Revisiting GNU Radio 4.0’s Low-Level API

(1)#---- explicitly define, document, sharpen separation-of-concern design of

```
flow-graph:{ – dealing with: global topology & scheduling, hot-swap of sub-graphs, …
    sub-graph:{ – domain-specific scheduling of blocks (i.e. CPU|GPU|DSP|FPGA|…)
        block:{ – ‘work(…)’, default+user-extensions/API, history, sched-prios, …
            port:{ – IN|OUT, <T>, CPU|GPU|…, MIN_SAMPLES, (MAX_SAMPLES), …
                buffer:{writer}{reader}}}}}
```
hierarchy

Ralph J. Steinhagen, Ivan Čukić, GNU Radio Meeting – online, 2022-11-10

Finland France Germany India Poland Romania Russia Slovenia Sweden UK

GSI HELMHOLTZ ASSOCIATION FAIR

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

photo courtesy: https://fieldertree.com/

Top level design goals

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- Preserve existing and keep growing a diverse GR eco-system and user-base.
- Keep Python interface a thin wrapper over C++ API
- Avoid Python-only implementations outside of OOT modules
- Modular runtime swappable components both in and out of tree
- Get block developers to "insert code here" without lots of boilerplate or complicated code

```
SLOC    Directory       SLOC-by-Language (Sorted)
14092   gr              cpp=13906,python=186
12139   blocklib        cpp=7530,python=3637,ansic=972
11226   grc             python=11170,sh=56
7150    kernel          cpp=6155,python=995
850     schedulers      cpp=843,python=7
598     bench           cpp=598
126     python          python=126
0       domains         (none)

Totals grouped by language (dominant language first):
cpp:          29032 (62.87%)
python:       16121 (34.91%) (14% w/o grc)
ansic:          972 (2.10%)
sh:              56 (0.12%)

Total Physical Source Lines of Code (SLOC)                = 46,181
Development Effort Estimate, Person-Years (Person-Months) = 11.19 (134.24)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 1.34 (16.09)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))      (GR core: ~4)
Estimated Average Number of Developers (Effort/Schedule)  = 8.34
Total Estimated Cost to Develop                           = $ 1,511,221
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
```

Not a concern …

… end-user Python & C++ top-level block API

non-issues – keep as is:

```python
from gnuradio import gr, module_x, module_y

fg = flowgraph()

b1 = module_x.block_a_f(...)
b2 = module_y.block_b_f(...)
b3 = module_y.block_c_f(...)

fg.connect([b1, b2, b3])
# or fg.connect(b1, "port_name", b2, "port_name")
# or fg.connect(b1, 0, b2, 0)

fg.start()
fg.wait()
```

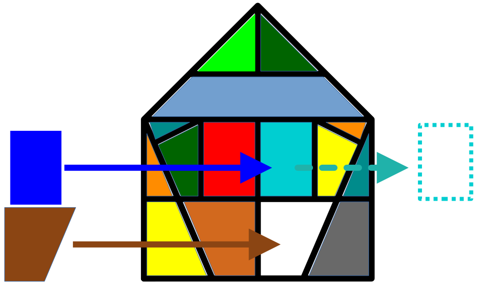
```python
class myblock : gr.block
  def __init__(*args, **kwargs):
    gr.block.__init__(...)


  def work(wio):
    # get np arrays from input ports
    # get mutable np arrays output ports

    # produce and consume

    return gr.work_return_t.OK
```

Primary Goals of proposed API Changes

- 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

Implementation – Performance

virtual inheritance vs. concepts: https://compiler-explorer.com/z/fe5Khcxfv

C++ source #1

```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);
}
```

x86-64 gcc (trunk) (Virtual Inheritance) — x86-64 gcc 12.2, -std=c++20 -O3

```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
```

Output of x86-64 gcc 12.2 (Compiler #1)

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

C++ source #2

```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);
}
```

x86-64 gcc (trunk) (via Concepts) — x86-64 gcc 12.2, -std=c++

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

Output of x86-64 gcc 12.2 (Compiler #2)

