ATER: Adaptive Task Execution Rate Regulation for Enhanced Real-Time Performance in ROS 2

Ruoxiang Li*, Ziwei Song*, Mingsong Lv†, Jen-Ming Wu‡, Chun Jason Xue§, Jianping Wang*, and Nan Guan*
**City University of Hong Kong*, Hong Kong SAR
†*The Hong Kong Polytechnic University*, Hong Kong SAR
‡*Hon Hai Research Institute*, Taiwan
§*Mohamed bin Zayed University of Artificial Intelligence*, United Arab Emirates

***Abstract*—Autonomous systems, such as autonomous vehicles and robots, typically use middleware like Robot Operating System (ROS) to manage communication, execution, and functionality. These systems sample sensing data at fixed rates from various sources and propagate it among independently developed tasks using a publish-subscribe scheme. A common issue is the nature of dynamic fluctuations in task execution time for processing this data, exacerbated by resource contention, dynamic environments, and scheduling strategies. This dynamism, combined with the lack of runtime coordination of task execution at the operating system level, leads to misaligned task execution rates, which is undetected by both middleware and operating systems. As a result, this *misalignment* (i.e., uncoordinated task execution) causes message drops, resource wastage, and degraded real-time performance. To address these challenges, we propose an Adaptive Task Execution rate Regulation (ATER) framework specifically designed for ROS 2-based systems. ATER consists of two key components: a *runtime observer* and a *task regulator*. It seamlessly integrates with the ROS 2 system without affecting its execution or requiring any modifications to its source code. By adapting the sensor data sampling rates at runtime, our framework effectively enhances the real-time performance of ROS 2-based systems through reduced message drops, efficient utilization of computational resources, and improved end-to-end latency.**

I. Introduction

With the increasing complexity of autonomous systems, such as autonomous vehicles and robots, software middleware like ROS/ROS 2 (the successor to ROS) [1], [2] and Cyber RT [3] running on top of the underlying operating systems, has become essential in managing communication, task executions, and integrating functionality libraries. Among these, ROS 2 stands out as widely recognized by both industry and academia, which is the focus of this paper. ROS 2's modular design philosophy allows for the integration of various independently developed components that propagate sensor data through a publish-subscribe (pub-sub) communication scheme. For instance, Autoware.Universe [4], [5], a popular open-source autonomous driving system developed on ROS 2, periodically captures sensor data from diverse sources, including camera and LiDAR, at predetermined rates. It consists of multiple components, such as sensing, localization, perception, planning, and control, each usually comprising several interrelated tasks for processing the sampled sensor data. This modular design philosophy reduces system development complexity and enhances the scalability and reusability of autonomous systems [6].

However, these components usually lack coordination at the operating system level, leading to uncoordinated task execution rates at runtime, i.e., the misalignment issue of task execution. In a typical ROS 2-based system, a series of tasks form a *task chain*, starting with a *timer task* that triggers periodic task instances. Subsequent tasks in the chain are activated by incoming messages. They process the output of preceding tasks before passing results along the chain. To handle the execution of these tasks, multiple ROS 2 *executors*, serving as the task execution managers, are responsible for scheduling threads to execute different parts of the task chain. The lack of coordination between these executors can lead to ***task execution rate misalignment***, where one executor publishes messages faster than the next can process them. This can cause message backlog and even message buffers to overflow, as faster executors produce messages at a rate that slower executors cannot handle, leading to discarded messages and wasted computational effort for these messages. This misalignment issue, characterized by discrepancies between message production and consumption rates among tasks, can be exacerbated by variability in task execution time. Such variability can be caused by the fluctuations in task commencement times and dynamically changing execution times due to resource contention [7]–[13], environmental complexity [14]–[16], and task scheduling strategies [9], [17], [18]. Neither the software middleware, such as ROS 2, nor the operating system is aware of this misalignment. This can lead to wasted resources and degraded real-time performance [19] in automotive systems, which rely on real-time data processing to make timely and predictable decisions. Existing works have largely overlooked the coordination of interconnected components in ROS 2-based systems and few has concerned such task execution rate misalignment problem.

To bridge this research gap, we introduce an **Adaptive Task Execution rate Regulation (ATER)** framework tailored for **ROS 2**-based systems. The ATER framework enables dynamical regulation of task execution rates during runtime, and has no interface with the execution of ROS 2 applications. Importantly, it operates without the need for modifying the source code of ROS 2 applications. ATER consists of two main components: a runtime observer and a task regulator. The runtime observer collects and analyzes the application execution events at runtime. Based on this information, the task regulator determines if the system needs to adjust task execution rates and generates optimized parameters if required. With these generated parameters, the ROS 2-based system can seamlessly adjust its execution rates to align with task's data processing capabilities. The design of ATER prioritizes the generation of system inputs as needed. This emphasis on producing inputs *on demand* ensures the aligned task execution rates in the target system. To validate the effectiveness of the proposed framework, we conducted experiments based on the ROS 2-based Autoware reference system [20] as the case study. The experiment results demonstrate that our method can improve the real-time performance of autonomous systems in terms of number of message drops, conserved computational resources and end-to-end latency.

II. Background

A. ROS 2

As the successor to Robot Operating System (ROS) [1], ROS 2 [2] is a middleware framework designed for developing robot software with higher efficiency and reliability. It provides a platform for managing communication between different components through the Data Distribution Service (DDS) [21], [22], a communication middleware based on a *publish-subscribe* architecture for distributed systems. In this architecture, a *publisher* continuously sends messages to a *topic*, which serves as the message channel for specific data type. Meanwhile, a *subscriber* receives messages from the topic it subscribes to. The two most commonly used types of tasks are *timer tasks* and *subscription tasks*, also known as timer callbacks and subscription callbacks. These *tasks* are functions that implement specific parts of the task's functionality. They encompass the computational workloads associated with ROS 2 applications and are triggered by specific events, such as timer expiration or incoming messages. A *timer* is associated with a timer task, which is triggered by each timer expiration at a regular interval. Each timer task instance publishes a message to the corresponding topic through its publisher at the end of its execution. A *subscriber* is associated with a subscription task, which is invoked upon receiving new messages from the subscribed topic published by the corresponding publisher. Each subscription task instance can receive the newly incoming messages, process them, and then publish a message to another topic through a different publisher. Messages between publishers and subscribers can be temporarily stored in a *message buffer* with a fixed capacity. This buffer ensures that messages are not lost in common cases and can be processed in the order they are received.

In ROS 2, *executors* are responsible for managing the execution of tasks. They handle the scheduling and execution of multiple registered tasks. ROS 2 offers different types of executors, such as the *single-threaded executor* and the *multi-threaded executor*. The single-threaded executor processes tasks sequentially in a single thread, ensuring that tasks are handled one at a time. On the other hand, the multi-threaded executor can manage multiple tasks concurrently using multiple threads, thereby improving the system's efficiency.

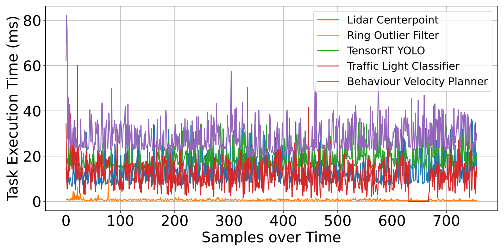


Fig. 1. Variable Task Execution Time in Autoware System.

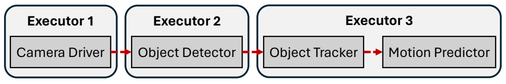


Fig. 2. An Illustrative System for Task Execution Rate Misalignment.

