Theory-Guided Adaptive Scheduling for ROS 2

Daniel Enright[0009−0005−8369−7556], Hoora Sobhani[0009−0009−6611−9374], and Hyoseung Kim[0000−0002−8553−732X]

University of California, Riverside
{denri006,hsobh002,hyoseung}@ucr.edu

Abstract. This paper presents Latency Management Executor (LaME), a theory-guided adaptive scheduling framework that enhances real-time performance in ROS 2 through dynamic resource allocation and hybrid priority-driven scheduling. LaME introduces the concept of thread-classes to dynamically adjust system configurations, ensuring response-time guarantees for real-time chains while maintaining starvation freedom for best-effort chains. By implementing adaptive resource allocation and continuous runtime monitoring, LaME provides robust response times even under fluctuating workloads and resource constraints. We implement our framework for the Autoware reference system and perform our evaluation on an Nvidia Jetson platform. Our results demonstrate that LaME successfully adapts to changing resource availability and workload surges, and effectively balances real-time guarantees with overall system throughput.

1 Introduction

The Robot Operating System (ROS) is widely regarded as one of the most popular middleware platforms for robotics, facilitating seamless integration of various software components and allowing rapid development of robotic systems. It provides essential tools for communication, hardware abstraction, and device control, fostering a collaborative ecosystem that has gained significant traction in both academia and industry. However, as robotics increasingly moves into domains requiring real-time guarantees, such as autonomous vehicles [13, 14], industrial automation [16], and medical robotics [17], the limitations of ROS's scheduling and resource management mechanisms become apparent. The standard ROS 2 framework lacks robust scheduling capabilities, which poses challenges in meeting timing constraints critical for real-time applications. Recent research has focused on enhancing ROS 2's real-time performance by integrating scheduling policies that constrain latency [3–6, 8, 9, 19, 20] and avoid task starvation [4, 22]. Despite this progress, ensuring robust, predictable behavior in ROS-based real-time systems still remains as an active research problem.

ROS applications typically consist of *processing chains* comprising multiple *callbacks* which are managed and executed by *executors*. A processing chain represents a sequence of callbacks that need to be executed in a specific order, such as processing sensor data, running control algorithms, and sending commands to actuators. These chains are composed of individual callbacks, which are functions that get executed in response to specific events or messages. Executors manage the execution of these callbacks and determine which callback runs at any given time. In the standard ROS 2 implementation, executors follow a round-robin scheduling policy, where callbacks are executed based on availability. While this approach ensures fairness, due to the lack of explicit priority considerations, chain response times can be prolonged as workload increases. Recent studies have introduced priority-based scheduling [3, 7, 8, 19] to reduce response times for real-time chains, but these can lead to the starvation of best-effort chains, potentially degrading overall system performance.

While formal schedulability analyses exist to bound chain response times for both standard and priority-driven ROS 2 executors, the significance of resource allocation has been somewhat neglected in the literature. Existing approaches commonly assume that the system designer has already determined resource allocation, with resource availability guaranteed through the `SCHED_DEADLINE` mechanism in Linux and modeled as a supply bound function; then the analysis is performed on this predetermined allocation. However, their guarantees hold only in static environments that assume constant system resources and accurate worst-case execution time estimations. When deployed in dynamic environments with varying resource availability, these analyses can break, rendering the application crippled with timing violations.

This paper presents a theory-guided approach to dynamically controlling the scheduling parameters and resource allocation for ROS 2 multi-threaded executors. The proposed Latency Management Executor (LaME) framework determines individual executor thread budgets, chain-to-threadclass allocation, and achieves *affinity-based scheduling* where a chain executes on either a single thread (partitioned scheduling) or multiple threads (constrained global scheduling). Our framework monitors the response times of all chains and leverages the theoretical analysis of ROS 2 executors [19] to ensure that the real-time processing chains meet their deadlines while maintaining the fairness and starvation freedom of best-effort chains. The framework continuously updates its predictions on response times and scheduling control decisions based on the feedback from runtime monitoring of execution times, enabling adaptation to changing workload characteristics and external disturbances, such as arbitrary workload surges and clock frequency throttling.

2 System Model

This section describes the scheduling-related abstractions of ROS 2 and our system model.

Callbacks. In ROS 2, software is organized into nodes, each comprising multiple callbacks that are executed in response to specific events such as message arrivals or timer triggers. Callbacks are the smallest schedulable entity in ROS 2, triggered by timers, message arrivals, or other events. Callbacks are managed by *executors*, which may be responsible for scheduling callbacks from one or more nodes that are assigned to them. Let τ_i denote a callback by:

$$\tau_i =: (E_i, \pi_i)$$

- E_i: Worst-case execution time (WCET) for τ_i.
- π_i: The priority of the callback τ_i within the executor.

Note that in the default ROS 2 executor, each callback is assigned an implicit priority by its class (timer, subscription, service, client, and waitable) [6, 8, 20] and within each class, in a nondeterministic order [2]. However, our proposed work explicitly assigns priorities for callbacks based on their respective chains' criticality level. Callbacks are non-preemptively scheduled by executors, i.e., once a callback begins execution, it runs on the executor to completion. For the purposes of this paper, we constrain the WCET of a callback, E_i, to its execution time in isolation, i.e., when there are no other interfering workloads on the CPU.

Processing Chains. Processing chains are sequences of data-dependent callbacks that must execute in a specific order to fulfill a certain task. Each callback within a chain has dependencies on the data or events generated by previous callbacks in the sequence. In the literature on real-time ROS 2 work [6, 8, 9, 19, 20], these chains are assumed to have a single starting point and a single ending point. We specifically refer to such a chain as a *linear* chain Γ_c, represented by:

$$\Gamma_c := ([\tau_{c_1}, \tau_{c_2}, ..., \tau_{c_n}], E_c, T_c, D_c, \zeta_c)$$

- $[\tau_{c_1}, \tau_{c_2}, ..., \tau_{c_n}]$: The sequence of callbacks executed by each instance of a chain Γ_c. Callback $\tau_{c_{i+1}}$ can start execution only when its predecessor τ_{c_i} completes [6, 8, 19, 20].
- E_c: Cumulative WCET of the chain, i.e., $E_c = \sum_{i=1}^{n} E_{c_i}$.
- T_c: Period of the chain, which is determined by the arrival rate of the first callback τ_{c_1}.
- D_c: Deadline of the chain, which is assumed to be constrained ($D_c \leq T_c$).
- ζ_c: Criticality of the chain. In this work, we divide chains into two criticality classes: *real-time (RT)* and *best-effort (BE)* chains. For RT chains, $\zeta_c > 0$, with a higher value meaning higher criticality. For BE chains, $\zeta_c = 0$.

Here, the criticality ζ_c is an indicator of the importance of meeting their deadlines. For the RT chains ($\zeta_c > 0$), our goal is to guarantee that their deadlines will always be met, i.e., the worst-case response-time of the chain, R_c, is guaranteed to be less than or equal to its deadline. For BE chains ($\zeta_c = 0$), our goal is to achieve fairness, with bounded response times (but could be longer than the deadlines) whenever possible.

While the above representation follows that of prior work [6, 8, 19, 20], many real-world ROS 2 applications include *nonlinear* chains, where some callbacks branch out, e.g., one callback triggering two or more subsequent callbacks. Hence, a nonlinear chain has a single starting callback and multiple ending callbacks. It can also be thought of as multiple linear chains sharing common callbacks. Prior work has dealt with this issue by virtually duplicating shared callbacks for analysis purposes. However, this becomes a nontrivial problem when criticality and resource management are involved. For example, each branch of a nonlinear chain may maintain a different criticality value or even belong to a different criticality class than any of the other branches. Our goal is to manage these issues at the executor level and we explain how this is handled in Sec. 4.2.

Executor and Threads. In ROS 2, an executor is launched as a process that is either single or multi-threaded. Each thread manages the execution of callbacks within the executor, while being itself a schedulable entity at the operating system level. The executor maintains a *waitset* (previously known as a readyset [6, 19, 20]), which is a data structure storing ready callbacks received from the underlying communication layer. Each thread fetches and executes one ready callback from the waitset at a time, and updates the waitset (i.e., refilling it from the communication layer) when it is empty or no callback in the waitset is eligible to run (due to callback groups that are used to limit concurrency in a multi-threaded executor). The waitset is protected by a mutex.

Prior work on real-time ROS 2 analysis [6, 12, 19, 20] has applied the resource reservation concept to executor threads to ensure guaranteed resource supply for schedulability analysis. Following the same approach, we characterize an executor thread by $r_k = (C_k^r, T_k^r)$, where C_k^r is the budget and T_k^r is the replenishment period. The supply bound function of r_k is then given by $sbf_k(\Delta) = \frac{C_k^r}{T_k^r}(\Delta - 2(T_k^r - C_k^r))$ if a time interval $\Delta \geq 2(T_k^r - C_k^r)$ [18]. This means r_k is guaranteed to get at least $sbf_k(\Delta)$ units of CPU time during an arbitrary time interval Δ, and it can be realized using `SCHED_DEADLINE` in Linux.

