

GNU Radio 4.0: Standing on the Shoulders of Giants

An Overview of New Features and Significant Enhancements

Josh Morman¹, Derek Kozel¹, Ralph J. Steinhagen²

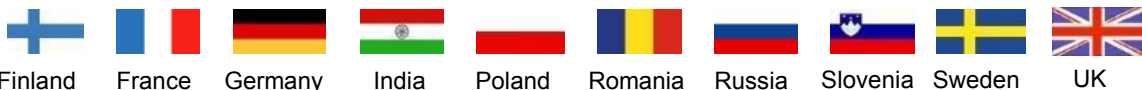
on behalf of: the GR Architecture Team, Björn Balazs³, Giulio Camuffo³, Ilya Doroshenko³, Alexander Krimm², Semën Lebedev², Ivan Čukić³, Matthias Kretz², Frank Osterfeld³, ...

¹ GNU Radio 4.0 (lead)

² FAIR – Facility for Anti-Proton and Ion Research & GSI, Darmstadt, Germany

³ KDAB Berlin, Germany

GRCon123
2023-09-07
TEMPE, ARIZONA



Modernisation Goals

GNU Radio organically grew the past 20 years ...

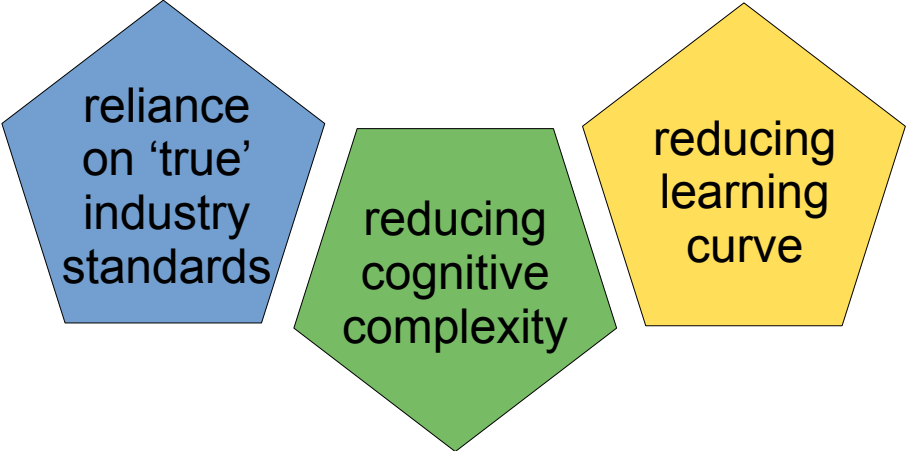


... GR 4.0 opportunity: preserve what is good, prune what is unhealthy to keep the project growing and maintainable for another 20 years

photos courtesy: <https://fieldertree.com/>
https://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants

Modernisation Goals

simplify onboarding for new contributors to participate/contribute more effectively



academia



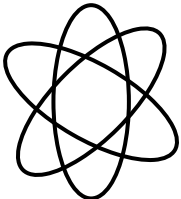
Promote ...

collaboration



government

knowledge exchange



science



adoption



industry

Modernisation Goals

improve performance, industrial integration+deployment

1. Preserve and Grow the existing diverse GR Ecosystem.

- thin Python interface over C++ API
- avoid Python-only implementations (except OOT modules)
- swappable runtime components (both in and out of tree)
- simplified block development: get block developers to "insert code here" without lots of boilerplate or complicated code

2. Clean- and Lean Code-Base Redesign

3. Performance Optimisations

4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)

5. Advanced Processing Features

6. Broaden Cross-Platform Support (including WebAssembly)

7. User-pluggable Work Scheduler Architecture

8. Overall Project Direction



1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – “Hello GNU Radio World!”

```
struct BasicMultiplier : public node<BasicMultiplier> {  
    IN<float>    in;  
    OUT<float>   out;  
    float       scaling_factor = static_cast<float>(1);
```

```
    [[nodiscard]] constexpr float  
    process_one(const float &a) const noexcept {  
        return a * scaling_factor;  
    }  
};
```

Key Take-Aways:

- **Simplified Block Development:** stand-alone creation is more intuitive. Code is the single source of truth.
[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

1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – with SIMD acceleration

```
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public node<BasicMultiplier<T>> {
    IN<T>      in;
    OUT<T>     out;
    T          scaling_factor = static_cast<T>(1);
};
```

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

```
ENABLE_REFLECTION_FOR_TEMPLATE_FULL((typename T), (BasicMultiplier<T>), in, out, scaling_factor);
```

1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – classic bulk operation I/II

```
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public node<BasicMultiplier<T>> {
    IN<T>      in;
    OUT<T>     out;
    T          scaling_factor = static_cast<T>(1);

    void // alternate interface
    process_bulk(std::span<const T> in, std::span<T> out) {
        std::ranges::transform(in, out.begin(), [sf = scaling_factor](const T& val) {
            return val * sf;
        });
    }
};
```

```
ENABLE_REFLECTION_FOR_TEMPLATE_FULL((typename T), (BasicMultiplier<T>), in, out, scaling_factor, context);
```

Fun Fact (aka. beware of 'premature optimisations'):

Benchmarking proved that using 'process_one(...)' is numerically more performant than 'process_bulk(...)'

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

1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – classic bulk operation II/III

```
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public node<BasicMultiplier<T>> {
    IN<T>      in;
    OUT<T>     out;
    T          scaling_factor = static_cast<T>(1);

    void // alternate interface with variable amount of input and output consumed
    process_bulk(ConsumableSpan auto& in, PublishableSpan auto& out) const noexcept {
        // [...] user-defined processing logic [...]
        in.consume(3UL); // consume 3 samples
        out.publish(2UL); // publish 2 samples → effectively a 3:2 re-sampler
    }
};

ENABLE_REFLECTION_FOR_TEMPLATE_FULL((typename T), (BasicMultiplier<T>), in, out, scaling_factor, context);
```


1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – Decimator & Interpolator

```
template<typename T>
requires (std::is_arithmetic<T>())
struct Resampler : public node<Resampler<T>, PerformDecimationInterpolation, PerformStride> {
    IN<T>      in;
    OUT<T>     out;

    void // 'in' and 'out' matched N x numerator & N x denominator samples
    process_bulk(std::span<const T> in, std::span<T> out) const noexcept {
        // [...] user-defined re-sampling logic [...]
    }
};
ENABLE_REFLECTION_FOR_TEMPLATE_FULL((typename T), (Resampler<T>), in, out);
```

optional NTPP parameters → “only-pay-for-what you use”

PerformDecimationInterpolation, PerformStride

[...] user application code:

```
graph flow; // flow-graph object owning the blocks/connections
auto &block = flow.make_node<Resampler<float>>({"numerator", 1024UL}, {"denominator", 1UL});
// skipping 10k samples
block.settings().set({"stride", 10'000UL}); // std::map<std::string, pmt_t> interface
// alternate direct interface
block.stride = 10'000UL;
block.update_active_parameters(); // N.B. synchronises PMT-map representation
```

Shout-out to: Semën Lebedev for fleshing this out and covering most corner cases



1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – “Hello GNU Radio World!” V2

```
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public node<BasicMultiplier<T>> {
    IN<T>      in;
    OUT<T>     out;
    T          scaling_factor = static_cast<T>(1);
    std::string context;      // ↔ multiplexing settings
};
```

```
template<t_or_simd<T> V>
[[nodiscard]] constexpr V
process_one(const V &a) const noexcept {
    return a * scaling_factor;
}
};
```

```
ENABLE_REFLECTION_FOR_TEMPLATE_FULL((typename T), (BasicMultiplier<T>), in, out, scaling_factor, context);
```

1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – Settings Management

```
template<typename T>
requires (std::is_arithmetic<T>())
struct BasicMultiplier : public node<BasicMultiplier<T>> {
    IN<T>      in;
    OUT<T>     out;
    T          scaling_factor = static_cast<T>(1);
    std::string context;      // ↔ multiplexing settings

    void settings_changed(const property_map &old, const property_map &new) {
        // optional function that is called whenever settings change
    }
};
```

```
template<t_or_simd<T> V>
[[nodiscard]] constexpr V
process_one(const V &a) const noexcept {
    return a * scaling_factor;
};
```

two settings update mechanisms:

a) via thread-safe getter/setter (std::map<string, pmt_t>)

b) via streaming tags

N.B. 'process_XXX' is (default) invoked with the first sample after settings have been applied

c) via async message port (std::map<string, pmt_t>)

```
ENABLE_REFLECTION_FOR_TEMPLATE_FULL((typename T), (BasicMultiplier<T>), in, out, scaling_factor, context);
```

Shout-out to: John Sallay for providing the new PMT library extension (pls. buy him a beer).



1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – FIR pre-production code

```
template<typename T>
requires std::floating_point<T>
struct fir_filter : node<fir_filter<T>, Doc<R""(
@brief Finite Impulse Response (FIR) filter class
```

The transfer function of an FIR filter is given by:
 $H(z) = b[0] + b[1]*z^{-1} + b[2]*z^{-2} + \dots + b[N]*z^{-N}$

```
)"">> {
    IN<T>          in;
    OUT<T>         out;
    std::vector<T> b{}; // feedforward coefficients
    history_buffer<T> inputHistory{ 32 };

    void
    settings_changed(const property_map & /*old_settings*/, const property_map &new_settings) noexcept {
        if (new_settings.contains("b") && b.size() >= inputHistory.capacity()) {
            inputHistory = history_buffer<T>(std::bit_ceil(b.size()));
        }
    }

    constexpr T
    process_one(T input) noexcept {
        inputHistory.push_back(input);
        return std::inner_product(b.begin(), b.end(), inputHistory.rbegin(), static_cast<T>(0));
    }
};
```

1. Preserve and Grow the existing diverse GR Ecosystem

"insert code here" without lots of boilerplate or complicated code – IIR pre-production code

```
template<typename T, IIRForm form = std::is_floating_point_v<T> ? IIRForm::DF_II : IIRForm::DF_I>
requires std::floating_point<T>
struct iir_filter : node<iir_filter<T, form>, Doc<R"">(
@brief Infinite Impulse Response (IIR) filter class
```

b are the feed-forward coefficients (N.B. b[0] denoting the newest and n[-1] the previous sample)
a are the feedback coefficients

```
)">> {
    IN<T>          in;
    OUT<T>         out;
    std::vector<T> b{ 1 }; // feed-forward coefficients
    std::vector<T> a{ 1 }; // feedback coefficients
    history_buffer<T> inputHistory{ 32 };
    history_buffer<T> outputHistory{ 32 };

    void
    settings_changed(const property_map & /*old_settings*/, con
    // [...] adjust history buffer sizes in case filters are c
}
```

```
[[nodiscard]] T
process_one(T input) noexcept {
    if constexpr (form == IIRForm::DF_I) {
        //  $y[n] = b[0] * x[n] + b[1] * x[n-1] + \dots + b[N] * x[n-N]$ 
        //  $- a[1] * y[n-1] - a[2] * y[n-2] - \dots - a[M] * y[n-M]$ 
        inputHistory.push_back(input);
        const T output = std::inner_product(b.begin(), b.end(), inputHistory.rbegin(), static_cast<T>(0)) //feed-forward
            - std::inner_product(a.begin() + 1, a.end(), outputHistory.rbegin(), static_cast<T>(0)); //feedback path
        outputHistory.push_back(output);
        return output;
    } else { /* handle other IIR forms */ }
};
```

Note for the eagle-eyed: this is not your dad's C/C++98 ...

