2024 IEEE Real-Time Systems Symposium (RTSS) | 979-8-3315-4026-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/RTSS62706.2024.00039

Job-Level Batching for Software-Defined Radio on Multi-Core

Abigail Eisenklam, Will Hedgecock, Bryan C. Ward
Vanderbilt University
Nashville, TN USA

***Abstract*—Conventional wireless communication is built upon hardware-based signal processing. This enables high performance, but is inflexible as the signal-processing algorithms are "baked in" to the hardware. Software-defined radio (SDR) is an emerging solution in which more of the signal-processing logic is implemented in software instead of hardware. This allows for adaptability to spectrum conditions (e.g., jamming or congestion), changes to protocols, and software updates that improve signal-processing logic – features that are beneficial in many consumer and military applications. However, the high sampling rate (kHz to MHz or faster) of many SDR applications poses significant challenges for real-time scheduling of such workloads. To manage high sampling rates on general-purpose processors, which process samples sequentially instead of in parallel, as can be done using hardware acceleration, samples must be buffered, or "batched" together, to minimize overheads and maximize locality. To address this characteristic of high-frequency signal processing, this paper presents an extension of traditional real-time scheduling models called the *marginal cost model*, which reflects the fact that when batching many samples, the marginal cost of processing additional samples is often much less than the cost of processing the first sample. Empirical evaluations are presented from the open source GNU Radio SDR framework to validate the marginal cost model. Experiments are then presented that demonstrate the trade-offs between batching and worst-case latency for synthetic SDR workloads. Finally, a case study is presented to demonstrate the utility of the presented model and batching techniques in real-world signal-processing applications.**

***Index Terms*—Real-time, scheduling, SDR, signal-processing**

I. Introduction

Software-defined Radio (SDR) is an increasingly important technology in which radio signals are processed in software instead of dedicated harware. SDR therefore offers flexible, adaptable, and upgradeable wireless communication. By updating software or running different algorithms, better performance can be realized, protocols can be updated, and communication can be shifted to different parts of the spectrum. This flexibility is beneficial in many application domains, including cyber-physical systems, IoT devices, cellular networks, and satellite communications. SDR is especially important in many military applications as its enables dynamic, secure, and resilient communication even in the presence of spectrum-contested or jammed environments [1].

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.

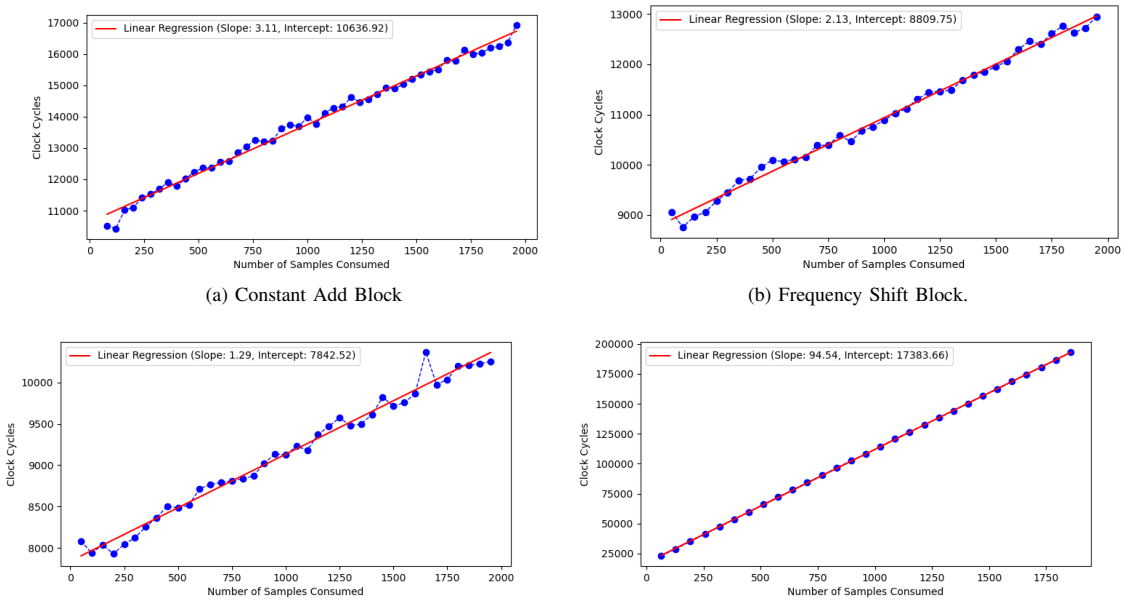
Often in signal processing (hardware- and software-based), applications are modeled using *flowgraphs*. These flowgraphs are composed of *blocks*, that is, computational units that implement various signal-processing algorithms. Data samples pass through these blocks in order to transform an input signal to output. In traditional signal processing, flowgraphs are implemented directly on high-performance, application-specific hardware. Thus, flowgraphs can be executed efficiently and in parallel, but they cannot be updated. In software-based signal processing, however, blocks of computation are executed on the CPU. As a result, flowgraphs can be easily updated, but they suffer from (i) less parallelism and (ii) contention for scheduling time with other tasks. Thus, SDR applications can suffer from significant jitter and end-to-end latency when implemented on general-purpose processors, hampering SDR's ability to be used in many real-time applications [2].

The cause of this latency and jitter can ultimately be traced back to fundamental challenges that arise when scheduling computations with high sampling rates. While there has been considerable work on real-time flowgraph scheduling (see [3] for a recent summary), many of these papers implicitly target applications with much slower frequencies (*e.g.*, video frame rates of 30-120Hz), as is common in control processing in embedded and cyber-physical systems, rather than the high frequencies seen in signal processing (e.g., audio sampled at 44kHz, or Wi-Fi at 2.4GHz). To keep the latency of these high-frequency applications low, samples must be sent through the flowgraph at a high rate and with relatively small execution times, raising challenges in how to manage overheads.

To illustrate these challenges and highlight how existing SDR implementations address them, we focus on the popular open-source project GNU Radio [4]. In GNU Radio, flowgraphs are composed of blocks. Each block implements some elementary signal-processing function (e.g., filter, multiply, etc.) and is assigned its own thread by the GNU Radio runtime environment. These blocks are connected by buffers, or FIFO data queues that move data between consecutive computations. To reduce overheads, GNU Radio generally does not wake up a thread, *i.e.*, make it ready, until its input buffer(s) is at least half full. Upon execution, the block then processes all inputs in its input buffer(s). This approach reduces the total number of block invocations (and thus, context switches) that are required to process a workload. It also promotes cache affinity (both instruction and data), which significantly reduces overhead and observed utilization. However, this approach is

979-8-3315-4026-5/24/$31.00 ©2024 IEEE
DOI 10.1109/RTSS62706.2024.00039

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.



(a) Constant Add Block

(b) Frequency Shift Block.

(c) Constant Multiply Block

(d) Decimating FIR Filter

Fig. 1: Execution Cost vs. Samples Processed for GNU Radio Blocks

unpredictable, as scheduling decisions are left to the general-purpose scheduler of the underlying operating system. Moreover, blocks can process different numbers of samples, or *batch sizes*, in different invocations. This leads to variable execution times for individual blocks, and significant jitter in the overall application [2].

To better understand this relationship between batch size and execution times, we executed several GNU Radio signal-processing blocks on a quad-core Raspberry Pi 5 and controlled for the number of samples processed in a given invocation using GNU Radio's "Head" blocks. We used the Linux FIFO scheduler to ensure non-preemptive execution and CPU affinity masks to isolate the workload to dedicated cores. We then measured the execution times using GNU Radio Performance Counters [5]. Graphs showing the execution time of four GNU Radio blocks as a function of the number of samples processed are shown in Fig. 1. We observe in Fig. 1b that the cost to execute the first sample is approximately $8,800$ clock cycles, while the marginal cost of executing each additional sample is only approximately 2 clock cycles. As seen in Fig. 1, the same trend holds across all the block types that we profiled. This clearly shows why GNU Radio blocks opportunistically process as many samples as possible in a single invocation to save on total execution time.

Real-Time SDR. These results demonstrate that batching is fundamental to supporting high-frequency signal-processing applications. However, to our knowledge, no real-time scheduling models or analyses have considered batching block invocations, or *jobs*, in this way. In fact, there has been significant work on approaches to *splitting* jobs [6]–[8] to reduce tardiness bounds, or breaking jobs up into smaller non-preemptive sections, as in limited-preemptive scheduling [9]. To enable real-time SDR, we argue that signal-processing workloads must be batched more carefully to manage overheads and reduce utilization while still supporting real-time predictability and performance.

