Real-Time Scheduling and Batching in GNU Radio for Bounded Response Times

Tiancheng He TIANCHENG.HE@VANDERBILT.EDU
Vanderbilt University, Nashville, TN 37212 USA

Bryan C. Ward BRYAN.WARD@VANDERBILT.EDU
Vanderbilt University, Nashville, TN 37212 USA

Abstract

This paper investigates the application of real-time scheduling algorithms and theory to GNU Radio. The existing GNU Radio scheduler includes heuristics aimed at improving throughput, but such heuristics can also lead to temporal nondeterminism and conflict with real-time scheduling analysis techniques. This paper shows how to apply the Linux `SCHED_DEADLINE` real-time scheduler, and how to more predictably *batch* processing of multiple samples or blocks to reduce system overheads and enable the application of real-time scheduling theory. This enables the analytical calculation of latency bounds while also serving as a tool to understand tradeoffs in how much processing can be supported on a given platform without overloading it. The paper presents two batching methods: *Intra-block* batching processes fixed-size sample batches within a block, while *inter-block* batching merges different blocks to reduce context switches and improve data locality. Experimental results demonstrate significant improvements in overheads due to batching. Batching-based real-time models are presented that reflect these effects, and analytical evaluations are conducted to show tradeoffs between batching and latency bounds. Finally, these concepts are demonstrated in GNU Radio with a simple NBFM receiver.

1. Introduction

GNU Radio is the most widely used open-source framework for software-defined radio (SDR). Its success stems from a design philosophy optimized for flexibility and throughput: blocks are scheduled opportunistically, each block is assigned its own thread, and the runtime aims to maximize throughput. This architecture has enabled GNU Radio to flourish as a rapid prototyping tool for wireless communication.

However, this throughput-oriented design comes at a cost. GNU Radio applications can exhibit significant jitter and unpredictable latency, especially when handling high sample rates. Benchmarking studies have shown that opportunistic batching—where blocks process as many samples as possible when scheduled—can reduce average overhead but introduces variability in execution times across invocations (Bloessl et al., 2019). This nondeterministic batching strategy does not comport with existing real-time scheduling and analysis techniques and so there are no analytical bounds on end-to-end latency. This can be an impediment to adoption especially in application areas where real-time communication is necessary, and high jitter can be problematic. Furthermore, it leaves developers without strong tools to reason about the *worst-case* timing, and therefore how changes in workloads affect worst-case performance.

There has been a significant body of work over the last 50 years on real-time scheduling, which is aimed at providing analytical guarantees that timing requirements will be satisfied at runtime. This is necessary in many cyber-physical applications, especially those involving control (e.g., avionics, autonomous vehicles, etc.), as a violation of timing constraints could have implications of safety or mission effectiveness. Real-time scheduling and analysis requires explicitly modeling tasks with parameters such as execution costs, deadlines, and periodicity, and with such parameters can deriving provable analytical bounds on latency and therefore guarantee timing requirements will be satisfied at runtime. However, a key distinguishing feature between prior real-time scheduling work and GNU Radio workloads is the sample rate, which can be orders of magnitude faster than tens of hertz common in control systems.

Our goal in this paper is to bridge these two worlds and enable real-time scheduling algorithms and analysis to be applied in the context of high-frequency software-defined

Proceedings of the 15^{th} GNU Radio Conference, Copyright 2025 by the author(s).

radio applications. That bridge is *batching*. By introducing deterministic batching at both the intra-block and inter-block levels, GNU Radio workloads can be parameterized using well-defined real-time models, and can be scheduled using off-the-shelf real-time scheduling algorithms, such as `SCHED_DEADLINE` in Linux. Furthermore, these models enable real-time analysis results to be directly applied to capitalize on 50 years of real-time scheduling theory results to bound latency and maximize the capacity of a given computing platform.

