The GNU Radio scheduler and internals

Making samples flow

Marcus Müller

17 June 2019

Structure

Marcus Müller

Bearer of a couple of roles

- Research assistant at  / 

  - Exercise classes for KIT EEs'
    Probability Theory and *Communications Theory* courses ($>$ 300 students) and
    Applied Information Theory, *Advanced Radio Communications II* (ca 13 dB fewer students)
- Freelancing Engineer
  - Technical Consulting
  - Contract Development
  - Customer-Specific Training Courses
- Ettus Support Grumpiness supplier
- Maintainer of the GNU Radio project

Marcus Müller

Bearer of a couple of roles

- Research assistant at  / 

  - Exercise classes for KIT EEs'
    Probability Theory and *Communications Theory* courses ($>$ 300 students) and
    Applied Information Theory, *Advanced Radio Communications II* (ca 13 dB fewer students)
- Freelancing Engineer
  - Technical Consulting
  - Contract Development
  - Customer-Specific Training Courses
- Ettus Support Grumpiness supplier
- **Maintainer of the GNU Radio project**

Marcus Müller

Contact

Depending on what you want to talk to me about, contact me using

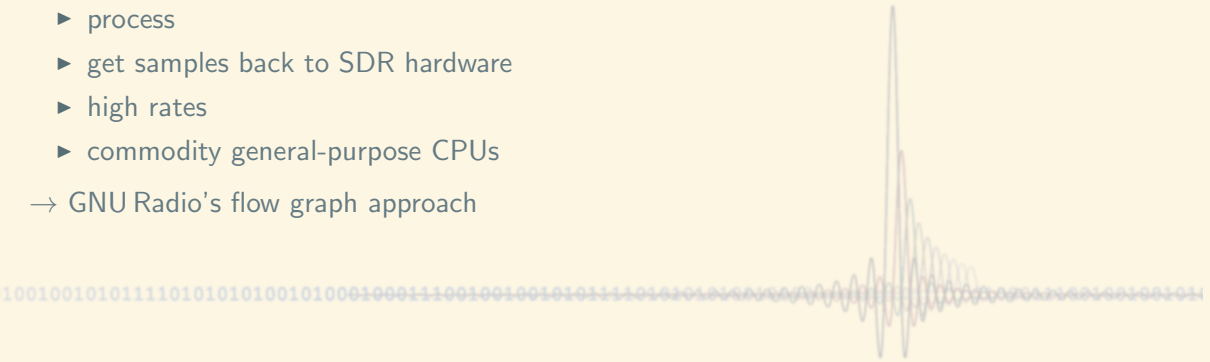
- University Research & Teaching: `mueller@kit.edu`
- GNU Radio aspects: Preferably, `discuss-gnuradio@gnu.org`, for confident matters `mmueller@gnuradio.org`
- Ettus support: `support@ettus.com` (ask for Marcus The Younger)
- Freelancing & Private: `mueller@hostalia.de`

The SDR Problem

. . . as interpreted by GNU Radio

- get samples from SDR hardware
- process
- get samples back to SDR hardware
- high rates
- commodity general-purpose CPUs

→ GNU Radio's flow graph approach

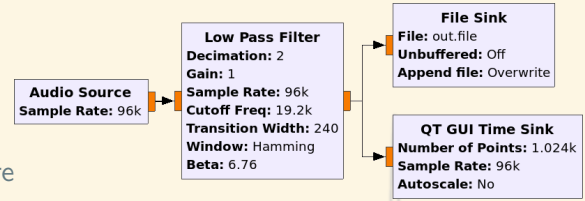


The SDR Problem

. . . as interpreted by GNU Radio

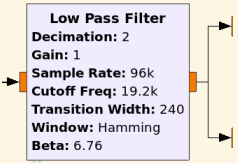
- get samples from SDR hardware
- process
- get samples back to SDR hardware
- high rates
- commodity general-purpose CPUs

→ GNU Radio's flow graph approach



GNU Radio's Signal Flow Graphs

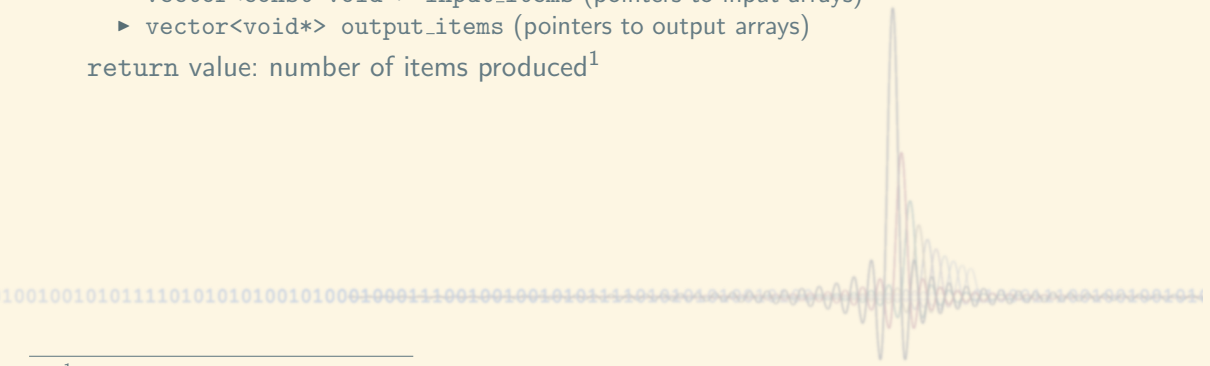
- Blocks have in- and outputs
- an output can be connected to many inputs
- an input can only be connected to one output
- developers concentrate on writing these blocks
- GNU Radio takes care of the data marshalling
  - writing a block: just enter your code into a `work` function