In support of this goal, we introduce batching in real-time SDR using the *Processing Graph Method* (*PGM*) scheduling framework originally developed by the U.S. Navy, which models signal-processing applications as directed, acyclic processing graphs [10]. PGM is an intuitive model for GNU Radio-like SDR applications, as the blocks in a GNU Radio flowgraph have a natural mapping to *nodes* in a processing graph. Similarly, the *buffers* that connect these blocks can be mapped to processing graph *edges*. Importantly, prior work has developed optimal schedulers and analyses for both uniprocessor [11] and multiprocessor [12] soft real-time scheduling of PGM graphs, albeit without consideration for the effects of batching. Like this prior work, our focus is on *soft* real-time (SRT) scheduling of processing graphs. This is because many signal-processing applications do not have specific signal-processing deadlines, *per se*. Rather, our soft real-time analysis informs the maximum frequency and throughput a platform can support subject to certain latency or utilization constraints.

Marginal Cost Model. PGM allows us to model the structure and data relationships in signal-processing applications. However, it is insufficient for understanding how batching affects factors such as latency and utilization, as it assumes that node execution times are fixed (and externally provided). Therefore, to formalize the effects of batching, we present a new scheduling model called the *marginal cost model*. In this model, we batch consecutive jobs of a task into a

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

single job by increasing the number of samples that the task consumes per invocation. This causes the batched job to execute less frequently but process more data in a single invocation. By reducing context-switches and promoting cache affinity, these batched jobs require less execution time than the sum of their individual un-batched parts, consistent with our empirical observations (*e.g.*, Fig. 1). Furthermore, batching reduces the amount of synchronization necessary to manage all the dependencies between blocks in the flowgraph. Given this model, we present methods to batch workloads efficiently subject to application and platform requirements.

Contributions:

- We evaluate the marginal cost of batched sample processing in GNU Radio (Figs. 1b and 1).
- We present the first end-to-end latency analysis for multicore PGM graphs scheduled under a global earliest-deadline-first-like (G-EDF-like) scheduler. (§ III)
- We develop the marginal cost model (§ IV) and present two batching techniques to capitalize upon batching analytically. (§ V)
- We present evaluations of randomly generated synthetic task systems to demonstrate the utilization and latency trade-off enabled by batching. (§ VI)
- We extend GNU Radio to support the Linux `SCHED_DEADLINE` and `SCHED_FIFO` scheduling policies, and present a case study applying batching techniques to a real signal-processing application. (§ VII)

II. Preliminaries

Often in real-time systems (and many other high-performance computing environments) complex data-processing applications are modeled as directed acyclic graphs (DAGs), where vertices represent distinct code segments, or tasks, and edges represent the precedence constraints between those tasks. By modeling an application as a task graph, rather than a single sequential program, implementations can exploit parallelism. For example, the widely studied Synchronous Data Flow (SDF) model for digital signal-processing, originally proposed by Lee and Messerschmitt [13], enables both task-level parallelism (parallel execution of the tasks in a graph) and DAG-level parallelism (parallel execution of multiple invocations of that graph). The Processing Graph Method (PGM), which we adopt is an extension of the SDF model, with the additional feature that PGM graphs can specify thresholds for each queue (see § II). Importantly, the streaming-based approach used in both SDF and PGM allows for successive invocations of a flowgraph to execute concurrently, unlike many real-time DAG scheduling models [14]–[18].

In the remainder of this section, we formally define the PGM framework, which was originally developed for signal-processing applications [10], and review important scheduling results that pertain to it. Of particular relevance to this paper are the results of Liu and Anderson [12], who ensured bounded tardiness for PGM graphs scheduled by a global earliest-deadline-first-like (G-EDF-like)[1] scheduler with no utilization loss. Since deadlines are at a constant offset from the release point, bounding deadline tardiness also implies bounded latency. This is an especially apt correctness condition for signal processing, as in many cases there are not strict deadlines *per se*; rather, practitioners are more concerned with the maximum frequency or bandwidth that can be supported given the available resources. Our work on batching also assumes that deadlines are not hard (otherwise batching would inherently violate deadline constraints). Rather, we seek to achieve bounded (and ideally, low) latency given utilization constraints or throughput requirements. Therefore, in the remainder of this paper, we consider soft real-time scheduling with the requirement that tardiness (and hence latency) is bounded.

In order to show that a PGM graph scheduled under a G-EDF-like scheduler has bounded deadline tardiness, Liu and Anderson [12] performed a series of transformations from the PGM graph to a set of sporadic tasks. In turn, seminal results [6], [19], [20] on tardiness bounds for sporadic-tasks scheduled under G-EDF or G-EDF-like schedulers can be immediately applied. Here, we review the transformation of Liu and Anderson [12], which is necessary for proving certain batching properties in § V. Finally, we define metrics that are descriptive of application performance, namely utilization and latency, and describe how they can be inferred from certain properties of the sporadic task system.

Processing graph method. A PGM graph, G, consists of a set of nodes connected by directed edges. Each node represents a distinct signal-processing algorithm, or *task*, and each edge represents a first-in, first-out queue that propagates data between successive tasks. Because PGM graphs are acyclic, every PGM graph has at least one *source node* — that is, a node with no incoming edges — and at least one *sink node*, a node with no outgoing edges. Futhermore, each edge in a PGM graph is specified by three parameters: a produce amount, a threshold, and a consume amount. Suppose there is a directed edge from some node, u, to another node, w. The *produce amount* for this edge specifies the number of data tokens that are output to the queue each time u executes. We use the notation $p_{u\to w}$ to denote the produce amount of the edge from u to w. The *threshold* for this same edge specifies the number of data tokens that must be present in the queue before the successor, w, can execute. Upon execution, w will consume $c_{u\to w}$ data tokens from the queue, where $c_{u\to w}$ denotes the *consume amount* for the edge between u and w. Note that the produce and consume amounts for a particular edge need not be equal. The same is true of the consume amount and the threshold, however if the threshold label is omitted from an edge, we assume they are equal. Fig. 2 shows an example of a PGM graph.

[1]A G-EDF-like scheduler is one with job-level fixed priority where the priority point is some constant offset from the release point. Global EDF is an example – the priority of a job is defined by the constant offset of the relative deadline from the release point. FIFO is another example in which the priority point is defined to be the release point. For simplicity in the remainder of this paper we refer to the priority point as the deadline.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

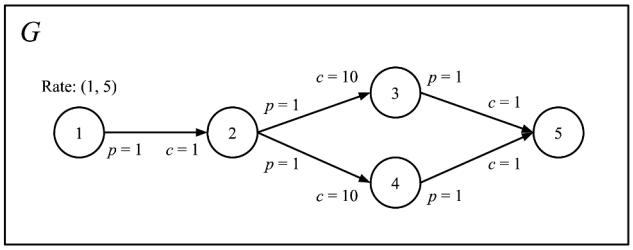


Fig. 2: An example PGM graph, G. The source node (node 1) is specified by Rate = (x, y) and each edge is specified by a produce and consume amount.

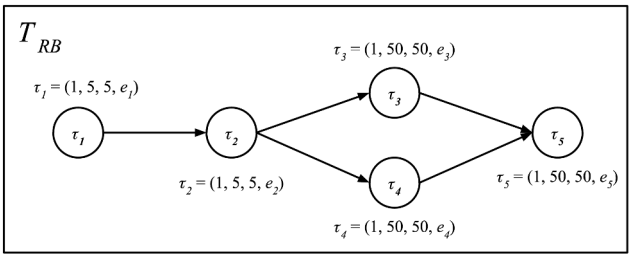


Fig. 3: The RB-parameters, $RB(x, y, d, e)$, for each task in G.

Rate-based task model. In order to convert a PGM graph, G, into a set of schedulable tasks, the analyses presented in [11] and [12] derive *execution rates* for each node in G. The execution rate (x, y) indicates that a task will execute x times in any time interval of length y. Each invocation of a task is called a *job*. This execution rate, along with the task's execution cost, *i.e.* the amount of processor time that the task requires to execute fully, are used to specify that task in a framework called the Rate-Based (RB) Task Model. We use τ to denote an individual task, and T_{RB} to denote a set of RB tasks generated from a PGM graph, G. Then, as proven in [11], the RB-parameters of each non-source task, $\tau \in T_{RB}$, can be calculated recursively using Lem. 1.