Let us use Π to denote a set of executor threads, $\Pi = \{r_1, r_2, ..., r_k\}$. Then, the supply bound function of Π is given by $sbf_\Pi(\Delta) = \sum_{r_k \in \Pi} sbf_k(\Delta)$ [10, 19]. We use $|\Pi|$ to represent the number of threads within Π. For the standard ROS 2 multi-threaded executor, only a single Π is enough to represent all threads in the executor, as it treats all threads equally with the same budget per thread. In contrast, as we will show later, our work introduces groups of threads, i.e., $\mathcal{P} = \{\Pi_1, \Pi_2, ...\}$, to achieve more fine-grained resource management.

3 Related Work

There have been several recent studies focusing on improving the real-time capabilities of the ROS 2 executor. Casini et al. [6] provided a response time analysis capable of bounding the response times of processing chains across one or more single-threaded executors in ROS 2. Tang et al. [20] improved upon this analysis and reduced the pessimism in the response time bounds. Choi et al. [8] posited an improvement to the executor to support priority-driven callback scheduling to reduce the response times for critical processing chains and provided a response-time analysis for those chains under its scheduling framework. Jiang et al. [12] provided a response time analysis capable of the response times of chains within the ROS 2 multi-threaded executor. Sobhani et al. [19] provided an improvement to the ROS 2 multi-threaded executor, enabling priority-driven global callback scheduling, and provided an analysis bounding chain response times with constrained and arbitrary deadlines under both the standard and priority-driven multi-threaded executors. Teper et al. [22] found the problem of callback starvation in ROS 2 multi-threaded executors and proposed design enhancements to address this issue. All existing analysis work, except [8], use a supply bound function to characterize resource availability and recommend using `SCHED_DEADLINE` to conform to their system model and to achieve performance isolation among executor threads. While these studies have enabled real-time schedulability analysis in ROS 2, they assume that resource allocation is given in advance, which is the problem our work addresses.

The most closely related prior work to this paper is ROS-Llama proposed by Blass et al. [5]. While ROS-Llama focuses on resource allocation and budget management, it has several notable limitations. First, it considers only *single-threaded executors*. If an application's utilization exceeds one CPU core, it remains the user's responsibility to determine which chains to assign to which (single-threaded) executors, i.e., analogous to static partitioned scheduling. Moreover, if any chain's utilization exceeds one core, the user must manually split the chain and assign subchains to different executors. In contrast, our framework supports dynamic chain-to-threadclass allocation, allowing for both *partitioned* and *constrained global callback scheduling* in ROS 2 multi-threaded executors. Our work can execute high-utilization chains as-is on a threadclass with multiple threads across multiple cores without requiring manual partitioning. Second, ROS-Llama does not consider chain criticality levels, i.e., it treats BE and RT chains equally with no prioritization, making all chains compete with each other. Our framework addresses this limitation through explicit isolation between RT and BE chains, where the RT chains are *always* prioritized over BE chains. Our work also provides bounded response times for RT chains while guaranteeing fairness for BE chains and starvation freedom for them when possible.

Goals: The primary objectives of our work are to effectively manage latency in robotic systems that are complex, dynamically evolving, and subject to changing hardware environments. The challenge lies in balancing the response times of time-critical chains with ensuring fairness and preventing the starvation of best-effort chains, while dealing with numerous control parameters that have nonlinear effects on system performance. Additionally, we aim to ensure the stability and convergence of the system in the presence of scheduling anomalies and varying assumptions about available resources.

4 Latency Management Executor Architecture

In this section, we present our Latency Management Executor (LaME), which is a redesign of the ROS 2 multi-threaded executor to provide precise control over chain execution timing. Specifically, LaME incorporates five key enhancements: (i) threadclasses as a new abstraction for managing executor threads; (ii) explicit chain-to-thread affinity management to enable both partitioned and constrained global scheduling; (iii) a hybrid chain scheduling policy that differentiates between real-time and best-effort chains; (iv) an online response-time test to check

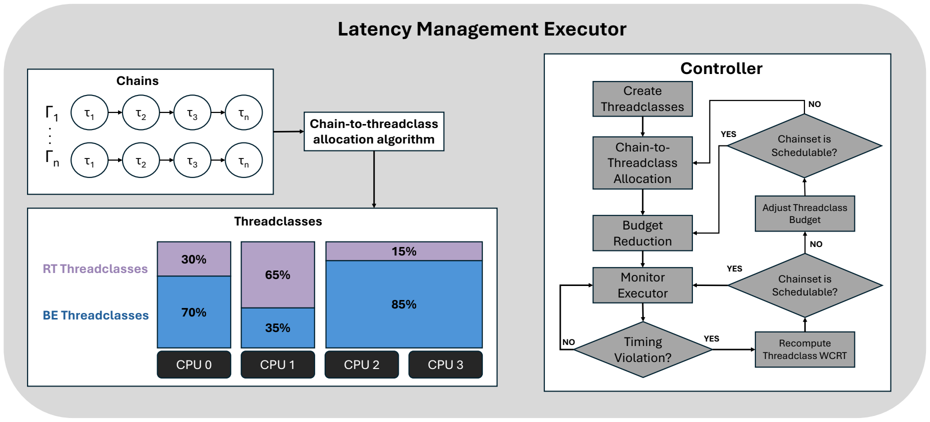


Fig. 1: Architecture of LaME

if real-time chains can meet their deadlines and best-effort chains can run with no anticipated starvation; and (v) an adaptive resource controller that dynamically adjusts resource and chain allocations. Note that our framework is designed to be a drop-in replacement for the existing executor; hence, it can be directly applied to existing ROS 2 applications with no compatibility issues.

In the following, we will present the key abstractions and architectural mechanisms (Sec. 4.1), the methods to enable online timing analysis in ROS (Sec. 4.2), and the adaptive resource controller design and its algorithms (Sec. 4.3).

4.1 Abstractions and Mechanisms

Threadclasses. *Threadclasses* are the key abstractions that our framework introduces to the ROS 2 executor. The primary objective of the threadclasses is to form groups of individual executor threads to perform more sophisticated chain allocation and resource management while still conforming to the system models of existing theoretical foundations.

Each threadclass, Π, is composed of one or more executor threads, each pinned to a separate CPU core, with each thread assigned identical `SCHED_DEADLINE` budgets and a list of chains that it has been designated to service. This grouping of executor threads into threadclasses ensures that threads within the same threadclass have uniform scheduling parameters, specifically the same `SCHED_DEADLINE` budget for its assigned set of chains (chainset). This allows us to perform a Worst-Case Response Time (WCRT) analysis per threadclass: by treating all threads in a threadclass uniformly, we can effectively consider each threadclass as an individual single-threaded or multi-threaded executor, which enables the use of most existing analysis for ROS [6, 12, 19, 20].

The grouping of threads into threadclasses also aids in maintaining isolation between real-time (RT) and best-effort (BE) chains by creating their own dedicated threadclasses, i.e., $\{\Pi_{RT_1}, \Pi_{RT_2}, ...\}$ and $\{\Pi_{BE_1}, \Pi_{BE_2}, ...\}$ for RT and BE threadclasses, respectively.

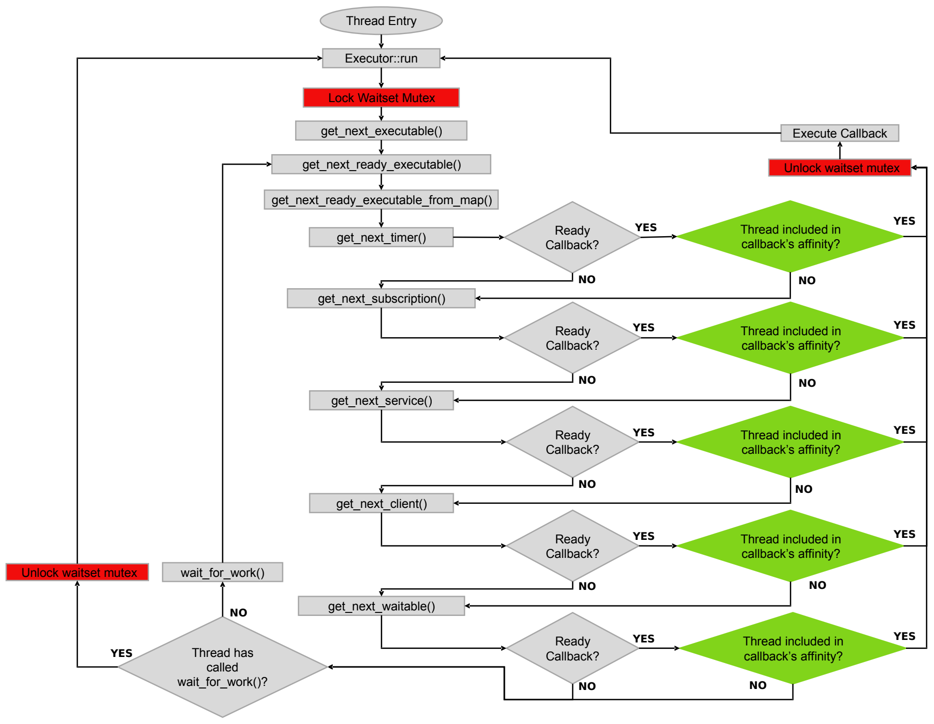


Fig. 2: Flowchart of an executor thread's main loop w/ callback affinity checking

CPU-time Partitioning with Threadclasses Figure 1 illustrates how LaME implements the threadclass abstraction to achieve CPU-time partitioning. LaME creates two distinct threadclasses per CPU core: an RT threadclass Π_{RT_m} for RT chains and a BE threadclass Π_{BE_m} for BE chains on core m. Each threadclass is initially populated with one thread, resulting in two threads (one RT, one BE) per core. Consistent with our threadclass definition, all threads within the same threadclass share identical `SCHED_DEADLINE` parameters, enabling uniform resource allocation and predictable analysis. These threads are isolated to specific cores using Linux cgroups, enforcing the physical separation between threadclasses. This ensures that workload fluctuations in one threadclass (e.g., BE chains) cannot consume resources allocated to the other (RT chains), maintaining the performance guarantees necessary for many real-time applications.

