Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors

Hoora Sobhani* Hyunjong Choi† Hyoseung Kim*
*University of California, Riverside
†San Diego State University
hsobh002@ucr.edu, hyunjong.choi@sdsu.edu, hyoseung@ucr.edu

***Abstract*—The second generation of Robotic Operating System, ROS 2, has gained much attention for its potential to be used for safety-critical robotic applications. The need to provide a solid foundation for timing correctness and scheduling mechanisms is therefore growing rapidly. Although there are some pioneering studies conducted on formally analyzing the response time of processing chains in ROS 2, the focus has been limited to single-threaded executors, and multi-threaded executors, despite their advantages, have not been studied well. To fill this knowledge gap, in this paper, we propose a comprehensive response-time analysis framework for chains running on ROS 2 multi-threaded executors. We first analyze the timing behavior of the default scheduling scheme in ROS 2 multi-threaded executors, and then present priority-driven scheduling enhancements to address the limitations of the default scheme. Our framework can analyze chains with both arbitrary and constrained deadlines and also the effect of mutually-exclusive callback groups. Evaluation is conducted by a case study on NVIDIA Jetson AGX Xavier and schedulability experiments using randomly-generated chains. The results demonstrate that our analysis framework can safely upper-bound response times under various conditions and the priority-driven scheduling enhancements not only reduce the response time of critical chains but also improve analytical bounds.**

I. Introduction

The Robotic Operating System (ROS) is an open-source middleware framework that has been widely used for robotic systems in academia and industry. The software modularity and composability of ROS have helped the community achieve efficient and productive robotic software developments. However, the architecture limitations and several deep-rooted shortcomings of ROS had been unveiled over the decades, resulting in the development of its second generation, ROS 2, which is a complete refactoring of the previous version.

One of the major considerations in ROS 2 has been improving real-time capabilities while inheriting the successful concepts of its predecessor. As an example, to support real-time data distribution, ROS 2 employs the Data Distribution Service (DDS) as the underlying communication framework. Although ROS 2 has been shown to provide better real-time support for robotic systems, it is yet incomplete to be applicable to hard real-time or safety-critical applications. To guarantee stringent timing constraints in these applications, designers need to safely upper-bound the end-to-end latency (i.e., response time) of *processing chains*. Although there are many prior studies on the response-time analysis of chains, the unique scheduling behavior of ROS 2 calls for new formal modeling and analysis of its timing abstractions and scheduling architecture.

The pioneers in formally analyzing the response time of chains on ROS 2 are [1–3]. As mentioned in these studies, ROS 2 introduces "executors" as the abstraction of operating system (OS) processes, providing two built-in types: *single-threaded* and *multi-threaded*. A single-threaded executor executes callbacks sequentially, while a multi-threaded executor distributes pending callbacks across multiple threads (i.e., callbacks can execute in parallel). These studies focus on the response-time analysis of callbacks and chains *only* on single-threaded executors. Specifically, [1] and [2] mapped a single-threaded executor to a single reservation server to derive analysis; [3] proposed priority-driven scheduling and executor-to-core allocation but for single-threaded executors.

As of yet, the scheduling behavior of ROS 2 multi-threaded executors has not been studied well. However, plenty of studies in the real-time systems area have demonstrated that multi-threading improves system concurrency and throughput by effectively utilizing multiple processors while preserving timing correctness, e.g., real-time multi-threading in self-driving cars [4]. Therefore, in this paper, we aim to analyze and improve the timing behavior of ROS 2 multi-threaded executors. But, the tremendous amount of non-determinism in multi-threaded executors, such as an unpredictable distribution of callbacks across threads and unsynchronized polling points of threads, makes the analysis particularly challenging. In addition, the lack of systematic support for chain priority in ROS 2 prevents the effective utilization of parallel resources, resulting in delayed processing of critical chains.

This paper tackles the aforementioned issues. We first present a response-time analysis (RTA) framework for chains running on ROS 2 multi-threaded executors. To improve the end-to-end response time of critical chains, we also propose priority-driven scheduling enhancements that make the executor strictly respect the priority of the corresponding chain when scheduling individual callbacks. These enhancements bring significant benefits in timing analysis as well as observed performance on a real platform. The detailed contributions of our work are as follows:

- We discuss difficulties in analyzing the timing behavior of chains on multi-threaded executors (Sec. IV). In particular, we redefine the properties of two ROS-specific scheduling behaviors, polling points and processing windows, and ex-

plain why the latest single-threaded analysis based on them is not applicable to multi-threaded executors.
- We develop an RTA framework for ROS 2 multi-threaded executors (Sec. V). Our analysis considers chains with both constrained and arbitrary deadlines, and upper-bounds the response time of chains executed by multi-thread executors. We also analyze the effects of *callback groups* that are used to control the concurrency of select callbacks.
- We propose priority-driven scheduling enhancements and the corresponding analytical extensions to our RTA framework (Sec. V). The priority-driven scheduling approach mitigates the aforementioned non-determinism issues and the resulting analytical pessimism and helps reduce the response time of critical chains.
- For evaluation, we performed a case study movitated by autonomous driving software on an embedded platform as well as schedulability experiments using randomly-generated workloads (Sec. VI). The results support the effectiveness of our RTA framework and demonstrate how priority-driven scheduling enhancements improve both observed and computer upper-bounds on the response time of chains

II. Related Work

Many studies have been conducted on improving real-time capabilities [5, 6] and evaluating the empirical real-time performance of ROS [7, 8]. In [6], Wei et al. proposed to run two OSes on the same platform, i.e., real-time ROS nodes on Nuttx (RTOS) and non-real-time ones on Linux, to provide isolated execution environments. Saito et al. [5] developed ROSCH-G, which is a real-time extension to ROS with a CPU/GPU coordination mechanism provided as a loadable kernel module. Carlos et al. [7] measured the worst-case latency between two nodes and observed deadline miss behavior in a Linux system with the PREEMPT-RT patch. In [8], the effects of various QoS configurations under different vendor-specific DDS implementations were evaluated empirically. However, some of these studies were conducted on the first generation of ROS [5, 6] and the others did not consider formal modeling or analysis of ROS 2 [7, 8].

Formal timing analysis of end-to-end latency has recently received much attention for processing chains that follow either the publisher-subscriber or read-execute-write model. Davare et al. [9] and Schlatow et al. [10] captured an upper bound on the end-to-end latency of a chain based on the worst-case response time of individual tasks. In [11–13], the authors proposed analytical methods to bound the end-to-end latency of a chain for fixed-priority scheduling. Choi et al. [14] focused on improving the end-to-end latency of chains and proposed chain-based fixed-priority scheduling. However, these approaches cannot be directly applied to ROS 2 due to discrepancies in the scheduling model.

The literature on the response-time analysis of processing chains on ROS 2 executors is quite limited. The first work that formally analyzed the timing behavior of ROS 2 executors is the work by Casini et al. [1]. They unveiled the details of the ROS 2 callback scheduling policy implemented in single-threaded executors and presented the response-time analysis of callbacks and processing chains on single-threaded executors. Tang et al. [2] proposed an improved response-time analysis by characterizing the details of processing windows (we will revisit this in Sec. IV), and studied the effect of callback priorities under the standard ROS 2 scheduling policy that results in round-robin-like behavior [15]. In [3], Choi et al. developed a priority-driven chain-aware scheduler, called PiCAS, which modifies the default ROS 2 policy to strictly follow assigned chain priorities. They also presented callback priority assignment, allocation of nodes to executors and executors to CPU cores, and response-time analysis under their scheduler. Unlike PiCAS which uses fixed-priority scheduling, Arafat et al. [16] proposed a dynamic-priority scheduling scheme to improve chain latency, especially in an overloaded scenario. Teper at el. [17] focused on cause-effect chains where intermediate callbacks can be released independently by their own timers, and proposed an end-to-end latency analysis of cause-effect chains in ROS 2. However, all of these focus on single-threaded executors, none on multi-threaded executors.

Both single-threaded and multi-threaded ROS 2 executors have yet another important feature called *callback groups* that have not been considered in the prior analysis work [1–3, 16–18]. While the authors of [19] explored the effect of callback groups in single-threaded Micro-ROS executors (a variant of ROS 2 for microcontrollers), they did not link this to formal timing analysis. In this paper, we take this into account in our RTA framework.

Recently, Jiang et al. [20] presented response-time analysis for processing chains with constrained deadlines running on a ROS 2 multi-threaded executor. While the reader might find it similar to our work, we make unique contributions in that our work analyzes chains with both constrained and arbitrary deadlines and provides priority-driven scheduling enhancements and the corresponding analytical extensions to ROS 2 multi-threaded executors.

III. Background and System Model

In this section, we briefly review the ROS 2 architecture and introduce our system model.

A. ROS 2 Architecture

Fig. 1 illustrates the ROS 2 middleware architecture that sits on top of the OS and provides a set of libraries and tools for robotic application development. In particular, the ROS 2 client library (rcl) consists of language-specific libraries (rclcpp and rclpy for C++ and Python, respectively), and the middleware library (rmw) provides publisher-subscriber interfaces for Data Distribution Service (DDS) and intra-process communication.