The contributions of this paper are as follows:

- **Integration of real-time scheduling into GNU Radio:** Demonstrates how GNU Radio applications can be executed under the Linux `SCHED_DEADLINE` scheduler, enabling end-to-end timing analysis using established analyses.
- **Intra-block batching:** Introduces a fixed-size batching mechanism within blocks, improving predictability of execution times while minimizing initialization costs and facilitating schedulability analysis.
- **Inter-block batching:** Proposes the merging of blocks into super-blocks, reducing context-switch overheads and improving cache performance.
- **Comprehensive evaluation:** Provides experimental evidence that batching significantly improves predictability, reduces overheads, and increases achievable utilization compared to the standard GNU Radio scheduler.

2. Real-Time Scheduling Background

Real-time scheduling provides a framework for reasoning about *when* computations occur, not just what results they produce. In contrast to throughput-oriented schedulers such as the default in GNU Radio, real-time schedulers are designed to be predictable in the worst case to ensure any timing requirements can be satisfied at runtime. In SDR, predictable timing can improve jitter, reduce latency, and improve quality of service. Furthermore, real-time analysis techniques can provide analytical bounds on latency and guarantee throughput. However, to apply real-time scheduling algorithms and analysis, the workloads of GNU Radio processing must be modeled.

2.1. Real-Time Models and Scheduling

Real-time systems are modeled as a set of *tasks*, where each task generates a sequence of *jobs* over time. Each job is characterized by:

- **Execution time (e):** worst-case processor time required to complete the job.

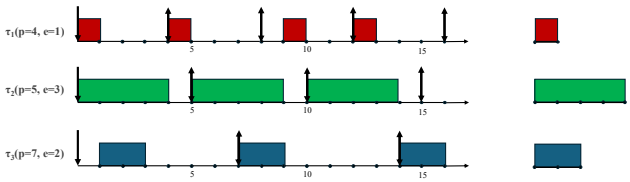

Figure 1. Example of three tasks under real-time scheduling.

- **Period (p):** the minimum time between job releases.
- **Deadline (d):** the target completion time after release.[1] Often $p = d$.

These parameters form the basis for schedulability analysis.

Scheduling algorithms. Scheduling algorithms determine which job should execute at any given moment. General-purpose operating systems prioritize fairness and responsiveness, but real-time scheduling algorithms are designed to ensure that jobs meet their timing requirements. Fig. 1 shows a simple example of the earliest-deadline-first (EDF) scheduling algorithm, in which ready jobs with the earliest deadline are executed first. EDF is a common real-time scheduling algorithm with many useful analytical and real-time optimality properties.

EDF properties. EDF is of particular interest because it is *optimal* for uniprocessor soft real-time workloads: if any schedule can meet all deadlines, EDF can. Furthermore, it has been shown that on a multiprocessor system, EDF is soft-real-time optimal in that it can guarantee bounded latency. In other words, the platform can be fully utilized without latency growing unboundedly for any tasks (Devi, 2006). Modern Linux kernels include real-time scheduling classes. The most relevant for this work is `SCHED_DEADLINE`, which implements a *global* EDF policy, in which if there are m processors, the m earliest-deadline jobs are scheduled at any time. Each task under `SCHED_DEADLINE` is assigned (e, p, d) parameters, corresponding to execution budget, period, and deadline. The scheduler enforces these parameters, ensuring predictable CPU allocation and enabling real-time analysis on commodity hardware. This makes `SCHED_DEADLINE` a natural bridge between GNU Radio workloads and real-time scheduling theory.

[1] In soft real-time systems, a deadline miss is not catastrophic but instead represents a degradation in quality of service. In this setting, the deadline is better interpreted as a priority point that determines how the scheduler orders jobs, rather than a "do-or-die" constraint.

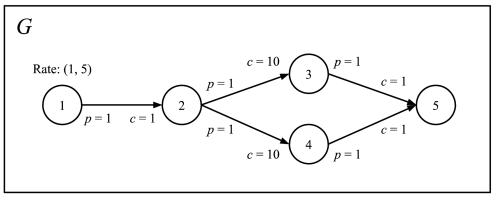

Figure 2. Example of a Processing Graph Method (PGM) graph. Blocks are nodes and buffers are edges with specified produce, threshold, and consume parameters.