What is a Block?

- Instance of subclass of `gr::block`
- most importantly: `general_work` method
  arguments:
  - `int noutput_items` ($\min_k$(available space on output k))
  - `vector<int> ninput_items` (available items per input)
  - `vector<const void*> input_items` (pointers to input arrays)
  - `vector<void*> output_items` (pointers to output arrays)

  `return` value: number of items produced[1]



[1] or declare yourself using `produce(k, how_many)` and return `WORK_CALLED_PRODUCE`

What is a Block?

- Instance of subclass of `gr::block`
- most importantly: `general_work` method
  arguments:
  - `int noutput_items` ($\min_k$(available space on output k))
  - `vector<int> ninput_items` (available items per input)
  - `vector<const void*> input_items` (pointers to input arrays)
  - `vector<void*> output_items` (pointers to output arrays)

  `return` value: number of items produced[1]
- `forecast`: "If asked to produce N output items, how much items per input will you need?"
- not forced to consume all input
  - can `consume` less at single-item granularity
  - "leftovers" beginning of next call's `input_items`
- `gr::sync_block`, `decimator`, `interpolator`: Convenience classes that simplify `general_work` to `work`, implement `forecast` (in- to output relation known)

[1] or declare yourself using `produce(k, how_many)` and return `WORK_CALLED_PRODUCE`

From Blocks to Graphs

Instantiate blocks, connect them:

- GNU Radio has graph-encapsulating blocks (`hier_block`, `top_block`)
- these have `connect(src, output_port, sink, input_port)`
- builds a directed graph internally

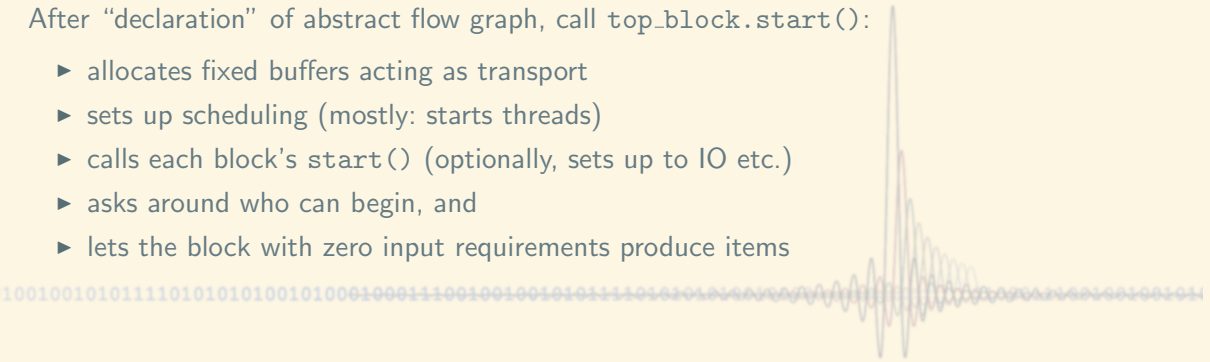
From Blocks to Graphs

Instantiate blocks, connect them:

- GNU Radio has graph-encapsulating blocks (`hier_block`, `top_block`)
- these have `connect(src, output_port, sink, input_port)`
- builds a directed graph internally

After "declaration" of abstract flow graph, call `top_block.start()`:

- allocates fixed buffers acting as transport
- sets up scheduling (mostly: starts threads)
- calls each block's `start()` (optionally, sets up to IO etc.)
- asks around who can begin, and
- lets the block with zero input requirements produce items