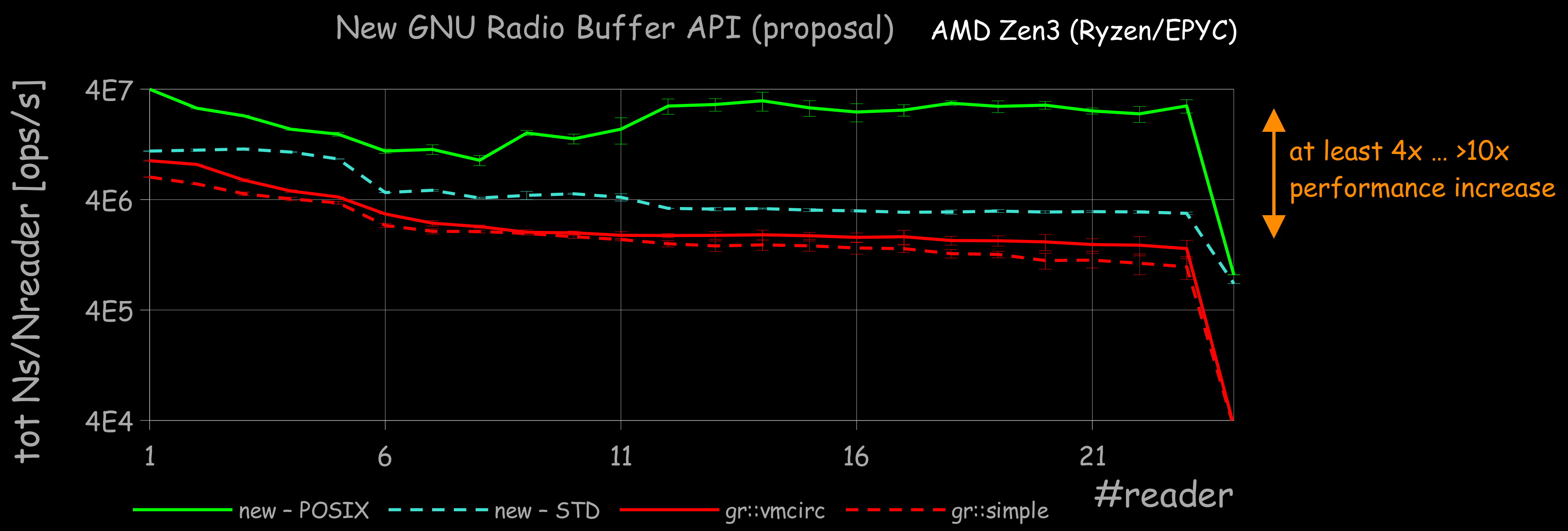
```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 42
```

… at least all that is possible and in the hot-path

`N.B. constexpr != consteval`

source: Ben Deane & Jason Turner "constexpr ALL the Things!", CppCon 2017++

Example: New Buffer API Proposal – Throughput Benchmark (PR)



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

Example: New Buffer API Proposal – Add. Performance Metrics

- throughput performance → 3 … >10x improvement
  - `constexpr`, lock-free, no unnecessary `virtual` calls, ...
- safer & more secure: no raw types, …, harder to misuse API, unit-tested, ...
  - flexibility: pmt-allocators → more portable, reusable & lower-threshold extendable (e.g CUDA buffers <25 SLOCs)
- less code/more focused API: 365 SLOCs (476 lines total, 2 files) vs. 766 SLOCs (1145 lines, 9 files)

→ reduces cognitive complexity for reviewers, maintainers, new code contributors, ...

```
class buffer_properties – 21 methods/constructors
10 mandatory/inherited member fields
[…]

class buffer – 37 methods/constructors
16 mandatory/inherited member fields
[…]

class buffer_reader – 26 methods/constructors
6 mandatory/inherited member fields
[…]
```

Total 84 methods/constructors

developers need to know (+ context how these are being used)

```
concept Buffer: – 4 methods/constructors
       T(min_size)
size_t size()
auto   newReaderInstance() → BufferReader;
auto   newWriterInstance() → BufferWriter;

concept BufferReader: – 4 methods
auto    get(n_items) → std::span<T>
bool    consume(n_items)
int64_t position() → std::int64_t
size_t  available() → std::size_t

concept BufferReader: – 3 methods (+2 optional)
void   publish(WriterCallback, n_items, /* writePos,*/ args…)
bool   tryPublish(WriterCallback, n_items, /*writePos,*/ args…)
size_t available()
```

Before moving ahead … need to slow down and make a step back

Avoiding the risk diving into easy and rather trivial details ...

… while keeping an eye on the full vertical stack and not losing track of the big picture.

Today's goal proposal: sharpen the definition and design of

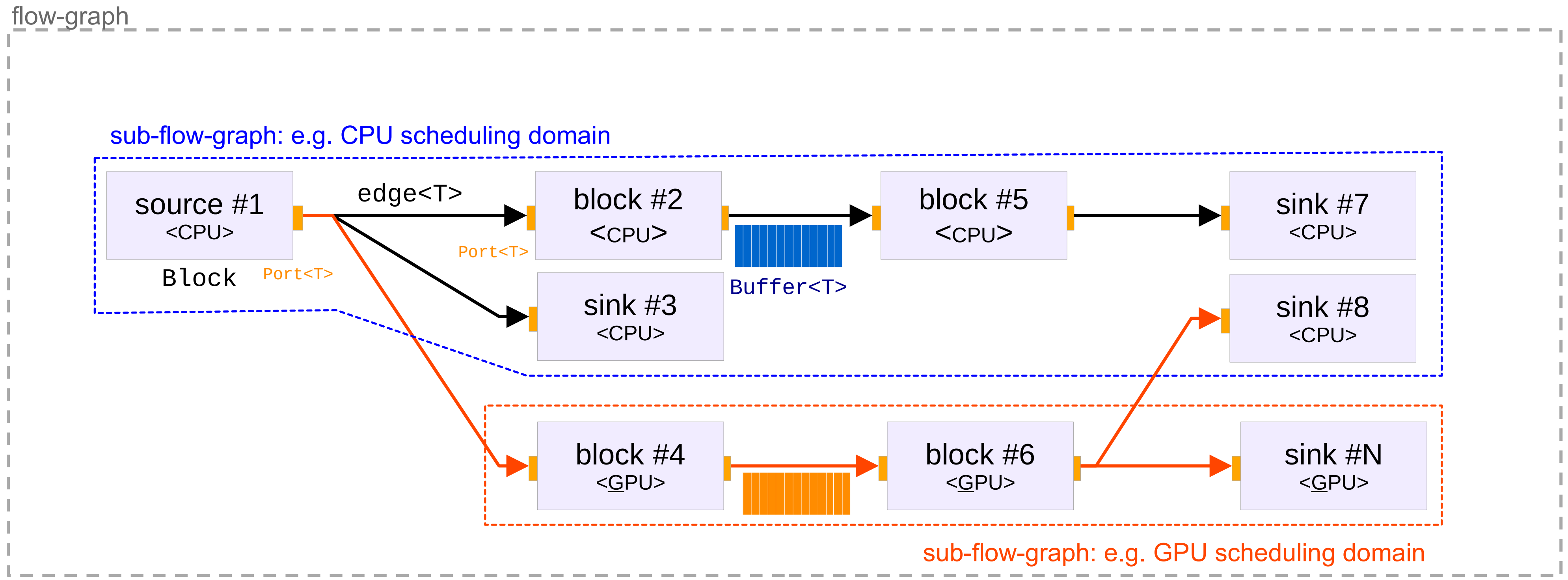
A) what is the primary function of the components:
`buffer` → `port` → `edge` → `block` → `work() & work(wio)` → `(sub_)flow_graph` → `scheduler`