Chain-to-Threadclass Mapping via Callback Affinity. Conceptually, our framework maps entire chains to specific threadclasses, ensuring that all callbacks within a chain are processed by threads belonging to the same threadclass. However, since the standard ROS 2's execution model operates at the callback level for all threads in the executor, we implement this mapping through a new callback-to-thread affinity API in the `rclcpp` layer.

The callback affinity API translates our high-level chain-to-threadclass mapping into a practical implementation that works within ROS 2's architecture. When a chain is assigned to a threadclass, all callbacks belonging to that chain inherit an affinity setting that limits their execution to only the threads in that threadclass. This ensures chain-level allocation decisions are consistently enforced at the callback execution level.

To implement this API, we added a new data member within each callback class that specifies its thread affinity as a bitmask of eligible executor threads. Figure 2 illustrates the workflow of executor threads with callback's affinity enabled, showing our additional affinity checks (green decision blocks). When an executor thread encounters a ready callback, it first verifies that the callback's affinity mask includes the current thread ID—meaning the callback belongs to a chain assigned to this thread's threadclass. If the affinity check fails, the thread continues searching for other work; otherwise, it proceeds to execute the callback.

This mechanism allows us to implement both partitioned scheduling (when chains are assigned to a dedicated thread, i.e. $|\Pi| = 1$) and constrained global scheduling (when chains can execute on a subset of threads, i.e. $|\Pi| > 1$), while maintaining the conceptual integrity of chain-to-threadclass mapping within ROS 2's callback-oriented execution model.

Callback Scheduling Policies. Beyond creating specific RT and BE threads and corresponding threadclasses, our framework allows RT threads to schedule callbacks differently than BE threads. RT threads consider callback priority when deciding which callback to execute. Callback priority is assigned by our framework and will be explained later in this section. RT threads always execute the highest priority available callback matching its affinity setting, while BE threads will execute only the first available callback that matches its thread-affinity setting. The RT threads also perform a quick check for any ready callbacks not currently in the waitset (set of callbacks ready to be executed) before deciding which callback to execute, following the behavior of the priority-driven ROS 2 executor [8]. The BE threads only perform this check if the waitset is completely empty, following the behavior of the default ROS 2 executor.

RT Mutex. In our experiments, we observed unbounded blocking in the default executor due to the use of a shared waitset mutex (shown by red boxes in Figure 2) under `SCHED_DEADLINE`. Specifically, threads can be throttled by the operating system while holding the mutex during the critical section (area protected by the red mutex blocks in Figure 2), stalling other threads until the lockholder is rescheduled and releases the lock. To solve this problem, we relied on changing the mutex from a `std::mutex` to a futex with priority inheritance, which uses the kernel's implementation of an RT Mutex and helps to mitigate executor stalling.

Priority Assignment. Our framework implements a priority assignment strategy for both chains and individual callbacks. As discussed in Sec. 2, we classify chains into two distinct criticality classes: real-time (RT) and best-effort (BE). All BE chains and their callbacks have a criticality and priority level of 0, i.e., $\zeta_c = 0$ and $\forall \tau_i \in \Gamma_c : \pi_i = 0$. For real-time chains, chain priority is directly derived from its criticality level ($\zeta_c > 0$) through the criticality-as-priority assignment policy [15], i.e., chains with higher criticality get higher priority, as we want to minimize interference on critical chains. Within this scheme, we assign priorities to individual callbacks based on their chain membership using the PiCAS prioritization rule [8]: callback priority is determined first by chain priority, with callbacks from more critical chains receiving higher priorities; within each chain, priority increases along the execution path from first to last callback. For each chain, we establish its priority based on the priority of its final callback, creating end-to-end priority consistency. This dual-level priority assignment ensures that when RT threads apply priority-based scheduling, callbacks belonging to higher-criticality chains execute before those from lower-criticality chains. The BE threads, in contrast, ignore these priorities and execute callbacks in pseudo-round-robin fashion, aligning with the default ROS 2 execution model.

4.2 LaME Response Time Bounding

Our framework incorporates processing chains directly into the executor's implementation, rather than treating them as an external concept. This integration provides a concrete representation of the callback dependencies and data flows within the system. By tracking these relationships, the executor can identify which callbacks form a chain and monitor their collective execution behavior.

LaME collects practical metrics such as callback execution times and end-to-end chain response times, which serve as essential inputs for response time analysis. From the executor thread's perspective, the callback execution time is measured using the thread-local timer (`CLOCK_THREAD_CPUTIME_ID`), capturing only the cpu-time used to execute the callback while the thread is running, ignoring preemption and interference. This enables us to measure the exact execution time of the callback as if it were run in isolation. This data enables the executor to apply appropriate scheduling decisions based on actual system behavior. Additionally, with chains as explicit entities, the executor can implement consistent priority assignments for related callbacks, ensuring that callbacks within the same chain receive coordinated scheduling treatment.

These mechanisms create a direct connection between the theoretical response time analysis and the practical execution environment. By making chains "visible" to the executor, we establish the foundation needed for applying systematic response time bounding to both RT and BE workloads in ROS 2.

Response-Time Analysis for RT Chains Our framework implements WCRT analysis to ensure the schedulability of RT threadclasses during resource allocation and online adaptation. Although our framework design allows the use of other analyses, we adopt the latest chain response-time analysis developed for multi-threaded ROS 2 executors [19].

Specifically, we used Theorem 2 in [19], which computes the response-time bounds for chains under priority-driven multi-threaded executors when chains have constrained deadlines.

Theorem 1 (from [19]). *The response time of a chain $\Gamma_c = [\tau_{c_1}, \tau_{c_2}, ..., \tau_{c_n}]$ with a **constrained deadline** on a **priority-driven** ROS 2 executor with $|\Pi|$ threads is upper-bounded by $R_c = \Delta + \overline{sbf}_k(E_{c_n} - 1)$, if $dbf(\Delta) < sbf_\Pi(\Delta)$ holds for the following demand bound function $dbf(\Delta)$:*

$$dbf(\Delta) = |\Pi| \cdot (E_c - E_{c_n}) + \sum_{\substack{\Gamma_x \in S_{RT} \setminus \{\Gamma_c\} \\ \wedge \pi_x > \pi_c}} W_x(\Delta, D_x - E_x) + \sum_{\forall \tau_l \in mlp(\tau_{c_1})} \min(E_l - 1, \Delta)$$

where S_{RT} is the set of linear chains assigned to Π, π_x is the priority of a chain Γ_x ($\pi_x = \zeta_x$ due to our criticality-as-priority assignment), $\overline{sbf}_k(x)$ is the pseudo-inverse function of sbf_k ($\overline{sbf}_k(x) = \min\{\Delta | sbf_k(\Delta) = x\}$), $W_x()$ is the workload function for a constrained-deadline chain (Lemma 5 in [19]), and $mlp(\tau_{c_1})$ returns at most $|\Pi|$ callbacks with lower-priority than τ_{c_1} with one callback from each chain in S.

The above theorem is directly applicable to each RT threadclass Π since all chains on Π are RT chains with constrained deadlines and all threads $r_k \in \Pi$ share the same budget and replenishment period by design. We will explain how to linearize nonlinear chains and how to obtain S_{RT} later in this subsection.

Ensuring Starvation Freedom for BE Chains Our framework aims to provide fair access to system resources for BE chains even when RT chains are present. While the BE thread scheduling policy itself provides basic fairness, we extend this by theoretically ensuring starvation freedom.