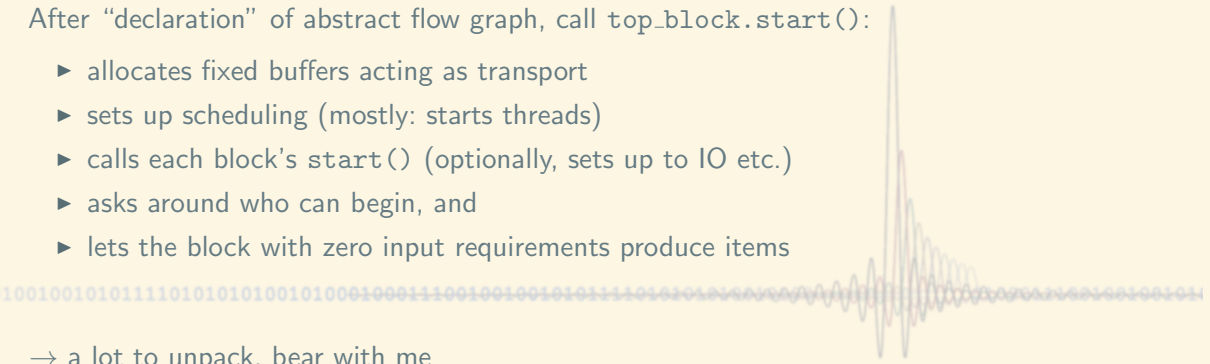
From Blocks to Graphs

Instantiate blocks, connect them:

- GNU Radio has graph-encapsulating blocks (`hier_block`, `top_block`)
- these have `connect(src, output_port, sink, input_port)`
- builds a directed graph internally

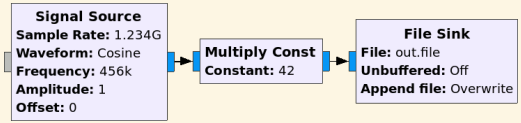
After "declaration" of abstract flow graph, call `top_block.start()`:

- allocates fixed buffers acting as transport
- sets up scheduling (mostly: starts threads)
- calls each block's `start()` (optionally, sets up to IO etc.)
- asks around who can begin, and
- lets the block with zero input requirements produce items



$\rightarrow$ a lot to unpack, bear with me

Signal Flow Architecture

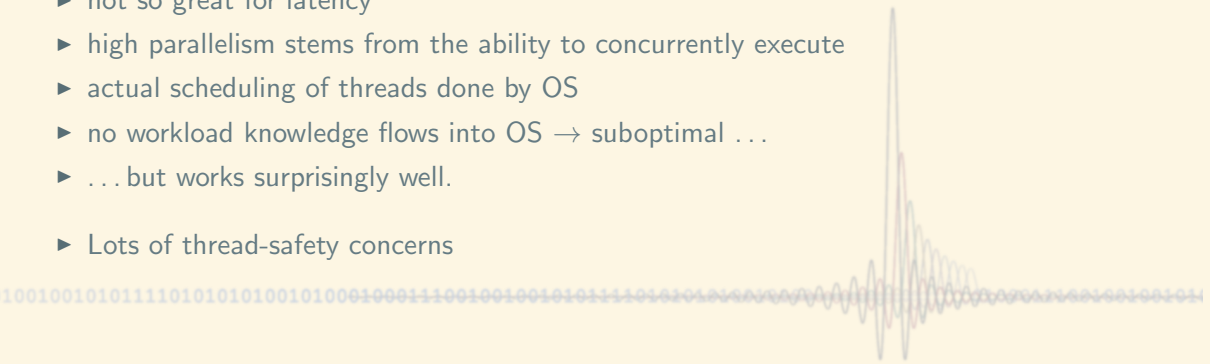


- GNU Radio is a backpressure-driven parallel signal processing architecture
- Blocks produce as much output as they can at once, given
  - available input data ready at the start of processing
  - available output data memory
- Every block runs in its own thread
- asked to produce min(buffer size / 2, available output buffer)
- Block can start working again while downstream block is still consuming
- $\rightarrow$ high parallelism

Scheduling Mechanism – Abstracted

- *Scheduler* might be too strong a word
- *back pressure* limits processing speed
- great for throughput
- not so great for latency
- high parallelism stems from the ability to concurrently execute
- actual scheduling of threads done by OS
- no workload knowledge flows into OS → suboptimal . . .
- . . . but works surprisingly well.

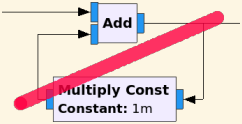
- Lots of thread-safety concerns



Message Passing

Extension to stream-based architecture

Shortcomings of the stream-only architecture



- for causality (and historical) reasons: No loops
- What if we need to sporadically update something (e.g. the equalizer taps from a downstream channel estimator)?

