Supporting Mixed-Criticality and Mutually Exclusive Callback Groups in Multi-Thread ROS 2

Abdullah Al Arafat*, Kurt Wilson†, Shareef Ahmed‡, Zhishan Guo†
*Florida International University, †North Carolina State University, ‡University of South Florida
aarafat@fiu.edu, {kwilso24, zguo32}@ncsu.edu, shareefahmed@usf.edu

***Abstract*—The second generation of the Robot Operating System, ROS 2, is widely used in the development of robotics and autonomous systems. These systems are inherently mixed critical, as they integrate tasks with different levels of importance—ranging from high-assurance control loops to low-priority monitoring. Although mixed-criticality systems and their related theories are well-established in the real-time systems community, the mode switch mechanisms are not supported natively in ROS 2. Previous research on ROS 2 has primarily focused on formal modeling and timing analysis based on its default scheduling mechanisms, and their extensions to handle mixed critical workloads are nontrivial. Moreover, functionalities in ROS 2 may belong to the same mutually exclusive (ME) callback group, such that only one of them can execute at the same time, leading to priority inversions in their schedules, which have to be accounted for in the schedulability analysis. This work presents the first effort to enable mixed-criticality scheduling and handle ME callback groups in ROS 2 by modifying its executor design. We propose a novel scheduling algorithm, FPP-GEDF-VD with OMLP locking protocols, along with its corresponding schedulability analysis with bounded (priority-inversion) blocking time, tailored for multi-threaded ROS 2 execution of MC workloads with ME callback groups. The effectiveness of our approach is demonstrated through both synthetic workloads and real-world case studies.**

I. Introduction

ROS 2 has emerged as a foundational middleware framework for building modern robotic and autonomous systems. Over nearly a decade, real-time systems researchers have extensively studied ROS 2's timing behavior, producing a well-understood timing model and a range of schedulability analyses for ROS 2. Most of these efforts focus on the default ROS 2 scheduling framework [1], [2], [3], [4], [5], [6]. However, the scheduling model of ROS 2 is quite unique—significantly differing from standard real-time scheduling policies—which limits the direct applicability of prior results from classical real-time scheduling theory. Beyond the default scheme, several works have introduced fixed or dynamic priorities into ROS 2 workloads scheduler to leverage existing real-time scheduling techniques [7], [8], [9], [10].

Modern robotic and autonomous systems are inherently mixed-critical, since they combine tasks with widely varying levels of importance, from high-assurance control loops to low-priority monitoring. These are referred to as mixed-criticality (MC) workloads. To address the complexity and pessimism in modeling worst-case execution times (WCET) of MC workloads, Vestal proposed a seminal framework in MC system [11]. Since then, research in MC systems has flourished within the real-time community, as surveyed in [12]. In an MC system, the scheduler operates in multiple modes: at runtime, if a critical task exceeds its expected execution budget for the current mode, the system switches to a higher mode to preserve timing guarantees for the high-critical tasks.

Although MC scheduling theory is well-established, ROS 2 does not natively support mode-switch mechanisms. Enabling MC in ROS 2 is therefore non-trivial. The only prior work by Wilson *et al*. [13] developed a single-threaded ROS 2-based system to address the reaction time variations on physical events. However, [13] did not fully develop the mode-switch mechanism (not even for single-threaded ROS 2) to enable MC scheduling of MC system, despite the fact that MC models and enabling system mode switches would naturally benefit many ROS 2-based applications. This is the first work on ROS 2 that handles mixed-critical workloads and system mode switches.

Besides, the multi-threaded ROS 2 executor maintains mutually exclusive (ME) callback groups to prevent concurrent access to shared resources by allowing only one per group to execute simultaneously, leading to priority inversions (pi) in the schedule. This is the first work on ROS 2 that handles ME callback groups by adapting an asymptotically optimal locking protocol, called the OMLP [14], to its kernel/executor.

We also consider the integrated case where MC workloads with ME callback groups are implemented on the same ROS 2 platform, and present a *fixed-preemption point global earliest deadline first with virtual deadlines* (FPP-GEDF-VD) scheduling algorithm with the OMLP, and present the corresponding schedulability tests and system implementation details.

Paper Organization. Sec. II presents ROS 2 background and related works. Sec. III describes how we modify the ROS 2 executor to support mixed-criticality scheduling with dynamic priorities, enabling the scope to leverage well-established MC theories to better schedule the processes with varying resource requirements. Sec. IV adds the OMLP locking protocols to handle priority-inversion blocking due to ME callback groups, and describes system design details. Sec. V (and Sec. VI, resp.) focuses on schedulability analysis without (and with) callback groups. Sec. VII evaluates the proposed approaches via simulation/demos and a system-level case study. Finally, Sec. VIII concludes the work and points out future directions.

II. Background and Related Works

This section presents a high-level overview of ROS 2 and its related works. Please refer to [2], [3], [15], [6] for a more

detailed introduction to ROS 2 from a real-time perspective.

ROS 2. ROS 2 is a middleware framework that bridges the operating system and application layers in robotics applications. It provides a client library (*rcl*) and middleware (*rmw*) that interface with open-source/commercial DDS implementations [16], [17], [18]. ROS 2 applications consist of nodes organized into logical groups, each composed of one or more callbacks—timer, subscriber, service, or client—executed via publish-subscribe mechanisms. The timer callback is periodically released, and all other callbacks (denoted with a common term 'regular callbacks') are event-triggered.

Callbacks in ROS 2 are scheduled by an executor, which maintains a `readySet` of callbacks. The default priority is: timer $\succ$ subscriber $\succ$ service $\succ$ client. ROS 2 provides both single-threaded and multi-threaded executors, where the latter supports callback groups: *mutually exclusive* (ME) (only one runs at a time) and *reentrant* (multiple can run concurrently). The executor maintains a `readySet` to register the eligible callbacks to execute. The `readySet` is only refreshed when it is empty or has no eligible callbacks (known as Polling Points). As a result, if a callback becomes eligible (while outside the readySet) while there are other eligible callbacks in the `readySet` (this duration is known as the Processing window), the newly eligible callback will not be considered until the next refresh. The `readySet` is refreshed only when it is empty or contains no eligible callbacks.

Related Works on ROS 2. Casini *et al.* [3] first formally model the default scheduling framework and derive a response time bound for ROS 2. Several works have refined the timing analysis, such as using polling point semantics [4] and starvation modeling [1]. Teper et al. [5] introduced end-to-end timing analysis, and then modeled workloads as DAGs [19]. Choi, Xiang, and Kim [9] proposed fixed-priority chains and static callback-thread mapping to reduce self-blocking. These efforts primarily focus on single-threaded executors.