ROS 2 applications consist of *nodes*, each of which in turn consists of a set of *callbacks*. ROS 2 callbacks are categorized into four types: timer callbacks are triggered when the timer period is up; subscription callbacks are triggered when a subscribed message arrives; service and client callbacks are

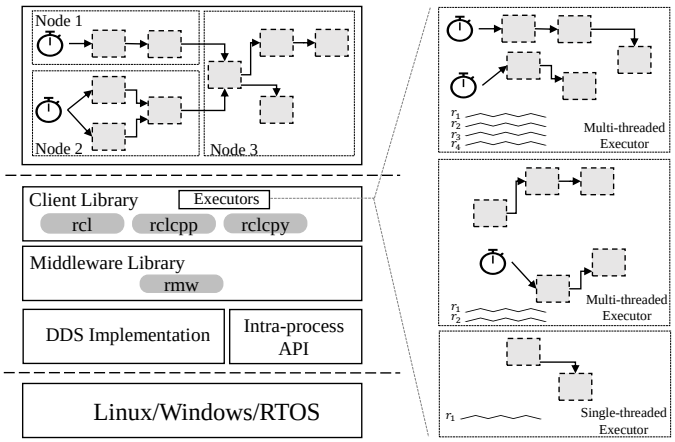


Fig. 1: ROS 2 architecture and application model

triggered by a service request and a response to a service request, respectively. There is an inherent priority order among these callback types such that timer callbacks have the highest and client callbacks have the lowest priority [1–3, 18]. For callbacks of the same type, their priorities are implicitly determined by declaration order in the node. While callbacks are actual executable entities, nodes are just containers of callbacks and serve as an abstraction to allocate callbacks to executors, i.e., once a node is assigned to an executor, all callbacks of that node are executed by that executor.

A set of callbacks with data dependencies forms a *processing chain*, or simply called a chain. Each callback can be associated with one or more chains, and chains can be formed by callbacks from different nodes. We will give a more detailed explanation of chains in Sec. III-B.

An *executor* is the ROS 2 abstraction of OS-level scheduling entities and it executes callbacks assigned to it. Each executor maintains a *ReadySet*, which is a cached set of "ready" regular (non-timer) callbacks [1, 2].[1] The ReadySet is updated only when it is empty (or there is no callback in the set eligible to execute[2]). This is the only point when the executor communicates with underlying layers, and is called a *polling point* in the literature. The time interval between two consecutive polling points is called a *processing window*, in which the executor processes regular callbacks in its ReadySet plus incoming timer callbacks. Therefore, callbacks released after one polling point are not processed until the next polling point, causing priority inversion. Also, the use of the ReadySet makes at most one instance of any callback be processed in one processing window. Hence, in addition to the priority inversion problem, a callback instance might be blocked for multiple processing windows before it gets scheduled if there are multiple pending instances of the same callback. Sec. IV discusses more details on these issues in the context of multi-threaded executors.

In addition, ROS 2 executors provide *callback groups* to control the concurrency of callback execution. There are two options: *reentrant* and *mutually-exclusive*. The reentrant callback group allows an executor to execute any ready callback with no other restriction. Hence, as long as a preceding callback in a chain has completed execution, the next callback becomes ready and can be considered for scheduling. On the other hand, the mutually-exclusive callback group limits any callback within this group not to be executed in parallel. In other words, the execution of callbacks within a mutually-exclusive group is all serialized even if the executor has multiple threads. This could be a useful option if callbacks were originally programmed for a single-threaded environment but assigned to a multi-threaded executor, or they access shared data with no synchronization in mind. However, the use of mutually-exclusive groups introduces another type of dependency in callback scheduling. Our analysis in Sec. V takes into account the effects of these two options.

[1]Timer callbacks are added to the ReadySet instantly after their release.

[2]Ready callbacks in the ReadySet may not be eligible to run due to callback groups that enforce concurrency. We will explain more details later.

B. System Model

We consider a ROS 2 system Γ running on a multi-core platform. The system Γ is composed of a set of independent chains, i.e., $\Gamma = \{\Gamma_{\mathcal{C}}, \Gamma_{\mathcal{C}'}, \Gamma_{\mathcal{C}''}, ...\}$. We present our model for callbacks, chains, and executors as below. Without loss of generality, we assume the discrete-time model in our system, in which a time interval is a non-negative integer multiplier of the system time unit (e.g., clock cycle).

Callbacks. A callback $\tau_{\langle \mathcal{C},j \rangle}$ is denoted as the j^{th} callback of a chain $\Gamma_{\mathcal{C}}$, where $1 \leq j \leq \|\Gamma_{\mathcal{C}}\|$. We characterize each callback $\tau_{\langle \mathcal{C},j \rangle}$ with:

- $E_{\langle \mathcal{C},j \rangle}$: The worst-case execution time (WCET) of an instance of a callback $\tau_{\langle \mathcal{C},j \rangle}$.
- $\pi_{\langle \mathcal{C},j \rangle}$: The priority of $\tau_{\langle \mathcal{C},j \rangle}$ (smaller values mean lower priority).

Note that callbacks belonging to a chain do not have their own periods or deadlines since they follow their chain's timing constraints. If the system has a callback that does not belong to any chain but has a timing constraint, it can be modeled as a single-callback chain for analysis purposes.

Chains. A chain $\Gamma_{\mathcal{C}} = \{\tau_{\langle \mathcal{C},1 \rangle}, \tau_{\langle \mathcal{C},2 \rangle}, ..., \tau_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}\}$ consists of $\|\Gamma_{\mathcal{C}}\|$ callbacks with a sequential execution order, i.e., $\tau_{\langle \mathcal{C},j+1 \rangle}$ can start only after $\tau_{\langle \mathcal{C},j \rangle}$ finishes. The chain $\Gamma_{\mathcal{C}}$ can be either timer-triggered (i.e., the first callback $\tau_{\langle \mathcal{C},1 \rangle}$ is a timer callback) or event-triggered (i.e., $\tau_{\langle \mathcal{C},1 \rangle}$ is a regular callback released by a periodic sensor input or message from other sub-systems). Due to data dependency, any subsequent callback becomes ready to run only when its predecessor finishes execution. For presentation simplicity, the periods of all subsequent callbacks are denoted with the same period as their chain. We characterize $\Gamma_{\mathcal{C}}$ as follows:

$$\Gamma_{\mathcal{C}} := \left(E_{\mathcal{C}}, T_{\mathcal{C}}, D_{\mathcal{C}}, \pi_{\mathcal{C}}\right)$$

- $E_{\mathcal{C}}$: The cumulative WCET of an instance of the chain $\Gamma_{\mathcal{C}}$, i.e., $E_{\mathcal{C}} = \sum_{j=1}^{\|\Gamma_{\mathcal{C}}\|} E_{\langle \mathcal{C},j \rangle}$.
- $T_{\mathcal{C}}$: The period of an instance of $\Gamma_{\mathcal{C}}$.
- $D_{\mathcal{C}}$: The relative deadline of an instance of $\Gamma_{\mathcal{C}}$.
- $\pi_{\mathcal{C}}$: The priority of $\Gamma_{\mathcal{C}}$ (smaller values mean lower priority). Following the criticality-as-priority assignment [21], we assume chains with higher criticality have higher priority.

We denote the i^{th} instance of a chain $\Gamma_{\mathcal{C}}$ and its callbacks as $\Gamma^i_{\mathcal{C}} = \{\tau^i_{\langle \mathcal{C},1\rangle}, \tau^i_{\langle \mathcal{C},2\rangle}, \tau^i_{\langle \mathcal{C},3\rangle}, ..., \tau^i_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle}\}$.

The response time of the chain $\Gamma_{\mathcal{C}}$ is denoted as $R_{\mathcal{C}}$, which means the time from when the first callback $\tau_{\langle \mathcal{C},1\rangle}$ is released until the last callback $\tau_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle}$ finishes execution. The chain is said to be *schedulable* if $R_{\mathcal{C}} \leq D_{\mathcal{C}}$. By this model, a burst release of chain instances is possible depending on the period of the chain, i.e., the next chain instance can be released before the completion of the previous instance. In other words, the response time and the deadline of a chain, $R_{\mathcal{C}}$ and $D_{\mathcal{C}}$ respectively, can be greater than its period, $T_{\mathcal{C}}$. Later, we will first analyze the case for constrained deadlines ($\forall \mathcal{C} : D_{\mathcal{C}} \leq T_{\mathcal{C}}$) and then extend it to arbitrary deadlines ($\exists \mathcal{C} : D_{\mathcal{C}} > T_{\mathcal{C}}$).

Executors. We consider a multi-threaded executor Π consisting of m worker threads, i.e., $\Pi = \{r_1, r_2, ..., r_m\}$. Each thread r_k runs on a different CPU core to maximize concurrency and has a resource reservation for guaranteed resource supply. Hence, each thread is characterized by $r_k = (C^r_k, T^r_k)$ where $1 \leq k \leq m$, meaning that r_k provides C^r_k units of CPU time every T^r_k units. The supply bound function of r_k, $sbf^*_k(\Delta)$, which lower-bounds the amount of resource supply during an interval Δ, is given by [22]:

$$sbf^*_k(\Delta) = \begin{cases} \Delta - (\kappa+1)(T^r_k - C^r_k) & \text{if } \Delta \in [(\kappa+1)T^r_k - 2\cdot C^r_k, (\kappa+1)T^r_k - C^r_k] \\ (\kappa-1)\cdot C^r_k & \text{otherwise} \end{cases} \tag{1}$$

where $\kappa = \max\left(\left[\frac{\Delta-(T^r_k-C^r_k)}{T^r_k}\right], 1\right)$. This captures the longest initial delay that the periodic resource reservation can incur, i.e., $2T^r_k - 2C^r_k$.

For ease of integration with schedulability analysis, a linear approximation of the supply bound function has been widely used [22]:

$$sbf_k(\Delta) = \begin{cases} \frac{C^r_k}{T^r_k}(\Delta - 2(T^r_k - C^r_k)) & \text{if } \Delta \geq 2(T^r_k - C^r_k) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Based on this, we can derive the following.

Def. 1 (sbf)**.** *The supply bound function of a multi-threaded executor* Π *is given by*

$$sbf_{\Pi}(\Delta) = \sum_{r_k \in \Pi} sbf_k(\Delta) \tag{3}$$

Note that worker threads of an executor do not need to be released at the same time. As long as all threads have started before $t = 0$, Eq. (3) will hold.