3. DAG Models and the Processing Graph Method

A GNU Radio flowgraph can be represented as a *directed acyclic graph* (DAG). Each block in a flowgraph corresponds to a node in the DAG, and the buffers that connect blocks correspond to edges. This DAG representation highlights the inherent precedence constraints: a downstream block can only execute once its predecessor has produced sufficient data. While GNU Radio implements this model inside its runtime, real-time analysis requires a more precise graph-based formalism.

Several DAG-based models exist in the literature. Among them, the *Synchronous Dataflow* (SDF) model introduced by Lee and Messerschmitt (Lee & Messerschmitt, 1987) has been widely used for signal processing. SDF represents each block as consuming and producing a fixed number of tokens, which makes scheduling analyzable. However, SDF lacks the ability to capture buffer thresholds, which play an important role in GNU Radio's scheduling decisions (e.g., blocks that only fire when buffers are half full).

3.1. Processing Graph Method as a Bridge

The *Processing Graph Method* (PGM) extends SDF by explicitly modeling thresholds, making it well-suited to represent GNU Radio flowgraphs. In a PGM graph, each edge is annotated with three parameters: a *produce amount*, a *consume amount*, and a *threshold*. Consider an edge from node u to node w: every execution of u produces $p_{u\to w}$ tokens, while w can only fire when at least $t_{u\to w}$ tokens are available, at which point it consumes $c_{u\to w}$ tokens. These semantics directly mirror GNU Radio's runtime behavior, where buffers accumulate samples and downstream blocks wake up only after thresholds are satisfied. Figure 2 illustrates a simple PGM graph.

By mapping GNU Radio flowgraphs into the PGM formalism, we obtain a bridge between GNU Radio's flowgraph dependencies and the analyzable world of real-time DAG scheduling. PGM graphs retain the streaming semantics of GNU Radio, allowing multiple invocations of a flowgraph to overlap in time while enabling schedulability analysis using well-studied real-time theory.

Prior work has shown how a PGM graph can be transformed to a rate-based task model, and subsequently to the classical independent tasks described previously of EDF, enabling seminal EDF results to be applied to PGM graphs (Liu & Anderson, 2010). We briefly review these prior results.

3.2. Rate-Based Task Parameters

To make analysis tractable, each PGM node can be transformed into a *rate-based (RB) task*. A task $\tau_u = (x_u, y_u)$ has an execution rate, meaning it executes x_u times in any interval of length y_u. From this, we assign a relative deadline

$$d_u = \frac{y_u}{x_u},$$

and require $d_u \leq d_v$ whenever there is an edge from u to v. This ensures rates do not increase along any path in the graph. The execution cost e_u of each node is also provided, so an RB task is fully specified as

$$\tau_u = (x_u, y_u, d_u, e_u).$$

3.3. Sporadic Task Model

RB tasks can be further transformed into classical independent sporadic tasks with three parameters: period p, relative deadline d, and execution cost e. Following (Liu & Anderson, 2010), we set $p = y/x$ and assume implicit deadlines ($d = p$). Thus, for each task $\tau_v \in T^S$:

$$\tau_v = (p_v, d_v, e_v) = \left(\tfrac{y_v}{x_v}, \tfrac{y_v}{x_v}, e_v\right).$$

3.4. Utilization and Latency

From the sporadic task model, we can compute utilization for each task

$$u_v = \frac{e_v}{p_v},$$

and total utilization $U = \sum_{\tau_v \in T^S} u_v$. In soft real-time systems, $U \leq M$ is a sufficient condition for schedulability on M processors under any G-EDF-like policy (Devi, 2006). Bounded tardiness in turn implies bounded end-to-end latency. Following Goddard (Goddard, 1998), we define latency L as the maximum time between the arrival of data at a source node and its completion at a sink node. This allows us to reason about how batching affects both utilization and latency in GNU Radio flowgraphs.

