Real-Time Scheduling Support in GNU Radio 4

Tiancheng He[1], Joseph Goh[2], Syed W. Ali[2], Joel Isaacson[1], Nicholas Carter[2], Samarjit Chakraborty[2], James H. Anderson[2], and Bryan C. Ward[1]

[1]Dept. of Computer Science, Vanderbilt University, Nashville, TN 37212 USA
[2]Dept. of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599 USA

[1]{TIANCHENG.HE, JOEL.J.ISAACSON, BRYAN.WARD}@VANDERBILT.EDU
[2]{JGOH, SWALI, CHAYNICK, SAMARJIT, ANDERSON}@CS.UNC.EDU

Abstract

GNU Radio 4 (GR4) introduces a modular scheduling framework in which application-defined schedulers can replace the runtime's default. This is a powerful foundation for more predictable and performant software-defined radio. However, the framework mainly encourages customization of how blocks are distributed across worker threads. All workers share an execution loop that traverses blocks in a fixed order. Consequently, when a block runs follows from its position in that order rather than from the urgency of its work, leaving room for tighter control from a real-time perspective.

This paper presents work in progress on real-time scheduling support in GR4, built atop its modular scheduler API, with the goal of allowing latency-sensitive flowgraphs to be scheduled, and ultimately analyzed, using results from real-time scheduling theory. The order in which a worker thread serves its blocks is made a configurable policy that composes with GR4's existing schedulers. Static- and dynamic-priority policies, such as rate monotonic and earliest deadline first, respectively, are implemented on this interface. A lightweight execution tracer is also introduced, allowing detailed examination of timing behavior and scheduler overheads. Evaluation on IEEE 802.11p receiver flowgraphs demonstrate how different workloads can benefit from the right scheduling policy. This paper and the accompanying presentation discuss these motivations, connect them to the GR4 scheduler design, report the current state of this development effort, and invite feedback from the GNU Radio community.

Proceedings of the 16^{th} GNU Radio Conference, Copyright 2026 by the author(s).

1. Introduction

A key improvement of GNU Radio 4 (GR4) over its predecessor, GNU Radio 3 (GR3), is its modular scheduling framework. GR4's default schedulers affix signal-processing blocks to shared worker threads, improving cache locality and reducing context-switching overhead over GR3's thread-per-block design. Developers may further select or implement schedulers that best fit their workloads by specifying how blocks are assigned to and ordered for execution in each worker thread. By allowing explicit specification of scheduling behavior, GR4 aims to afford developers greater flexibility in optimizing the computational efficiency and predictability of their software-defined radio (SDR) workloads.

However, the base scheduler class in GR4 today leaves room for tighter, more analyzable control over execution behavior that would benefit latency-sensitive SDR workloads. All schedulers that inherit from the base class share a core execution loop that walks each worker thread's runlist in a fixed sequential order. This round-robin (RR) loop does not consider the urgency of a given block's work, such as the work's target deadline or the rate at which a block should be serviced. For instance, a block outputting a low-latency audio stream should ideally be serviced frequently and soon after its input data samples become available. Currently, such a block would be routinely de-prioritized over other blocks ahead in the runlist. Furthermore, GR4 schedulers today offer no way to express the priority of a block's work. While custom schedulers may configure the ordering of blocks in a given worker's runlist, as well as how blocks are distributed across multiple worker threads, the default RR loop prevents more fine-grained control.

This paper presents our work in progress to introduce real-time scheduling support in GR4. Atop the existing scheduling framework, we have implemented a number of well studied and widely used real-time scheduling policies. Our scheduling framework allows work to be prioritized at block and sample granularity, with the exact policy and prioritization schemes being user-configurable and modular.

Our contributions are as follows.

(i) We have added support for priority-based scheduling for GR4 flowgraphs, which may be customized and combined with existing scheduler classes via the `SchedulingPolicyLike` template.

(ii) We have used the above template to implement widely used real-time scheduling algorithms, including fixed-priority (FP) algorithms like rate monotonic (RM) and dynamic-priority algorithms such as earliest deadline first (EDF).

(iii) We have created a lightweight tracing library to supplement GR4's existing profiling tools, allowing for detailed execution-timeline reconstruction and post-hoc analysis with minimal perturbation of the running workload.

(iv) We present preliminary results from our evaluation of our scheduling policies using IEEE 802.11p flowgraphs.

Organization. Sec. 2 reviews how GR4 schedules a flowgraph today, the real-time scheduling concepts we build on, and our system model. Sec. 3 describes how we have implemented our scheduling-policy classes and their supporting features. Sec. 4 compares RR, RM, and EDF on IEEE 802.11p receivers. Sec. 5 reviews related work, and Sec. 6 concludes.

2. Background

In this section, we provide necessary background on real-time scheduling and scheduling in GR4.

2.1. Flowgraph Scheduling in GR4 Today

As of the first GR4 release candidate (RC1), how a given scheduler governs flowgraph execution is specified primarily by two components: the `customInit()` method, intended to be overridden by derived scheduler classes, and `SchedulerBase::poolWorker()`, which, as currently implemented, is not to be user-overridden.