As we detail in Section 4.3, our controller analyzes and allocates CPU budgets to different threadclasses. Within this process, we apply response time analysis based on [19], specifically Theorem 1 and 3, to check if chains assigned to BE threadclasses can complete within a controller period.[1]

Theorem 2 (from [19]). *The response time of a chain $\Gamma_c = [\tau_{c_1}, \tau_{c_2}, ..., \tau_{c_n}]$ with a **constrained deadline** on a **standard** ROS 2 executor with $|\Pi|$ threads is upper bounded by $R_c = \Delta + \overline{sbf}_k(E_{c_n} - 1)$, if $dbf(\Delta) < sbf_\Pi(\Delta)$ holds for the following $dbf(\Delta)$:*

$$dbf(\Delta) = |\Pi| \cdot (E_c - E_{c_n}) + \sum_{\Gamma_x \in S_{BE} - \{\Gamma_c\}} W_x(\Delta, D_x - E_x)$$

where S_{BE} is the set of linear chains assigned to Π.

Theorem 3 (from [19]). *The response time of a chain $\Gamma_c = [\tau_{c_1}, \tau_{c_2}, ..., \tau_{c_n}]$ with an **arbitrary deadline** on a **standard** ROS 2 executor with $|\Pi|$ threads is upper-bounded by $R_c = \Delta + \overline{sbf}_k(E_{c_n} - 1)$, if $dbf(\Delta) < sbf_\Pi(\Delta)$ holds for the following $dbf(\Delta)$:*

$$dbf(\Delta) = |\Pi| \cdot (E_c - E_{c_n}) + (\sum_{\Gamma_x \in S_{BE}} W^*_x(\Delta, D_x - E_x)) - E_c$$

where $W^*_x()$ is the workload for an arbitrary-deadline chain (Lemma 6 in [19]).

[1] The authors of [22] pointed out a flaw in the analysis of the default ROS scheduling policy in [12] and [19]. The authors of [19] have since released an amended version of their analysis, correcting the error in bounding response times. We reference this updated version of their analysis in this paper.

Algorithm 1 Linearize and categorize chains

```
1: Input: Original chainset including nonlinear chains ($S$)
2: Output: Linearized RT and BE chainsets ($S_{RT}$ and $S_{BE}$)
3: $S^{'} \leftarrow \emptyset$; $S_{RT} \leftarrow \emptyset$; $S_{BE} \leftarrow \emptyset$
4: for all $\mathit{\Gamma} \in S$ do  // Linearize all chains in $S$
5:     if $\mathit{\Gamma}$ is nonlinear then
6:         $S^{'} \leftarrow S^{'} \cup \ linearize(\mathit{\Gamma})$
7:     else
8:         $S^{'} \leftarrow S^{'} \cup \{\mathit{\Gamma}\}$
9:     end if
10: end for
11: for all $\mathit{\Gamma} \in S^{'}$ do  // Categorize chains into RT/BE
12:     if $\mathit{\Gamma}$ is RT then
13:         $S_{RT} \leftarrow S_{RT} \cup \mathit{\Gamma}$
14:     else
15:         $S_{BE} \leftarrow S_{BE} \cup \mathit{\Gamma}$
16:     end if
17: end for
18: return $S_{RT}$ and $S_{BE}$
```

For each BE chain on a BE threadclass, we first apply Theorem 2 and check if the response time is bounded to be within its period. If any chain's response time exceeds its period, this means that multiple instances of the same chain may co-exist at runtime (in other words, the execution of previous or next instances of a chain may overlap) because we do not enforce chain-level instance skipping. Hence, we switch to Theorem 3 to take into account such cases and check if the response time is bounded by the controller period. If all BE chains pass this test, we can determine that the given budget for the BE threadclass is sufficient to guarantee that every BE chain can execute at least once within each controller period, ensuring starvation freedom.

However, under overloaded conditions, we may not be able to ensure starvation freedom for all BE threadclasses. In these cases, our system still offers fairness through the default ROS 2 scheduling policy. This combination of analytical budget allocation and scheduling policy fairness ensures that BE chains receive appropriate system resources proportional to their execution demands, even when theoretical starvation freedom cannot be guaranteed.

Analysis for Nonlinear Chains in Different Threadclasses When we encounter nonlinear chains, our executor must decompose them into linear subchains to make them amenable to analysis. This decomposition is particularly important when the branches of a nonlinear chain belong to different criticality classes (RT and BE), which must be assigned to threadclasses governed by different analyses.

Figure 3 illustrates this concept with a single nonlinear chain $\mathit{\Gamma}_1$ containing both RT and BE components. The chain begins with three callbacks (shown in green) that are shared by both components. After the third callback, the chain

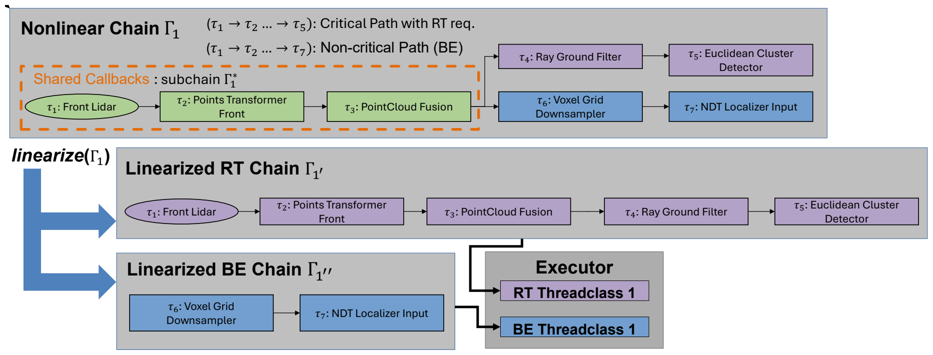

Nonlinear Chain Γ_1
$(\tau_1 \rightarrow \tau_2 \ldots \rightarrow \tau_5)$: Critical Path with RT req.
$(\tau_1 \rightarrow \tau_2 \ldots \rightarrow \tau_7)$: Non-critical Path (BE)
Shared Callbacks : subchain Γ_1^*
τ_1: Front Lidar
τ_2: Points Transformer Front
τ_3: PointCloud Fusion
τ_4: Ray Ground Filter
τ_5: Euclidean Cluster Detector
τ_6: Voxel Grid Downsampler
τ_7: NDT Localizer Input
linearize(Γ_1)
Linearized RT Chain $\Gamma_{1'}$
Linearized BE Chain $\Gamma_{1''}$
Executor
RT Threadclass 1
BE Threadclass 1

Fig. 3: Example of nonlinear chain splitting.

branches into two paths: an RT branch (violet) and a BE branch (blue). These branches are assigned to their respective threadclasses based on their criticality class. For the shared callbacks (green; subchain Γ_1^*), our policy merges them with one of the RT branches—specifically, the very first RT branch when multiple exist—to ensure they execute on an RT threadclass for prioritized processing; if no RT branch exists, they are merged with one of the BE branches.

Algorithm 1 shows the process of linearizing and categorizing chains for analysis and resource allocation purposes. It first iterates over all chains in the input set S (line 4). For each nonlinear chain, it applies the aforementioned linearization procedure (denoted as $linearize(\Gamma)$) to decompose it into linear subchains, adding these to an intermediate set S' (line 6); linear chains are added directly (line 8). Next, it categorizes the linearized chains in S' by adding RT chains to S_{RT} and BE chains to S_{BE} (line 13-15). This ensures that subsequent analysis can handle the subchains appropriately within their respective threadclasses.

This decomposition creates an analytical challenge: the response time of a chain depends on callbacks potentially assigned to a different threadclass. For instance, when analyzing the BE subchain (blue), we must account for the response time of the shared root callbacks (green) that are assigned to the RT threadclass. This transforms our requirements from a simple per-threadclass approach to one that must consider multiple independently scheduled threadclasses.

To solve this problem, we adopt the Compositional Performance Analysis (CPA) approach used in prior work [6, 19]. When analyzing the linearized RT chain, e.g., $\Gamma_{1'}$ in Figure 3 comprising five callbacks in total, we first perform a partial analysis on the shared callbacks, e.g., the first three callbacks in the figure. This involves creating a virtual chainset that includes all other chains in the threadclass, plus a partial chain containing only the callbacks shared across subchains. The partial response time is stored and later added to the response time calculation of subchains in other threadclasses. This compositional approach allows us to accurately bound the end-to-end response time of nonlinear chains spanning multiple threadclasses, effectively treating them as if they were scheduled on separate executors. Note that this procedure is for analysis and resource allocation purposes only, and callback inputs and outputs remain unaffected.

4.3 Adaptive Resource Controller

Having outlined the key elements of the LaME framework, we now detail the components that work together to regulate its execution behavior. The controller runs as a standalone thread with non real-time priority Linux to prevent interference with executor threads. It operates concurrently with the executor and continuously monitoring it for feedback. By default, the controller activates periodically with a configurable interval (10s in our implementation) to assess system performance (e.g., changes in callback execution time). Additionally, the controller is triggered immediately whenever a real-time chain timing violation is detected, enabling prompt response to deadline misses.

The controller regulates three key parameters: the chain-to-threadclass allocation, the threads-to-threadclass allocation (i.e., the number of threads per threadclass), and the budgets assigned to threadclasses.