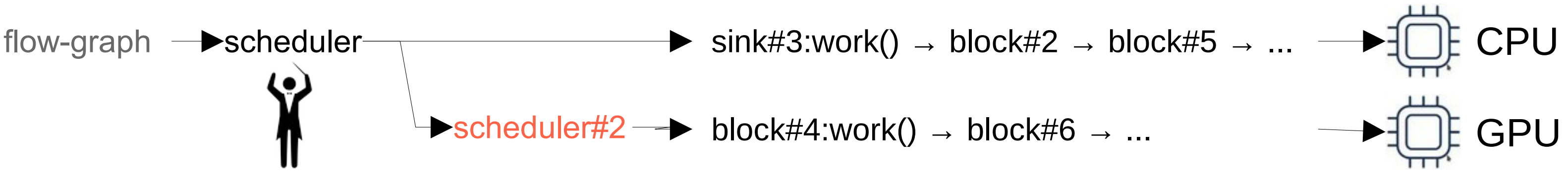
B) what is the minimum information needed per component to fulfil their function?
- smaller envelope → lower cognitive complexity → easier learning, testing, maintenance, exchangeability, security hardening, ...

→ to guide the discussion and difference between:

- top-level functional requirements,
- how these are abstracted into library components & contracts, and
- the specific interface implementation

Simplified Graph Topology





Scheduler

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

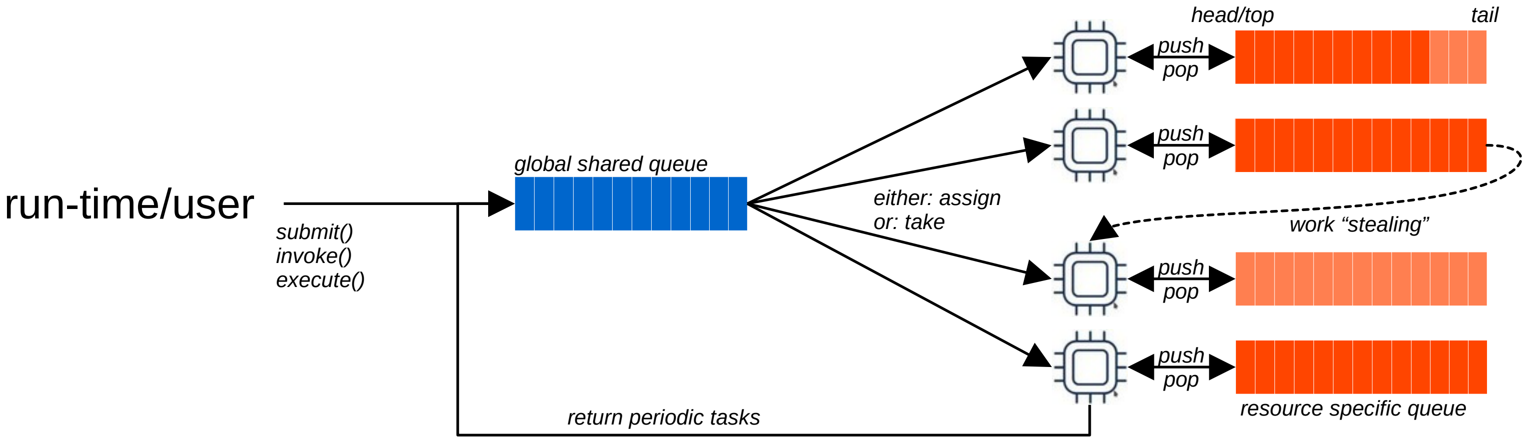
→ invert the dependency hierarchy and adopt existing scheduler designs to the problem

