PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2

Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim
University of California, Riverside
{hchoi036, yxiang013, hyoseung}@ucr.edu

Abstract—**In ROS (Robot Operating System), most applications in time- and safety-critical domain are constructed in the form of callback chains with data dependencies. Due to the shortcomings in its real-time support, ROS does not provide a strong timing guarantee and may lead to disastrous results. Although ROS2 claims to enhance the real-time capability, ensuring predictable end-to-end chain latency still remains a challenging problem. In this paper, we propose a new** *priority-driven chain-aware scheduler* **for the ROS2 framework and present end-to-end latency analysis for the proposed scheduler. With our scheduler, callbacks are prioritized based on the given timing requirements of the corresponding chains so that the end-to-end latency of critical chains can be improved with a predictable bound. The proposed scheduling design includes priority assignment and resource allocation considering all ROS2 scheduling-related abstractions, e.g., callbacks, nodes, and executors. To the best of our knowledge, this is the first work to address the inherent limitations of ROS2 in end-to-end latency by proposing a new scheduler design. We have implemented our scheduler in ROS2 running on NVIDIA Xavier NX. We have conducted case studies and schedulability experiments. The results show that the proposed scheduler yields a substantial improvement in end-to-end latency over the default ROS2 scheduler and the latest work in real-world scenarios.**

I. Introduction

ROS (Robot Operating System) [4] is one of the most popular open-source middleware frameworks for robotic applications. The advent of ROS has revolutionized the developers community both in industry and academia, providing a tremendous number of tools, robot systems, and best-practices to build new applications [3]. Behind such an achievement of productive robotic software developments during a short period of time, the software modularity and composability have played a major role. However, over the decades, ROS has revealed shortcomings in real-time support for timing- and safety-critical applications. This had motivated the development of the second generation of ROS, ROS2, in the community.

ROS2 [6] has been developed since 2017, with major considerations on improving real-time capabilities. Although most concepts are inherited from the original ROS framework, ROS2 employs unique features including inter-node communication through Data Distribution Service (DDS), which is the industry standard for real-time data distribution [2]. The new architecture of ROS2 aims to better support real-time robotic software. However, it still does not guarantee stringent timing constraints and system designers need to empirically tune timing-related parameters.

Guaranteeing the timing constraints of safety-critical applications in ROS2 is a crucial functionality since violating those constraints may cause catastrophic consequences, e.g., a late response from the ROS navigation package for a path following control of a self-driving vehicle may cause a rear-end accident. Also, providing such timing guarantees is challenging due to the following reasons. First, such applications generally form a *chain* which is composed of a set of callbacks with data dependencies. Therefore, the system designer is required to know a safe upper bound on the end-to-end latency of a chain. Second, although many prior studies have proposed analysis techniques for end-to-end chain latency [7, 9, 12, 21], they cannot be directly applied to the ROS2 framework due to its unique scheduling behavior caused by various abstractions including executors and nodes (detailed in Sec. III-B).

To the best of our knowledge, the only recent work on formally analyzing the ROS2 scheduling architecture for real-time guarantees is the study done by Casini et al. [11]. They particularly focused on analyzing the chains of callbacks within an executor, which is one of the core scheduling entities of ROS2. It is pioneering real-time work for ROS2 and it provides an analytical foundation to the research community. However, there are many unresolved issues, e.g., it only focuses on the scheduling behavior within an executor and does not consider the allocation of scheduling entities to executors and CPU cores in a multi-core system. Furthermore, the end-to-end latency analysis suffers from the limitations of the ROS2 scheduling architecture, motivating the development of a new scheduler design.

In this paper, we propose PiCAS, a priority-driven chain-aware scheduler for ROS2 in a multi-core environment. The goal of our work is to minimize end-to-end chain latency by prioritizing the execution of callbacks of chains based on their criticality and timing requirements. We have implemented PiCAS in the ROS2 "Eloquent Elusor" version and evaluated its performance on a real embedded platform. The main contributions of this paper are shown as follows:

- We present the design of PiCAS that includes the priority assignment of callbacks and the allocation of nodes to executors and executors to CPU cores in a multi-core platform. PiCAS makes callbacks and executors execute while respecting the end-to-end timing requirement of chains.
- We develop analysis to upper-bound the end-to-end latency of chains under the proposed PiCAS framework. The analysis provides safe bounds of end-to-end latency of chains.

- We have conducted case studies using practical scenarios on an embedded platform and schedulability experiments using random workloads. Experimental results indicate our proposed scheduler yields a significant improvement in end-to-end latency over the default ROS2 scheduler and the latest analytical work.

The rest of the paper is organized as follows. Sec. II reviews prior work. Sec. III gives the background on ROS2 and our system model used in this paper. Sec. IV describes specific challenges, and Sec. V presents our proposed chain-aware scheduling scheme. The analysis of the end-to-end latency is presented in Sec. VI. In Sec. VII, evaluation results are shown. Finally, Sec. VIII concludes the paper with future work.

II. Related work

Most work on ROS has focused on improving real-time capabilities [26, 27, 29]. In [26], Saito et al. proposed a priority-based message transmission mechanism by allowing publishers to send data based on their priorities. Wei et al. [29] proposed a hybrid OS platform which executes a real-time ROS node (on Nuttx) and a non-real-time ROS node (on Linux) separately by running two operating systems. In [27], ROSCH-G, a loadable kernel framework, is proposed as a real-time extension to ROS with a CPU/GPU coordination mechanism. However, these studies do not provide analytical methods to guarantee real-time timing constraints or are only applied to the first generation of ROS.

For ROS2, Maruyama et al. [23] conducted an empirical evaluation with various QoS configurations under different vendor-specific DDS implementations. In [16], the worst-case latency between two nodes is measured and the deadline miss behavior is observed in a PREEMPT-RT patched Linux kernel system. Both studies evaluated the performance of ROS2 using measurement-based approaches, but did not provide formal modeling or analysis.

Many studies have been conducted on analyzing end-to-end chain latency in a publisher-subscriber model or with read-execute-write semantics. Palencia et al. proposed approaches to analyze tasks with precedence constraints in multi-core systems [24, 25]. In [15, 28], methods to capture an upper bound on the end-to-end latency of tasks are presented based on the worst-case response time. Kloda et al. [21], Abdullah et al. [7], and Becker et al. [9] presented analytical methods to bound the end-to-end latency of a chain under fixed-priority scheduling. The most recent work by Choi et al. [12] proposed chain-based fixed-priority scheduling to improve the end-to-end latency of chains. However, none of these analytical approaches cannot be directly applicable to ROS2 due to differences in the scheduling model.

The literature on the latency of ROS2 processing chains is quite limited. To the best of our knowledge, [11] is the only work on modeling the ROS2 scheduler and providing a response-time analysis of chains. The authors of that paper investigated callback scheduling behavior within an executor and used resource reservation to model the resource availability of a given executor. The end-to-end latency of a chain is analyzed by first computing response time of each sub-chain that consists of callbacks within an executor, and then adding the response times of sub-chains spanning across executors based on the compositional performance analysis (CPA) [18]. Their approach lays the groundwork to analyze the default ROS2 scheduler. However, it remains unanswered how to allocate resources and further improve the end-to-end latency of critical chains.

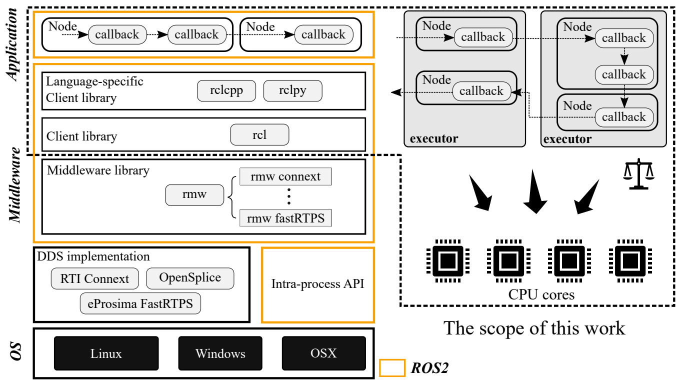


Fig. 1: ROS2 architecture and the scope of our work

III. Background and System Model

A. ROS2 architecture

ROS2 is a unified implementation of multiple layers of abstraction as illustrated in Fig 1. Applications are supported by language-specific client libraries such as C++ and Python officially and many other programming languages from the ROS community. The ROS client library (*rcl*) provides APIs to ensure consistent behavior between programs written in different languages. The ROS middleware library (*rmw*) is a communication interface between *rcl* and the Data Distribution Service (DDS) and is implemented DDS vendor-specific. DDS is an industry standard real-time communication system and is newly added to ROS2 to exchange messages between the publishers and the subscribers of *nodes*.

B. Scheduling-related abstractions

The fundamental scheduling-related abstractions of ROS2 include *callbacks*, *nodes*, and *executors*.