A scheduler's `customInit()` method assigns blocks in a flowgraph to the runlist of each worker thread, the number of which depend on the system and whether the execution policy is specified to be single-threaded or multi-threaded. By selecting a scheduler or implementing one with a new `customInit()` method, the ordering of blocks in each runlist and how they are distributed across threads may be customized. During execution, dispatched worker threads run the `poolWorker()` method. `poolWorker()` then enters a loop, traversing its runlist and calling the `work()` method for each block. This, in effect, invokes blocks in RR fashion, with blocks whose input buffers are insufficiently full being skipped. Fig. 1 shows both methods in simplified form. Because `poolWorker()` also performs many functions relating to managing the lifecycle of a flowgraph, it is not readily modifiable by users looking to modify the RR scheduling behavior.

`Simple::customInit()`

$n \leftarrow$ number of worker threads
for each block j, in the order added to the graph **do**
  append block j to runlist $j \bmod n$
end for

`SchedulerBase::poolWorker(runlist)`

$c \leftarrow 0$
repeat
  if $c = 0$ **then**
    handle scheduler and block messages
    adopt new blocks, clean up removed ones
    $s \leftarrow$ scheduler lifecycle state
  end if
  $c \leftarrow (c + 1) \bmod R$
  if s is RUNNING **then**
    for all blocks b in the runlist, in order **do**
      call `work(max_work_items)` on b
    end for
    if every block reported DONE **then**
      break
    end if
  else
    sleep `timeout_ms`; $c \leftarrow 0$
  end if
until s is no longer active

Figure 1. GR4's stock scheduling in RC1, simplified. `Simple` deals blocks to worker runlists in the order they were added to the graph. Each worker then runs `poolWorker()` over its own runlist. R sets the number of passes a worker makes over its runlist before performing other housekeeping functions.

2.2. Real-Time Scheduling

The concept of "real-time" processing can take on different goals and meanings in different problem domains. Here we clarify our definition of real-time scheduling, and provide relevant background from the real-time systems research community. In some fields, such as computer graphics, "real time" often refers simply to streaming, smooth, or otherwise interactive processing—for clarity, we call this "real fast" not real time. In contrast, in safety-critical domains, such as avionics and automotive applications, which motivate most work in the real-time scheduling community, "real-time" refers to the ability to have predictable worst-case performance that satisfies strict timing requirements. For example, in a drone flight-control system, a missed control actuation could destabilize the drone causing a crash. The field of real-time scheduling therefore has built both analytical scheduling theory as well as corresponding

systems implementations to optimize system performance to maximize resource efficiency while maintaining *temporal correctness*, i.e., not violating timing constraints.

To take advantage of real-time scheduling principles, we review two important concepts, *scheduling algorithms* and *schedulability analysis*. Scheduling algorithms determine, at any time instant what should be scheduled or executed at that time point. A *static-priority* algorithm gives each task a fixed priority that does not change. This is akin to prioritizing one flowgraph block over another. A *dynamic-priority* algorithm allows the priority of blocks to change over time. For example, a dynamic-priority algorithm can therefore prioritize blocks based upon the age of the inputs they are processing. Static-priority algorithms are often easier to understand and implement, while dynamic-priority algorithms can have higher overheads, but benefit from several analytical properties, described more below.

With a model of the workload to be scheduled, coupled with a given scheduling algorithm, a *schedulability analysis* is an algorithm that can determine whether all timing requirements are guaranteed to be satisfied at runtime. In the context of a software-defined radio application, such guarantees could include bounds on end-to-end latency of a flowgraph, or guarantees that the system will not drop samples or packets as the system is guaranteed to have the computational capacity to support the workload. The dynamic-priority scheduling algorithm *Earliest Deadline First* (*EDF*) algorithm, which assigns a *deadline* to each unit of work and prioritizes based upon it, on a uniprocessor system with simple task models is *optimal* in that it can maximize the potential of the available hardware resources while guaranteeing schedulability. The static-priority scheme *Rate Monotonic* (*RM*) scheduling algorithm actually emits the same schedule as EDF when the periods, or invocation frequency of the blocks, are harmonic, or multiples of each other. Otherwise, well-studied response-time analyses can be applied to determine how long tasks will take to complete when considering competing workloads, and can be used to evaluate and guarantee schedulability.

In this work, we aim to build the capability in GR4 to be able to control the priority of tasks, which in turn enables you to implement different scheduling algorithms. We demonstrate this by implementing RM and EDF scheduling in GR4. After profiling the execution times and incorporating periodicity, schedulability analyses can then be applied to the workload in GR4 to be able to provide timing guarantees in GR4.

2.3. System Model

We now describe our system model for computational tasks in GNU Radio 4 and how they map to real-time scheduling.