FAIR GSI

Scheduler – Proposal

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

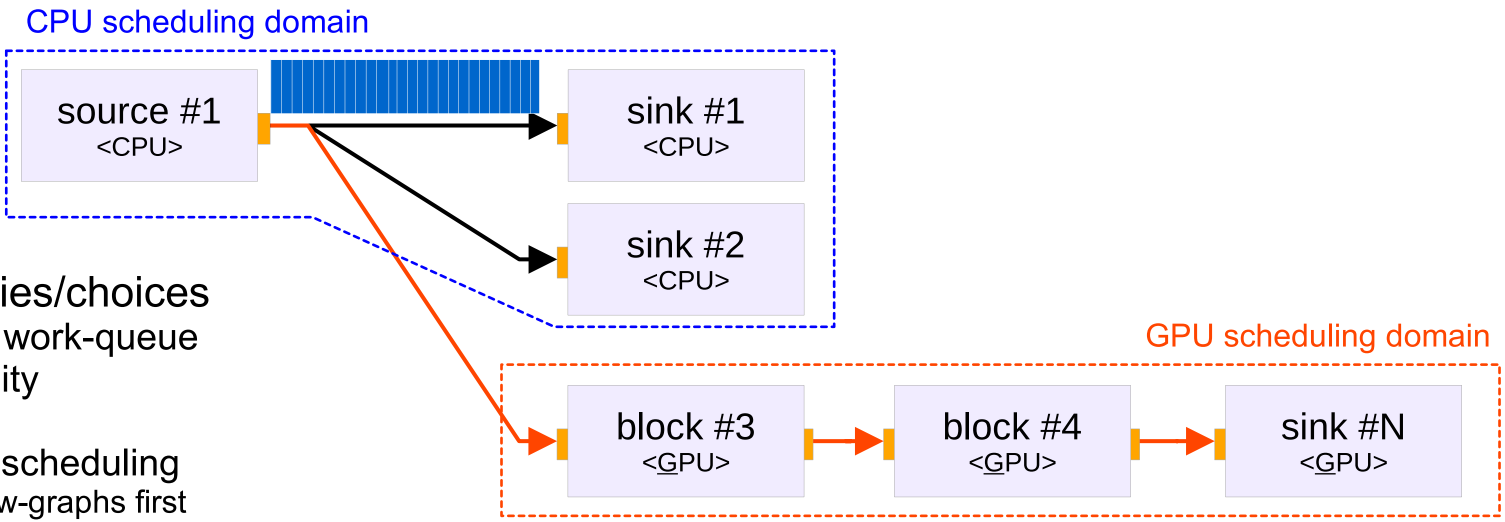
- a `scheduler`' is a process that assigns a task i.e. `block::work()′ function to be executed an available computing resources (CPU|GPU|...).
  - A) `work()′ encapsulates impl. specific `work(wio′) function (wio ↔ ports, connection, buffers, …)
  - B) only non-blocking work functions, and
  - C) only 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 ↔ CPU shielding/affinity
- high-level scheduler implementation specific design choices:
  'single global queue' vs. 'per-core queues & work stealing`



