A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance

Tobias Blaß[1] Daniel Casini[2] Sergey Bozhko[3] Björn B. Brandenburg[3]
[1]*Robert Bosch GmbH and Saarland University, Saarland Informatics Campus (SIC), Germany*
[2]*TeCIP Institute and Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna Pisa, Italy*
[3]*Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus (SIC), Germany*

Abstract—**Robots are commonly subject to real-time constraints. To ensure that such constraints are met, recent work has analyzed the response times of processing chains under ROS 2, a popular robotics framework. However, prior work supports only scalar worst-case execution time bounds and does not exploit that the ROS 2 scheduling mechanism is starvation-free.**

This paper proposes a novel response-time analysis for ROS 2 processing chains that accounts for both the high execution-time variance typically encountered in robotics workloads and the starvation freedom of the default ROS 2 callback scheduler. Experimental results from both synthetic callback graphs and a real ROS 2 workload empirically show the proposed analysis to be much more accurate (often by a factor of 2× or more).

I. Introduction

Among the frameworks commonly used to program robots, ROS [1] is undoubtedly one of the most widespread: according to the ROS community metrics released in July 2020 [2], it is used by thousands of users, with a 33% increase in users over 2019. Reasons for its popularity include the flexible communication infrastructure, a low barrier to entry, and foremost a vibrant ecosystem, which allows developers to quickly integrate extensively tested components instead of re-implementing common functionality from scratch.

Timing correctness is a central challenge in such systems. In recent years, the real-time community has started to address this issue by developing response-time analyses for ROS 2 systems [3]–[5]. Compared to traditional real-time systems, ROS[1] systems impose two additional challenges. First, ROS systems do not consist of independent tasks, but of a network of interacting callbacks. The analysis has to cope with dependencies among callbacks and must bound the response times of entire *chains* of callbacks. Second, callbacks are multiplexed onto shared *executor threads*, which execute callbacks using a custom scheduling policy that differs significantly from common real-time schedulers. The analysis thus not only has to consider the scheduling of executor threads by the OS, but also callback sequencing *within* executors.

In this paper, we improve upon prior analyses of ROS along three dimensions: a refined model of processor demand, a less pessimistic callback activation model, and an improved response-time analysis that exploits starvation freedom in the ROS executor's callback scheduling algorithm.

The improved processor-demand model addresses the problem that prior analyses support only scalar *worst-case execution times* (WCETs), meaning that the processor demand of any activation of a callback is characterized with a single, often pessimistic bound. In contrast, ROS callbacks often exhibit highly variable execution times [6], which are poorly represented by a single scalar. Such execution times are better described as *execution-time curves*. First introduced by Quinton *et al.* as ET^+ functions [7], execution-time curves are a general way to describe the cumulative processor demand of multiple consecutive instances. In this paper, we model the processor demand of callbacks with execution-time curves. Taking advantage of the richer description, however, poses significant challenges in the analysis (Sec. IV-D).

In an orthogonal direction, we improve the callback activation model to better account for activations among callbacks assigned to the same executor. As a result, we obtain more accurate bounds on intra-executor activations (Sec. V), without imposing additional constraints on the callback graph.

Finally, we develop a new analysis to exploit a key property of the scheduling mechanism in ROS executors, namely that it provides *starvation freedom*: executors execute callbacks in a round-robin fashion, preventing any individual callback from hogging the executor. It has been previously recognized that this makes it difficult, or even impossible, to properly prioritize callbacks [3, 5]. However, this design can also *reduce* worst-case response times by limiting interference from competing callbacks, in particular if competing callbacks are activated in bursts, but prior analyses fail to capture this positive aspect. In this paper, we propose a new response-time analysis that exploits the benefits of the round-robin property (Sec. IV).

We evaluated the proposed analysis on both synthetic callback graphs and a widely used ROS package. The empirical results show that each of the three improvements leads to tighter response-time bounds, and that jointly they yield much more accurate bounds overall. It also shows that the new round-robin analysis and the traditional busy-window approach are most effective under different circumstances, so that a combination of both yields the best response-time bounds (Sec. VI).

II. Background

The ROS framework is composed of a stack of layers, as illustrated in Fig. 1. ROS users and developers interact with the language-specific layers at the top, which provide a language-specific API to define a ROS application. In this work, we

[1]For brevity we omit the version and refer to ROS 2 simply as ROS.

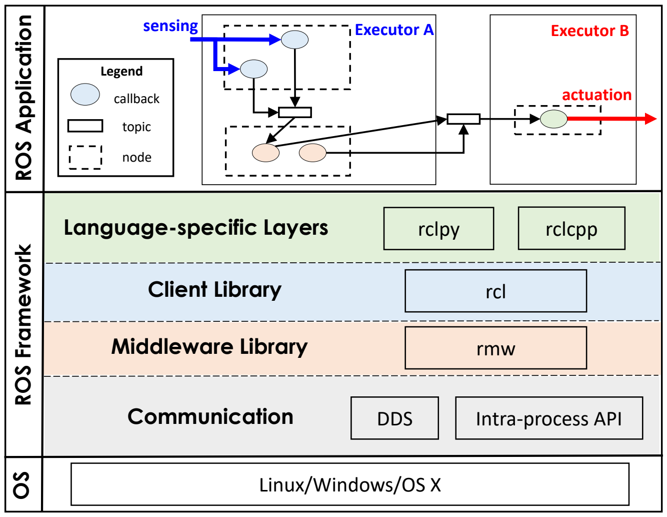


Fig. 1: Layered structure of ROS.

focus on the C++ interface, which we consider the presently most suitable choice for time-sensitive applications.

From a logical point of view, ROS applications are composed of callbacks and topics. Communication follows the *publish-subscribe* paradigm: callbacks publish messages to topics, where each message triggers the callbacks subscribing to the topic. Alternatively, callbacks may also be triggered by timers. The resulting graph of interacting callbacks and topics (illustrated in the upper part of Fig. 1) implements the complex and dynamic behavior of the robot.

ROS additionally provides an asynchronous remote procedure call (RPC) facility via *services* and the associated *service handler* and *client* callbacks. From the perspective of the ROS scheduler, these callbacks behave identically to regular subscription callbacks. We thus do not distinguish between topics and services in this paper; our analysis applies to both equally.

Below the language-specific layer, the *rcl* library implements the core functionalities of ROS. This guarantees consistent behavior across different programming languages and provides a common system model. The *rcl* layer sits on top of the *rmw* layer, which provides a common API to the underlying communication middleware (usually DDS [8]).

The ROS scheduling policy and execution model is implemented by *executors*, which choose the order in which pending callbacks are executed. While ROS supports in principle arbitrary user-defined executors, it provides two built-in implementations, a single- and a multi-threaded one, which are commonly used. This paper focuses exclusively on the single-threaded executor provided by ROS, the default solution.

The executor selects the next callback to run by considering each class of callbacks in order: first timers, then subscriptions, service handlers, and finally clients. Within each class, callbacks are considered in registration order, i.e., the order in which they were registered with the runtime system at process startup. Overall, the combination of callback type and registration time uniquely determines the priority of each callback. Once selected, the chosen callback must run to completion before the executor can select the next callback to run.

One peculiarity of the ROS scheduler is that the list of ready instances is not updated before each scheduling decision. For most classes of callbacks, each callback's readiness status is polled only irregularly from the underlying layers and cached in the *rclcpp* layer. We refer to such callbacks as *polled* callbacks. In contrast, the readiness status of a class of *privileged* callbacks is updated before each scheduling decision.

Whether a callback is polled or privileged is determined by the ROS implementation. In versions up to ROS 2 "*Dashing*," timers are privileged but all other classes are polled [3]. In later versions, timers are polled callbacks as well [9].

The executor runs callbacks in the order described above. Polled callbacks are removed from the cache on completion. The executor polls the list of ready callbacks to refresh the cache only when all callbacks have been removed, i.e., when the cache has become empty, and then starts anew with the processing of cached callbacks. We refer to points in time when the executor's pending callback cache is refreshed as *polling points*, and to the interval between two successive polling points as a *processing window*. An important property of the cache is that it stores at most one ready instance of each callback at a time. The executor thus runs at most one instance of each polled callback per processing window.

It is worth stressing how the ROS callback scheduler differs from common schedulers in the literature. First, it prioritizes callbacks by kind: timers are inherently prioritized over subscriptions, which are prioritized over service handlers, which in turn are prioritized over clients. Second, it does not consider all pending callbacks as eligible for execution. Instead, polled callbacks only become eligible at the first polling point after their activation. Third, the executor executes at most one instance of each polled callback per processing window, irrespective of the number of queued activations.

III. System Model

Following prior work [3], we model the system as a set of callbacks $\mathcal{C}$, each of which activates a potentially infinite sequence of instances. Each callback $c_i \in \mathcal{C}$ is statically assigned to one of k single-threaded executors $E_1, \ldots, E_k$; for notational convenience, we let e_i denote callback c_i's assigned executor. We assume a steady-state system (i.e., callbacks neither leave nor join the system at runtime) and a discrete-time model wherein all time parameters are integer multiples of a basic time unit $\epsilon \triangleq 1$ (e.g., a processor cycle).

Some of the inputs of a ROS system usually come from outside the callback system, for example, devices. This is commonly implemented through external driver threads that convert external stimuli into ROS publications. For consistency, we model such threads as executors with a single pseudo-callback. This callback, which we refer to as an *event source*, represents the computational demand and the communication behavior of the driver thread.