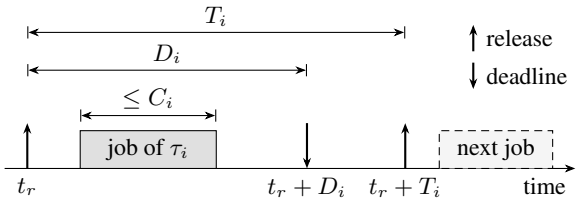

Figure 2. A job of τ_i released at t_r executes for at most C_i and must complete by its absolute deadline $t_r + D_i$. The next job of τ_i is released no earlier than $t_r + T_i$.

Each block in a GR4 flowgraph is modeled as a real-time *task*, where the i^{th} such task is denoted by τ_i. Each task invocation, i.e., call to a block's `work()` method that results in processing of data, is referred to as a *job*. We consider a job to be *released* as soon as sufficient data is available for the associated block to be executed.

Each τ_i may be described by its *period* T_i, its minimum or expected inter-arrival time of its jobs, *worst-case execution time (WCET)* C_i, and its relative *deadline* D_i, the time by which the job must complete. As illustrated in Fig. 2, a job of τ_i released at time t_r would have its absolute deadline at $t_r + D_i$, execute for up to C_i time units, and its subsequent job would be expected to release at or after $t_r + T_i$. We assume each job, once scheduled, executes to completion and may not be preempted by other jobs.

Task attributes from GR4 blocks. The real-time task attributes described above may be derived from GR4 blocks' properties such as sample arrival rates and processing-batch sizes. Unlike in GR3, where users could specify desired invocation rates (e.g., 100 Hz) for each block, in GR4, users are not expected to annotate every block with such a field. Instead, execution rates may be inferred from the rate at which samples arrive to a block's input buffer and further controlled by restricting its allowed range of processing-batch sizes, the minimum or maximum number of samples each `work()` invocation may process.

For instance, consider a block τ_i whose input buffer is expected to receive data samples at a frequency of f_i Hz. Given a fixed batch size of B_i, the expected invocation rate of the block would be f_i/B_i, and its corresponding real-time task's period would be given by $T_i = B_i/f_i$. Often, real-time tasks' relative deadlines are set equal to their periods, such that each job of a task must complete before the following job is released. Thus, one may also set $D_i = T_i$.

To allow for simple application of these derivation rules, we have added parameters to each block where real-time task parameters may be specified and are implementing a utility to derive said parameters automatically at flowgraph initialization as `SchedulingAnalysis.hpp`.

3. Design

This section describes the design of our real-time scheduling features.

3.1. Modular Scheduling-Policy Classes

A core aim of our real-time scheduling implementation was to build a modular and configurable interface, keeping in line with GR4's original modular-scheduling approach. To this end, we have implemented three classes of scheduling policies, each determining the way in which blocks are prioritized for execution at runtime in worker threads' `poolWorker()` loop. The three classes are: (a) RR, which uses the existing default worker loop, (b) *static priority*, where each block has its priority level assigned at flowgraph initialization, and (c) *dynamic priority*, where the priority level of blocks may be updated at runtime.[1] Each of the priority-based policy classes may be customized by changing the way in which priority levels are assigned to blocks or their jobs. We have implemented the RM and EDF scheduling algorithms as exemplar static-priority and dynamic-priority policies, respectively. We describe these policies in further detail later in this section.

Each concrete scheduling policy may then be specified to existing scheduler types, such that users may combine them with existing runlist-assignment strategies such as the pre-provided `Simple`, `BreadthFirst`, or `DepthFirst` schedulers. Specifically, users may pass a `SchedulingPolicyLike` object as a template argument to any scheduler as shown in Fig. 3. Depending on the class of scheduling policy chosen, the appropriate `poolWorker()` loop body is chosen at compile time, allowing RR policies to compile and behave identically to existing scheduler implementations.

```
namespace gs = gr::scheduler;
gs::Simple<
  gs::ExecutionPolicy::multiThreaded,
  gr::profiling::null::Profiler,
  gs::EdfPolicy> sched;
sched.exchange(std::move(graph));
sched.runAndWait();
```

Figure 3. Instantiating a scheduler and executing a flowgraph with the EDF policy. When no `SchedulingPolicyLike` argument is provided, `RoundRobinPolicy` is used by default.

3.2. Static-Priority Policies and RM

Under static-priority policies, each block may be annotated with a priority level prior to flowgraph execution. Because priority levels are fixed throughout the lifecycle of the flowgraph, the runlist may be pre-sorted by priority level at initialization time. The main worker-thread loop then executes a modified `SchedulerBase::traverseBlockListOnce()` shared by all static-priority policies, as detailed in Fig. 4. Unlike RR scheduling, under static-priority policies, if a higher-priority block has work available, it is always scheduled before any lower-priority blocks.

```
index ← 0; selections ← 0
while index < n_blocks and selections < bound do
  if block[index] is finished then
    index ← index + 1
  else
    (performed, status) ←
    block[index].WORK(batchCeiling[index])
    if status = ERROR then
      return ERROR
    else if status = DONE then
      mark block[index] finished; index ← index + 1
    else if performed > 0 then
      selections ← selections + 1; index ← 0   ▷ restart at the top
    else
      index ← index + 1                        ▷ starved; try next priority
    end if
  end if
end while
return
```

Figure 4. Simplified execution loop performed by `traverseBlockListOnce()` under static-priority scheduling policies. Each iteration, *block*, the pre-sorted runlist, is scanned in decreasing-priority order to look for eligible blocks to execute. After *selections* executions, the loop exits to allow other bookkeeping tasks to be performed.