Scheduler – Proposal

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

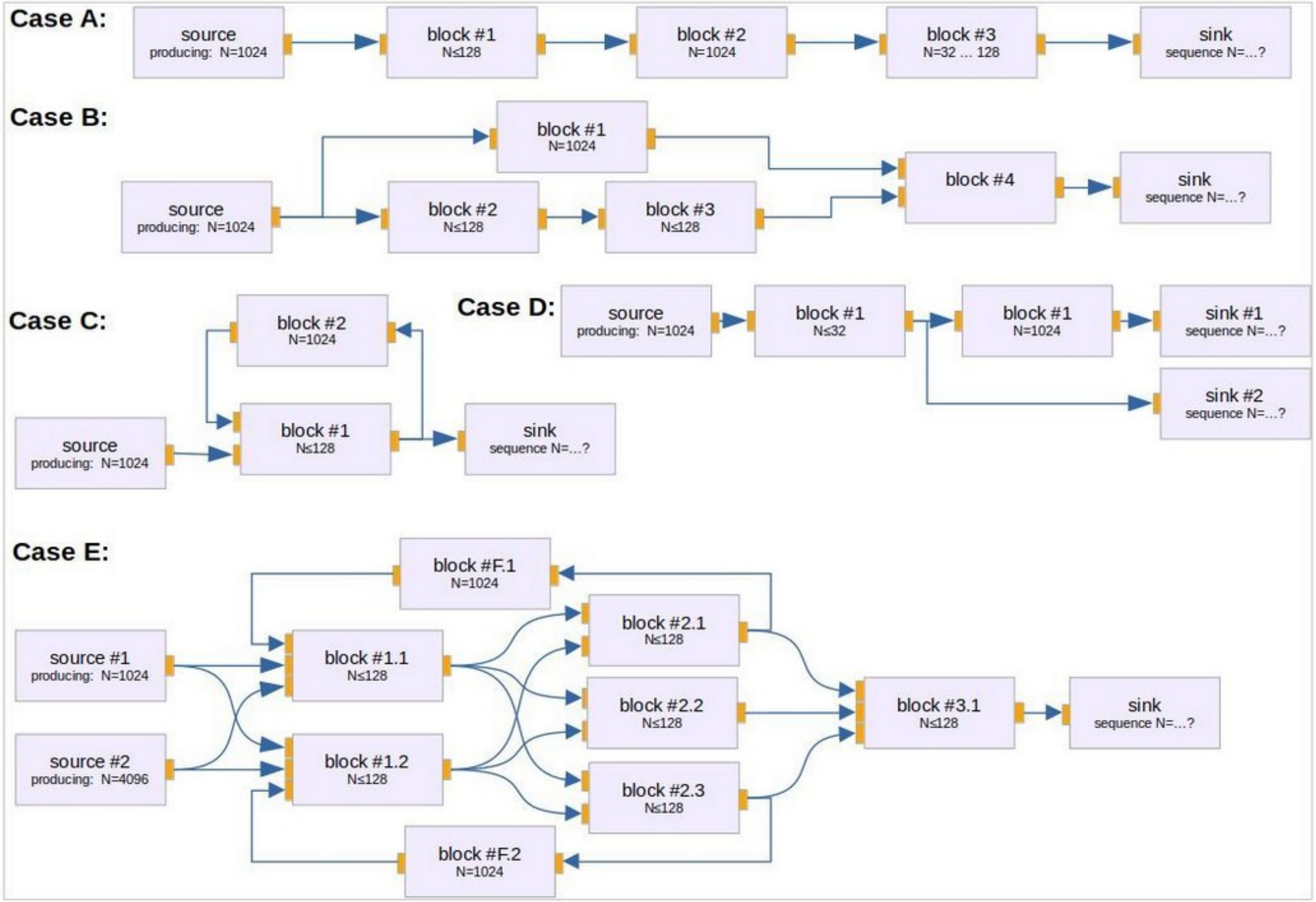
- need to be mindful that we need multiple distinct scheduler for, e.g.
  - CPU: default, fair, real-time, O(1), … (e.g. prefer small data chunks ↔ L1/L2/L3 cache & SIMD performance)
  - GPU: … (e.g. large chunks crossing CPU-GPU boundary, small for parallelising in-GPU processing ↔ >500 cores)
- scheduling 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
  - s

Some Topologies specific designed to trip-up scheduler



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

FAIR GSI

Scheduler – Proposal

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.
  - `scheduler(std::shared_ptr<graph>, Args…)` – replacing: `initialize,make...`
  - `void start() | stop() | wait() | kill()` → as before
  - `template<taskName, executeOnce, priority, cpuID, std::invocable Callable, typename... Args, …>`
    `std::future<R>| void execute(Callable &&func, Args &&...funcArgs)`

- primary task: invoke ‘`block::work()`’ (`Callable` contract)
  - encapsulates further business logic ‘`work(wio)`’ (wio ↔ port, edge, buffer, …)
  - `executeOnce:`
    true → call task once, or false (`graph`)→ recirculate `task` back into the queue once it finished

- secondary task: check if ‘`block::work()`’ can/should be executed – choices:
  A) either: very slim per-`block` interface similar to `` `[input, output]_blkd_cb_ready` ``
  B) or: more explicit interface containing blocks info on port, graph-edges, buffer min/max, …
     - *IMO (rstein): should follow-up on this path to open-up and allow for more advanced/sophisticated schedulers*

(2) #---- explicitly define & document separation-of-concern design of flow-graph:{ sub-graph:{ block:{ port:{ buffer: ...}}}}} hierarchy

node

`struct block {…};`

- domain definition:
  - CPU|GPU|DSP|FPGA|...
- collection of `port<T, IN|OUT|…, CPU|GPU, …>`
  - user-defined
  - block instantiates ports
- port/scheduler preference
  - MIN_SAMPLE/port → user API
  - MAX_SAMPLE/port → user API
  - PRIORITY → user API
  - exec-metrics → fair scheduling
- callback() – lib-level function
  - default behaviour
  - user extensions/modifiers
    - history, locks, …
  - calling: ...
- … user-defined work(…)
  - *<… user-code here … >*

`template<T, IN|OUT, … >`
`struct port {…};`

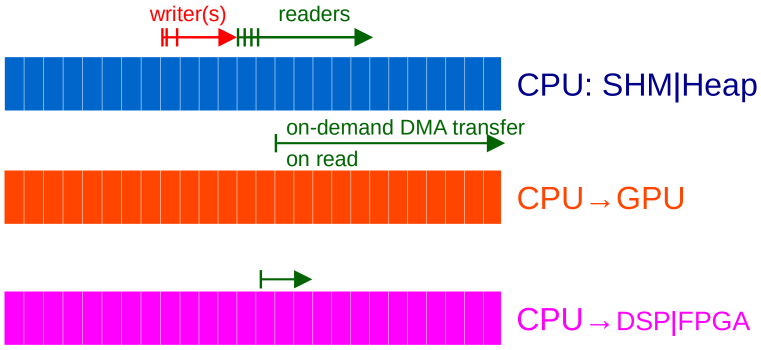
- definitions:
  - port name: i.e. “in0”, “out”, …
  - buffer item type: <T>
  - direction: IN|OUT|BOTH|MSG|...
  - domain: CPU|GPU|DSP|FPGA|...
- holds actual graph edge
  - creates buffers as needed ↔ implementation based on IN|OUT & domain constraints
- single/collection(??) of `Buffer`
  - *N.B dev-users may provide specialisation*
  - created ad-hoc/as-needed based on sub-flow-graph (dis-)connections
  - e.g. size := Nwriter * Nreader / Ncores * 2 * max({MIN_S-, {{block}::MAX_S...})

edge

- graph/connection topology (only)