Lemma 1. *[11] For any non-source task τ_u in T_{RB}, let V denote the set of predecessor tasks of τ_u. For any τ_v in V, let Rate$_v = (x_v, y_v)$ be a valid execution rate. Then, the execution rate Rate$_u = (x_u, y_u)$ is valid for τ_u if*

$$y_u = lcm\left\{ \frac{c_{v\to u} \cdot y_v}{gcd(p_{v\to u} \cdot x_v, c_{v\to u})} \mid \forall v \in V \right\} \quad (1)$$

$$x_u = y_u \cdot \frac{p_{v\to u}}{c_{v\to u}} \cdot \frac{x_v}{y_v},\ \exists\, v \in V. \quad (2)$$

Similarly to [12], we calculate a relative deadline for each RB task using $d_u = y_u/x_u$, and we require that $d_u \leq d_v$ if there exists an edge from node u to v in G, *i.e.*, that the execution rate does not increase through any path in a PGM graph (as is often the case in signal-processing applications). Furthermore, we assume there is a single source node, whose execution rate is periodic, *i.e.*, $x = 1$, and that this rate can be inferred from the application sampling rate. Lastly, we assume that the worst-case execution cost, e, for each task is provided (in our case, from our GNU Radio block profiling). From here on, we use the notation $\tau_u = (x_u, y_u, d_u, e_u)$ to specify the RB-parameters of some task $\tau_u \in T_{RB}$.

Sporadic task model. Finally, the RB task set can be transformed into a sporadic task set, for which tardiness bounds can be determined analytically. In the 3-parameter sporadic task model, each task is specified by a period, relative deadline, and execution cost, (p, d, e). These parameters are determined in [12] by setting the period, p, of the task equal to y/x, where y and x are the task's RB-parameters, and by assuming implicit deadlines, *i.e.*, that the relative deadline, d, is equal to the period. Thus, for each task τ_v in a sporadic task set T_S, $\tau_v = (p_v, d_v, e_v) = (y_v/x_v, y_v/x_v, e_v)$.

Utilization. After transforming an arbitrary PGM graph, G, into a sporadic task set, T_S, we can calculate the utilization, u_v, of each task, τ_v, via $u_v = e_v/p_v$. The *utilization* of a task describes the fraction of total processor time that must be allocated to that task. By summing the utilizations of each task in a soft real-time system, we can determine whether the task set is schedulable on M processors. In fact, as long as the sum of the utilizations, denoted U, does not exceed M, the task set is guaranteed to be SRT schedulable under any G-EDF-like scheduling policy [19].

Latency. An equivalent schedulability condition to $U \leq M$ checks whether we can bound the *tardiness* of each task in the task set, where tardiness is defined as the amount of time between a task's deadline and its actual completion time. In the context of signal-processing, bounded tardiness indicates that the *latency* of an application does not increase to indefinitely, *i.e.* get progressively worse throughout the program duration.

In this paper, we define the *end-to-end latency* of a data sample as the total time between the production of that sample by a source node and its consumption by a sink node. There are two different types of latency that contribute to the total end-to-end latency that a sample will experience, namely *inherent latency* and *imposed latency*. Inherent latency results from the structure and rate relationships within the PGM graph, while imposed latency results from the specific policy used to schedule the graph. Similarly to [11], we calculate the end-to-end latency, L, that a sample experiences through a PGM graph by summing these two types of latency.

$$L = InherentLatency + ImposedLatency \quad (3)$$

Because inherent latency is "inherent" to the structure of the PGM graph and is independent of the scheduling policy or platform, the formal equation for inherent latency derived in [11] holds. We present this equation in Lem. 2, but first, we define a variable that is needed for its calculation. Suppose we have a PGM graph G. Let j and w be a source node and a sink node in G, respectively. Let $j \rightsquigarrow w$ represent some path from j to w. Then the variable $F_{j\rightsquigarrow w}$ denotes the number of executions of j that are necessary before w has sufficient data to execute for the first time [11].

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

Lemma 2. *[11] Let G be a PGM graph with source node j. Let $\{j \rightsquigarrow w\}$ be the set of paths from j to a sink node w. The inherent latency a sample incurs from j to w is*

$$InherentLatency = (max(\{F_q \mid q \in j \rightsquigarrow w\}) - 1) \cdot p_j \quad (4)$$

where, $max(\{F_q \mid q \in j \rightsquigarrow w\})$ is the maximum value of F over all possible paths from j to w in G, and p_j is the period of the source node.

Note that the equations used to calculate F for a single path, which can be found in Lemma 4.2.1 of [11], are omitted here due to space constraints. The path that maximizes F, is found by enumerating all paths from j to w in G.

Unlike [11], this paper is concerned with non-preemptive (NP) G-EDF-like scheduling in the multiprocessor case. Thus, while we can reuse the above equation for inherent latency, we must derive our own equation for the *imposed* latency of PGM applications under NP G-EDF. To do this, we must first consider that in soft real-time systems with bounded tardiness, tasks may complete later than their deadline. Thus, we must include the potential tardiness of tasks in the upper bound on imposed latency. Note that the time between the release of a soft real-time task and its completion is called its *response time*. Thus, the worst-case response time (WCRT) of a job is equal to its relative deadline plus the upper bound on its tardiness, *i.e.*, $r = d + b$, where r is the response time, d is the task's relative deadline, and b is the tardiness bound. We note that there exist tardiness bounds for G-EDF-like schedulers, both preemptive and non-preemptive [19]. As an example, we review the NP G-EDF tardiness bound:

Lemma 3. *[19] Let T_S be a sporadic task set, U be the sum of the utilization of each task in T_S, and*

$$\Lambda = \begin{cases} U - 1, & \text{if } U \text{ is integral} \\ \lfloor U \rfloor, & \text{otherwise} \end{cases} \quad (5)$$

Then, on M processors, G-NP-EDF ensures a tardiness bound of

$$b_k \leq \frac{\sum_{i=1}^{\Lambda+1} \epsilon_i - e_{\min}}{M - \sum_{i=1}^{\Lambda} \mu_i} + e_k \quad (6)$$

for every task τ_k of a sporadic task system T_S in which $U \leq M$, where $\sum_{i=1}^{\Lambda+1} \epsilon_i$ represents the sum of the $\Lambda+1$ largest execution costs of tasks in T_S and $\sum_{i=1}^{\Lambda} \mu_i$ the Λ largest utilizations.

The transformation from PGM to a sporadic task system [12] defines sporadic release times such that they satisfy the precedence constraints of the PGM graph. Thus, traditional bounds for pure sporadic tasks can be applied directly. Liu and Anderson [12] also show that jobs can further be "early released" when their predecessor jobs complete but before the minimum job separation in the sporadic task model without negatively affecting the tardiness bounds. In the following section we use the tardiness bounds of Lem. 3 to derive a bound on imposed latency, and thus total latency.

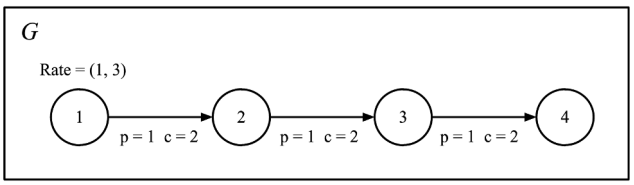


Fig. 4: Single path PGM graph, G.

III. End-To-End Latency

While previous work has shown that G-EDF-like schedulers are able to soft-real-time-optimally schedule a PGM graph on a multiprocessor system [12], to our knowledge there are no known bounds for the *end-to-end latency* that may occur in such systems. Here, we aim to derive such an end-to-end latency bound, but first, we present an example that demonstrates the two types of latency that contribute to it.

As shown in Lem. 2, inherent latency describes the number of source-node invocations that are necessary before there is sufficient data in the system to contribute to an output by a sink node. Take the simple PGM graph in Fig. 4, for example. Task τ_1 must execute 8 times before τ_4 has sufficient data to execute, due to the 2:1 consume-to-produce relationships in G. If the task set generated from G is deemed schedulable, then source node is guaranteed to execute with Rate = (1, 3), that is, once every 3 time units. Thus, it takes exactly $(F_q - 1) \cdot p_j = (8 - 1) \cdot 3 = 21$ time units for sufficient data to enter the system.

Fig. 5 shows the release times (denoted by up arrows) and completion times (denoted by down arrows) for jobs of the corresponding sporadic task set generated from G. Each task's release and completion times are shown on their own timeline for clarity. The 21 time units that can be attributed to inherent latency are labeled at the bottom left side of the timeline. Note that in Fig. 5, we show only the releases of jobs that contribute to the first output of data by τ_4. The dotted lines show how data moves through G via these task executions.