A widely used static-priority policy, which we have implemented in GR4, is RM, which assigns higher priorities to tasks with higher execution rates (i.e., lower periods). While non-preemptive RM scheduling cannot optimally ensure that all jobs meet their deadlines like its preemptive counterpart, it is nevertheless well studied and simple to apply. Users may also manually specify their desired priority levels on a per-block basis using `FixedPriorityPolicy`. Instead of assigning priorities with the aim of meeting deadlines, one might assign higher priority levels to blocks with more stringent quality-of-service requirements or with higher criticality.

3.3. Dynamic-Priority Policies and EDF

Unlike static-priority policies, dynamic-priority policies require the priority level of each job to be computed at runtime, which in turn requires job releases and completions to be tracked. As described in Sec. 2.3, a job of a block is considered released whenever a sufficient number of samples

[1] Static priority and dynamic priority are more formally referred to as task-level fixed priority and job-level fixed-priority, respectively, in real-time scheduling literature.

to execute its `work()` function have arrived in its input buffers. To avoid busy probing of input buffers, job releases are detected and recorded primarily when input buffers are written to by another block, with the exception of blocks whose input samples arrive from non-GR4-block sources.

Released jobs have their deadlines and respective priority levels recorded and are placed into a priority queue, implemented as a heap. Dynamic-priority policies' `SchedulerBase::traverseBlockListOnce()` method, described in Fig. 5 then uses the queue to select the highest-priority job to execute.

```
1:  H ← ∅; selections ← 0
2:  for i ← 0 to n_blocks − 1 do
3:    if ELIGIBLE(i) then
4:      H ← H ∪ {(KEY(i), i)}                ▷ front job's deadline
5:    end if
6:  end for
7:  MAKEHEAP(H)                               ▷ min-key on top
8:  while selections < bound and H ≠ ∅ do
9:    c ← POPMIN(H)
10:   if ELIGIBLE(c) then
11:     RUNONE(c)                  ▷ work, retire, release successor
12:     if ELIGIBLE(c) then
13:       PUSH(H, (KEY(c), c))                ▷ re-key to next job
14:     end if
15:   end if                                  ▷ else: stale entry, discard
16: end while
    return
```

Figure 5. Simplified execution loop performed by `traverseBlockListOnce()` under dynamic-priority scheduling policies. Each iteration, the highest-priority job, popped from H, the priority heap, is executed. Any new job releases that result from that execution are recorded and H is updated accordingly. After *selections* executions, the loop exits to allow other bookkeeping tasks to be performed.

We have implemented EDF as our exemplar dynamic-priority policy. As the name implies, under EDF, the job with the earliest deadline is chosen to be executed at each scheduling decision. Non-preemptive EDF is known to be optimal within its class, and will thus ensure that all jobs meet their deadlines if possible.[2] However, as is the case with static-priority policies, developers may implement their own priority-assignment methods for dynamic-priority scheduling policies. For instance, a first-in-first-out (FIFO) policy may incur less runtime overhead as jobs may stored in a simple queue rather than a priority queue.

[2]Specifically, non-preemptive EDF is optimal with respect to the class of non-preemptive, work-conserving scheduling algorithms, i.e., schedulers that do not insert idleness when unfinished work is available (Jeffay et al., 1991; George et al., 1996).

3.4. Lightweight Tracing and Latency Analysis

For detailed profiling, analysis, and verification purposes, the ability to produce detailed execution traces is desired.

Existing profiler. GR4 ships with `Profiler.hpp`, which may be used to emit and record trace markers in its schedulers. When compiled in, events within the scheduler may be recorded in Chrome Trace JSON format (The Chromium Authors, 2016), allowing event timelines to be examined. However several aspects of its design make it less suitable for constructing a detailed execution timeline and for isolating small but significant and overheads.

First, the profiler is not designed to be reached by block code, preventing tracing within blocks' `work()` methods. Second, events are recorded in wall-clock time using `std::chrono::high_resolution_clock`, which may shift upon NTP synchronization, at µs granularity. The relatively low fidelity and unstable clock domain makes it difficult to measure small overheads and correlating trace markers against OS kernel events, which are typically recorded using `steady_clock` or other monotonic CPU clocks. Third, trace events are recorded in a manner that is likely to perturb normal execution. Each trace marker emitted requires `std::string` and `std::vector` operations, making the cost per event non-trivial. Furthermore, all threads record to a shared ring buffer, contending for the same cache line(s) and also blocking on a periodic consumer thread that writes buffer contents to disk every 100 ms. This makes it all but guaranteed that trace events perturb the scheduler execution it is intended to measure.