`template<class T>`
`concept Buffer {…};`

- interfaces:
  - Constructor, size()
  - typed BufferWriter, BufferReader



`block` – Proposal – specific API/contract

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- `block` has [a collection of] ports – ☑ good-as-is
- a node with a `block::work()` method and other properties/methods to aid in work – ☑ good-as-is
- `block::work()` can be called outside of a GR Scheduler context – ☑ good-as-is
  - e.g. instantiate a block, call work() with appropriate buffer parameters
  - In python with some wrapping, I should be able to call `‘myblock.work([np arrays],[np arrays])’`
    - This is why work has work_io structs passed in rather than directly operating on the internally stored ports
- Parameters - PMT objects that hold values that can be instantiated via constructor or dynamically changed – ☑ good-as-is
- Constructor - Prefer this to remain a block_args struct so constructor signature doesn't change when constructor args are added or removed – ☑ good-as-is
- N.B. existing interfaces:
  - gr::node: `9 mandatory/inherited fields – 22 methods/constructors`
  - gr::block: `14 mandatory/inherited fields – 42 methods/constructors`

block – Proposal – specific API/contract

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- `template<node_name, typename ...PortsAndTypes>`
  `class XYZ : public gr::node<…> → block_base {`
- `auto name()                    →  std::string_view`
- `template<Port T>`
  `void addPort(T&& port)         → adding port during runtime`

N.B. Parameter proposal: ~36 methods, could be reduced to:

- `block(map<std::string, pmtf::pmt>)` → reusing John & Josh's PMT library (i.e. don't do work twice)
- `auto getParameter(string pName = "")      → map<string, pmtf::pmt>`
- `auto setParameter(map<string, pmtf::pmt>)  → <diagnostic return tbd.>`
- `N.B. +few others but are primarily implementation specific`
- `N.B. string→value mapping largely/fully constexpr (i.e. little/no runtime costs)`

`possibly free-standing: both as typed and un-typed (RTTI ↔ Python)`

- `auto getPortDefinitions(src) → vector<<Direction, PortName, TypeName>>`
- `auto getPort(src, portName)   →  std::shared_ptr<port_base>`
- `bool connect(src, src_port, dst, dst_port)`

FAIR GSI

block – Proposal – C++ API flavour

proof-of-concept: https://compiler-explorer.com/z/xnxMTxs3j

```
template<typename T, typename U = T, … >
requires (gr::util::is_one_of<T, supported_type>::value)
class myCopyBlock : public gr::node<"copy", gr::IN<"IN", T>, gr::OUT<"OUT", T>, int32_t, float, std::complex<float>> {
    // custom data members

public:
    myCopyBlock(std::map<std::string, int> args = {{"answer"s, 42}, {"catch"s, 22}}) { /* [..] */ }
    static auto make(std::map<std::string, int> args) { return std::make_shared<myCopyBlock>(args); }

    constexpr bool work() { // generic -- called by scheduler
        constexpr gr::Port auto in = this->template inputPort<"IN">();
        //gr::Port auto in_err = outputPort<"IN">(); // correctly fails to compile
        constexpr gr::Port auto out = this->template outputPort<"OUT">();

        static_assert(in.name() == "IN", "requested input port does not match name");
        static_assert(out.name() == "OUT", "requested output port does not match name");

        // assemble the wio ....  here: simple mock-only
        // CALL user-level work(wio)...
        return work(in.getReader(), out.getWriter());
    }

    constexpr bool work(/*BufferReader*/ auto input, /*BufferWriter*/ auto output) { // top-level user-specific code
        // ...
        return true; // return status
    }

    // [..]

    void registerPythonBindings() const noexcept { this->template initPythonBindings<decltype(this)>(); }
};
```

SupportedTypes<Ts...>
by this block

port<T> – Proposal

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- `port<T>’ has a collection of `edge<T>’s *[added by rstein]*
- A typed <T> representation of the incoming or outgoing data to/from a block – ☑ good-as-is
  - Type: STREAM or MESSAGE (these are 2 distinct things as stream triggers work() and message triggers other callback method) – ☑ good-as-is
    - *N.B: comment rstein: not a critical implementations-wise (synch- vs. async) MESSAGE could be identical to STREAM w/o required minimum data (i.e. $N_{min} \geq 1$)*
  - Name: String – ☑ good-as-is
    - auto name() → std::string_view
  - Index: TBD - would be nice to still be able to index ports by integer → use-case?
    - functional use-case and/or implementation driven?
  - Buffer: Return a reference to the buffer reader or writer associated with the port – ☑
  - Connect Method: Indicate the
- proposal: add information that is relevant for scheduling and creating buffers here