Once sufficient data has entered the system, each task along the path must execute exactly one more time before the first output by τ_4. This is because we guarantee that the execution rate does not increase through any path of G. In the worst case, the time it takes for these jobs (shaded darker blue in Fig. 5) to finish is equal to the sum of their worst-case response times (because jobs cannot be released until the completion of their predecessors). This additional latency represents the imposed latency under G-EDF. Therefore, the total latency for the first sample output by G is upper-bounded by $21 + r_1 + r_2 + r_3 + r_4$. We formalize this worst-case bound on the imposed latency in Lem. 4.

Lemma 4. *Let G be a PGM graph with source node j and sink node w. Let Q be the set of paths from j to w that maximize $F_{j \rightsquigarrow w}$. If the sporadic task set, T_S, generated from G is SRT schedulable with bounded deadline tardiness on M processors, then the imposed latency a sample incurs from j to w is upper*

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

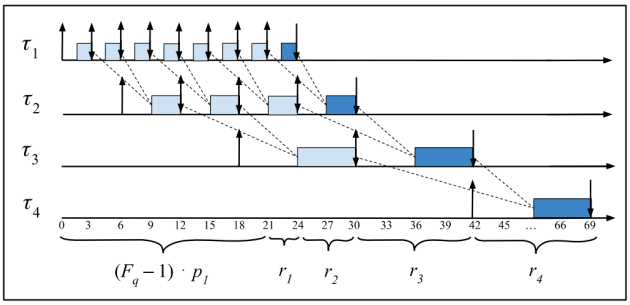


Fig. 5: Worst-case release/completion times for jobs of the sporadic task set generated from G (of Fig. 4) that contribute to the *first* output.

bounded by

$$ImposedLatency \leq \max\big(\{\sum_{i \in q} r_i \mid q \in Q\}\big) \tag{7}$$

i.e., the largest sum of the worst-case response times of the tasks τ_i along any path $q \in Q$.

Proof. Recall that the source node j executes periodically, according to some rate $(1, y)$. Therefore, the last execution of τ_j before τ_w will have enough data to execute for the first time is guaranteed to be released at time $(F_q - 1) \cdot p_j$. Because this is the final execution of τ_j before there is enough data in G for τ_w to execute, the data must pass through each node in q at least one more time before it can be output by w. However, the data will not pass through any node more than once, as we require that the execution rate of nodes does not increase through a path in G. In the worst case, each task invocation along this path takes the maximum amount of time to complete, *i.e.*, the job completes at its release time plus its worst-case response time. Thus, successor jobs cannot be released until the release of their predecessor along q plus the predecessors WCRT time. Therefore, the maximum time it can take for a data sample to get from j to w along q is the sum of the worst-case response times of the tasks along path q. Because there can be multiple paths from j to w that maximize F, we take the maximum sum of these WCRTs over all paths in Q. Thus, the upper bound on imposed latency is $\max\big(\{\sum_{i \in q} r_i \mid q \in Q\}\big)$. □

Now that we have bounds on both inherent latency and imposed latency, we can determine a bound on the total latency of samples processed by a sporadic task set generated from a PGM graph and scheduled under NP G-EDF.

Theorem 1. *Let G be a PGM graph with source node j and sink node w. If the sporadic task set, T_S, generated from G is SRT schedulable with bounded deadline tardiness on M processors, then the total latency, L, that a sample incurs from j to w is upper bounded by*

$$L \leq (\max(\{F_q \mid q \in j \rightsquigarrow w\}) - 1) \cdot p_j + \max\big(\{\sum_{i \in q} r_i \mid q \in Q\}\big). \tag{8}$$

Proof. The proof of Thm. 1 follows from Lem. 2, Lem. 4, and the definition of total latency. □

Latency vs. utilization. Notice that the inherent latency of a PGM graph grows as the period of the source task increases (as does the imposed latency, less obviously). Therefore, if we increase the period of the source task, *i.e.*, send larger batches of data through the graph at increased intervals, the latency of the corresponding application grows. However, larger batch sizes provide the benefit of minimizing context-switch overhead and promoting data locality, factors that, generally speaking, reduce utilization. Often times, on the tightly constrained processing resources of many real-time systems, high utilization from any one application is undesirable. Thus, a practical question becomes, how do we select batch sizes to minimize utilization subject to a particular application's latency constraints? We address this in the following sections.

IV. The Marginal Cost Model

As shown in Fig. 1, there is a fixed cost associated with starting the execution of a block, but the marginal cost of executing successive samples is significantly smaller. It is this observation that motivates the marginal cost model presented in this section. Note that this orders-of-magnitude difference between costs is due to the overhead of loading caches with instructions/data after a context switch, as well as the synchronization that is needed to signal that a block is ready to execute. We use the marginal cost model to account for these overheads (which can be relatively large considering the high-frequency of signal processing) in a task's execution time.

Suppose we have a sporadic task $\tau_v = (p_v, d_v, e_v)$. We use I_v to represent the time spent loading program instructions and/or data onto the CPU, including time spent rebuilding cache affinity, when τ_v begins execution. We call this I_v the *initialization cost* of τ_v. We then use Δ_v to represent the time required for τ_v to process one sample, that is, the *marginal cost* per sample. Finally, let c_v denote the number of samples consumed by a single invocation of τ_v.

Def. 1. Under the marginal cost model, the execution cost, e_v, of τ_v is

$$e_v = I_v + \Delta_v \cdot c_v. \tag{9}$$

where I_v the initialization cost, Δ_v is the marginal cost, and c_v is the number of samples consumed by τ_v.

As seen in Figs. 1b and 1, the relationship between samples processed and total execution time is approximately linear. Thus, we let the y-intercept of the linear-regression model represent I and the slope represent the marginal cost to process an additional sample, Δ. The observation that I is orders of magnitude larger than Δ motivates the batching techniques in the following section.

V. Batching Techniques

In this section, we define two batching techniques called *Uniform* and *Rate-Exploiting Batching* and discuss their effects on utilization and latency. Before introducing these techniques we first define how to batch an *individual* task in a way that

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

preserves the task's consume-to-produce ratio, *i.e.*, preserves the logic of the block's computation.

Suppose we want some task, τ_u, to process N times as many data tokens in a single invocation. Note that from here on we use N to denote the factor by which we are batching. In a PGM graph, this batching can be enforced by multiplying the consume amount of all incoming edges to u, and the produce amount of all outgoing edges from u, by the factor N. Def. 2 formalizes this single-node batching procedure. Note that by batching u, we effectively delay the invocation of u until N times as much input data has accumulated in u's input queues.

Def. 2. Let G be a PGM graph and let u be an arbitrary node of G. To batch node u by some positive integer N, we set

$$c^B_{v\to u} = N \cdot c_{v\to u}, \ \ \forall v \in V \tag{10}$$

where V denotes the set of predecessors to u, and

$$p^B_{u\to w} = N \cdot p_{u\to w}, \ \ \forall w \in W \tag{11}$$

where W denotes the set of successors of u.

Note that the superscript B is used to distinguish the value of parameters post-batching from their non-batched counterparts.

A. Uniform Batching

Consider the scenario wherein a designer wishes to implement some SDR application on a multi-core system, but has insufficient cores or utilization for the computational workload. The designer has no choice but to lessen the computational workload at the expense of latency (assuming sufficient memory for the buffers between nodes). Towards this end, we present Uniform Batching.

Suppose we have a PGM graph, G. To batch G uniformly, we batch each node of G by the same factor N. To make claims about the schedulability of the PGM graph, G^B, that results from Uniform Batching, we must first transform it into a sporadic task set using the series of steps described in § II. Recall that the first step is to transform G^B to an RB task set, T^B_{RB}. In Lem. 5, we define the RB-parameters of each batched task $\tau^B_u \in T^B_{RB}$ in terms of the RB-parameters of the original non-batched task, $\tau_u \in T_{RB}$. Defining each τ^B_u in terms of τ_u allows us to directly compare the two task sets and thus quantify the effects of batching on latency and utilization.

Lemma 5. *Let T_{RB} be the set of RB tasks generated from some PGM graph, G. Then for each $\tau_v \in T_{RB}$ we have $\tau_v = (x_v, y_v, d_v, e_v)$. Suppose we batch each node of G by some positive integer, N, and generate a new RB task set, T^B_{RB}. Then the RB-parameters for each $\tau^B_v \in T^B_{RB}$ are valid if*

$$(x^B_v, y^B_v, d^B_v, e^B_v) = (x_v, Ny_v, Nd_v, e^B_v) \tag{12}$$

Proof. Because G is a DAG, it has a topological ordering. For any two tasks τ_v and τ_u in T_{RB} we say $\tau_v \prec \tau_u$ if node v precedes u in the topological sort of G. We show, by strong induction on the order of a task $\tau_u \in T_{RB}$, that for any $\tau^B_u \in T^B_{RB}$, the parameters in Eq. 12 are valid.