- primarily use STD-only C++20 (& compatible header-only C++26 libs) for enhanced performance and brevity
N.B. no external platform specific dependencies ↔ portability
- driven/limited by libc++ implementation for WASM compatibility
(N.B. MSVC & gcc's stdlibc++ are both more advanced)
- embrace modern C++ while avoiding overly bleeding-edges

Great CppCon 2022 talk by Daniela Engert (YouTube, ~1h):
[Contemporary C++ in Action](#)

1. Preserve and Grow the existing diverse GR Ecosystem

Not a real concern ... end-user Python & C++ top-level block API

Your Python-User Feedback is appreciated:

given the choice of a possible green-field re-design, do you prefer to have ...

a) the same interface as GR 3.X i.e. full access to all nooks and grannies of the full 'work(wio)' function?

- possible, but high(er) core lib maintenance costs because of the various corner cases (↔ status quo)*

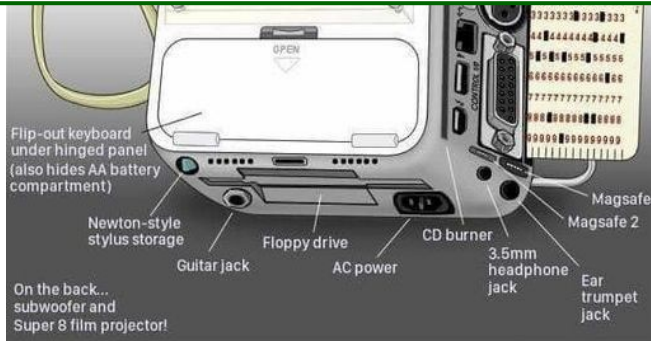
b) the reduced equivalent of the 'process_one(...)' and 'process_bulk(...)' function?

- reduced 'attack-surface' and easier/more flexible core lib maintenance*
- getting a better/easier interface for the 90% use case*

c) none ... e.g. I roll my own Python (pybind11, pypy, ...), Java, Rust, etc. bindings

d) other ...

Please get in contact with the GR Architecture Team! It's an Open Design!



```
def work(wio):  
    # get np arrays from input ports  
    # get mutable np arrays output ports  
  
    # produce and consume  
  
    return gr.work_return_t.OK
```


Modernisation Goals

improve performance, industrial integration+deployment

- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign**
 - favour 'composition' over 'inheritance'
 - boosts maintainability and adaptability
 - preserve tried-and-tested functionalities
- 3. Performance Optimisations
- 4. Tag-Based Timing System Integration
- 5. Advanced Processing Features
- 6. Broaden Cross-Platform Support
- 7. User-pluggable Work Scheduler Architecture
- 8. Overall Project Direction



Software must be adaptable to frequent changes

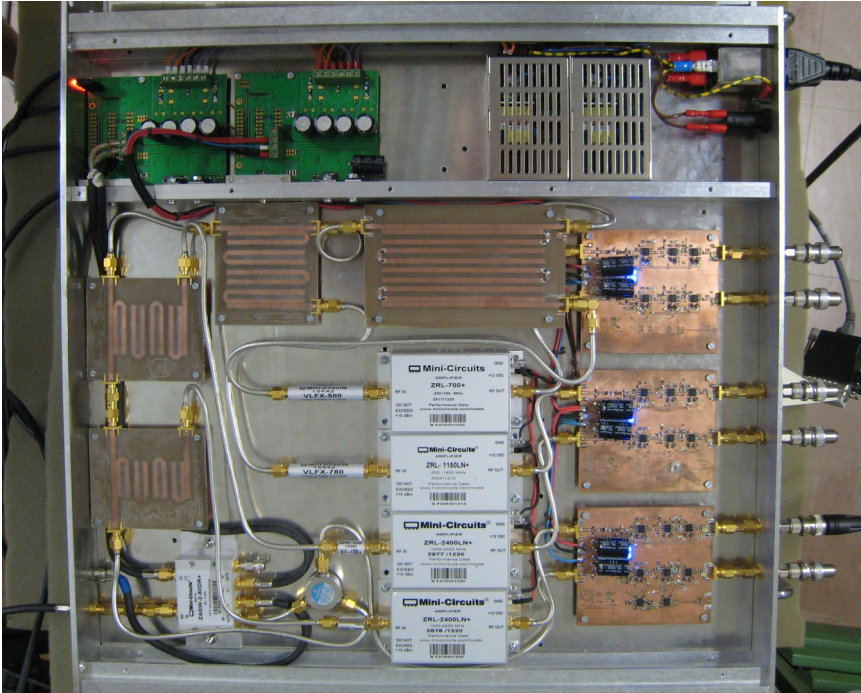
Software must be adaptable to frequent changes

- few are library developers
- more are application developers, i.e. users of the library
- most are application users
- all need to know ‘what’, ‘when’ and ‘where’ functionalities are implemented
 - common terminology – remain mindful about non-RF engineers and applications
 - **aim: intuitive design before domain-language before documentation of concepts**
 - common understanding of dependencies and interfaces
 - directed flow-graphs are great low-/high-level representations (‘mechanical sympathy’)
 - aim for the rest: present C++ STD → C++ Core Guidelines → C++ Best Practices*, ...

*e.g. “[Make Your API Hard To Use Wrong](#)”, Scott Meyers, IEEE Software, July/August 2004

Software must be adaptable to frequent changes

- mechanical sympathy – why GR flow-graphs resonate well with RF engineers



Δ -Signal

full-range
0.4 GHz

0.8 GHz
1.2 GHz

1.6 GHz
full-range

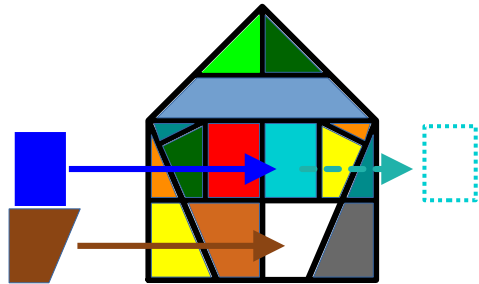
2. Clean- and Lean Code-Base Redesign

favour 'composition' over 'inheritance'

- low-level library: 'what', 'when' and 'where' functionalities are implemented
 - safe, secure and better performance @ IO- and memory latency & bandwidths limits
 - only pay for what you use (aka. 'zero-overhead principle')
 - compile-time type-safety & concepts are overhead free ↔ virtual inheritance & RTTI aren't
 - modern, lean-and-clean support of exchangeability & extendability through 'composition'

→ a) stronger separation-of-concern, transparent & 'intuitive' design*

→ b) light-weight, minimal, reduced to strictly-needed API & open for user-extensions



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

modular library: user can opt-in what to use and what is needed
...free to extend, modify, synthesis new ideas

*from a perspective of novice/new users with some RF, signal-processing, computer-science background

2. Clean- and Lean Code-Base Redesign

virtual inheritance vs. strict typing & concepts: <https://compiler-explorer.com/z/fe5Khcxvf>

The image shows three side-by-side windows in Compiler Explorer, each displaying a different C++ implementation and its assembly output.

Window 1 (Left): Virtual Inheritance

```
1 // via virtual inheritance
2 struct base {
3     const int data;
4     base(int val) : data(val) {}
5     virtual int get() { return data; };
6 };
7
8 struct derived : public virtual base {
9     derived(int val) : base(val + 41) {}
10    int get() override { return data; }
11 };
12
13 int func_virtual_inheritance(base& a) noexcept {
14     return a.get();
15 }
16
17 int main(int argc, const char**) {
18     derived d(argc);
19     return func_virtual_inheritance(d);
20 }
```

Window 2 (Middle): Concepts

```
1 // via concepts
2 #include <concepts>
3
4 template<class T>
5 concept Base = requires(T t) {
6     { t.get() } -> std::same_as<int>;
7 };
8
9 struct my_class {
10    const int data;
11    my_class(int val) : data(val + 41) {}
12    int get() { return data; }
13 };
14
15 int func_concepts(Base auto& a) noexcept {
16     return a.get();
17 }
18
19 int main(int argc, const char**) {
20     my_class c(argc);
21     return func_concepts(c);
22 }
```

Window 3 (Right): Template Class with Concept

```
1 main:
2     lea    eax, [rdi+41]
3     ret
```

The assembly output for the first two windows shows the generation of a vtable for the base class and the derived class, and the execution of the virtual function call. The third window shows a simple function call and return.

2. Clean- and Lean Code-Base Redesign

strict typing & concepts – block implementation as the single source of truth derive

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

Modernisation Goals

improve performance, industrial integration+deployment

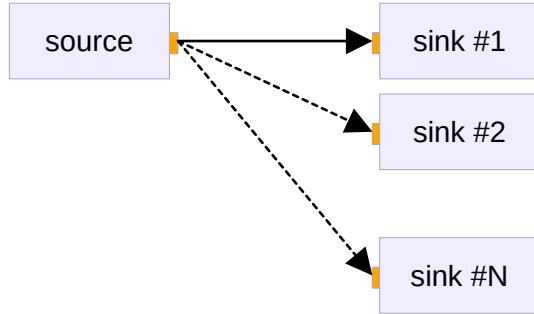
- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign
- 3. Performance Optimisations**
 - high-performance, type-strict IO buffers
 - zero-overhead for graphs known at compile-time
 - out-of-the-box hardware acceleration (SIMD, GPU, etc.)
 - optimise linear flow dependency sub-graphs (e.g. avoid/minimise need for buffers)
- 4. Tag-Based Timing System Integration
- 5. Advanced Processing Features
- 6. Broaden Cross-Platform Support
- 7. User-pluggable Work Scheduler Architecture
- 8. Overall Project Direction



3. Performance Optimisations

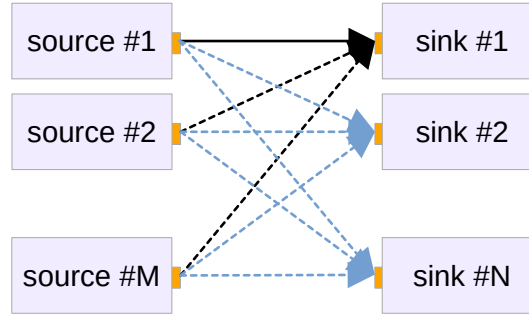
new high-performance, type-strict IO buffers – Possible Use-Cases

Fan-Out:



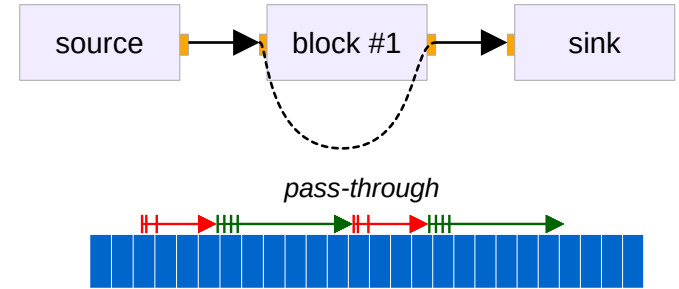
- multiple observer
- classic GR flow-graph use

Fan-In/ Aggregate:



- message passing
- decoupling between user-vs. real-time worker threads, e.g.
 - PMT block property updates from stream tags & user-thread

Multi-Cascade:



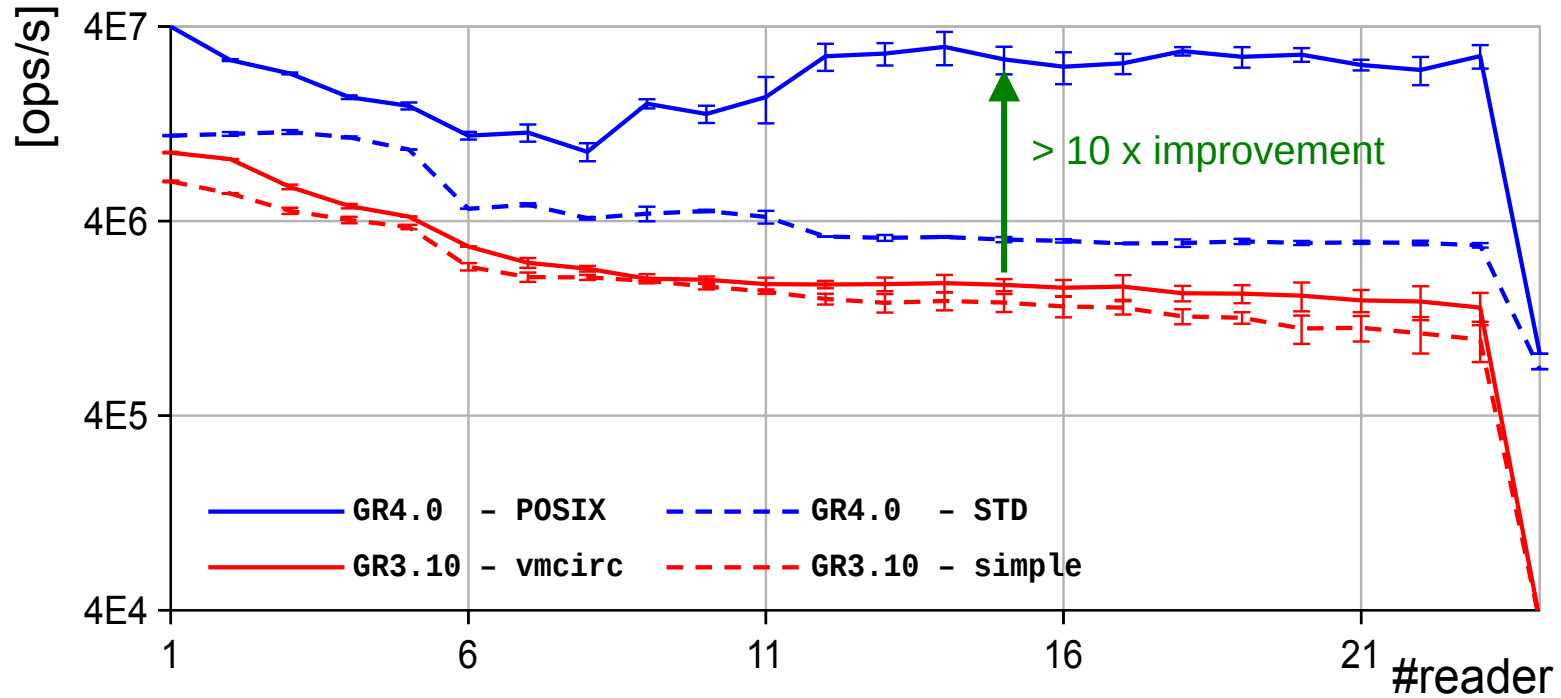
- cascaded reader/writer sharing same buffer → minimises copying
- good for blocks that monitor and rarely modify data

Important (hopefully positively perceived) changes:

- *type-strictness: new circular_buffer can propagate any type i.e. fundamentals but notably also aggregate types → e.g. DataSet<T>*
- *simplified async message and sync stream handling (i.e. the same)*

3. Performance Optimisations

high-performance, type-strict IO buffers



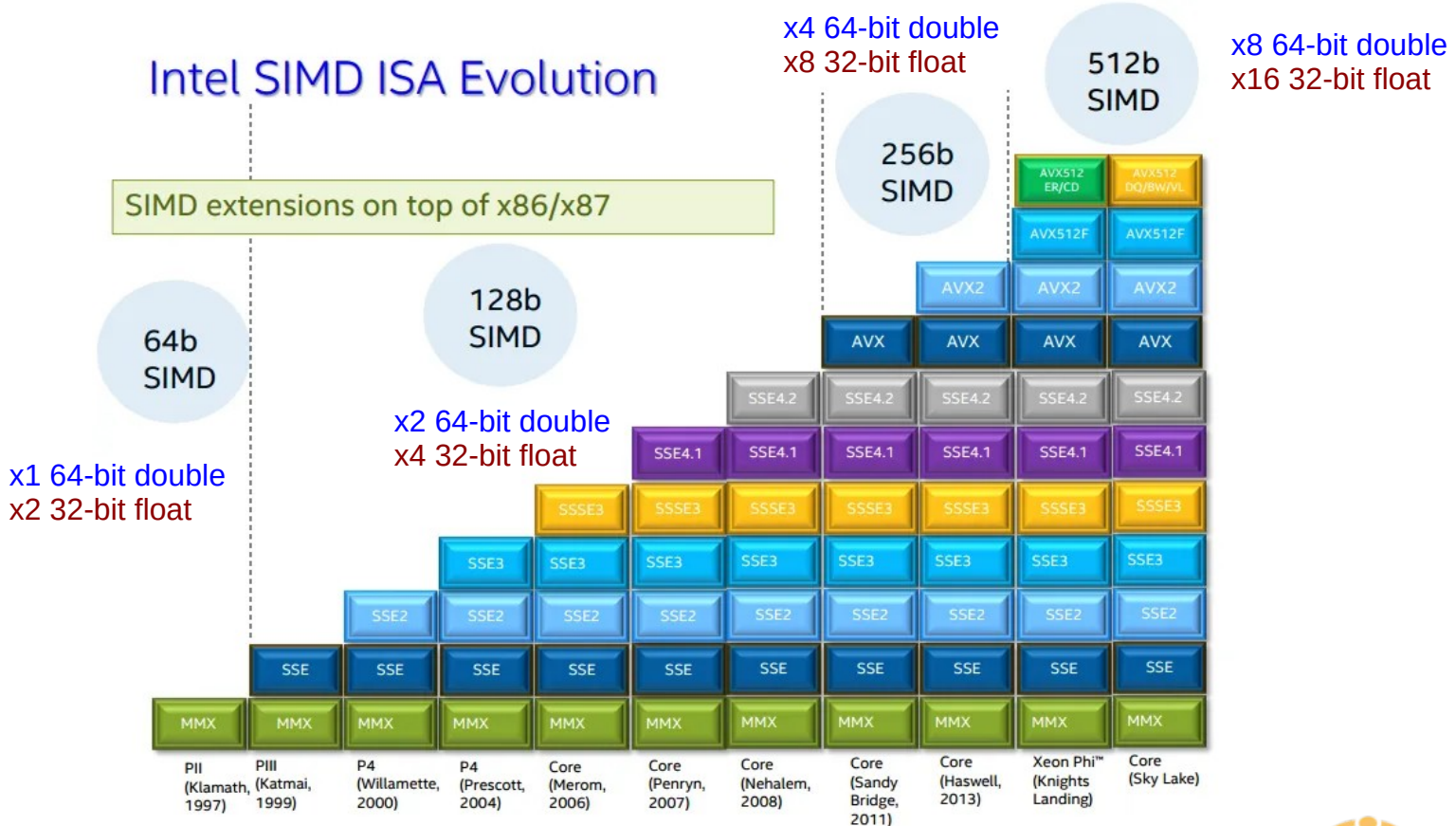
main key-ingredients:

- made new `circular_buffer<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

Performance Optimisations

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



<https://en.algorithmica.org/hpc/simd/>



Performance Optimisations

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

SIMD: ‘Single Instruction, Multiple Data’

- utilise all parallelism per CPU core
(N.B. code often utilises only ~10% of the CPU die!!)
- more efficient use of memory bandwidth (and caches)
- reduces latency ↔ real-time systems
- improves efficiency and FLOP/power ratio
- portable & intuitive design of data-parallel blocks
- C++ dev can focus on algorithms/physics
- **significant improvements depending on algorithm**

Compile-time merging of blocks

- forgo buffers if connection is known at compile-time
- enables compiler to “see” and optimise merged algorithm
- avoids loads & stores ⇒ less memory/cache required
- avoids synchronisation costs

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)->div(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 - process_one(..)	24k / 663k = 4%	105 us	259 us	882 us	97.5M
runtime src->mult(2.0)->div(2.0)->add(-1)->sink - process_bulk(..)	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)

Big shout-out to: Dr. Matthias Kretz (GSI/FAIR) & C++ ISO Committee SG6 Numerics Chair for adopting us/GR and sponsoring the lib <simd> (↔ will be part of C++26)



Modernisation Goals

improve performance, industrial integration+deployment

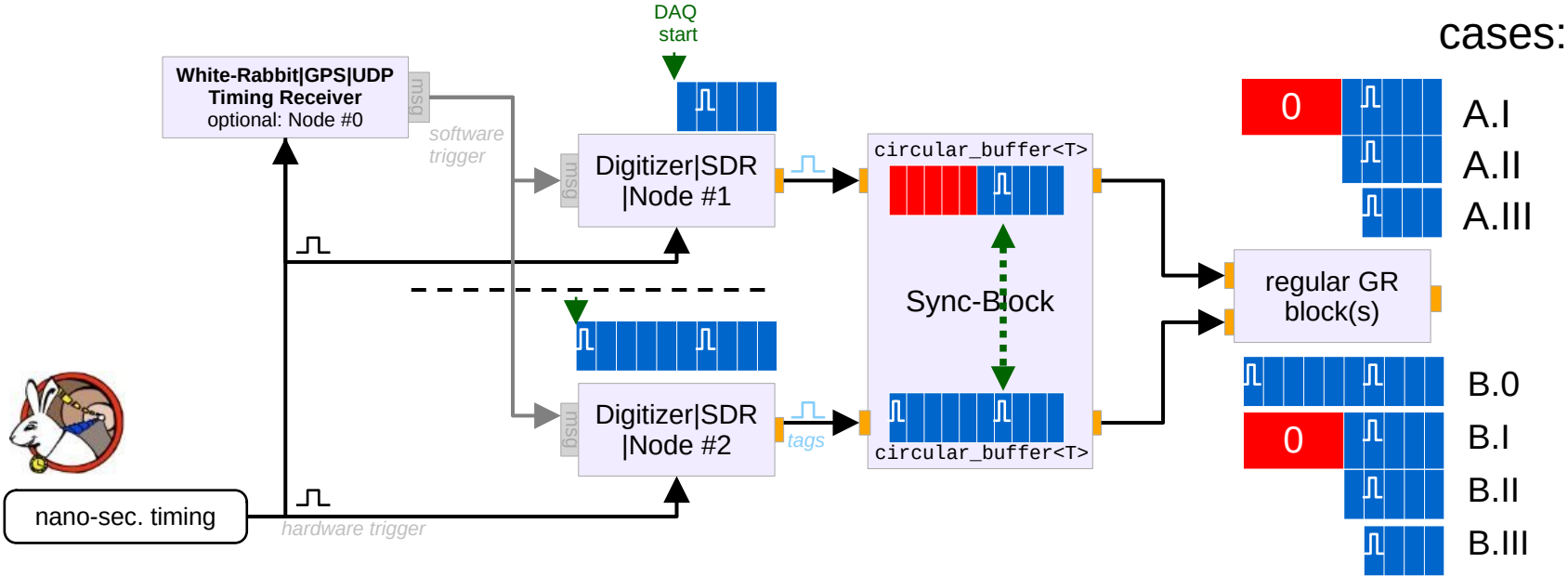
- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign
- 3. Performance Optimisations
- 4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)**
- 5. Advanced Processing Features
- 6. Broaden Cross-Platform Support
- 7. User-pluggable Work Scheduler Architecture
- 8. Overall Project Direction



4. Tag-Based Timing System Integration

synch. data (streams) from different flow-graphs & nodes

- 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_t’s;
 TRIGGER_NAME, TRIGGER_TIME, TRIGGER_OFFSET