- **Callback** is the minimal schedulable entity in ROS2. There are 5 types of callbacks in ROS2 [11]: timer, subscription, service, client, and waitable callbacks. The timer callback periodically arrives at its own rate, i.e., time-triggered. The others are triggered by external events, i.e., event-triggered. Basically, the transport of messages between publishers and subscribers can be achieved by implementing callback functions in ROS2.
- **Node** is a collection of callback functions, organized by application programmers for modularity and logical partitioning of features. Each node also serves as the minimum allocation unit to executors; hence, all callbacks within the same node are executed by the same executor and they cannot be allocated to two or more executors. In

general, each application is composed of multiple nodes with multiple callbacks per node.
- **Executor** is the OS-level schedulable entity running on CPU cores, i.e., a thread, and executes the callbacks assigned to it. The allocation of callbacks to executors is done through the node abstraction. Once nodes are allocated to an executor, all the callbacks of those nodes are handled by the executor, regardless of the origin of the callbacks. Callback scheduling within an executor is quite different from conventional priority-based real-time task scheduling, as reported in [11]. An executor features two unique behaviors in callback scheduling. First, the priority of callbacks is determined by their types. Timer callbacks always have the highest priority and the others get the next highest priorities in the order presented before. All callbacks are executed non-preemptively. Second, an executor updates the ready status of non-timer callbacks in their respective queues by interacting with the communication middleware layer (*rmw*) shown in Fig. 1. This update occurs when all queues are empty (called a *polling point*), and such a delayed update of callback readiness makes the priority assignment of non-timer callbacks ineffective [11] and lets chains run in a round-robin-like manner.

Chain. On top of the scheduling-related abstractions of ROS2, chains can be constructed by application developers. A chain is a semantic abstraction defined by message exchanges between callbacks of one or more nodes. ROS2 does not define any property on a chain and the executor does not take into account the timing and resource requirements of a chain in callback scheduling. However, since the end-to-end latency of a chain has a major impact on the performance of a safety-critical real-time system, we focus on the scheduling, resource allocation, and analysis of chains in this paper.

Overload handling. ROS2 features an overload handling mechanism in case a timer callback has missed one or more of its periods. The overload handling mechanism occurs at the beginning of timer callback execution (by running `rcl_timer_call` function in the rcl layer). First, the `next_call_time` variable is updated by adding the period of the timer callback to its current value so that the new value indicates the time to trigger the next timer callback. Then, if `next_call_time` is behind the current time, the first step is repeated so that it points to the earliest future time. Hence, missed timer jobs due to overload are skipped naturally and the timer callback can execute in the next future period. In the occurrence of overload, the maximum delay imposed on the end-to-end latency of a time-triggered chain is at most one period of the timer callback, regardless of the number of timer jobs skipped in the past. This is because the release time of a chain instance is effectively determined by the start time of the period where the timer callback job is executed and starts that chain instance. We will make use of this behavior to capture the maximum blocking delay from a prior chain instance in Lemma 4 of Sec. VI. Note that this mechanism is similar to the job-skipping approaches for deadline missed jobs in the literature [13, 14, 17, 22].

C. System model

This paper considers multi-core system where all CPU cores run at the same fixed clock frequency. Below we introduce our model for callbacks, nodes, and executors.

1) Callback model: The system has M real-time callbacks, each is either a timer callback, which is triggered by a periodic timer, or a regular callback, which is triggered by an event from another callback (e.g., the completion of its prior callback in a chain). Each callback has one chain associated with it. A callback τ_i is characterized as follows:

$$\tau_i := (C_i, D_i, T_i, \pi_i)$$

- C_i: The worst-case execution time of a job of τ_i.
- D_i: The relative deadline of a callback τ_i, which is equal to the deadline of its associated chain.
- T_i: The period of a callback τ_i, which is equal to the period of its associated chain ($D_i \leq T_i$).
- π_i: The priority of a callback τ_i within its executor

2) Node model: We use $\mathcal{N}$ to denote a set of nodes as follow:

$$\mathcal{N} =: \{n_1, ..., n_j, ..., n_N\}$$

and the utilization of each node n_j is given by:

$$U(n_j) = \sum_{\forall t_i : \tau_i \in n_j} \frac{C_i}{T_i}$$

It is worth noting that nodes do not have priorities since they are not schedulable entities. They only impose restrictions on callback-to-executor allocation, e.g., callbacks in the same node cannot be separately allocated to two or more executors.

3) Executor model: We denote a set of executors as below:

$$\mathcal{E} =: \{e_1, ..., e_j, ..., e_E\}$$

The priority of j-th executor is denoted by π_{e_j} and $\mathcal{E}$ is sorted in descending order of priority, i.e., $\pi_{e_j} > \pi_{e_{j+1}}$. From a real-time perspective, the scheduling of executors by an OS has a large impact on the timing behavior of callbacks. In this work, we allocate each executor to one CPU core, and schedule the executors of each core with `SCHED_FIFO`, which is the fixed-priority preemptive real-time scheduling policy in Linux with a priority range from `1` to `99`. Therefore, the maximum number of executors with unique priority, E, is limited to `99`.

4) Chain model: Each chain consists of one or more callbacks. A chain Γ^c is denoted as below:

$$\Gamma^c := [\tau_s, \tau_{m1}, \tau_{m2}, ..., \tau_e]$$

- τ_s: The start callback a chain Γ^c.
- τ_{m*}: The intermediate callbacks of a chain Γ^c.
- τ_e: The end callback of a chain Γ^c.

The priority of a chain Γ^c is denoted by π_{Γ^c} and the superscript c is the identifier of the chain Γ^c. This model has been widely used in prior work to analyze the end-to-end latency of a chain with inter-dependency among tasks. Following the time-triggered ROS2 chain model used in [11], the start callback of a chain is assumed to be a timer callback and the others

TABLE I: Chain set

	Chains
Chain 1	$\Gamma^1 =: [\tau_1, \tau_2, \tau_3]$
Chain 2	$\Gamma^2 =: [\tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}]$

	Specifications [sec]
Timer callback (τ_1 and τ_4)	$C_i = 0.109, T_i = 1$
Regular callbacks	$C_i = 0.131$

TABLE II: End-to-end latency results [sec]

Single executor	Mean	Max	Min	STD
Chain 1	36.865	72.752	0.505	21.223
Chain 2	36.730	73.149	0.773	21.154

Executor per chain	Mean	Max	Min	STD
Chain 1	0.370	0.392	0.366	0.004
Chain 2	48.795	97.783	0.772	28.304

be regular callbacks. In case of event-triggered chains, as discussed in [11], one can model the first regular callback of a chain (triggered by an external event) as a timer callback with the event's arrival period. Note that a periodic real-time task in conventional task models can be represented as a single timer callback chain in our model.

We use $\mathcal{C}_{\Gamma^c}$ to denote the total WCET (worst-case execution time) of a chain Γ^c:

$$\mathcal{C}_{\Gamma^c} = \sum_{\forall i : \tau_i \in \Gamma^c} C_i$$

Implications of chain priority. We assume that the priority of a chain, π_{Γ^c}, is given by the system designer based on its criticality or importance in the system. We call it *semantic priority* since it is not part of the original ROS2 framework but should govern the priority assignment of callbacks and executors in order to satisfy application-level requirements.

IV. Challenges

In this section, we elaborate on the challenges of the current ROS2 framework based on the scheduling behavior we have observed from experiments using an embedded platform. The experiments were conducted using the ROS2 "Eloquent Elusor" version running on NVIDIA Xavier NX. Two safety-critical chains consisting of 10 callbacks are used as described in Table I. We assume that chain 1 is more critical than chain 2, i.e., $\pi_{\Gamma^1} > \pi_{\Gamma^2}$. τ_1 and τ_4 are timer callbacks with a period of 1 sec and all other callbacks are regular callbacks. With the assumption of a uniprocessor system, the following two cases are performed: (1) all callbacks are in a single executor, and (2) callbacks of each chain are allocated to a separate executor, i.e., one executor per chain.

Now, we discuss the following two major challenges observed from our experiments.

Challenge 1. Fairness-based scheduling within executors. The unique scheduling behavior of an executor discussed in Sec. III-B is that it schedules timer callbacks always first and makes the priority assignment of other regular callbacks ineffective. Besides, since the ROS2 scheduler do not distinguish callbacks by their chains, all callbacks are scheduled without the notion of chain-level timing requirements in mind. Fig. 2a shows the scheduling timeline of callbacks within a single executor when two chains are run together. As we can observe, ROS2 gives fair progress of both chains over time, and such fairness-based scheduling may jeopardize the timeliness of safety-critical chains, resulting in extremely high latency as depicted in Table II.

Challenge 2. Priority assignment for executors. By default, executors are scheduled by the Completely Fair Scheduler (CFS) [30] of the Linux kernel. Under this scheduler, it is hard to predictably prioritize the executors that are running callbacks from critical chains. Fortunately, the system developers can manually assign OS-level priority to executors although ROS2 does not provide an official interface to configure it. However, there exist no general guideline on the priority assignment of ROS2 executors. The most intuitive way is to assign the highest priority to an executor running the most critical chain, as we have done in Fig. 2b. Here, the executor containing all callbacks of chain 1 has the real-time priority of 99 with the `SCHED_FIFO` policy and the other executor of chain 2 has 98. However, this method does not resolve the problem of undesirably high latency for chain 2, shown in Table II, which happens due to self-interference between the instances of chain 2 itself.