Solution: Asynchronous messaging

- Message ports get their own connections
- Insert *handle messages* step in block execution loop
- Message handler are executed when `work` is **not**
  - No risk of changing block state while doing DSP
- Should probably replace 90% of block setters

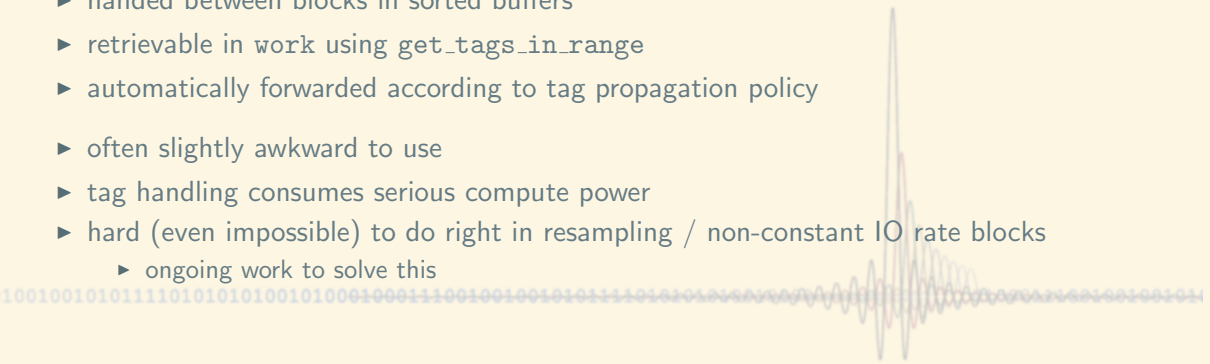
Stream Tags

Extension to stream-based architecture

Often: Need to know *when* in the stream something happens (frame detectors, . . .)

- *Tag* the appropriate item
- Tags are just tuples of (position, key, value)
- handed between blocks in sorted buffers
- retrievable in `work` using `get_tags_in_range`
- automatically forwarded according to tag propagation policy

- often slightly awkward to use
- tag handling consumes serious compute power
- hard (even impossible) to do right in resampling / non-constant IO rate blocks
  - ongoing work to solve this



Scheduling Mechanism in detail

Thread-per-Block (TPB)S cheduler (formerly: single-threaded scheduler (STS), now defunct)

- Each block gets its own executing thread[2]

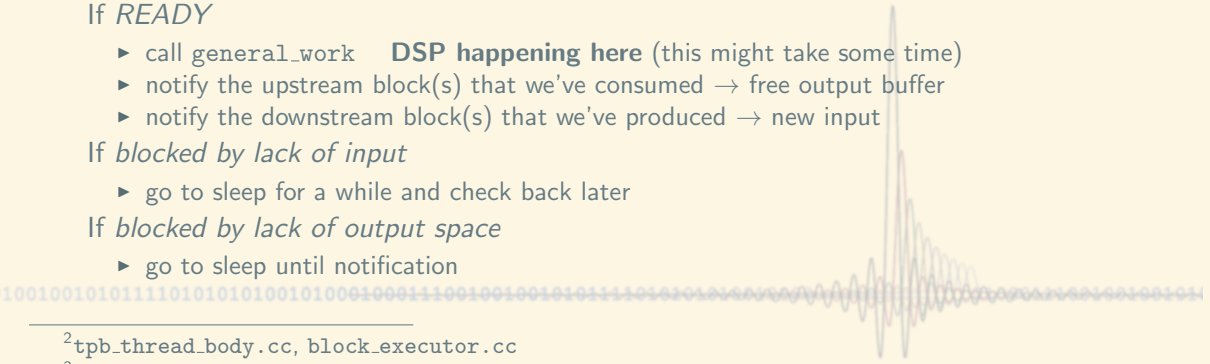
When notified[3],

- ask the block (`forecast`) whether it can produce output, given the available input and output space.

  If *READY*
  - call `general_work` **DSP happening here** (this might take some time)
  - notify the upstream block(s) that we've consumed → free output buffer
  - notify the downstream block(s) that we've produced → new input

  If *blocked by lack of input*
  - go to sleep for a while and check back later

  If *blocked by lack of output space*
  - go to sleep until notification



[2] `tpb_thread_body.cc`, `block_executor.cc`

[3] ignoring asynchronous message passing

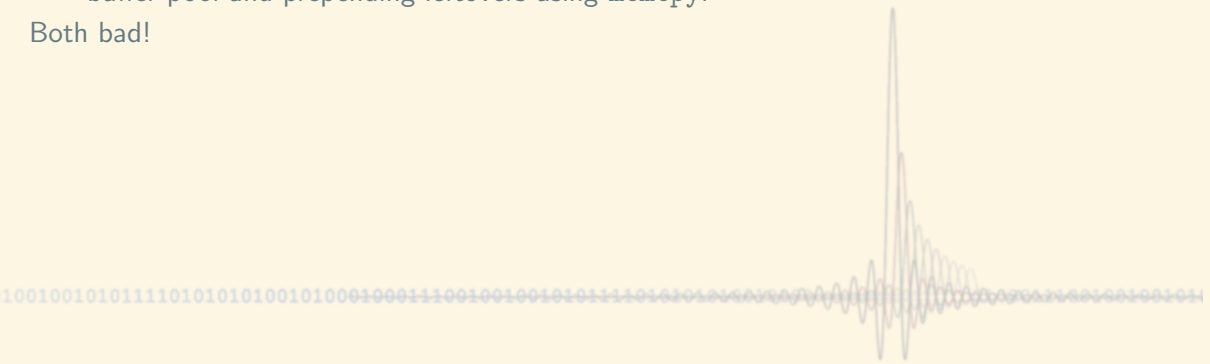
Fixed Buffers

- Freedom to not consume all input, and
- Guarantee that memory is contiguous

implies either

- infinitely large buffers or
- buffer pool and prepending leftovers using `memcpy`.