All recent works on multi-threaded executors highlight the challenges due to concurrency and callback groups. It is concurrently reported in [7], [20], [15] that in default ROS 2 multi-threaded scheduling, priority inversion can happen among mutually exclusive callback groups. Teper et al. [6] further demonstrated the starvation due to priority inversion of the callbacks in a multi-threaded executor. They proposed a software-level modification to ROS 2 executor to prevent starvation due to priority inversion, and the proposed design is verified to be starvation-free and deadlock-free using a model checker. However, priority-inversion (pi) blocking due to the ME callback groups can still persist, and no bounds for pi-blocking are provided. Sobhani, Choi, and Kim [10] adopted fixed-priority scheduling for better performance. Other ROS-related efforts include GPU integration [21], [22], timing disparity analysis [23], message synchronization [23], [24], analysis using modeling tools such as UPPAAL [25], and general ROS 2 performance evaluations [2].

Some prior works [7], [8], [26] have modified both the single-threaded and multi-threaded ROS 2 executors to support dynamic priority scheduling. Notably, although [7] used ME callback groups by applying a non-asymptotically-optimal locking mechanism in its design, it is not clear how the priority inversion cases are analytically handled in [7].

To our knowledge, the only prior attempt to enable MC in ROS 2 was by Wilson *et al.* [13]. However, there are significant differences between our work and [13]. [13] addresses the reaction time of a system to physical events by using simulation to place bounds on the required reaction times under specific conditions or system states. To meet changes in the reaction time requirements, the system can de-prioritize or drop some tasks/task-chains to reduce the blocking time of critical tasks. While some of the mechanisms, such as chain priorities, dropping, are similar to our work, the motivation, conditions, and implementation are different. This work also considers varying execution times between jobs. Unlike [13], the effects of the mode switch apply immediately to running tasks. Moreover, the implementation and experiments in [13] focus on single-threaded executors, and do not address the challenge of mutual exclusion in multithreaded scenarios, nor handle priority inversion due to ME callback groups well.

With multithreaded execution, tasks that use shared resources must be carefully managed. Unlike prior works, we utilize the OMLP algorithm to mitigate pi-blocking and starvation, which can occur in default ROS 2.

III. System Model

We consider a set of n independent processing chains[1] $\Gamma = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ as the workload of ROS 2. Each processing chain (in short, *chain*) consists of a sequence of callbacks. Executors select and dispatch the callbacks in threads to execute following scheduling policies. Our focus in this paper is limited to scheduling ROS 2 workloads inside a single 'multi-threaded' executor. We consider only integer time instances aligned with the processor clock tick's granularity.

Callbacks. In ROS 2, a callback is a piece of code with a particular functionality. According to Footnote 1, each callback belongs to a chain. We denote the jth callback of ith chain as $c_{i,j}$. Each callback $c_{i,j}$ is associated with a label $L_{i,j} \in \{\text{LO}, \text{HI}\}$ that represents the criticality of $c_{i,j}$, *i.e.*, $c_{i,j}$ is either a LO or HI-critical callback. The worst-case execution time (WCET) of a callback $c_{i,j}$ is represented by a two tuple $\left(e^L_{i,j}, e^H_{i,j}\right)$, where $e^L_{i,j}$ and $e^H_{i,j}$ are LO-WCET and HI-WCET, respectively, and $e^L_{i,j} \leq e^H_{i,j}$. For LO-critical callback, $e^L_{i,j} = e^H_{i,j}$. Each callback can potentially release infinitely many instances, where the timer callback is periodically released, and regular callbacks are event-triggered. The kth instance of $c_{i,j}$ is denoted as $c^k_{i,j}$. Each callback instance is scheduled to execute non-preemptively.

A ROS 2 callback system has a single *reentrant callback group* and may have multiple *mutually exclusive callback groups*. Each callback either belongs to the reentrant callback

[1] Any ROS 2 workload graph can be decomposed into independent processing chains, where each chain will contain an independent replica of a shared callback [3], such that no callback (ROS 2 nodes) are shared.

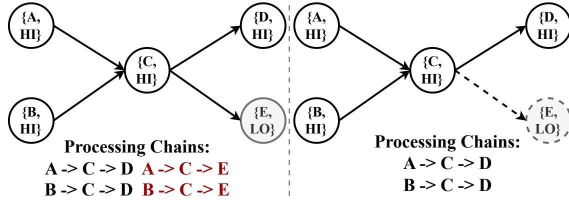


Fig. 1: Example of ROS 2 callback graph. Each callback is denoted with {id, criticality}. Under each callback graph, the corresponding processing chains are listed. Both graphs represent a single system in different operating modes: the left side graph is for LO-mode, whereas the right side graph is for HI-mode, dropping the LO-critical callback from the graph.

group or belongs to one of the mutually exclusive callback groups. For notational simplicity, we index the callback groups by integers where index 0 denotes the reentrant callback group, and each of the positive integers denotes a mutually exclusive callback group. Then, let $\mathcal{G}_{i,j}$ be the index of the callback group the callback $c_{i,j}$ belongs to.

Chains. A chain $\mathcal{C}_i = (c_{i,1}, c_{i,2}, \cdots, c_{i,|\mathcal{C}_i|})$ is a sequence of $|\mathcal{C}_i|$ callbacks, where $c_{i,1}$ is the first callback and $c_{i,|\mathcal{C}_i|}$ is the last callback of the chain. Depending on the type of the first callback, a chain can be classified as time-triggered (if $c_{i,1}$ is a *timer* callback) or event-triggered (if $c_{i,1}$ is a *regular* callback) chain. Except for the first callback, any $c_{i,j}$ instance can only become ready to execute once $c_{i,j-1}$ instance finished its execution, since each callback instance is released by the previous callback's instance in the chain publishing its results (*i.e.*, intermediate callbacks in the chain cannot be time-triggered callbacks). A time-triggered chain $\mathcal{C}_i$ periodically releases an instance of $c_{i,1}$. A chain can potentially release infinite instances, and the kth instance of $\mathcal{C}_i$ is denoted as $\mathcal{C}_i^k$, where $\mathcal{C}_i^k = (c_{i,1}^k, c_{i,2}^k, \ldots, c_{i,|\mathcal{C}_i|}^k)$. A chain $\mathcal{C}_i$ is characterized via tuple $(L_i, \{E_i^L, E_i^H\}, D_i, T_i)$, where

- $L_i \in \{\text{LO}, \text{HI}\}$ is the criticality label of the chain $\mathcal{C}_i$. A chain $\mathcal{C}_i$ is a HI-critical chain ($L_i = \text{HI}$) if all of its callbacks are HI-critical, *i.e.*, $\forall_j, L_{i,j} = \text{HI}$; otherwise, it is a LO-critical chain ($L_i = \text{LO}$) (see Fig. 1).
- E_i^L and E_i^H are the LO-WCET and HI-WCET of the chain $\mathcal{C}_i$, resp. Let $E_{i,n}^L = \sum_{j=1}^n e_{i,j}^L$ be the sum of first n callbacks' LO-WCET of $\mathcal{C}_i$, while $E_{i,0}^L = 0$. Similarly, $E_{i,n}^H = \sum_{j=1}^n e_{i,j}^H$, $E_i^L = E_{i,|\mathcal{C}_i|}^L$ and $E_i^H = E_{i,|\mathcal{C}_i|}^H$.
- T_i is the minimum inter-arrival time, called the *period*, between two chain instances.
- D_i is the relative deadline of the chain. We assume constrained deadlines, *i.e.*, $D_i \leq T_i$.