Our design. To supplement the existing profiler while meeting our needs we have implemented `Trace.hpp`, an alternative, lightweight tracer for GR4. Our tracer addresses each of the above limitations by construction.

First, `Trace.hpp` is deliberately free of dependencies so that it may be included by `Block.hpp` as well as `Scheduler.hpp` without enlarging their include graphs. Trace markers may therefore be emitted from inside `work()` itself. Second, markers are timestamped with `std::chrono::steady_clock` at nanosecond resolution. This allows more fine-grained measurements and the ability to merge with kernel timelines, such as those collected via `ftrace`. This is also the clock in which our scheduling policies express job release times and absolute deadlines, so latencies and and deadline misses may be computed exactly. Third, emitting a marker appends one 32 B trivially copyable record to a ring owned by the emitting thread. There is no shared state between threads and hence no contended cache line, allocation, string formatting, I/O, or possibility of blocking. The ring overwrites its oldest record rather than filling, so no capacity test or

backpressure mechanism is required. Records leave memory only when a capture is explicitly written out by calling `gr::trace::dump()`, after execution has completed.

In addition to the design decisions above, we use trace markers with a considerably widened range of expressible record categories. These categories include deadline misses, the reason a worker idled rather than worked, the structure and overhead of each scheduling sweep, and per-block and per-worker aggregate statistics. Within `work()`, each invocation records the exact number of input and output samples it processed. This permits end-to-end datapath latency to be calculated precisely and execution time to be regressed against the number of samples processed. We are also in the process of implementing automated tools to perform said latency and timing analysis. Finally, our tracer implementation replicates some of the existing profiler's desirable characteristics, such as having zero cost when compiled out and the ability to output to Chrome Trace JSON format for import into popular analysis tools such as Catapult (The Chromium Authors, 2015) and Perfetto (Google, 2025), as illustrated in Fig. 6.

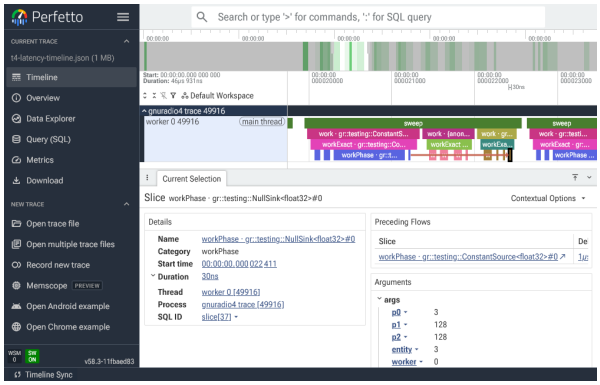

Figure 6. Example of a flowgraph-execution timeline recorded using our tracer and viewed in Perfetto.

4. Experiments and Evaluation

In this section, we present preliminary results from our evaluation of our implemented scheduling policies using IEEE 802.11p flowgraphs. While our implementation is a work in progress, we expect the observed trends to hold across future work.

4.1. Platform

We ran our experiments in a virtual machine (VM) on a Proxmox hypervisor equipped with a 32-core AMD Ryzen Threadripper PRO 5975WX. To accurately replicate execution behavior on a smaller CPU, the VM was restricted to 8 cores, affinity-masked to one CCX on the host, and provided x64-64-v2-AES processor features. The host CPU cores and VM's virtual CPU cores used for the evaluation were further isolated to execute *only* GR4 worker threads to mitigate potential perturbation by OS or background tasks. The VM ran Ubuntu 26.04.1 LTS with Linux 7.0 and was provided 24 GB of RAM.

Table 1. The five scenarios. Rates are per receiver, in graph order. The number of workers equals the number of receivers.

Scenario	Receivers (Msps)	Total (Msps)
One Rx	2.5	2.5
Two equal Rx	2.5, 2.5	5
Light	1.25, 1.25, 2.5, 5	10
Medium	1.25, 2.5, 2.5, 5	11.25
Heavy	2.5, 2.5, 5, 5	15

4.2. Workload

For our experiments, we ported `gr-ieee802-11` (Bloessl et al., 2018), a GR3 implementation of IEEE 802.11 protocols, to GR4. Specifically, we utilized the IEEE 802.11p receiver flowgraph, which contains a mix of simpler blocks (e.g., multiply, delay, average) and computationally expensive blocks (e.g., FFTs, decoding, parsing). This workload is ideal for real-time analysis because the IEEE 802.11p specification imposes hard timing constraints for vehicle-based wireless transmission, requiring bounded end-to-end latency of 100 ms (Kenney, 2011; Committee, 2016).

We ran up to 4 receivers concurrently, denoted Rx 0 through Rx 3. Each receiver was configured with an input sample rate in million samples per second (Msps). We tested using implicit deadlines ($D_i = T_i$) based on their input sample rates for all blocks except the throttle and source blocks; thus, for EDF and RM, the higher the input rate, the earlier the deadline, whereas RR is unaffected. The source and throttle blocks ran uncapped and were prioritized above other work (for EDF, they were given a 1 µs relative deadline, and deadline misses for these blocks were ignored). All other blocks uniformly batched 1,024 samples per invocation.