To find the minimum time interval required to obtain a certain amount of resource supply x, we use the pseudo-inverse function of $sbf_k(\Delta)$.

Def. 2 ($\overline{sbf}_k$ [23])**.** *The pseudo-inverse function of* sbf_k *is defined as follows:*

$$\overline{sbf}_k(x) = \min\{\Delta | sbf_k(\Delta) = x\} \tag{4}$$

where x *is the amount of resource that is needed.*

IV. Challenges in Multi-threaded Executor RTA

As explained earlier, a ROS 2 executor has a ReadySet which is updated at a polling point (PP), and the time interval between two consecutive polling points is called a processing window (PW). The latest work [2] derived the following lemmas on PWs in a single-threaded executor to improve analysis accuracy over [1].

- (Lemma 1 in [2]) "At most one instance of a regular callback executes in a processing window."
- (Lemma 2 in [2]) "Let $\Gamma_{\mathcal{C}}$ be an arbitrary chain. Suppose $\tau^i_{\langle \mathcal{C},1\rangle}$ (the first regular callback of the i^{th} instance of the $\Gamma_{\mathcal{C}}$) executes in processing window pw_n, then the earliest processing window for $\tau^l_{\langle \mathcal{C},1\rangle}$ $(l > i)$ to execute is pw_{n+l-i}."
- (Lemma 3 in [2]) "At most one regular callback instance of a chain instance executes in a processing window."
- (Lemma 4 in [2]) "The regular callback instances of a chain instance execute in consecutive processing windows one by one."

The definitions of PP and PW are rather straightforward in a single-threaded executor as there is only one thread updating ReadySet. However, in a multi-threaded executor (implemented in the `rclcpp` package of ROS 2 Galactic and newer versions to date), the ReadySet is shared among all threads. Hence, one or more threads might become idle (i.e., ReadySet is empty or has no eligible callback to execute) and update ReadySet, while other threads are still executing their callbacks from the previous version of ReadySet. Those who were executing callbacks when ReadySet was updated cannot even notice such an update. Therefore, we need to revise the definitions of PP and PW for multi-threaded executors:

- **Polling Point (PP)**: A time point when *at least one thread* becomes idle.
- **Processing Window (PW)**: The time interval between two consecutive PPs, *regardless of which thread triggered the new PP*. In the n^{th} PW, pw_n, there might be some threads that are still executing callbacks whose start times were in previous PWs, pw_{n-p}. These callbacks are considered *carry-in* callbacks for pw_n. Also, some callbacks that started in pw_n may continue to execute in pw_{n+q}. Such callbacks are considered as *carry-out* callbacks for pw_n.

Based on the above definitions, some of the lemmas derived in [2] are invalid or conditionally valid for a multi-threaded executor. Below we fix them or confirm their validity.

- (Lemma 1 in [2]) This lemma stays valid in a multi-threaded executor *only if* all chains in the system Γ have *constrained deadlines*. When chains in Γ have *arbitrary deadlines*, then at most m instances of each regular callback can be in execution in the same PW, where m is the number of threads in a multi-threaded executor. This is because a new PP can be triggered by any idling thread; hence, there can be $m-1$ instances carried-in from previous PWs by $m-1$ threads and 1 new instance by 1 thread that triggered the current PW.
- (Lemma 2 in [2]) Since this lemma directly follows the lemma 1 in [2], similarly, it stays valid *only if* all chains in Γ have *constrained deadlines*. However, for chains with

arbitrary deadlines, the earliest PW for $\tau^l_{\langle \mathcal{C},1\rangle}$ $(l > i)$ to execute would be $pw_{n+\lfloor \frac{l-i}{m} \rfloor}$.
- (Lemma 3 in [2]) This lemma remains valid with the same proof as in [2].
- (Lemma 4 in [2]) This lemma is *not* valid anymore because the new definition of PP does not guarantee that all callbacks finish their execution when the new PW begins (i.e., the execution of some callbacks may span over multiple PWs). Hence, there is no guarantee on which PW the succeeding callback instance can get into ReadySet.

These revisions may help develop a timing analysis for a multi-threaded executor, by following the approach of [2] that focuses on PW analysis. However, there are still many difficulties to be solved. At first, analyzing when a new PP happens and how long a PW takes is not as straightforward as in a single-threaded executor. Moreover, it would be hard to determine how many PWs each callback would take to complete its execution and how many PWs exist between two consecutive callbacks. The difficulty multiplies when chains with arbitrary deadlines are considered. Instead, in this paper, we take a different approach: analyzing the response time of a chain without having PW in mind. Our proposed analysis is built by extending the conventional non-preemptive global task scheduling analysis [24] and taking into account semantic differences introduced by chains, callback dependencies, and the ReadySet management. This approach also allows us to incorporate priority-driven scheduling enhancements that yield a significant benefit to critical chains.

V. Proposed RTA Framework

This section presents our proposed response-time analysis (RTA) framework. We first review the conventional non-preemptive fixed-priority (NP-FP) global task scheduling analysis developed for non-ROS systems (Sec. V-A). Then, for a single ROS 2 multi-threaded executor with *reentrant* callback groups, we analyze the response time of a chain with a constrained deadline (Sec. V-B). Based on this, we present priority-driven scheduling enhancements and the corresponding analysis (Sec. V-C), and extend these to chains with arbitrary deadlines (Sec. V-D). Finally, we relax our assumptions by incorporating the effects of *mutually-exclusive* callback groups into our analysis (Sec. V-E) and discussing how to analyze the end-to-end response time of a chain that spans across multiple executors regardless of their types (Sec. V-F).

A. Review of NP-FP Task Scheduling

According to the NP-FP global task scheduling analysis [25], to obtain the worst-case response time of a given non-preemptive task τ_i, we need to find the latest time that a τ_i's job starts its first unit of execution. Assuming this first unit finishes after a time interval Δ from the release of the job, the response time of τ_i is $\Delta + \overline{sbf}_k(E_i - 1)$ where E_i is the WCET of τ_i and $\overline{sbf}_k(E_i - 1)$ is the minimal time interval required by any processor (worker thread r_k) to execute $E_i - 1$. To find Δ, there should be enough resource supply for the system's demand. For at least one unit of workload of τ_i to execute in Δ, the following inequality has to hold:

$$dbf(\Delta) < sbf_\Pi(\Delta) \tag{5}$$

where $dbf(\Delta)$ and $sbf_\Pi(\Delta)$ stand for the demand-bound function and the supply-bound function, respectively. Note that $sbf_\Pi(\Delta) = m \cdot \Delta$ is the maximum resource supply in a m-processor system.

To determine $dbf(\Delta)$, we first need to calculate the workload of other interfering tasks in Δ. Consider a set of non-preemptive tasks with constrained deadlines. The workload of a task τ_j with the WCET of E_j, the period of T_j, and the deadline of D_j (i.e., $\tau_j := \langle E_j, T_j, D_j \rangle$) during an arbitrary time interval Δ can be upper-bounded by the following, as proposed in [26]:

Lemma 1 (Workload [26])**.** *In an arbitrary time interval Δ, the jobs of τ_j with a constrained deadline can execute up to*

$$W_j(\Delta, \alpha) = \lfloor \frac{\Delta + \alpha}{T_j} \rfloor \cdot E_j + \min\left(E_j, \Delta - \lfloor \frac{\Delta + \alpha}{T_j} \rfloor \cdot T_j\right) \tag{6}$$

where α is an extra time to capture carry-in jobs in Δ, i.e., $\alpha = R_j - E_j$ if the response time R_j is known, and $\alpha = D_j - E_j$ otherwise.

Then blocking from lower-priority tasks needs to be considered in $dbf(\Delta)$. In [24], a NP-FP global scheduling analysis is provided for periodic non-preemptive tasks, which limits the number of carry-in jobs from lower-priority tasks that can block τ_i in Δ by the number of processors, m.

Lemma 2 (LeSh [24] in [25]'s presentation)**.** *The response time of a periodic non-preemptive task $\tau_i = \langle E_i, T_i \rangle$ is upper-bounded by $R_i = \Delta + \overline{sbf}_k(E_i - 1)$, if $dbf(\Delta) < sbf_\Pi(\Delta)$ holds for the following $dbf(\Delta)$:*

$$\begin{aligned} dbf(\Delta) = &\sum_{\tau_h \in hp(\tau_i)} W_h(\Delta, R_h - E_h) \\ &+ \sum_{\tau_l \in mlp(\tau_i)} \min(E_l - 1, \Delta) \end{aligned} \tag{7}$$

where $hp(\tau_i)$ is the set of higher-priority tasks than τ_i, and $mlp(\tau_i)$ is the subset of lower-priority tasks than τ_i that give the m largest $\min(E_l - 1, \Delta)$ values (i.e., $|mlp(\tau_i)| = m$).

The first term of $dbf(\Delta)$ in Eq. (7) upper-bounds the amount of interference from higher-priority tasks until τ_i begins execution, and the second term captures the blocking time from lower-priority jobs that have started execution before Δ. If $dbf(\Delta) < sbf_\Pi(\Delta)$ holds, at least one time unit is available in $[t, t + \Delta)$ for the job of a non-preemptive task τ_i released at t to start its execution; hence, the response time is bounded by $R_i = \Delta + \overline{sbf}_k(E_i - 1)$. This equation can be solved by a fixed-point iteration, with $\Delta = 1$ as a start condition.

Note that Eq. (7) uses $R_h - E_h$ for α of the workload function given in Eq. (6). To avoid the need to compute the response time of high-priority tasks in advance, $R_h - E_h$ can be replaced with $D_h - E_h$, where D_h is the deadline of τ_h, if τ_h is schedulable ($\because D_h \geq R_h$).