These tasks are interrelated; for instance, changing the chain allocation of a real-time threadclass may change the amount of budget or threads required for that threadclass to execute its chains within their deadlines. Because of this inter-dependence, finding an optimal solution would not be tractable, especially when using an online adaptation of the analysis where the runtime must be constrained.

As a runtime solution, we propose a two-step approach to managing these parameters. The flowchart portion of Figure 1 illustrates the controller's execution logic. The first step starts by using the analysis approach in Sec. 4.2 to perform chain-to-threadclass allocation for RT chains (Alg. 2), while simultaneously determining threadclass parameters—such as the number of threads per threadclass—through a merging process that groups threads from multiple cores into fewer, larger threadclasses when needed to ensure schedulability. For this part, we perform the response time analysis on each RT threadclass while assuming that it has a budget of 100%. Once the RT chains have been assigned to their threadclasses, we perform the BE chain-to-threadclass allocation (Alg. 2; details will follow). The second step minimizes the budgets of the RT threadclasses while ensuring that their chainsets remain analytically schedulable, and subsequently checks starvation freedom for BE threadclasses by using the analysis in Sec. 4.2 (Alg. 3). Once these budgets are reduced, the controller continuously monitors the executor for real-time chain timing violations and verifies best-effort chain starvation freedom according to its activation period.

If a timing violation does occur, the controller adjusts the thread budgets to guarantee response times for RT chains. Should this adjustment fail—for instance, due to reduced resource capacity from system changes like thermal throttling [11]—the controller will repeat the chain-to-threadclass allocation and budget adjustments, adapting the system to maintain performance under the new conditions. Of course, if the system is entirely overloaded, and bounds cannot be generated for RT threadclasses, we degrade the least critical RT chains to BE mode in order to preserve the schedulability of more critical real-time chains.

Dynamic Chain-to-Threadclass Allocation Alg. 2 depicts our algorithm for creating threadclasses and performing chain-to-threadclass allocation. The

Algorithm 2 Chain-to-Threadclass Allocation

```
1: **Input:** Set of CPU cores ($\mathcal{M}$), RT and BE chainsets ($S_{RT}$ and $S_{BE}$)
2: **Output:** Threadclass allocations for RT and BE chains ($\mathcal{P}_{RT}$ and $\mathcal{P}_{BE}$)
3: $\mathcal{P}_{RT} \leftarrow \emptyset$; $\mathcal{P}_{BE} \leftarrow \emptyset$ // Sets of RT and BE threadclasses
4: for all $m \in \mathcal{M}$ do
5:    Create two threads ($r$ and $r'$) pinned to core $m$
6:    $\Pi_{RT} \leftarrow \{r\}$; $\Pi_{BE} \leftarrow \{r'\}$
7:    $\mathcal{P}_{RT} \leftarrow \mathcal{P}_{RT} \cup \{\Pi_{RT}\}$; $\mathcal{P}_{BE} \leftarrow \mathcal{P}_{BE} \cup \{\Pi_{BE}\}$
8: end for
9: retry:
10: $\forall \Pi \in \mathcal{P}_{RT} : \Pi.chains \leftarrow \emptyset$; $\forall \Pi \in \mathcal{P}_{BE} : \Pi.chains \leftarrow \emptyset$
11: for all $\Gamma_{RT} \in S_{RT}$ in descending chain-priority order do
12:    $\mathcal{P}_{visited} \leftarrow \emptyset$
13:    $\mathcal{P}^*_{RT} \leftarrow \mathcal{P}_{RT}$; $\mathcal{P}^*_{BE} \leftarrow \mathcal{P}_{BE}$ // Make copies of $\mathcal{P}_{RT}$ and $\mathcal{P}_{BE}$
14: next_threadclass:
15:    $\Pi \leftarrow lowest\_util(\mathcal{P}_{RT} \setminus \mathcal{P}_{visited})$ // Find $\Pi$ w/ lowest per-thread utilization
16:    if $\Gamma_{RT}$ is schedulable on $\Pi$ then
17:       $\Pi.chains \leftarrow \Pi.chains \cup \{\Gamma_{RT}\}$
18:    else
19:       if $\mathcal{P}_{RT} \neq \mathcal{P}_{visited}$ then
20:          $\mathcal{P}_{visited} \leftarrow \mathcal{P}_{visited} \cup \{\Pi\}$
21:          goto: next_threadclass
22:       end if
23:       // All $\Pi \in \mathcal{P}_{RT}$ have been visited; Merge and retry
24:       if $|\mathcal{P}_{RT}| == 1$ then // Nothing to merge; Demote $\Gamma_{RT}$ to BE
25:          $S_{RT} \leftarrow S_{RT} \setminus \{\Gamma_{curr}\}$; $S_{BE} \leftarrow S_{BE} \cup \{\Gamma_{curr}\}$
26:          $\mathcal{P}_{RT} \leftarrow \mathcal{P}^*_{RT}$; $\mathcal{P}_{BE} \leftarrow \mathcal{P}^*_{BE}$ // Restore merges done for $\Gamma_{RT}$
27:          continue
28:       end if
29:       // $merge(\Pi_1, \Pi_2)$: Merges chains and threads from $\Pi_1$ and $\Pi_2$
30:       // $corresponding\_be(\Pi_{RT})$: BE threadclass on the same cpu as $\Pi_{RT}$
31:       $\Pi_1 \leftarrow lowest\_util(\mathcal{P}_{RT})$; $\Pi_2 \leftarrow lowest\_util(\mathcal{P}_{RT} \setminus \{\Pi_1\})$
32:       $\Pi_1 \leftarrow merge(\Pi_1, \Pi_2)$; $\mathcal{P}_{RT} \leftarrow \mathcal{P}_{RT} \setminus \{\Pi_2\}$
33:       $\Pi_3 \leftarrow corresponding\_be(\Pi_1)$; $\Pi_4 \leftarrow corresponding\_be(\Pi_2)$
34:       $\Pi_3 \leftarrow merge(\Pi_3, \Pi_4)$; $\mathcal{P}_{BE} \leftarrow \mathcal{P}_{BE} \setminus \{\Pi_4\}$
35:       goto: retry
36:    end if
37: end for
38: for all $\Gamma_{BE} \in S_{BE}$ do
39:    $\Pi \leftarrow lowest\_util(\mathcal{P}_{BE})$
40:    $\Pi.chains \leftarrow \Pi.chains \cup \{\Gamma_{BE}\}$
41: end for
```

process begins by initializing empty sets for RT and BE threadclasses ($\mathcal{P}_{RT}$ and $\mathcal{P}_{BE}$, respectively). For each CPU core in the input set $\mathcal{M}$, the algorithm creates two threads pinned to that core: one for RT and one for BE. It then forms a single-threaded RT threadclass (Π_{RT}) containing the RT thread and a single-threaded BE threadclass ($\mathit{\Pi}_{BE}$) containing the BE thread, adding both to their respective sets (lines 4-8). At this point, the RT thread gets 100% budget while the BE thread gets 0%; the budget adjustment is performed later by Alg. 3.

Next, the algorithm allocates RT chains to RT threadclasses using a worst-fit decreasing strategy, prioritizing chains in descending order of criticality (lines 9-37). At the start of each allocation attempt, it clears any existing chain assignments from all threadclasses in $\mathcal{P}_{RT}$ and $\mathcal{P}_{BE}$ (line 10). For each RT chain $\mathit{\Gamma}_{RT}$, it identifies the RT threadclass $\mathit{\Pi}$ with the lowest per-thread utilization among those not yet visited for this chain. If adding $\mathit{\Gamma}_{RT}$ to $\mathit{\Pi}$ keeps the threadclass schedulable (based on the analysis given by Theorem 1), the chain is assigned there (line 17). Otherwise, it marks the threadclass as visited (lines 19-22) and retries with the next lowest-utilization threadclass (line 15). If no suitable threadclass is found after checking all, the algorithm tries merging some threadclasses (e.g., merging two single-threaded RT threadclasses into one multi-threaded RT threadclass), as doing so might accommodate the chain $\mathit{\Gamma}_{RT}$ (lines 23-35).

Before merging threadclasses, the algorithm checks if only one RT threadclass remains in $\mathcal{P}_{RT}$ (i.e., all threadclasses have been merged); if so, it demotes the current chain $\mathit{\Gamma}_{curr}$ to BE status by removing it from S_{RT} and adding it to S_{BE}, restore all merges that would have been done for $\mathit{\Gamma}_{RT}$ using the copies $\mathcal{P}^*_{RT}$ and $\mathcal{P}^*_{BE}$, and moves on to the next chain (lines 24-28). Otherwise, it merges the two RT threadclasses with the lowest utilizations ($\mathit{\Pi}_1$ and $\mathit{\Pi}_2$) into $\mathit{\Pi}_1$, removes $\mathit{\Pi}_2$ from $\mathcal{P}_{RT}$, and performs a corresponding merge for their paired BE threadclasses ($\mathit{\Pi}_3$ and $\mathit{\Pi}_4$) into $\mathit{\Pi}_3$, removing $\mathit{\Pi}_4$ from $\mathcal{P}_{BE}$. Then, the RT allocation retries from the beginning to verify chain schedulability post-merge.