We omit the details of the analysis here, but emphasize that this body of prior work on real-time scheduling allows us to convert a GNU Radio flowgraph into real-time tasks with associated execution times, periods, and deadlines that can be scheduled with EDF under Linux `SCHED_DEADLINE`.

After this transformation, prior real-time analysis techniques can be applied to guarantee bounded end-to-end latency even while fully utilizing all CPUs. We refer the interested reader to (Eisenklam et al., 2024; Liu & Anderson, 2010) for further details of the analysis.

4. The Marginal Cost Model

To minimize context switching and other initialization costs, the GNU Radio scheduler only invokes a block to process samples when sufficient samples are ready to be processed. The total time that a block executes is therefore a function of the number of samples processed. In our previous work (Eisenklam et al., 2024), we developed the marginal cost model to model this phenomenon and observed that there is a fixed initialization cost at the start of the job, while subsequent samples can be processed much more quickly.

To illustrate this, we conducted a series of profiling experiments on GNU Radio blocks running on a quad-core Raspberry Pi 5. Using GNU Radio's "Head" blocks, we controlled the number of samples processed per invocation, applied CPU affinity masks to isolate workloads, and scheduled block threads non-preemptively under the Linux FIFO policy. Execution times were recorded using GNU Radio Performance Counters.

Figure 3 shows a representative result for the `add_const` block. The relationship between the number of samples processed and the measured execution time is highly linear, with a regression fit of the form

$$e(c) = I + \Delta \cdot c,$$

where c is the number of samples processed in an invocation, I is the *initialization cost*, and Δ is the *marginal cost per sample*. In this example, the initialization cost dominates, with $I \approx 11{,}081$ cycles, while the marginal cost is only $\Delta \approx 2.8$ cycles per sample. This disparity highlights why GNU Radio's default opportunistic batching strategy—where blocks process all available input samples at once—is effective at amortizing the large fixed cost across many samples.

To formalize this behavior, we defined the **Marginal Cost Model** (Eisenklam et al., 2024). Suppose a sporadic task $\tau_v = (p_v, d_v, e_v)$ corresponds to the execution of a block. The execution cost e_v can be decomposed as:

$$e_v = I_v + \Delta_v \cdot c_v,$$

where I_v is the initialization cost of block τ_v, Δ_v is the marginal cost per sample, and c_v is the number of samples processed in the invocation. This model explicitly captures the trade-off between per-invocation overheads and per-sample costs, providing a foundation for analyzing batching strategies. As later sections show, both intra-block and inter-block batching can be understood as techniques for reducing the impact of the large I_v term by amortizing it across more samples or across multiple blocks.

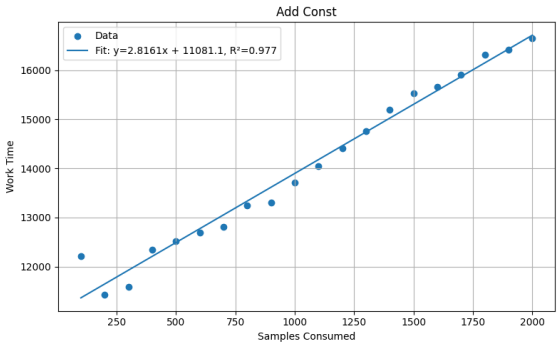

Figure 3. Execution time of the `add_const` block as a function of the number of samples consumed. The linear regression fit highlights the large initialization cost I and the much smaller marginal cost per sample Δ.

5. Intra-Block Batching Techniques

The marginal cost model highlights the tradeoff between fixed per-invocation overheads and marginal per-sample costs. To reduce overheads and improve predictability in GNU Radio, this paper reviews two batching techniques: *uniform batching* and *rate-exploiting batching* (Eisenklam et al., 2024).