We measured *end-to-end response times*, the time from a batch exiting the receiver throttle to that data reaching the sink node. A batch is late if its end-to-end response time exceeds the receiver's assigned period. We set the number of worker threads equal to the number of receivers. Hypervisor-related interference events, such as whole-process stalls, filtered out from our response time measurements. Specifically, response times that exceeded the receiver's median five-fold that occurred in a cluster across *all* receivers simultaneously, accounting for approximately 0.007% of data points, were filtered out.

4.3. Experiment overview

Our experiments were conducted as follows. The first experiment used few receivers and aimed to leverage low-overhead round-robin scheduling, where prioritizing specific blocks is unnecessary. The second experiment varied multiple receiver sample rates, where scheduling decisions are meaningful. The third experiment examined only the high-rate receivers in a system with other, lower-rate receivers. We denote a set of receivers and their configurations as a *scenario*. Tab. 1 details the scenarios used in the three experiments. We ran each scenario 20 times for 60 s per scheduling policy, for 300 runs totaling about 5 hours.

4.4. Experiment 1: Simple Receivers

We tested two scenarios in this experiment: 'One Rx' and 'Two equal Rx.' For each scenario and scheduling policy, we recorded the mean, median, 25th and 75th percentiles, and max observed end-to-end response times. With few receivers, Tab. 2 confirms the expected results: RR's low overheads are more valuable than deadline-based scheduling decisions when the workload is simple. However, adding a second receiver showed that RR disproportionately prioritized the first receiver, whereas EDF kept end-to-end response times consistent by prioritizing work with earlier deadlines, as shown in Fig. 7.

Table 2. End-to-end response time in µs, One receiver and two equal 2.5 Msps receivers. Lower is better; best in bold.

Scenario	Metric	RR	EDF	RM
One RX	Mean	**67**	69	101
One RX	Max	**557**	560	570
Two Equal	Rx 0 Mean	**82**	87	115
Two Equal	Rx 1 Mean	119	**86**	111

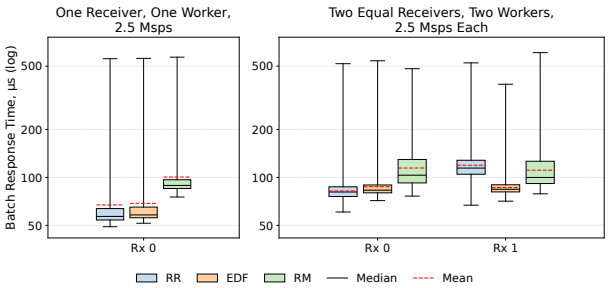

Figure 7. Batch response time with one receiver on one worker (left) and two identical 2.5 Msps receivers on two workers (right). Boxes span the 25th to 75th percentile and whiskers span the full range. With two receivers, RR favored the graph registered first, while EDF served both similarly.

4.5. Experiment 2: Multi-Rate Receivers

In this experiment, we tested four concurrent receivers configured with 'light,' 'medium,' and 'heavy' input-arrival rates. Tab. 3 and Fig. 8 report the input rates and response time metrics for the 'light' workload.

Table 3. Mean end-to-end response time in µs, Light scenario.

Receiver	RR	EDF	RM
Rx 0 (1.25 Msps)	**136**	149	173
Rx 1 (1.25 Msps)	**141**	161	169
Rx 2 (2.5 Msps)	143	**138**	140
Rx 3 (5 Msps)	151	126	**112**

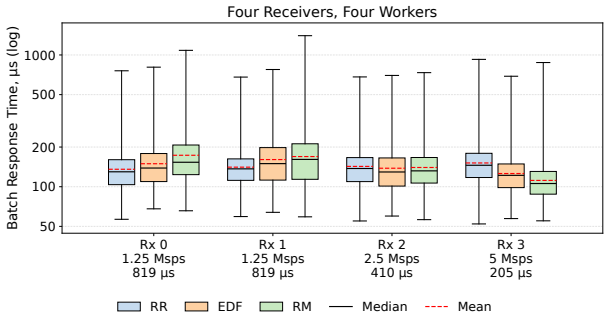

Figure 8. End-to-end response times per receiver in the 'light' scenario. The trend of RR prioritizing earlier receivers is clear: under RR each subsequent receiver showed increased end-to-end response times such that Rx 2 and Rx 3 performed worse under RR than EDF. RM performed best for the highest-rate receiver since RM prioritizes by rate.

We observed that RR's response times grew worse for every receiver past the first, and that EDF outperformed RR on Rx 3 and Rx 4. We also noted that RM performed best on the fastest receiver, which we investigate next.

Highest-rate receivers. When examining only the highest-rate receivers' response times, as shown in Tab. 4, RM offered the lowest mean end-to-end response times. RM also significantly reduced the median, 25th, and 75th percentiles for the highest-rate receiver, as shown in Fig. 9. This trend was especially pronounced under the 'heavy' scenario.