We divide the set of all callbacks by type and let $\mathcal{C}^{\text{tmr}}$, $\mathcal{C}^{\text{evt}}$, $\mathcal{C}^{\text{sub}}$, $\mathcal{C}^{\text{srv}}$, and $\mathcal{C}^{\text{clt}}$ denote the set of all timers, event sources, subscribers, services, and clients, respectively. The last three categories are collectively referred to as the set of *message-driven callbacks* $\mathcal{C}^{\text{msg}} = \mathcal{C}^{\text{sub}} \cup \mathcal{C}^{\text{srv}} \cup \mathcal{C}^{\text{clt}}$. We further let $\mathcal{C}_k$ denote the subset of all callbacks assigned to executor E_k, so that $\mathcal{C}_k^{\text{tmr}}$ denotes the subset of all timers assigned to executor E_k, etc. Finally, $lp_k(c_i)$ and $hp_k(c_i)$ denote all callbacks in $\mathcal{C}_k$ with lower or higher priority than c_i, respectively.

We distinguish between the set of polled callbacks $\mathcal{C}^{\text{pol}}$ and the set of privileged callbacks $\mathcal{C}^{\text{prv}}$, with $\mathcal{C}^{\text{pol}} \cup \mathcal{C}^{\text{prv}} = \mathcal{C}$. In ROS versions up to "*Dashing*," only message-driven callbacks are polled: $\mathcal{C}^{\text{pol}} \triangleq \mathcal{C}^{\text{msg}}$. In later versions, timers are polled, too: $\mathcal{C}^{\text{pol}} \triangleq \mathcal{C}^{\text{msg}} \cup \mathcal{C}^{\text{tmr}}$. Event sources are always privileged.

Each callback c_i is described by an *activation curve* $\eta_i(\Delta)$, which upper-bounds the number of activations within any time window of length Δ [10]–[12] ($\eta_i(\Delta) = 0$ if $\Delta \leq 0$). For timer callbacks and event sources, the activation curve is given as part of the problem specification. For message-driven callbacks, the activation curve is derived by the analysis (Sec. V).

Instance lifecycle. Each callback activates an instance whenever a type-dependent event occurs: an external stimulus for event sources, a new period for timers, or an incoming message for message-driven callbacks. We refer to the k-th instance of a callback c_i as c_i^k. Each callback instance passes through several phases, which are illustrated in Fig. 2.

Conceptually, an instance of a message-driven callback c_i^k is *triggered* when a message is published on the topic to which c_i subscribes. However, the triggering message may incur a *propagation delay* while it traverses the ROS stack and potentially the network. Consequently, the instance c_i^k *activates* only some time later when the triggering message arrives at c_i's assigned executor e_i. We let $\delta_{i,j}$ denote an upper bound on the propagation delay between any two callbacks c_i and c_j, and further assume that there is negligible propagation delay within the same executor, i.e., $e_i = e_j \Rightarrow \delta_{i,j} = 0$.

Once activated, an instance c_i^k is said to be *pending* until it completes. If c_i is a polled callback, then c_i^k resides in a middleware buffer after activation until it is *sampled* by its executor at a future polling point. Once sampled, c_i^k becomes eligible for execution in the subsequent processing window. After all higher-priority sampled instances have executed, the executor then *selects* c_i^k and runs it until c_i^k *completes*. Due to the polling-point semantics, there is at most one sampled instance of a polled callback at a time.

The lifecycle of a polled, message-driven callback instance is illustrated in the top part of Fig. 2, where instance c_2^1 is triggered by a message published to c_2's topic. Once the message arrives at E_1, it activates c_2^1, which is sampled by E_1 after some delay at the next polling point.

A privileged callback's lifecycle is somewhat simpler since it does not involve a separate sampling step. There is hence no difference between the time of activation and the time of sampling. In contrast to polled callbacks, there can be multiple simultaneously sampled instances of a privileged callback.

The bottom part of Fig. 2 illustrates the lifecycle of a privileged timer callback c_4. Executor E_2 immediately samples

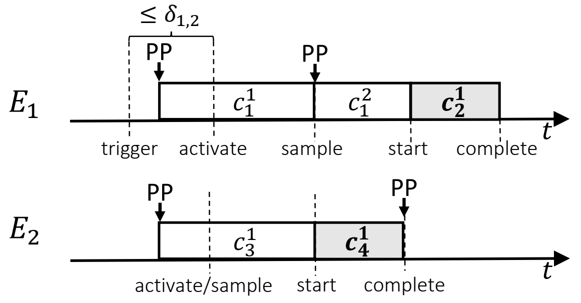


Fig. 2: Lifecycle of a message-driven callback c_2 and a privileged timer c_4

c_4^1 upon activation, and, once the current instance c_3^1 completes, the executor selects and starts c_4^1 without further delay.

Execution-time model. In most of the real-time systems literature (including Casini *et al.* [3]), execution-time requirements are modeled as scalar worst-case execution times (WCETs). While this is both safe and convenient from an analysis perspective, it can be overly pessimistic for many ROS applications. Since callbacks typically execute different code paths depending on the contents of the messages, they can exhibit varying execution-time patterns over time [6].

We therefore model the execution-time needs of each callback c_i as a *cumulative execution-time curve* $ET_i(n)$ [7], which bounds the maximum cumulative execution time of any n consecutive instances of c_i. This model is strictly more expressive: it can represent the scalar WCET model (in which case $ET_i(n)$ is linear), but can also incorporate additional information about series of activations.

Processing chains. The trigger relationships among callbacks form a directed acyclic graph (DAG) $\mathcal{D} = \{\mathcal{C}, \mathcal{E}\}$. An edge $(c_i, c_j) \in \mathcal{E}$ encodes that an instance of c_i may trigger one instance of c_j at some point during its execution. We define the set of predecessors and successors associated with each callback c_i as $pred(c_i) = \{c_j \in \mathcal{C} : \exists\ (c_j, c_i) \in \mathcal{E}\}$ and $succ(c_i) = \{c_j \in \mathcal{C} : \exists\ (c_i, c_j) \in \mathcal{E}\}$, respectively. A callback without predecessors is said to be a *source callback*.

In ROS, a conceptually single piece of functionality is often realized jointly by multiple consecutive callbacks. We refer to such a path through the callback graph as a *callback chain*. Given such a chain $\gamma_i = (c_s, \ldots, c_e)$, a *chain instance* $\gamma_i^k = (c_s^{l_1}, \ldots, c_e^{l_k})$ is a sequence of instances of the comprising callbacks where each instance triggers the next instance in the sequence. The chain instance activates when its first element activates and completes when its last element completes.

Executors. Callbacks are subject to hierarchical scheduling at two levels. The first level of scheduling is the operating system scheduler, which schedules all executor threads as well as any unrelated threads. To support a wide range of systems, we do not assume any particular process scheduling algorithm. Instead, we assume only the existence of a *supply-bound function* $sbf_k(\Delta)$ that lower-bounds the amount of service provided to an executor E_k within any time window of length Δ. Such supply-bound functions are known for many commonly-used OS schedulers [13]–[16].

The second level of scheduling is performed by each executor when it selects the next instance to run. As described in Sec. II, this scheduler has the following properties:

- **SQ** *Sequentiality*: Different instances of the same callback are executed in activation order.
- **NP** *Non-Preemptiveness*: The executor selects a new instance only when the previous instance has completed.
- **WC** *Work-Conservation*: The executor never idles if an instance is pending.
- **PP** *Polling Points*: A polling point occurs when the executor needs to select the next instance to execute but no sampled instance is available.
- **SM** *Sampling*: At a polling point, the executor samples up to one instance of each polled callback in activation order. Instances of privileged callbacks are sampled immediately upon activation.
- **PR** *Selective prioritization*: The executor runs sampled instances (and only sampled instances) in priority order.

While the algorithm shares the first three properties with any standard fixed-priority non-preemptive scheduler, the last three properties represent the unique behavior of the ROS executor.

IV. Round-Robin Analysis

We now present a response-time analysis for processing chains based on the round-robin behavior of the ROS callback scheduler. We begin with some definitions. The *worst-case response time* of a callback c_i or a chain γ_i is the largest possible difference between activation and completion of any instance of c_i (respectively, γ_i). We let $R(c_i)$ (respectively, $R(\gamma_i)$) denote the response-time bound of c_i (respectively, γ_i).

A callback instance c_i^x is pending during an interval $[t_1, t_2)$ if it is pending at any instant $t \in [t_1, t_2)$. c_i^x suffers *interference* from another instance c_j^y at time t if c_j^y occupies the shared executor at t and c_i^x is incomplete at t. Interference is *direct* if c_i^x is pending at time t and *indirect* if c_i^x is not yet pending. As a special case, we call interference by prior instances of c_i *self-interference*. A chain instance suffers interference if any of its callback instances suffers interference.

A. Motivation

Property **SM** of the ROS callback scheduler leads to round-robin-like behavior: no more than one instance per polled callback runs in each processing window, independently of its priority or the number of pending instances.

This scheduling approach ensures a notion of fairness among polled callbacks. Consider two polled callbacks, c_1 and c_2. Assume c_1 is triggered periodically and has two pending instances, whereas c_2 is triggered in infrequent bursts and has ten pending instances. Due to Property **SM**, only two instances of c_2 interfere with c_1's two pending instances. A traditional busy-window analysis (e.g., [3]) would pessimistically account for ten interfering instances instead. The analysis proposed in this section improves upon prior work by bounding the number of processing windows needed to complete a chain instance.

B. Interference Bounds

In the following, we exploit Property **SM** to establish bounds on total interference. As a preliminary, Lemma 1 bounds the number of pending callback instances in arbitrary time intervals.

