Priority-Driven Real-Time Scheduling in ROS 2: Potential and Challenges

Hyunjong Choi, Daniel Enright, Hoora Sobhani, Yecheng Xiang, and Hyoseung Kim

University of California, Riverside

{hchoi036, denri006, hsobh002, yxian013, hyoseung}@ucr.edu

Abstract—To ensure timely and safe operations of robotic applications in a highly dynamic and uncertain environment, predictable end-to-end behavior of systems is essential. Although ROS (Robot Operating System) is one of the most prevalent robotic middleware frameworks, it has shown limitations in real-time support over the past decade. With this paper, we argue that the real-time performance and predictability of ROS can be significantly improved by enabling priority-driven scheduling in the framework. To support this argument, we first review our recent work on priority-driven chain-aware scheduling and evaluate it with real-world scenarios through integration into the open-source *reference system*, which was developed by Apex.AI for ROS 2 executor benchmarking. Experimental results on a resource-constrained platform, i.e., Raspberry Pi 4, demonstrate that priority-driven scheduling outperforms the current ROS 2 default scheduling scheme in terms of various key performance indicators, e.g., latency, message drop, and jitter. In addition, we discuss two other challenges, multi-threaded executor design and accelerator support, which have not yet been studied but are essential for better real-time performance in ROS 2.

I. INTRODUCTION

ROS (Robot Operating System) has gained the spotlight among developers in the robotics community by facilitating software modularity and composability in the development of robotic applications. However, over the past decade, ROS has shown major shortcomings in real-time support required for safety-critical applications. Although ROS 2, the new version of ROS, aims to better support real-time capabilities by employing a new software architecture and the Data Distribution Service (DDS), it still remains challenging to guarantee stringent timing constraints in ROS-based systems.

Ensuring predictable end-to-end latency is crucial for applications in a safety-critical domain [7]. However, meeting this requirement in a practical framework like ROS 2 is not a trivial problem due to the following reasons. First, robotic applications generally form a set of processing chains whose data and temporal dependencies are hard to analyze. Second, ROS 2 has a complex and unique scheduling behavior caused by multiple schedulable entities, e.g, callbacks, nodes, and executors, across various abstraction layers, making it difficult to apply existing real-time techniques. Lastly, due to the open-source nature of ROS, many programmers independently develop software components interacting with each other; hence, it is hard to integrate them into a system for a given mission while satisfying their performance requirements.

Research on real-time ROS 2 processing chains has started only recently. Casini et al. [6] proposed a pioneering analysis

technique to upper bound the response time of processing chains by modeling ROS executors with resource reservations. Tobias et al. [5] presented enhanced analysis to offer tighter bounds. These studies laid the groundwork to analyze systems developed using ROS 2. On the other hand, we took a completely different approach. While previous studies focused on the unmodified, default ROS 2 scheduling scheme, we found that major limitations, such as long end-to-end latency and pessimism in the analysis, are due to the poor support of prioritization in the existing scheduling scheme. As a result, we developed PiCAS [8], a priority-driven chain-aware scheduling framework for ROS 2, to improve the end-to-end latency of processing chains with predictable bounds. PiCAS also answers how to allocate resources to further improve responsiveness of critical chains.

While some real-time challenges have been studied as discussed above, there are still many open problems that need to be explored for ROS 2. All previous work on ROS 2 processing chains, including our own, only assume single-threaded executors. However, as we will show in Fig. 3, multi-threaded executors have the potential to offer better latency and higher throughput in a system equipped with multiple CPU cores. Besides, since ROS 2 is being widely used in the development of intelligent autonomous systems with machine learning algorithms, unpredictable timing behavior could appear from shared hardware accelerators such as GPU and FPGA, which would be a serious problem in resource-constrained embedded robotic platforms.

In this paper, we will explore the potential of the priority-driven scheduling approach to improve the real-time performance and predictability of ROS 2. We will first review our prior work, PiCAS, and then evaluate it on Raspberry Pi 4 with the reference system [3], which was developed by Apex.AI to benchmark the performance of ROS 2 executors under real-world autonomous driving scenarios. Then we will discuss the two open problems that we are currently working on, i.e., multi-threaded executor design and real-time GPU acceleration support, which would be essential for ROS 2 to serve as a practical yet reliable real-time software infrastructure.