5.1. Uniform Batching

Uniform batching enforces a constant batch size for each block in the flowgraph. Instead of allowing execution to depend on buffer occupancy, each block is configured to process exactly B samples per invocation. This eliminates variability in execution times, making it straightforward to parameterize tasks for real-time schedulers such as `SCHED_DEADLINE`. Furthermore, larger batch sizes amortize initialization costs I_v across more samples, thereby reducing effective utilization. However, excessively large batch sizes can increase end-to-end latency, as data waits longer in buffers before being processed.

5.2. Rate-Exploiting Batching

While uniform batching improves predictability, it ignores the heterogeneous production and consumption rates between blocks. Rate-exploiting batching adapts the batch size of each block to align with the dataflow rates of its neighbors. For example, if an upstream block produces p samples per invocation and a downstream block consumes c samples, the batch sizes can be chosen as integer multiples of $\mathrm{lcm}(p, c)$ as Fig. 4 illustrates. This alignment re-

duces the number of partial buffer fills and synchronizations, lowering the total number of block activations and further reducing context-switching overhead.

In addition, rate-exploiting batching preserves dataflow consistency by ensuring that batched executions respect the original precedence constraints of the DAG. This technique combines the predictability of fixed batch sizes with improved efficiency from rate-aware alignment.

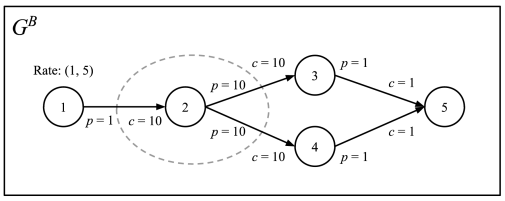

Figure 4. Example of inter-block batching.

6. Inter-Block Batching Techniques

While intra-block batching is aimed at amortizing the fixed initialization cost over many samples, we next study the possibility of further reducing context switches and initialization costs by merging multiple blocks together. Our hypothesis is that in so doing we not only reduce the fixed costs of context switching, but also improve data locality and ensure that produced samples can be consumed and processed while still cache hot.

To illustrate inter-block batching, we first consider a small example graph shown in Fig. 5. Each node represents a block, and edges represent the data dependencies between them. Within such a graph, there are two natural ways to merge nodes into super-blocks: *serial merging*, where nodes along a chain are grouped together, and *parallel merging*, where multiple nodes that consume from the same input are grouped.

The choice of which merging style to apply depends on both (1) the rate relationships between nodes and (2) the degree to which cache reuse or amortized overhead can be exploited. We now illustrate each type of merging in turn.

6.1. Inter-block Serial Merging

Serial merging occurs when two or more nodes are connected in sequence and operate at the same rate. By executing these nodes within the same thread, cache affinity is improved and synchronization overhead is reduced. This is especially effective when the merged nodes repeatedly access the same data structures or coefficients.

In Fig. 5, nodes τ_2 and τ_4 form a natural chain with matched rates, making them candidates for serial merging. After merging, they are replaced by a single super-node, reducing thread wakeups and context switches. This leads to a smaller effective initialization cost I while preserving the marginal cost model.

6.2. Inter-block Parallel Merging

Parallel merging occurs when multiple nodes share the same input and produce outputs at the same rate. In such cases, the input data can be loaded once and reused by several computations, rather than requiring separate invocations for each node. This reduces redundant memory traffic and amortizes the initialization cost across multiple tasks.

In Fig. 5, nodes τ_2 and τ_5 originate from the same predecessor. By merging them into a composite task, duplicated buffer accesses are eliminated and execution jitter is reduced, leading to more predictable performance.

7. Evaluation

This section evaluates the effectiveness of intra-block and inter-block batching for improving the predictability and efficiency of GNU Radio workloads. The intra-block results are obtained from micro-benchmarks of individual GNU Radio blocks, while the inter-block results are based on large-scale experiments using randomly generated PGM applications.