First we rewrite Lem. 1 to describe batched task sets. Recall that Lem. 1 defines all RB-parameters recursively, *i.e.*, the RB-parameters of a task, τ^B_u, are determined using the RB-parameters of tasks in the set of predecessors, V. Therefore, $\forall\ \tau^B_v \in V$, $\tau^B_v = (x^B_v, y^B_v, d^B_v, e^B_v)$ is well-defined. We rewrite Lem. 1 in terms of x^B_v and y^B_v and adjust the consume and produce amounts as described in Def. 2.

$$y^B_u = lcm\left\{ \frac{Nc_{v\to u} \cdot y^B_v}{gcd(Np_{v\to u} \cdot x^B_v, Nc_{v\to u})} \mid \forall v \in V \right\} \tag{13}$$

$$x^B_u = y^B_u \cdot \frac{Np_{v\to u}}{Nc_{v\to u}} \cdot \frac{x^B_v}{y^B_v}, \ \exists\, v \in V. \tag{14}$$

We use Equations (13) and (14) later in the proof.

Base case: τ_u has order 1, *i.e.*, it is the source task. Since τ_u is the source task, there are no tasks that precede it in the topological sort. Therefore, it has no input edges and is specified in both the PGM model and the RB-task model by a rate, (x_u, y_u). Recall that the rate (x_u, y_u) implies that the task executes x_u times over the interval y_u. By Def. 2, if we batch τ_u by N, we must recalculate the produce amounts of each of its output queues by letting $p^B_{u\to w} = Np_{u\to w}, \ \forall w \in W$, where W is the set of successors to u. Assuming the application's sampling frequency remains fixed, a source task must wait for additional data to accumulate if it is to produce additional data. That is, to produce N times as much data in each of the x_u invocations of a task, the interval in which those x_u jobs execute must be lengthened to Ny_u. We let $y^B_u = Ny_u$ and calculate $d^B_u = y^B_u / x^B_u = Ny_u / x_u$. Thus, if τ_u is a source task, $\tau^B_u = (x_u, Ny_u, Nd_u, e^B_u)$ and Eq. 12 holds.

Inductive Step: Assume that Eq. 12 holds for all RB tasks with order between 2 and k. We will show that for any task with order equal to $k+1$, Eq. 12 also holds.

Let τ_u be a task with order equal to $k+1$. Because $k+1 > 2$, τ_u is a non-source task, and thus it has a nonempty set, V, of predecessors. By definition of a topological sort, each of these predecessors have order smaller than $k+1$. Thus, by the inductive hypothesis, together with the proof of the base case, $(x^B_v, y^B_v, d^B_v, e^B_v) = (x_v, Ny_v, Nd_v, e^B_v)$ for all $v \in V$. We substitute x_v for x^B_v and Ny_v for y^B_v in Eq. (13) and Eq. (14) and obtain updated RB parameters:

$$y^B_u = lcm\left\{ \frac{Nc_{v\to u} \cdot Ny_v}{gcd(Np_{v\to u} \cdot x_v, Nc_{v\to u})} \mid \forall v \in V \right\} \tag{15}$$

$$x^B_u = Ny_u \cdot \frac{Np_{v\to u}}{Nc_{v\to u}} \cdot \frac{x_v}{Ny_v}, \ \exists\, v \in V, \tag{16}$$

which can be reduced to

$$y^B_u = N \cdot lcm\left\{ \frac{c_{v\to u} \cdot y_v}{gcd(p_{v\to u} \cdot x_v, c_{v\to u})} \mid \forall v \in V \right\} \tag{17}$$

$$x^B_u = y_u \cdot \frac{p_{v\to u}}{c_{v\to u}} \cdot \frac{x_v}{y_v}, \ \exists\, v \in V. \tag{18}$$

Note that we have isolated the original definitions for y_u and x_u as they appear in Lem. 1 and thus can reduce Eq. (17) and Eq. (18) to

$$y^B_u = Ny_u \tag{19}$$

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

$$x_u^B = x_u \tag{20}$$

Thus, Eq. 12 holds for all $\tau^B \in T_{RB}^B$. □

To quantify the effect of Uniform Batching on utilization, we apply the marginal cost model. Note that we have not yet defined e^B, that is, the execution cost of a task that has been batched by some factor, N. By Eq. 9, we can calculate e_u and e_u^B for some arbitrary task τ_u using

$$e_u = I_u + \Delta_u \cdot c_u \tag{21}$$

$$e_u^B = I_u + \Delta_u \cdot Nc_u. \tag{22}$$

Rewriting e_u^B in terms of e_u we get

$$e_u^B = e_u + \Delta_u \cdot (N-1)c_u \tag{23}$$

Thm. 2 combines Lem. 5 and Eq. 23 to formally observe the effect of Uniform Batching on RB task sets.

Theorem 2. *Let T_{RB} be the set of RB tasks generated from some PGM graph, G. Then for each $\tau_i \in T_{RB}$ we have $\tau_i = (x_i, y_i, d_i, e_i)$. Suppose we batch each node of G by some positive integer, N, and generate a new RB task set, T_{RB}^B. Then the RB-parameters for each $\tau_i^B \in T_{RB}^B$ become*

$$(x_i^B, y_i^B, d_i^B, e_i^B) = (x_i, Ny_i, Nd_i, e_i + \Delta_i(N-1)c_i). \tag{24}$$

where Δ_i is the marginal cost of τ_i, and c_i is the number of samples initially consumed by an invocation of τ_i.

Now that we have specified the RB parameters of task sets under Uniform Batching, we can quantify the effects on latency and utilization.

Theorem 3. *Let G be a PGM graph consisting of n nodes and let T_S be a sporadic task set generated from G. Suppose we batch each node of G by some positive integer, N. Then the total utilization of the resulting task set with regard to the original parameters $\tau_i = (p_i, d_i, e_i),\ \forall \tau_i \in T_S$ is*

$$U^B = \sum_{i=1}^{n} \frac{e_i + \Delta_i c_i (N-1)}{Np_i} \tag{25}$$

where Δ_i is the marginal of τ_i, and c_i is the number of samples originally consumed by an invocation of τ_i.

Proof. By Thm. 2, when we batch G by N and generate the corresponding RB task set, we obtain $\tau_i^B = (x_i, Ny_i, Nd_i, e_i + \Delta_i(N-1)c_i)$ for each τ_i^B in T_{RB}^B. Therefore, when we transform T_{RB}^B into a sporadic task set, we get $\tau_i^B = (Np_i, Nd_i, e_i + \Delta_i(N-1)c_i)$ for all τ_i^B in T_S^B. Thus, by the definition of total utilization of a sporadic task set, Eq. 25 holds. □

Recall that $e_i = I_i + \Delta_i \cdot c_i$. Thus, if $I_i > 0, \forall \tau_i \in T_S$, then $\Delta_i c_i < e_i, \forall \tau_i \in T_S$. Cor. 1 follows.

Corollary 1. *If $I_i > 0$ for any $\tau_i \in T_S$, then $U_B < U$. That is, total utilization decreases as a result of Uniform Batching.*

Thm. 3 provides us with a nice theoretical framework, however without concrete values for I or Δ, we cannot quantify the true reduction in utilization. In general, the larger the value of I is compared to Δ, the more reduction we expect to see in utilization under Uniform Batching. However, I and Δ are highly system dependent, and need to be bounded for analysis. Therefore, in § VI we use the values of I and Δ derived in our GNU Radio batch profiling experiments to evaluate the reduction in utilization for synthetic workloads under Uniform Batching, but first, we analyze the effect of Uniform Batching on latency.