Modernisation Goals

improve performance, industrial integration+deployment

- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign
- 3. Performance Optimisations
- 4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)

5. Advanced Processing Features

- transactional and multiplexed settings
- synchronous chunked data processing (for event-based and transient-recording signals)

- 6. Broaden Cross-Platform Support
- 7. User-pluggable Work Scheduler Architecture
- 8. Overall Project Direction



5. Advanced Processing Features

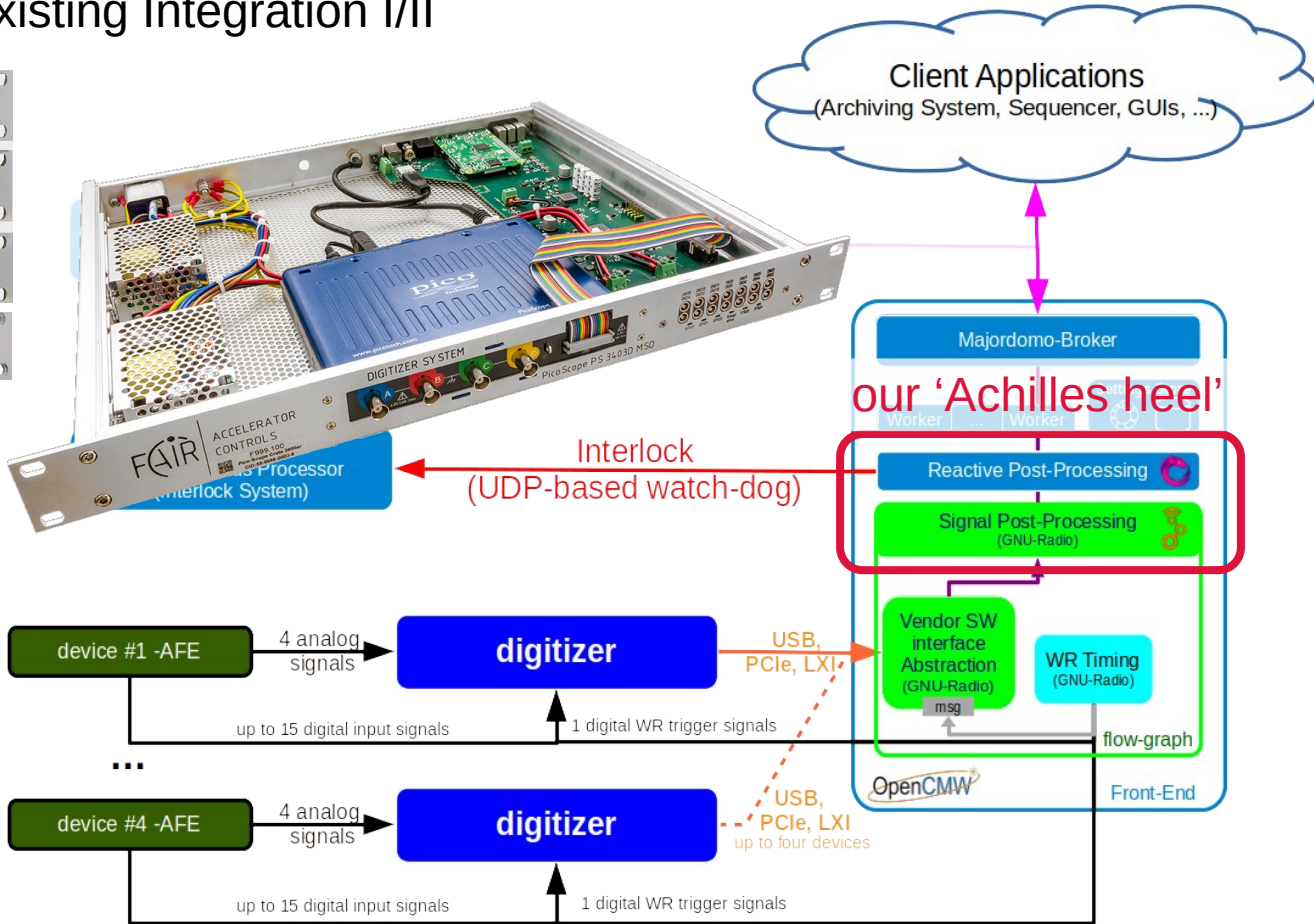
Setting the scene – Issue with the existing Integration I/II



... open-hardware but not exclusive standard at FAIR:
hundreds of other digitizers supported thanks to GNU Radio

Primary OpenDigitizer Applications:

- A) First-line diagnostics
↔ “distributed ns-level synchronised oscilloscope/SDR/DSA/...”
- B) Building blocks for higher-level diagnostics, monitoring, and feedback systems
- C) Rapid Prototyping: accelerate integration of R&D prototype into robust 24h/7 operation

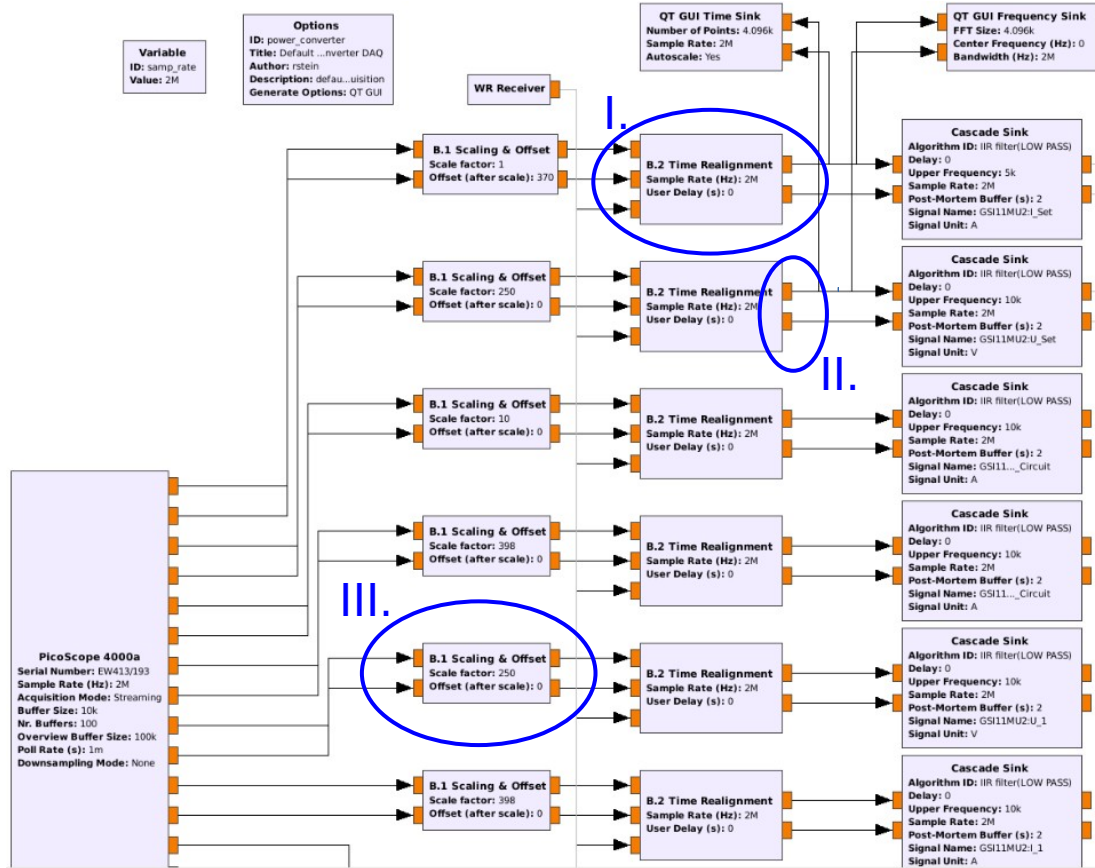


>500 DAQs & post-processing/monitoring feedback services
all sharing the same OpenCMW, GNU Radio, OpenDigitizer, and UI/UX software stack



5. Advanced Processing Features

Setting the scene – Issue with the existing Integration II/II

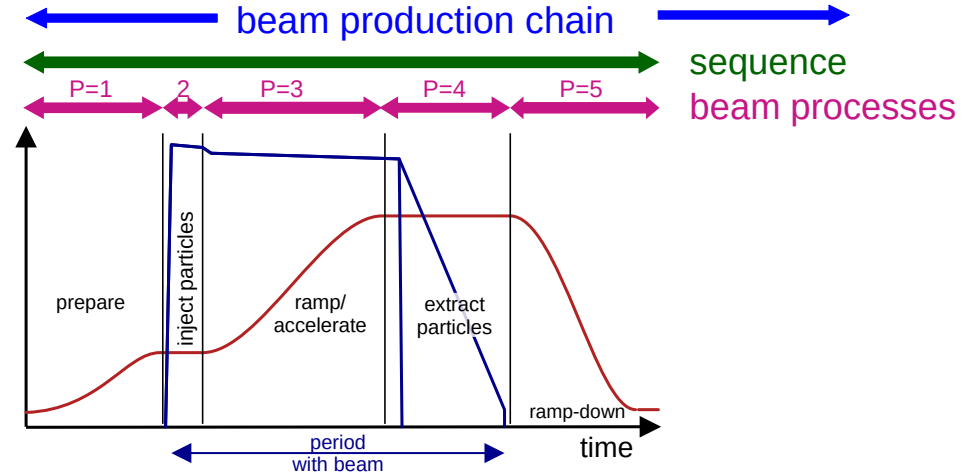
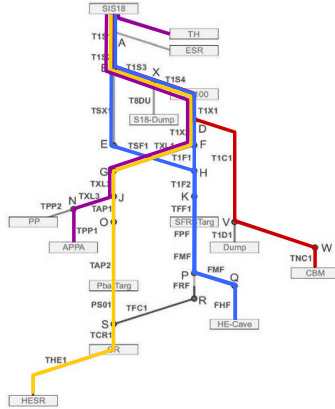
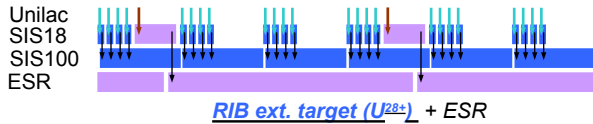
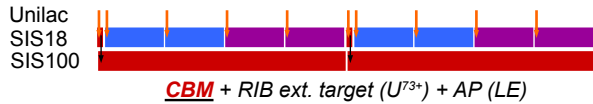
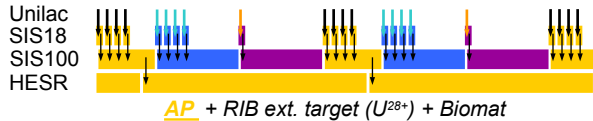


Three noteworthy things:

- I. ns-level signal synchronisation across 300++ front-end controllers (FECs) via (<https://github.com/fair-acc/gr-digitizers>)
 - a) 'White-Rabbit' timing receiver
 - b) GPS pps signals
 - c) SW-trigger (i.e. UDP multicast)
- II. mean + stdev processing
 - a) ... scientific rigour
 - b) ... signal-integrity checks
 - ↔ used in feed-back loops (automatic stop/fail-safe)
- III. run-time flow-graph modifications (<https://github.com/fair-acc/gr-flowgraph>)
 - a) block parameters (e.g. gains, timing-triggered threshold/interlock functions, χ^2 -fits, conditional processing, ...)
 - b) online- & user-defined post-processing (~T&M equipment)

5. Advanced Processing Features

transactional and multiplexed settings – FAIR/CERN/... are multi-mission/user platforms