II. PRIORITY-DRIVEN CHAIN-AWARE SCHEDULING

A. Background

Our priority-driven chain-aware scheduling, PiCAS [8], was motivated by the two major problems of the current ROS 2 framework. First, ROS 2 consists of multiple layers of

abstractions that are not aware of the criticality of processing chains. The unique scheduling behavior of an executor, which schedules timer callbacks always first, makes other callbacks' priorities ineffective. Hence, the current ROS 2 executor ignores the urgency of task chains and results in a fairness-oriented scheduling behavior. Second, the current ROS 2 framework lacks systematic support for resource allocation and latency analysis. This causes poor resource utilization and non-deterministic end-to-end behavior.

To solve these issues, PiCAS enables prioritization of critical computation chains across complex abstraction layers of ROS 2 (see Fig. 1 for overview). We re-designed the current ROS 2 scheduling architecture with the following considerations: (1) higher-priority chain should execute earlier than lower-priority chains, and (2) if the instances of the same chain are assigned to the same CPU core, they should execute in their arrival order. The latter is to reduce self-interference between instances of the same chain, thereby preventing undesirable latency increases.
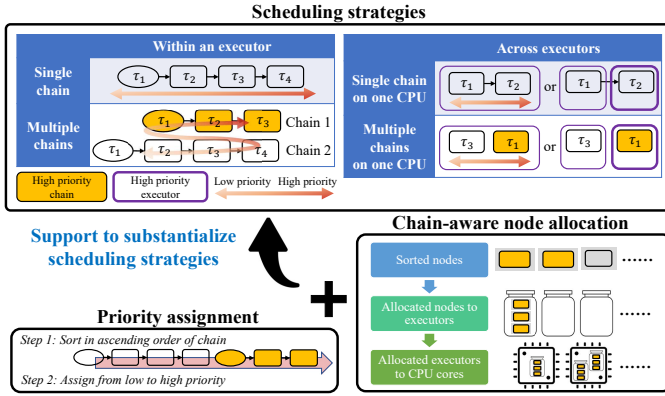

Fig. 1: PiCAS framework

Based on the above considerations, we developed chain scheduling strategies within an executor and across executors. To realize these scheduling strategies, PiCAS introduces a callback priority assignment scheme. It assigns strictly higher priority to callbacks of more critical chains, and within each chain, it prioritizes callbacks in the front to those in the back to avoid the self interference problem. PiCAS also includes a chain-aware node allocation algorithm to allocate given nodes to executors, and then maps executors to available CPU cores while following the scheduling strategies. This algorithm tries to allocate all nodes associated with the same chain to the same CPU core whenever possible in order to minimize interference between different chains.

B. Evaluation of Priority-Driven Scheduling

To understand the benefit of the priority-based scheduling approach under a realistic scenario, we evaluate PiCAS with the reference system [3] that was developed by Apex.AI and introduced at the ROS 2 Real-Time Executor Workshop held in conjunction with ROSCon 2021 [2]. The reference system resembles the lidar-based perception pipeline of Autoware.Auto [4], as illustrated in Fig. 2. We integrated PiCAS
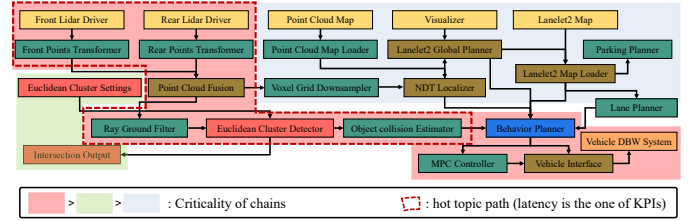

Fig. 2: Chain configuration of Autoware model

into the reference system running on the Galactic version of ROS 2 in a Raspberry Pi 4 platform.

Key performance indicators (KPIs). For ease of benchmarking, the reference system evaluates various KPIs such as:

- *Latency of hot topic path*: In a real-world scenario, the reference system should recognize obstacles as quickly as possible to avoid collisions. Thus, the lower latency from the Front Lidar to the Object Collision Estimator (the red dotted line shape in Fig 2) is better.
- *Number of dropped messages*: Since old sensor data is less valuable than newly sensed data, the old ones can be dropped in favor of the newest sample, but at the cost of information is lost. Therefore, the fewer number of dropped messages is better.
- *Timing jitter of Behavior Planner*: The Behavior Planner node should execute periodically, as accurate as possible according to its set frequency (100 msec). Thus, the lower jitter and drift of this node are better.