Lemma 1. *Let c_i be any callback. In any interval of length Δ, at most $\eta_i(\Delta + R(c_i) - \epsilon)$ instances of c_i are pending.*

Proof. Consider an arbitrary interval $[t, t+\Delta)$. If $\Delta = 0$, then the bound holds trivially, so assume $\Delta > 0$. Clearly, callback instances activated at or after time $t+\Delta$ are not pending before $t+\Delta$. By definition of the response-time bound $R(c_i)$, instances of c_i activated at or before $t - R(c_i)$ are complete by time t. Thus, only instances activated in $(t - R(c_i), t+\Delta)$ are pending during $[t, t+\Delta)$. The lemma follows since the length of $(t - R(c_i), t+\Delta)$ is $\Delta + R(c_i) - \epsilon$. ∎

Next, we introduce a bound that exploits Property **SM** to bound the number of callback instances that directly interfere with a callback c_i. The bound depends only on c_i's activation curve and not on the activation curve of the interfering callbacks. In the following lemmas, let N denote a given upper bound on the number of polling points in an arbitrary interval $[t_1, t_2)$. We later show in Lemma 7 how to obtain such a bound N. For brevity, we let $[\![p]\!]_1$ denote the *indicator function* that evaluates to 1 if the predicate p is true and to 0 otherwise.

Lemma 2. *Let $c_i \in \mathcal{C}_k^{\mathrm{pol}}$ and $c_j \in \mathcal{C}_k^{\mathrm{pol}} \setminus \{c_i\}$. Let N be a bound on the number of polling points in E_k during an interval $[t_1, t_2)$. If an instance of c_i completes at time t_2, then at most $N + [\![c_j \in hp_k(c_i)]\!]_1$ instances of c_j run during $[t_1, t_2)$.*

Proof. Consider separately the last polling point before time t_2 (called *last polling point* hereafter), and the up to $N-1$ polling points in $[t_1, t_2)$ that precede the last polling point (called *internal polling points* hereafter). Each of these polling points samples at most one instance of c_j (Property **SM**). In the case of internal polling points, all such instances run before the next polling point and thus before t_2. In the case of the last polling point, the instance of c_i that completes at t_2 is among the sampled instances. During the processing window, the executor E_k runs only callbacks of higher priority than c_i before c_i (Property **SM**). Thus, instances of callbacks in $lp_k(c_i)$ run after t_2, and instances in $hp_k(c_i) \cup c_i$ run before t_2. There may be further instances running in $[t_1, t_2)$ that have been sampled before t_1. These instances are all sampled at the same polling point, namely the last polling point preceding t_1 (Property **PP**). There is thus at most one such instance per callback (Property **SM**). Overall, c_j executes in $[t_1, t_2)$ up to $N-1$ instances sampled at internal polling points, at most 1 instance sampled before t_1, and, if $c_j \in hp_k(c_i)$, at most 1 instance sampled at the last polling point, for a total of at most $N + [\![c_j \in hp_k(c_i)]\!]_1$ instances. ∎

Lemmas 1 and 2 bound the number of callbacks that run during an interval in different ways. The minimum of both bounds is a safe bound, too, as the following corollary notes.

Corollary 1. *Let $c_i \in \mathcal{C}_k^{\mathrm{pol}}$ and $c_j \in \mathcal{C}_k^{\mathrm{pol}} \setminus \{c_i\}$. Let N be an upper bound on the number of polling points in E_k during an interval $[t_1, t_2)$. If an instance of c_i completes at time t_2, then for any $0 \leq \Delta \leq t_2 - t_1$ at most*

$$\min\left(\eta_j(\Delta + R(c_j) - \epsilon), \quad N + [\![c_j \in hp_k(c_i)]\!]_1\right)$$

instances of c_j run during $[t_1, t_1 + \Delta)$.

Thanks to Corollary 1, we can bound the total amount of direct interference a callback c_i suffers from other callbacks.

Definition 1. For any polled callback c_i assigned to E_k, the *direct interference bound function* $I_i(\Delta, N)$ is given by

$$\begin{aligned} I_i(\Delta, N) \triangleq \sum_{c_j \in \mathcal{C}_k^{\mathrm{prv}}} & ET_j(\eta_j(\Delta + R(c_j) - \epsilon)) \\ + \sum_{c_j \in \mathcal{C}_k^{\mathrm{pol}} \setminus \{c_i\}} & ET_j(v_j) \end{aligned}$$

with $v_j = \min\left(\eta_j(\Delta + R(c_j) - \epsilon), N + [\![c_j \in hp_k(c_i)]\!]_1\right)$.

Lemma 3 shows $I_i(\Delta, N)$ to be sound.

Lemma 3. *Let $c_i \in \mathcal{C}_k^{\mathrm{pol}}$. Let N be an upper bound on the number of polling points in E_k during an interval $[t_1, t_2)$. If an instance of c_i completes at time t_2, then for any $0 \leq \Delta \leq t_2 - t_1$, instances of callbacks in $\mathcal{C}_k \setminus c_i$ consume at most $I_i(\Delta, N)$ units of processor service during $[t_1, t_1 + \Delta)$.*

Proof. First note that, w.r.t. each callback, instances running during $[t_1, t_1 + \Delta)$ form a consecutive sequence of instances. Thus, if n instances of an interfering callback c_j run during $[t_1, t_1 + \Delta)$, then their total demand is bounded by $ET_j(n)$. In case of privileged callbacks only pending instances can run. Therefore, Lemma 1 yields a bound on the number of callbacks that may be executed in $[t_1, t_1 + \Delta)$. In case of polled callbacks, Corollary 1 shows that v_j bounds the number of instances that may run in $[t_1, t_1 + \Delta)$. Since each callback in $\mathcal{C}_k$ is either privileged or polled, $I_i(\Delta, N)$ bounds the total demand across all callbacks in $\mathcal{C}_k \setminus \{c_i\}$. ∎

Next, we bound the direct self-interference caused by earlier instances of the callback under analysis.

Definition 2. For any $c_i \in \mathcal{C}^{\mathrm{pol}}$, the *self-interfering instances bound* is given by $si_i(\Delta) \triangleq \max(0, \eta_i(\Delta + R(c_i) - \epsilon) - 1)$.

Lemma 4 shows Definition 2 to be sound.

Lemma 4. *Let c_i be an arbitrary callback and c_i^x one of c_i's instances. During any interval $[t_1, t_1 + \Delta)$ for any $\Delta \geq 0$, at most $si_i(\Delta)$ instances directly self-interfere with c_i^x.*

Proof. If $\Delta = 0$, then $[t_1, t_1 + \Delta)$ is empty and does not contain any self-interfering instance, and hence $si_i(\Delta) \geq 0$ is an upper bound. If $\Delta > 0$, then $\Delta + R(c_i) - \epsilon \geq 0$. By Lemma 1, at most $\eta_i(\Delta + R(c_i) - \epsilon) = si_i(\Delta) + 1$ instances of c_i are pending during any interval of length Δ. To suffer direct self-interference, c_i^x must be pending at some point in $[t_1, t_1 + \Delta)$. By definition of the response-time bound $R(c_i)$, instances released at or prior to $t_1 - R(c_i)$ complete by time t_1; c_i^x is hence activated after time $t_1 - R(c_i)$. Thus, one of the $si_i(\Delta) + 1$ pending instances is c_i^x itself, which implies that at most $si_i(\Delta)$ instances cause direct self-interference. ∎

C. Response-Time Bound

With Definitions 1 and 2 in place, we now bound the response time of any given subchain $\gamma = (c_s, \ldots, c_e)$ assigned to executor E_k ending in a polled callback c_e. Let $\gamma^a = (c_s^x, \ldots, c_e^y)$ be an arbitrary instance of this subchain. Let A denote γ^a's activation time and F its completion time, so that its response time is given by $F - A$.

As a first step towards a response-time bound, we note that throughout $[A, F)$ some callback of γ^a is always pending.

Lemma 5. *At any time in $[A, F)$, at least one of the callback instances comprising $\gamma^a = (c_s^x, \ldots, c_e^y)$ is pending.*

Proof. Since all callbacks in γ^a are assigned to E_k and the intra-executor propagation delay is zero, each callback instance $c_i^q \in \gamma^a \setminus c_e^y$ is still running when its successor $c_j^w \in \gamma^a$ is activated, with $c_j \in succ(c_i)$. Thus, at least one of the callbacks in γ^a is pending throughout $[A, F)$. ∎

Since at least one callback instance in γ^a is pending at every polling point during $[A, F)$, each polling point samples at least one instance of a callback in γ, which implies an upper bound on the number of polling points.

Definition 3. For any callback c_i, its *polling-point bound* $pp(c_i)$ is defined as $pp(c_i) \triangleq \eta_i(R(c_i))$ if $c_i \in \mathcal{C}^{\mathrm{pol}}$, and simply as $pp(c_i) \triangleq 0$ otherwise. For a subchain γ, the aggregate bound $pp(\gamma)$ is defined as $pp(\gamma) \triangleq \sum_{c_i \in \gamma} pp(c_i)$.

Lemmas 6 and 7 prove the correctness of these bounds.

Lemma 6. *Let c_i^x be an arbitrary callback instance. Let t_a denote c_i^x's activation time and t_f denote its completion time. There are at most $pp(c_i)$ polling points in $[t_a, t_f)$.*

Proof. If c_i is not a polled callback, c_i^x is sampled immediately upon activation. Since a polling point occurs only if there are no sampled instances (Property **PP**), there can be no polling point in $[t_a, t_f)$. If c_i is a polled callback each polling point in $[t_a, t_f)$ samples one instance of c_i since at least one instance of c_i, namely c_i^x, is pending during the entire interval $[t_a, t_f)$. The number of polling points in $[t_a, t_f)$ is therefore bounded by the number of instances of c_i that are sampled in $[t_a, t_f)$. The last of these instances is c_i^x (Property **PP**), which is pending at t_a. Due to Property **SQ**, any instance of c_i that is sampled at or after t_a, but before c_i^x is sampled, must be activated before c_i^x, which implies that any such instance is also pending at time t_a. The number of polling points in $[t_a, t_f)$ is thus bounded by the number of instances of c_i pending at time t_a. By Lemma 1 with $\Delta = \epsilon$, at most $\eta_i(R(c_i))$ instances of c_i are pending at time t_a (i.e., during $[t_a, t_a + \epsilon)$). ∎

Lemma 7. *There are at most $pp(\gamma)$ polling points in $[A, F)$.*

Proof. By Lemma 5, at every time in $[A, F)$, at least one callback of γ^a is pending. Every polling point in $[A, F)$ lies thus between activation and completion of at least one of the callbacks of γ^a (Property **SM**). By Lemma 6, $pp(c_i)$ bounds the number of polling points between the activation and the completion of each callback c_i^y of γ^a. The sum of the individual polling-point bounds of the callbacks comprising γ hence yields an upper bound on the total number of polling points between the activation and completion of γ^a. ■

Since γ^a's last callback instance c_e^y completes at time F, this bound fulfills the condition on N and the associated interval $[t_1, t_2)$ in Lemmas 2 and 3 and Corollary 1.