Theorem 4. *Let G be a PGM graph with source node j and sink node w. Let T_S be a sporadic task set generated from G that is SRT schedulable on M processors under NP G-EDF. Suppose we batch each node of G by some positive integer, N. Then the total latency a sample incurs from j to w, with regard to the original parameters $\tau_i = (p_i, d_i, e_i),\ \forall \tau_i \in T_S$, is upper bounded by*

$$L^B \leq (\max(\{F_q \mid q \in j \rightsquigarrow w\}) - 1) \cdot Np_j + \max\big(\{\sum_{i \in q}(Nd_i + b_i^B) \mid q \in Q\}\big). \tag{26}$$

Proof. Under Uniform Batching, the produce and consume relationships between tasks scale uniformly. Thus F_q is unaffected by Uniform Batching. As shown in the proof of Thm. 3, when we batch G by N and generate the corresponding sporadic task set, we get $\tau_i^B = (Np_i, Nd_i, e_i + \Delta_i(N-1)c_i)$ for all τ_i^B in T_S^B. Thus we simply replace p_j with Np_j and $r_i = d_i + b_i$ with $Nd_i + b_i^B$ for all $i \in q$. □

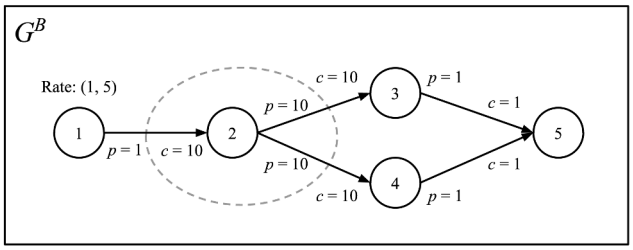
Note that we do not expand b_i^B, as the tardiness bound for sporadic tasks are not simple function of the task's parameters, and instead depend on the specifications of the underlying system. We therefore empirically evaluate tardiness and latency in § VI. However, note that if the sum of the tardiness bounds is small relative to the end-to-end latency bound, then the latency bound scales approximately linearly with the batch size N. We leave the evaluation of this observation to be empirically measured in § VI. Next, we present our second batching technique.

B. Rate-Exploiting Batching

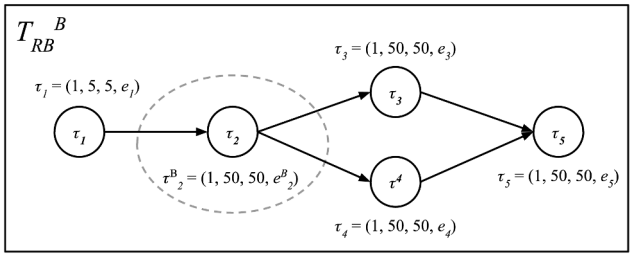
Rate-Exploiting Batching, as the name suggests, exploits disparate executions rates that arise in RB task sets to reduce the utilization of a processing-graph workload. Unlike Uniform Batching, however, Rate-Exploiting Batching can achieve this reduction with minimal impact on latency.

Often in signal-processing applications, certain algorithms must consume more data than they produce. Take filters, for example. A node that implements a filtering algorithm may need to consume 10 data tokens to produce a single output. See nodes 3 and 4 from Fig. 2, for example. Note, however, that the predecessor to these nodes (node 2) produces only one data token each time it executes, and thus will need to execute 10 times before nodes 3 and 4 have sufficient data to execute, with each invocation incurring its own initialization cost. The idea behind Rate-Exploiting Batching is as follows. Rather than invoking this predecessor node 10 individual times, we

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.



(a) PGM graph, G^B, obtained by batching τ_2 in G (see Fig. 2) by $N = 10$.



(b) RB task set T^B_{RB} calculated from G^B.

Fig. 6: The resulting PGM graph and RB-task set after batching τ_2 in Fig. 2 by $N = 10$.

can delay the task invocation until all 10 data tokens can be processed at once, effectively batching all 10 jobs.

In Alg. 1 we show how to detect these disparate produce and consume amounts in a PGM graph, G, and generate the resulting graph, G^B. Note that Alg. 1 ensures that a node is only batched when all of its outgoing edges have the same ratio, c/p, and that c/p is an integer larger than one. When this condition is satisfied, we say the task is *eligible* to be batched, and we set the batch size, N, equal to c/p. Note that this definition prohibits sink nodes from being eligible for Rate-Exploiting Batching. Also note that Alg. 1 detects these rates in reverse-topological order, allowing Rate-Exploiting Batching to be applied recursively up each path in G. As seen in Alg. 1, Rate-Exploiting Batching can be applied to any number of nodes in a PGM graph depending on its structure and data relationships. Thus, we leave the evaluation of Rate-Exploiting Batching to the empirical measurements in § VI.

Next, we demonstrate the trade-offs between utilization and latency that is enforced by batching using randomly generated synthetic workloads.

VI. Evaluation

In this section, we evaluate the effects of Uniform and Rate-Exploiting Batching on the utilization and latency of randomly generated signal-processing applications sampling at 1 MHz.

Experiment Setup. For each combination of parameter values in Tbl. Ia we generated 1,000 PGM workloads. Depending on whether the workload was "Light" or "Heavy" we selected the number of nodes and the branching degree of those nodes from distinct distributions (see Tbl. Ib). We set the rate of the source node to match a 1 MHz sampling rate and randomly generated produce and consume values such that each node had an equal probability to be a one-to-one function (*e.g.*, constant multiply, frequency shift, etc.) or a decimating function (*e.g.*, filter). Depending on the function type, we randomly selected Δ and δ from the distributions shown in Tbl. Ic. The values of Δ and δ are based on the empirical results in § IV. We then transformed the PGM graphs to sporadic task sets and calculated the total utilization and end-to-end latency bound when scheduled under non-preemptive G-EDF.[2]

Algorithm 1: Rate-Exploiting Batching

Input: Set V of nodes in G sorted in reverse-topological order
Output: G^B

```
foreach v ∈ V do
    E_O ← set of outgoing edges from v;
    if |E_O| > 0 then
        ratios(|E_O|);
        i ← 0;
        foreach e ∈ E_O do
            if c_e/p_e is integer then
                ratios[i] ← c_e/p_e;
            else
                ratios[i] ← 1;
            i ← i + 1;
        N ← gcd(ratios);
        if n > 1 then
            E_I ← set of incoming edges to v;
            foreach e ∈ E_I do
                c_e ← N c_e;
            foreach e ∈ E_O do
                p_e ← N p_e;
return G^B;
```

Batch Size	[1, 2, 3, …, 24, 25]
Graph Size	[Light, Heavy]

(a) Parameters

Graph Size	Light	Heavy
Num. Nodes	[5, 6, …, 15]	[15, 16, …, 25]
Branching Degree	[1, 2, 3]	[1, 2, 3, 4]

(b) Distributions for Graph Size Sub-Parameters

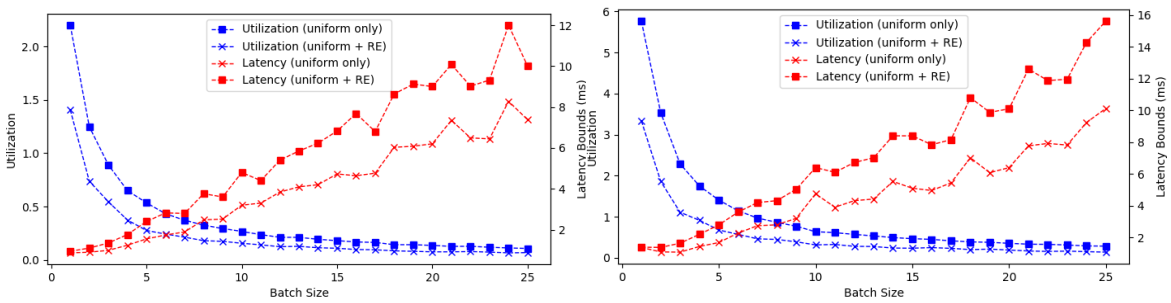
Block Type	Probability	I (μs)	Δ (μs)
1:1	0.5	rand(3, 5)	0.001
1:rand(2, 10)	0.5	rand(6, 8)	0.005

(c) Distribution for I and Δ

TABLE I: Parameter Values for Randomly Generated PGMs

[2]These experiments can be reproduced using the artifact linked from: https://my.vanderbilt.edu/bryancward/publications/.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.



(a) Graph Size: Light

(b) Graph Size: Heavy

Fig. 7: Utilization and Latency vs. Batch Size

Uniform-Batching Results. In Fig. 7, we show how total utilization and end-to-end latency change as a function of the batch size. Each data point represents the average utilization (blue) or end-to-end latency bound (red) across the 1,000 randomly generated applications for the given batch size. Notice in Fig. 7b that a batch size of $N = 4$ causes the average utilization to drop from almost 6 to almost 2. Thus, batching by $N = 4$ allows applications that were previously only guaranteed to be SRT schedulable on six processors, to achieve the same guarantee on only two processors. Notice also that this sharp decrease in the necessary processing resources was achieved without a significant impact on end-to-end latency. In fact, the upper bound on end-to-end latency increased by only 0.06 ms. As the batch size increases, however, the reduction in total utilization shows diminishing returns. However, the upper bound on end-to-end latency continues to increase approximately linearly. From this we make the following observation.