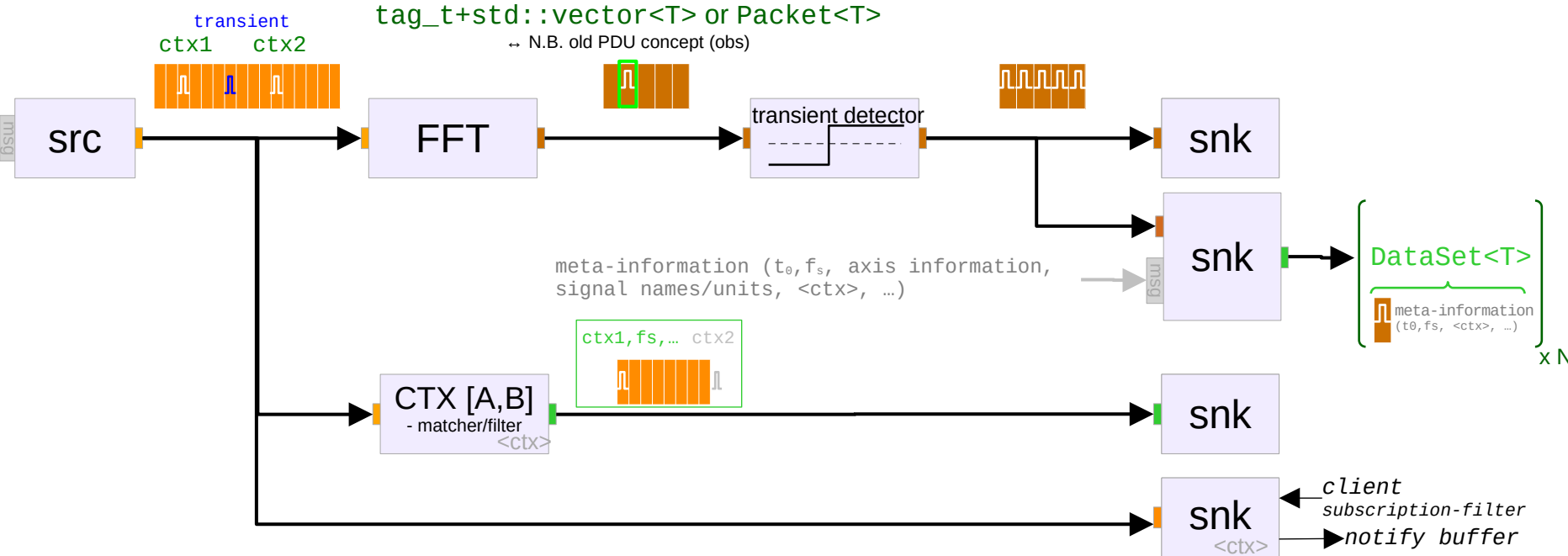


- Device/Block Settings Challenges:
 - frequent synchronised settings changes (10k+ devices!)
 - require dynamic coarse → fine-grained scope
 - data transport & signal processing group-delays
- settings need to be synchronised & multiplexed
- solution: adaptive timed B+-tree + transactions (see appendix)

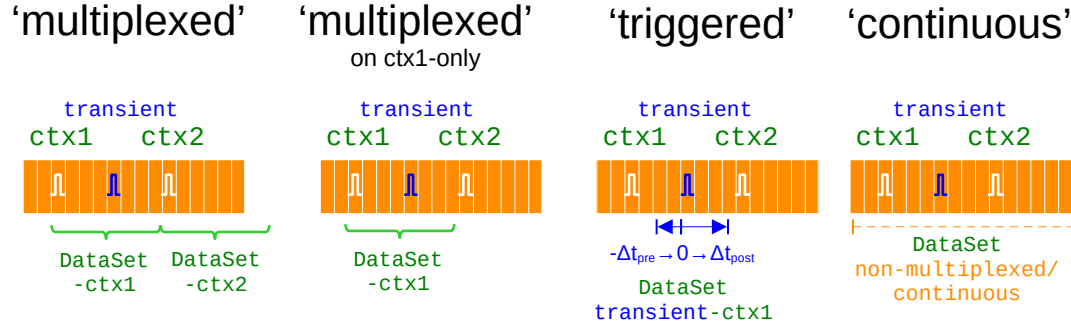


5. Advanced Processing Features

synchronous chunked data processing → new DataSet<T>

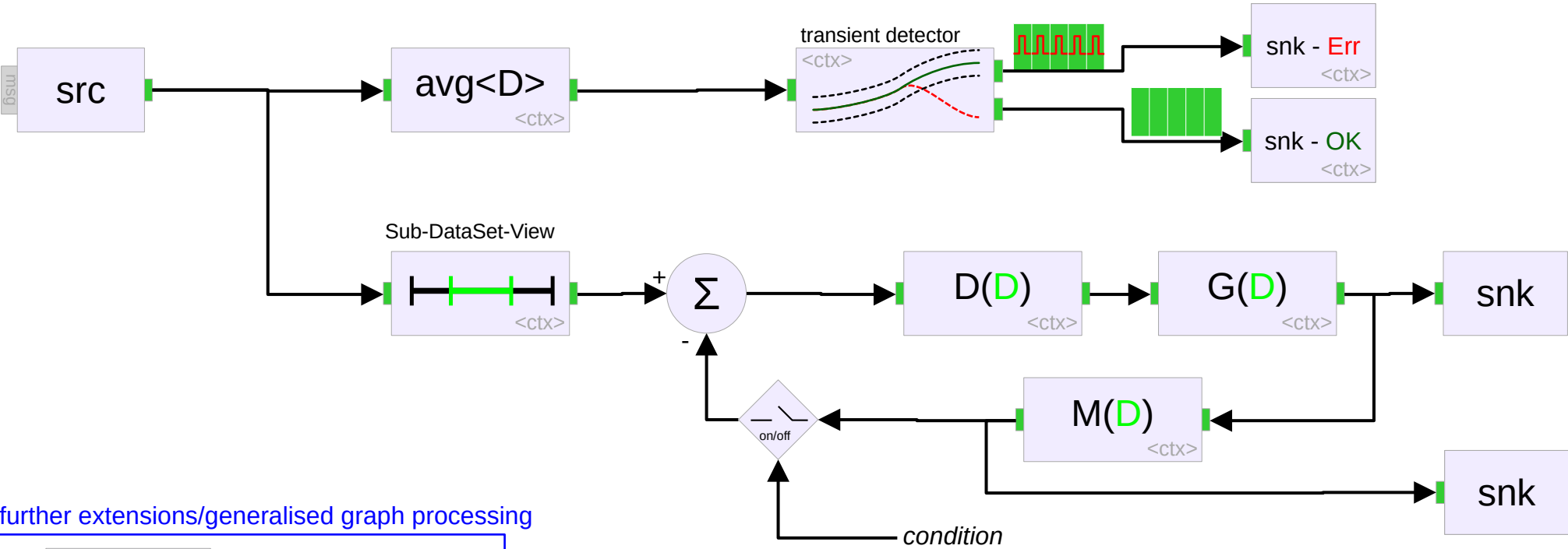


filter modes:



5. Advanced Processing Features

synchronous chunked data processing → new DataSet<T>



further extensions/generalised graph processing

- ⌚ for loop <ctx>
- ⌚ while loop <ctx>

???

5. Advanced Processing Features

synchronous chunked data processing, three new types: `Packet<T>` → `Tensor<T>` → `DataSet<T>`

```
template<typename T>
struct DataSet { // – numeric/measurement based data (e.g. generation of graphs/plotting)
    Packet
    | Tensor
    | std::int64_t timestamp = 0; // UTC timestamp [ns]
    // axis layout:
    std::vector<std::string> axis_names; // e.g. time, frequency, ...
    std::vector<std::string> axis_units; // axis base SI-unit
    std::vector<std::vector<T>> axis_values; // explicit axis values

    // signal data layout:
    std::vector<std::int32_t> extents; // extents[dim0_size, dim1_size, ...]
    std::variant<layout_right, layout_left, std::string> // row-major, column-major, "special"
    layout;

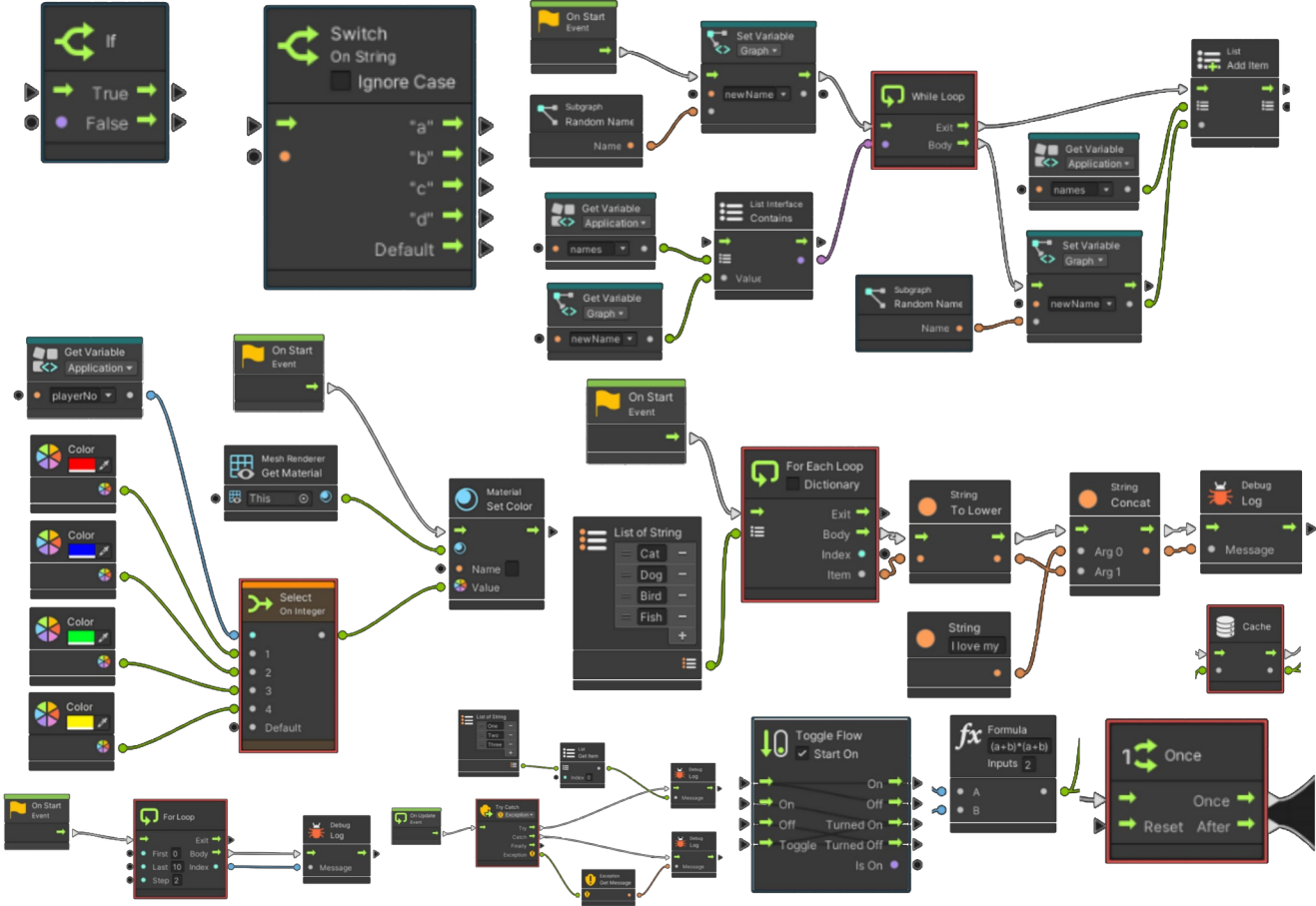
    // signal data storage:
    std::vector<std::string> signal_names; // size = extents[0]
    std::vector<std::string> signal_units; // size = extents[0]
    std::vector<T> signal_values; // size = \PI_i extents[i]
    std::vector<T> signal_errors; // size = \PI_i extents[i]
    std::vector<std::vector<T>> signal_ranges; // [[min_0, max_0], [min_1, ...]

    // meta data
    std::vector<std::map<std::string, pmt::pmtv>> meta_information;
    std::vector<std::map<std::int64_t, pmt::pmtv>> timing_events; // ↔ gr::tag_t
    // [...] constructors, accessors, ...
};
```

Future Vision/Extension: Inspiration from Unity Control Node ...