Finally, BE chains are allocated to BE threadclasses using a similar worst-fit strategy: each $\mathit{\Gamma}_{BE}$ is assigned to the BE threadclass with the lowest utilization, without schedulability checks during allocation (though starvation freedom is verified later via response-time analysis) (lines 38-41). This ensures even workload distribution while minimizing overload risks.

Dynamic Threadclass Budget Adjustment Now we reduce the budget of RT threadclasses to minimize the resources allocated to them while ensuring schedulability, allowing the remaining CPU budget on each core to be reassigned to the corresponding BE threadclass counterpart for improved best-effort performance. Hence, this adjustment process focuses on finding the minimal schedulable `SCHED_DEADLINE` budget for each RT threadclass.

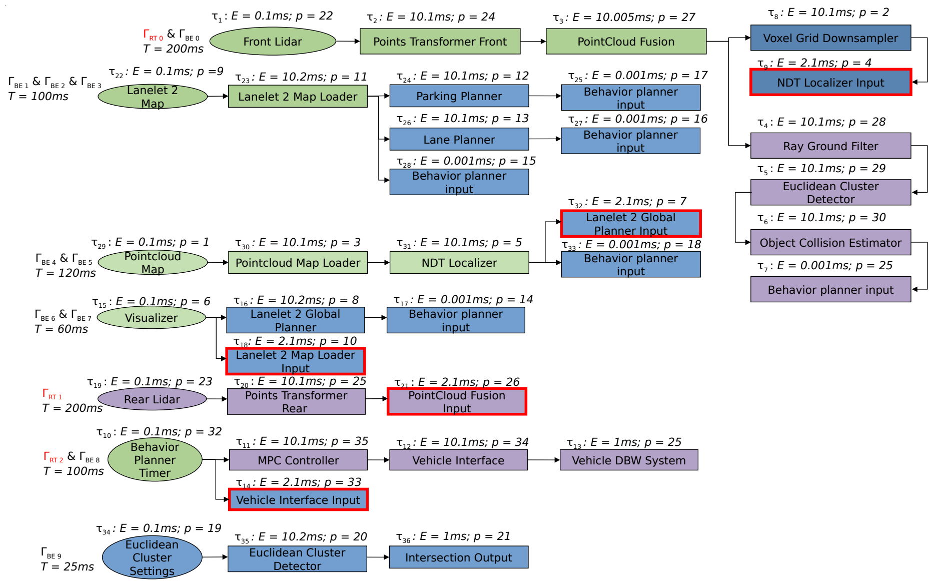
We present our algorithm for the threadclass budget adjustment in Alg. 3. We execute this algorithm for each RT threadclass $\mathit{\Pi} \in \mathcal{P}_{RT}$. If an RT threadclass has no assigned chains (lines 5- 7), we merge it and its corresponding BE counterpart with the RT threadclass that has the maximum per-thread utilization and its corresponding BE counterpart. This merger consolidates resources, preventing idle threadclasses and ensuring efficient use of cores by combining underutilized or empty threadclasses with busier ones.

As shown in lines 8-23, the adjustment mechanism utilizes a binary search approach on the budget range, starting from a minimum value and being limited to 100%, represented by the system-defined period. To control the search gran-

Algorithm 3 Reduce Real-Time Threadclass Budget

```
1: Input: An RT threadclass ($\Pi_{RT} \in \mathcal{P}_{RT}$), Budget step size ($step$)
2: $min\_budget \leftarrow 0\%$; $max\_budget \leftarrow 100\%$; $best\_budget \leftarrow 0\%$ // Per-thread budget
3: // $max\_util(\mathcal{P})$: returns a threadclass ($\Pi \in \mathcal{P}$) with the max per-thread utilization
4: // $merge(\Pi_1, \Pi_2)$: merges RT threadclasses and their BE counterparts
5: if $\Pi_{RT}.chains = \emptyset$ then
6:     $\Pi_{RT} \leftarrow merge(\Pi, max\_util(\mathcal{P}_{RT} \setminus \{\Pi_{RT}\}))$
7: end if
8: while $min\_budget < max\_budget$ do
9:     $mid\_budget \leftarrow \frac{min\_budget + max\_budget}{2}$
10:    $schedulable \leftarrow$ true
11:    for all $\Gamma_c \in \Pi_{RT}.chains$ in descending chain-priority order do
12:        Compute $R_c$ using Theorem 1, with $C^r_k = mid\_budget \cdot T^r_k$ for $\forall r_k \in \Pi_{RT}$
13:        if $R_c > D_c$ then
14:            $schedulable \leftarrow$ false
15:            break
16:        end if
17:    end for
18:    if $schedulable$ then
19:        $max\_budget \leftarrow mid\_budget$
20:    else
21:        $min\_budget \leftarrow mid\_budget + step$
22:    end if
23: end while
24: $best\_budget \leftarrow \max(min\_budget, \min(mid\_budget, max\_budget))$
25: for all $r_k \in \Pi_{RT}$ do
26:     $C^r_k \leftarrow best\_budget \cdot T^r_k$
27: end for
28: $remaining\_budget \leftarrow 100\% - best\_budget$
29: $\Pi_{BE} \leftarrow corresponding\_be(\Pi_{RT})$
30: for all $r_k \in \Pi_{BE}$ do
31:     $C^r_k \leftarrow remaining\_budget \cdot T^r_k$
32: end for
33: Check $R_c$ for $\forall \Gamma_c \in \Pi_{BE}$ using Theorems 2 and 3
```

ularity, we use a step size (denoted as *step* in the algorithm), which increments the minimum budget when the midpoint is unschedulable; in our implementation, we set $step = 1\%$ to balance precision and analysis overhead, as discussed further in Section 5 regarding the controller overhead costs. We search for the optimal budget for the threadclass by iteratively evaluating the schedulability of the chainset through an online response-time analysis explained in Sec. 4.2. This iteration continues until the system converges on the least possible budget that maintains schedulability for all chains assigned to the threadclass. Line 24 determines the optimal budget after the binary search converges, selecting the highest lower bound (*via* max) while ensuring it does not exceed the last known schedulable midpoint (*via* $\min(mid\_budget, max\_budget)$), as mid_budget retains the value from the final iteration where schedulability was tested. The while



$\Gamma_{RT\,0}$ & $\Gamma_{BE\,0}$ T = 200ms
τ_1 : E = 0.1ms; p = 22
Front Lidar
τ_2 : E = 10.1ms; p = 24
Points Transformer Front
τ_3 : E = 10.005ms; p = 27
PointCloud Fusion
τ_8 : E = 10.1ms; p = 2
Voxel Grid Downsampler
τ_9 : E = 2.1ms; p = 4
NDT Localizer Input
τ_4 : E = 10.1ms; p = 28
Ray Ground Filter
τ_5 : E = 10.1ms; p = 29
Euclidean Cluster Detector
τ_6 : E = 10.1ms; p = 30
Object Collision Estimator
τ_7 : E = 0.001ms; p = 25
Behavior planner input
$\Gamma_{BE\,1}$ & $\Gamma_{BE\,2}$ & $\Gamma_{BE\,3}$ T = 100ms
τ_{22} : E = 0.1ms; p =9
Lanelet 2 Map
τ_{23} : E = 10.2ms; p = 11
Lanelet 2 Map Loader
τ_{24} : E = 10.1ms; p = 12
Parking Planner
τ_{25} : E = 0.001ms; p = 17
Behavior planner input
τ_{26} : E = 10.1ms; p = 13
Lane Planner
τ_{27} : E = 0.001ms; p = 16
Behavior planner input
τ_{28} : E = 0.001ms; p = 15
Behavior planner input
τ_{32} : E = 2.1ms; p = 7
Lanelet 2 Global Planner Input
$\Gamma_{BE\,4}$ & $\Gamma_{BE\,5}$ T = 120ms
τ_{29} : E = 0.1ms; p = 1
Pointcloud Map
τ_{30} : E = 10.1ms; p = 3
Pointcloud Map Loader
τ_{31} : E = 10.1ms; p = 5
NDT Localizer
τ_{33} : E = 0.001ms; p = 18
Behavior planner input
$\Gamma_{BE\,6}$ & $\Gamma_{BE\,7}$ T = 60ms
τ_{15} : E = 0.1ms; p = 6
Visualizer
τ_{16} : E = 10.2ms; p = 8
Lanelet 2 Global Planner
τ_{17} : E = 0.001ms; p = 14
Behavior planner input
τ_{18} : E = 2.1ms; p = 10
Lanelet 2 Map Loader Input
$\Gamma_{RT\,1}$ T = 200ms
τ_{19} : E = 0.1ms; p = 23
Rear Lidar
τ_{20} : E = 10.1ms; p = 25
Points Transformer Rear
τ_{21} : E = 2.1ms; p = 26
PointCloud Fusion Input
$\Gamma_{RT\,2}$ & $\Gamma_{BE\,8}$ T = 100ms
τ_{10} : E = 0.1ms; p = 32
Behavior Planner Timer
τ_{11} : E = 10.1ms; p = 35
MPC Controller
τ_{12} : E = 10.1ms; p = 34
Vehicle Interface
τ_{13} : E = 1ms; p = 25
Vehicle DBW System
τ_{14} : E = 2.1ms; p = 33
Vehicle Interface Input
$\Gamma_{BE\,9}$ T = 25ms
τ_{34} : E = 0.1ms; p = 19
Euclidean Cluster Settings
τ_{35} : E = 10.2ms; p = 20
Euclidean Cluster Detector
τ_{36} : E = 1ms; p = 21
Intersection Output