The utilization of a chain $\mathcal{C}_i$ is defined as $u_i(x) = \frac{E_i^x}{T_i}$, where $x \in \{L, H\}$. Thus, $u_i(L)$ is the utilization of $\mathcal{C}_i$ in LO-system mode. The total utilization of the system is defined as $U(\Gamma, x) = \sum_{\forall i \in \Gamma} \frac{E_i^x}{T_i}$, *e.g.*, $U(\Gamma, L) = \sum_{\forall i \in \Gamma} \frac{E_i^L}{T_i}$ is the total utilization of the task set Γ in LO-mode.

Executor. We consider a multi-threaded executor $\mathcal{E}$ consisting of m working threads $\mathcal{E} = \{\pi_1, \pi_2, \ldots, \pi_m\}$. Aligning with previous works in multi-threaded executor for ROS 2 [7], [10], we consider the one-to-one assignment of each thread π_i to a processor core for maximizing the concurrent executions of callbacks. We assume processors are homogeneous. We further assume a dedicated resource supply to each thread from the corresponding processor and, without loss of generality, all processors are unit-speed ones. So, the total resource supply for m threads is m units per time unit. In the rest of the paper, we will use the terms 'thread' and 'processor' interchangeably, as threads are one-to-one mapped to processors.

System Modes. We consider two operating modes: HI-system mode (abbreviated HI-mode) and LO-mode. The system starts in LO-mode, where each callback $c_{i,j}$ is allowed to execute for up to its $e_{i,j}^L$. If any HI-critical callback overruns its $e_{i,j}^L$, a mode switch from LO-mode to HI-mode is initiated.

Instead of the default ROS 2 executor scheduling scheme, we use the recently proposed ROS 2 multi-threaded executor with a dynamic-priority-based scheduler [7], together with the modifications needed to support mixed-criticality scheduling and mutually exclusive callback groups (see Sec. IV). However, as in default ROS 2, callbacks remain non-preemptive.

Scheduling Algorithm. We employ the fixed-preemption-point global earliest-deadline-first with virtual deadlines (FPP-GEDF-VD) algorithm to schedule callbacks on a multi-threaded executor. An executing chain can be preempted by a higher-priority chain only after its current callback completes. We assign each HI-critical chain a "virtual deadline" $D_i^v = x \cdot D_i$, where $x \leq 1$ is a deadline-shrinkage factor for all HI-chains. Virtual deadlines promote HI-chains in LO-mode, ensuring that after a mode switch, they still have a sufficiently large window to finish any carry-over work.

Correctness Criteria. A MC ROS 2 schedule is correct if:

- Under LO-mode, each callback $c_{i,j}$ signals completion once it uses at most its LO-WCET $e_{i,j}^L$, and each chain $\mathcal{C}_i$ finishes on or before its virtual deadline.
- Under HI-mode, each callback $c_{i,j}$ of any HI-critical chain $\mathcal{C}_i$ may execute for up to $e_{i,j}^H$ and must complete by its (true) deadline. All LO-critical chains are dropped immediately when switching from LO-mode to HI-mode.

IV. System Implementation of ROS 2

We build upon the dynamic priority executor from [7] to support mixed-criticality scheduling. The dynamic priority executor changes the default ROS 2 `readySet` into a unified `readyQueue`. The default `readySet` is split into multiple sets, one of each callback type. During callback selection, all the callbacks of a single callback type are considered before moving on to the next type.

The callbacks in the `readyQueue` are sorted by priority, and all callback types are kept in the same queue. Thus, callbacks run in order of priority when they execute in order of their queue positions. To implement EDF scheduling, the priorities of each callback are set to their absolute deadlines.

In addition to the modification of [7] to support mixed-criticality, we add the global OMLP locking protocol to handle potential priority inversion blocking (or, *pi-blocking*) related to ME callback groups. Fig. 2 shows the overall overview of the MC multi-threaded executor implementation.

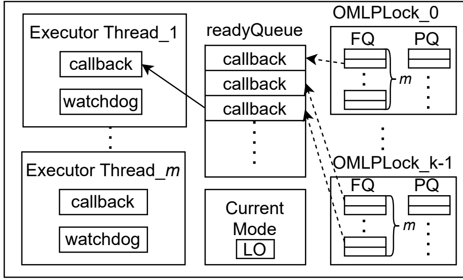


Fig. 2: Implementation overview of a MC multi-threaded ROS 2 executor.

Global OMLP. The global OMLP [14] is a suspension-based locking protocol that has asymptotically optimal pi-blocking under *suspension-oblivious* analysis. The global OMLP ensures $O(m)$ pi-blocking by utilizing a dual-queue structure, with an m-element FIFO queue fed into by a priority queue, as shown in Fig. 3. A callback of an ME callback group is enqueued in the FIFO queue (resp., priority queue) if there are fewer than (resp., at least) m pending callbacks of that callback group. The callback instance at the head of the FIFO queue holds the lock and is eligible to be scheduled. When the callback at the head of the FIFO queue (i.e., the resource holder) completes, it is dequeued, the next callback (if any) in the FIFO queue becomes eligible to be scheduled, and the highest-priority callback (if any) in the priority queue is moved to the tail of the FIFO queue. The global OMLP uses *priority inheritance* [27] to ensure a lock holder is scheduled when a top-m-highest-priority job is waiting on the lock. In ROS 2, non-preemptive execution of callbacks ensures the progress of lock-holding callback instances once they are scheduled. However, priority inheritance is still needed to ensure that when a callback instance holding a lock finishes, the next queued callback will be scheduled if it is blocking a callback instance whose priority is among the top m.

OMLP Implementation. To maintain compatibility with existing ROS 2 systems, we do not add the OMLP queues to the `CallbackGroup` class, and instead add a `CallbackGroup` $\leftrightarrow$ `OMLPLock` mapping within the executor. All callbacks within the same callback group share a pointer to a single `CallbackGroup`, which stores a flag indicating whether an executor thread is currently executing a callback from that group. The default ROS 2 executors check this flag to ensure that the mutual exclusion constraint is met.

The `OMLPLock` class holds the two queues needed for the OMLP: FQ and PQ. FQ is a FIFO queue that holds up to m ready callbacks. Any callbacks released while FQ is full are placed into a a priority queue PQ that follows the same sorting method as the main `readyQueue`. To enforce mutual exclusion within the group, only the head of FQ can execute.

One `OMLPLock` object exists for each ME callback group. The FQ and PQ queues of the `OMLPLocks` do not replace the `readyQueue`. A newly released callback is added to both the `readyQueue` and the appropriate `OMLPLock` queue, as shown in the procedure *ReleaseCallback(job)* in Algorithm 1.