Basic Scripting of more complex signal flow/processing mechanisms

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



Modernisation Goals

improve performance, industrial integration+deployment

- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign
- 3. Performance Optimisations
- 4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)
- 5. Advanced Processing Features
- 6. Broaden Cross-Platform Support (including WebAssembly)**
- 7. User-pluggable Work Scheduler Architecture
- 8. Overall project direction



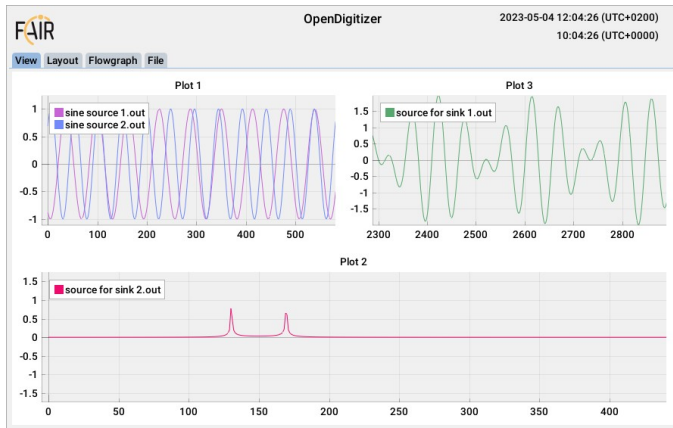


6. Broaden Cross-Platform Support

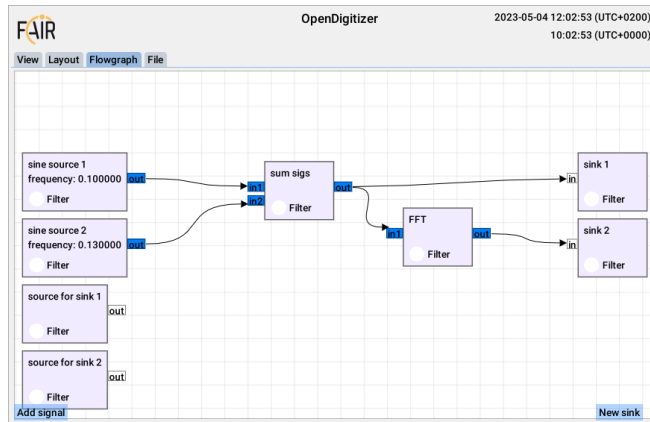
emphasis on GCC, Clang & Emscripten (↔ WASM/WebAssembly, UI) Support

- UI tooling is important for adoption, debugging and as a real world benchmark
→ core component of OpenDigitizer reimplementation.
- Simple to use basic functionality for day to day usage but no limitations for troubleshooting and expert users
 - direct access to the processing flowgraphs.
 - aim at full compatibility with GNU Radio Companions ‘.grc’ file format
- Images show the current state of the working implementations and are subject to further development.

Default dashboard view (editable)



Display and modify service and post-processing flow graphs



Store and load custom dashboards

The screenshot shows the OpenDigitizer dashboard management interface. It displays a list of dashboards with columns for name, URL, and last used date. The interface includes a menu bar (View, Layout, Flowgraph, File) and a status bar with the date and time.

Dashboard Name	URL	Last used
dashboard2	http://localhost:8080/dashboards	04/05/2023
dashboard1	http://localhost:8080/dashboards	23/12/0006
dashboard3	http://localhost:8080/dashboards	23/12/0006

Modernisation Goals

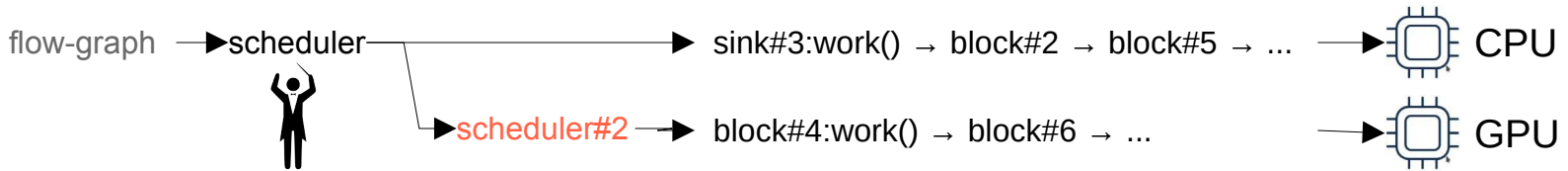
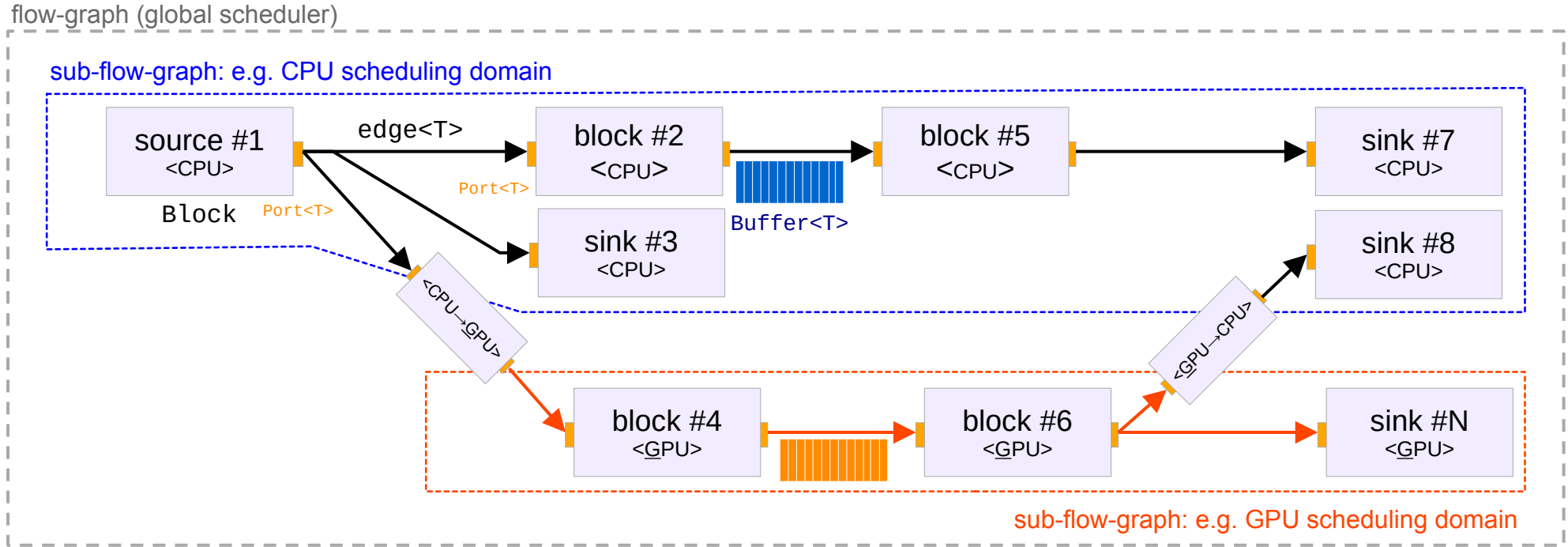
improve performance, industrial integration+deployment

- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign
- 3. Performance Optimisations
- 4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)
- 5. Advanced Processing Features
- 6. Broaden Cross-Platform Support (including WebAssembly)
- 7. User-pluggable Work Scheduler Architecture** adaptable to
 - domain (e.g., CPU, GPU, NET, FPGA, DSP, ...)
 - scheduling constraints – throughput vs. latency constraints
- 8. Overall Project Direction



7. User-pluggable Work Scheduler Architecture

Simplified Graph Topology adaptable to domain (e.g., CPU, GPU, NET, FPGA, DSP, ...)



7. User-pluggable Work Scheduler Architecture

Original Scheduler definition: <https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91>

- The scheduler interface is responsible for execution of part (or all) of a flowgraph. Schedulers are assumed to have an input queue and the only public interface is for other entities (either from the runtime or other schedulers) push a message into the queue that can represent some action.
- These messages can be:
 - Indication that streaming data has been produced on a connected port
 - An asynchronous PMT message (indication to run callback)
 - Other runtime control (start, stop, kill)

to note:

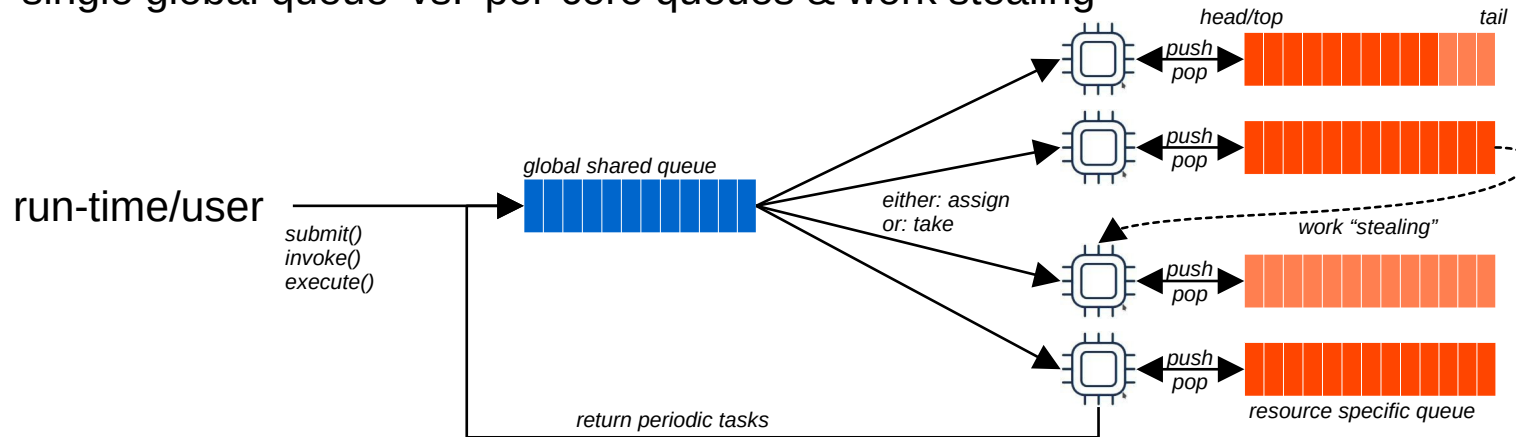
- description is effectively of an 'orchestrator' within a 'microservice architecture' (alt) using a message passing system to synchronising individual service task.
- message-passing has it's costs and is not the most effective pattern for signal-processing

→ invert the dependency hierarchy and adopt existing scheduler designs to the problem