B. Motivation and Challenges

1) Motivation: In ROS 2-based systems, the execution time of software tasks may have large and frequent fluctuations. Figure 1 shows the task execution time for data processing measured from the autonomous driving system, Autoware.Universe [4], [5]. The dynamic variability in task execution time is common, particularly for those processing heavy data volume from sensors like cameras and LiDAR. For example, Autoware.Universe involves multiple tasks aimed at processing image and point-cloud data for 2D/3D object detection, traffic light recognition, and velocity and path planning. This variability can be further worsened by resource competition [7]–[13] and dynamic environments [14]–[16]. Multiple tasks competing for system resources, such as CPU/GPU, memory, network, can lead to increased task execution times and task's commencement time variations. Additionally, the execution times can experience severe fluctuation due to changing workloads caused by the environmental dynamics, since different environmental conditions can lead to differences in the amount of information captured by sensor data. For example, more complex environments contain more information, such as a greater number of objects to be detected in images [15], requiring more processing time and contributing to high variability in task execution time. Furthermore, communication delays and scheduling jitters, both of which are inherently dynamic, can affect the timing of task commencement time. Consequently, the variation in execution time will affect the future invocation times of the tasks, causing variable task execution rates that deviate from the expected system specification.

Varying task execution rates will cause message drops due to message buffer overflow, leading to resource waste and degraded end-to-end latency performance. For example, in Figure 2, four tasks are assigned to three executors. Meanwhile, the detector task competes with the tracker and predictor tasks for computation resources. The camera driver task might sample and publish sensor data at a high rate (i.e., *message publishing/producing rate*), which the detector task can manage. Unfortunately, the tracker and predictor tasks have longer execution times for data processing, resulting in a lower *message subscribing/consuming rate*. Consequently, messages are published to the tracker task faster than it can be processed, leading to a backlog and potential message buffer overflow, i.e., message drops, between Executor 2 and Executor 3. The dropped messages are already generated by the task that utilized significant computational resources. But these messages are not processed by the subsequent tasks in Executor 3 and do not contribute to the overall functionality of the system, leading to wasted computational efforts. And the execution of the detector task for producing these messages competes for resources with the tracker and predictor tasks, prolonging their execution times and increasing the end-to-end latency. To summarize, misalignment of message publishing and subscribing rate, referred to as the *misalignment of task execution rate*, can lead to message drops, wastage of computational resources, and compromised real-time performance. We define this misalignment problem for ROS 2 formally in Section III-B, and introduce our insight to handle this in Section IV-A.

2) Challenges: To address the task execution rate misalignment problem, we claim that two main challenges need to be considered:

Challenge I: How to efficiently and continuously monitor execution information at runtime? The first challenge is obtaining the necessary execution information to identify the occurrence of misalignment in task execution rates at runtime, without disrupting the operation of ROS 2 systems. Existing work [8], [23]–[27] mainly focus on employing two-stage methods. In these methods, the target system is executed alongside tracing tools for a specific duration, resulting in a large volume of tracing data that are stored locally for subsequent offline analysis, rather than being monitored in real-time. Online methods in [28], [29] aim to handle latency monitoring issues and do not apply to the misalignment problem of task execution rates studied in this paper. Therefore, it is essential to develop comprehensive mechanisms for data collection and maintenance mechanisms that facilitate continuous tracing and management of task execution information. This can enable timely detection of the aforementioned misalignment problem.

Challenge II: How to effectively adjust task execution rates? Another critical challenge involves determining the appropriate adjustment of task execution parameters and effectively regulating them accordingly. These parameters include task scheduling parameters (such as task priorities) and resource allocation limits. However, directly adjusting these parameters does not guarantee predictable changes in task execution rates due to the complex and difficult-to-model relationship between them. Moreover, adaptive task priorities can introduce problems like thread starvation and priority inversion, potentially compromising system stability. Dynamic resource allocation methods often lack scalability and are dependent on specific hardware configurations, making them less adaptable across different hardware platforms. The intuition is to regulate the number of sensor data samples. By adjusting the number of message publications from the sensor data sampling point, we can control the triggering of task execution and the propagation of messages along the task chain. This adjustment can regulate task execution rates and eliminate or mitigate message drops that occur during task execution. However, the appropriate method for achieving this, specifically in terms of the sensor data sampling rate, remains unclear.

III. Problem Definition

A. System Model

In the following, we introduce the system model that covers the concepts related to ROS 2. We consider the ROS 2-based system composed of multiple single-threaded executors (*executors* for simplicity) running on a multi-core platform. Let $\mathcal{E} = \{\mathbf{E}_1, \mathbf{E}_2, ..., \mathbf{E}_{|\mathcal{E}|}\}$ denote the set of $|\mathcal{E}|$ executors. The system comprises a set of independent task chains. For clarity, we will focus on a single task chain in detail, although our method applies to all task chains in the system. In Section IV-B, we will discuss further how to specify the task chains for the target ROS 2 application. Let us denote this specific task chain by $\Gamma = \{\tau_{tmr}, \tau_1, ..., \tau_{|\Gamma|}\}$, which includes an ordered sequence of tasks.

- **Timer Task** τ_{tmr}: The first task in a task chain is a timer task that periodically samples sensor data in response to timer triggers. The timer task $\tau_{tmr} = (R_{tmr}, C_{tmr}, \pi^p_{tmr})$ is characterized by its sensor data sampling rate $R_{tmr} \geq 0$ (i.e., the inverse of the timer period) and the execution time $C_{tmr} \geq 0$. This timer task periodically samples sensor data at the beginning of the execution and publishes messages to the corresponding topic π^p_{tmr} upon completion.
- **Subscription Tasks** $\tau_1, ..., \tau_{|\Gamma|}$: After the timer task, a sequence of ordered subscription tasks $\tau_1, ..., \tau_{|\Gamma|}$ follows, where $|\Gamma|$ represents the total number of the subscription tasks in Γ. A subscription task triggered by incoming messages processes data and publishes newly generated data to other subscription tasks via topics. Each subscription task $\tau_i = (\pi^s_i, C_i, \pi^p_i)$ is specified by the subscribed topic π^s_i (from which it receives messages that trigger its execution), the execution time $C_i \geq 0$ and the topic π^p_i for message publication. In addition, each subscription task temporarily stores the messages received from the subscribed topic π^s_i in a message buffer. The message buffer capacity for the subscription task τ_i is constrained by $B_i \geq 1$.

For any tasks $\tau_i, \tau_{i+1} \in \Gamma$, $\pi^s_{i+1} = \pi^p_i$. Specifically, $\pi^s_1 = \pi^p_{tmr}$. The task chain $\Gamma = \{\Gamma_1, \Gamma_2, ...\}$ is divided into an ordered sequence of task segments, where the i-th task segment $\Gamma_i = \{\tau_{i,1}, \tau_{i,2}, ..., \tau_{i,|\Gamma_i|}\}$ contains one or several ordered tasks managed by the same executor $\mathbf{E}_i \in \mathcal{E}$. Particularly, the first task $\tau_{1,1} \in \Gamma_1$ is a timer task. The total execution time of all tasks in executor $\mathbf{E}_i$ is denoted as $W_i = \sum_{\tau_{i,l} \in \Gamma_i, l \in [1, |\Gamma_i|]} C_{i,l}$. Tasks within an executor or across different executors communicate through inter-node communication [19], where a task publishes messages to a specific topic, and those messages are received by the subscription task that subscribes to that topic. We assume there is no message loss during communication.

B. Task Execution Rate Misalignment

In this section, we formally define the task execution rate misalignment problem, which highlights the discrepancy between message publishing and subscribing rates *across different executors* in ROS 2-based systems. For any two executors $\mathbf{E}_{i-1} \in \mathcal{E}$ and $\mathbf{E}_i \in \mathcal{E}$ ($i \in [2, |\mathcal{E}|]$), we define the task pair $\kappa_i = (\tau_{i-1,|\Gamma_{i-1}|}, \tau_{i,1})$ s.t.,

- $\tau_{i-1,|\Gamma_{i-1}|}$ represents the last task in $\mathbf{E}_{i-1}$,
- $\tau_{i,1}$ represents the first task in $\mathbf{E}_i$,
- $\pi^p_{i-1,|\Gamma_{i-1}|}$ denotes the publication topic of $\tau_{i-1,|\Gamma_{i-1}|}$,
- $\pi^s_{i,1}$ denotes the subscribed topic of $\tau_{i,1}$ and
- $\pi^p_{i-1,|\Gamma_{i-1}|} = \pi^s_{i,1}$.