7.1. Experimental Setup

- **Inter-block experiments:** We generated 1,000 PGM applications for each parameter configuration, with the source node set to a 1 MHz sampling rate. Each node was randomly designated as a one-to-one function (e.g., multiply, shift) or a decimating function (e.g., filter). We applied both uniform batching and rate-exploiting batching transformations, converted the PGMs to sporadic task sets, and evaluated them under a Global EDF scheduling policy.
- **Intra-block experiments:** We benchmarked individual GNU Radio blocks by varying the number of input samples processed per invocation, while recording execution times in processor ticks using GNU Radio Performance Counters. Linear regression was used to model the relationship between samples processed and execution cost, capturing both the per-invocation startup cost (intercept) and the per-sample marginal cost (slope). We then compared these parameters before and after merging serially connected blocks.

7.2. Intra-Block Batching Results

This subsection builds on the framework introduced in our previous work on job-level batching for SDR (Eisenklam et al., 2024). We first evaluate the benefits of intra-

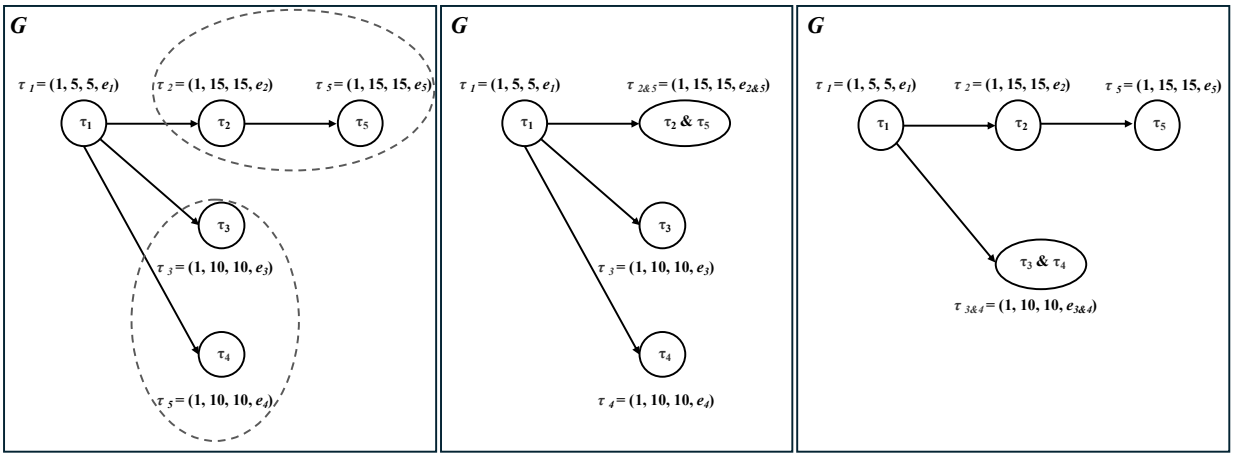

Figure 5. Example of inter-block batching. Both serial and parallel merging opportunities are shown.

block batching on randomly generated PGM graphs. Recall that in intra-block batching, consecutive nodes in a flowgraph are merged into a super-node when doing so reduces context-switching and improves cache affinity without violating execution-rate constraints. The main metrics of interest are (i) the total system utilization under G-EDF scheduling, and (ii) the end-to-end latency bound derived from the sporadic task transformation.

Experiment Setup. For each configuration of graph parameters, we generated 1,000 random PGM graphs with node rates drawn from the same distributions used in prior work. Each source node was fixed to a 1 MHz sampling rate, and nodes were randomly assigned as either one-to-one transformations (e.g., frequency shift, multiply-const) or decimating functions (e.g., filtering). As in (Liu & Anderson, 2010), we transformed each graph into an equivalent sporadic task set and computed both total utilization and latency under global EDF. We then compared these metrics before and after applying intra-block batching.