An executor thread starts at the beginning of the

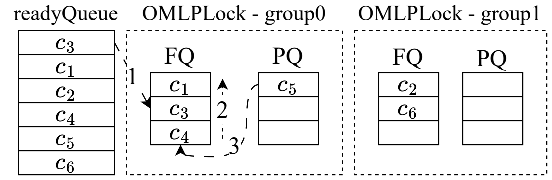


Fig. 3: The `readyQueue` and the `OMLPLocks` of a 3-threaded executor with two mutually-exclusive callback groups. During a callback selection, an executor thread checks the first callback of the `readyQueue`, c_3. Since the callback is part of an ME group, it checks the callback's associated `OMLPLock`, `group0`. There is another callback at the head of that group's FQ, c_1, with a lower priority that inherits the priority of c_3. The executor checks whether c_1 can be executed, and starts it. After c_1 completes, it is removed from both the `readyQueue` and from `group0`'s FQ. c_3 becomes the new head of FQ, and if there is another callback in PQ, it is moved to the end of FQ.

`readyQueue`. The executor checks the properties of each callback, including group membership, data availability from the *rmw*, and criticality level to decide if the callback can be run. If so, it is removed from the `readyQueue` and executed within the thread.

OMLP Priority Inheritance. In the default ROS 2 executors, the callback group check is simple: the executor checks whether a flag within the `CallbackGroup` that indicates whether a member of that group is currently executing; if set, the callback is skipped. We modified the group membership check so that it finds the `OMLPLock` object for the callback. In the OMLP, the head of FQ inherits the priority of the highest-priority callback within the FQ and PQ. Since the `readyQueue` is traversed in order of priority, the highest priority callback of a group will be evaluated first, even if it is not the head of its group FQ. If the executor finds a callback that is not the head of its group FQ, it will substitute it (without changing any of the queues) with the head of FQ before performing the other checks. This implements priority inheritance without potentially expensive resorts of the `readyQueue`. This step is shown at the end of the procedure *get_next_executable()* in Algorithm 1.

If the task passes all of the checks, it is executed within the executor thread. Once the task completes, it is removed from FQ. If available, the highest-priority callback in PQ is removed and placed at the end of FQ, shown in the procedure *ExecutorThread()* in Algorithm 1.

Mixed Criticality (MC). We implement MC scheduling by adding three parameters to each callback: criticality, LO-WCET, HI-WCET. Each callback chain has a x value that specifies the scaling of the relative deadline applied to the chain (depending on the system mode). To detect when a task exceeds its declared runtime, each executor thread has an accompanying watchdog thread. The watchdog thread is synchronized with the executor thread with two semaphores: `cb_begin` and `cb_end`. When a callback is started by an executor thread, the thread `posts` to `cb_begin`. After receiv-

ing `cb_begin`, the watchdog thread calls `sem_timedwait` on `cb_end` along with the expected runtime of the executing job. When the job completes, the executor `posts` to `cb_end`, allowing the `sem_timedwait` in the watchdog thread to complete and return 0, indicating that the callback completed on time. If the job takes *longer* than the expected runtime, then the `sem_timedwait` call expires and returns -1, indicating that the job took too long. The algorithm is given in the procedure *WatchdogThread()* in Algorithm 1.

To perform the mode-switch, the executor checks every callback in the `readyQueue` and the OMLP *FQ*s. If a callback is a HI-critical callback, its deadline is moved to its actual absolute deadline (d_i). If the callback is a LO-critical callback, it is removed from the queue. This ensures that all callbacks in the queues are HI-critical callbacks and are sorted according to their actual absolute deadlines. Callbacks are only added to the queue if they are HI-callbacks and allowed to execute up to their HI-WCET.

Aborting LO-critical callbacks at mode-switch instant. If any LO-critical callbacks are executing during a mode-switch, the executor sends `SIGUSR2`, which is a POSIX signal that the system integrator can assign meaning to. In our implementation, the callback must respond to this signal by giving up any resources and exiting as quickly as possible. For use cases like shared data structures, consistency of shared resources can be ensured if callbacks are *interruptible* [28], [29] and can handle the stop request.

A. Mixed-Criticality Scheduling Model for Executor

Our proposed executor maintains a `readyQueue` Ω during runtime to record the dynamic priority of all eligible callbacks. The dynamic priority of a callback $c_{i,j}$ is determined using the absolute deadline of chain $\mathcal{C}_i$; *i.e.*, all callbacks within a chain share the same deadline. For instance, if the arrival time of chain instance $\mathcal{C}_i^k$ is a_i^k and the system is in LO-mode (resp., HI-mode), then the absolute deadline of the chain instance is $d_i^k = a_i^k + D_i^v$ (resp., $d_i^k = a_i^k + D_i$). Now, any callback $c_{i,j}^k$ (for $1 \leq j \leq |\mathcal{C}_i|$) will have an absolute deadline of d_i^k. A callback with an earlier deadline has a higher priority than one with a later deadline. In other words, the callback scheduling decisions are determined following the GEDF-VD algorithm.

An executor thread is either 'busy' when it is executing a callback instance, and 'idle' when it is not. A *dispatch point* occurs whenever a thread transitions to the idle state. At the is point, the ready queue Ω is updated with all pending callbacks. Among the callbacks in Ω, callbacks are checked in the order of priority (highest priority first). The idle thread selects the highest priority callback that is eligible to run. A callback runs non-preemptively as soon as it is selected. A thread *sleeps* if it fails to find a callback, while it can be woken by the release of the next callback, which leads to a repetition of the process.

To update Ω, the executor checks all callback types in the system for eligible callbacks. Any new releases will be placed in Ω according to the priority provided by the scheduling parameters. Callbacks in the Ω persist between updates so that the queue does not need to be entirely rebuilt during updates.

Algorithm 1: Mixed-Criticality Executor with Watchdog and OMLP

```
Procedure ExecutorThread()
    watchdog_thread ← start_thread(watchdog_func, cb_begin,
      cb_end);
    while rclcpp::ok() do
        wait_for_work();
        sched_mutex.lock();
        next_executable ← get_next_executable();
        readyQueue.remove(next_executable);
        sched_mutex.unlock();
        watchdog_thread.current_task ← next_executable;
        sem_post(cb_begin);
        run(next_executable);
        sem_post(cb_end);
        // Remove from FQ, and promote one job
           from PQ
        omlp_lock ← omlpLocks[next_executable];
        if omlp_lock ≠ null then
            omlp_lock.FQ.remove(0);
            if omlp_lock.PQ.size() > 0 then
              omlp_lock.FQ.push(omlp_lock.PQ.pop()) ;
Procedure WatchdogThread()
    while rclcpp::ok() do
        sem_wait(cb_begin);
        start_time ← now();
        expected_exec_time ← next_executable.expected_runtime;
        expected_stop_time ← start_time + expected_exec_time;
        result ← sem_timed_wait(cb_end, expected_stop_time);
        if result = 0 then // job stopped on time
            continue;
        else
            // job ran over-time
            MODE ← HI;
            stop_lo_callbacks();
            sem_wait(cb_end);
Procedure get_next_executable()
    for job in readyQueue do
        // can_run checks if the job can run in
           the current mode, has data in the
           RMW, etc
        if not can_run(job) then continue ;
        // OMLP Priority Inheritance
        omlp_lock ← omlpLocks[job.group]
        if omlp_lock ≠ null ∧ omlp_lock.FQ[0] ≠ job then
            // Use the head of FQ, if possible
            if can_run(omlp_lock.FQ[0]) then job =
              omlp_lock.FQ[0] ;
        return job;
    return null;
Procedure ReleaseCallback(job)
    readyQueue.insert(job);
    omlp_lock ← omlpLocks[job];
    if omlp_lock ≠ null then
        if omlp_lock.FQ.size() < M then
        omlp_lock.FQ.push(job) ;
        else omlp_lock.PQ.insert(job) ;
```