Considering the task chain Γ across the executors in $\mathcal{E}$, the corresponding $|\mathcal{E}| - 1$ task pairs can be represented as $\mathcal{K} = \{\kappa_1, \kappa_2, ..., \kappa_{|\mathcal{E}|-1}\}$. To define task execution rate misalignment, we examine any task pair $\kappa_i = (\tau_{i-1,|\Gamma_{i-1}|}, \tau_{i,1})$ from $\mathcal{K}$. Task execution rate misalignment occurs when the number of messages published by task $\tau_{i-1,|\Gamma_{i-1}|} \in \mathbf{E}_{i-1}$ within a specific time interval exceeds the number of messages subscribed to and processed by task $\tau_{i,1} \in \mathbf{E}_i$ within the same time interval.

Briefly speaking, task execution rate misalignment occurs when a task in one executor publishes messages at a faster rate (*publishing rate*) than a subsequent task in another executor can consume them (*subscription rate*). This misalignment problem, as discussed in Section II-B1, can result in message drops, resource wastage, and degraded end-to-end latency performance. The tolerance for task execution rate misalignment varies across different systems. Systems with higher tolerance for this problem may only need to address this issue when the misalignment reaches an extremely severe level. However, systems like autonomous driving systems, which have limited resources and strict constraints on end-to-end latency performance, must address this problem promptly.

It is worth mentioning that when discussing the task execution rate misalignment problem, we should focus on tasks *across different executors*, rather than within the same executor. This choice is based on two facts. First, executors enable effective task isolation between different functional modules, minimizing interference between tasks, and aligning with the modular philosophy of ROS 2, which is commonly used in ROS 2-based systems. Second, tasks managed by the same executor adhere to the ROS 2 scheduling strategy [30], [31], which guarantees timely consumption of messages produced by one task by another [19]. This synchronization ensures that tasks within the same executor have an appropriate, and potentially exact, match in execution rate. However, this level of synchronization is not guaranteed for tasks across different executors, since they are not explicitly managed in a harmonious manner under the scheduling strategy.

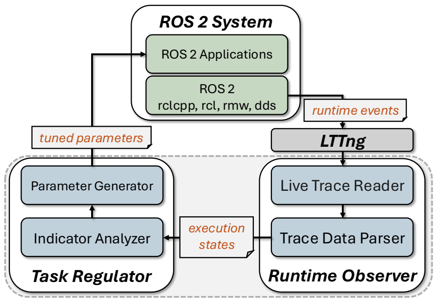


Fig. 3. ATER Framework.

In the following section, we will describe our proposed Adaptive Task Execution Rate framework designed to tackle the aforementioned misalignment problem.

IV. ATER Framework

A. Overview

Figure 3 shows the architecture of a ROS 2 system with our proposed ATER framework. The target ROS 2 system refers to user's ROS 2 application implemented on top of ROS 2 core layers (i.e., rclcpp, rcl, rmw, and DDS). ATER runs alongside with the target system, actively monitoring its runtime execution events, collecting these events, and generating execution parameters that are then utilized by the target system. It is applicable to all ROS 2 applications that involve task chains. Importantly, ATER achieves this without requiring any modifications to the source code of ROS 2 applications. And, it remains agnostic to the hardware platform hosting the target ROS 2 system. ATER can be deployed on a separate system alongside the target ROS 2 system, which ensures that it operates without impeding the execution of the ROS 2 application.

ATER mainly consists of two components: a *runtime observer* and a *task regulator*. The runtime observer employs a live trace reader and a trace data parser to address **Challenge I**. The live trace reader captures runtime events collected from ROS 2 core layers, through the live recording mode [32] of the next-generation Linux Trace Toolkit (LTTng) [33], a low-overhead tracing framework. The trace data parser processes these events to extract the system's execution states and continuously maintains the historical information of these states. Then, the execution state information is sent to the task regulator, which addresses **Challenge II**. The task regulator analyzes the *task execution rate indicators* based on the received execution states. It then generates tuned parameter, i.e., the sensor data sampling rate R_{tmr}, and publishes a message carrying such parameter to the ROS 2 application. Finally, the ROS 2 application can reset the timer task's sensor data sampling rate by first canceling the timer with *rclcpp::TimerBase::cancel()* and then re-creating a new timer with the sampling rate R_{tmr}. This process is completed at runtime with negligible impact on the application's execution. By tackling Challenge I and II, the ATER framework establishes a closed feedback loop to continuously monitor the target system, analyze the tracing data, and regulate execution parameters accordingly to maintain optimal task execution rates. This adaptive approach ensures that the system can respond to dynamically changing task execution rates, minimize misalignment, save resources and enhance real-time performance.

It is important to explain why we can adjust task execution rates by directly regulating the sensor data sampling rate R_{tmr}. The intuition is that if there are messages that will inevitably be dropped at some point in the task chain, it would be better to drop them at the beginning of the task chain. This is also motivated by two reasons. First, the sensor data sampled by the timer task is encapsulated into messages that propagate through the entire task chain. Second, all subscription tasks in the task chain are triggered by these incoming messages for execution. So by adjusting the sensor data sampling rate, we can manage the number of message publications from source and the propagation of messages along the task chain. This allows us to control the triggering of task execution, i.e., the task execution rate, along the task chain. Therefore, we can analyze the message processing capabilities of tasks and identify any bottlenecks among task pairs in the task chain. Based on this information, we can adjust the sensor data sampling rate to sample sensor data *on demand*, adapting to the bottlenecks. In summary, we can dynamically adjust the sensor data sampling rate of the timer task, either increasing or decreasing it as needed, in response to changes in the message processing capabilities of subsequent tasks. This ensures a balanced task execution rate across the executors throughout the entire task chain. Recall the example in Figure 2, when the number of dropped messages between Executor 2 and 3 increases, the detector task publishes a large number of messages that surpass the processing capabilities of tasks in Executor 3, exacerbating the misalignment issue. By reducing the sensor data sampling rate of the camera driver task, the propagation of messages to the detector task can be decreased. Consequently, the detector task can process and publish a reduced number of messages, thus reducing the waste of computational resources and the message drops between Executor 2 and 3.

B. Runtime Observer

The runtime observer collects and parses runtime execution events related to the ROS 2 application. The runtime events are collected from ROS 2 core layers using LTTng without instrumenting the source code of the application and interrupting the target system's execution. This is achieved based on the chain-aware ROS 2 evaluation tool (CARET) [25], which traces execution events using LTTng. CARET adds trace point instrumentation to the ROS 2 core layers and the DDS processes, allowing for the monitoring of the required execution events. However, CARET only supports offline analysis, requiring that tracing data be stored locally after recording for later usage. To obtain the execution events at runtime, we expand CARET to support online tracing by integrating LTTng live recording mode [32] into it. In this live mode, LTTng sends trace events to a listening relay daemon for live reading. By this, ATER can obtain task's execution information for identification of the task execution rate misalignment at runtime, addressing **Challenge I**.

The live trace reader is designed based on the LTTng live reader, i.e., Babeltrace2 [34]. It connects to the same listening relay daemon of LTTng through LTTng live protocol [35], which also allows remote connections and the propagation of events over the network. It can receive runtime execution events as long as the LTTng recording is active. Intuitively, we should compare the number of messages published and subscribed by two adjacent executors within the same time interval to identify the occurrence of task execution rate misalignment. To achieve this, the live trace reader collects the runtime trace events for every observation period with a time interval of Δt. The trace events include the timestamps of message publishing, subscription task's commencement timestamps, and tasks' execution times.