Both bad!



Fixed Buffers

One ring buffer to bind them. . .

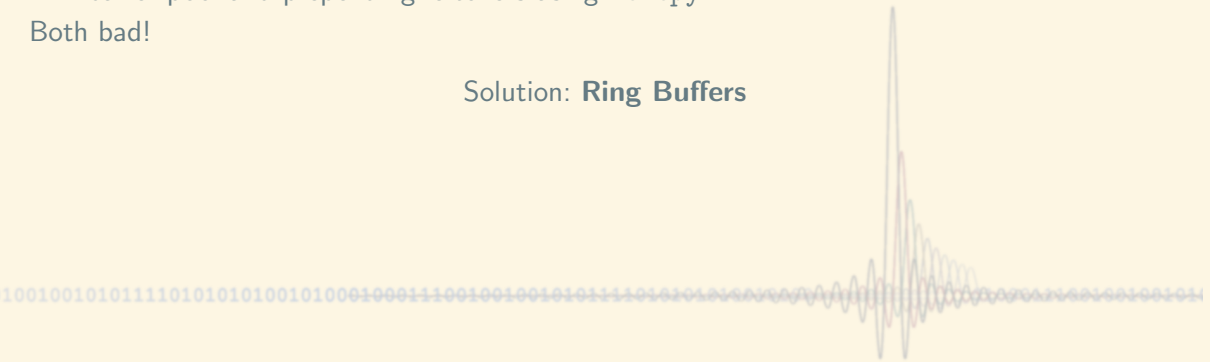
- Freedom to not consume all input, and
- Guarantee that memory is contiguous

implies either

- infinitely large buffers or
- buffer pool and prepending leftovers using `memcpy`.

Both bad!

Solution: **Ring Buffers**



Fixed Buffers

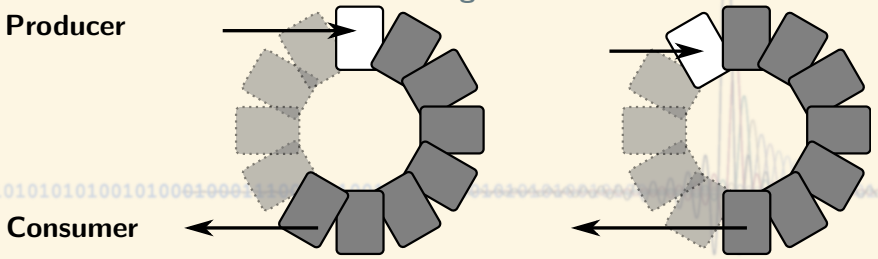
- Freedom to not consume all input, and
- Guarantee that memory is contiguous

implies either

- infinitely large buffers or
- buffer pool and prepending leftovers using `memcpy`.

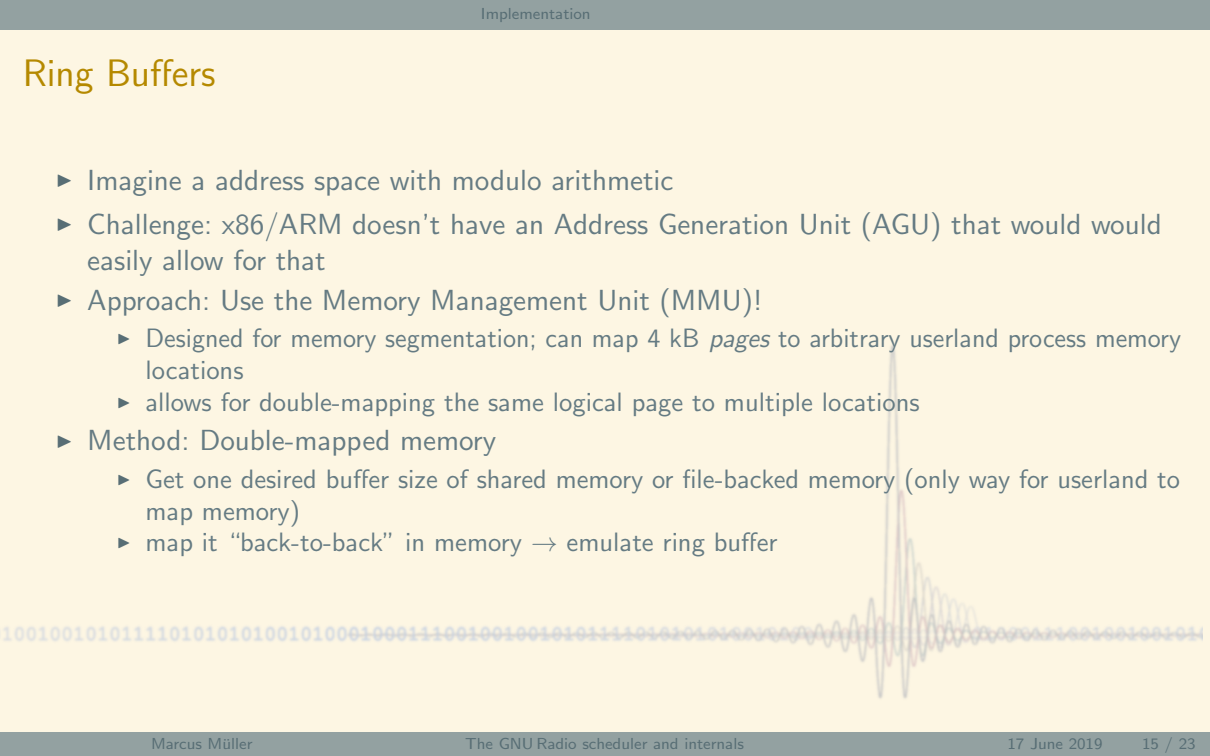
Both bad!