FAIR GSI

port<T> – Proposal – specific API/contract

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- `template<name, T, port_direction_t, port_domain_t, minSize, <sched_info>, …> port() = default;`
- `auto name()` → `const std::string_view`
- `auto type()` → `T in supported types (std::variant<...>)`
- `auto port_type()` → `port_type_t (STREAM, MESSAGE)`
- `auto port_direction()` → `port_direction_t (INPUT, OUTPUT, BIDIRECTIONAL)`
- `bool optional()`
- `auto available()` ↔ maps to `Buffer<T>::Buffer[Reader, Writer].available()`
- `auto port_domain()` → `port_domain_t (CPU, GPU, NET, FPGA, …)`
- `auto edges()` → `collection<edge<T>|edge_base>`
- `auto remove_edge(edge<T>|edge_base)` → `edge<T>`
- `auto add_edge(edge<T>)` → `<tbd.>`
- *… additional mandatory methods (!?) …*
- NEW `<sched_info>`:
  - `size_t  min_buffer_size()`
  - `size_t  max_buffer_size()` → `user API`
  - `size_t  priority()` → `user API` → `real-time scheduling`
  - `auto    exec_metrics()` → `<domain object tbd.>` → `fair scheduling`
  - *… additional scheduling constraints (!?) …*

port<T> – Proposal – specific API/contract – contd.

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- either:
  - bool connected()
  - auto connect(BufferFactory<T> f = DefaultBufferFactory()) → std::shared<Buffer<T>> – initialises buffers
  - auto disconnect() → std::shared<Buffer<T>> – shuts-down buffer
- or:
  - bool active()
  - auto activate(BufferFactory<T> f = DefaultBufferFactory()) → std::shared<Buffer<T>> – initialises buffers
  - auto deactivate() → std::shared<Buffer<T>> – shuts-down buffer
- auto get_reader() → Buffer<T>::BufferReader → PR#6348
- auto get_writer() → Buffer<T>::BufferWriter → PR#6348

edge<T> – Proposal – specific API/contract

https://gist.github.com/mormj/9d0b14d6db59ee7f313755c76498cc91

- template<T, weight>
  edge(port<T> src_port, port<T> dst_port) = default;
- auto name() → const std::string_view (was/is identifier())
- auto type() → T in supported types (std::variant<...>)
- auto weight() → float (edge in weighted graph) → real-time scheduling
- either:
  - bool connected()
  - auto connect() → std::shared<Buffer<T>> – initialises buffers
  - auto disconnect() → std::shared<Buffer<T>> – shuts-down BufferReader
- or:
  - bool active()
  - auto activate() → std::shared<Buffer<T>> – initialises buffers
  - auto deactivate() → std::shared<Buffer<T>> – shuts-down BufferReader

Appendix

YAML based block design workflow – some thoughts

- learning/documenting a custom IDL can make life of novice/new users harder
  → N.B. recent experience with new (experienced) developers new to GR
  - not against code-gen per se if this helps and lower the learning curve for new users
  - at the same time: coupled to a necessity to be usable by GRC unnecessarily complicates other parts and unnecessarily hard for exp. Python/C++ developers

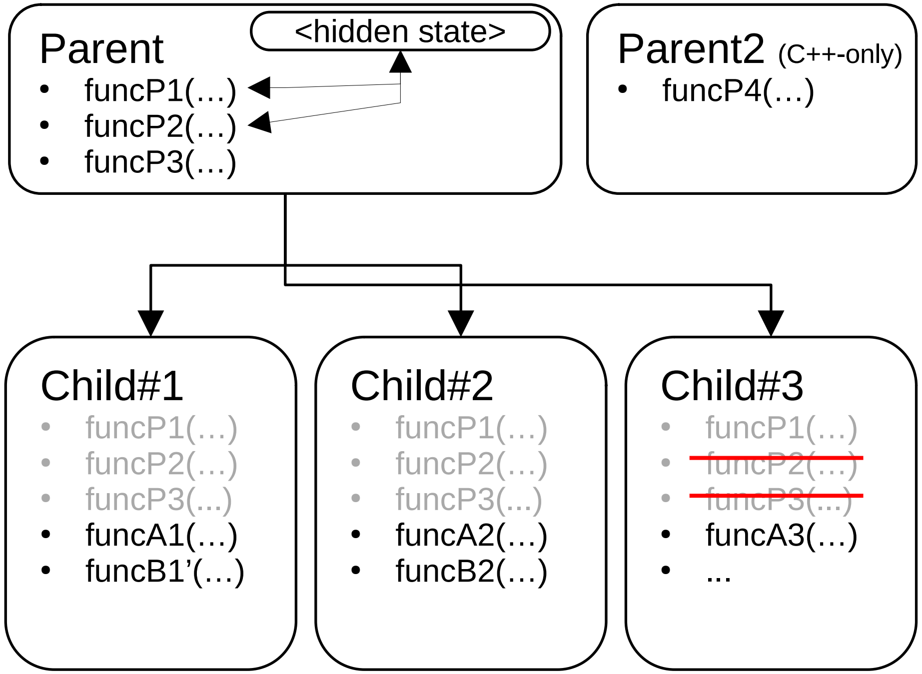
- Maybe GR 4.0 could allow/follow both paths:
  A) yml-based block generation → user modifies templated work(wio) → loaded by GRC
  B) native C++/Python blocks (N.B. port signature ↔ numpy arrays etc.)
  → `` `gr::generateDescription(gr::YML)′ `` function to generate GRC defs @ runtime
    - N.B. we have a global registry of all at run-time available blocks
    - this is also self-consistency check in case user modified the generated function (btw. a common source of errors for IDL-based serialisers)