The trace data parser receives the runtime events from the live trace reader to extract and maintain the system's runtime execution states for further analysis. We first run ATER at the start-up stage to obtain the initialization information about the target ROS 2 application. This information is used to construct the ROS 2 architecture, which includes the static details such as timer and publishing information, subscription task information, executor information, task chain information. With this information, users can specify the task chain to be managed, which may encompass either the entire task chain or a segment of a task chain. However, the first task in this task chain *must* be a timer task. The method of identifying such a task chain depends on the user's discretion and the target ROS 2 application. For example, it could be a critical task chain within the application, or users could conduct preliminary tests to find a potential task chain with bottlenecks.

Once the task chain to be managed is identified, ATER can continuously monitor the runtime execution events of the specified ROS 2 application and process this data at regular periods. For each observation period, the trace data parser receives the collected runtime events. By integrating this with the application's architecture information, it can analyze the trace data to extract the runtime execution states of the application. Here, we first define specific execution state information used by the task regulator, while additional state information are provided as needed in the following sections. For each task pair $\kappa_i \in \mathcal{K}$, we define the following execution state information:

- $N^p_{i-1,j}$: Number of messages published by task $\tau_{i-1,|\Gamma_{i-1}|}$ within the j-th observation period.
- $N^s_{i,j}$: Number of messages subscribed by task $\tau_{i,1}$ within the j-th observation period.
- $N^b_{i,j}$: Number of messages in the message buffer of $\tau_{i,1}$ at the end of the j-th observation period.
- $N^d_{i,j}$: Number of messages dropped from the message buffer of $\tau_{i,1}$ within the j-th observation period.
- $R^s_{i,j}$: Execution rate (subscription rate) of $\tau_{i,1}$ within the j-th observation period, equal to $N^s_{i,j}/\Delta t$.
- $R^p_{i-1,j}$: Execution rate (publishing rate) of task $\tau_{i-1,|\Gamma_{i-1}|}$ within the j-th observation period, equal to $N^p_{i-1,j}/\Delta t$.

- $W_{i,j}^{avg}$: Average of execution times of tasks in $\mathbf{E}_i$ within the j-th observation period.

C. Task Regulator

In this section, we present the design of the task regulator in ATER, which comprises two main components: an indicator analyzer and a parameter generator, addressing **Challenge II** by answering the following two critical questions:

- When to regulate task execution rates?
- How to generate sensor data sampling rates for regulation?

1) Indicator Analyzer: To answer the first question, we develop indicators to determine when to regulate the task execution rates by distinguishing whether the sensor data sampling rate needs to be decreased or increased.

Decreasing Sensor Data Sampling Rate. Task execution rate misalignment occurs when a task in an executor publishes messages at a rate faster than the subsequent task in another executor can consume them. Within the task chain, there is a bottleneck that corresponds to specific task pair $\kappa_i = (\tau_{i-1,|\Gamma_{i-1}|}, \tau_{i,1}) \in \mathcal{K}$, where $\tau_{i-1,|\Gamma_{i-1}|} \in \mathbf{E}_{i-1}$ and $\tau_{i,1} \in \mathbf{E}_i$ $(i \in [1, |\mathcal{K}|])$. Such task pair satisfies the condition that the message publishing rate $R_{i-1,j}^p$ of $\tau_{i-1,|\Gamma_{i-1}|}$ is larger than the message subscription rate $R_{i,j}^s$ of $\tau_{i,1}$ during the j-th observation period. This is limited by the message processing capabilities of the tasks in $\mathbf{E}_i$. These bottlenecks can result in a significant increase in message drops between executors, making it crucial to measure and understand the severity of these drops for adjusting task execution rates accordingly.

As discussed previously, we adjust task execution rates, i.e., decreasing the message publishing rate R_{i-1}^p, by decreasing sensor data sampling rate R_{tmr}. To determine the necessity for decreasing sensor data sampling rate, we develop the *Message Drop Indicator (MDI)* to capture the occurrence and severity of message drops. Formally, for any task pair $\kappa_i \in \mathcal{K}$ $(i \in [1, |\mathcal{K}|])$, considering the j-th observation period, we define the *Message Drop Indicator (MDI)* as follows:

$$MDI_{i,j} = \begin{cases} 1 & \text{if} \quad \frac{N_{i,j}^d}{N_{i,j}^p + N_{i,j-1}^b} \geq \theta_i \\ 0 & \text{otherwise} \end{cases}$$

where

- $N_{i,j}^d = 0$, **if** $\sum_{h=0}^{j}(N_{i,h}^p - N_{i,h}^s) - \sum_{h=0}^{j-1} N_{i,h}^d \leq B_i$, **else**
- $N_{i,j}^d = \sum_{h=0}^{j}(N_{i,h}^p - N_{i,h}^s) - \sum_{h=0}^{j-1} N_{i,h}^d - B_i$

The denominator represents the total number of messages that can be subscribed to within j-th observation period, and the numerator denotes the number of messages dropped within j-th observation period. The threshold $\theta_i \in [0, 1]$ $(i \in [1, |\mathcal{K}|])$ represents the system's tolerance for message drops, which can be set based on the empirical data and specific requirements of the system. A $MDI_{i,j}$ value larger than θ_i indicates a high percentage of message drops, i.e., significant message drops, while a lower value suggests insignificant or no message drops. Figure 4 demonstrates the potential number of messages drops in relation to the total number of messages buffered and published. The red line represents the maximum number of messages can be dropped within j-th observation period. The area below the red line represents all possible numbers of message drops. The blue line represents the message drop indicator, where the slope is the threshold θ_i ($\theta_i = 0.5$ in this case). Thus, the number of message drops can fall into the blue region (e.g., point P_1) or the green region (e.g., point P_2). According to the definition of MDI, values in the blue region indicate a significant number of message drops, while values in the green region indicate an insignificant number of message drops.

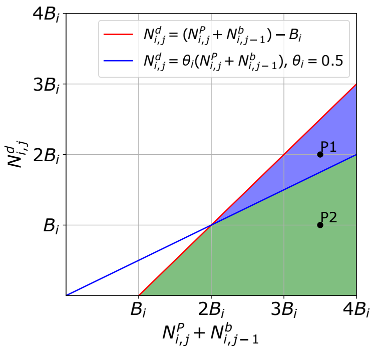


Fig. 4. Illustration of Message Drop Indicator.

The Message Drop Indicator quantifies the percentage of dropped messages compared to the total number of messages that can be consumed within a single observation period, which includes both newly published messages and those buffered from the previous observation period. MDI serves as a metric to evaluate the extent of message drops between two executors within each observation period. By using MDI, we can identify bottlenecks in the task chain, i.e., the task pairs where the task execution rate misalignment occurs. This allows ATER to determine when to decrease the sensor data sampling rate to effectively mitigate message drops.

Increasing Sensor Data Sampling Rate. Previously, we discussed the case of decreasing sensor data sampling rate when there is some bottleneck task pair $\kappa_i = (\tau_{i-1,|\Gamma_{i-1}|}, \tau_{i,1})$ with $R_{i,j}^s < R_{i-1,j}^p$ within j-th observation period. The message subscription rate of $\tau_{i,1}$ is limited by the message processing capabilities of the tasks in $\mathbf{E}_i$. If we represent the message processing capabilities using the average of task execution times $W_{i,j}^{avg}$, we can observe that $1/W_{i,j}^{avg} = R_{i,j}^s < R_{i-1,j}^p$. In this case, we can decrease the sensor data sampling rate to lower the message publication rate R_{i-1}^p.