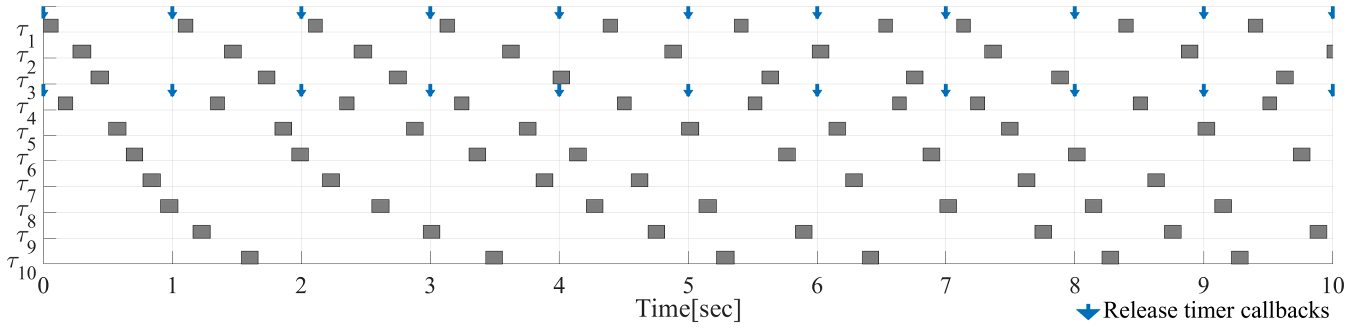
Note that the priority assignment for executors becomes more challenging if callbacks from chains with different criticalities are mixed in a single node because such callbacks cannot be separately allocated to different executors.

V. Priority-driven Chain-aware Scheduling

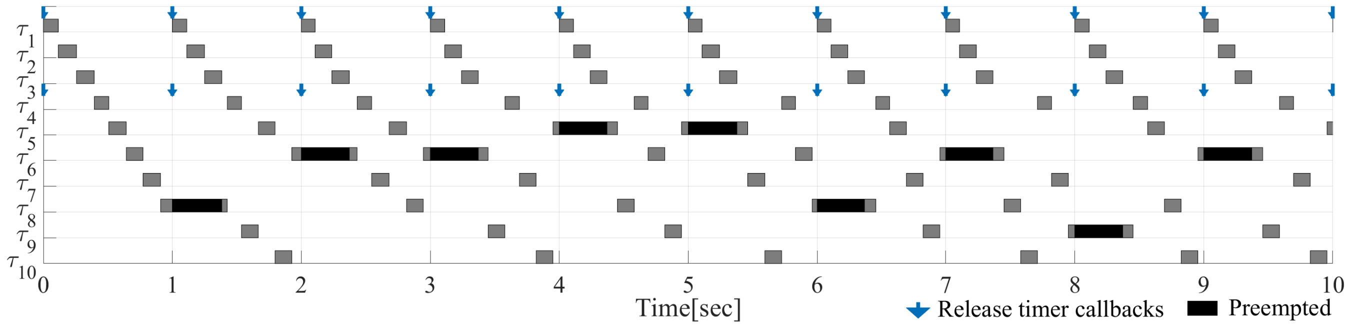
This section presents our priority-driven chain-aware scheduling framework for ROS2, called PiCAS. To improve the end-to-end latency of a chain, the current ROS2 scheduling architecture could be re-designed with the following considerations: (1) higher-priority chains should execute earlier than lower-priority chains (Fig 2a), and (2) for each chain, a prior instance of the chain should complete its execution before the newly released instance starts execution if they are scheduled on the same CPU (Fig 2b). The latter is to reduce self-interference between instances of the same chain, thereby preventing undesirable latency increases. Based on the above considerations, we state the desired properties for our scheduler in the following lemma.

Lemma 1. *For the chain $\Gamma^c := [\tau_1, ...\tau_i, ..., \tau_j, ..., \tau_N]$ whose callbacks are on the same CPU, a prior chain instance is guaranteed to complete its execution before a new instance starts execution if the following conditions are met: (1) τ_j has a higher callback priority than τ_i ($j > i$), and (2) τ_j runs on an executor with the same or higher priority than τ_i's executor.*

Proof: The proof can be done by contradiction. Suppose that a new chain instance starts execution before its prior instance completes, i.e., τ_1 executes before the completion of τ_j ($1 < j \leq N$) on the same CPU. Such behavior occurs in the following cases: (1) τ_1 has a higher priority than τ_j when they are on the same executor, (2) τ_1 has a lower priority than



(a) Scheduling with a single executor



(b) Scheduling with two executors (one executor per chain)

Fig. 2: ROS2 scheduling example in a uniprocessor environment

τ_j but is assigned to a higher-priority executor than τ_j's, and (3) τ_1 has a higher priority than τ_j and belongs to a higher-priority executor than τ_j's. These contradict at least one of the conditions. Hence, the proof is done. ■

Note that, if the callbacks of a given chain Γ^c are distributed across multiple CPU cores, a new instance of that chain may have a chance to start earlier than the completion of its prior instance. However, Lemma 1 still applies to the consecutive subset of Γ^c assigned to one CPU core. The conditions of the lemma ensure that the corresponding subset of instances are executed in their arrival order, and thus, any instance of Γ^c does not get interference from its future instances.

Based on Lemma 1, the rest of this section first outlines our scheduling strategies considering two aspects: chains running within an executor (Sec. V-A) and across executors (Sec. V-B). It then presents the proposed callback priority assignment (Sec. V-C) and chain-aware node allocation (Sec. V-D) algorithms that substantialize these strategies.

	Regular callbacks only	**Timer and Regular callbacks**
Single Chain	Regular callbacks from the same chain	Timer- and regular callbacks from the same chain
Multiple Chains	Regular callbacks from multiple chains	Timer- and regular callbacks from multiple chains

Fig. 3: Chain/callback classification for scheduling purposes

A. Strategies for chains running within an executor

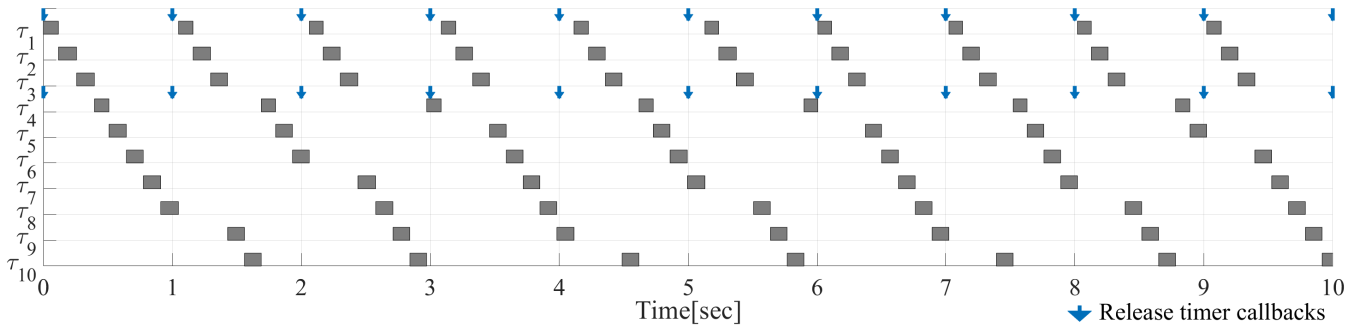
We first describe our strategies for chains running within an executor. These strategies affect how the callbacks of such chains are scheduled in an executor, and they are derived from our classification of chains and callbacks shown in Fig. 3, with each strategy mapped to one of the categories. It is worth noting that callback scheduling is orthogonal to node configurations (given by program code) and node-to-executor allocations (determined by our algorithm in Sec. V-D). This is because, once the callbacks of nodes are assigned to executors, each executor handles assigned callbacks regardless of their origin nodes.

Strategy I. Regular callbacks from a single chain. If an executor has only regular callbacks from a single chain $\Gamma^c =: [\tau_1, ..., \tau_i, ..., \tau_j, ..., \tau_N]$, the priorities of those callbacks should be assigned in the reverse order of their sequence in the chain, i.e., τ_j $(j > i)$ gets a higher priority than τ_i, in order to satisfy the first condition of Lemma 1.

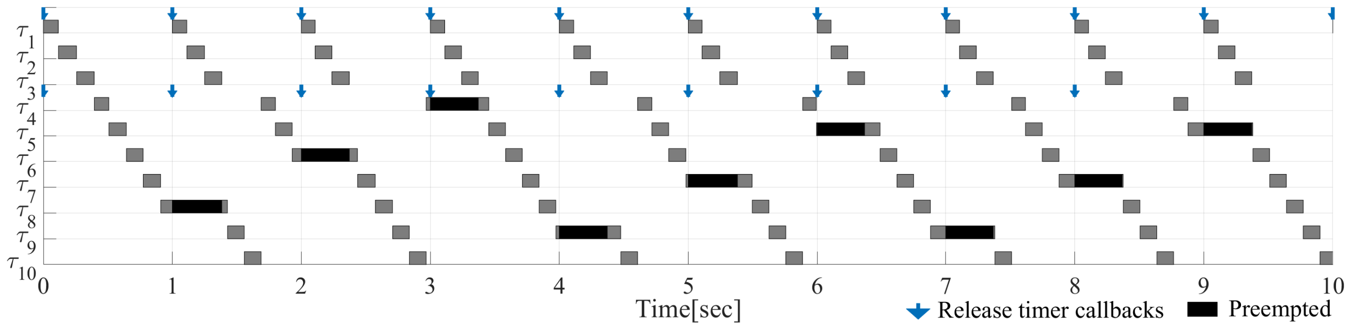
Strategy II. Timer and regular callbacks from a single chain. If an executor contains both timer and regular callbacks of a single chain Γ^c, the regular callbacks should be given higher priorities than the timer callback to satisfy the first condition of Lemma 1 (because the timer is placed before all other regular callbacks in the chain). The scheduling of the regular callbacks should follow Strategy I.

