNG   SUM-1,100
SUM   ADR   BUFF
BUFF  FLT   12,1,.001,.002,.004,.008,.016,.032
            .063,1/8,.230,1/4,1/5,.057,DELAY
DELAY DEL   1,SCALE,SUB-2
            PRINT/3,SUB1/2      XXX
SCALE AMP   2,CLIP
CLIP  CLM   T2
T2    OUT   5
            ...1,1              XXX
SUB   SUB   SUM-2
            SUB1/1,PRINT/4      XXX
T1    INP   ...1,SUB/1
            PRINT/2,1,1         XXX
SUB1  SUB   R                   THESE CARDS AND THE CARDS MARKED
R     FWR   PRINT/1             XXX WOULD BE OMITTED IF
PRINT PRT   150                 PRINTING WERE NOT DESIRED
      END
```









20/23

performance, safety & design improvements

BLODI (BLockDIagram)

1961

SPICE

1973

UC Berkeley

SIMULINK®

1984

LabVIEW

1986

MathWorks®

NATIONAL INSTRUMENTS™

SIMATIC S7 GRAPH (PLCs)

1990ies

SIEMENS

GNU Radio

THE FREE & OPEN SOFTWARE RADIO ECOSYSTEM

2000

NSF

GPL v3 Free Software Free as in Freedom

DARPA

& vendor-neutral

~2006

unity

4.0beta

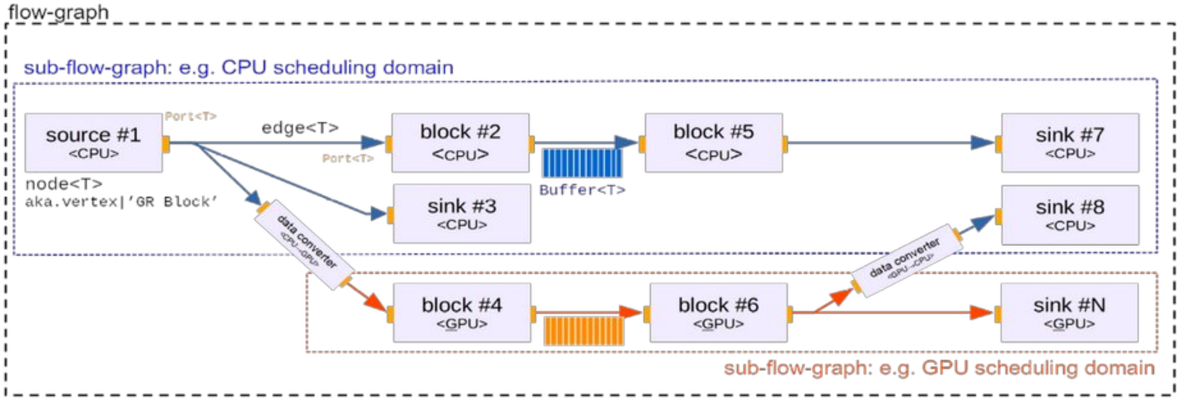
2024

GPL v3 Free Software Free as in Freedom

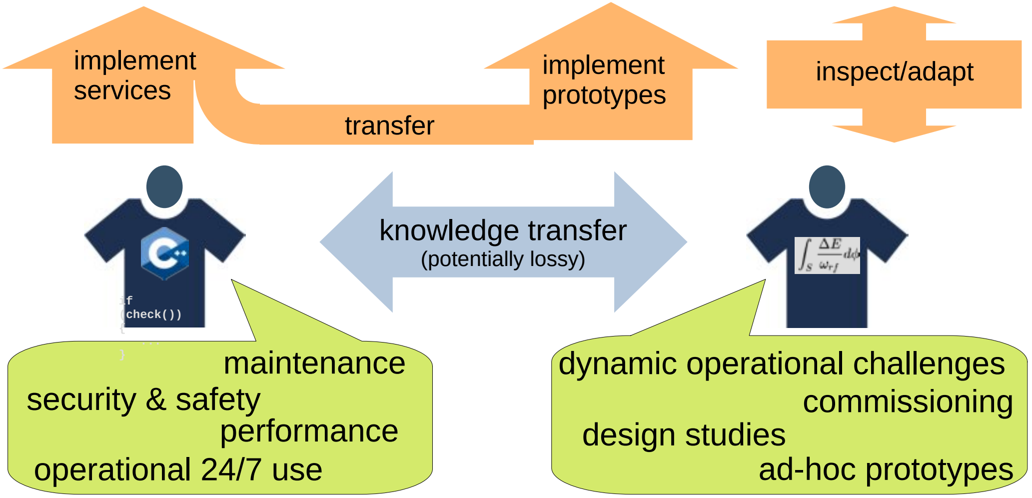
LGPL v3 Free Software Free as in Freedom

Why we invest into GNU Radio 4.0^{beta}

Mechanical Sympathy & Graph-Based Signal-Flow Description



UI/ WEBASSEMBLY



maintenance
security & safety
performance
operational 24/7 use

dynamic operational challenges
commissioning
design studies
ad-hoc prototypes

simplifies onboarding for new partners and users to participate & contribute more effectively

GSI/FAIR PSP 2.14.17 Project - FAQ

Overview of SYS-contributed FLOSS

- **OpenCMW** *http://opencmw.io/*
  - … a lightweight, extendable, open-source middleware abstraction for C++/Java
  - … flexibility through unifying diverse transport and serialiser protocols (JSON, YAML, HTML, binary, rda3-data)
  - … core development finished in 2021 – RBAC integration pending (target: by 2025)
- **GNU Radio (GR)** *https://www.gnuradio.org/, https://github.com/gnuradio/gnuradio4*
  - … an open-source ecosystem for signal processing, widely adopted across gov-funded laboratories, industry, and academia.
  - … minimises GSI/FAIR's resource commitment and maintenance burden (‘bus factor’)
  - … enhances capabilities by leveraging external developments, used since 2017 now transitioning to GR 4.0.
- **OpenDigitizer** https://github.com/fair-acc/opendigitizer
  - … reconfigurable full-stack framework for aggregating and processing data from diverse accelerator and experiment devices.
  - … SDR: end-user adaptable to fulfil different roles with the same/different HW/SW (i.e. ‘multi-mission’ operation)
  - … builds upon OpenCMW & GNU Radio, allowing graphical, C++, or Python implementations.
  - … “take and learn from the best”: incorporates features from UCAP, OASIS, DIP, Fixed-Displays (via WASM), ...
- **ChartFX** https://github.com/fair-acc/chart-fx
  - … scientific charting library, key in identifying bottlenecks and functionalities missing from current control systems.
  - … used in most diagnostics control room applications; a precursor to OpenCMW development.
- **Why these “new” Developments?**
  - … strong functional need for middle-tier processing capabilities for commissioning and reaching FAIR beam parameters
  - … existing incomplete implementation lacked critical core functionalities & performance