However, in some scenarios, it becomes necessary to increase the sensor data sampling rate. This can occur when previously identified bottlenecks are no longer constraining due to the improvement of the message processing capabilities of corresponding tasks. To adhere to system limitations, ATER will *not* increase the sensor data sampling rate beyond the pre-defined rate. In this case, tasks in $\mathbf{E}_i$ tend to have shorter execution times ($W_{i,j}^{avg}$). It indicates that the message publishing rate $R_{i-1,j}^p$ of $\tau_{i-1,|\Gamma_{i-1}|}$ can not keep up with the message processing capabilities of tasks in $\mathbf{E}_i$, i.e., $1/W_{i,j}^{avg} > R_{i-1,j}^p$, within j-th observation period. The message subscription rate of $\tau_{i,1}$ is limited by the message publishing rate of $\tau_{i-1,|\Gamma_{i-1}|}$, and $R^s_{i,j} = R^p_{i-1,j}$ must hold within j-th observation period. To fully leverage the enhanced message processing capabilities, it is necessary to increase the task execution rate by increasing the sensor data sampling rate, to improve the information capture and throughput of the task chain.

Unfortunately, we can not identify the above scenarios by measuring the extent of message drops as in MDI, since there should be no message drops when $R^s_{i,j} = R^p_{i-1,j}$ holds. To address this, we focus on the execution state $W^{avg}_{i,j}$ for $\mathbf{E}_i$ and $R^s_{i,j}$ for the subscription task $\tau_{i,1}$ within j-th observation period. First, we use the normal distribution with mean W^{mean}_i and deviation W^{dev}_i to model the variable task execution times. To achieve this, the runtime observer extracts the average of execution times of tasks in $\mathbf{E}_i$ within any observation period. The indicator analyzer receives such execution time states for each observation period and maintains the mean W^{mean}_i and deviation W^{dev}_i of the execution times over all observation periods. To determine if the message processing capabilities of tasks in $\mathbf{E}_i$ have improved, we can compare $W^{avg}_{i,j}$ with the execution time distribution characterized by W^{mean}_i and W^{dev}_i. A lower $W^{avg}_{i,j}$ in this distribution indicates potential improvement in the message processing capability of tasks in $\mathbf{E}_i$. Moreover, we can further determine if the message publishing rate should be increased by comparing $W^{avg}_{i,j}$ and $R^s_{i,j}$. Based on these insights, for any task pair $\kappa_i \in \mathcal{K}$ ($i \in [1, |\mathcal{K}|]$) within j-th observation period, we define the *Increasing Sampling Rate Indicator (ISRI)* as follows:

$$ISRI_{i,j} = 1 \quad \text{if} \quad W^{avg}_{i,j} < W^{mean}_i - \alpha_i W^{dev}_i \wedge W^{avg}_{i,j} < \frac{\lambda_i}{R^s_{i,j}}$$

otherwise, $ISRI_{i,j} = 0$. $\alpha_i \in \mathbb{R}^+$ determines the threshold for the deviation from the mean of task execution times. $\lambda_i \in [0, 1]$ ($i \in [1, |\mathcal{K}|]$) is the factor to control the sensitivity of ISRI to the relation between $W^{avg}_{i,j}$ and $R^s_{i,j}$. By adjusting them, users can customize the behavior of ISRI according to their specific statistical information. Please note that ISRI will be enabled only if there are no message drops for any task pair in the task chain, i.e., $\forall i \in [1, |\mathcal{K}|]$, $N^d_{i,j} = 0$.

2) Parameter Generator: This section answers the second question: how to generate the sensor data sampling rate for task execution rate regulation. Based on Message Drop Indicator and Increasing Sampling Rate Indicator, the parameter generator calculates the tuned sensor data sampling rate by considering two cases: decreasing or increasing sensor data sampling rate. For each task pair $\kappa_i \in \mathcal{K}$, we define the following execution state information that can be extracted using the trace data parser:

- $R^{s,min}_i$: Minimum execution rate (subscription rate) of task $\tau_{i,1}$ over all observation periods.
- $R^{s,avg}_i$: Average execution rate (subscription rate) of task $\tau_{i,1}$ over all observation periods.
- $R^{s,max}_i$: Maximum execution rate (subscription rate) of task $\tau_{i,1}$ over all observation periods.

Decreased Sensor Data Sampling Rate. As described in Section IV-A, ATER regulates the task execution rates by adjusting the sensor data sampling rate in order to control the number of published messages as required. The parameter generator determines the decreased sensor data sampling rate whenever there are task pairs with MDI values equal to 1, indicating the occurrence of the task execution rate misalignment. Among these task pairs, we identify the bottleneck in the task chain, which limits the propagation of messages through the chain. This bottleneck task pair corresponds to the executor with the lowest message processing capability, i.e., it can potentially process the fewest number of messages within the upcoming observation periods. Then, the decreased sensor data sampling rate can be calculated based on its execution state, i.e., the message subscription rate. Formally, $\forall \kappa_i \in \mathcal{K}, i \in [1, |\mathcal{K}|]$ and $MDI_{i,j} = 1$, just after the j-th observation period, we derive the decreased sensor data sampling rate $\overrightarrow{R_{tmr}}$ as follows:

$$\overrightarrow{R_{tmr}} = \frac{\min_{i \in \Omega}\{\overline{R^s_i}\varphi\Delta t - \sum_{h=1}^{i} N^b_{h,j}\}}{\varphi\Delta t}$$

where

$$\overline{R^s_i} \in \{R^{s,min}_i, R^{s,avg}_i, R^{s,max}_i\}$$

$$\Omega = \{i \mid MDI_{i,j} = 1,\ \forall i \in [1, |\mathcal{K}|]\}$$

$\varphi \in \mathbb{Z}^+$ is a tunable factor to set the desired number of observation periods for decreasing the sensor data sampling rate. $\overline{R^s_i}$ is the potential execution rate of $\tau_{i,1}$ in the φ desired observation periods, which can be set empirically. The equation considers all task pairs that satisfy $MDI_{i,j} = 1$ in the task chain. For each task pair κ_i, the term $\overline{R^s_i}\varphi\Delta t$ represents the potential number of messages that can be processed by task $\tau_{i,1}$ in the φ upcoming observation periods. The term $\sum_{h=1}^{i} N^b_{h,j}$ represents the total number of messages currently stored in the buffers corresponding to task pairs $\{\kappa_1, \kappa_2, ..., \kappa_i\}$. The numerator of the equation identifies the minimum number of messages that can be processed by the bottleneck task pair.

Increased Sensor Data Sampling Rate. Similarly, the parameter generator calculates the increased sensor data sampling rate when there are task pairs with ISRI values equal to 1. From these task pairs, we find the one that corresponds to the executor with the lowest message processing capability within the upcoming observation periods potentially. The increased sensor data sampling rate is then calculated based on the message subscription rate of this executor. Formally, $\forall \kappa_i \in \mathcal{K}$, $i \in [1, |\mathcal{K}|]$ and $ISRI_{i,j} = 1$, just after the j-th observation period, the increased sensor data sampling rate $\overleftarrow{R_{tmr}}$ is derived as follows:

$$\overleftarrow{R_{tmr}} = \frac{\min_{i \in \Phi}\{\overline{R^s_i}\sigma\Delta t - \sum_{h=1}^{i} N^b_{h,j}\}}{\sigma\Delta t}$$

where

$$\overline{R^s_i} \in \{R^{s,min}_i, R^{s,avg}_i, R^{s,max}_i\}$$

$$\Phi = \{i \mid ISRI_{i,j} = 1,\ \forall i \in [1, |\mathcal{K}|]\}$$

The parameter $\sigma \in \mathbb{Z}^+$ is a tunable factor to specify the desired number of observation periods to increase the sensor data sampling rate. The parameter $\overline{R_i^s}$ represents the potential execution rate of $\tau_{i,1}$ in σ desired observation periods.