Note that not all callbacks in Ω are eligible to run. Depending on the membership of callback groups, a callback instance $c_{i,j}$ in Ω is either 'eligible' or 'ready and blocked' (*R-blocked*):

- If the callback $c_{i,j}$ is a member of the reentrant callback group, as soon as $c_{i,j}$ enters Ω, it is *eligible* to run.
- If the callback $c_{i,j}$ is a member of an ME callback group, there can be two cases. **Case A:** if there are no other callbacks (including an instance of $c_{i,j}$ itself) from the

same ME group currently executing in a thread, then the callback becomes *eligible* as soon as it enters Ω. **Case B:** otherwise, the callback $c_{i,j}$ is *R-blocked* and skipped during task selection.

When a callback is R-blocked, *pi-blocking* may occur. Specifically, pi-blocking occurs when a callback is R-blocked, and this blocking cannot be attributed to a higher-priority callback.

We support ME callback groups by requiring each callback in an ME callback to hold a mutual exclusion (mutex) lock before being ready to be scheduled. Thus, a callback in an ME callback group *issues* a request for the mutex lock associated with the ME callback group and becomes R-blocked when it is released and remains so until it is granted the lock. We use the **global OMLP** locking protocol (discussed earlier) to grant access to such mutex locks.

A mode switch happens in response to a HI-critical task exceeding its e^L value. After mode-switch, LO-critical eligible callbacks in Ω become ineligible, and the virtual absolute deadlines of the HI-critical tasks are replaced with their absolute deadlines.

V. Schedulability Analysis without Callback Groups

This section presents the schedulability analysis for the ROS 2 processing chains under FPP-GEDF-VD scheduling without callback groups, and then the next section will present the schedulability analysis with callback groups.

Our executor design enables us to utilize a standard schedulability analysis framework, such as busy window-based analysis, for GEDF scheduling of tasks with fixed preemption points on homogeneous multiprocessor platforms. The schedulability analysis of ROS 2 processing chains within an MC, multi-threaded executor under GEDF can be directly mapped to the MC scheduling of tasks with fixed preemption points on homogeneous multiprocessors. Unfortunately, no existing work provides a schedulability analysis for MC tasks of fixed preemption points on homogeneous multiprocessors.

We first define the runtime behavior of the MC system based on its operation in LO and HI modes and during mode switches.

Definition 1 (Transient State)**.** *Let the system be operating in* LO*-mode and a mode-switch to* HI*-mode be initiated at time t_s. The transient state is defined as the longest time interval $[t_s, t_s + T_{transient}]$ during which all the processors (i.e., threads in ROS 2 executor) in the system remains busy executing tasks.*

In the transient state, the system needs to process carry-over jobs with adjusted workload parameters (such as execution time and relative deadlines) that were released in the LO-mode but remain unfinished at the mode-switch instant, as well as all the newly released jobs in the HI-mode.

Definition 2 (Steady State)**.** *Any other state than the transient state in the system is defined as steady state.*

In a steady state, the system's behavior is completely governed by either $\mathcal{C}_i$'s ($\in \Gamma$) with $\{E_i^L, D_i^v\}$ in LO-mode or $\mathcal{C}_i$'s ($\in \Gamma^H$) with $\{E_i^H, D_i\}$ in HI-mode, and any transient effects (such as carry-over callback instance or parameter changes) from a previous mode are no longer present.

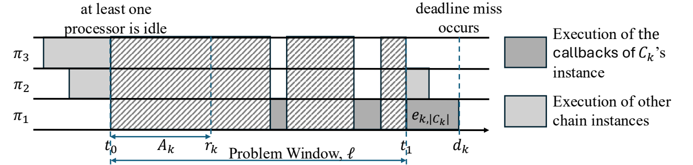


Fig. 4: Problem window for ROS 2 chains with non-preemptive callbacks.

Following, we first derive the schedulability conditions for the steady state, and then for the transient state.

Let us define $[\![a]\!]_b := \min\{a, b\}$, $[\![a]\!]^b := \max\{a, b\}$, $[\![a]\!]_c^b := \min\{\max\{a, b\}, c\}$, and $a\%b := a \mod b$ for the compactness of notations.

A. Schedulability Analysis for the Steady State

Suppose a processing chain set Γ is not schedulable. Let $\mathcal{C}_k^j$ be the first chain instance that missed the deadline. Similar to the preemptive [30] and non-preemptive [31] GEDF schedulability analysis, we use a problem window-based analysis, which is defined as follows and illustrated in Fig. 4.[2]

Problem Window (PW). Given the chain instance $\mathcal{C}_k^j$, which is released at r_k and has an absolute deadline at d_k. Let $t_0 - 1$ be the latest instant earlier than r_k, where at least one processor is idle, satisfying that all processors are busy in $[t_0, r_k]$. Let $A_k = r_k - t_0$, a non-negative integer. As the callbacks are non-preemptively executed, we do not need to analyze until d_k. Let $t_1 = d_k - e_{k,|\mathcal{C}_k|} + 1$ be the latest instant such that if the last callback of $\mathcal{C}_k^j$ starts executing at or after t_1, $\mathcal{C}_k^j$ must miss the deadline. So the length of the problem window (i.e., PW) is $\ell = t_1 - t_0 = A_k + D_k - e_{k,|\mathcal{C}_k|} + 1$.

The necessary condition for the deadline miss to occur is that the worst-case work done in PW by all chain instances in Γ except $\mathcal{C}_k^j$ is larger than $(\ell - E_k + e_{k,|\mathcal{C}_k|}) \cdot m$ (the crosshatched area in Fig. 4). We will compute the worst-case work done by any chain in PW, denoted as $\mathcal{I}(\mathcal{C}_i)$, and $\sum_{\forall \mathcal{C}_i} \mathcal{I}(\mathcal{C}_i)$ as an upper bound on total work done in PW. We denote a chain as *carry-in* if it releases an instance before t_0, has a deadline after t_0, and contributes a fraction of its execution in PW. Other chains are *non-carry-in* chains. Let denote the work done by a carry-in chain as $\mathcal{I}^{CI}(\mathcal{C}_i)$ and non-carry-in chain as $\mathcal{I}^{NC}(\mathcal{C}_i)$. As at least one processor is idle at t_0, at most $m-1$ chains can be carry-in chains in PW.

In the following, we first calculate the interference of non-carry-in chains ($\mathcal{I}^{NC}(\mathcal{C}_i)$) and then carry-in chains ($\mathcal{I}^{CI}(\mathcal{C}_i)$). We define $\mathcal{C}_i^{last}$ as the last instance released in PW by $\mathcal{C}_i$.

Computing $\mathcal{I}^{NC}(\mathcal{C}_i)$.