Results. Fig. 6 and Fig. 7 report the average utilization and latency bounds as a function of the batch size for light and heavy graph configurations, respectively. Several observations follow:

- **Utilization reduction:** For small batch sizes ($N = 2$ to $N = 6$), intra-block batching reduces total utilization substantially. For example, in the light-graph case, a batch size of $N = 4$ reduces the average utilization by nearly two-thirds, lowering the number of processors required for schedulability guarantees.
- **Latency trade-off:** The corresponding increase in end-to-end latency is minimal for small batch sizes (less than 0.1 ms on average). However, latency grows approximately linearly with the batch size, showing the expected trade-off between efficiency and responsiveness.
- **Diminishing returns:** Larger batch sizes continue to reduce utilization, but with diminishing marginal benefit, while latency continues to increase. This demonstrates that moderate batch sizes strike the best balance between schedulability and bounded latency.

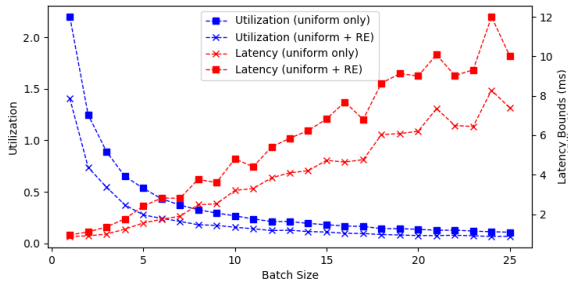

Figure 6. Utilization and latency vs. batch size for light graph configurations.

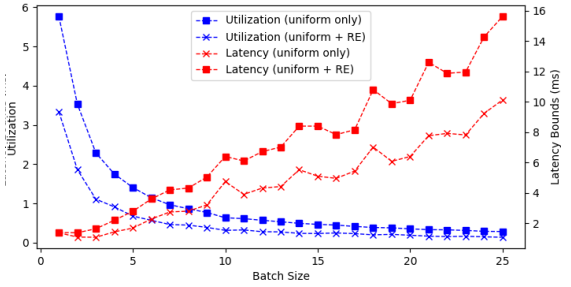

Figure 7. Utilization and latency vs. batch size for heavy graph configurations.

Case Study: Narrowband FM Receiver. To demonstrate the applicability of intra-block batching to real-world GNU Radio workloads, we evaluated a GNU Radio narrowband FM (NBFM) receiver (Duggan, 2024). The flowgraph consists of a source followed by filtering, demodulation, and sink blocks operating at a sample rate of 576 kHz. For reproducibility, the ZMQ source was replaced with a built-in signal generator, the audio sink with a null sink, and GUI visualization blocks were removed.

To measure performance under real-time scheduling, we extended GNU Radio with API hooks to Linux scheduling policies (FIFO and SCHED_DEADLINE for EDF). On a quad-core Raspberry Pi 5, cores were isolated for GNU Radio execution, and kernel preemption and interrupts were disabled to ensure accurate EDF scheduling. For each batch size, we determined the execution time, period, and relative deadlines of blocks using the methodology described in Section 4, and then measured both utilization and end-to-end latency.

As shown in Fig. 8, intra-block batching reduces total utilization substantially while introducing only modest latency overheads. The results mirror the synthetic microbenchmark trends: a point of diminishing returns is observed around $N = 750$, after which larger batch sizes provide little additional utilization reduction while steadily increasing latency. This demonstrates that intra-block batching is both analytically beneficial and practically effective in deployed GNU Radio applications.

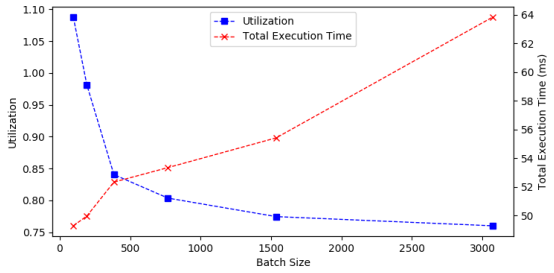

Figure 8. Case study results for a GNU Radio NBFM receiver. Intra-block batching reduces utilization across the flowgraph with only modest increases in latency.