In the last preparatory step, we observe a simple structural property of self-interference.

Lemma 8. *Let $c_i^y, \ldots, c_i^{y+n}$ be $n+1$ consecutive instances of a callback c_i. If the last instance c_i^{y+n} runs for ω_i^{y+n} time units, then the first n instances demand at most $\min\left(ET_i(n+1) - \omega_i^{y+n},\ ET_i(n)\right)$ time units of processor service.*

Proof. As $c_i^y, \ldots, c_i^{y+n}$ is a sequence of $n+1$ consecutive callback instances, their overall execution time is bounded by $ET_i(n+1)$. The last element c_i^{y+n}, by assumption, runs for ω_i^{y+n} time units. The first n elements thus run for at most $ET_i(n+1) - \omega_i^{y+n}$ time units. Similarly, $c_i^y, \ldots, c_i^{y+n-1}$ is a sequence of n consecutive callback instances; their total execution time hence is bounded by $ET_i(n)$. ■

The next two lemmas finally bound the response time $F - A$ by bounding first the *start time* S of the last callback instance, and then its completion time F. The start time is a useful stepping stone because, from this point on, other callbacks can no longer interfere with γ^a (Property **NP**).

Using the established interference bounds, Lemma 9 finds the latest point in time where c_e^y, the last callback instance of γ^a, must consume its first unit of supply, which bounds S.

Lemma 9. *Let γ^a be an arbitrary instance of subchain γ, let A denote γ^a's activation time, let c_e^y denote the last callback instance in γ^a, suppose that c_e^y requires ω_e^y time units of processor service, and let $N \triangleq pp(\gamma)$. If S^* is the least positive solution (if any) of the inequality*

$$sbf_k(S^*) \geq \epsilon + I_e(S^*, N) + \min\left(ET_e(si_e(S^*)+1) - \omega_e^y,\ ET_e(si_e(S^*))\right),$$

then c_e^y starts running in $[A, A+S^*)$.

Proof. By Lemmas 3 and 7, $\epsilon + I_e(S^*, pp(\gamma))$ strictly exceeds the total direct interference due to all callbacks in $\mathcal{C}_k \setminus \{c_e\}$. By Lemma 4, there are at most $si_e(S^*)$ directly self-interfering instances of c_e. Since the self-interfering instances are consecutive, the total interference due to these instances is bounded by Lemma 8 (with $n = si_e(S^*)$). We now show that c_e^y completes in $[A, A+S^*)$. Since A is the activation time of γ^a, by Lemma 5, there is always a pending callback instance of γ^a until c_e^y completes, which implies that the executor does not idle (Property **WC**). Since S^* satisfies the stated inequality, the amount of service supplied by the executor exceeds the total demand by callback instances directly interfering with c_e^y (either due to other callbacks or self-interference) by at least ϵ units of service. It follows that the only instance that can run while c_e^y is pending without interfering with it is c_e^y itself. Therefore, c_e^y starts running in $[A, A+S^*)$. ■

From S^*, we obtain a bound on the response time of γ^a.

Theorem 1. *Let γ^a be an arbitrary instance of subchain γ, let A denote γ^a's activation time, let c_e^y denote the last callback instance in γ^a, and suppose that c_e^y requires ω_e^y time units of processor service. Let S^* be defined as in Lemma 9. If R^* is the least positive solution (if any) of the inequality*

$$sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \omega_e^y,$$

then R^* is a response-time bound for γ (i.e., $F - A \leq R^*$).

Proof. Due to Property **NP**, a callback instance cannot be interfered with once it starts running. Recall from Lemma 9 that c_e^y starts running in $[A, S^*)$, and that it suffers at most $sbf_k(S^*) - \epsilon$ time units of direct interference before it starts running. By assumption, c_e^y runs for ω_e^y time units. Therefore, c_e^y necessarily completes once the executor has provided $sbf_k(S^*) - \epsilon + \omega_e^y$ units of supply. Since R^* satisfies the stated inequality, the executor provides at least $sbf_k(S^*) - \epsilon + \omega_e^y$ units of supply in $[A, A+R^*)$. Consequently, c_e^y completes in $[A, A+R^*)$ and $A + R^* - A = R^*$ is a response-time bound for γ^a. Furthermore, since γ^a is an arbitrary instance of γ upon which we have placed no restrictions, R^* bounds the response time of *any* instance of γ. ■

D. Eliminating ω_e^y

Both S^* and R^* in Theorem 1 depend on ω_e^y, the exact runtime of the last component of the chain under analysis, which is unknown at analysis time. The bound thus cannot be directly applied in a response-time analysis. While ω_e^y can be trivially bounded by 0 from below and by $ET_e(1)$ from above, such an estimate would be needlessly pessimistic. In the following, we refine Lemma 9 and Theorem 1 to be independent of ω_e^y (hereafter simply referred to as ω).

Since the argument does not depend on details of the interference bounds (and will be reused in Sec. V), we consider a more general version of the problem. Let $f, g : \mathbb{N} \to \mathbb{N}$ be any two monotonically increasing functions with $f(t) > 0$ and $g(t) \geq 0$ for all t. For a given $\omega \in \mathbb{N}$ s.th. $0 \leq \omega \leq ET_e(1)$, let $s(\omega)$ denote the least positive $S^* \in \mathbb{N}$ that satisfies

$$sbf_k(S^*) \geq f(S^*) + \min(ET_e(g(S^*)), ET_e(g(S^*)+1) - \omega) \quad (1)$$

and let $r(\omega)$ denote the least positive $R^* \in \mathbb{N}$ that satisfies

$$sbf_k(R^*) \geq sbf_k(s(\omega)) - \epsilon + \omega. \quad (2)$$

We refer to Ineqs. (1) and (2) as the *defining inequality* of $s(\omega)$ and $r(\omega)$, respectively.

In the following, we derive a bound on $\max_{\omega \geq 0} r(\omega)$ for arbitrary f and g that is independent of ω. This bound is then applied to Lemma 9 and Theorem 1, which are a special case of the above system of inequalities (with $f(x) \triangleq \epsilon + I_e(x, N)$

and $g(x) \triangleq si_e(x)$). Our argument makes use of the following trivial property of supply-bound functions.

Property 1. An executor cannot provide more than ϵ units of supply in an interval of length ϵ: $\forall x.\ sbf_k(x) \leq sbf_k(x-\epsilon)+\epsilon$.

To get started, we establish that $s(\omega)$ not only satisfies its defining inequality (Ineq. (1)), but in fact yields an equality. For brevity, let $z(S^*)$ denote the right-hand side of Ineq. (1).

Lemma 10. *If* $0 \leq \omega \leq ET_e(1)$ *and Ineq.* (1) *has a positive solution, then* $sbf_k(s(\omega)) = z(s(\omega))$.

Proof. By contradiction: suppose $sbf_k(s(\omega)) > z(s(\omega))$. By Property 1, $sbf_k(s(\omega)-\epsilon) \geq sbf_k(s(\omega)) - \epsilon > z(s(\omega)) - \epsilon$. Since time is discrete, $sbf_k(s(\omega)-\epsilon) > z(s(\omega)) - \epsilon$ implies $sbf_k(s(\omega)-\epsilon) \geq z(s(\omega))$. $s(\omega)-\epsilon$ is thus a solution of Ineq. (1), too. Since $s(\omega)$ is by definition the least *positive* solution of Ineq. (1), it follows that $s(\omega)-\epsilon = 0$, which implies $z(s(\omega)) = 0$ since $sbf_k(s(\omega)-\epsilon) = sbf_k(0) = 0$. However, as $f(s(\omega)) > 0$, this implies $\min(ET_e(g(s(\omega))), ET_e(g(s(\omega))+1)-\omega) < 0$, which is impossible since $\forall x. ET_e(x) \geq 0$ and $\omega \leq ET_e(1) \leq ET_e(g(s(\omega))+1)$. ■

The next three lemmas characterize $s(\omega)$ by identifying a value ω^m such that s is constant up to ω^m and monotonically decreasing afterwards. Based on ω^m, we then identify the maximum of $r(\omega)$. We begin by establishing monotonicity.

Lemma 11. $s(\omega)$ *is monotonically decreasing.*

Proof. Since $\omega \in \mathbb{N}$, it suffices to establish $s(\omega+\epsilon) \leq s(\omega)$ for any $\omega \geq 0$. To this end, we show that $S^* = s(\omega)$ is a solution to the defining inequality of $s(\omega+\epsilon)$, of which $s(\omega+\epsilon)$ is by definition the least solution.

$$\begin{aligned} &f(S^*) + \min(ET_e(g(S^*)), ET_e(g(S^*)+1) - (\omega+\epsilon)) \\ \leq &f(S^*) + \min(ET_e(g(S^*)), ET_e(g(S^*)+1) - \omega) \\ = &f(s(\omega)) + \min(ET_e(g(s(\omega))), ET_e(g(s(\omega))+1) - \omega) \\ \leq &sbf_k(s(\omega)) = sbf_k(S^*) \qquad \{\text{Def. } s(\omega)\} \end{aligned}$$ ■

Lemma 11 implies that s is maximized at $\omega = 0$. We next observe that there exists a "tipping point" such that the minimum term in Ineq. (1) resolves to its first argument below the tipping point and to its second argument otherwise.