Comparison of approaches. We compare the priority-driven scheduling approach (ROS2-PiCAS) with the default ROS 2 scheduling scheme (ROS2-default). Two different executor configurations are considered for ROS2-default: single-threaded and multi-threaded executors. With the single-threaded executor, all nodes are allocated to one thread running on a single CPU core. So, we compare this to PiCAS with one single-threaded executor. The multi-threaded executor runs with as many worker threads as the number of available CPU cores. Since PiCAS does not currently support multi-threaded executors, we use multiple single-threaded executors for PiCAS, i.e., four single-threaded executors on four cores of Raspberry Pi 4, and compare this with the multi-threaded executor of the default ROS 2.
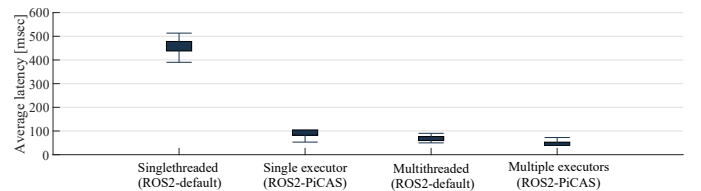

Fig. 3: Average end-to-end latency of hot topic path

End-to-end latency of hot topic path. Fig. 3 illustrates the observed average latency of the hot topic path under four different cases. ROS2-PiCAS with a single executor reduces average latency by up to 86% compared to the single-threaded ROS2-default, and shows comparable performance to

the multi-threaded ROS2-default. This result demonstrates the significant benefit of the priority-driven scheduling approach, which help autonomous vehicles recognize obstacles much faster and avoid them in a timely manner while using the same amount of resources.

In case of the multi-threaded executor, ROS2-default performs not as good as ROS2-PiCAS with multiple executors. This is interesting since the default multi-threaded executor follows the global scheduling approach that is naturally better in reclaiming unused resources than partitioned scheduling, which the multiple single-threaded executors of ROS2-PiCAS represent. We suspect that this is not due to an inherent flaw of the multi-thread executor but due to the lack of proper prioritization support.

TABLE I: Number of dropped messages

	Singlethreaded (ROS2-default)	Single executor (ROS2-PiCAS)	Multithreaded (ROS2-default)	Multi. executors (ROS2-PiCAS)
Mean	0.8681	0.0282	0	0
STD	0.3347	0.1651	0	0

Number of dropped messages. Table I shows the number of dropped messages. As expected, ROS2-PiCAS outperforms ROS2-default in a single-threaded executor setup. Note that we do not see any message drops for the multi-threaded ROS2-default and the ROS2-PiCAS with multiple executors.
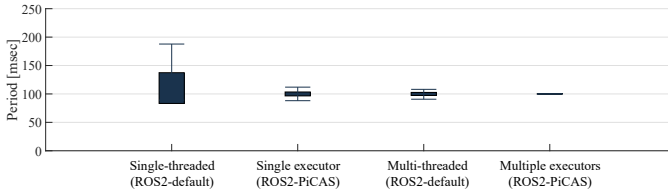

Fig. 4: Behavior Planner jitter

Behavior Planner jitter. Fig 4 illustrates the observed execution period of the Behavior Planner. Any deviation from 100 msec indicates a timing jitter, so a narrow range of observed values is better. As can be seen, ROS2-PiCAS outperforms ROS2-default in all configurations. Such a small uncertainty of the priority-driven scheduling approach can help improve the predictability of the ROS 2 framework.

III. REAL-TIME SUPPORT FOR MULTI-THREADED EXECUTORS

Although ROS 2 provides multi-threaded executors, prior studies [5, 6, 8] have considered only single-threaded executors. In general, multithreading allows effective utilization of multiple processors and helps improve system concurrency and throughput. The benefit of real-time multithreading has been demonstrated in the context of self-driving cars [10]. We also have observed that the default multi-threaded executor of ROS 2 has better latency performance than its single-threaded counterpart, as shown in Fig. 3. Despite such a benefit of the ROS 2 multi-threaded executor, to the best of our knowledge, there is no prior work on analyzing and improving the timing behavior of the multi-threaded executor for ROS 2. Therefore,

in this section, we discuss challenges that arise with the ROS 2 multi-threaded executor.