V. Experiments

We implemented the ATER framework in Python 3. We conducted all experiments on a desktop computer with an Intel(R) Core(TM) i7-10700 CPU running at 2.90 GHz. The computer was installed with ROS 2, specifically the Humble Hawksbill version running on Ubuntu 22.04.3 LTS. The source code[1][2] is openly available for public access.

We design the case study benchmark based on the Autoware autonomous driving reference system [20]. In this system, all tasks execute synthetic workloads running on the real ROS 2 system. The case study system consists of five single-threaded executors, each with either a timer task or a subscription task, as illustrated in Figure 5, and each subscription task has a message buffer of size $B_i = 1$ (same as Autoware [4], [5]). It is important to explain why the buffer size is set to 1. According to the findings in [19], message drops between two executors cannot be prevented anyway if $1/W_i^{avg} < R_{i-1}^p$. Therefore, having a larger buffer size here does not provide benefits, but can imply large message backlogs and growing end-to-end latency. For all experiments, the partitioned scheduling paradigm is used, where each executor is mapped to a single core without interference from the other programs and cannot migrate across cores. To demonstrate the effectiveness of ATER, we evaluate the case study system under four different settings, detailed in Table I.

In all settings, the timer task has a sampling rate of 12.5Hz and an execution time of $C_{tmr} = 5$ms. The subscription tasks τ_1, τ_2 and τ_3 have dynamically variable execution times that take on the normal distributions. The execution time of τ_1, τ_2 and τ_3 follows normal distributions $\mathcal{N}(60, 5^2)$ms, $\mathcal{N}(30, 5^2)$ms, and $\mathcal{N}(50, 5^2)$ms, respectively. In settings 1 and 4, the execution time of subscription task τ_2 follows normal distributions of $\mathcal{N}(90, 20^2)$ms and $\mathcal{N}(50, 5^2)$ms, respectively. In settings 2 and 3, the execution time of τ_2 alternates between two normal distributions. For the first 30 seconds, it follows $\mathcal{N}(50, 5^2)$ms, and from 30 to 60 seconds, it follows $\mathcal{N}(120, 10^2)$ms, and so on. The choice of using normal distribution execution times and alternating execution in settings 2 and 3 was made to simulate realistic scenarios and capture the variability in task execution times that can occur in practical systems as discussed in Section II-B1. Specifically, settings 1 and 2 are designed to introduce thread resource competition between τ_1 and τ_3, while settings 3 and 4 do not involve any resource competition. Empirically, we set the system parameters as follows: $\Delta t = 8$s, $\theta_i = 0.05$, $\lambda_i = 0.8$, $\alpha_i = \varphi = \sigma = 1$, and $\overline{R_i^s} = R_i^{s,avg}$.

During evaluation, we first assess the case study system under four settings without enabling ATER, establishing **Baseline**. Then, we conduct another round of evaluations under the same settings, but with ATER actively regulating the task execution rates, experiment results for which are denoted as **ATER**. The case study system is evaluated in terms of the four metrics: Number of Message Drops, End-to-End Latency, CPU Time Savings, and Timer Reset Overhead.

A. Number of Message Drops

In this section, we analyze the number of message drops in settings 1, 2, and 3. As shown in Figure 6, we can observe the number of message drops over 30 observation periods under Baseline and with ATER enabled. In all settings, ATER significantly reduces the number of message drops compared to Baseline. In setting 1, the number of message drops remains consistently low, often equal to 1 or 0. In setting 2 and 3, the execution time of τ_2 fluctuates, causing periodic misalignment in the task execution rate due to the alternation of the execution time of τ_2 between two normal distributions. In Figures 6-(b) and (c), τ_2 follows an execution time distribution of $\mathcal{N}(120, 10^2)$ms, the misalignment issue exacerbates, leading to a significant increase in message drops under Baseline. However, ATER effectively mitigates the message drops for both settings. The reason is that ATER can regulate task execution rate in a timely manner: it decreases the sensor data sampling rate (indicated by the vertical dashed red lines in Figures 6-(b) and (c) when τ_2 experiences high execution times and increases the sampling rate (indicated by the vertical dashed green lines) when τ_2 has lower execution times.

B. End-to-End Latency

ATER can reduce message drops with adaptive task execution rate regulation. This helps reduce the execution of tasks that compete for resources. To showcase this benefit, we evaluate the case study in terms of end-to-end latency. We measure end-to-end latency over 30 observation periods for both Baseline and ATER. This metric reflects the time taken for a message to travel from its generation in the timer task τ_{tmr} to its completion at the end of the last task τ_4. We compare the latency performance across the four settings to demonstrate the effectiveness of ATER. Figures 7-(a) and (b) illustrate that ATER reduces the maximum and average end-to-end latency in settings 1 and 2, where thread resource competition exists between τ_1 and τ_3. ATER also maintains a more stable end-to-end latency distribution with lower latency jitter. This improvement is attributed to ATER's ability to adjust the sensor data sampling rate, thereby reducing message drops among tasks τ_1, τ_2, and τ_3. Specifically, fewer message drops between τ_1 and τ_2 alleviate resource contention on τ_3, leading to improved end-to-end latency. According to Figures 7-(c) and (d), the end-to-end latency performance under ATER remains comparable to the baseline. Those latency improvements are not observed in setting 3 and 4. The reason is that the absence of thread resource contention between τ_1 and τ_3 weakens the benefits of ATER framework. Nevertheless, ATER can still reduce the number of message drops from τ_1, conserving CPU time that would have been spent processing those dropped messages and allowing more computational

[1]https://github.com/ruoxianglee/ater-framework
[2]https://github.com/ruoxianglee/caret-with-live-mode

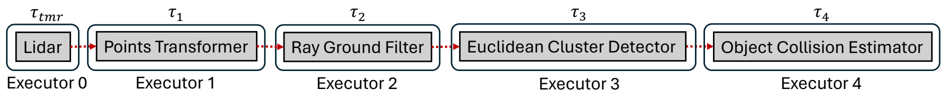
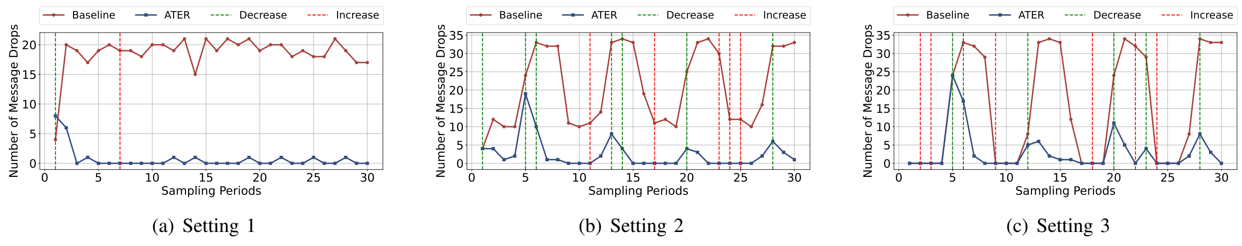


Fig. 5. Case Study.

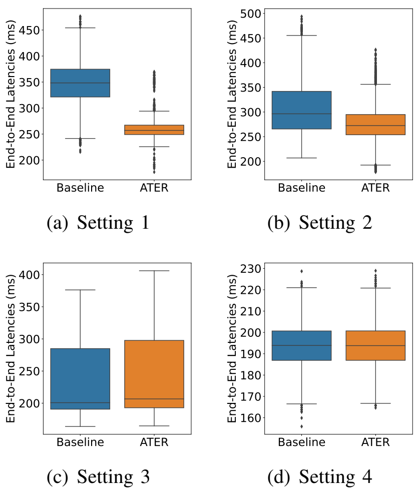
TABLE I
CASE STUDY SETTINGS.