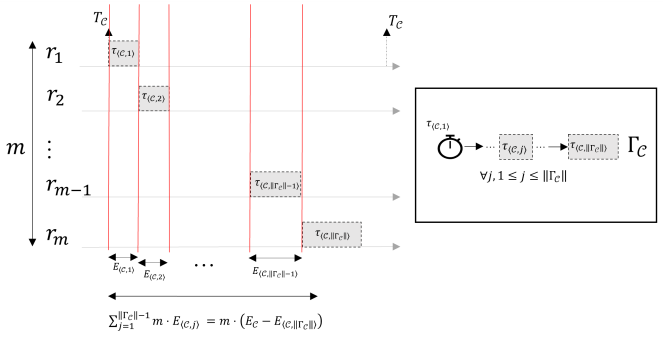


Fig. 2: Proof of Lemma 3

B. Response Time of Chains

Since the ROS 2 multi-threaded executor follows global work-conserving non-preemptive scheduling, we can derive a response-time analysis for a chain $\Gamma_{\mathcal{C}}$ with a constrained deadline in a similar form as in Lemma 2, by finding the longest time interval Δ such that the last callback of $\Gamma_{\mathcal{C}}$ ($\tau_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle}$) has at least one unit of execution during $[t, t+\Delta)$ where t is the release time of $\Gamma_{\mathcal{C}}$. Then, the response time of the chain could be bounded by $\Delta + \overline{sbf}_k(E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle} - 1)$, analogous to the case for NP-FP task scheduling.

However, there are several differences to consider. The first issue is an additional blocking caused by the precedence dependencies between callbacks of the chain under analysis. This happens because the next callback cannot start execution until its previous callback is completed, even if there exist other idle threads in the executor. While these idle threads can be utilized by callbacks from other chains at runtime, for analysis purposes, such idle threads can be assumed to be occupied by an artificial workload that needs to be added to $dbf(\Delta)$.

Lemma 3. *Consider two adjacent callbacks of a chain $\Gamma_{\mathcal{C}}$, $\tau_{\langle\mathcal{C},j\rangle}$ and $\tau_{\langle\mathcal{C},j+1\rangle}$, on a m-threaded executor. The precedence-dependency blocking caused by $\tau_{\langle\mathcal{C},j\rangle}$ introduces an additional workload as interference to the start of $\tau_{\langle\mathcal{C},j+1\rangle}$, which is upper-bounded by*

$$B_{\langle\mathcal{C},j+1\rangle} = m \cdot E_{\langle\mathcal{C},j\rangle} \tag{8}$$

Proof. Suppose $\Gamma_{\mathcal{C}}$ is the only chain in the system and we are interested in the latest time that $\tau_{\langle\mathcal{C},2\rangle}$ can start its execution from the release time of $\Gamma_{\mathcal{C}}$. Since $\tau_{\langle\mathcal{C},2\rangle}$ cannot start until $\tau_{\langle\mathcal{C},1\rangle}$ completes, from $\tau_{\langle\mathcal{C},2\rangle}$'s view, $\tau_{\langle\mathcal{C},1\rangle}$ behaves as if it occupied all m threads. This artificial workload equals to $m \cdot E_{\langle\mathcal{C},1\rangle}$, which needs to be added to dbf of $\tau_{\langle\mathcal{C},2\rangle}$. By induction, as shown in Fig. 2, the same happens for any callback $\tau_{\langle\mathcal{C},j\rangle}$ regardless of on which thread it is running. □

Based on this, we can derive the following.

Lemma 4. *For the last callback of a chain $\Gamma_{\mathcal{C}}$, the precedence-dependency blocking caused by all of its preceding callbacks is given by $m \cdot (E_{\mathcal{C}} - E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle})$.*

Proof. By Lemma 3, $\sum_{j=1}^{\|\Gamma_{\mathcal{C}}\|-1} m \cdot E_{\langle\mathcal{C},j\rangle} = m \cdot (E_{\mathcal{C}} - E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle})$. □

The next issue is due to the ReadySet management. Since ReadySet is a cached set of ready callbacks and is updated only at PPs, the callbacks of the chain $\Gamma_{\mathcal{C}}$ under analysis can be blocked multiple times by lower-priority callbacks of other chains during $\Gamma_{\mathcal{C}}$'s execution, and only the execution of the last callback of $\Gamma_{\mathcal{C}}$ is not interfered due to the nature of non-preemptive scheduling [1, 2]. Therefore, we need to treat the callback instances of all other chains (regardless of their priorities) as *interfering* callbacks, as if they had higher priority than any callback $\tau_{\langle\mathcal{C},j\rangle} \in \Gamma_{\mathcal{C}}$. For each interfering callback, the task-level workload function given in Eq. (6) can be used directly since a periodic non-preemptive task τ_j in Eq. (6) is equivalent to a callback in our model. The entire workload of an interfering chain $\Gamma_{\mathcal{C}'}$ can be upper-bounded as follows:

Lemma 5. *In an arbitrary time interval Δ, the instances of a chain $\Gamma_{\mathcal{C}'}$ with a constrained deadline can execute up to*

$$W_{\mathcal{C}'}(\Delta, \alpha) = \lfloor \frac{\Delta + \alpha}{T_{\mathcal{C}'}} \rfloor \cdot E_{\mathcal{C}'} + \min\left(E_{\mathcal{C}'}, \Delta - \lfloor \frac{\Delta + \alpha}{T_{\mathcal{C}'}} \rfloor \cdot T_{\mathcal{C}'}\right) \tag{9}$$

where α is an extra time to capture carry-in instances of $\Gamma_{\mathcal{C}'}$.

Proof. Similar to Eq. (6), in the first term, the total number of instances of $\Gamma_{\mathcal{C}'}$ (including a carry-in) that contributes to the workload with its entire WCET ($E_{\mathcal{C}'}$) is obtained by $\lfloor \frac{\Delta+\alpha}{T_{\mathcal{C}'}} \rfloor$ and multiplied by $E_{\mathcal{C}'}$. The workload of the carry-out instance is bounded by the second term. □

Based on Lemmas 3 and 5, we can obtain the following theorem to find the response time of a chain $\Gamma_{\mathcal{C}}$.

Theorem 1. *The response time of a chain $\Gamma_{\mathcal{C}} = \{\tau_{\langle\mathcal{C},1\rangle}, \tau_{\langle\mathcal{C},2\rangle}, ..., \tau_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle}\}$ with a **constrained deadline** on a **standard** ROS 2 multi-threaded executor with m threads is upper bounded by $R_{\mathcal{C}} = \Delta + \overline{sbf}_k(E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle} - 1)$, if $dbf(\Delta) < sbf_{\Pi}(\Delta)$ holds for the following $dbf(\Delta)$:*

$$dbf(\Delta) = m \cdot (E_{\mathcal{C}} - E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle}) + \sum_{\forall\Gamma_x \in \Gamma - \{\Gamma_{\mathcal{C}}\}} W_x(\Delta, D_x - E_x) \tag{10}$$

$dbf(\Delta)$ can be solved by a fixed-point iteration with an initial condition of $\Delta = 1$.

Proof. Eq. (10) captures all possible workloads in $[t, t+\Delta)$, where t is the release time of an instance of $\Gamma_{\mathcal{C}}$, until the last callback of $\Gamma_{\mathcal{C}}$ can start its first unit of execution. By Lemma 3, we know that the maximum workload caused by precedence dependencies is $m \cdot (E_{\mathcal{C}} - E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle})$. In addition, the maximum workload from the instances of other chains is upper-bounded by the sum of W_x given by Lemma 5. As there is no other source of workloads in Δ, the last callback of $\Gamma_{\mathcal{C}}$ can start at least one unit of its execution in Δ if $dbf(\Delta) < sbf_{\Pi}(\Delta)$ holds, and the response time of $\Gamma_{\mathcal{C}}$ is bounded by $\Delta + \overline{sbf}_k(E_{\langle\mathcal{C},\|\Gamma_{\mathcal{C}}\|\rangle} - 1)$. □

C. Priority-Driven Enhancements

As explained earlier, the ReadySet management scheme of the standard ROS 2 multi-threaded executor requires our

analysis to capture all callback instances of other chains as interfering callbacks, regardless of their priorities. This is particularly problematic for critical chains as their response times could be unnecessarily penalized by non-critical chains. However, prior work on a single-threaded executor [3, 18] demonstrated that assigning callback priorities based on their respective chain priorities and scheduling ready callbacks by strictly based on their priorities can alleviate analytical pessimism and improve chain response time. Hence, we apply this approach to multi-threaded executors and discuss its implications in analysis.

The implementation of priority-driven scheduling in a multi-threaded executor is rather straightforward. Instead of having each thread look for ready callbacks in ReadySet and letting it update only when it is empty, we can modify the executor code such that each thread updates ReadySet whenever it needs to choose a ready callback to execute. As a result, a newly-arrived high-priority callback does not need to wait for the other callbacks already fetched in ReadySet to finish their executions so that the next PP is triggered. Updating ReadySet in this manner may seem like a lot of extra overhead, but we found the frequency of ReadySet updates in the priority-driven multi-threaded executor is not much higher than in the standard one. Sec. VI-A analyzes this overhead.

For callback priority assignment, we directly adopt the chain-aware assignment scheme proposed by [3], shown in Alg. 1. Basically, it makes sure that callbacks from higher-priority chains get higher callback priority than those from lower-priority chains. Within each chain, earlier callbacks get lower priority than later callbacks, i.e., $\pi_{\langle \mathcal{C},j \rangle} < \pi_{\langle \mathcal{C},j+1 \rangle}$. As discussed in Sec. III-B, we follow the criticality-as-priority assignment [21] for chains; thus, this assignment yields higher priorities to callbacks from critical chains. Also, since the standard ROS 2 executor interface does not have APIs to set callback priorities (callback priorities are determined implicitly by declaration order), we also adopted such APIs from [3] and applied them to the multi-threaded executor code base.