7. User-pluggable Work Scheduler Architecture

Modified Work Scheduler Paradigm/Proposal building upon that ... I/II

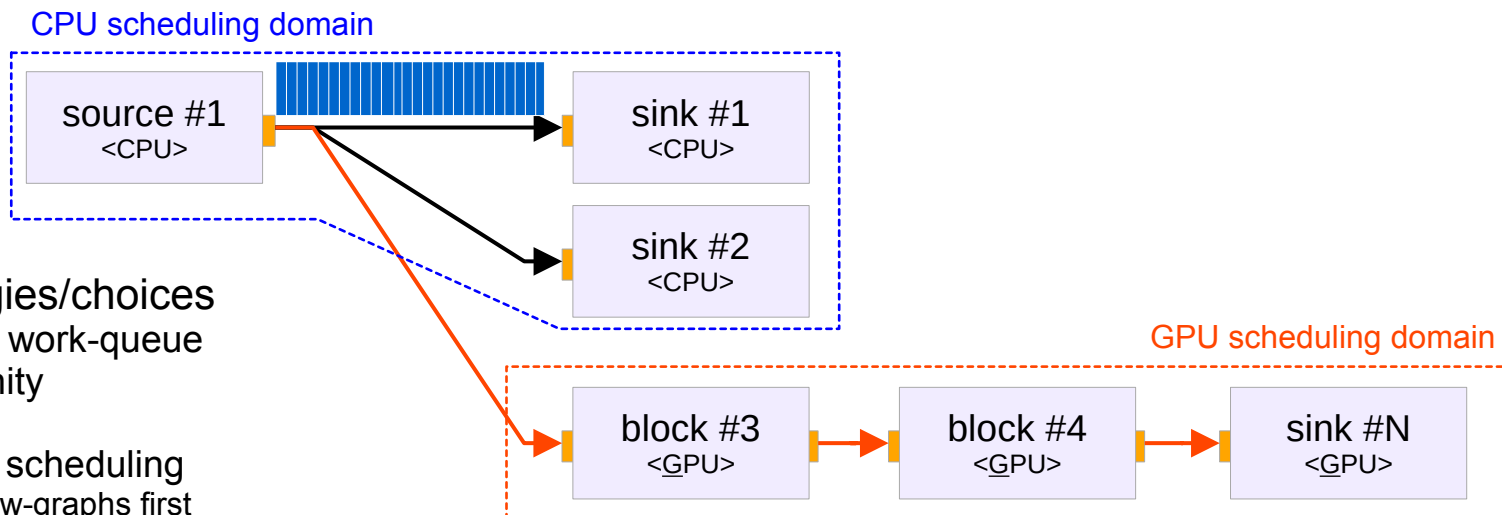
- a scheduler is a process that assigns a task i.e. ``block::work()` function to be executed on available computing resources (CPU|GPU|...).
 - A) ``work()` encapsulates impl. specific ``work(wio)` function ($wio \leftrightarrow$ ports, connection, buffers, ...)
 - B) only non-blocking work functions, and
 - C) only as many threads as there are available computing resources
 - one core can execute only one thread at a time
 - avoids unfair/non-deterministic scheduling, context-switching & keeps L1/L2/L3 caches hot \leftrightarrow CPU shielding/affinity
- high-level scheduler implementation specific design choices: 'single global queue' vs. 'per-core queues & work stealing'



7. User-pluggable Work Scheduler Architecture

Modified Work Scheduler Paradigm/Proposal building upon that ... II/II

- need to be mindful that we need multiple distinct scheduler for, e.g.
 - CPU: default, fair, real-time, $O(1)$, ... (e.g. prefer small data chunks \leftrightarrow L1/L2/L3 cache & SIMD performance)
 - GPU: ... (e.g. large chunks crossing CPU-GPU boundary, small for parallelising in-GPU processing \leftrightarrow >500 cores)
- scheduling decision needs to be done by scheduling thread (N.B. 'by block worker' only as fall-back)
- different scheduling strategies use different prioritisation & graph-based queues

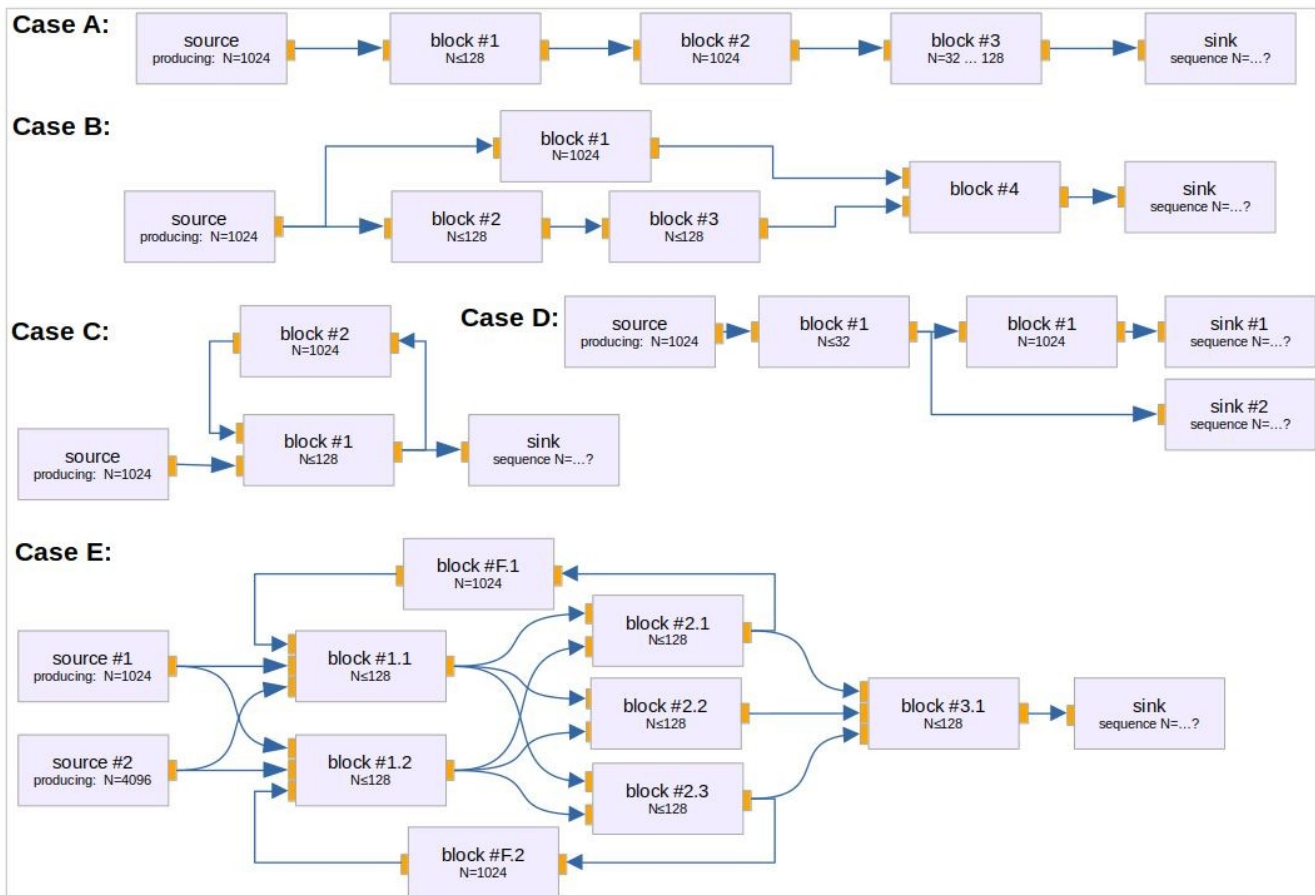


some scheduling strategies/choices

- global vs. per-thread/core work-queue
- CPU shielding/thread affinity
- **static scheduling**
- round-robin vs. prioritised scheduling
 - dependent/pre-requisite flow-graphs first
 - real-time vs. non-real-time sub-flow-graphs
 - data chunk-size based

7. User-pluggable Work Scheduler Architecture

Some Topologies specific designed to trip-up schedulers 🤩 😇



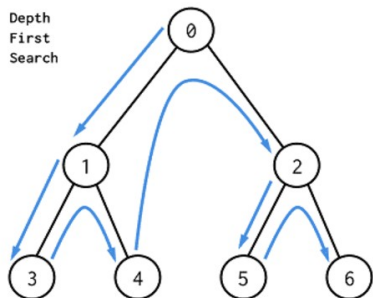
exercise:
what is the correct, best, and most efficient execution order?

7. User-pluggable Work Scheduler Architecture

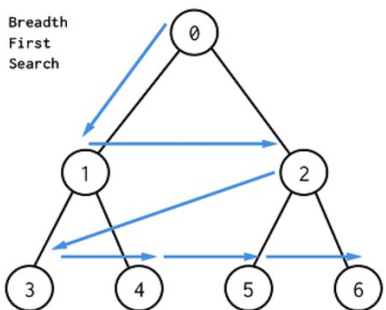
Implemented initially only the most basic scheduler strategies to test and verify API

0. Busy-Looping → naive implementation

1. Depth-first



2. Breadth first



Other possible Algorithms:

<https://github.com/fair-acc/graph-prototype/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 the outlined benchmark topologies.

Big shout-out to: Alexander Krimm for fleshing out the first PoC schedulers (pls. buy him a beer)

Modernisation Goals

improve performance, industrial integration+deployment

- 1. Preserve and Grow the existing diverse GR Ecosystem.
- 2. Clean- and Lean Code-Base Redesign
- 3. Performance Optimisations
- 4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)
- 5. Advanced Processing Features
- 6. Broaden Cross-Platform Support (including WebAssembly)
- 7. user-pluggable work scheduler architecture adaptable to

8. Overall Project Direction



9. Overall Project Direction

Main Theme: Powerful CPU Core

- Addressing HPC limitations identified in dev-4.0
- [Building upon Josh's et al. foundation \(see his intro\)](#)
- Optimising performance and scalability
 - [Scheduler User Challenge](#)
- **Usability Enhancements:
Classic GNU Radio Look & Accessibility**
- Integrating trusted design features of traditional GR
- Simplifying & improving user interface for efficiency
- Expanding accessibility and user-friendly features

Steps for graph-prototype → GR 4.0

<https://github.com/fair-acc/graph-prototype>

1. C++ GR Framework: establishing a robust and flexible core foundation, follow-us on:
<https://github.com/orgs/fair-acc/projects/5/views/1>
<https://github.com/fair-acc/opendigitizer/issues/46>
2. GRC Integration: aligning functionalities for a cohesive user interface.
3. Python Integration: Harnessing the capabilities of Python for extended functionality.



9. Overall Project Direction

The next Frontier – FPGA Integration

Need:

- Ensuring agile real-time signal processing capabilities
- Efficiently integrating with low-level RF feedback systems

Challenge:

- Transitioning from semi-static firmware configurations
- Overcoming dependencies on proprietary tool-chains
- Simplifying the deployment process for diverse users

Vision:

- Advancing FPGA capabilities for dynamic adaptability
- Supporting real-time reconfiguration w/o disruptions
- Unified platform for both SW- and HW-processing

Modernisation Goals

improve performance, industrial integration+deployment

- 1. **Preserve and Grow the existing diverse GR Ecosystem.**
- 2. **Clean- and Lean Code-Base Redesign**
- 3. **Performance Optimisations**
- 4. **Tag-Based Timing System Integration**
- 5. **Advanced Processing Features**
- 6. **Broaden Cross-Platform Support (including WebAssembly)**
- 7. **User-pluggable Work Scheduler architecture**
- 8. **Overall Project Direction**



Thank You!

looking forward to:



Looking forward to technical dialogue
and building partnerships ... Questions?