Strategy III. Regular callbacks from multiple chains. Consider two chains, Γ^c and $\Gamma^{c'}$, where $\pi_{\Gamma^c} < \pi_{\Gamma^{c'}}$ (i.e., $\Gamma^{c'}$ has a higher semantic priority than Γ^c). If an executor contains regular callbacks from both Γ^c and $\Gamma^{c'}$, all callbacks of $\Gamma^{c'}$ should be assigned higher priorities than those of Γ^c. In addition, the priority assignment of callbacks for each chain should follow Strategy I individually.

Strategy IV. Timer and regular callbacks from multiple



(a) Scheduling with a single executor



(b) Scheduling with two executors (one executor per chain)

Fig. 4: Execution timeline with the proposed scheduling framework

chains. If an executor contains both timer and regular callbacks from multiple chains (e.g., Γ^c and $\Gamma^{c'}$ where $\pi_{\Gamma^c} < \pi_{\Gamma^{c'}}$), the timer callback from a higher semantic priority chain ($\Gamma^{c'}$) should have a higher priority than that from a lower semantic priority chain (Γ^c). Since the timer callback is the starting point of each chain instance, such a prioritization ensures more critical chain instances to precede the less critical ones. Then, each chain should follow Strategy II individually to conform to Lemma 1.

B. *Strategies for chains running across executors*

Next we discuss scheduling strategies for chains running across multiple executors. Since each executor is allocated to one CPU core and scheduled by the OS's preemptive fixed-priority scheduler, we need to consider chain scheduling on each CPU that may have multiple executors assigned to it. The following two strategies will serve as the basis for our allocation algorithm in Sec. V-D. Note that executors here are assumed to follow Strategies I to IV given in Sec V-A.

Strategy V. A single chain on one CPU. Suppose a CPU has callbacks from only a single chain Γ^c. In this case, the executor containing the lower-index callbacks τ_i of Γ^c should have the same or lower priority than the other executors on the same CPU that execute the higher-index callbacks τ_j $(j > i)$ of Γ^c. This is to satisfy the second condition of Lemma 1.

Strategy VI. Multiple chains on one CPU. Suppose a CPU has callbacks from multiple chains (e.g., Γ^c and $\Gamma^{c'}$ on the same CPU where $\pi_{\Gamma^c} < \pi_{\Gamma^{c'}}$). In this case, the executor that contains the callbacks of $\Gamma^{c'}$ should have at least the same or higher priority than those including the callbacks of Γ^c, in order to respect the semantic priority of chains.

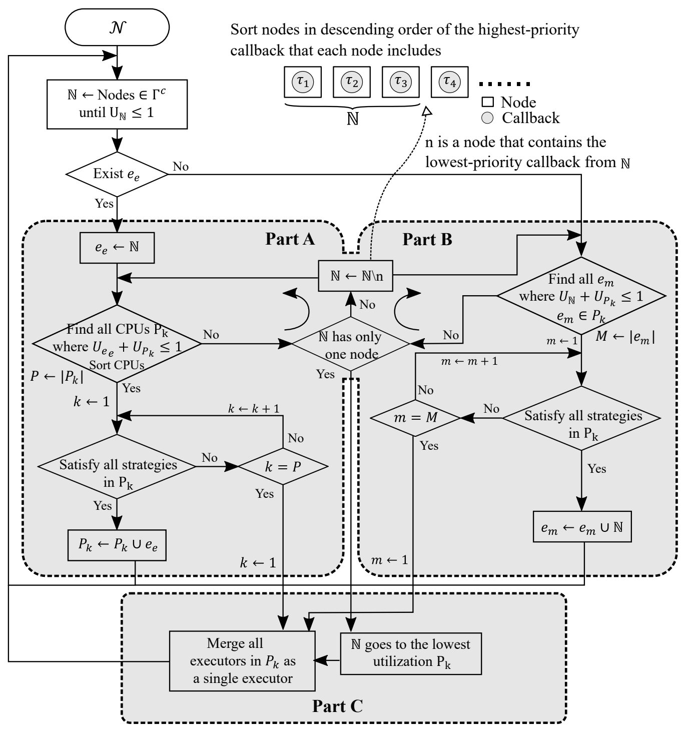
C. *Priority assignment of callbacks*

To realize the aforementioned scheduling strategies, we first propose a callback priority assignment algorithm in this subsection. Our algorithm enables callbacks within each executor to implement Strategies I to IV and is given in Alg. 1.

Algorithm 1 Callback priority assignment

```
Input: Γ: chains
1: $\Gamma \leftarrow$ sort in ascending order of semantic priority $\pi_\Gamma$
2: $p \leftarrow 1$                          ▷ Initialize current priority
3: for all $\Gamma^c \in \Gamma$ do
4:     for all $\tau_i \in \Gamma^c$ do
5:         $\tau_i \leftarrow p$
6:         $p \leftarrow p + 1$
7:     end for
8: end for
```

The proposed algorithm takes a hierarchical approach to assign priorities to callbacks, considering first the semantic priorities of their origin chains and then the relative priority ordering within each chain. In Alg. 1, chains are sorted in ascending order of their semantic priorities (line 1) and the outer loop iterates starting from the lowest semantic priority chain (line 3). According to our chain model in Sec. III-C, the callbacks of each chain are already arranged from the start (low index) callback to the end (high index) callback. Hence, the inner loop assigns a lower priority to a callback with a lower index for each chain Γ^c (line 4 to line 7), i.e., the timer callback gets the lowest priority in the chain and the end callback gets the highest priority.


$\mathcal{N}$
Sort nodes in descending order of the highest-priority callback that each node includes
τ_1
τ_2
τ_3
τ_4
$\mathbb{N}$
Node
Callback
n is a node that contains the lowest-priority callback from $\mathbb{N}$
$\mathbb{N} \leftarrow$ Nodes $\in \Gamma^c$ until $U_{\mathbb{N}} \leq 1$
Exist e_e
No
Yes
Part A
Part B
$e_e \leftarrow \mathbb{N}$
$\mathbb{N} \leftarrow \mathbb{N} \backslash n$
Find all e_m where $U_{\mathbb{N}} + U_{P_k} \leq 1$ $e_m \in P_k$
Find all CPUs P_k where $U_{e_e} + U_{P_k} \leq 1$ Sort CPUs
$\mathbb{N}$ has only one node
$P \leftarrow |P_k|$
$k \leftarrow 1$
$k \leftarrow k+1$
$m \leftarrow 1$
$M \leftarrow |e_m|$
$m \leftarrow m+1$
Satisfy all strategies in P_k
$k = P$
$m = M$
$P_k \leftarrow P_k \cup e_e$
$e_m \leftarrow e_m \cup \mathbb{N}$
Merge all executors in P_k as a single executor
$\mathbb{N}$ goes to the lowest utilization P_k
Part C

	Parameters		
$\mathcal{N}$	Nodes	e_m	Non-empty executors
$\mathbb{N}$	A node set consists of callbacks of a chain Γ^c ($U_{\mathbb{N}} \leq 1$)	M	The number of e_m
e_e	Empty executor	P	The number of P_k
U_{P_k}	Utilization of CPU core P_k		
n	A node that has the lowest priority callback of Γ^c in $\mathbb{N}$		

Fig. 5: Diagram of the proposed node allocation scheme

D. Chain-aware node allocation scheme

This subsection presents our chain-aware node allocation scheme for ROS2. The proposed scheme allocates given nodes to executors, and then maps these executors to available CPU cores while following the scheduling strategies mentioned before. This scheme also tries to minimize interference between chains by allocating all nodes associated with one chain to the same CPU core whenever possible. The allocation scheme is done offline and does not introduce runtime overhead. It is worth noting that nodes cannot be split arbitrarily by a resource-allocation algorithm because nodes are composed by programmers for modularity and logical partitioning of features in ROS2.[1] In addition, data-dependency relationship between nodes is unaffected by the node allocation since communication between nodes is done explicitly by messages and does not change regardless of which executor they use.

Fig. 5 illustrates the flow diagram of the proposed node allocation scheme. The scheme takes as input the maximum number of executors to use (M), the number of available CPU cores (P), and the set of nodes to be allocated ($\mathcal{N}$). It first sorts nodes in $\mathcal{N}$ in descending order of the highest-priority callback that each node includes. This means that nodes associated with a higher semantic priority chain are allocated first by the scheme. Let us use Γ^c to denote the highest semantic priority chain that has not been allocated yet (so its associated nodes are in $\mathcal{N}$). The scheme selects a subset of nodes, $\mathbb{N}$, by fetching nodes from $\mathcal{N}$ one at a time until all the nodes associated with Γ^c are fetched or the utilization of $\mathbb{N}$ exceeds 1. Then the scheme checks if there is an empty executor e_e and moves on to the actual allocation phase consisting of three parts. Part A allocates $\mathbb{N}$ to e_e and e_e to a feasible CPU core. Part B is the case where e_e does not exist; it finds a non-empty executor e_m that is feasible for $\mathbb{N}$. Part C handles all leftover nodes that were not allocated to executors by the first two parts. Details on each part are given below.