Algorithm 1 Chain-aware callback priority assignment [3]

```
1: Input: Γ:chains
2: Γ ← sort in ascending order of chain priority π
3: p ← 1                                  ▷ Initialize current priority
4: for all Γ_C ∈ Γ do
5:     for all τ_⟨C,j⟩ ∈ Γ_C do
6:         π_⟨C,j⟩ ← p
7:         p ← p + 1
```

With the executor modifications for strictly priority-driven callback scheduling and the callback priority assignment, a chain of interest is no longer blocked multiple times by callbacks from lower-priority chains. Now we provide a response-time analysis for priority-driven multi-threaded executors.

Theorem 2. *The response time of a chain* $\Gamma_{\mathcal{C}} = \{\tau_{\langle \mathcal{C},1 \rangle}, \tau_{\langle \mathcal{C},2 \rangle}, ..., \tau_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}\}$ *with a* ***constrained deadline*** *on a* ***priority-driven*** *ROS 2 multi-threaded executor with* m *threads is upper-bounded by* $R_{\mathcal{C}} = \Delta + \overline{sbf}_k(E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle} - 1)$*, if* $dbf(\Delta) < sbf_{\Pi}(\Delta)$ *holds for the following* $dbf(\Delta)$*:*

$$\begin{aligned} dbf(\Delta) = m \cdot (E_{\mathcal{C}} - E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}) \\ + \sum_{\forall \Gamma_x \in \Gamma - \{\Gamma_{\mathcal{C}}\} \wedge \pi_x > \pi_{\mathcal{C}}} W_x(\Delta, D_x - E_x) \\ + \sum_{\forall \tau_l \in mlp(\tau_{\langle \mathcal{C},1 \rangle})} \min(E_l - 1, \Delta) \end{aligned} \tag{11}$$

where $mlp(\tau_{\langle \mathcal{C},1 \rangle})$ *returns at most* m *largest callbacks with lower-priority than* $\tau_{\langle \mathcal{C},1 \rangle}$ *where it includes only one callback from each chain (i.e.,* $|mlp(\tau_{\langle \mathcal{C},1 \rangle})| = min(m, |\{\Gamma_y | \pi_y < \pi_{\mathcal{C}}\}|)$*).*

Proof. In priority-driven scheduling, we need to upper-bound (i) the workload caused by precedence dependencies, (ii) the interference from higher-priority chain instances, (iii) and the blocking time from lower-priority chain instances. Item (i) remains the same as in Eq. (10) since it is inherent to the chain under analysis, not subject to the scheduling policy used.

By Alg. 1, all callbacks of a higher-priority chain have higher priorities than the chain of interest. Therefore, to upper-bound (ii), the workload function in Lemma 5 can be used for higher-priority chains than $\Gamma_{\mathcal{C}}$, i.e., $\forall \Gamma_x \in \Gamma - \{\Gamma_{\mathcal{C}}\} \wedge \pi_x > \pi_{\mathcal{C}}$.

For (iii), lower-priority callbacks can block the execution time of higher-priority callbacks only when they started execution at least for one unit before Δ. In a m-threaded executor, the number of lower-priority callbacks that can do so is up to m. Thus, to upper-bound the blocking time, we can find the m largest callbacks from other chains with lower callback priority than $\tau_{\langle \mathcal{C},1 \rangle}$, which is the lowest-priority callback of $\Gamma_{\mathcal{C}}$ by Alg. 1. Here, the precedence dependency within a chain limits up to one lower-priority callback from each chain can contribute to the blocking time. In other words, each chain cannot have more than one callback that has started before Δ and continues execution in Δ. This number is bounded by $|\{\Gamma_y | \pi_y < \pi_{\mathcal{C}}\}|$ since only chains with lower chain priority than $\Gamma_{\mathcal{C}}$ have callbacks with lower callback priority than $\tau_{\langle \mathcal{C},1 \rangle}$. As a result, the number of lower-priority callbacks τ_l contributing to the blocking time is bounded by the minimum of these two conditions, and the total blocking time is obtained by summing $\min(E_l - 1, \Delta)$ since those callbacks already started one unit of execution before Δ. □

D. Chains with Arbitrary Deadlines

This section extends our response-time analysis to arbitrary-deadline chains, where new instances of a chain can arrive earlier than the completion of previous instances. First, we need to revise the workload function given in Lemma 5 because it is valid for constrained deadlines only. The implicit assumption made in that lemma (and also Lemma 1) is that each chain has only at most one instance as carry-in and at most one instance as carry-out in a time interval Δ. With arbitrary deadlines ($T_{\mathcal{C}} < D_{\mathcal{C}}$), a chain can have multiple of its previous or next instances as carry-in jobs or as carry-out jobs, respectively, as their executions can overlap. Therefore, the workload of an arbitrary-deadline chain should be captured solely based on its period.

Lemma 6. *In an arbitrary time interval Δ, the instances of a chain $\Gamma_{\mathcal{C}'}$ with an arbitrary deadline can execute up to*

$$W^*_{\mathcal{C}'}(\Delta, \alpha) = \lceil \frac{\Delta + \alpha}{T_{\mathcal{C}'}} \rceil \cdot E_{\mathcal{C}'} \tag{12}$$

where α is an extra time to capture carry-in instances of $\Gamma_{\mathcal{C}'}$.

Proof. The worst-case workload can be upper-bounded by capturing the maximum possible arrivals. Therefore, we replace the floor function in Lemma 5 with the ceiling function and the second term of Lemma 5 is no longer needed. □

With the revised workload function, we can extend Theorem 1 to arbitrary deadlines as follows:

Theorem 3. *The response time of a chain $\Gamma_{\mathcal{C}} = \{\tau_{\langle \mathcal{C},1 \rangle}, \tau_{\langle \mathcal{C},2 \rangle}, ..., \tau_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}\}$ with an **arbitrary deadline** on a **standard** ROS 2 multi-threaded executor with m threads is upper-bounded by $R_{\mathcal{C}} = \Delta + \overline{sbf}_k(E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle} - 1)$, if $dbf(\Delta) < sbf_{\Pi}(\Delta)$ holds for the following $dbf(\Delta)$:*

$$\begin{aligned} dbf(\Delta) = m \cdot (E_{\mathcal{C}} - E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}) \\ + (\sum_{\forall \Gamma_x \in \Gamma} W^*_x(\Delta, D_x - E_x)) - E_{\mathcal{C}} \end{aligned} \tag{13}$$

Proof. The difference from Theorem 1 is the second and third terms, which capture the maximum workload from interfering chain instances. Here, interfering chain instances include *other* instances of $\Gamma_{\mathcal{C}}$ itself because they can cause self-interference to the instance being analyzed. By considering $\Gamma_{\mathcal{C}}$ in the summing term ($\forall \Gamma_x \in \Gamma$, instead of $\forall \Gamma_x \in \Gamma - \{\Gamma_{\mathcal{C}}\}$), $W^*_{\mathcal{C}}(\Delta, D_{\mathcal{C}} - E_{\mathcal{C}})$ gives all instances of $\Gamma_{\mathcal{C}}$ including the instance being analyzed. Therefore, to avoid double-counting, $E_{\mathcal{C}}$ needs to be subtracted from $dbf(\Delta)$. □

Next is the extension of Theorem 2 for priority-driven multi-threaded executor analysis.

Theorem 4. *The response time of a chain $\Gamma_{\mathcal{C}} = \{\tau_{\langle \mathcal{C},1 \rangle}, \tau_{\langle \mathcal{C},2 \rangle}, ..., \tau_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}\}$ with an **arbitrary deadline** on a **priority-driven** ROS 2 multi-threaded executor with m threads is upper-bounded by $R_{\mathcal{C}} = \Delta + \overline{sbf}_k(E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle} - 1)$, if $dbf(\Delta) < sbf_{\Pi}(\Delta)$ holds for the following $dbf(\Delta)$:*

$$\begin{aligned} dbf(\Delta) = m \cdot (E_{\mathcal{C}} - E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}) \\ + (\sum_{\forall \Gamma_x \in \Gamma \wedge \pi_x \geq \pi_{\mathcal{C}}} W^*_x(\Delta, D_x - E_x)) - E_{\mathcal{C}} \\ + (\sum_{\forall \tau_l \in mlp^*(\tau_{\langle \mathcal{C},1 \rangle}, \Delta)} \min(E_l - 1, \Delta)) \end{aligned} \tag{14}$$

where $mlp^*(\tau_{\langle \mathcal{C},1 \rangle}, \Delta)$ returns at most m largest callbacks with lower priority than $\tau_{\langle \mathcal{C},1 \rangle}$ while it includes only one callback from each instance of chains released in Δ (i.e., $|mlp^*(\tau_{\langle \mathcal{C},1 \rangle}, \Delta)| = min(m, \sum_{\pi_y < \pi_{\mathcal{C}}} \lceil \frac{\Delta + D_y - E_y}{T_y} \rceil)$).

Proof. As explained in the proof of Theorem 3, interfering chain instances include other instances of $\Gamma_{\mathcal{C}}$ itself. Hence, the second term considers $\Gamma_{\mathcal{C}}$ by $\pi_x \geq \pi_{\mathcal{C}}$ instead of $\pi_x > \pi_{\mathcal{C}}$. As in Eq. (13), $E_{\mathcal{C}}$ needs to be deducted to avoid double-counting.