In order to make use of the multi-threaded executor in a system with stringent timing requirements, the very first step required is to formally analyze its timing behavior as people did for the single-threaded executor. However, unlike the single-threaded executor, the analysis of processing chains on a multi-threaded executor is more challenging due to the runtime callback distribution across multiple threads and the unsynchronized polling points of the threads. Such challenges makes it difficult to extend the existing ROS 2 analysis techniques to multi-threaded executors directly. For analysis purposes, we are currently modeling single-threaded and multi-threaded executors as partitioned and global schedulers, respectively. Throughout this modeling, we aim to extend the conventional non-preemptive global task scheduling techniques, e.g., [11], to the ROS 2 environment by taking into account semantic differences such as callback dependencies, chains, polling points, and ready set management. We are also working on extending PiCAS to multi-threaded executors to enable priority-driven scheduling and to achieve better end-to-end latency and predictability. Once done, we can compare the performance of priority-driven callback scheduling in a multi-threaded executor vs. in multiple single-threaded executors.

ROS 2 provides an interesting feature for multi-threaded executors, called the *callback group*, which can be used to enforce concurrency rules for callbacks. There are two types of callback groups: *mutually-exclusive* and *reentrant*. Based on the type of the callback groups, the timing behavior of the system and the end-to-end latency of chains will be different. This opens new problems that motivated us to further explore: i) how these callback groups might affect the timing behavior of ROS 2 executors, ii) how we can analytically model the end-to-end latency of chains for each type, and iii) how these can be configured to improve real-time performance. We believe studies on these issues can lead to more efficient scheduling approaches in ROS 2, e.g., assigning callbacks to groups and then scheduling the callback groups.

IV. CHALLENGES WITH REAL-TIME GPU ACCELERATION

This section addresses issues with applications that rely on GPU accelerated kernels. Many applications designed with ROS 2 utilize asynchronous and unstructured models for kernel execution on GPU accelerators. While this encourages direct resource allocation and accelerator kernel calling from individual ROS 2 nodes, this may incur unpredictable real-time behavior, especially when many nodes request the same accelerator resource. Utilizing shared accelerator resources for complex software stacks, including autonomous vehicle (AV) stacks, is inevitable with modern computer and accelerator architecture. Our on-going work focuses on providing real-time GPU kernel execution management on resource-constrained systems.

A. Problems with Shared Accelerators

With an increasing amount of shared accelerator utilization among complex software stacks, comes consequences that
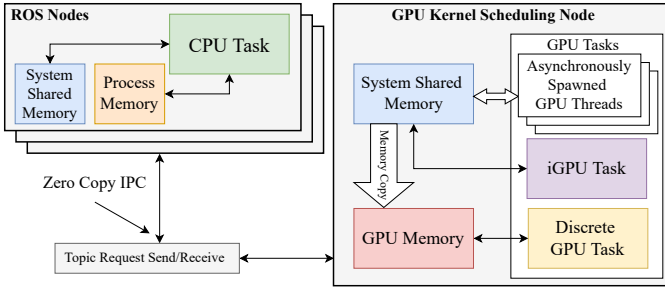
Fig. 5: ROS 2 GPU server framework

compromise real-time guarantees for safety-critical workflows. For ROS and ROS 2 specific AV stacks, many individual processing chains may necessitate the use of GPU-based accelerators for various perception, localization, mapping, and other tasks. For systems that maintain ample accelerator resources, blocking time for high-priority chains induced by GPU kernel execution from low-priority chains may be uncommon. However, for resource-constrained systems, high-priority chains may suffer from severe delays and deadline misses due to priority inversion when low-priority chains have been already utilizing the shared accelerator resource.