Solution: **Ring Buffers**



Ring Buffers

- Imagine a address space with modulo arithmetic
- Challenge: x86/ARM doesn't have an Address Generation Unit (AGU) that would would easily allow for that
- Approach: Use the Memory Management Unit (MMU)!
  - Designed for memory segmentation; can map 4 kB *pages* to arbitrary userland process memory locations
  - allows for double-mapping the same logical page to multiple locations
- Method: Double-mapped memory
  - Get one desired buffer size of shared memory or file-backed memory (only way for userland to map memory)
  - map it "back-to-back" in memory $\rightarrow$ emulate ring buffer



Ring Buffers

- Imagine a address space with modulo arithmetic
- Challenge: x86/ARM doesn't have an Address Generation Unit (AGU) that would would easily allow for that
- Approach: Use the Memory Management Unit (MMU)!
  - Designed for memory segmentation; can map 4 kB *pages* to arbitrary userland process memory locations
  - allows for double-mapping the same logical page to multiple locations
- Method: Double-mapped memory
  - Get one desired buffer size of shared memory or file-backed memory (only way for userland to map memory)
  - map it "back-to-back" in memory → emulate ring buffer

Ring Buffers

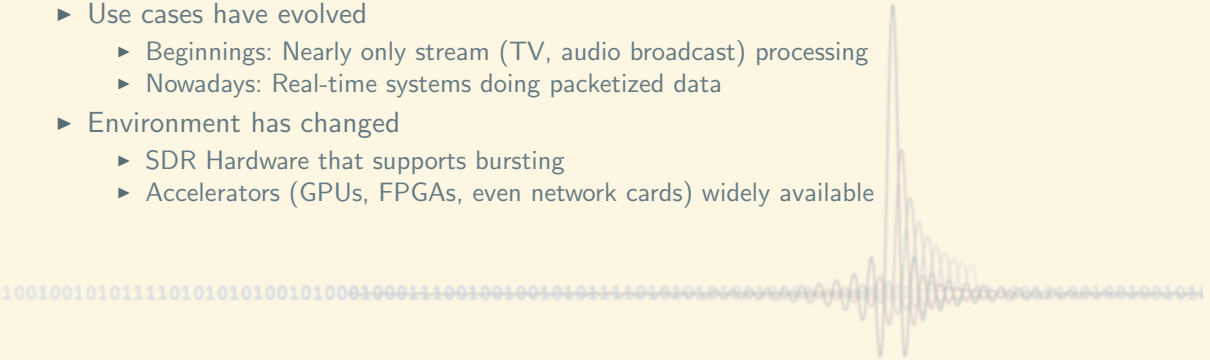
- Imagine a address space with modulo arithmetic
- Challenge: x86/ARM doesn't have an Address Generation Unit (AGU) that would would easily allow for that
- Approach: Use the Memory Management Unit (MMU)!
  - Designed for memory segmentation; can map 4 kB *pages* to arbitrary userland process memory locations
  - allows for double-mapping the same logical page to multiple locations
- Method: Double-mapped memory
  - Get one desired buffer size of shared memory or file-backed memory (only way for userland to map memory)
  - map it "back-to-back" in memory → emulate ring buffer

Short Intra-Talk Q& A

Any questions so far?