For blocking time, we know that lower-priority callbacks can block higher-priority callbacks only when they started execution for at least one unit before Δ, and the maximum number of such callbacks is bounded to m. The difference from the constrained-deadline case is that each of such callbacks contributing to the blocking time can be originated from each instance of other chains because there may exist multiple outstanding instances of the same chain during Δ. This number is bounded by $\sum_{\pi_y < \pi_{\mathcal{C}}} \lceil \frac{\Delta + D_y - E - y}{T_y} \rceil$. Hence, the number of lower-priority callbacks τ_l contributing to the blocking time is bounded by the minimum of these two conditions, and the total blocking time can be obtained by maximizing the sum of $\min(E_l - 1, \Delta)$.

□

E. Mutually-Exclusive Callback Groups

Our analysis in the previous sections is for chains with reentrant callback groups. In this section, we study the effects of mutually-exclusive callback groups which introduce yet another type of precedence dependency in callback scheduling, adding more workload to $dbf(\Delta)$. As mentioned in Sec. III-A, the use of a mutually-exclusive callback group limits any callback within this group not to be executed in parallel. In other words, the execution of callbacks within a mutually-exclusive group is all serialized.

If the mutually-exclusive callback group option is enabled for some callbacks, Theorems 1 and 3 (standard ROS 2 multi-threaded executor) need to be extended as follows.

Theorem 5. *With **mutually-exclusive callback groups**, the response time of a chain $\Gamma_{\mathcal{C}} = \{\tau_{\langle \mathcal{C},1 \rangle}, \tau_{\langle \mathcal{C},2 \rangle}, ..., \tau_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle}\}$ on a **standard** ROS 2 multi-threaded executor with m threads is upper-bounded by $R_{\mathcal{C}} = \Delta + \overline{sbf}_k(E_{\langle \mathcal{C},\|\Gamma_{\mathcal{C}}\| \rangle} - 1)$, if $dbf(\Delta) < sbf_{\Pi}(\Delta)$ holds for the following $dbf(\Delta)$:*

$$\begin{aligned} dbf(\Delta) = \text{RHS} \quad \text{of} \quad \text{Eq. (10)} \quad \text{or} \quad \text{Eq. (13)} \\ + \sum_{j=1}^{\|\Gamma_{\mathcal{C}}\|} m \cdot \text{strangers\_in\_group}(\tau_{\langle \mathcal{C},j \rangle}) \end{aligned} \tag{15}$$

where strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}$) *sums up the execution time of callbacks in the same mutually-exclusive group as $\tau_{\langle \mathcal{C},j \rangle}$, except those from the chain under analysis, Γ_C.*

The pseudo-code of the strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}$) function is given by Alg. 2.

Algorithm 2 strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}$)

```
retval = 0;
group = get_callback_group(τ⟨C,j⟩) - {τ⟨C,j⟩};
for all τ⟨x,k⟩ ∈ group do
    if x ≠ C then
        retval = retval + E⟨x,k⟩
return retval;
```

Proof. This theorem differs from Theorem 1 or Theorem 3 by only the extra workload added to the end of $dbf(\Delta)$. So, we prove the necessity of the last term, $\sum_{j=1}^{\|\Gamma_{\mathcal{C}}\|} m \cdot$ strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}$), in $dbf(\Delta)$.

Based on the characteristics of a mutually-exclusive callback group, once a callback τ_k of the group G_1 executes, $m \cdot E_k$ is the safe upper-bound on the blocking time that τ_k causes to its group-mate callbacks in G_1, as shown by Lemma 3. Thus, the maximum blocking to a callback τ_j from all of its group-mates can be bounded by $m \cdot \sum_{\forall \tau_k \in G_1 \wedge j \neq k} E_k$, where the term $\sum_{\forall \tau_k \in G_1 \wedge j \neq k} E_k$ is the return value of strangers_in_group(τ_j).

The effect of mutually-exclusive callback groups on a chain $\Gamma_\mathcal{C}$ is maximized when all callbacks of $\Gamma_\mathcal{C}$ are in different groups and all of their group-mates block their execution as long as possible. Therefore, $m \cdot$ strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}$) should be considered as part of $dbf(\Delta)$ for each $\tau_{\langle \mathcal{C},j \rangle} \in \Gamma_\mathcal{C}$.

As can be seen, strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}$) excludes the blocking from group-mate callbacks that are from $\Gamma_\mathcal{C}$, the chain under analysis (line 4 of Alg. 2). This is because the blocking effect from callbacks of the same chain has been already considered as precedence-dependency blocking in the first term of Eq. (10) and Eq. (13). Therefore, we avoid double-counting them in Alg. 2. □

Theorems 2 and 4 for priority-driven multi-threaded executors are changed as follows in the presence of mutually-exclusive groups.

Theorem 6. *With **mutually-exclusive callback groups**, the response time of a chain $\Gamma_\mathcal{C} = \{\tau_{\langle \mathcal{C},1 \rangle}, \tau_{\langle \mathcal{C},2 \rangle}, ..., \tau_{\langle \mathcal{C},\|\Gamma_\mathcal{C}\| \rangle}\}$ on a **priority-driven** ROS 2 multi-threaded executor with m threads is upper-bounded by $R_\mathcal{C} = \Delta + \overline{sbf}_k(E_{\langle \mathcal{C},\|\Gamma_\mathcal{C}\| \rangle} - 1)$, if $dbf(\Delta) < sbf_\Pi(\Delta)$ holds for the following $dbf(\Delta)$:*

$$dbf(\Delta) = \text{RHS of Eq. (11) or Eq. (14)} + \sum_{j=1}^{\|\Gamma_\mathcal{C}\|} m \cdot \text{hp\_strangers\_in\_group}(\tau_{\langle \mathcal{C},j \rangle}, \Delta) \quad (16)$$

where hp_strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}, \Delta$) *sums up the execution time of higher-priority callbacks instances than $\tau_{\langle \mathcal{C},j \rangle}$ in the same mutually-exclusive group as $\tau_{\langle \mathcal{C},j \rangle}$ except those from the chain under analysis, Γ_C.*

The hp_strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}, \Delta$) function is given by Alg. 3.

Algorithm 3 hp_strangers_in_group($\tau_{\langle \mathcal{C},j \rangle}, \Delta$)

```
1: retval = 0;
2: group = get_callback_group(τ⟨C,j⟩) - {τ⟨C,j⟩};
3: for all τ⟨x,k⟩ ∈ group do
4:     if x ≠ C ∧ π⟨x,k⟩ > π⟨C,j⟩ then
5:         retval = retval + ⌈(Δ+D_x−E_x)/T_x⌉ · E⟨x,k⟩
6: return retval;
```

Proof. The proof is similar to that for Theorem 5. The only difference here is that higher-priority group-mate callbacks are the ones who execute first and cause the blocking time to a callback of interest. Also, since a callback can be blocked by multiple instances of its higher-priority group-mates, we need to consider the maximum possible number of instances of higher-priority group-mates in the time interval Δ, which is bounded by $\lceil \frac{\Delta + D_x - E_x}{T_x} \rceil$ for any chain Γ_x as shown in Lemma 6. □

F. End-to-End Response Time across Executors

The previous sections focused on the response time of a chain on one multi-threaded executor. However, a chain may span across multiple executors where each executor can be either single-threaded or multi-threaded. To find the end-to-end response time of a chain spanning across multiple executors, one can utilize the Compositional Performance Analysis (CPA) approach, as discussed in [1]. For a set of executors, we can assign a reservation to each single-threaded or multi-threaded executor. Then, we find the response time of individual sub-chain that is executed by one executor. The analysis of such sub-chains can be done using our analysis in this section if executed by a multi-threaded executor, or the analysis from previous work [1–3] if executed by a single-threaded executor. One can also use our analysis with $m = 1$ for a single-threaded executor, though it would give a more pessimistic upper bound than those single-thread analyses. Then, to find the end-to-end response time of a chain across multiple executors, we can sum up the response time of all sub-chains associated with the chain of interest plus the propagation delay between executors. This works because the activation of each sub-chain is triggered by the completion of the preceding sub-chain and can be delayed by the amount of propagation delay. It is worth noting that sub-chains on subsequent executors can have release jitters at runtime, e.g., the preceding sub-chain finishes earlier than its worst-case response time. One might suspect that such jitters introduce more interference to the other chains; however, our analysis already captures the effect of jitters with α in the workload functions (Eq. (9) and Eq. (12)).

For example, consider a chain $\Gamma_\mathcal{C}$ consisting of n sub-chains spanning across n different executors as shown in Fig. 3. Due to precedence dependencies among callbacks inside the chain, callbacks of a sub-chain $\Gamma_{\mathcal{C}n}$ can not start their execution until the last callback of the preceding sub-chain $\Gamma_{\mathcal{C}(n-1)}$ is finished. Once the sub-chain $\Gamma_{\mathcal{C}(n-1)}$ finishes, it takes δ_{n-1} (the propagation delay between executors $n-1$ and n) for the first callback of $\Gamma_{\mathcal{C}n}$ to get ready. This pattern applies to all sub-chains of $\Gamma_\mathcal{C}$. Therefore, $R_\mathcal{C}$ can be obtained by the sum of the response time of all sub-chains and their propagation delays, i.e., $R_\mathcal{C} = \sum_{i=1}^{n-1}(R_{\mathcal{C}i} + \delta_i) + R_{\mathcal{C}n}$.

It is possible that two or more sub-chains of the same chain have been assigned to the same executor. If these sub-chains are adjacent to each other, they can be merged into a single sub-chain and analyzed. If they are not adjacent, we can treat them as two independent chains; hence, each sub-chain is considered an interfering chain to the other sub-chain for analysis purposes.