First, we will compute $\mathcal{I}^{NC}(\mathcal{C}_i)$ with $i = k$, which is the worst-case work done by the $\mathcal{C}_k$'s instances in A_k. As there is no carry-in instance, total work done is given by:

$$\mathcal{I}_1^{NC}(\mathcal{C}_k) = \left\lfloor \frac{A_k}{T_k} \right\rfloor \cdot E_k \tag{1}$$

[2] To apply the analysis to both LO- and HI-mode steady states, we omitted criticality levels from parameters, *e.g.*, we use E_k instead of E_k^L or E_k^H.

Next, we will compute $\mathcal{I}^{NC}(\mathcal{C}_i)$ with $i \neq k$.

Lemma 1. *Total work done by a non-carry-in chain $\mathcal{C}_i$ ($i \neq k$) in PW is given by,*

$$\mathcal{I}_2^{NC}(\mathcal{C}_i) = \begin{cases} \left\lfloor \frac{\ell}{T_i} \right\rfloor \cdot E_i + [\![E_i]\!]_\gamma ; & \text{if } \alpha \leq L \\ \left\lfloor \frac{\ell}{T_i} \right\rfloor \cdot E_i + [\![\mathbb{A}]\!]_\gamma ; & \text{if } \alpha > L \wedge \beta < A_k \\ \left\lfloor \frac{\ell}{T_i} \right\rfloor \cdot E_i + \left[\!\!\left[\sum_{l=1}^{[\![|\mathcal{C}_i|]\!]_{|\mathcal{C}_k|-1}} (e_{i,l} - 1) \right]\!\!\right]_\gamma ; & \\ & \text{if } \alpha > L \wedge \beta \geq A_k \end{cases} \tag{2}$$

where $L = A_k + D_k$, $\alpha = \left\lfloor \frac{\ell}{T_i} \right\rfloor T_i + D_i$, $\beta = \left\lfloor \frac{\ell}{T_i} \right\rfloor T_i$, $\gamma = [\![L - E_k + 1 - \beta]\!]^0$, *and*

$$\mathbb{A} = \begin{cases} E_i; & \text{if } A_k - \beta > E_{i,|\mathcal{C}_i|-1} \\ E_{i,q} + \sum_{l=q+1}^{[\![|\mathcal{C}_i|]\!]_{|\mathcal{C}_k|-q+1}} (e_{i,l} - 1); & \\ \quad \text{if } E_{i,q-1} < A_k - \beta \leq E_{i,q} \end{cases} \tag{3}$$

Proof. Note that the worst case of $\mathcal{I}^{NC}(\mathcal{C}_i)$ ($i \neq k$) occurs when one of $\mathcal{C}_i$ instances is released at the time-instant t_0 [30], [31]. Excluding $\mathcal{C}_i^{last}$, total work done by $\mathcal{C}_i$ is $\lfloor \frac{\ell}{T_i} \rfloor \cdot E_i$. Depending on the ordering of d_k and the deadline of $\mathcal{C}_i^{last}$, there are two cases to compute the execution of $\mathcal{C}_i^{last}$ in PW:
Case 1: $\mathcal{C}_i^{last}$ has deadline no later than d_k. Based on the position of the absolute deadlines of $\mathcal{C}_i^{last}$ and $\mathcal{C}_k^j$, we find: $\alpha \leq L$, where $\alpha = \left\lfloor \frac{\ell}{T_i} \right\rfloor T_i + D_i$. The total execution of $\mathcal{C}_i^{last}$ must be included in the work done by $\mathcal{C}_i$ in PW (see Fig. 5a).
Case 2: $\mathcal{C}_i^{last}$ has deadline later than d_k. Here, $\alpha > L$. Based on $\mathcal{C}_i^{last}$'s release time, we consider two sub-cases:

Case 2a: $\mathcal{C}_i^{last}$ is released earlier than r_k. The earlier release time of $\mathcal{C}_i^{last}$ than r_k can be represented as $\beta < A_k$, where $\beta = \left\lfloor \frac{\ell}{T_i} \right\rfloor \cdot T_i$. Since $\beta < A_k$, the callbacks of $\mathcal{C}_i^{last}$ can execute in $A_k - \beta$ before the first callback of $\mathcal{C}_k^j$. Total computation of $\mathcal{C}_i^{last}$ is computed precisely using $\mathbb{A}$ in Eq. 3, where in the first case, entire $\mathcal{C}_i^{last}$ can finish execution before the start of $c_{k,1}^j$. While in the second case, the first q callbacks of $\mathcal{C}_i^{last}$ start executing before $c_{k,1}^j$. The remaining callbacks can non-preempting block $\mathcal{C}_i^{last}$, where any $c_{i,n}^{last}$ can block $\mathcal{C}_k^j$ at max $e_{i,n} - 1$ as it has started one-time unit earlier than $\mathcal{C}_k^j$'s callback becomes eligible to execute (illustrated in Fig. 5b).

Case 2b: $\mathcal{C}_i^{last}$ is released at or after r_k ($\beta \geq A_k$). Since $\mathcal{C}_k^j$ has higher priority than $\mathcal{C}_i^{last}$, $c_{k,1}^j$ can not be blocked by $\mathcal{C}_i^{last}$'s callbacks. The remaining $|\mathcal{C}_k| - 1$ callback instances of $\mathcal{C}_i$ can experience non-preempting blocking by $\mathcal{C}_i^{last}$ and the interference is shown in the third case of Eq. 2. Fig. 5c illustrates this case. □

So, the non-carry-in interference of a chain is computed as,

$$\mathcal{I}^{NC}(\mathcal{C}_i) = \begin{cases} \mathcal{I}_1^{NC}(\mathcal{C}_i) & \text{if } i = k \\ \mathcal{I}_2^{NC}(\mathcal{C}_i) & \text{if } i \neq k \end{cases} \tag{4}$$

Computing $\mathcal{I}^{CI}(\mathcal{C}_i)$.

Let us define a function to compute the work done by the carry-in job of any chain $\mathcal{C}_i$ in PW as:

$$f_i(\delta) = [\![\delta - T_i + D_i]\!]_{E_i}^0 ,$$

where δ is the difference between t_0 and the first release instant r_i^1 of $\mathcal{C}_i$ in PW, *i.e.*, $\delta = r_i^1 - t_0$. Maximum available window for carrying-in instance of $\mathcal{C}_i$ to execute in PW is $\delta - (T_i - D_i)$.

First, we will compute $\mathcal{I}^{CI}(\mathcal{C}_i)$ with $i = k$, which is given,

$$\mathcal{I}_1^{CI}(\mathcal{C}_k) = \left\lfloor \frac{A_k}{T_k} \right\rfloor \cdot E_k + f_k(A_k \% T_k). \tag{5}$$

Next, we will compute $\mathcal{I}^{CI}(\mathcal{C}_i)$ with $i \neq k$.