Lemma 12. *There is an* ω^m *such that*

$$\omega \geq \omega^m \Leftrightarrow ET_e(g(s(\omega))) \geq ET_e(g(s(\omega))+1) - \omega.$$

Proof. For brevity, we refer to the right-hand side of the stated equivalence as the ω^m-criterion. There is always at least one value that fulfills the ω^m-criterion, namely $ET_e(1)$, since ET_e is sub-additive ($ET_e(n) \geq ET_e(n+1) - ET_e(1)$ for any n).

Therefore, there also exists a *least* value of ω for which the ω^m-criterion holds. We now show that this least value satisfies the stated equivalence: that is, the value of ω^m is given by the least ω for which the ω^m-criterion holds.

$\Leftarrow$: follows immediately, since by definition ω^m is the least value that statisfies the ω^m-criterion.

$\Rightarrow$: We show that ω *not* fulfilling the ω^m-criterion implies $\omega < \omega^m$. Let ω^* be a value of ω for which $ET_e(g(s(\omega^*))) < ET_e(g(s(\omega^*))+1) - \omega^*$. (If no such ω^* exists, then $\omega^m = 0$ and the claim holds trivially.) Then $s(\omega^*)$ fulfills $s(\omega^m)$'s defining inequality: by definition of ω^* and $s(\omega^*)$, we have $sbf_k(s(\omega^*)) \geq f(s(\omega^*)) + ET_e(g(s(\omega^*))$, and since $\forall x. ET_e(g(s(\omega^*)) \geq \min(ET_e(g(s(\omega^*)), x)$, hence also $sbf_k(s(\omega^*)) \geq f(s(\omega^*)) + \min(ET_e(g(s(\omega^*)), ET_e(g(s(\omega^*))+1)-\omega^m))$, which is $s(\omega^m)$'s defining inequality. Since $s(\omega^*)$ is positive and $s(\omega^m)$ is the least positive solution of ω^m's defining inequality, this implies $s(\omega^*) \geq s(\omega^m)$, which implies $\omega^* \leq \omega^m$ since s is monotonically decreasing (Lemma 11). Further, $\omega^* \neq \omega^m$ since ω^m fulfills the ω^m-criterion while ω^* does not. Therefore, $\omega^* < \omega$. ■

As a result, $s(\omega)$ is a constant function for any $\omega < \omega^m$, which implies that all values in $[0, \omega^m)$ maximize s.

Lemma 13. *If* $0 \leq \omega < \omega^m$, *then* $s(\omega) = s(0)$.

Proof. If $\omega < \omega^m$, then $ET_e(g(s(\omega))+1) - \omega > ET_e(g(s(\omega)))$ by Lemma 12. Therefore, $s(\omega)$ is the least positive solution of the inequality $sbf_k(S^*) \geq f(S^*) + ET_e(g(S^*))$, which obviously does not depend on ω. Since by assumption $\omega^m > 0$ (otherwise $\omega < \omega^m$ does not exist since $0 \leq \omega$), we have $s(\omega) = s(0)$ for $0 \leq \omega < \omega^m$. ■

Based on Lemmas 10 to 13, we now identify the possible maxima of the r function for $0 \leq \omega \leq ET_e(1)$.

Lemma 14. $\max_{0 \leq \omega \leq ET_e(1)} r(\omega) \in \{\omega^m, \omega^m - \epsilon\}$

Proof. We distinguish two cases: $\omega < \omega^m$ and $\omega \geq \omega^m$.
Case 1: If $\omega < \omega^m$, then $s(\omega) = s(0)$ (Lemma 13) and thus $sbf_k(r(\omega)) \geq sbf_k(s(0)) - \epsilon + \omega$. Since sbf_k is monotonically increasing, $r(\omega)$ is hence also monotonically increasing and is thus maximized if ω is maximized, i.e., at $\omega = \omega^m - \epsilon$.
Case 2: If $\omega \geq \omega^m$, then by Lemma 12 the minimum term in Ineq. (1) is equal to $ET_e(g(s(\omega))+1) - \omega$. By Lemma 10, Ineq. (1) is in fact an equality, which allows us to replace $sbf_k(s(\omega))$ with the right-hand side of Ineq. (1). Therefore, $r(\omega)$ is the least positive value satisfying:

$$\begin{aligned} sbf_k(r(\omega)) &\geq sbf_k(s(\omega)) - \epsilon + \omega \\ &= f(s(\omega)) + ET_e(g(s(\omega))+1) - \omega - \epsilon + \omega \\ &= f(s(\omega)) + ET_e(g(s(\omega))+1) - \epsilon. \end{aligned}$$

$r(\omega)$ is thus monotonically increasing in $s(\omega)$ (as sbf_k, f, g, and ET_e are all monotonically increasing). Since $s(\omega)$ is monotonically *decreasing* (Lemma 11), $s(\omega)$ is maximized if ω is minimized, i.e., at $\omega = \omega^m$. ■

Based on the maxima of $r(\omega)$, the next lemma provides two simplified inequalities that no longer depend on ω.

Lemma 15. *If* S^* *is the least positive solution of*

$$sbf_k(S^*) \geq f(S^*) + ET_e(g(S^*)), \tag{3}$$

$\Omega \triangleq ET_e(g(S^*)+1) - ET_e(g(S^*))$, and R^* is the least positive solution of

$$sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \Omega, \tag{4}$$

then $R^* \geq \max_{0 \leq \omega \leq ET_e(1)} r(\omega)$.

Proof. We first show that S^* upper-bounds $s(\omega^m)$ and, if $\omega^m > 0$, $s(\omega^m - \epsilon)$. If $\omega^m = 0$, then by Lemma 12 $ET_e(g(s(\omega^m))) \geq ET_e(g(s(\omega^m)) + 1)$. Since ET_e is monotonically increasing, this implies $ET_e(g(s(\omega^m))) = ET_e(g(s(\omega^m)) + 1)$. As a result, Ineq. (3) is equivalent to the defining inequality of s (Ineq. (1)) for $\omega = \omega^m$ and S^* is therefore equal to $s(\omega^m)$. If $\omega^m > 0$, Ineq. (3) is equivalent to the defining inequality of s (Ineq. (1)) for $\omega = \omega^m - \epsilon$ (Lemma 12), and hence $S^* = s(\omega^m - \epsilon)$. As s is monotonically decreasing (Lemma 11), this implies $S^* \geq s(\omega^m)$.

We now show that $\Omega \geq \omega^m$. If $\omega^m = 0$, then Ω is trivially an upper bound, so assume $\omega^m > 0$ and thus $S^* = s(\omega^m - \epsilon)$. By the definition of ω^m, it then holds that:

$$\begin{aligned}
&ET_e(g(s(\omega^m - \epsilon))) < ET_e(g(s(\omega^m - \epsilon)) + 1) - (\omega^m - \epsilon)\\
&\Leftrightarrow ET_e(g(S^*)) < ET_e(g(S^*) + 1) - (\omega^m - \epsilon)\\
&\Leftrightarrow (\omega^m - \epsilon) < ET_e(g(S^*) + 1) - ET_e(g(S^*))\\
&\Leftrightarrow \omega^m < ET_e(g(S^*) + 1) - ET_e(g(S^*)) + \epsilon\\
&\Leftrightarrow \omega^m < \Omega + \epsilon \Leftrightarrow \omega^m \leq \Omega
\end{aligned}$$

Finally, we establish that R^* upper-bounds both $r(\omega^m)$ and, if $\omega^m > 0$, $r(\omega^m - \epsilon)$ by showing that R^* fulfills r's defining inequality (Ineq. (2)) for $\omega^* \in \{\omega^m, \omega^m - \epsilon\}$.

$$\begin{aligned}
&sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \Omega\\
\Rightarrow &sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \omega^* \qquad \{\Omega \geq \omega^* \geq \omega^* - \epsilon\}\\
\Rightarrow &sbf_k(R^*) \geq sbf_k(s(\omega^*)) - \epsilon + \omega^* \qquad \{S^* \geq s(\omega^*)\}
\end{aligned}$$

Thus, by Lemma 14, $R^* \geq r(\omega)$ for $0 \leq \omega \leq ET_e(1)$. ■

Having solved the general case, we now apply Lemma 15 to the response-time analysis in Lemma 9 and Theorem 1.

Theorem 2. *Let γ^a be an arbitrary subchain instance. Let A be γ^a's activation time, let F be γ^a's completion time, and let c_e^y denote the last callback instance in γ^a. Let S^* be the least positive solution (if any) of the following inequality.*

$$sbf_k(S^*) \geq \epsilon + I_e(S^*, pp(\gamma)) + ET_e(si_e(S^*)) \tag{5}$$

Let $\Omega \triangleq ET_e(si_e(S^*)+1) - ET_e(si_e(S^*))$, and let R^* be the least positive solution (if any) of the following inequality.

$$sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \Omega \tag{6}$$

Then R^* is a response-time bound for γ: $F - A \leq R^*$.

Proof. Follows from Lemma 15 for $f(x) \triangleq \epsilon + I_e(x, N)$ and $g(x) \triangleq si_e(x)$, Lemma 9, and Theorem 1. ■

Since Theorem 2 does not depend on ω_e^y, it is suitable for an *a priori* response-time analysis. Specifically, an implementation can find the least solution S^* through fixed-point iteration and then compute a response-time bound. If a solution S^* cannot be found, then Theorem 2 is not applicable.

Furthermore, since the computation of γ^a's response-time depends on other response-time bounds in a cyclical fashion (e.g., Definitions 1 and 2), another outer fixed-point iteration is necessary [11] until the response-time bounds for all callbacks have reached a global fixed point (or until some predefined threshold is exceeded). This process always terminates since the response-time estimates never decrease during the search.