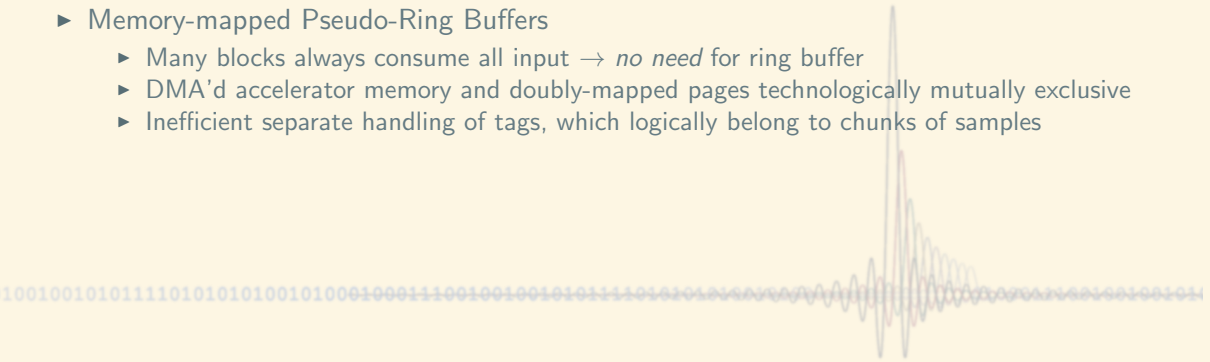
Problems and Challenges

- GNU Radio has ca. 18 years of history
- Not all decisions made in that period apply to the current architecture
  - to be completely honest, not even all decisions were good
- Use cases have evolved
  - Beginnings: Nearly only stream (TV, audio broadcast) processing
  - Nowadays: Real-time systems doing packetized data
- Environment has changed
  - SDR Hardware that supports bursting
  - Accelerators (GPUs, FPGAs, even network cards) widely available



Challenges for Scheduling

- Every block gets own thread getting scheduled on randomly (OS-)chosen CPU core
  - **clearly** suboptimal
- Memory-mapped Pseudo-Ring Buffers
  - Many blocks always consume all input → *no need* for ring buffer
  - DMA'd accelerator memory and doubly-mapped pages technologically mutually exclusive
  - Inefficient separate handling of tags, which logically belong to chunks of samples



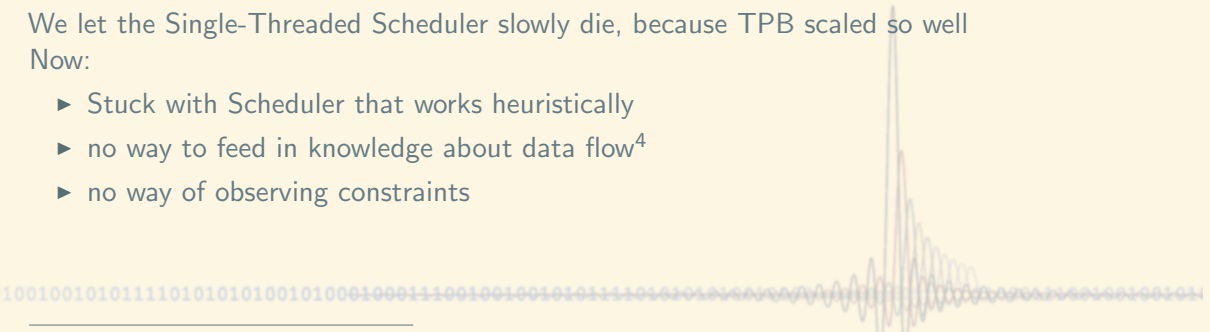
Scheduling Shortcomings

We can do better than letting OS randomly decide which blocks gets executed when on which CPU core

- Memory locality is much more important than using large chunks of data
- Developer knows more about data dependencies than OS

We let the Single-Threaded Scheduler slowly die, because TPB scaled so well
Now:

- Stuck with Scheduler that works heuristically
- no way to feed in knowledge about data flow[4]
- no way of observing constraints



[4] Aside from pinning blocks to CPU cores. See: Kirby Cartwright's *A Case Study in Optimizing GNU Radio's ATSC Flowgraphs*, GRCon 2017

Better Scheduling

Proposed Changes we came up with so far

Not happening in GNU Radio 3.8!