B. Maintaining Real-time Support with Accelerators

A solution that we are currently exploring to address to these dependability issues relies on a GPU-server-based approach within the ROS2 software stack. In conventional autonomous vehicle software design, the callbacks of each node directly invoke the GPU to execute kernels. Our current approach will utilize a separate ROS 2 node that acts as a GPU server – handling GPU access requests from all nodes in the stack. This idea is motivated by our earlier work on real-time GPU server [9]. The GPU server architecture will employ priority-based scheduling with support for request-level preemption. We are also considering concurrent kernel execution with real-time spatial GPU multitasking [13, 14] and prioritized CUDA streams [15] for better resource utilization and lower response time. In Fig. 5, describing the overall architecture, ROS nodes will request that a specific GPU kernel be executed on a specific set of data. Intuitively, this will cause additional delays due to extra memory copies between nodes. However, minimizing data copy delays with efficient zero-copy IPC methods like Iceoryx [1] and shared memory transport allows this architecture to support a very low-overhead accelerator resource management framework. The GPU-server node will maintain a structure of all GPU kernels and will schedule the execution of a kernel on a node's data in accordance with the corresponding chain's priority. Handling GPU kernel scheduling in the software stack rather than leaving it to the OS or GPU driver will give applications granular control over how specific chains access the GPU. Other methods of GPU multitasking and scheduling, such as Nvidia's Multi-Process Service (MPS) [12], can allow for multiple processes to perform concurrent kernel execution on different SM's, but do not provide any real-time, priority-based, or preemptive support for processing chains in ROS 2.

V. CONCLUSION

In this paper, we presented the benefit of enabling priority-driven scheduling in the ROS 2 framework and discussed open challenges. We integrated our prior work on priority-driven chain-aware scheduling into the reference autonomous system and evaluated several key performance indicators under a real-world scenario. The results of the case study demonstrate that the priority-driven scheduling approach significantly outperforms the existing ROS 2 scheduling scheme with respect to the average end-to-end latency, dropped messages, and jitter of periodic nodes. However, previous work, including our own, has been conducted under the assumption of a single-threaded executor, and the extension of existing techniques to the multi-threaded executor still remains as open problems. Besides, real-time support of ROS 2 with shared accelerators such as GPU and FPGA is another challenge that should be resolved for modern intelligent applications. We discussed these challenges and outlined directions to address them following the priority-driven approach.

Our focus in this paper has been around the default ROS 2 executor design and implementation, but there are existing executors, such as the cbg executor [16] and those proposed in the ROS 2 Executor Workshop [2]. We plan to evaluate the effectiveness of our approach against them in the future.

REFERENCES

[1] Eclipse iceoryx - true zero-copy inter-process-communication. https://github.com/eclipse-iceoryx/iceoryx, accessed March 2022.
[2] ROS2 Executor: How to make it efficient, real-time and deterministic? https://www.apex.ai/roscon-21, accessed March 2022.
[3] ROS2 Real-Time Working Group: Reference system. https://github.com/ros-realtime/reference-system, accessed March 2022.
[4] Autoware Foundation. https://gitlab.com/autowarefoundation/autoware.auto, accessed May 2022.
[5] T. Blaß et al. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance. In *RTSS*, 2021.
[6] D. Casini et al. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *ECRTS*, 2019.
[7] H. Choi et al. Chain-based fixed-priority scheduling of loosely-dependent tasks. In *ICCD*, 2020.
[8] H. Choi et al. PiCAS: New design of priority-driven chain-aware scheduling for ROS2. In *RTAS*, 2021.
[9] H. Kim et al. A server-based approach for predictable GPU access with improved analysis. *Journal of Systems Architecture*, 88:97–109, 2018.
[10] J. Kim et al. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *ICCPS*, 2013.
[11] J. Lee. Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling. *IEEE Transactions on Computers*, 66(10):1816–1823, 2017.
[12] Nvidia. Nvidia multi-process service. https://docs.nvidia.com/deploy/mps/index.html, accessed March 2022.
[13] S. Saha et al. STGM: Spatio-temporal GPU management for real-time tasks. In *RTCSA*, 2019.
[14] Y. Wang et al. Balancing energy efficiency and real-time performance in GPU scheduling. In *RTSS*, 2021.
[15] Y. Xiang and H. Kim. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *RTSS*, 2019.
[16] Y. Yang and T. Azumi. Exploring real-time executor on ROS 2. In *ICESS*, 2020.