V. Busy-Window Analysis

We now integrate the busy-window principle with the preceding analysis. A key benefit of extending the analysis horizon in this manner is that it allows for the derivation of less pessimistic callback activation curves.

First, some preliminary definitions: We say that a callback instance is *carried in* at time t if it is pending at both t and $t-1$. An instant t is a *quiet time* of executor E_k if no instances assigned to E_k are carried-in. An interval $[t_1, t_2)$ is a *busy window* w.r.t. a callback instance c_i^x if both t_1 and t_2 are quiet times of c_i^x's executor, no quiet time of c_i^x's executor occurs between t_1 and t_2, and c_i^x is activated in $[t_1, t_2)$.

Since the propagation delay within executors is zero, all components of a subchain instance γ^x share the same busy window. We thus define the busy window of a chain instance γ^x as the busy window of its components.

We again refer to the subchain instance under analysis as γ^a, which is activated at time A and completes at time F. For simplicity (and w.l.o.g.), we assume that the time axis is normalized such that γ^a's busy window starts at time zero.

A. Activation Curves

With basic definitions in place, we now introduce the improved callback activation curves. To begin with, first recall the general case from prior work [3], wherein the activation curve $\eta_j(\Delta)$ for any non-source callback c_j is defined in terms of the activation curves of c_j's predecessors:

$$\eta_j(\Delta) \triangleq \sum_{c_i \in pred(c_j)} \eta_i(\Delta + R(c_i) - \epsilon + \delta_{i,j}). \tag{7}$$

The bound follows since instances activated in $[0, \Delta)$ must be triggered in $[-\delta_{i,j}, \Delta)$. An instance must be pending to trigger a successor callback, and by Lemma 1 at most $\eta_i(\Delta + \delta_{i,j} + R(c_i) - \epsilon)$ instances of c_i are pending in $[-\delta_{i,j}, \Delta)$.

Under the busy-window assumption, the analysis can further exploit that no instance served by γ_a's executor is carried in at time 0. If two callbacks c_i and c_j are assigned to γ_a's executor, then instances of c_i activated before time 0 cannot activate instances of c_j at or after time 0. Definition 4 and Lemma 16 provide an activation curve variant that exploits this property within γ_a's executor and falls back to $\eta_i(\Delta)$ otherwise.

Definition 4. For a message-driven callback c_j assigned to γ_a's executor, the *busy-window activation curve* $\eta^b_j(\Delta)$ is given by

$$\eta^b_j(\Delta) \triangleq \sum_{c_i \in pred(c_j)} \begin{cases} \eta^b_i(\Delta) & e_i = e_j, \\ \eta_i(\Delta + \delta_{i,j} + R(c_i) - \epsilon) & \text{otherwise.} \end{cases}$$

If c_j is not message-driven, then $\eta^b_j(\Delta) \triangleq \eta_j(\Delta)$ instead.

Lemma 16. *Let $\Delta > 0$, and $c_j \in \mathcal{C}_k$. If time 0 is a quiet time of c_j's executor, then at most $\eta^b_j(\Delta)$ instances of c_j are activated during $[0, \Delta)$.*

Proof. By induction over the callback graph.
Induction base: For source callbacks, $\eta^b_j(\Delta) = \eta_j(\Delta)$, which bounds the number of activations of c_j in general.
Induction step: Consider any edge (c_i, c_j). If c_i and c_j do not share an executor, then the number of activations along (c_i, c_j) is bounded by $\eta_j(\Delta + \delta_{i,j} + R(c_i) - \epsilon)$, which bounds the number of activations along an arbitrary edge (Eq. (7)). If c_i and c_j share an executor, then propagation delay is zero and no instance of c_i is carried in at time 0. Thus, all instances of c_i pending in $[0, \Delta)$ are also activated in $[0, \Delta)$. Only instances pending in $[0, \Delta)$ can trigger an instance of c_j during $[0, \Delta)$. By the induction hypothesis, at most $\eta^b_i(\Delta)$ instances of c_i are activated in $[0, \Delta)$. Since each instance triggers at most one instance of c_j, $\eta^b_i(\Delta)$ bounds the activations along the edge (c_i, c_j). The above argument holds for each predecessor of c_j, independently of any other predecessors. We can thus repeat the argument for all predecessors of c_j and bound the total activations of c_j by the sum of all per-predecessor bounds. ■

B. Response-Time Bound

The improved activation curve replaces η_i in the interference bound to reduce pessimism. In addition to Δ, the length of the interval to consider, and N, a bound on the number of polling points, the function takes c^x_i's activation time t_a as a third parameter. It then combines two ways to bound the number of instances of a polled callback c_j during $[0, \Delta)$. The first way is to bound all activations in $[0, \Delta)$ as $\eta^b_j(\Delta)$, which exploits that the considered interval starts at a quiet time but does not exploit Property **SM**. The second way is to bound all activations in $[0, t_a)$ as $\eta^b_j(t_a)$, and to then bound the interference in $[t_a, \Delta)$ (i.e., after c^x_i's activation) under consideration of Property **SM**.

Definition 5. For any polled callback c_i assigned to E_k, the *busy-window interference bound* is given by

$$I^b_i(\Delta, N, t_a) \triangleq \sum_{c_j \in \mathcal{C}^{\text{prv}}_k} ET_j(\eta^b_j(\Delta)) + \sum_{c_j \in \mathcal{C}^{\text{pol}}_k \setminus \{c_i\}} \min \begin{cases} ET_j(\eta^b_j(\Delta)) \\ ET_j(v_j) \end{cases}$$

with $v_j = \eta^b_j(t_a) + N + [\![c_j \in hp_k(c_i)]\!]_1$.

Lemma 17 proves the bound to be sound.

Lemma 17. *Let $c_i \in \mathcal{C}^{\text{pol}}_k$. Let t_a and t_f denote the activation time and completion time of an instance of c_i. Let N upper-bound the number of polling points in E_k during the interval $[t_a, t_f)$. If time 0 is a quiet time of E_k then for any $0 \leq \Delta \leq t_f$, instances of callbacks in $\mathcal{C}_k \setminus \{c_i\}$ consume at most $I^b_i(\Delta, N, t_a)$ units of processor service during $[0, \Delta)$.*

Proof. Since I^b_i sums over all callbacks in $\mathcal{C}_k$, it suffices to show that both $ET_j(\eta^b_j(\Delta))$ and, in the case of polled callbacks, $ET_j(v_j)$ bound how many instances of each interfering callback c_j run during $[0, \Delta)$. By Lemma 16, $\eta^b_j(\Delta)$ bounds how many instances of c_j are activated in the interval $[0, \Delta)$ (since by assumption time 0 is a quiet time of E_k).

It remains to be shown that v_j is a valid upper bound, too. Lemma 16 again shows that $\eta^b_j(t_a)$ bounds how many instances of c_j run during $[0, t_a)$. By Lemma 2 and Corollary 1, $N + [\![c_j \in hp_k(c_i)]\!]_1$ upper-bounds the number of instances running in $[t_a, t_f)$. Their sum therefore bounds the number of instances running in $[0, t_a) \cup [t_a, t_f) = [0, t_f) \supseteq [0, \Delta)$. ■

Analogously to Definition 2, we also define a busy-window-aware self-interference bound $si^b_i(t_a) \triangleq \eta^b_i(t_a + \epsilon) - 1$, using $si^b_i(t_a)$ in place of $si_i(\Delta)$ to leverage the busy-window activation curve $\eta^b_i(\Delta)$. Unlike in Definition 2, no $\max(0, \ldots)$ term is required since $t_a + \epsilon > 0$ and therefore $\eta^b_i(t_a + \epsilon) \geq 1$.

Put together, the two new interference bounds yield a result similar to Theorem 2.

Theorem 3. *Let γ^a be an arbitrary instance of subchain γ. Let A denote γ^a's activation time, F its completion time, and c^y_e the last callback instance in γ^a. If time 0 is a quiet time of γ's executor, S^* is the least positive solution (if any) of*

$$sbf_k(S^*) \geq \epsilon + I^b_e(S^*, pp(\gamma), A) + ET_e(si^b_e(A)),$$

$\Omega \triangleq ET_e(si^b_e(A) + 1) - ET_e(si^b_e(A))$, and F^* is the least positive solution (if any) of the inequality

$$sbf_k(F^*) \geq sbf_k(S^*) - \epsilon + \Omega,$$

then $F \leq F^*$ and $F^* - A$ is a response-time bound for γ.

Proof. Recall there are at most $pp(\gamma)$ polling points in $[A, F)$ (Lemma 7). By Lemma 17, $\epsilon + I^b_e(S^*, pp(\gamma), A)$ hence strictly exceeds the total interference due to all callbacks in $\mathcal{C}_k$ during $[0, S^*)$. Since no instance of c_e is carried in at time 0, $si^b_e(A)$ bounds the number of instances of c_e except c^y_e that are pending (and hence self-interfering) in $[0, A]$. By Lemma 8, the total self-interference is then given by $\min\big(ET_i(si^b_e(A) + 1) - \omega^y_e, ET_i(si^b_e(A))\big)$, where ω^y_e is c^y_e's execution cost. Analogously to the proofs of Lemma 9 and Theorems 1 and 2, the claim then follows via Lemma 15 with $f(x) \triangleq \epsilon + I^b_e(x, pp(\gamma), A)$ and $g(x) \triangleq si^b_e(A)$. ■

Theorem 3 yields a response-time bound, namely $F^* - A$, which however depends on an unknown offset A. We next derive a sparse, finite set of offsets that suffice to be considered.

C. Search Space for the Activation Offset A

To start, Lemma 18 derives an upper bound on A.