Part A. This part of the scheme begins when there is an empty executor e_e. It assigns $\mathbb{N}$ to e_e, and finds all feasible CPU cores P_k whose utilization including e_e does not exceed 1, i.e., $U_{e_e} + U_{P_k} \leq 1$. If such a CPU core is not found, the scheme removes a node n that contains the lowest-priority callback from $\mathbb{N}$, sends n back to $\mathcal{N}$ (so that it can be reconsidered), and finds feasible CPUs for $\mathbb{N}$ again until there remains one node in $\mathbb{N}$. If $\mathbb{N}$ has only one node and no feasible CPU is found, it is sent to Part C. Otherwise, among all feasible CPUs found, $\mathbb{N}$ is assigned to the one that has the lowest utilization and satisfies Strategies V and VI. If none satisfies those strategies, $\mathbb{N}$ will be handled Part C. Note that each executor has a unique OS-level real-time priority in the range from 1 to 99 under the `SCHED_FIFO` policy, and we use the highest-priority empty executor first.

Part B. In this part, $\mathbb{N}$ is allocated to a non-empty executor e_m that has been already assigned to a CPU core P_k, i.e., $e_e \in P_k$. Similar to Part A, the scheme finds all feasible non-empty executors e_m where $U_{\mathbb{N}} + U_{P_k} \leq 1$. If no feasible executor exists, the scheme extracts a node n from $\mathbb{N}$ and search for available executors iteratively until only one node remains in $\mathbb{N}$. If $\mathbb{N}$ has only one node and cannot find any feasible executor, this node will be handled by Part C. $\mathbb{N}$ is assigned to the executor that has the lowest utilization and satisfies Strategies I to VI. When no executor meets the strategies, $\mathbb{N}$ will be handled by Part C.

Part C. This part handles $\mathbb{N}$ that could not be allocated to executors or CPUs by Parts A and B. There are two reasons why $\mathbb{N}$ could not be allocated. First, a feasible CPU core P_k was found but Strategies V and VI were not satisfied. In this case, the scheme merges all executors on P_k into a single executor so that the two strategies are trivially satisfied. Secondly, all CPU cores have a utilization higher than 1. Since ROS2 can handle overloaded cases based on the mechanism described in Sec. III-B, we do allocate $\mathbb{N}$ to the CPU core whose utilization is the lowest. We will assess the scheduling performance of PiCAS and the default ROS2 scheduler for overloaded cases in Sec. VII.

E. Example of priority-driven chain-aware scheduling

Recall the chain set in Table I which results in high end-to-end latency under ROS2's default scheduling, as discussed in Sec III. Now, we re-run the chain set under the proposed PiCAS framework. As illustrated in Fig 4, our scheduler executes

[1]To split a single node and allocate it to two or more CPUs, the node's software code needs to be rewritten, which is beyond the scope of this work.

TABLE III: End-to-end latency results with PiCAS [sec]

Single executor	Mean	Max	Min	STD
Chain 1	0.436	0.506	0.368	0.038
Chain 2	1.196	1.738	0.741	0.348
Executor per chain	**Mean**	**Max**	**Min**	**STD**
Chain 1	0.369	0.394	0.366	0.004
Chain 2	1.255	1.731	0.737	0.352

callbacks of a higher semantic priority chain first. Besides, in accordance with Lemma 1, the prior chain instance of each chain completes execution before starting the execution of its new chain instance in a uniprocessor environment. Table III depicts the end-to-end latency for the following two cases: (1) all callbacks are in a single executor, and (2) callbacks of each chain are allocated to a separate executor. For both cases, we have observed that PiCAS significantly improves the latency of both chains.

VI. Analysis of End-to-end Latency

This section presents the end-to-end latency analysis of a chain under the proposed chain-aware scheduling for ROS2. For the analysis purpose, a consecutive subset of a chain Γ^c on one CPU core (called a segment Φ_i) can be considered as a single artificial callback. This is because, once the timer callback of Γ^c arrives based on its period, the rest callbacks are triggered directly by the completion of their prior callbacks and there is no self-induced delay between them. If a chain executes over multiple CPU cores, it can be decomposed into multiple segments. Therefore, the end-to-end analysis of a chain can be done by two steps: (i) computing the worst-case response time (WCRT) of each segments of a chain, and (ii) adding up the WCRTs of all segments of the given chain. Fig. 6 illustrates an example of a chain with three segments. We will present each step and explain how the analysis is done.

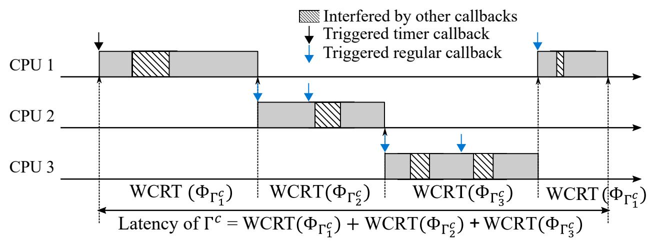

▨ Interfered by other callbacks
Triggered timer callback
Triggered regular callback
CPU 1
CPU 2
CPU 3
WCRT $(\Phi_{\Gamma^c_1})$
WCRT$(\Phi_{\Gamma^c_2})$
WCRT$(\Phi_{\Gamma^c_3})$
WCRT $(\Phi_{\Gamma^c_1})$
Latency of Γ^c = WCRT$(\Phi_{\Gamma^c_1})$ + WCRT$(\Phi_{\Gamma^c_2})$ + WCRT$(\Phi_{\Gamma^c_3})$

Fig. 6: Latency analysis of a chain in a multi-core system

A. *Worst-case response time of a segment Φ_i*

At first, we compute the WCRT of each segment of a chain. A segment Φ_i is considered as a single artificial callback whose priority is inherited from the lowest-priority callback of Φ_i, and its WCET is the sum of all callbacks' WCET in that segment.

Interference from higher semantic priority chains. In order to analyze the WCRT of a target segment Φ_i, we need to upper bound the maximum interference imposed by other callbacks. In our framework, such callbacks run in higher-priority executors than Φ_i's executor on the same CPU, or are higher-priority callbacks in the same executor as Φ_i. Considering these, the following lemma captures the maximum number of arrivals of an interfering callback τ_k for an arbitrary time interval t.

Lemma 2. *The maximum number of arrivals of a callback $\tau_k \in \Gamma^{c'}$ that causes interference to a target segment $\Phi_i \subset \Gamma^c$ during an arbitrary time window t is bounded by:*

$$\eta_i(t, \tau_k) = \begin{cases} \left\lceil \dfrac{t}{\mathcal{T}_{\Gamma^{c'}}} \right\rceil & \text{, } \textit{if } \pi_{\Gamma^{c'}} > \pi_{\Gamma^c} \wedge P(\Phi_i) = P(\tau_k) \\ 0 & \text{, } \textit{otherwise} \end{cases} \qquad (1)$$

where $\mathcal{T}_{\Gamma^{c'}} = \max(T_{\Gamma^{c'}}, \sum_{\tau_j \in \Gamma^{c'}} C_j)$, if all callbacks of $\Gamma^{c'}$ are on the same CPU, and $\mathcal{T}_{\Gamma^{c'}} = T_{\Gamma^{c'}}$, otherwise; $T_{\Gamma^{c'}}$ is the period of $\Gamma^{c'}$; and $P(\tau_k)$ is τ_k's CPU core.

Proof: The proof can be done by considering the following two cases. The first case is that the chain of τ_k has a higher semantic priority than the chain of Φ_i ($\pi_{\Gamma^{c'}} > \pi_{\Gamma^c}$) and the interfering task τ_k is on the same CPU as Φ_i ($P(\Phi) = P(\tau_k)$). In this case, τ_k can interfere with Φ_i whenever it arrives. If the system is not overloaded, τ_k can arrive every $T_{\Gamma^{c'}}$. On the other hand, if the system is overloaded and all callbacks of $\Gamma^{c'}$ are on the same CPU, the arrival interval of τ_k cannot be shorter than the sum of the execution times of all callbacks in $\Gamma^{c'}$ because its prior chain instance always completes before a new instance starts, as proved in Lemma 1. Note that the two conditions of this lemma are satisfied by our callback priority assignment and node allocation schemes. Therefore, $\mathcal{T}_{\Gamma^{c'}}$ takes the maximum of these two to bound τ_k's arrival period.

The second case includes all the other conditions that are not considered by the first case. (i) If $\pi_{\Gamma^{c'}} > \pi_{\Gamma^c}$ but τ_k is on a different core, τ_k obviously does not cause any interference to Φ_i. (ii) If $\pi_{\Gamma^{c'}} = \pi_{\Gamma^c}$, it means τ_k and Φ_i are from the same chain and they do not interfere with each other by Lemma 1. (iii) If $\pi_{\Gamma^{c'}} < \pi_{\Gamma^c}$, one thing we need to consider is whether τ_k can be in a higher-priority executor than Φ_i's executor. However, by Strategy VI, τ_k is allocated to the same or lower priority executor than Φ_i. If τ_k is in a lower-priority executor, obviously no interference occurs. If τ_k is in the same executor, it gets lower priority than Φ_i by Alg. 1 and cannot cause interference delay to Φ_i; τ_k can cause *blocking* delay in this case, which is captured separately using a different term. ■

Blocking time from lower priority callback. Due to the non-preemptive scheduling nature within an executor, a lower-priority callback can cause blocking delay at most once to the target segment Φ_i. The maximum blocking time B_i for Φ_i is therefore bounded by the longest execution time of a lower-priority callback τ_j in the same executor and is given by:

$$B_i = \max_{\substack{\forall j: \tau_j \in e(\Phi_i) \wedge \tau_j \notin \Gamma^c \\ \wedge \pi_j < \pi_i}} \{C_j\} \qquad (2)$$

where $e(\Phi_i)$ is the executor of Φ_i.