Fig. 4: Individual Chain Breakdown for the Modified Autoware Reference System

loop exits either when $min\_budget \geq max\_budget$ (convergence on a feasible budget) or implicitly through the search process, but not via a direct break for unschedulability in the outer loop; since schedulability checks use mid_budget, it preserves the last viable budget when the loop ends. Even in the worst case, where no reduced budget maintains schedulability, the algorithm ensures Π remains schedulable by falling back to a higher budget (up to 100%), as prior allocation steps in Alg. 2 have verified feasibility at full capacity.

Once the optimal budget is determined, it is applied to the threadclass, and the corresponding executor threads within the threadclass are configured accordingly with `SCHED_DEADLINE` parameters. Additionally, the remaining CPU-time after assigning the RT threadclass its budget is assigned to its BE threadclass counterpart, whose threads are scheduled on the same CPU cores as those of the RT threadclass. Immediately after assigning BE threadclass budgets, we perform the response time analysis on those threadclasses, as explained in Sec. 4.2, to check if their chain response times are bounded and free from starvation.

5 Evaluation

In this section, we evaluate the performance of ROS 2 applications under our control framework and profile the associated overhead. The evaluation was performed on the Nvidia Jetson AGX Xavier embedded platform. [2]

[2] The source code of our implementation is available at `https://github.com/rtenlab/reference-system-latency-management`.

5.1 System Setup

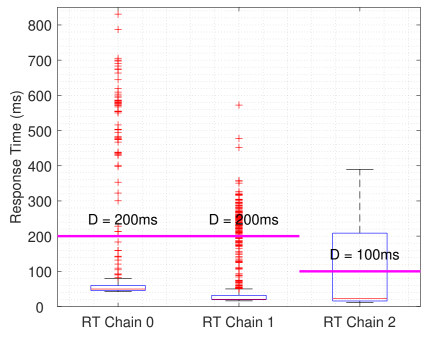
Our experiments use a modified version of the Autoware reference system [1] deployed on a ROS 2 executor. For all experiments, the system was run using the default DDS settings, with DDS threads having standard priority in Linux and free to migrate among cores. Note that the Autoware reference system has a utilization demand exceeding 1.0 (one CPU core), which prevents direct comparison with ROS-Llama [5]. As discussed in Sec. 3, ROS-Llama supports only single-threaded executors, so comparing against it would require manually partitioning chains and assigning them across multiple executors, introducing subjectivity that could confound any performance comparison.

Modifications to subscription-based fusion nodes: Since the original Autoware reference system contains several subscription-based fusion callbacks which existing chain-based analysis techniques [3, 6, 8, 19] cannot directly analyze, we needed to modify it. The default behavior for subscription-based fusion nodes is for the callback immediately proceeding a fusion callback can be triggered by either of its input callbacks, as long as the other input callback has already executed. This created a problem for us to use the analysis from [19] to bound chain response times since it is not possible to extrapolate periodic models of each chain. Therefore, we segmented the nonlinear subscription-based fusion nodes into two separate classes of fusion callbacks: input and trigger, following the approach taken in prior work [21, 23]. Specifically, input callbacks are constrained to only caching the input from the previous callback in the chain, whereas the trigger callbacks always trigger the subsequent callbacks in the chain. The explicit chains that we were able to define after using this modified chain configuration are shown in Figure 4. Indicated by red outlined boxes in the figure, we show the callbacks that were modified from subscription-based fusion nodes to be input callbacks to fusion operations.

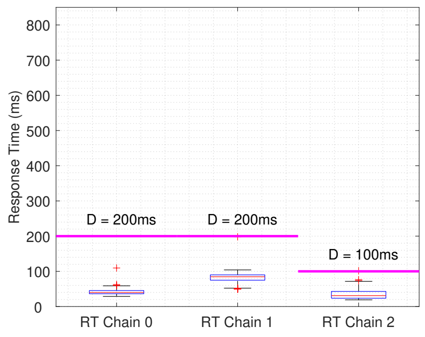
Chain Breakdown: There are 13 processing chains, divided into 3 real-time (RT) and 10 best-effort (BE) chains. The RT chains, shown in violet boxes in Figure 4, are comprised of callbacks on the reference system's *hot path* and behavior planner chains, where the hot path chain is considered more critical than the behavior planner chain [1]. As shown in the figure, the hot path chains are identified as $\Gamma_{RT\ 0}$ and $\Gamma_{RT\ 1}$, while the behavior planner chain is identified as $\Gamma_{RT\ 2}$. The BE chains, shown in blue boxes in the figure, are the remainder of the best-effort chains in the system. The green boxes in the figure indicate callbacks shared among multiple subchains of a nonlinear chain. Each callback in the reference system contains a synthetic prime number search workload whose WCET varies by the upper limit of the prime search. The worst observed execution times, as measured by our executor, associated with each of the callbacks' workloads are shown by E in Figure 4.

5.2 Experiment Scenario 1: Autoware Reference System

This experiment was designed to compare the performance of the Autoware Reference System, described in Figure 4, under arbitrary overloads on the default

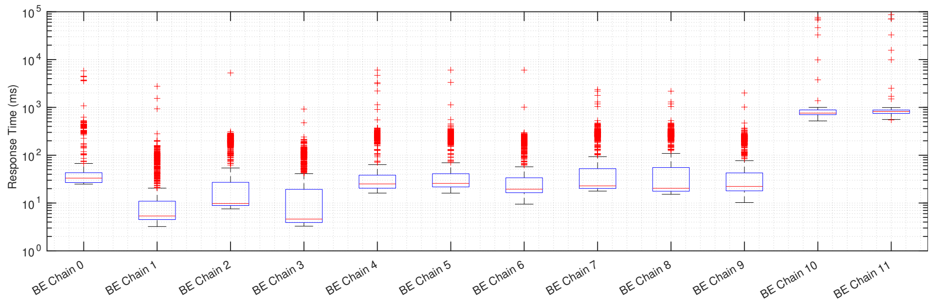


(a) Default ROS 2 Executor

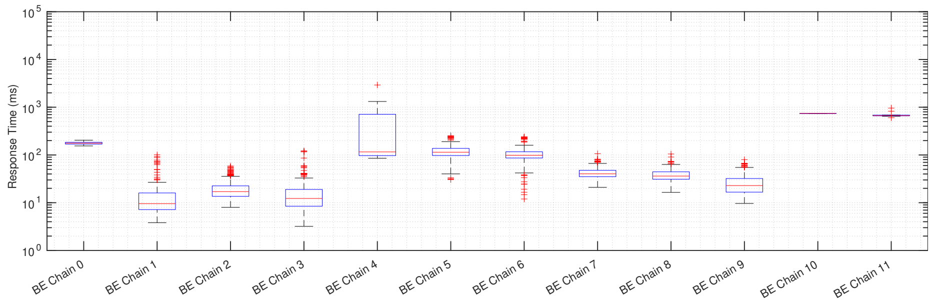


(b) LaME (proposed work)

Fig. 5: RT chain response times with the default ROS 2 and LaME executors. Pink horizontal lines indicate RT chain deadlines.



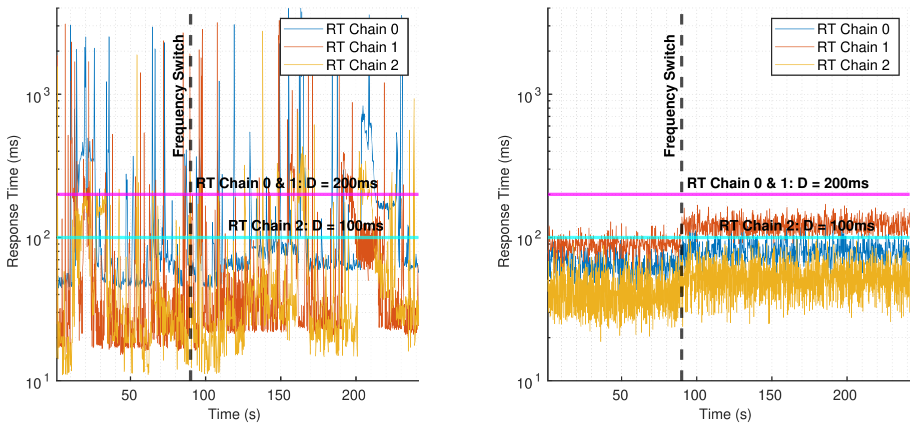
(a) Default ROS 2 Executor



(b) LaME (proposed work)

Fig. 6: BE chain response times under the default ROS 2 and LaME executors.