Obs. 1. As batch size increases, the reduction in total system utilization demonstrates diminishing returns, while the upper bound on end-to-end latency increases approximately linearly. The point at which the increase in latency outweighs the reduction in utilization is dependent on the system specifications and application requirements.

Rate-Exploiting-Batching Results. Fig. 7, also compares the utilization and latency of these randomly generated PGM graphs before and after applying Rate-Exploiting Batching (labeled RE). Notice that Rate-Exploiting Batching reduces the total application utilization even further, *i.e.* lowers the utilization curve. Consider the same batch size of $N = 4$ that previously reduced the number of processors required for SRT schedulability guarantees from 6 to 2. By applying Rate-Exploiting Batching, the average utilization dropped below 1, thus achieving SRT schedulability guarantees on only one core. In general, the batch sizes that cause the largest decreases in utilization have the smallest effect on end-to-end latency. For example, in Fig. 7b, the largest reductions in utilization are realized by batch sizes of $N = 2$ to about $N = 6$, as are the smallest increases in end-to-end latency bounds. From this we make the following observation.

Obs. 2. Rate-Exploiting Batching provides additional reductions in utilization, and the greatest relative benefit is achieved when it is applied to workloads with smaller batch sizes.

As described in Obs. 1, the reduction in utilization diminishes for larger batch sizes. Thus, the additional benefits from RE batching are small relative to the consolidated, low-utilization workload.

By modeling data-flow applications using the marginal cost model and applying the batching techniques demonstrated in this section, system designers can measure this point of diminishing return, and make intelligent decisions regarding the trade off between latency and utilization. In fact, by carrying out a similar analysis, designers can minimize demand for processing resources subject to real-time constraints on end-to-end latency. In many systems employing SDR, there may be other applications competing for processor time, and therefore it may be advantageous to further decrease utilization to support those applications, at the expense of slightly higher signal-processing latency, especially if bounds on this latency can be guaranteed.

VII. Case Study

To demonstrate the applicability of our marginal cost model and batching techniques to real-world signal-processing applications we present the following case study. Fig. 8 shows a modified open-source GNU Radio flowgraph that implements a narrowband FM (NBFM) receiver [21]. The flowgraph consists of a source block connected to a chain of processing blocks with varying consume and produce requirements (*e.g.* the FFT filter must consume three samples for every three it produces, whereas Multiply Const is a one-to-one function that controls volume). Note that we replaced the ZMQ Source node from the original application (which receives signals via socket connection) with a default GNU Radio signal source generator operating at the same sample rate of 576kHz. We additionally replaced the original Audio Sink with a Null Sink and removed a GUI signal visualization block.

Background. In order to enable the types of measurements required to carry out this case study, we modified the source code of GNU Radio to allow its internal scheduling algorithms to interact with the built-in real-time scheduling capabilities present in most Linux-based operating systems, namely First-

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

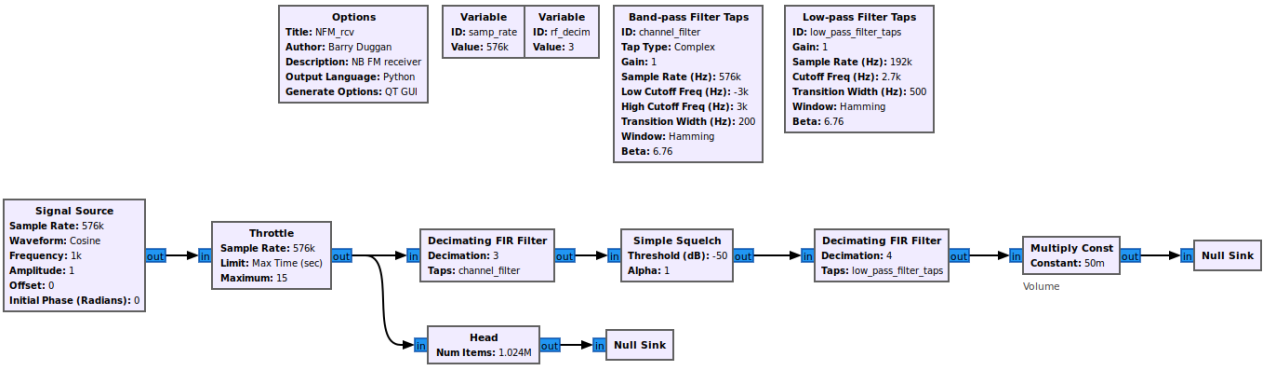


Fig. 8: Modified GNU Radio Narrowband FM Receiver Flowgraph [21].

In First-Out (FIFO) and Earliest Deadline First (EDF) scheduling. To expose this functionality, a new set of API functions were added to GNU Radio to allow the user to 1) select their desired real-time scheduling policy at runtime, and 2) in the case of EDF, specify the deadline, period, and runtime parameters for each EDF-scheduled block within a flowgraph. Internally, this API utilizes the Linux `SCHED_FIFO` policy for FIFO block scheduling and `SCHED_DEADLINE` for Global EDF, as these policies represent well-tested and commonly available backends with drop-in compatibility in the GNU Radio framework. In EDF, for example, each node in a GNU Radio flowgraph contains a single computational task corresponding to a Linux thread that the OS then schedules according to the EDF parameters specified at the time of initialization of the flowgraph.

Experiment Setup. For this experiment, we utilized the scheduling API exposed by our custom GNU Radio version to specify the runtime, deadline, and period parameters for EDF scheduling of each individual block in the NBFM receiver flowgraph. The EDF scheduler is *only* activated for cores 1-3 on a quad-core Raspberry Pi 5, and these cores are completely isolated from the general Linux scheduler and from running any processes or threads other than those belonging to GNU Radio. Additionally, kernel preemption is disabled, and all IRQ requests are excluded from being serviced on these cores, ensuring that once a task is scheduled for execution on a core, it will run to completion without interruption.

Similarly to the experiments presented in § VI, we wanted to plot the average utilization and end-to-end latency for the flowgraph scheduled under a NP G-EDF scheduling policy, and across various batch sizes. This required that we determine the execution time, period, and relative deadline for each block. To determine the periods (and relative deadlines) we first derived an execution rate for the source node from the desired batch size and the 576kHz sample rate. We then used Lem. 1 and the produce and consume relationships between the blocks to calculate the remaining execution rates. To determine the execution time for each block in the flowgraph across the various batch sizes, we then performed identical experiments to those in Fig. 1 to estimate the block's initialization/marginal costs.

Results. Fig. 9 plots utilization, as well as the total execution time required for the flowgraph to process 30,720 samples, as a function of the batch size. Similarly to Fig. 7, we see a point of diminishing returns at a batch size of about 750, which indicates that $N = 750$ is an advantageous choice for balancing the utilization vs. latency trade-off for this workload on this platform. Note, however, that if lower latencies are desirable, smaller batch sizes can enable lower latency. As demonstrated by this case study, the marginal cost model and batching techniques presented in this paper allow system designers to make informed decisions when implementing high-frequency dataflow applications, all while verifying that real-time constraints are met.

VIII. Related Work

Prior work has modeled and analyzed the performance of the Synchronous Dataflow (SDF) model of computation from a real-time perspective [22]–[26]. Although these approaches enable inter-graph parallelism, unlike some more generalized real-time DAG scheduling models [14]–[18], they do not consider batch size as a design parameter, nor do they provide a framework by which to analyze the trade-off between utilization and latency. Also included in this category is Goddard [11], who was the first to derive node execution rates from PGM graphs (an extension of SDF) and bound latency for rate-based task sets on uniprocessors. In [12], Liu and Anderson transform these rate-based task sets to sporadic task sets, and derive tardiness bounds for PGM graphs scheduled on multiprocessor soft real-time systems. Again, neither [11] nor [12] enable selection of batch sizes, nor do they provide an analysis of any design trade-offs. Another body of work has evaluated the trade-off between buffer sizes and throughput for SDF graphs [27], [28], which is useful for managing memory requirements.