Based on Lemma 1 and Eq. 2, the worst-case response time of a segment Φ_i can be calculated as follow:

Lemma 3. *The worst-case response time of a segment $\Phi_i \subset$*

Γ^c*, denoted by* $R^n_{c,i}$*, is bounded by the following recurrence:*

$$R^{n+1}_{c,i} \leftarrow B_i + \sum_{\forall j:\tau_j \in \Phi_i} C_j + \sum_{\substack{\forall k:\tau_k \in e(\Phi_i)\vee \\ \tau_k \in e_{HP}}} \eta_i(R^n_{c,i}, \tau_k) \times C_k \tag{3}$$

where e_{HP} *is a set of executors with higher priority than the executor of* Φ_i*. The recurrence starts with* $R^0_{c,i} = B_i + \sum_{\forall j:\tau_j \in \Phi_i} C_j$*.*

Proof: It is obvious from Lemma 1 and Equation 2. ■

Note that Lemma 3 captures the WCRT of a segment Φ_i that is a subset of the chain Γ^c. Hence, it can also be used to compute the WCRT of just a single callback τ_j, by setting $\Phi_i = \{\tau_j\}$. In this case, the computed WCRT of τ_j represents the time from when τ_j is triggered to when it completes its execution.

B. End-to-end latency of a chain Γ^c

Finally, we analyze the end-to-end latency of a given chain in the proposed chain-aware scheduling framework.

Delay from prior chain instance. Recall the scheduling behavior of chain instances when the conditions of Lemma 1 are satisfied. Since the next released chain instance can start its execution only after the completion of its prior chain instance, the next instance may experience an additional blocking delay from the prior chain instance.

Lemma 4. *The maximum blocking delay caused by a prior instance of the chain* Γ^c*, denoted by* $S(\Gamma^c)$*, is given by:*

$$S(\Gamma^c) = \begin{cases} 0 & \text{, if } \sum_{\Phi_i \subset \Gamma^c} R^n_{c,i} \le T_{\Gamma^c} \\ T_{\tau_s} & \text{, otherwise} \end{cases} \tag{4}$$

where T_{Γ^c} *is the period of* Γ^c*.*

Proof: If the cumulative WCRT of $\Phi_i \subset \Gamma^c$ is less than or equal to the period of the chain, no blocking delay occurs for the next chain instance since the previous instance finishes before the next one is released. Otherwise, the next instance can be delayed at most one cycle of the chain's period by the overload handling mechanism of ROS2 as explained in Sec III-B. Thus, the proof is done. ■

Theorem 1. *The end-to-end latency of a chain* Γ^c *is computed by:*

$$L_{\Gamma^c} \leftarrow \sum_{\Phi_i \subset \Gamma^c} R^n_{c,i} + S(\Gamma^c) \tag{5}$$

where Φ_i *is a segment of the chain* Γ^c*.*

Proof: The proof is straightforward. Since a chain is composed of callbacks that are triggered by the completion of their prior callbacks, the end-to-end latency of the chain is obtained by summing up the WCRT of all segments of the chain. Besides, we consider the blocking delay $S(\Gamma^c)$ that can be caused by a prior chain instance. ■

VII. Evaluation

This section evaluates our proposed PiCAS framework by comparing it with the default ROS2 scheduler and the state-of-the-art analysis work. We first conduct case studies on a real platform running self-driving software to identify the practical effectiveness of PiCAS. We then perform schedulability experiments using randomly-generated workloads to explore the performance characteristics of PiCAS and the default ROS2 scheduler.

A. Implementation

We have implemented the proposed scheduler in the Eloquent Elusor version of ROS2 running on Ubuntu 18.04 in an NVIDIA Xavier NX platform which is equipped with a six-core ARM®v8.2 1.4GHz processor. Since the ROS2 scheduler features a unique callback scheduling policy discussed in Sec. III-B, our implementation mainly modifies the callback scheduling policy: (i) the condition for updating ready callbacks in the executor's queues, and (ii) the priority assignment for individual callbacks. In ROS2, an executor (*rclcpp* in Fig. 1) interacts with the communication layer (*rmw* in Fig. 1) to fetch ready callbacks onto its queues. Our scheduler implementation updates those queues whenever a callback completes. If one or more callbacks are ready in the queues, our scheduler choose to execute the one with the highest priority determined by the proposed priority assignment (Alg. 1), instead of following ROS2's default assignment.

Note that the above modifications can be made without a large software endeavor so that the ROS developer and research communities can access and evaluate it easily.

B. Case study

The case study is organized into three parts. The first part focuses on a simple uniprocessor system for comparative evaluation with the state-of-the-art analysis work. The second and third parts evaluate more practical and complex multi-core systems, considering non-overload and overloaded situations.

Comparison of approaches. We compare our work with two other existing approaches. The first one is the ROS2 default scheduler, which runs executors with the Linux's default scheduling policy. This is a baseline but is not directly analyzable. The second one is the state-of-the-art analysis work [11], which also uses the ROS2 default scheduler but assumes that each executor has a resource reservation. In summary, the following three methods are compared:

- **ROS2**: ROS2 default scheduler with no analysis.
- **ROS2-SD**: ROS2 default scheduler with resource reservation and the worst-case response time analysis [11].
- **ROS2-PiCAS**: Our Priority-driven Chain-Aware Scheduler with the proposed end-to-end analysis.

Resource reservation for ROS2-SD can be implemented using the `SCHED_DEADLINE` policy in the Linux system. We also used the source code provided in [5] by the authors of [11] for the analysis of ROS2-SD. Since the analysis of ROS2-SD in [5] does not support testing an overloaded system, we only apply the ROS2-SD analysis to non-overloaded setups. For analysis purposes, we measured the execution time of each callback in isolation from 5,000 runs on our embedded platform and chose the maximum observed one as the WCET of that callback.

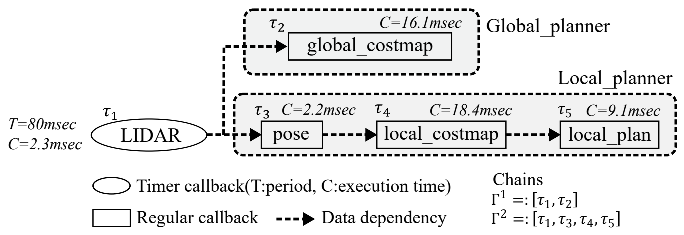
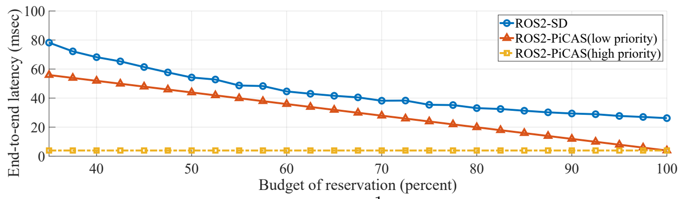
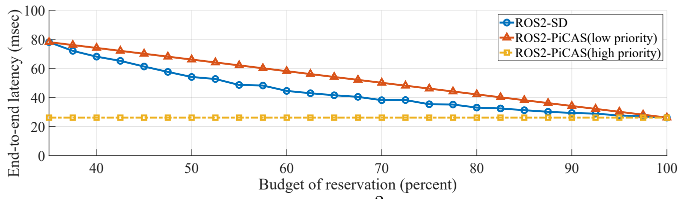


Fig. 7: Case study I (uniprocessor system)

Case study I (uniprocessor system). We first evaluate chain latency for a given available budget, which is motivated by the case study used in [11]. Fig 7 illustrates this case study that consists of two chains in a uniprocessor environment. In accordance with the case study in [11], callbacks are allocated to the same reservation with a given budget amount and the priorities of callbacks are given in descending order of callback's index for ROS2-SD, i.e., a lower-index callback has a higher priority. In that sense, we consider that chain 1 (Γ^1) has a higher semantic priority than chain 2 (Γ^2), and use this information in ROS2-PiCAS for callback priority assignment and node allocation. To evaluate the effect of budget in ROS2-PiCAS, we created a periodic thread that runs with the same period (i.e., 80 ms) as the two chains (Γ^1 and Γ^2), but has the highest real-time priority in the system. Using this thread, we were able to limit the available budget for ROS2-PiCAS by adjusting the execution time of that thread, e.g., 20 ms of thread execution time per 80 ms leaves 75% of budget for ROS2-PiCAS.