Setting	τ_{tmr}	τ_1	τ_2	τ_3	τ_4	Competition
1	$R_{tmr} = 12.5$Hz,$C_{tmr} = 5$ms	$C_1 \sim \mathcal{N}(60, 5^2)$ms	$C_2 \sim \mathcal{N}(90, 20^2)$ms	$C_3 \sim \mathcal{N}(30, 5^2)$ms	$C_4 \sim \mathcal{N}(50, 5^2)$ms	τ_1, τ_3
2	$R_{tmr} = 12.5$Hz,$C_{tmr} = 5$ms	$C_1 \sim \mathcal{N}(60, 5^2)$ms	$C_2 \sim \mathcal{N}(50, 5^2)$ms or $C_2 \sim \mathcal{N}(120, 10^2)$ms	$C_3 \sim \mathcal{N}(30, 5^2)$ms	$C_4 \sim \mathcal{N}(50, 5^2)$ms	τ_1, τ_3
3	$R_{tmr} = 12.5$Hz,$C_{tmr} = 5$ms	$C_1 \sim \mathcal{N}(60, 5^2)$ms	$C_2 \sim \mathcal{N}(50, 5^2)$ms or $C_2 \sim \mathcal{N}(120, 10^2)$ms	$C_3 \sim \mathcal{N}(30, 5^2)$ms	$C_4 \sim \mathcal{N}(50, 5^2)$ms	No
4	$R_{tmr} = 12.5$Hz,$C_{tmr} = 5$ms	$C_1 \sim \mathcal{N}(60, 5^2)$ms	$C_2 \sim \mathcal{N}(50, 5^2)$ms	$C_3 \sim \mathcal{N}(30, 5^2)$ms	$C_4 \sim \mathcal{N}(50, 5^2)$ms	No



(a) Setting 1 (b) Setting 2 (c) Setting 3

Fig. 6. Comparison of Message Drops.



(a) Setting 1 (b) Setting 2

(c) Setting 3 (d) Setting 4

Fig. 7. Comparison of End-to-End Latencies.

resources to be allocated to τ_3. Furthermore, Figures 7-(c) and (d) show that ATER does not negatively impact end-to-end latency performance in typical scenarios without resource contention or message drops.

C. CPU Time Savings

To further demonstrate the advantages of reduced message drops, we measure the CPU time savings, which can be calculated considering the number of message drops and the execution times of the corresponding tasks. CPU time was measured over 30 observation periods. As illustrated in Figure 8, ATER contributes to a significant reduction in CPU time usage across all settings. Especially in settings 2 and 3, when τ_2 follows an execution time distribution of $\mathcal{N}(120, 10^2)$ms, resulting in a sharp increase in τ_2's, ATER achieves substantial CPU time savings. This is due to the significant increase in message drops under Baseline, which ATER effectively mitigates. By reducing the number of message drops, ATER effectively saves CPU time that would otherwise be spent processing those dropped messages. This result indicates that ATER achieves more efficient resource allocation and decreases unnecessary computational overhead.

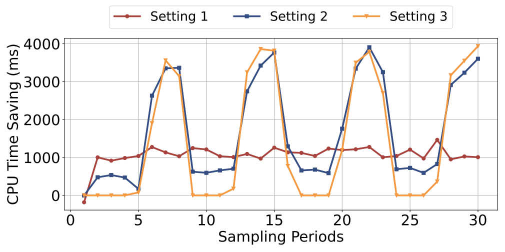


Fig. 8. CPU Time Saving.

D. Timer Reset Overhead

ATER is designed to regulate task execution rates by directly adjusting the sensor data sampling rate. This is achieved by resetting the timer for the timer task in the task chain. To ensure that this operation does not significantly impact the system's execution, we need to evaluate the time overhead associated with resetting the timer of a task chain. We recorded the overhead across all four settings, with more than 40,000 samples, 10,000 for each setting. As shown in Figure 9, the overhead varies between 40 and 100 microseconds, typically remaining below 80 microseconds. Compared to the timer

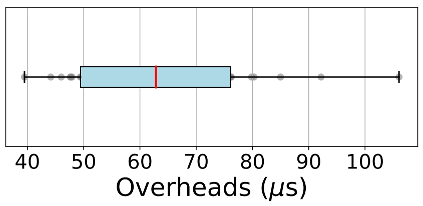


Fig. 9. Timer Rate Reset Overhead.

execution period, which is on the order of milliseconds, this overhead is negligible and acceptable.

VI. Related Work

Resource Contention and Management in ROS 2. [7] evaluated the real-time performance of the ROS 2 callback-group-level executor in terms of latency. The author analysed thread resource contention between executors with different priorities, and found that executors with lower priorities can result in higher latency. [8] showed that the executor and hardware thread contention can induce high end-to-end latency. While timer frequency tuning can help reduce this latency, adjusting the task priorities does not improve the end-to-end latency. [9] proposed deadline-based dynamic scheduling for the single-threaded executor. Several work [10]–[12], [36] focused on accelerator management and resource contention problem in ROS 2. [13] addressed the resource contention in parallel tasks for autonomous driving systems using deep reinforcement learning approach for parallel task scheduling. Other studies [14]–[16] focused on scheduling approaches to address dynamic uncertainty in systems due to environmental complexity. They either just evaluated ROS 2 systems without considering the task execution rate misalignment problem we focus on, or attempted to eliminate the resource contention through scheduling and system design optimization.

Performance Evaluation and Monitoring in ROS 2. Research such as [8], [23]–[27], [37], [38] aimed to evaluate ROS 2 RMW's implementation or performance in terms of task latency, communication latency, and end-to-end latency. [25] introduced the chain-aware ROS 2 evaluation tool based on LTTng [33] for performance analysis. [26] proposed a follow-up method for ROS 2 scheduling analysis. [8] showed the influence of different software and hardware configurations on performance in terms of communication and computation latency. [39] presented an offline method to trace ROS 2 applications for timing models based on extended Berkeley Packet Filter (eBPF) [40]. [41] applied it as the offline timing model extraction method to manage end-to-end jitters in ROS 2 computation chains. These focus on offline performance analysis for ROS 2, without supporting online monitoring. [42] proposed an automatic thread scheduling management framework to control the latency of critical cause-effect chains based on runtime latency management. [28], [29] combined online monitoring with the distribution of deadlines for safety-critical event chains with real-time constraints. These methods primarily focus on latency monitoring and do not consider the task execution rate misalignment problem studied in this paper.

Formal Analysis for Scheduling in ROS 2. [17], [18], [30], [31], [43], [44] focus on the formal analysis of the task scheduling problem in ROS 2 default executor. [9], [18], [45]–[47] aimed to address the limitations of ROS 2's default scheduling strategy by enhancing or redesigning the executor. These methods try to improve predictability or schedulability of ROS 2 by optimizing and analyzing the scheduling strategy, without considering the task execution rate misalignment problem from the perspective of sensor data sampling rate. [19] proposed preventing sensor data undersampling and message loss by carefully configuring the timer period and subscription buffer size. They proved through formal analysis that message drops across executors cannot be prevented if the messages cannot be processed timely, theoretically supporting the generality of the task execution rate misalignment problem in ROS 2 but not providing a solution.

Feedback-based Real-Time Scheduling. Feedback-based scheduling has been extensively studied in previous work, such as [48]–[54]. However, they mainly focus on systems with *independent* tasks or are solely dedicated to optimizing control tasks, as seen in [51], [52]. Feedback-driven scheduling techniques could potentially be adapted to ROS 2, but their applicability remains unexplored [42]. More importantly, ATER constructs an explicit workload model derived directly from ROS 2 application runtime behavior, enabling dynamic adjustment of task execution rates through fine-grained monitoring of *inter-task dependencies*, and *inter-task behaviors*. This accounts for the structural relationships and communication patterns inherent to ROS 2 philosophy, rather than relying solely on coarse-grained system metrics (e.g., deadline miss rates). To the authors' best knowledge, no existing methods apply the adaptive task execution rate regulation to ROS 2.