Lemma 18. *Let γ be an arbitrary chain, c_e the last callback of γ, and let A^* denote the least positive solution (if any) of*

$$sbf_k(A^*) \geq \epsilon + I_e^b(A^*, pp(\gamma), A^*) + ET_e(\eta_e^b(A^*)).$$

Then any instance of γ is activated strictly less than A^* time units after the beginning of its busy window.[2]

Proof. By contradiction: suppose an instance γ^a is activated at time $t_a \geq A^*$. W.l.o.g. let time 0 denote the start of γ^a's busy window. By Lemma 17, $I_e^b(A^*, pp(\gamma), t_a)$ bounds the total interference that callbacks other than c_e impose upon γ^a during the time window $[0, A^*)$. $ET_e(\eta_e^b(A^*))$ bounds the demand that c_e imposes upon γ^a during $[0, A^*)$. By the inequality in the lemma statement, the supply available to callbacks served by E_k during $[0, A^*)$ then necessarily exceeds the possible total demand during that time. Since γ^a is not pending before $t_a \geq A^*$, this implies that the executor must idle at some point during $[0, A^*)$, i.e., there is a quiet time in $(0, A^*)$. Since γ^a's busy window starts at time 0, this implies that γ^a's busy window ends before γ^a's activation, which is a contradiction. Therefore, any instance γ^a activates at most $A^* - \epsilon$ time units after the beginning of its busy window. ∎

To obtain a sparse search space, the next lemma identifies that only "steps" in the η_j^b bounds for the callbacks in E_k need to be considered to find a response-time bound.

Lemma 19. *Let γ^x be an arbitrary instance of γ, c_e be the last callback in γ, and A denote γ^x's activation time. If $A > 0$,*

$$\forall c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_e\} \;.\; \eta_j^b(A) = \eta_j^b(A - \epsilon), \text{ and}$$

$$\eta_e^b(A) = \eta_e^b(A + \epsilon),$$

then the response-time bound (given by Theorem 3) for γ^x is lower than that for an instance activated at time $A - \epsilon$.

Proof. Let γ^w be an instance of γ activated at time $A - \epsilon$. We show that $I_j^b(\Delta, N, A) = I_j^b(\Delta, N, A - \epsilon)$ and $si_e^b(A) = si_e^b(A - \epsilon)$. Hence F^* as defined in Theorem 3 is the same for γ^w and γ^x. The lemma follows since $F^* - A < F^* - (A - \epsilon)$.

In the definition of $I_e^b(\Delta, N, t_a)$ only the term $ET_j(v_j)$ depends on the t_a parameter. This term appears for any $c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_e\}$. Since for each such c_j by assumption $\eta_j^b(A) = \eta_j^b(A - \epsilon)$, it follows that v_j is equal for $t_a = A$ and $t_a = A - \epsilon$. Therefore, $I_j^b(\Delta, N, A) = I_j^b(\Delta, N, A - \epsilon)$.

In the definition of $si_e^b(t_a)$, only the term $\eta_e^b(t_a + \epsilon)$ depends on t_a. Since $\eta_e^b(A + \epsilon) = \eta_e^b(A)$ it follows that $si_e^b(A) = si_e^b(A - \epsilon)$ ∎

The search space for activation offsets is thus defined as

$$\mathcal{A} \triangleq \{0\} \cup \{A \leq A^* \mid \eta_e^b(A) \neq \eta_e^b(A + \epsilon) \vee \exists c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_e\} : \eta_j^b(A) \neq \eta_j^b(A - \epsilon)\}.$$

[2]Addendum: this document has been revised to correct an accidental misuse of notation in Lemma 18 (confusion of $\eta_e^b(A^*)$ and $si_i^b(t_a)$).

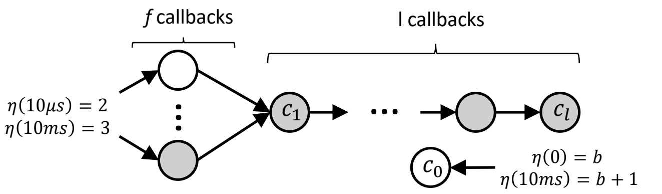


Fig. 3: Synthetic setup. The chain under analysis is shaded gray.

The analysis needs to consider only offsets in $\mathcal{A}$. The response-time bound for the subchain under analysis is then given by the maximum result obtained via Theorem 3 for each $A \in \mathcal{A}$. If A^*, or either of the fixed-point solutions F^* and S^* for any A, cannot be found, then Theorem 3 is not applicable.

D. Combined Analysis

Theorems 2 and 3 are independent analyses that should be used jointly: as neither dominates the other, it is generally advisable to apply both analyses to each subchain and to use the lesser of the two bounds on a per-subchain basis.

In the derivations of Theorems 2 and 3, no assumptions have been placed on the number of callbacks in the chain under analysis γ. Both analyses can therefore also be used to bound the response time of an individual polled callback by interpreting the callback under analysis as a single-element chain.

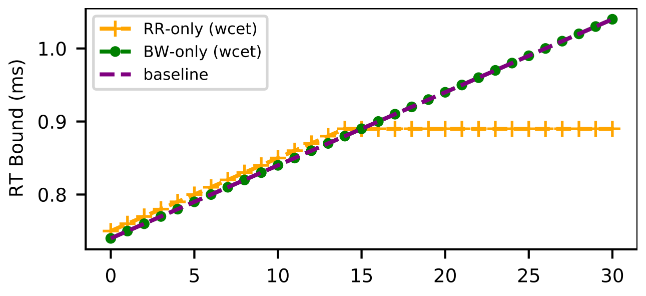
VI. Evaluation

We evaluated the proposed analyses on two case studies: a synthetic callback graph, designed to assess each analysis's advantages and disadvantages, and a real-world callback graph, to evaluate the analyses under realistic conditions.

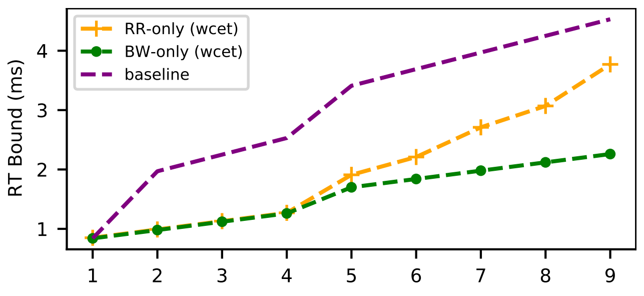
The analysis by Casini *et al.* [3] serves as a baseline. We compare it to the round-robin analysis in Theorem 2 (*RR-only*), the busy-window analysis in Theorem 3 (*BW-only*), and the combined analysis that computes the minimum of both approaches (*this-paper*).

To distinguish effects due to analysis improvements from effects due to the refined execution-time model, we also compare against variants of our analysis using only linear execution-time curves (i.e., $ET(n) \triangleq n \cdot ET(1)$ for $n \geq 1$). The resulting execution-time curves are equivalent to assuming a scalar WCET of $ET(1)$ for each callback instance. Consequently, this analysis variant removes the precision advantage of execution-time curves while keeping analysis improvements in place. In the figures, these variants are marked with the suffix "(wcet)."

Synthetic workload. The first case study, depicted in Fig. 3, consists of a single executor containing only polled callbacks. One callback, c_0, is triggered in bursts of up to b activations at once. The bursts are separated by at least 10 ms. A second set of callbacks, c_1 to c_l, forms an intra-executor chain of length $l = 6$. Callback c_1 is activated by f callbacks (the *fan-in*) located in the same executor, each with an activation curve that mandates a 10 µs distance between any two activations, but 10 ms between any three activations. To simplify comparisons with the baseline, which supports only scalar WCETs, the setup



(a) By burst length of c_0



(b) By width of the fan-in

Fig. 4: Response-time bound of the chain (synthetic workload).

is limited to scalar WCETs. The callback c_0 has a WCET of 10 µs, each callback in the chain has a WCET of 50 µs, and the fan-in callbacks have a WCET of 1 µs each. The executor is provisioned with a periodic supply of 700 µs every one ms.

The experiment varies two parameters: b, the burst length of c_0, and f, the width of the fan-in. As the parameters vary, we observe the response-time bound of the chain marked in Fig. 3.

Fig. 4a shows the response-time bounds reported by the three analyses as b changes (for a fixed fan-in width $f = 1$). Since *this-paper* is simply the minimum of *RR-only* and *BW-only*, it is not shown separately. The plot shows a steadily increasing response-time bound for both the baseline and *BW-only*. In contrast, the *RR-only* bound stays flat after $b = 14$. At this point, all processing windows involved in the chain are saturated with instances of c_0. Activating more instances of c_0 therefore does not increase the interference suffered by the chain. The results show the importance of accounting for starvation freedom, as both the baseline and the *BW-only* analysis significantly overestimate the interference from c_0.

Fig. 4b shows a second experiment in which we varied f, the width of the fan-in, for a fixed burst length of $b = 10$. The plot shows that all variants achieve similar bounds at $f = 1$. As the fan-in width increases, the *BW-only* response-time bound also increases in reaction to the increased activation rate of the chain. The *RR-only* analysis starts out similarly, but exhibits a more rapid increase starting at $f = 5$. The *baseline* analysis performs much worse, with particularly large jumps at $f = 2$ and $f = 5$. Overall, the *BW-only* bound improves upon the baseline by a factor of at least $2\times$ for $f > 1$.

The baseline's lower analysis precision is caused by its lack of support for intra-executor fan-in. Since c_1 has multiple predecessors for $f > 1$, the baseline analysis needs to separately bound the response time of the first fan-in callback and the following l-element chain, oblivious to the fact that they necessarily share a busy window. The *BW-only* analysis, in contrast, analyzes the entire chain as a whole and correctly accounts for busy-window constraints in all predecessors.

Overall, the two experiments show that both analysis approaches, *RR-only* (Theorem 2) and *BW-only* (Theorem 3), improve upon the baseline. Since they excel in different situations, combining both approaches is advisable.