Table 4. Mean batch response time in µs of the 5 Msps receiver built last in each mix.

Scenario	RR	EDF	RM
Light	151 µs	126 µs	112 µs
Medium	172 µs	138 µs	123 µs
Heavy	210 µs	212 µs	158 µs

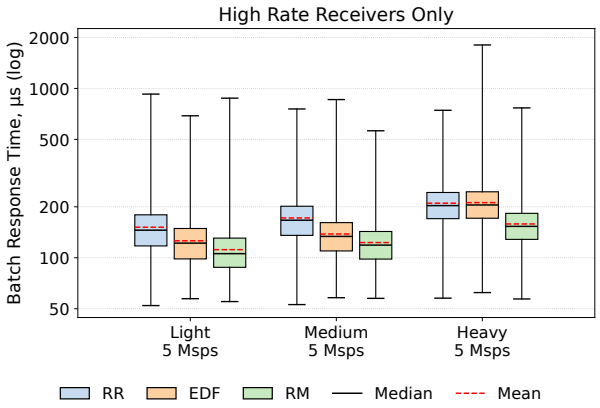

Figure 9. Response times of Rx 3 (5 Msps) in each scenario. RM served it best, at the cost of the slowest receivers (not shown).

Table 5. Trace-overhead measurement results.

Record categories enabled	Records	ns/record
`work`	12296	23.6
`release`	13321	27.6
`release|select|deadline`	20494	31.4
`work|workExact`	18444	35.3
`work|workExact|workPhases`	43024	46.6

4.6. Tracing Overheads

To measure the average overhead of emitting a single trace marker, we executed a simple flowgraph under our EDF policy with and without tracing enabled. The flowgraph was a single `Constant Source` → `Copy` → `NullSink` chain, executed with a fixed input-batch size of 512 samples per `work()` invocation for 524,288 placed in the source input buffer prior to initialization. We deliberately chose a small batch-size ceiling and computationally trivial blocks. This allowed us to observe a large number of trace events in aggregate, minimally affected by potential variance in actual block execution times. As shown in Tab. 5, the cost of recording a single trace event is in the tens of nanoseconds, staying under 50 ns even when recording three marker categories per event.

For comparison, we repeated the same experiment with the existing profiler. Because the built-in profiler tracks far fewer trace events by design, we executed the flowgraph with a much larger 8,388,608 input samples per run. While per-record costs to write to the in-memory buffer were comparable to our tracer at 43.8 ns, the write-to-disk operations consumed approximately 486 ns per record, and by design were not deferred to after execution completion.

5. Related Work

Much work has been contributed to enhance the scheduling and performance flowgraph execution in GNU Radio. The thread-per-block (TPB), scheduler which has been the GR3 default since 2008 (Blossom, 2008), leaves thread scheduling to the operating system and controls execution only through per-call chunk sizes (Bloessl et al., 2019). Upstream work has centered primarily on instrumentation, tuning knobs, and accelerator-aware custom buffers (Rondeau et al., 2013; GNU Radio Project, 2019; Sorber, 2021). Bloessl et al. showed that an application-aware scheduler could substantially outperform TPB (Bloessl et al., 2019). This motivated the `newsched` runtime with pluggable schedulers (Morman & Bloessl, 2021; Morman & GNU Radio contributors, 2022), which evolved into today's GR4 scheduler (Steinhagen et al., 2023). Neither GR3 nor GR4 currently provide deadline-aware scheduling or formal response-time guarantees.

Eisenklam et al. were the first to apply rigorous real-time scheduling theory and analysis to GNU Radio. Their work studies the execution times and graph response times of signal-processing tasks under batch-processing of sample data. For their evaluations, they modified GR3 to support Linux's `SCHED_DEADLINE` and `SCHED_FIFO` scheduling classes (Eisenklam et al., 2024). He & Ward (2025) extends this work, exploring the benefits and tradeoffs of merging of blocks at compile time and presenting a case study implemented in GR3 and using Linux's scheduling classes as before. To our knowledge, our work is the first to incorporate real-time scheduling capabilities in GR4.

6. Conclusion

This paper presented our work in progress toward real-time scheduling support in GR4. We have added priority-based scheduling to the worker loop through the `SchedulingPolicyLike` template, which composes with GR4's existing scheduler classes and leaves RR the default, so existing flowgraphs run unchanged. Atop that interface we have implemented RM and EDF as an exemplar static-priority and dynamic-priority scheduling policies, respectively.

We have also built a lightweight tracer that may be called from within a block's `work()` method, yielding nanosecond-resolution execution timelines at a cost low enough to measure the scheduler's own overheads. Our preliminary evaluation on IEEE 802.11p receivers showed that the best policy depended on the workload: RR sufficed when no block needed prioritizing, EDF balanced lateness across receivers of differing rates, and RM favored the fastest receiver at the slower ones' expense. All code is open source and provided at `https:`

//github.com/gosephjoh/gnuradio4 under the modular-scheduling branch (Goh et al., 2026).