Appendix

Modernisation Goals

improve performance, industrial integration+deployment

1. Preserve and Grow the existing diverse GR Ecosystem.

- thin Python interface over C++ API
- avoid Python-only implementations (except OOT modules)
- swappable runtime components (both in and out of tree)
- simplified block development: get block developers to "insert code here" without lots of boilerplate or complicated code

2. Clean- and Lean Code-Base Redesign

- favour 'composition' over 'inheritance'
- boosts maintainability and adaptability
- preserve tried-and-tested functionalities

3. Performance Optimisations

- high-performance, type-strict IO buffers
- zero-overhead for graphs known at compile-time
- out-of-the-box hardware acceleration (SIMD, GPU, etc.)
- optimise linear flow dependency sub-graphs (e.g. avoid/minimise need for buffers)

4. Tag-Based Timing System Integration (White Rabbit, GPS, SW-based etc.)

5. Advanced Processing Features

- transactional and multiplexed settings
- synchronous chunked data processing (for event-based and transient-recording signals)

6. Broaden Cross-Platform Support (including WebAssembly)

7. User-pluggable Work Scheduler Architecture adaptable to

- domain (e.g., CPU, GPU, NET, FPGA, DSP, ...)
- scheduling constraints (throughput, latency, ...)

8. Overall project direction



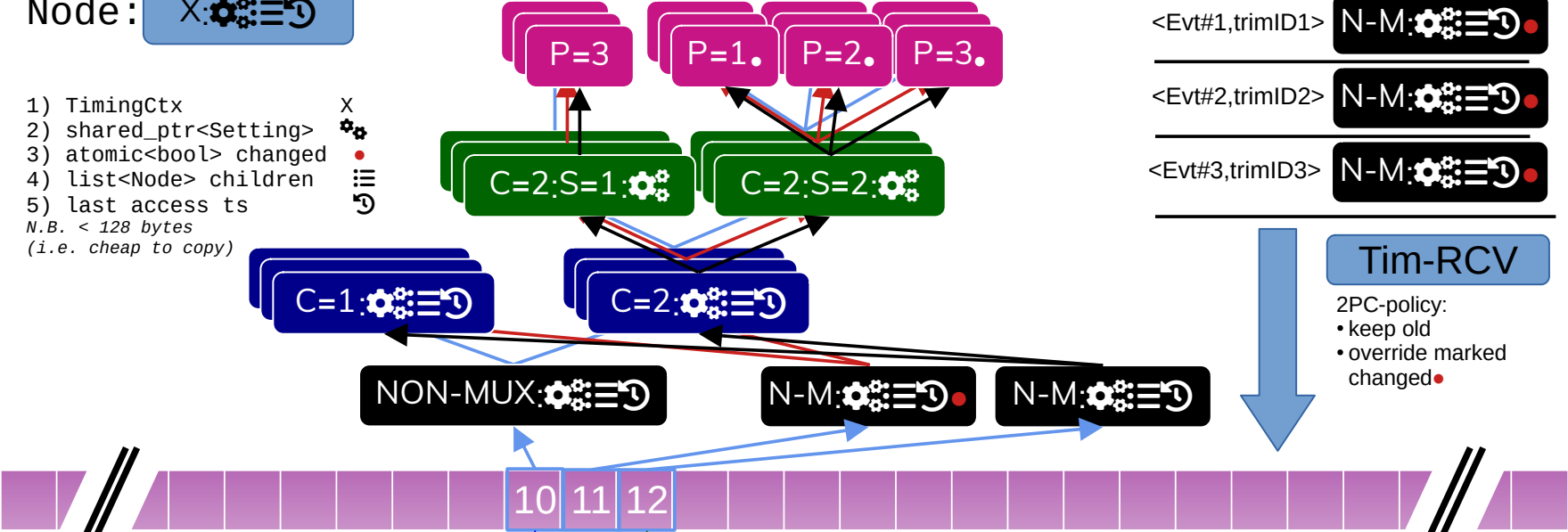
5. Advanced Processing Features

transactional and multiplexed settings – combine sample-by-sample & chunked signal processing

```
'FAIR.SELECTOR.C=<BPCID>:S=<SID>:P=<BPID>:T=<GID>'
```



- 1) TimingCtx
 - 2) shared_ptr<Setting>
 - 3) atomic<bool> changed
 - 4) list<Node> children
 - 5) last access ts
- N.B. < 128 bytes
(i.e. cheap to copy)



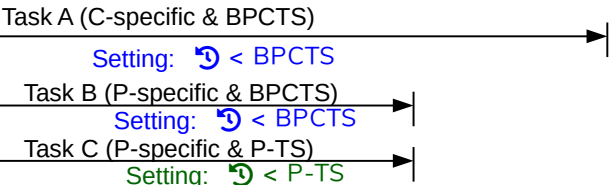
transaction shelve



Tim-RCV

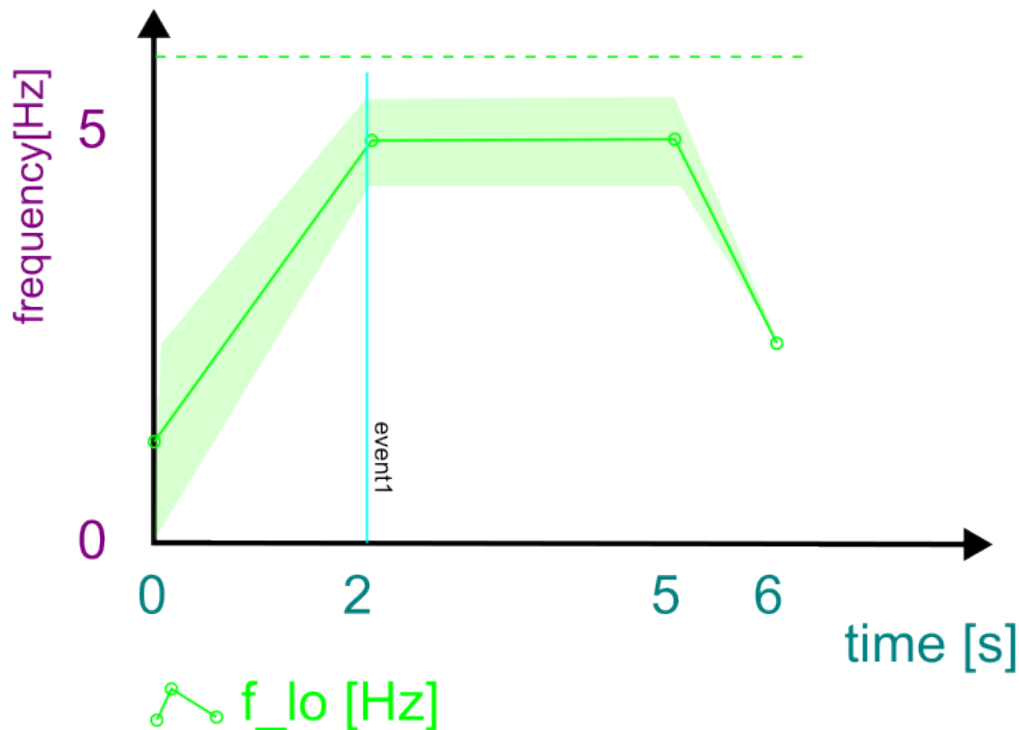
2PC-policy:

- keep old
- override marked changed



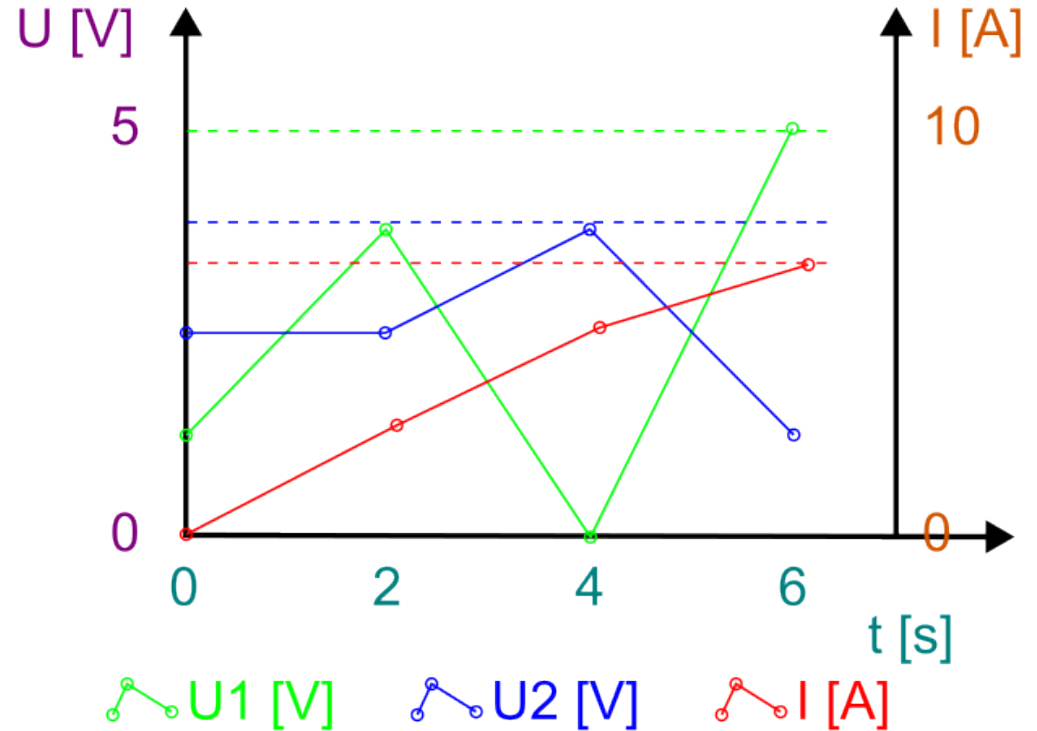
(3) # - - - DataSet<T> – Example: 1-dim function

```
{  
  timestamp: 123456789; // [ns]  
  axis_names: ["time", "frequency"];  
  axis_units: ["s", "Hz"];  
  axis_values: [[0, 2, 5, 6], [0, 5]];  
  extents: [1, 4];  
  layout: layout_right;  
  signal_names: ["f_lo"];  
  signal_units: ["Hz"];  
  signal_values: [1, 5, 5, 2];  
  signal_errors: [1, 0.5, 0.5, 0];  
  signal_ranges: [[0, 6]];  
  signal_status: [{"locked": true}];  
  timing_events: [{123456791: <pmtv>}];  
}
```



(3) # - - - - DataSet<T> – Example: N=3 x 1-dim function

```
{  
  timestamp: 123456789; // [ns]  
  axis_names: ["t", "U", "I"];  
  axis_units: ["s", "V", "A"];  
  axis_values: [[0, 6],[0, 5], [0, 10]];  
  extents: [3, 4];  
  layout: layout_right;  
  signal_names: ["U1", "U2", "I"];  
  signal_units: ["V", "V", "A"];  
  signal_values: [  
    1,3,0,5,  
    2,2,3,1,  
    0,2,4,5];  
  signal_errors: [];  
  signal_ranges: [[0,5],[0,4],[0,5]];  
  signal_status: [];  
  timing_events: [];  
}
```



(3) # - - - DataSet<T> – Example: Image/Matrix/Tensor

```
{
  timestamp: 123456789; // [ns]
  axis_names: ["x", "y", "\phi"];
  axis_units: ["m", "m", "°"];
  axis_values: [[0, 4], [0, 2]];
  extents: [1, 5, 3];
  layout: layout_right;
  signal_names: ["T"];
  signal_units: ["°"];
  signal_values: [
    0, 0, 0, 0, 0,
    0, 2, 3, 2, 0,
    0, 0, 0, 0, 0];
  signal_errors: [];
  signal_ranges: [0,3];
  signal_status: [];
  timing_events: [];
}
```