(a) Latency of chain 1 (Γ^1) for case study I



(b) Latency of chain 2 (Γ^2) for case study I

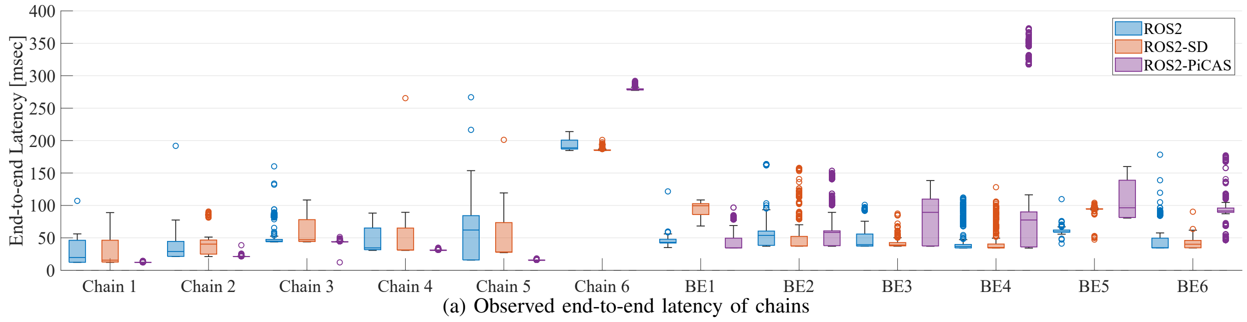
Fig. 8: Effect of available budget on end-to-end chain latency

Fig. 8 shows the end-to-end latency of chains under the two approaches. As expected, we can observe that the latency of both approaches reduces as the available budget increases. For ROS2-SD, the latency of chain 1 and chain 2 are the same because the built-in ROS2 scheduling policy makes the priority assignment ineffective for upper-bounding the response time of callbacks, as explained in [11]. However, chain 1 under ROS2-PiCAS has lower latency than ROS2-SD because our work prioritizes chains based on their semantic priority. We also carried out another experiment by increasing the executor priority of the two chains of ROS2-PiCAS higher than the periodic thread. As shown with yellow lines in Fig. 8, we observe that the latency is consistent regardless of the available budget because the chains are not interfered by the periodic thread in this case.

Case study II (multi-core system). To evaluate the effectiveness of chain-aware scheduling in practical scenarios, we add more complex chains inspired by the indoor self-driving stack of the F1/10 vehicle platform [1]. This case study consists of 6 real-time chains that we are interested in and 6 other best-efforts chains in a 4-core system. Fig. 10 shows the chain configuration in this case study. We assume that a lower-index chain has a higher semantic priority. With given priorities of chains, ROS2-PiCAS is used for node allocation. It first allocates chain 1 to chain 4 each to a different CPU core, and then distributes the others in a load-balancing manner. The utilization of each CPU core after allocation is 0.97 on average. Since no allocation scheme is provided by the default ROS2 scheduler as well as the prior work [11], we used the same node-to-core assignment and one executor per core for ROS2 and ROS2-SD. For ROS2-SD, we also set the resource reservation budget to 100% on each core.

Fig. 9a illustrates the observed end-to-end latency of chains under the three approaches. Fig. 9b shows the maximum observed latency from our measurements and compares it with the analyzed latency. We clarify that the observed latency in our experiments was obtained by recording the starting time of timer callback execution, not the release time, which was inevitable due to the lack of ROS2's support for setting the initial release offset. However, it is worth noting that the starting time of chain 1 to chain 4 are exactly the same as their release time under ROS2-PiCAS because they are each allocated to the highest priority executor on a different CPU core. Even if chain 5 and chain 6 are allocated to the same CPU cores as chain 1 (CPU 1) and chain 4 (CPU 4), respectively, we can easily estimate their release-time-based latency by considering their executors' priority (i.e., the second highest priority executor on each CPU core) and periods (i.e, harmonic with other chains) under ROS2-PiCAS. However, we cannot intuitively estimate the release-time-based latency under ROS2 and ROS2-SD.

As expected, ROS2-PiCAS outperforms the others on most real-time chains, e.g., chain 1 to chain 5. Besides, our latency analysis provides tighter upper-bounds for those real-time chains. However, we can see that the observed and analyzed latency of chain 6 under ROS2-PiCAS are worse than the others. We discuss this phenomenon for the following two reasons. First, under ROS2-PiCAS, a higher-priority chain 4 (T=100 ms and C=45.1 ms) can interfere with chain 6 (T=1000 ms and C=197.6 ms) multiple times, while under ROS2 and ROS2-SD, chain 4 and chain 6 can block only once each other due to the non-preemptive callback execution


(a) Observed end-to-end latency of chains

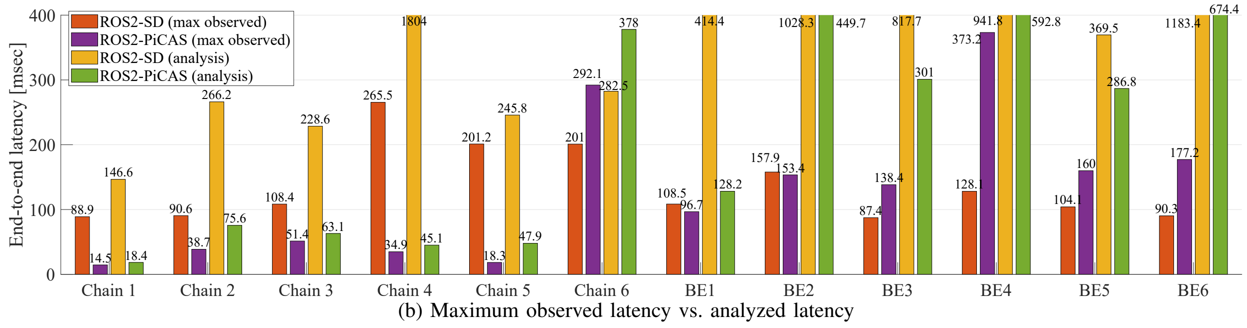
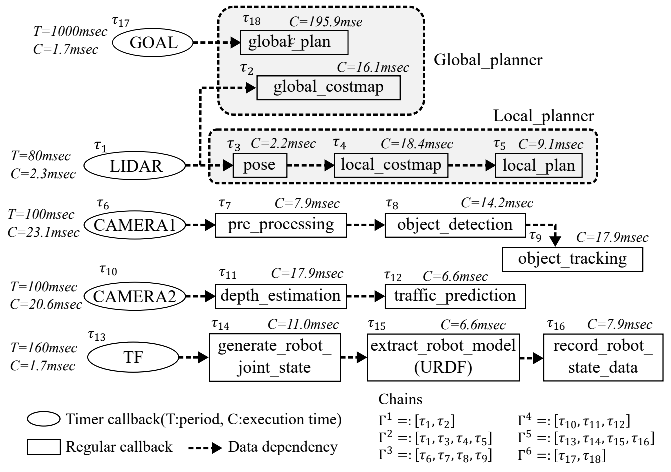
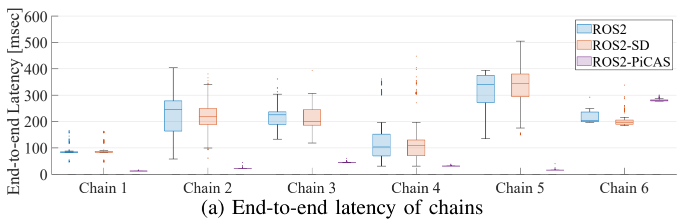

(b) Maximum observed latency vs. analyzed latency

Fig. 9: Case study II results for non-overload scenario



T=1000msec C=1.7msec τ17 GOAL
τ18 C=195.9mse global_plan
τ2 C=16.1msec global_costmap
Global_planner
Local_planner
T=80msec C=2.3msec τ1 LIDAR
τ3 C=2.2msec pose
τ4 C=18.4msec local_costmap
τ5 C=9.1msec local_plan
T=100msec C=23.1msec τ6 CAMERA1
τ7 C=7.9msec pre_processing
τ8 C=14.2msec object_detection
τ9 C=17.9msec object_tracking
T=100msec C=20.6msec τ10 CAMERA2
τ11 C=17.9msec depth_estimation
τ12 C=6.6msec traffic_prediction
T=160msec C=1.7msec τ13 TF
τ14 C=11.0msec generate_robot_joint_state
τ15 C=6.6msec extract_robot_model (URDF)
τ16 C=7.9msec record_robot_state_data
Timer callback(T:period, C:execution time)
Regular callback
Data dependency
Chains
Γ1 =: [τ1, τ2]
Γ2 =: [τ1, τ3, τ4, τ5]
Γ3 =: [τ6, τ7, τ8, τ9]
Γ4 =: [τ10, τ11, τ12]
Γ5 =: [τ13, τ14, τ15, τ16]
Γ6 =: [τ17, τ18]

Fig. 10: Case study II & III (multi-core system)


(a) End-to-end latency of chains

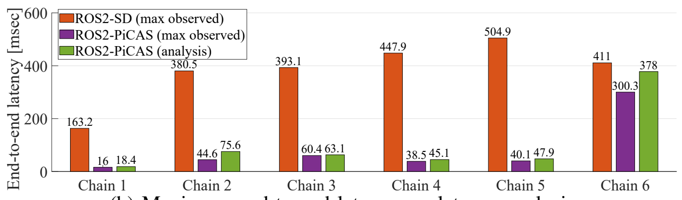

(b) Maximum end-to-end latency vs. latency analysis

Fig. 11: Case study III results for overloaded scenario

and fairness-based scheduling of the default ROS2 scheduler. Secondly, the observed latency of chain 6 under ROS2 and ROS2-SD does not include the blocking time from chain 4 because the observed latency is obtained using the starting time of a timer callback, as explained above. For the latter reason, more clear evidence can be seen in chain 4. Due to the non-preemptiveness, we expect that the latency of chain 4 under ROS2 and ROS2-SD should be at least larger than the execution time of chain 6. However, the observed latency does not include the execution time of chain 6. The large difference between the observed and analyzed latency of chain 4 under ROS2-SD (265.5 ms vs. 1804 ms shown in Fig. 9b) strengthens this argument. In conclusion, we found that PiCAS yields a substantial benefit in end-to-end chain latency and can schedule chains while respecting their semantic priority.