VII. Conclusion

This paper identifies the task execution rate misalignment problem and proposes the Adaptive Task Execution Rate Regulation (ATER) framework to address this issue in ROS 2-based systems. The ATER framework operates at runtime and consists of two main components: a runtime observer, which collects and analyzes task's runtime execution events, and a task regulator, which generates optimized execution parameters to regulate task execution rates. Conducted experiments under different settings on the case study system demonstrate that ATER effectively reduces message drops, conserves computational resources, and reduces end-to-end latency. This approach fills the gaps in current research and provides a new perspective to enhance the real-time performance of ROS 2 based autonomous systems. While our approach enhances system efficiency, further investigation into application-level trade-offs is warranted, such as potential temporal misalignment in multi-sensor fusion (e.g., LIDAR-camera synchronization) when reducing sampling rates, which could influence perception robustness. Additionally, refining the techniques for determining the sensor data sampling rates could be a focus for further enhancement.

Acknowledgment

This work was supported by the Research Grants Council of Hong Kong under Grant GRF 11208522.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[3] Baidu. (2020) Apollo cyber rt. [Online]. Available: https://cyber-rt.readthedocs.io/en/latest/

[4] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[5] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.

[6] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu, "Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1067–1081.

[7] Y. Yang and T. Azumi, "Exploring real-time executor on ros 2," in *2020 IEEE international conference on embedded software and systems (ICESS)*. IEEE, 2020, pp. 1–8.

[8] T. Betz, M. Schmeller, H. Teper, and J. Betz, "How fast is my software? latency evaluation for a ros 2 autonomous driving software," in *2023 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2023, pp. 1–6.

[9] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo, "Response time analysis for dynamic priority scheduling in ROS2," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 301–306.

[10] Y. Suzuki, T. Azumi, S. Kato, and N. Nishio, "Real-Time ROS extension on transparent CPU/GPU coordination mechanism," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2018, pp. 184–192.

[11] R. Li, T. Hu, X. Jiang, L. Li, W. Xing, Q. Deng, and N. Guan, "Rosgm: A real-time gpu management framework with plug-in policies for ros 2," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 93–105.

[12] D. Enright, Y. Xiang, H. Choi, and H. Kim, "Paam: A framework for coordinated and priority-driven accelerator management in ros 2," *arXiv preprint arXiv:2404.06452*, 2024.

[13] Q. Qi, L. Zhang, J. Wang, H. Sun, Z. Zhuang, J. Liao, and F. R. Yu, "Scalable parallel task scheduling for autonomous driving using multi-task deep reinforcement learning," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 11, pp. 13 861–13 874, 2020.

[14] J. Ma, L. Li, and C. Xu, "Autors: Environment-dependent real-time scheduling for end-to-end autonomous driving," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 12, pp. 3238–3252, 2023.

[15] L. Liu, Z. Dong, Y. Wang, and W. Shi, "Prophet: Realizing a predictable real-time perception pipeline for autonomous vehicles," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 305–317.

[16] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: a dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 453–471.

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

[18] H. Sobhani, H. Choi, and H. Kim, "Timing analysis and priority-driven enhancements of ros 2 multi-threaded executors," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 106–118.

[19] H. Teper, T. Betz, G. Von Der Brüggen, K.-H. Chen, J. Betz, and J.-J. Chen, "Timing-Aware ROS 2 Architecture and System Optimization," in *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Niigata, Japan: IEEE, 2023, pp. 206–215.

[20] E. Flynn. (2021) Autoware reference system. [Online]. Available: https://github.com/ros-realtime/reference-system/blob/main/autoware_reference_system/README.md

[21] 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.

[22] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

[23] C. Bédard, I. Lütkebohle, and M. Dagenais, "ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ros 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.

[24] Z. Li, A. Hasegawa, and T. Azumi, "Autoware_perf: A tracing and performance analysis framework for ros 2 applications," *Journal of Systems Architecture*, vol. 123, p. 102341, 2022.

[25] T. Kuboichi, A. Hasegawa, B. Peng, K. Miura, K. Funaoka, S. Kato, and T. Azumi, "Caret: Chain-aware ros 2 evaluation tool," in *2022 IEEE 20th International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, 2022, pp. 1–8.

[26] B. Peng, A. Hasegawa, and T. Azumi, "Scheduling performance evaluation framework for ros 2 applications," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2022, pp. 2031–2038.

[27] C. Bédard, P.-Y. Lajoie, G. Beltrame, and M. Dagenais, "Message flow analysis with complex causal links for distributed ros 2 systems," *Robotics and Autonomous Systems*, vol. 161, p. 104361, 2023.

[28] J. Peeck, J. Schlatow, and R. Ernst, "Online latency monitoring of time-sensitive event chains in safety-critical applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 539–542.

[29] ——, "Online latency monitoring of time-sensitive event chains in ros2," *TU Braunschweig*, 2021.

[30] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl, 2019, pp. 1–23.

[31] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ROS2 executors," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 231–243.

[32] T. L. Project. (2014) Lttng recording session mode. [Online]. Available: https://lttng.org/docs/v2.13/#doc-tracing-session-mode

[33] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[34] B. 2. (2020) Babeltrace 2. [Online]. Available: https://babeltrace.org/docs/v2.0/man1/babeltrace2.1/

[35] T. L. Tools. (2016) Lttng live reading protocol. [Online]. Available: https://github.com/lttng/lttng-tools/blob/master/doc/live-reading-protocol.txt

[36] H. Li, Q. Yang, S. Ma, R. Zhao, and X. Ji, "Robospike: Fully utilizing the heterogeneous system with subcallback scheduling in ros 2," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[37] ApexAI. (2019) Apexai: performance test. [Online]. Available: https://gitlab.com/ApexAI/performance_test

[38] iRobot. (2019) irobot: Ros2 performance. [Online]. Available: https://github.com/ros2/performance_test

[39] H. Abaza, D. Roy, S. Fan, S. Saidi, and A. Motakis, "Trace-enabled timing model synthesis for ros2-based autonomous applications," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[40] L. Rise, "Learning ebpf–programming the linux kernel for enhanced observability, networking, and security," 2023.

[41] H. Abaza, D. Roy, B. Trach, W. Chang, S. Saidi, A. Motakis, W. Ren, and Y. Liu, "Managing end-to-end timing jitters in ros2 computation

chains," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, 2024, pp. 229–241.

[42] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 264–277.

[43] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 41–53.

[44] Y. Tang, N. Guan, X. Jiang, X. Luo, and W. Yi, "Real-time performance analysis of processing systems on ros 2 executors," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 80–92.

[45] C. Randolph, "Improving the predictability of event chains in ros 2," *Ph. D. dissertation, Master's thesis*, 2021.

[46] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New design of priority-driven chain-aware scheduling for ROS2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 251–263.

[47] S. Liu, X. Jiang, N. Guan, Z. Wang, M. Yu, and W. Yi, "Rtex: an efficient and timing-predictable multi-threaded executor for ros 2," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[48] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, "The case for feedback control real-time scheduling," in *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. IEEE, 1999, pp. 11–20.

[49] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Design and evaluation of a feedback control edf scheduling algorithm," in *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*. IEEE, 1999, pp. 56–67.

[50] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, pp. 85–126, 2002.

[51] A. Cervin and J. Eker, "Feedback scheduling of control tasks," in *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No. 00CH37187)*, vol. 5. IEEE, 2000, pp. 4871–4876.

[52] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, "Feedback–feedforward scheduling of control tasks," *Real-Time Systems*, vol. 23, pp. 25–53, 2002.

[53] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni, "Adaptive reservations in a linux environment," in *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.* IEEE, 2004, pp. 238–245.

[54] D. Fontanelli, L. Palopoli, and L. Greco, "Deterministic and stochastic qos provision for real-time control systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 103–112.