Lemma 2. *Total work done by a carry-in chain $\mathcal{C}_i$ ($i \neq k$) is given by,*

$$\mathcal{I}_2^{CI}(\mathcal{C}_i) = \begin{cases} \left\lfloor \frac{A_k + D_k}{T_i} \right\rfloor \cdot E_i + f_i((A_k + D_k) \% T_i); \\ \quad \text{if } D_i \leq D_k \wedge S_i \geq e_{k,|\mathcal{C}_k|} \\ \left(\left\lfloor \frac{\ell - E_i}{T_i} \right\rfloor + 1 \right) \cdot E_i + f_i((\ell - E_i) \% T_i); \\ \quad \text{if } D_i \leq D_k \wedge S_i < e_{k,|\mathcal{C}_k|} \\ \left[\!\!\left[\left\lfloor \frac{A_k - 1}{T_i} \right\rfloor \cdot E_i \right]\!\!\right]^0 + \left[\!\!\left[e_{i,1} + \sum_{l=2}^{|\mathcal{C}_k|} (e_{i,l} - 1) \right]\!\!\right]_{S_k} \\ \quad + f_i((A_k - 1) \% T_i); \\ \quad \text{if } D_i > D_k \wedge |\mathcal{C}_i| \geq |\mathcal{C}_k| \\ \left\lfloor \frac{\ell'}{T_i} \right\rfloor \cdot E_i + [\![E_i - |\mathcal{C}_i|]\!]_{D_k - E_k} + f_i(\ell' \% T_i); \\ \quad \text{if } D_i > D_k \wedge |\mathcal{C}_i| < |\mathcal{C}_k| \end{cases} \tag{6}$$

where $S_l = D_l - E_l^H$, $\forall \mathcal{C}_l$ *and*
$\ell' = \left[\!\!\left[\ell - \left(E_i + \sum_{j=|\mathcal{C}_k|-|\mathcal{C}_i|+1}^{|\mathcal{C}_k|-1} e_{k,j} - |\mathcal{C}_i| + 1 \right) \right]\!\!\right]^{A_k}$.

Proof. First, we note that the worst-case interference for a chain $\mathcal{C}_i$ with a carry-in job occurs when $\mathcal{C}_i^{last}$ has maximum execution in PW [31]. To compute $\mathcal{I}_2^{CI}(\mathcal{C}_i)$, we will consider two cases based on the relative deadlines of chains $\mathcal{C}_k$ and $\mathcal{C}_i$:
Case 1: $D_i \leq D_k$. Depending on $S_i = D_i - E_i$ and $e_{k,|\mathcal{C}_k|}$, there are two different cases as follows:

Case 1a: $S_i \geq e_{k,|\mathcal{C}_k|}$. Fig. 6a illustrates the scenario for maximum interference for this case. To allow maximum window for the carry-in job, the deadline of $\mathcal{C}_i^{last}$ is aligned with d_k. The overall interference by $\mathcal{C}_i$ for this case is shown in the first case of Eq. 6.

Case 1b: $S_i < e_{k,|\mathcal{C}_k|}$. Fig. 6b illustrates the scenario for maximum interference for this case. To allow maximum execution of $\mathcal{C}_i^{last}$ in the problem window, the deadline of $\mathcal{C}_i^{last}$ has to be earlier than d_k. The overall interference by $\mathcal{C}_i$ for this case is shown in the second case of Eq. 6.
Case 2: $D_i > D_k$. For maximum interference of $\mathcal{C}_i$ in PW, both the interference from $\mathcal{C}_i^{last}$ and the carry-in job should be maximized. However, as $D_i > D_k$ and needs to maximize the window for carry-in instance, the priority of $\mathcal{C}_i^{last}$ can be lower than $\mathcal{C}_k^j$, and so the computation of $\mathcal{C}_i^{last}$ can only be done by non-preemptive blocking in PW. Depending on $|\mathcal{C}_i|$ and $|\mathcal{C}_k|$, the following two cases are considered:

Case 2a: $|\mathcal{C}_i| \geq |\mathcal{C}_k|$. As the number of callbacks in $\mathcal{C}_i$ is greater or equal to the number of callbacks in $\mathcal{C}_k$, at most $|\mathcal{C}_k|$

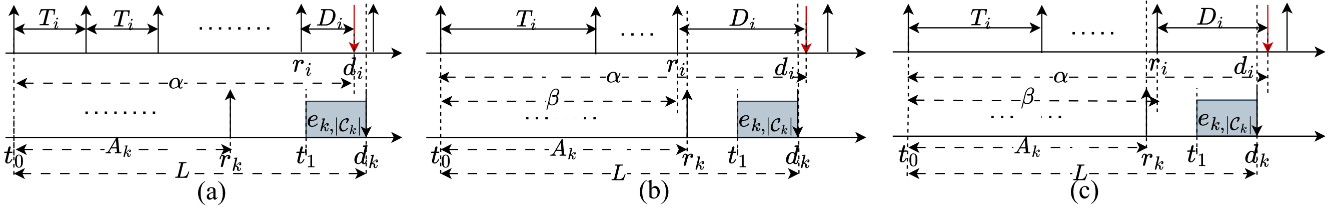


Fig. 5: Non-carry-in interference. (a) Case 1: $\alpha \leq L$, (b) Case 2a: $\alpha > L \wedge \beta < A_k$, and (c) Case 2b: $\alpha > L \wedge \beta \geq A_k$.

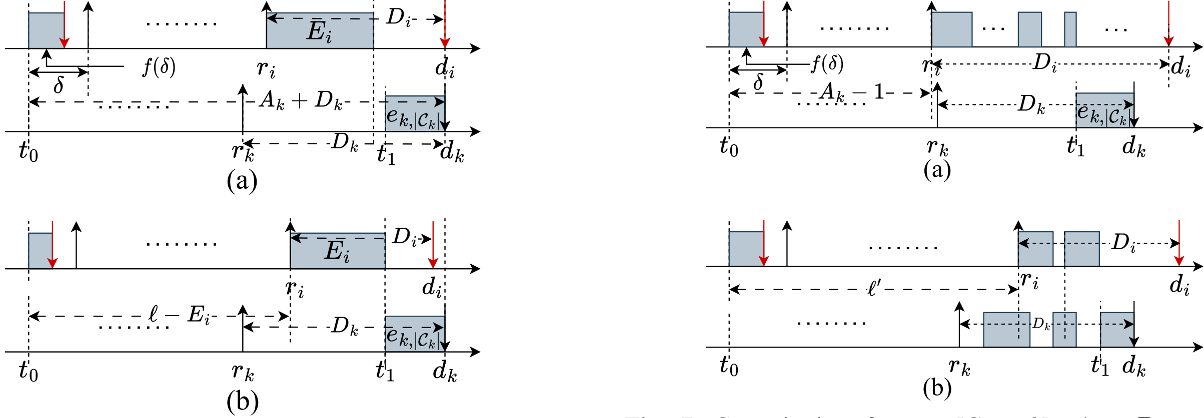


Fig. 6: Carry-in interference [Case 1] when $D_i \leq D_k$. (a) [**Case 1a**] $S_i \geq e_{k,|\mathcal{C}_k|}$: full execution of $\mathcal{C}_i^{last}$ can be done in problem window aligning with d_i and d_k; and (b) [**Case 1b**] $S_i < e_{k,|\mathcal{C}_k|}$: for full execution of $\mathcal{C}_i^{last}$ in the problem window, d_i must be less than d_k.