ROS 2 multi-threaded executor and our LaME executor. Both executors were configured to run on 3 CPU cores clocked at 2.3 GHz. The default ROS 2 executor was created with 3 threads, each given 100% CPU budget with 10ms replenishment period and real-time priority by `SCHED_DEADLINE`. Our LaME executor was limited to using up to 3 cores. Without any overload, we confirmed that both executors could meet the timing requirements of all RT and BE chains of the Autoware reference system. While the observed response times for both classes of chains were below their deadlines, the BE and RT chains were not



(a) Default ROS 2 Executor (b) LaME (proposed work)

Fig. 7: RT Chain Response Times under Frequency Throttling

bounded under the analysis for the default ROS 2 executor. To experiment with an overloaded case, we duplicated two chains, Γ_4 and Γ_5, and then modified the upper limit of the prime number search workload such that the execution time of each duplicated callback was increased to $100ms$.

Results: Figure 5b and Figure 5a, we report the measured response times for the RT chains under both executors. In this case, we show that the LaME executor outperforms the default executor, in the worst case for RT chains, by over 400% (850ms to 175ms). This is a direct result of our framework's ability to isolate workloads by threadclasses and strictly enforce their resource reservations, even in very unfavorable conditions. Furthermore, in Figure 6b, and Figure 6a, we show the response time for the BE chain under both executors. In this case, we designed the scenario such that all BE threadclasses are overloaded, with their chain's having unbounded response times. Nonetheless, the LaME executor shows much more consistent worst-observed response times for BE chains, also driven by LaME's workload isolation and resource allocation algorithms.

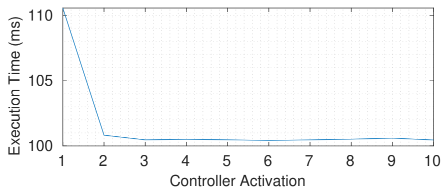
5.3 Experiment Scenario 2: Online Frequency Throttling

In this scenario, we performed the same experiment as in scenario 1, but we started the executor when the CPU frequency was pinned to 2.3GHz. After 90 seconds, the CPU frequency was changed to 1.4GHz. Figure 7b and Figure 7a, show the measured response-time over time plot for the three critical chains in the reference system with different executors.

Results: In Figure 7 we observe that there was an increase in reported response times after the frequency switch, as expected. As shown in Figure 7b, chains that are running on our executor report some timing violations immediately after the switch. This triggers the controller to reallocate CPU resources according to the increased demand of the chains. After several seconds, we show that all three chains are able to meet their deadlines without committing further timing violations, albeit with a higher average case response time. In Figure 7a, we see the same increase in the average-case response times of the RT chains, and we observe a huge increase in deadline misses.

5.4 Controller Runtime Overhead

In Figure 8, we show an example of the controller component of our executor's runtime. Figure 8a shows the activation time when it is evaluating the response times of the chains using $10ms$ as the replenishment period for the thread budgets during an experiment. In this case, we set the controller to activate every 10 seconds, to periodically verify starvation freedom for BE chains. Figure 8b shows the breakdown of the controller's overhead by component for its initial instance. The controller performs response time analysis when first doing the chain-to-threadclass allocation, during budget reduction, when periodically verifying performance for BE threadclasses, and when adjusting runtime parameters for RT threadclasses that exhibit timing violations. The longest runtime for the controller is generally when it first performs the initial chain-to-threadclass allocation and budget reduction, as this requires the most invocations of the response time analysis. It is important to note that this overhead does not interfere with the execution of callbacks under LaME. The controller threads run in parallel with the executor threads and belong to the `SCHED_OTHER` scheduling class in Linux and, when possible, isolated to specific non-executor cores.



Operation	**Time (ms)**	**Percentage**
Thread class creation	0.185	0.17%
Chain-to-thread allocation	0.636	0.58%
Initial budget reduction	109.287	99.25%
Total	110.108	100%

(a) Controller Runtime Per Activation (b) Controller Overhead Breakdown

Fig. 8: Controller Runtime Overhead

6 Conclusion

We presented a theory-guided control framework and multi-threaded executor that extends ROS 2 scheduling with dynamic resource management, considering both real-time requirements for real-time chains and fairness for best-effort ones. Our experiments on an Nvidia Jetson platform highlight the performance of our executor under unfavorable conditions, where we observed up to 400% better and guaranteed worst-case response times for real-time chains, compared to the default ROS 2 executor.

Acknowledgment

This work was sponsored by the National Science Foundation (NSF) grants 1943265 and 2312395.

References

1. ROS2 Real-Time Working Group: Reference system. `https://github.com/ros-realtime/reference-system` (accessed March 2022)
2. Issue #2532: Executor callbacks are no longer in a predictable order (accessed Oct 2025), `https://github.com/ros2/rclcpp/issues/2532`
3. Al Arafat, A., Wilson, K., Yang, K., Guo, Z.: Dynamic priority scheduling of multithreaded ros 2 executor with shared resources. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **43**(11), 3732–3743 (2024)
4. Blass, T., Casini, D., Bozhko, S., Brandenburg, B.B.: A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance. In: IEEE Real-Time Systems Symposium (RTSS) (2021)
5. Blass, T., Hamann, A., Lange, R., Ziegenbein, D., Brandenburg, B.B.: Automatic latency management for ROS 2: Benefits, challenges, and open problems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2021)
6. Casini, D., Blaß, T., Lütkebohle, I., Brandenburg, B.: Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In: Euromicro Conference on Real-Time Systems (ECRTS) (2019)
7. Choi, H., Enright, D., Sobhani, H., Xiang, Y., Kim, H.: Priority-driven real-time scheduling in ROS 2: Potential and challenges. In: RAGE (2022)
8. Choi, H., Xiang, Y., Kim, H.: PiCAS: New design of priority-driven chain-aware scheduling for ROS2. In: 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS) (2021)
9. Enright, D., Xiang, Y., Choi, H., Kim, H.: PAAM: A Framework for Coordinated and Priority-Driven Accelerator Management in ROS 2. In: 2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS) (2024)
10. Guan, N., Yi, W.: General and efficient response time analysis for edf scheduling. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2014)
11. Hosseinimotlagh, S., Kim, H.: Thermal-aware servers for real-time tasks on multi-core GPU-integrated embedded systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 254–266. IEEE (2019)
12. Jiang, X., Ji, D., Guan, N., Li, R., Tang, Y., Wang, Y.: Real-Time Scheduling and Analysis of Processing Chains on Multi-threaded Executor in ROS 2. In: RTSS (2022)
13. Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y., Azumi, T.: Autoware on board: Enabling autonomous vehicles with embedded systems. In: ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS) (2018)
14. Kim, J., Kim, H., Lakshmanan, K., Rajkumar, R.: Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems (ICCPS) (2013)
15. de Niz, D., Lakshmanan, K., Rajkumar, R.: On the scheduling of mixed-criticality real-time task sets. In: 2009 30th IEEE Real-Time Systems Symposium. pp. 291–300 (2009). `https://doi.org/10.1109/RTSS.2009.46`
16. Park, J., Delgado, R., Choi, B.W.: Real-time characteristics of ros 2.0 in multiagent robot systems: an empirical study. IEEE Access **8**, 154637–154651 (2020)

17. Schowitz, P., Sinha, S., Gujarati, A.: Response-Time Analysis of a Soft Real-time NVIDIA Holoscan Application. In: 2024 IEEE Real-Time Systems Symposium (RTSS) (2024)
18. Shin, I., Lee, I.: Compositional real-time scheduling framework with periodic model. ACM Transactions on Embedded Computing Systems (TECS) **7**(3), 1–39 (2008)
19. Sobhani, H., Choi, H., Kim, H.: Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2023)
20. Tang, Y., Feng, Z., Guan, N., Jiang, X., Lv, M., Deng, Q., Yi, W.: Response time analysis and priority assignment of processing chains on ROS2 executors. In: IEEE Real-Time Systems Symposium (RTSS) (2020)
21. Teper, H., Günzel, M., Ueter, N., von der Brüggen, G., Chen, J.J.: End-to-end timing analysis in ROS2. In: 2022 IEEE Real-Time Systems Symposium (RTSS) (2022)
22. Teper, H., Kuhse, D., Günzel, M., von der Brüggen, G., Howar, F., Chen, J.J.: Thread Carefully: Preventing Starvation in the ROS 2 Multi-Threaded Executor. In: 2024 International Conference on Embedded Software (EMSOFT) (2024)
23. Toba, H., Azumi, T.: Deadline miss early detection method for dag tasks considering variable execution time. In: 36th Euromicro Conference on Real-Time Systems (ECRTS) (2024)