7.3. Inter-block Batching Results

Compared to intra-block batching, our evaluation of inter-block batching is more limited. This is because applying inter-block batching in GNU Radio currently requires manual modification of the C++ block implementations to fuse their execution logic. While this demonstrates feasibility, it is not practical for conducting large-scale, systematic experiments across many flowgraphs. For this reason, we restrict our evaluation to a small set of representative cases.

To illustrate inter-block batching effects, we compare the execution time models of individual blocks with those of their merged counterparts. Recall from the marginal cost model (Sec. 4) that the execution time of a block as a function of processed samples can be well-approximated by a linear regression with an *intercept* (fixed cost, e.g., cache warm-up, context-switch overheads) and a *slope* (marginal cost per sample).

Figure 9 shows representative regression results for two individual blocks and their merged implementation. A key observation is that the intercept of the merged block is significantly smaller than the sum of the intercepts of the two individual blocks, while only slightly larger than the larger of the two. This demonstrates that merging eliminates redundant fixed costs by exploiting cache affinity and avoiding intermediate context switches. In contrast, the slope of the regression (per-sample cost) remains largely unchanged, indicating that inter-block batching does not significantly affect marginal computation cost, but yields improvements primarily by reducing initialization overheads.

This result highlights the practical benefit of inter-block batching: when consecutive blocks operate at the same rate, their consolidation can substantially reduce per-invocation fixed costs, thus lowering overall utilization without introducing additional latency. These findings complement the intra-block batching results, showing that both strategies tackle different components of execution overhead.

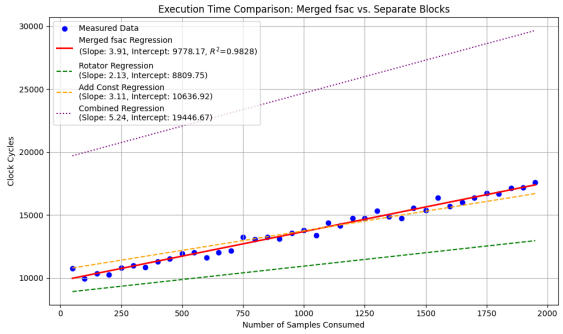

Figure 9. Linear regression results comparing execution time models of two individual blocks and their merged implementation. The merged intercept is much smaller than the sum of the two intercepts, demonstrating reduced overhead from inter-block batching.

8. Discussion

These experiments highlight complementary benefits of intra-block and inter-block batching. Intra-block batching reduces per-invocation startup overheads within blocks, while inter-block batching reduces context-switch and cache-miss overheads across blocks. Together, they provide a practical path toward reducing jitter, improving pre-

dictability, and enabling real-time performance in GNU Radio applications.

The current GNU Radio architecture renders inter-block batching a manual process. These results suggest that decoupling OS threads from individual blocks and allowing multiple blocks to be combined in a single thread may enable more flexible inter-block batching.

Acknowledgments

This work was supported by DARPA's Processor Reconfiguration for Wideband Spectrum Sensing (PROWESS) program under contract HR00112490302. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

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

Devi, Uma Maheswari. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006. URL https://doi.org/10.17615/6y9j-x297.

Duggan, B. Narrowband FM transceiver. https://wiki.gnuradio.org/index.php/Simulation_example:_Narrowband_FM_transceiver, 2024. Accessed: Sep. 4, 2025.

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, 2024. doi: 10.1109/RTSS62706.2024.00039.

Goddard, Steve. *On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems fromProcessing Graphs*. PhD thesis, University of North Carolina at Chapel Hill, 1998. URL https://www.cs.unc.edu/~jeffay/dissertations/goddard-98/goddard-98.pdf.

Lee, Edward Ashford and Messerschmitt, David G. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, 1987. doi: 10.1109/TC.1987.5009446.

Liu, Cong and Anderson, James H. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pp. 3–13, USA, 2010. IEEE Computer Society. ISBN 9780769542980. doi: 10.1109/RTSS.2010.38. URL https://doi.org/10.1109/RTSS.2010.38.