VI. Evaluation

We evaluate our proposed work through a case study on a real platform and schedulability experiments using randomly-

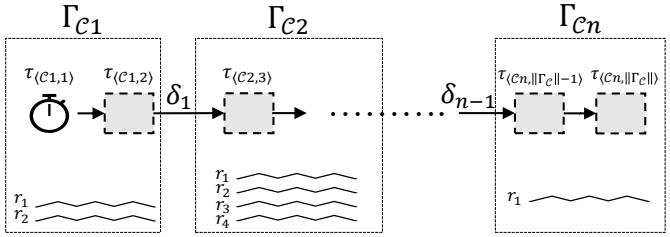


Fig. 3: Chain $\Gamma_{\mathcal{C}}$ spanning across multiple executors

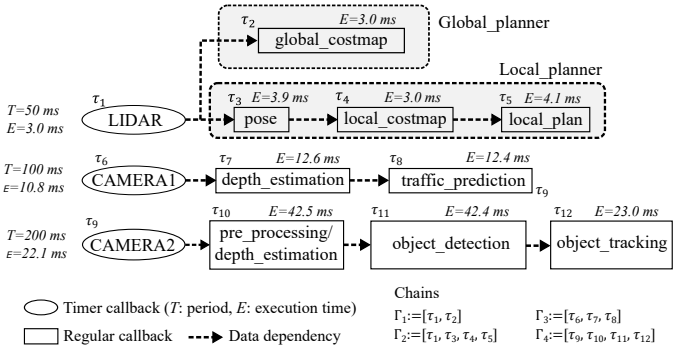
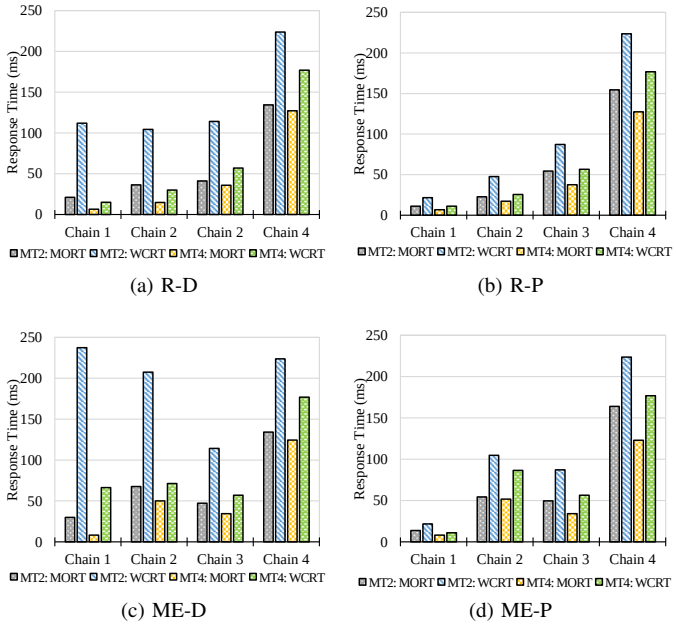


Fig. 4: Chain set for case study

generated workloads. For the case study, we used the Galactic version of ROS 2 on an NVIDIA Jetson AGX Xavier (AGX) platform. Four CPU cores were used in our experiments, each set to run at the maximum frequency (2.2 GHz). The proposed priority-driven scheduling enhancements for multi-threaded executors were implemented as modifications to the `rclcpp` package of ROS 2.[3]

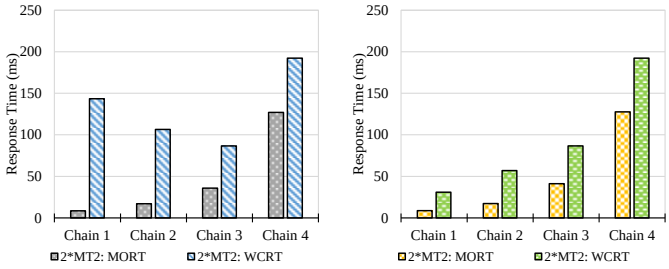


(a) R-D (b) R-P

(c) ME-D (d) ME-P

Fig. 5: Comparison of observed and analyzed response times

[3]Our source code is available at https://github.com/rtenlab/ros2-picas.



(a) 2*MT2-R-D (b) 2*MT2-R-P

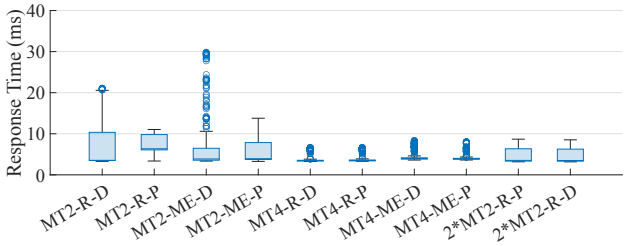
Fig. 6: Chains across multiple executors

A. Case Study on AGX

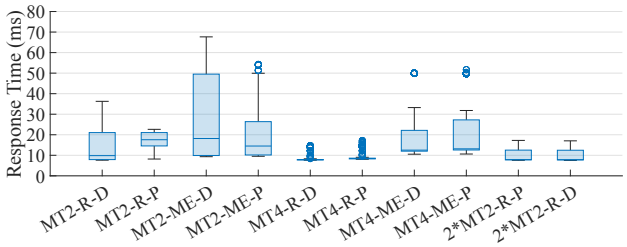
The purpose of this case study is to understand the performance characteristics of the ROS 2 multi-threaded executor and to compare the observed response time against the worst-case bounds obtained by analysis. Fig 4 depicts the chain set used here, which is inspired by an autonomous robotic system. It consists of four chains: Γ_1 ($\pi_1 = 4$; highest priority), Γ_2 ($\pi_2 = 3$), Γ_3 ($\pi_3 = 2$), and Γ_4 ($\pi_4 = 1$; lowest priority). We ran this chain set for 5 minutes on AGX under each executor configuration and took the 99th percentile as the maximum observed response time (MORT) of each chain. For the executor configurations, we considered a multi-threaded executor with m threads, where $m \in \{2, 4\}$. The executor is set to use cores equal to the number of threads they have; hence, $sbf_{\Pi}(\Delta) = m \cdot \Delta$. For each case, we also tested with and without our priority-driven scheduling enhancements and mutually-exclusive callback groups. We also considered cases where the chain set spans across two multi-threaded executors (each with two threads) with and without priority-driven enhancements. The total number of executor configurations is therefore $2 \times 4 + 2 = 10$.

For analytical bounds on the worst-case response time (WCRT), we considered our analyses for constrained-deadline chains on both standard (default) and priority-driven multi-threaded executors, assuming all callbacks are in a reentrant callback group (Theorems 1 and 2). We also examined our analyses for mutually-exclusive callback group (Theorem. 5 and 6) by gathering callbacks of the chains Γ_1 and Γ_2 in a mutually-exclusive group and gathering callbacks of the chains Γ_3 and Γ_4 in a reentrant group. Also, to evaluate our end-to-end latency analysis for chains spanning across multiple executors (Sec. V-F), we assigned callbacks $\{\tau_1, \tau_3, \tau_6, \tau_7, \tau_9, \tau_{10}\}$ to a 2-threaded executor and assigned the rest $\{\tau_2, \tau_4, \tau_5, \tau_8, \tau_{11}, \tau_{12}\}$ to another 2-threaded executors. Since these two 2-threaded executors run on four cores of the same processor, we assumed the propagation delay between two executors is negligibly small.

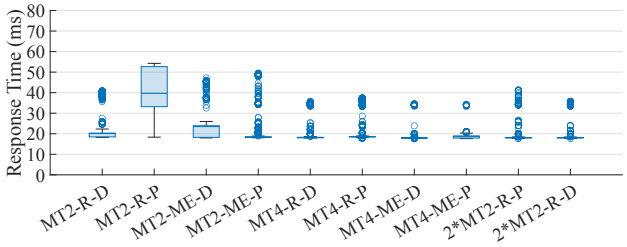
Figs. 5 and 6 compare the MORT and WCRT of each chain under different executor conditions. The caption of each sub-graph represents: the first part "R" and "ME" for the aforementioned settings of reentrant and mutually-exclusive callback groups, respectively; and, the following "D" and "P" for the ROS 2 default and the priority-driven scheduling



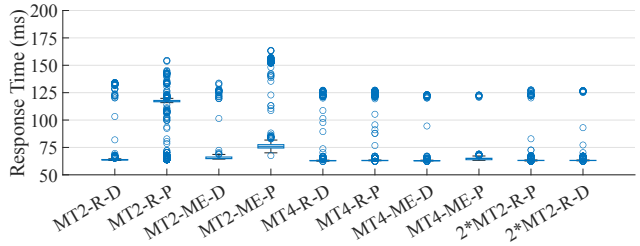
(a) Chain Γ_1



(b) Chain Γ_2



(c) Chain Γ_3



(d) Chain Γ_4

Fig. 7: Observed response time of chains

schemes. The legend denotes: "MT#" for a multi-threaded executor with # threads (e.g., MT4 is a 4-threaded executor); "2*MT2" for the case of two 2-threaded executors.

In all cases, our analysis could safely upper-bound the MORT. The priority-driven scheduling scheme outperforms the default ROS 2 scheduler in both reentrant and mutually-exclusive callback groups, especially for high-priority chains. These results indicate that both MORT and WCRT are reduced for high-priority chains with priority-driven scheduling and this enhancement not only reduces analysis pessimism but also improves the response time of critical chains at runtime. Although the response time of low-priority chains slightly increases with priority-driven scheduling, this is the cost to improve the response time of high-priority ones.

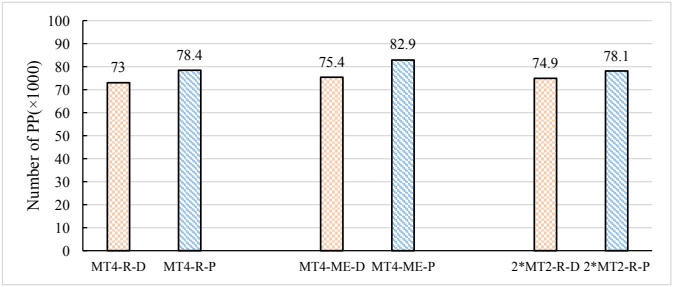


Fig. 8: Overhead w.r.t. the number of polling points (PP)

Fig. 7 illustrates more details on the observed response time distributions of the four chains under various executor configurations. It is clearly shown that the use of priority-driven scheduling reduces variations in the observed response time of critical (high-priority) chains, thereby improving perceived predictability.