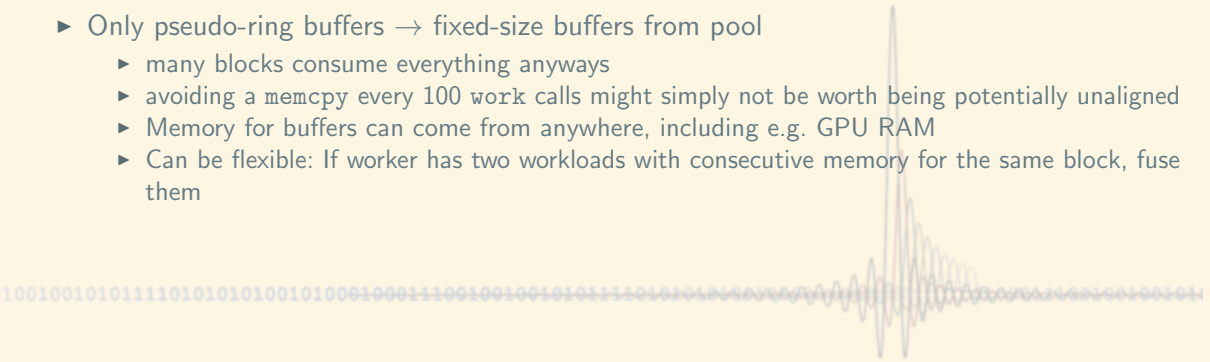
- Thread-per-Block → Worker Threads (matching CPU core count)
  - assign graph-sequential blocks to the same worker for memory locality
  - use queue to avoid passing notifications all over the place
  - *optionally* make queue smart enough to fuse workload items, reorder
- notifications → exchange workload items (block + buffer address + tag reference vector)
  - inherent binding of tags and items
  - workload items exchangeable between workers
  - packetized samples and stream just workload items
- add loads of diagnostics
  - check whether latency constraints are met
  - generate data for optimization
  - come up with (ML?) algorithms to improve performance of running flowgraph

Better Buffers

Proposed Changes we came up with so far

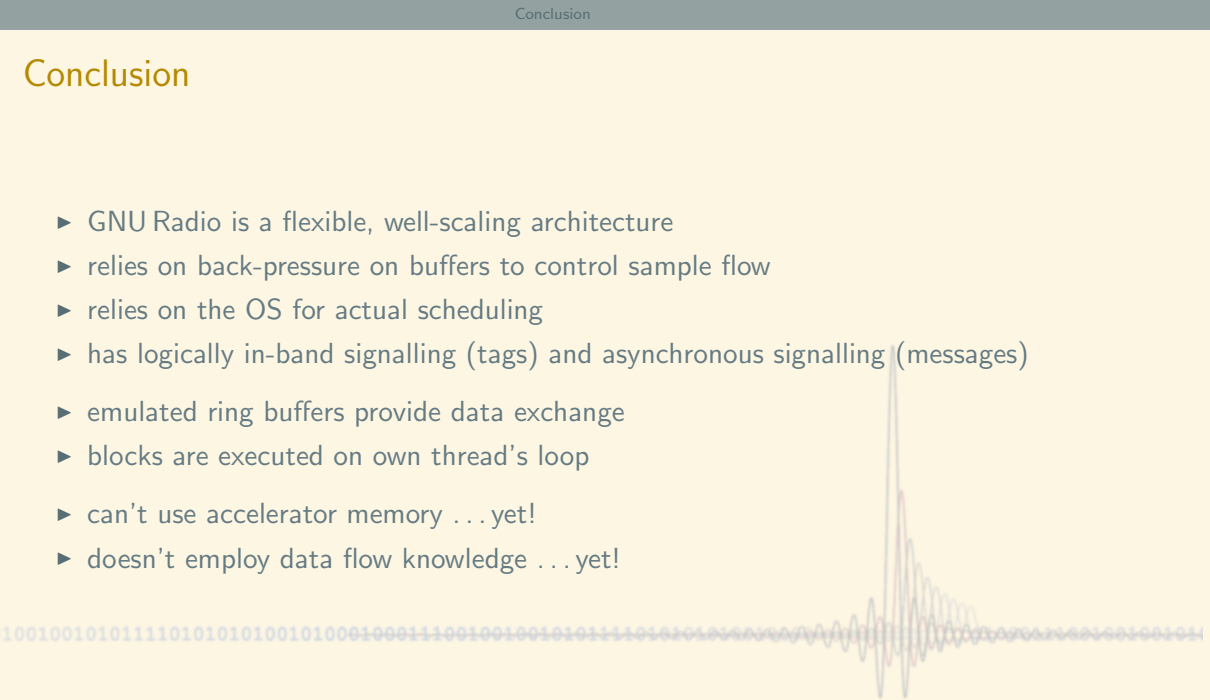
Not happening in GNU Radio 3.8!

- Only pseudo-ring buffers → fixed-size buffers from pool
  - many blocks consume everything anyways
  - avoiding a `memcpy` every 100 `work` calls might simply not be worth being potentially unaligned
  - Memory for buffers can come from anywhere, including e.g. GPU RAM
  - Can be flexible: If worker has two workloads with consecutive memory for the same block, fuse them



Conclusion

- GNU Radio is a flexible, well-scaling architecture
- relies on back-pressure on buffers to control sample flow
- relies on the OS for actual scheduling
- has logically in-band signalling (tags) and asynchronous signalling (messages)
- emulated ring buffers provide data exchange
- blocks are executed on own thread's loop
- can't use accelerator memory . . . yet!
- doesn't employ data flow knowledge . . . yet!



Questions & Answers

Ask away!