Outside of the real-time field, there has been extensive work to improve the performance and predictability of SDR applications on General Purpose Processors. Notable examples of these implementations include Sora [29], which provides

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

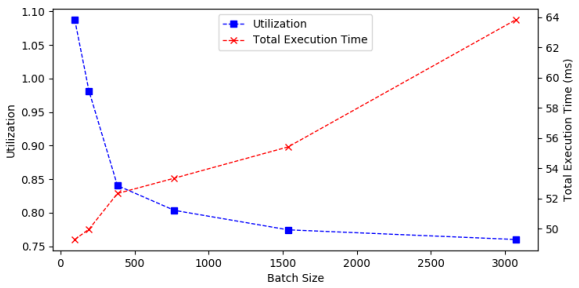


Fig. 9: NBFM Utilization and Execution Time vs. Batch Size

support for real-time wireless tasks by dedicating cores to SDR tasks, and USRP [30], which is commonly used in conjunction with GNU Radio. In fact, the work in [31] uses GNU Radio for real-time wireless signal classification. However, no changes (other than minimizing the number of block threads) are made to the best-effort scheduling approach taken by GNU Radio. More recent work attempts to benchmark, improve, and overhaul the GNU Radio scheduler [2], [32]. In fact, one of the goals of the upcoming GNU Radio 4.0 release is to deliver a new modular scheduler for application-specific development. To our knowledge, however, this paper is the first to enable real-time scheduling support for GNU Radio-like SDR applications on GPPs.

In regards to the marginal cost model, similarities can be drawn between our model and the job-family setup cost problem, in which jobs are assigned to families, and switching between jobs of different families incurs some non-negligible setup cost [33]. In the context of the marginal cost model and batching, one could consider jobs of the same task as belonging to the same family, and that batching jobs minimizes the number of "setup costs," or initialization costs, as we call them. However, the key insight of batching jobs is to minimize initialization costs by consolidating deadlines and adjusting them where possible, whereas under the job-family setup cost model, the goal is to reorder task executions subject to their fixed deadline constraints. Furthermore, there is limited opportunity to reorder jobs of a DAG-based task system, as there are strict precedence constraints between tasks that determine their order of execution, therefore solutions to the job-family setup cost problem do not directly apply.

IX. Discussion

The goal of this paper is to provide a practical framework to evaluate the utilization-latency trade-off that exists in high frequency DAG-based dataflow applications. Because we approach the problem from the real-time perspective, however, we also wish to discuss the optimality and computational complexity of our batching techniques.

Complexity. Both the job-family setup cost problem [33], and the problem of finding the minimum buffer sizes for an SDF graph subject to a throughput requirement [34], are known to be NP-hard. Based on the intractability of these similar problems, we conjecture that selecting the optimal batch sizes for minimizing latency subject to a utilization constraint is also NP-hard, although formalizing this claim is outside the scope of this paper.

Optimality. As shown in [12], G-EDF-like schedulers are able to soft-real-time-optimally schedule a PGM graph on a multiprocessor system. In other words, if total utilization $\leq M$, the graph can be scheduled under G-EDF on M cores with no utilization loss. Therefore, uniform batching upon G-EDF in the marginal cost model, which reduces the total utilization asymptotically as batch size grows, also enables a claim of optimality. In other words, if it is possible to construct a schedule in the marginal cost model that has bounded end-to-end latency, then there exists a uniform batch size that reduces the utilization to $\leq M$, which under G-EDF per [12], implies optimality. (Note that large batch sizes necessary for optimality in some cases may still be impractical based on memory constraints, latency requirements, etc., even if theoretically optimal.) Therefore, choosing batch sizes to achieve a finite bound can be done in polynomial time, but choosing batch sizes to minimize latency subject to a utilization constraint is conjectured to be NP-complete.

X. Conclusion

Software-defined radio (SDR) is an emerging solution to achieve high-performance wireless signal-processing with increased flexibility. However, current approaches to implementing SDR on general-purpose processors suffer from inefficient resource utilization and unpredictable timing behavior. In this paper, we presented a marginal cost model and batching techniques that enable system designers to empirically evaluate the trade-off between utilization and end-to-end latency, and make implementation decisions that are backed by real-time guarantees. We presented evaluations of randomly generated synthetic task sets and showed that our marginal cost model and batching techniques can be applied to make informed implementation decisions for a real signal-processing application. We extended GNU Radio to support real-time scheduling in Linux and presented a case study applying these batching concepts to narrow-band FM receiver.

References

[1] "DARPA PROWESS, "DARPA eyes adaptive real-time processors for future AI-enabled radios"." https://www.darpa.mil/news-events/2022-10-06, Oct. 2022.

[2] B. Bloessl, M. Müller, and M. Hollik, "Benchmarking and profiling the GNU radio scheduler," in *Proceedings of the 9th GNU Radio Conference '19*.

[3] S. Voronov, *Scheduling Real-Time Graph-Based Workloads*. PhD thesis, The University of North Carolina at Chapel Hill, 2023.

[4] "GNU radio." http://www.gnu.org/software/gnuradio/.

[5] T. W. Rondeau, T. O'Shea, and N. Goergen, "Inspecting gnu radio applications with controlport and performance counters," in *Proceedings of the Second Workshop on Software Radio Implementation Forum*, 2013.

[6] J. Erickson, *Managing Tardiness Bounds and Overload in Soft Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill, aug 2014. Ph.D. dissertation.

[7] J. Erickson and J. Anderson, "Reducing tardiness under global scheduling by splitting jobs," in *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pp. 14–24, July 2013.

[8] S. Voronov, S. Tang, T. Amert, and J. H. Anderson, "Ai meets real-time: Addressing real-world complexities in graph response-time analysis," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.

[9] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.
[10] N. R. Laboratory, "Processing graph method specification," tech. rep., Naval Research Laboratory, 1987.
[11] S. Goddard, *On the management of latency in the synthesis of real-time signal processing systems from processing graphs*. PhD thesis, University of North Carolina, Chapel Hill, NC, 1998.
[12] C. Liu and J. Anderson, "Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss," in *RTSS '10*.
[13] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, 1987.
[14] A. Saifullah, D. Ferry, K. Agrawal, C. Lu, and C. Gill, "Real-time scheduling of parallel tasks under a general dag model," 2012.
[15] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," IEEE Computer Society, 2013.
[16] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic dag task systems," 2014.
[17] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," 2013.
[18] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global edf scheduling of directed acyclic graphs on multiprocessor systems," 2013.
[19] U. Devi, *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, NC, 2006.
[20] U. Devi and J. Anderson, "Tardiness bounds under global edf scheduling on a multiprocessor," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005.
[21] B. Duggan, "Narrowband FM transceiver." https://wiki.gnuradio.org/index.php/Simulation_example:_Narrowband_FM_transceiver, 2024.
[22] M. Mohaqeqi, J. Abdullah, and W. Yi, "Modeling and analysis of data flow graphs using the digraph real-time task model," 2016.
[23] J. Khatib, A. Munier-Kordon, E. C. Klikpo, and K. Trabelsi-Colibet, "Computing latency of a real-time system modeled by synchronous dataflow graph," 2016.
[24] E. C. Klikpo and A. Munier-Kordon, "Preemptive scheduling of dependent periodic tasks modeled by synchronous dataflow graphs," 2016.
[25] A. Bouakaz, T. Gautier, and J.-P. Talpin, "Earliest-deadline first scheduling of multiple independent dataflow graphs," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.
[26] A. Singh and S. Baruah, "Global edf-based scheduling of multiple independent synchronous dataflow graphs," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017.
[27] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006.
[28] A. Bouakaz, P. Fradet, and A. Girault, "Symbolic buffer sizing for throughput-optimal scheduling of dataflow graphs," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.
[29] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "Sora: high-performance software radio using general-purpose multi-core processors," *Commun. ACM*, vol. 54, pp. 99–107, 2011.
[30] R. Akeela and B. Dezfouli, "Software-defined radios: Architecture, state-of-the-art, and challenges," *Computer Commun.*, vol. 128, pp. 106–125, 2018.
[31] S. K. K. D. C. Becker, A. Baset and S. Ramirez, "Experiences with using gnu radio for real-time wireless signal classification," in *Proceedings of the 3rd GNU Radio Conference '18*.
[32] J. Morman, M. Lichtman, and M. Müller, "The future of gnu radio: Heterogeneous computing, distributed processing, and scheduler-as-a-plugin," in *MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)*, 2022.
[33] "Scheduling families of jobs with setup times," *International Journal of Production Economics*, 1997.
[34] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software synthesis from dataflow graphs*, vol. 360. Springer Science & Business Media, 1996.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on September 21,2025 at 14:44:49 UTC from IEEE Xplore. Restrictions apply.