FAIR GSI

Composition over Inheritance

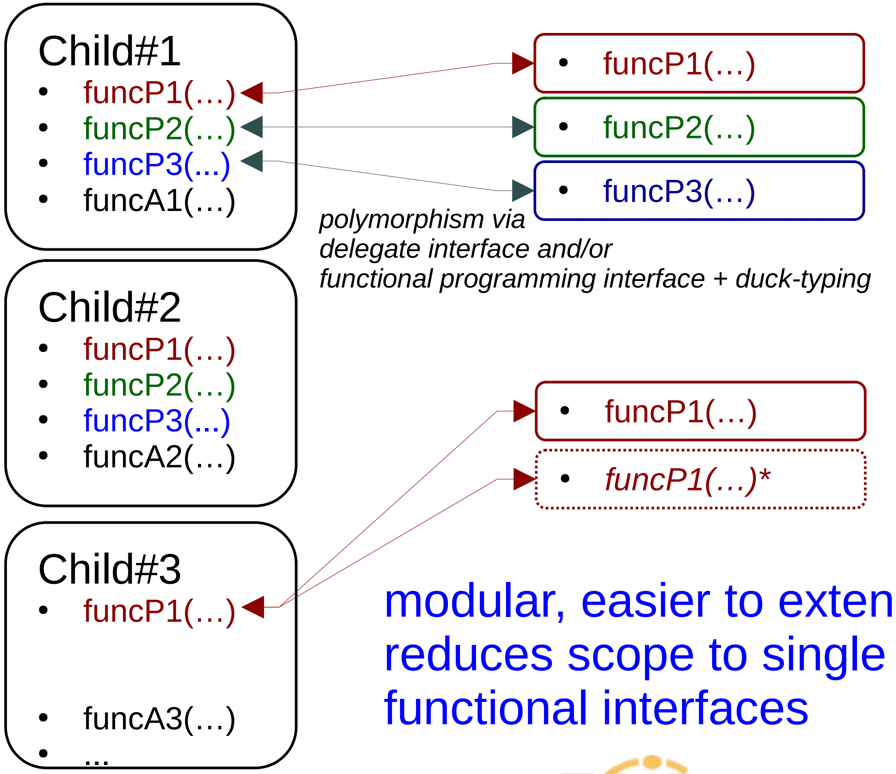
https://en.wikipedia.org/wiki/Composition_over_inheritance
https://en.wikipedia.org/wiki/Duck_typing

- Inheritance



hard to extend, to refactor, and to maintain
also: <hidden state> → issue w.r.t. HPC and thread-safety

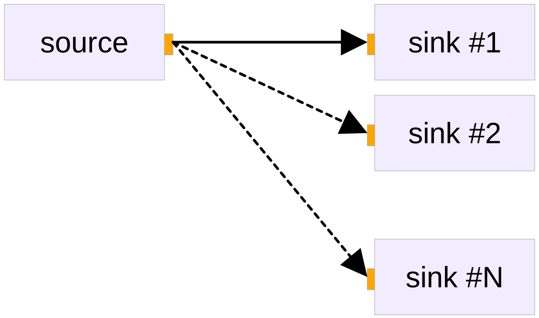
- Composition



modular, easier to extend, reduces scope to single functional interfaces

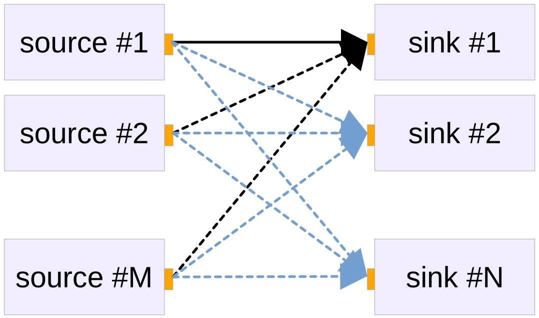
New Buffer API Proposal – Possible Use-Cases

Fan-Out:



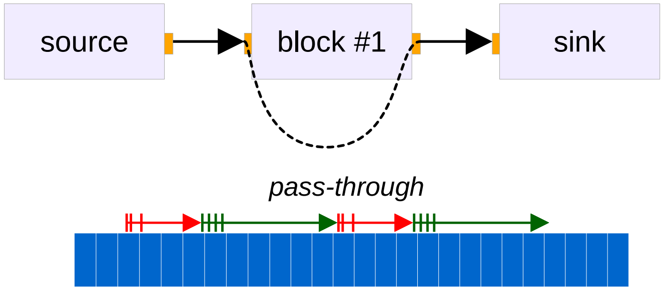
- multiple observer
- classic GR flow-graph use

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

Software must be adaptable to frequent changes

Meta-View on Software Design and GNU Radio

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