To assess the overhead of our priority-driven scheduling scheme, we measured the number of ReadySet updates during 5 minutes of running the case study under various executor configurations. Fig. 8 illustrates the results. As can be seen, the priority-driven scheduling scheme introduces less than 10% of additional ReadySet updates compared to the default scheme.

B. Schedulability Experiments

In this section, we explore the schedulability ratio of the proposed RTA framework using randomly-generated chain sets with constrained and arbitrary deadlines. The following abbreviations are used for each of the analysis approaches:

- **PWA_CD**: Proposed Worst-case Analysis (PWA) for chains with Constrained Deadlines (CD) – Theorem 1
- **PPWA_CD**: Priority-driven PWA for CD – Theorem 2
- **PWA_AD**: PWA for Arbitrary Deadlines (AD) – Theorem 3
- **PPWA_AD**: Priority-driven PWA for AD – Theorem 4

We used the UUniFast algorithm [27] with a total utilization from 0.8 to 4.0 with the step of 0.4 to generate 1,000 random chain sets, where each set includes 5 chains and each chain has 10 callbacks. At first, chains were generated for the constrained-deadline case with $T_C = D_C$ using the UUniFast algorithm. Then, for arbitrary deadlines, we duplicated these chains and doubled their deadlines. The number of threads is set to $m = 4$.

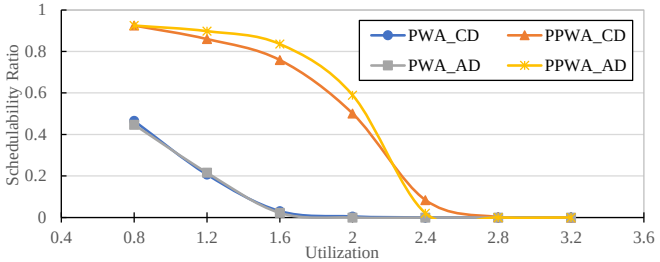


Fig. 9: Varying utilization of chain sets

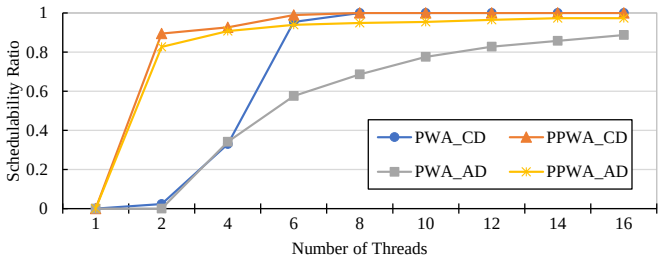


Fig. 10: Varying number of threads

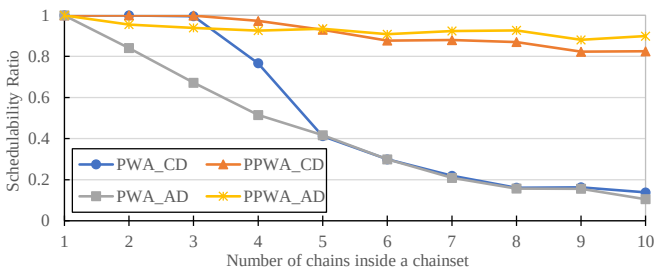


Fig. 11: Varying number of chains

Fig. 9 shows the schedulability ratio of chain sets by our four analysis approaches as the chain-set utilization increases. The schedulability ratio might be seen as rather low for the given utilization. However, it is worth noting that, in chain scheduling, the precedence dependencies among callbacks cause resource waste and therefore lead to a less schedulability ratio than conventional task scheduling with no dependencies. In general, priority-driven scheduling (PPWA_CD and PPWA_AD) significantly outperforms the default ROS 2 scheduling policy (PWA_CD and PWA_AD), with as much as 66% point higher in schedulability. This is primarily due to the fact that high-priority chains experience less delay from lower-priority chains under PPWA since it fetches ready callbacks immediately and schedules them strictly based on their assigned priority. The results of PPWA_CD and PPWA_AD look similar in this figure. However, given that PPWA_AD is tested with chain sets with doubled deadlines, it can be said that PPWA_AD has much higher pessimism than PPWA_CD. This is somewhat expected from our analysis in Theorem 4 which had to take into account multiple outstanding (= released and unfinished) instances of each chain.

We also explored the schedulability ratio of our analyses as the number of threads increases. Similar to above, we generated 1,000 random chain sets, each including 5 chains with 10 callbacks per chain. The utilization of each chainset is fixed at 1.0 in this experiment. The results are shown in Fig. 10. Note that scheduling a chain set with a utilization equal to 1.0 on a single thread is almost infeasible due to the precedence dependencies among callbacks. The schedulability ratio improves with a more number of threads ($m \geq 2$), but the degree of improvement appears differently for each analysis method. For PPWA_CD and PPWA_AD, the schedulability ratio increases sharply at $m = 2$ and becomes almost plateau afterward. On the other hand, the increase of PWA_CD and PWA_AD is slower and is almost linear to the number of threads. We can see that our proposed priority-driven enhancement can achieve better schedulability with a fewer number of threads.

Lastly, we explored the impact of the number of chains on schedulability. Here, we kept the total utilization to 1 and varied only the number of chains. Each generated chain has 10 callbacks, regardless of how many chains are generated for each chain set. Fig. 11 depicts the results. The schedulability ratio decreases with the number of chains for all analysis methods, meaning interference increases with the number of chains although the total utilization remains the same. Recall that, due to precedence dependencies, only one callback of each chain instance can interfere with the chain under analysis. However, by increasing the number of chains, the number of callbacks from chains increases, and also the number of chain instances in a time interval Δ could increase. However, Fig. 11 shows that priority-driven scheduling is less affected by this issue because it limits the source of interference to only higher-priority chains. We therefore conclude that priority-driven scheduling brings a significant benefit to real-time chains on ROS 2 multi-threaded executors.

VII. Conclusion

In this paper, we proposed a comprehensive response-time analysis framework for chains running on ROS 2 multi-threaded executors. We analyzed the timing behavior of the default scheduling scheme of ROS 2 multi-threaded executors and proposed priority-driven scheduling as an improvement over the default one. Our framework can analyze chains with both arbitrary and constrained deadlines and can take into account the effect of mutually-exclusive callback groups. We conducted the evaluation with a case study on NVIDIA Jetson AGX Xavier and schedulability experiments using randomly-generated chains. The results demonstrate that our analysis framework can safely upper-bound response times under various conditions. In addition, our priority-driven scheduling for ROS 2 multi-threaded executors not only reduces the response time of critical chains but also improves analysis results.

As ROS 2 is becoming more popular in academia and industry for more complex robotic systems with safety-critical features, more active research efforts need to be made to ensure timing correctness and improve system efficiency in ROS-based systems. There are several interesting future directions, including real-time support for accelerators, synchronization, memory-induced interference, which have been studied for conventional systems but not in the context of ROS or similar middleware environments. We believe that our work fills an important knowledge gap in this area and more interesting work could be built upon our framework.

Acknowledgment

This work was sponsored by the National Science Foundation (NSF) grant 1943265 and the Office of Naval Research (ONR) grant N00014-19-1-2496.

REFERENCES

[1] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[2] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ROS2 executors," in *IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[3] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New design of priority-driven chain-aware scheduling for ROS2," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[4] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[5] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "ROSCH: real-time scheduling framework for ros," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2018, pp. 52–58.

[6] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao, "RT-ROS: A real-time ros architecture on multi-core processors," *Future Generation Computer Systems*, vol. 56, pp. 171–178, 2016.

[7] 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," *arXiv preprint arXiv:1809.02595*, 2018.

[8] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

[9] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 278–283.

[10] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–10.

[11] J. Abdullah, G. Dai, and W. Yi, "Worst-case cause-effect reaction latency in systems with non-blocking communication," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1625–1630.

[12] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2016, pp. 159–169.

[13] T. Kloda, A. Bertout, and Y. Sorel, "Latency analysis for data chains of real-time periodic tasks," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2018, pp. 360–367.

[14] H. Choi, M. Karimi, and H. Kim, "Chain-based fixed-priority scheduling of loosely-dependent tasks," in *2020 IEEE The 38th IEEE International Conference on Computer Design (ICCD)*.

[15] Y. Tang, N. Guan, Z. Feng, X. Jiang, and W. Yi, "Response time analysis of lazy round robin," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[16] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo, "Response time analysis for dynamic priority scheduling in ros2," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 301–306.

[17] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-to-end timing analysis in ros2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 53–65.

[18] H. Choi, D. Enright, H. Sobhani, Y. Xiang, and H. Kim, "Priority-driven real-time scheduling in ros 2: Potential and challenges," *RAGE 2022*, p. 28, 2022.

[19] Y. Yang and T. Azumi, "Exploring real-time executor on ros 2," in *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2020, pp. 1–8.

[20] X. Jiang, D. Ji, N. Guan, R. Li, Y. Tang, and Y. Wang, "Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 27–39.

[21] D. De Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[22] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–39, 2008.

[23] N. Guan and W. Yi, "General and efficient response time analysis for edf scheduling," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[24] J. Lee and K. G. Shin, "Improvement of real-time multi-coreschedulability with forced non-preemption," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 5, pp. 1233–1243, 2014.

[25] J. Lee, "Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1816–1823, 2017.

[26] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 4, pp. 553–566, 2008.

[27] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.