Case study III (overloaded scenario). We now use more best-efforts chains to evaluate latency behaviors in an overloaded system setup. By adding 4 more best-effort chains to the above multi-core system, we made each CPU core's utilization 1.25 on average. As can be seen in Fig. 11, ROS2-PiCAS significantly outperforms the other approaches for real-time chains. In particular, ROS2-PiCAS achieves up to 85% and 90% reduction in the average end-to-end latency for the most critical chains, i.e., chain 1 and chain 2, respectively. Moreover, while the analysis for ROS2-PiCAS gives a tight upper-bound on real-time chains, the analysis of ROS2-SD failed to test this overloaded system.

Note that we did not observe any appreciable delay from inter-core timing interference in our experimental setup. However, shared memory resources such as caches, memory buses, and DRAM banks are critical sources that cause timing unpredictability in multi-core platforms [8, 19, 20]. We leave addressing this issue as part of our future work.

C. Schedulability experiments

In this subsection, we evaluate the schedulability of chains under ROS2-SD and ROS2-PiCAS. We also assess the analysis running time of these approaches. All experiments are conducted on a machine equipped with dual AMD EPYC 7452 2.35GHz processors.

Workload generation. For each system utilization, We use $1,000$ randomly-generated workload sets of callbacks. The utilization of a workload set is selected from $\{2.5, 3.0, 3.5\}$ for a 4-core environment. For each workload set, we use 45 callbacks that forms 9 chains, i.e., each chain consists of 5 callbacks. Each callback's utilization is obtained by the UUniFast algorithm [10]. Chain period is chosen randomly in the range of $[50, 1000]$ ms and the deadline is set equal to the period. The utilization of each chain does not exceed 1.0. After the generation of each workload set, chains are reordered such that lower-index chains have shorter period, and lower-index chains are assigned higher semantic priority.

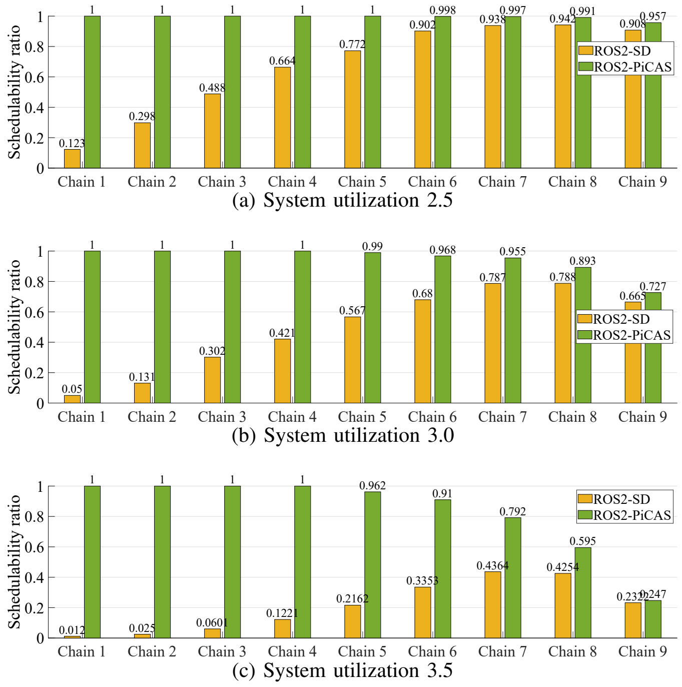


Fig. 12: Schedulability under ROS2-SD and ROS2-PiCAS

Comparison of schedulability tests. The results of schedulability ratio at each utilization are shown in Fig. 12. The schedulability ratio decreases as the utilization increases for both approaches. ROS2-PiCAS significantly outperforms ROS2-SD for all utilization setups. Under ROS2-PiCAS, the first four chains, i.e., chain 1 to chain 4, are always schedulable because they are each allocated to the highest priority executor on a different CPU core. Moreover, since ROS2-PiCAS prioritizes chains based on their semantic priority, the schedulability ratio decreases as the chain priority decreases (higher chain index). On the other hand, for ROS2-SD, we observe that the ratio decreases as the chain deadline gets shorter (lower chain index). This is mainly due to that ROS2-SD is agnostic to chain priority, as discussed with our case study I (Sec. VII-B).

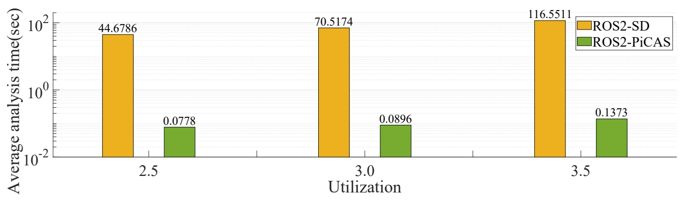


Fig. 13: Comparison of analysis running time

Comparison of analysis running time. In this experiment, we compare the analysis running time, which is the time to compute the end-to-end latency of each workload set, for ROS2-SD and ROS2-PiCAS. Each analysis tool is single-threaded. The analysis running time is obviously affected by the utilization of the workload set. We thus use the same workload sets as before, where the utilization per workload set is 2.5, 3.0, and 3.5. Fig. 13 shows the results of the average analysis time. The analysis time rises as the utilization increases under both approaches, but ROS2-SD is much slower than ROS2-PiCAS. This is because, unlike ROS2-PiCAS, the search space of the ROS2-SD analysis is for the longest possible busy interval (which tends to increase with the utilization) and the analysis iteratively searches for a global fixed point where all response times converge [11]. Therefore, we conclude that the proposed analysis for chain-aware scheduler is much faster than the existing approach, thereby applicable to complex systems and runtime admission control.

VIII. Conclusion

In this paper, we propose a priority-driven chain-aware scheduling and its end-to-end latency analysis framework for ROS2. To reduce the end-to-end latency based on the chains' criticality, we propose scheduling strategies for callbacks within and across executors. We then present callback priority assignment and chain-aware node allocation algorithms that substantialize those strategies. We implemented our chain-aware scheduler in the Eloquent Elusor version of ROS2 running on the NVIDIA Xavier NX platform. The results of the case study demonstrate that our proposed scheduler outperforms the existing ROS2 scheduling with respect to the end-to-end latency under practical scenarios. It has also been shown that our analysis technique upper-bounds the latency tightly and results in a better schedulability ratio compared to the start-of-the-art. As a next step, we will deploy our work to more complex and practical scenarios such as an autonomous driving software that is built on ROS2, e.g., autoware.auto. We also plan to extend our work for diverse ROS2 settings including the configuration of QoS for DDS communication.

Acknowledgment

We gratefully acknowledge the support from the Office of Naval Research (ONR) grant N00014-19-1-2496 and the National Science Foundation (NSF) award 1943265.

References

[1] F1tenth. https://f1tenth.org/, accessed October 2020.

[2] Object Management Group. https://www.omg.org/omg-dds-portal/, accessed October 2020.

[3] ROS Community Metrics Report. http://download.ros.org/downloads/metrics/metrics-report-2018-07.pdf, accessed October 2020.

[4] ROS Introduction. http://wiki.ros.org/ROS/Introduction, accessed October 2020.

[5] ROS2 response time analysis. https://github.com/boschresearch/ros2_response_time_analysis, accessed October 2020.

[6] Why ROS 2? http://design.ros2.org/articles/why_ros2.html, accessed October 2020.

[7] 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)*, pages 1625–1630. IEEE, 2019.

[8] B. Andersson, H. Kim, D. D. Niz, M. Klein, R. Rajkumar, and J. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):1–29, 2018.

[9] 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)*, pages 159–169. IEEE, 2016.

[10] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[11] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[12] 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)*.

[13] H. Choi and H. Kim. Work-in-progress: A unified runtime framework for weakly-hard real-time systems. *Brief Presentations of RTAS*, 2019.

[14] H. Choi, H. Kim, and Q. Zhu. Job-class-level fixed priority scheduling of weakly-hard real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 241–253. IEEE, 2019.

[15] 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*, pages 278–283, 2007.

[16] 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.

[17] Z. Hammadeh, S. Quinton, M. Panunzio, R. Henia, L. Rioux, and R. Ernst. Budgeting under-specified tasks for weakly-hard real-time systems. In *ECRTS 2017-29th Euromicro Conference on Real-Time Systems*, pages 1–22, 2017.

[18] 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*, 152(2):148–166, 2005.

[19] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014.

[20] H. Kim and R. Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(1):1–27, 2017.

[21] 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)*, volume 1, pages 360–367. IEEE, 2018.

[22] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 110–117. IEEE, 1995.

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

[24] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 26–37. IEEE, 1998.

[25] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*, pages 328–339. IEEE, 1999.

[26] Y. Saito, T. Azumi, S. Kato, and N. Nishio. Priority and synchronization support for ros. In *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 77–82. IEEE, 2016.

[27] 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)*, pages 52–58. IEEE, 2018.

[28] 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)*, pages 1–10. IEEE, 2016.

[29] 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*, 56:171–178, 2016.

[30] C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam, and W. Fun. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In *2008 International Symposium on Information Technology*, volume 4, pages 1–8, 2008.