Our continuing work is to improve dynamic-priority policies' scheduler overheads and to implement additional latency and execution-time analysis utilities. We also plan to explore scheduling policies and flowgraph optimization tooling which explicitly consider the timing effects of batch processing and block merging. Finally, we plan to explore implementation of global scheduling policies and provide tighter integration with OS schedulers such as Linux's scheduling classes.

Acknowledgments

This work was supported by a grant from the Amateur Radio Digital Communications (ARDC) foundation.

References

Bloessl, Bastian, Segata, Michele, Sommer, Christoph, and Dressler, Falko. Performance Assessment of IEEE 802.11p with an Open Source SDR-based Prototype. *IEEE Transactions on Mobile Computing*, 17(5):1162–1175, May 2018. doi: 10.1109/TMC.2017.2751474.

Bloessl, Bastian, Müller, Marcus, and Hollick, Matthias. Benchmarking and profiling the GNU Radio scheduler. *Proceedings of the GNU Radio Conference*, 4(1), September 2019. URL https://pubs.gnuradio.org/index.php/grcon/article/view/64.

Blossom, Eric. Multi-processor scheduler now available for testing. Post to the discuss-gnuradio mailing list, July 2008. URL https://lists.gnu.org/archive/html/discuss-gnuradio/2008-07/msg00159.html.

Committee, DSRC Technical. *On-Board System Requirements for V2V Safety Communications*, March 2016. URL https://doi.org/10.4271/J2945/1_201603.

Eisenklam, Abigail, Hedgecock, Will, and Ward, Bryan C. Job-level batching for software-defined radio on multicore. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pp. 375–387. IEEE, December 2024. doi: 10.1109/RTSS62706.2024.00039.

George, Laurent, Rivierre, Nicolas, and Spuri, Marco. Preemptive and non-preemptive real-time UniProcessor scheduling. Research Report RR-2966, INRIA, 1996. URL https://inria.hal.science/inria-00073732.

GNU Radio Project. Block thread affinity and priority. GNU Radio Wiki, 2019. URL https://wiki.gnuradio.org/index.php/Block_Thread_Affinity_and_Priority. Accessed 2026-09-21.

Goh, Joseph, He, Tiancheng, and Ali, Syed W. gnuradio4, 2026. URL https://github.com/gosephjoh/gnuradio4/tree/modular-scheduling. Fork of the GNU Radio 4.0 repository (https://github.com/fair-acc/gnuradio4). Licensed LGPL-3.0-or-later with a static linking exception.

Google. Perfetto, 2025. URL https://github.com/google/perfetto.

He, Tiancheng and Ward, Bryan C. Real-time scheduling and batching in GNU Radio for bounded response times. In *Proceedings of the 15th GNU Radio Conference*, 2025. URL https://events.gnuradio.org/event/26/contributions/770/attachments/226/604/GRCON25__Inter__and_Intra__Batching_Final.pdf.

Jeffay, Kevin, Stanat, Donald F., and Martel, Charles U. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the Twelfth Real-Time Systems Symposium*, pp. 129–139, San Antonio, TX, USA, December 1991. IEEE Computer Society Press. doi: 10.1109/REAL.1991.160366.

Kenney, John B. Dedicated short-range communications (DSRC) standards in the United States. *Proceedings of the IEEE*, 99(7):1162–1182, July 2011. doi: 10.1109/JPROC.2011.2132790.

Morman, Josh and Bloessl, Bastian. A modular future for GNU Radio. Talk, Free Software Radio devroom, FOSDEM 2021, February 2021. URL https://archive.fosdem.org/2021/schedule/event/fsr_a_modular_future_for_gnu_radio/.

Morman, Josh and GNU Radio contributors. newsched: The GNU Radio 4.0 runtime proof of concept. GitHub repository (archived), 2022. URL https://github.com/gnuradio/newsched.

Rondeau, Thomas W., O'Shea, Timothy, and Goergen, Nathan. Inspecting GNU Radio applications with ControlPort and performance counters. In *Proceedings of the Second Workshop on Software Radio Implementation Forum (SRIF '13)*, pp. 65–70, Hong Kong, China, August 2013. ACM. doi: 10.1145/2491246.2491259.

Sorber, David. The state of GNU Radio accelerator device support. Presentation at the 11th GNU Radio Conference (GRCon 2021), September 2021. URL https://events.gnuradio.org/event/8/contributions/37/.

Steinhagen, Ralph J., Kretz, Matthias, Krimm, Alexander, Kozel, Derek, Morman, Josh, Čukić, Ivan, and Osterfeld, Frank. GNU Radio 4.0 for real-time signal-processing and feedback applications at FAIR. In *Proceedings of the 14th International Particle Accelerator Conference (IPAC'23)*, pp. 4688–4691, Venice, Italy, May 2023. JACoW Publishing. doi: 10.18429/JACoW-IPAC2023-THPL099. Paper THPL099.

The Chromium Authors. Catapult: Performance tools for Chrome (including trace-viewer), 2015. URL https://github.com/catapult-project/catapult.

The Chromium Authors. Trace event format. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU, 2016. Accessed: 2026-09-21.