Real workload. In the second case study, we evaluated the analyses on a callback graph extracted from a real ROS 2 system using the Navigation 2 package [17]. The model stems from prior work [6], where it was obtained by measuring all model-relevant parameters (such as execution-time curves and activation frequencies) at runtime on a *Turtlebot 3* robot performing a simple navigation task. The robot ran ROS 2 version "Dashing," so timers were privileged.

Fig. 5 shows the results for the 19 (out of 54) callbacks in the system for which the response-time bounds are not trivial (e.g., due to lack of interference or extremely sparse activations). Since the absolute values of the bounds vary heavily between the callbacks, we instead show each bound as a ratio normalized by the *this-paper* bound. All ratios exceed 1, which shows that the analysis proposed in this paper produces lower response-time bounds than the baseline and all three depicted variants of the proposed analysis. For some callbacks (e.g., *7469/scan*), the baseline yields response-time bounds that are almost 80 times larger than the bounds produced by our analysis. The baseline's bound is still about 25 times larger than the bounds given by versions of the proposed analyses using linear execution-time bounds (i.e., with the "(wcet)" suffix), which shows that the bulk of the improvement in this case stems from analysis improvements, specifically the round-robin analysis. Similar gains are visible for, e.g., *7533/scan* and *7556/tf-static*.

The *BW-only* analysis also achieves significant improvements over the baseline in some of the callbacks (e.g., *7469/scan* and *7533/tf-static*). For about half the callbacks, the *BW-only* analysis does not improve upon the baseline, though. The reason is that the considered benchmark contains only few intra-executor edges, which limits the impact of the improved activation-curve propagation highlighted in Fig. 4b.

The results further show that the busy-window analyses (*baseline* and *BW-only*) are not dominated by the round-robin analysis: for instance, the *RR-only* bounds are significantly worse for the *7469/tf* callback. Again, this shows that it is best to combine both analyses instead of using either in isolation.

Finally, the experiment clearly shows the benefits of using a processor demand model that is more expressive than scalar WCETs. Consider callback *7469/tf* again: the response-time bounds of the WCET-based version of *this-paper* differ from the execution-time curve version by a factor of $60\times$. Assuming that each callback instance runs for its full WCET obviously

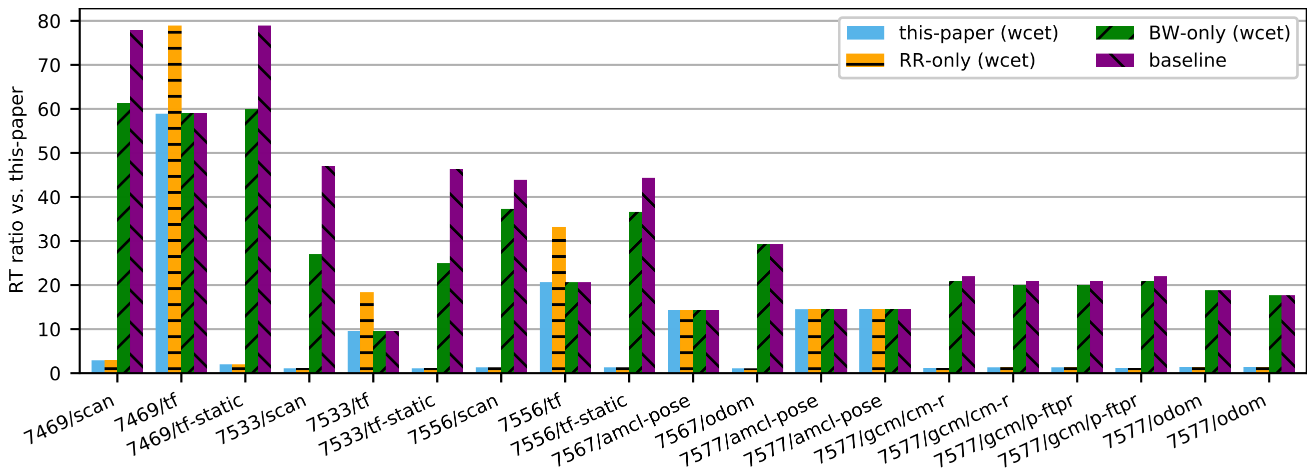


Fig. 5: Response-time bound of callbacks compared to the proposed analysis (lower is better).

leads to tremendous pessimism in the analysis of this callback.

Overall, the experiments confirm that both the round-robin analysis (Sec. IV) and the busy-window analysis (Sec. V) are needed, as both perform better than the other in some cases. The proposed analysis performs significantly better than the baseline, reducing response-time bounds by almost up to $80\times$ in the real-world case study.

VII. Related Work

Casini *et al.* proposed the first response-time analysis of ROS 2 processing chains [3]. Tang *et al.* [4] subsequently provided a more precise analysis for the special case of independent linear processing chains, at the cost of more limited applicability (i.e., each callback can belong to only one chain). Consequently, Tang *et al.*'s analysis [4] is not applicable to the workloads considered in Sec. VI, which contain branching processing chains.

In more distantly related work, Tang *et al.* [18] recently developed an analysis for a ROS-inspired scheduler that also exhibits a round-robin-like property. As their analysis considers only systems of independent tasks, it does not transfer easily to the kind of general ROS systems considered herein. Furthermore, prior timing analyses for other graph-based frameworks like OpenMP [19, 20] and Tensorflow [21] share similar goals and some of the challenges studied herein.

Besides providing analytical guarantees, researchers have explored real-time requirements in ROS 2 in a number of other ways, including empirical studies on latency effects [22]–[26]. In another line of work, Choi *et al.* [5] and Staschulat *et al.* [27] have explored alternative ROS executor designs with improved time-predictability. In contrast, our focus is mainline ROS as it is deployed by most ROS users today. Also targeting unmodified mainline ROS deployments, Blass *et al.* [6] proposed an automated, introspection-based method for provisioning ROS executors under reservation-based scheduling.

This paper adopts the execution-time curve model [7] to represent processor demand. While this is a natural choice in our context due to their similarity with activation curves, the literature provides various alternative processor demand models of varying complexity and expressiveness [28]–[30].

VIII. Conclusion

We have proposed a more precise response-time analysis for ROS systems. The analysis improves upon prior work through three key techniques: modeling processor demand as execution-time curves instead of scalar WCETs, accounting for the effects of quiet times and busy windows in the activation curve derivation, and exploiting the round-robin behavior of the ROS callback scheduler. The resulting analysis was demonstrated in both a synthetic and a real-world case study to yield significantly tighter response-time bounds compared to prior work on the ROS default scheduler.

In future work, it would be interesting to revisit how recent alternative executor designs [5, 27] compare to the default executor once the round-robin property is taken into account.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement no. 803111). This work has been partially supported by the EU H2020 project AMPERE under the grant agreement no. 871669.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[2] T. Foote, "ROS Community Metrics Report 2020," http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf, 2020.

[3] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[4] Y. Tang, F. Zhiwei, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors," *Proceedings of the 41st IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[5] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[6] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[7] S. Quinton, M. Hanke, and R. Ernst, "Formal Analysis of Sporadic Overload in Real-Time Systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012.

[8] Object Management Group, *Data Distribution Service (DDS)*, 2015.

[9] T. Blass, "Real-time Execution Management in the ROS 2 Framework," Ph.D. dissertation, to appear, 2022.

[10] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, 2001.

[11] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," *IEE Proceedings - Computers and Digital Techniques*, 2005.

[12] L. Thiele, S. Chakraborty, and M. Naedele, "Real-Time Calculus for Scheduling Hard Real-Time Systems," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2000.

[13] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, 2011.

[14] D. Casini, L. Abeni, A. Biondi, T. Cucinotta, and G. Buttazzo, "Constant Bandwidth Servers with Constrained Deadlines," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*, 2017.

[15] G. Lipari and E. Bini, "Resource Partitioning among Real-Time Applications," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[16] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[17] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The Marathon 2: A Navigation System," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[18] Y. Tang, N. Guan, Z. Feng, X. Jiang, and W. Yi, "Response Time Analysis of Lazy Round Robin," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[19] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing Characterization of OpenMP4 Tasking Model," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[20] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks," in *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[21] D. Casini, A. Biondi, and G. Buttazzo, "Analyzing Parallel Real-Time Tasks Implemented with Thread Pools," in *Proceedings of the 56th Annual Design Automation Conference (DAC)*, 2019.

[22] J. Park, R. Delgado, and B. W. Choi, "Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study," *IEEE Access*, 2020.

[23] T. Kronauer, J. Pohlmann, M. Matthe, T. Smejkal, and G. Fettweis, "Latency Overhead of ROS2 for Modular Time-Critical Systems," http://arxiv.org/abs/2101.02074, Tech. Rep., 2021.

[24] Y. Yang and T. Azumi, "Exploring Real-Time Executor on ROS 2," in *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2020.

[25] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS 2," in *Proceedings of the 13th International Conference on Embedded Software (EMSOFT)*, 2016.

[26] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a Distributed and Real-Time Framework for Robots: Evaluation of ROS 2.0 Communications for Real-Time Robotic Applications," http://arxiv.org/abs/1809.02595, Tech. Rep., 2018.

[27] J. Staschulat, I. Lütkebohle, and R. Lange, "The rclc Executor: Domain-Specific Deterministic Scheduling Mechanisms for ROS Applications on Microcontrollers: Work-in-progress," in *Proceedings of the 17th International Conference on Embedded Software (EMSOFT)*, 2020.

[28] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The Digraph Real-Time Task Model," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[29] A. K. Mok and D. Chen, "A Multiframe Model for Real-Time Tasks," *IEEE Transactions on Software Engineering*, 1997.

[30] S. K. Baruah, "A General Model for Recurring Real-Time Tasks," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998.