Fig. 7: Carry-in interference [Case 2] when $D_i > D_k$. (a) [**Case 2a**] $|\mathcal{C}_i| \geq |\mathcal{C}_k|$: maximum $|\mathcal{C}_k|$ callback instances of $\mathcal{C}_i^{last}$ can execute in the problem window if the $c_{i,1}^{last}$ start executing at $r_k - 1$; (b) [**Case 2b**] $|\mathcal{C}_i| < |\mathcal{C}_k|$: maximum interference occurred by maximizing ℓ' while all callback instances of $\mathcal{C}_i^{last}$ executed in the problem window.

callbacks from $\mathcal{C}_i^{last}$ can execute in PW by non-preemptive blocking. The $c_{i,1}^{last}$ must release and start execution one time unit earlier than r_k for the maximum interference in PW. Fig. 7a illustrates the scenario for maximum interference.

Case 2b: $|\mathcal{C}_i| < |\mathcal{C}_k|$. As $|\mathcal{C}_i| < |\mathcal{C}_k|$, all callbacks of $\mathcal{C}_i^{last}$ can execute in PW by non-preemptive blocking. To get the maximum interference of $\mathcal{C}_i$ in PW, callback instances of $\mathcal{C}_i^{last}$ must execute as late as possible. Fig. 7b illustrates the scenario for maximum interference, maximizing the window ℓ'. □

So, the carry-in interference of any chain is computed as,

$$\mathcal{I}^{CI}(\mathcal{C}_i) = \begin{cases} \mathcal{I}_1^{CI}(\mathcal{C}_i) & \text{if } \ i = k \\ \mathcal{I}_2^{CI}(\mathcal{C}_i) & \text{if } \ i \neq k \end{cases} \tag{7}$$

Let us define $\mathcal{I}^{\text{diff}}(\mathcal{C}_i) = \mathcal{I}^{CI}(\mathcal{C}_i) - \mathcal{I}^{NC}(\mathcal{C}_i)$ and $\Delta_{\mathcal{I}^{\text{diff}}}^{m-1}$ is the sum of first $m-1$ items of $\mathcal{I}^{\text{diff}}$ where $\mathcal{I}^{\text{diff}}$ is a non-increasing list of $\mathcal{I}^{\text{diff}}(\mathcal{C}_i)$.

Theorem 1 (Schedulability test for steady state). *A set of processing chains is schedulable in steady state under FPP-GEDF on m processors, if for any chain $\mathcal{C}_k$ and for any A_k, the following condition is satisfied:*

$$\sum_{\forall \mathcal{C}_i \in \Gamma} \mathcal{I}^{NC}(\mathcal{C}_i) + \Delta_{\mathcal{I}^{\text{diff}}}^{m-1} \leq (A_k + D_k - E_k + 1) \cdot m \tag{8}$$

Bounding A_k.

Lemma 3. *For a set of processing chains with utilization $U(\Gamma) < m$, if the condition (Eq. 8) in Theorem 1 is to be violated for any A_k, then it is violated for some A_k that satisfies the condition below:*

$$A_k < \frac{\sum_{\forall i} E_i + \Delta_E^{m-1} + m \cdot (E_k - e_{k,|\mathcal{C}_k|})}{m - U(\Gamma)} - D_k + E_k - 1$$

where Δ_E^{m-1} is the sum of first $m-1$ items of list E_i's ordered no-increasingly.

Proof. $\mathcal{I}^{NC}(\mathcal{C}_i)$ can be upper-bounded by,

$$\mathcal{I}^{NC}(\mathcal{C}_i) \leq u_i \cdot \ell + E_i$$

and $\mathcal{I}^{CI}(\mathcal{C}_i)$ can be upper bounded by,

$$\mathcal{I}^{CI}(\mathcal{C}_i) \leq u_i \cdot \ell + 2 \cdot E_i$$

To violate the condition in Theorem 1, it must be true,

$$\sum_{i \in \Gamma} u_i \cdot \ell + \Delta_E^{m-1} > (\ell - e_{k,|\mathcal{C}_k|+1}) \cdot m$$

Replacing $\ell = A_k + D_k - e_{k,|\mathcal{C}_k|} + 1$ and rearranging the above inequality, we get–

$$A_k < \frac{\sum_{\forall i} E_i + \Delta_E^{m-1} + m \cdot (E_k - e_{k,|\mathcal{C}_k|})}{m - U(\Gamma)} - D_k + E_k - 1$$

□

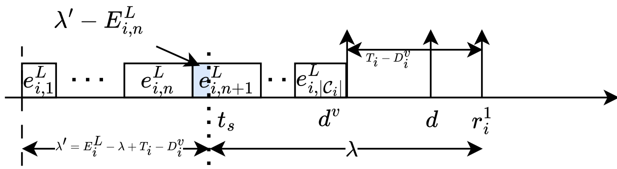


Fig. 8: Illustration of $h_i()$ computation in Lemma 4.

Note. (i) The above schedulability analysis is generally applicable for any non-MC workloads. Therefore, this test can be directly used for non-MC FPP-GEDF scheduling of ROS 2 executor. (ii) In case of using the above analysis for LO**-mode steady state**, all the deadlines should be replaced by virtual deadlines, *i.e.*, $D_i = D^v_i$, and $D_k = D^v_k$; the execution times by LO-WCET, *i.e.*, $E_i = E^L_i$, and $e_{i,j} = e^L_{i,j}$. For HI**-mode steady state**, only HI-chains need to be considered with actual deadlines D_i and HI-WCETs, *i.e.*, $E_i = E^H_i$, and $e_{i,j} = e^H_{i,j}$.

B. Schedulability Analysis for the Transient State

Similar to the steady state analysis, consider a chain instance $\mathcal{C}^j_k$, released at r_k with an absolute deadline (located in HI-mode) d_k, missed the deadline. Let us consider t_s as the mode-switch instant and define $B_k = r_k - t_s$. Let $t_1 = d_k - e^H_{k,|\mathcal{C}_k|} + 1$ be the latest instant such that if the last callback of $\mathcal{C}^j_k$ starts executing at or after t_1, $\mathcal{C}^j_k$ must miss the deadline. So the length of PW is $\ell = t_1 - t_s$.

As the starting point of PW is the mode-switch instant, all chains can have a carry-in instance in PW. So, we will consider only carry-in interference for transient state analysis.

Note that $\mathcal{C}^j_k$ can be released in LO-mode and then miss the deadline in HI-mode, *i.e.*, mode-switch can occur after r_k. Therefore, B_k can be negative, too. Therefore, we will compute the interference $\mathcal{I}^{ST}(\mathcal{C}_i)$ for any chain $\mathcal{C}_i$ in PW for two scenarios: $\mathcal{I}^{ST_{<0}}(\mathcal{C}_i)$ if $B_k < 0$ and $\mathcal{I}^{ST_{\geq 0}}(\mathcal{C}_i)$ if $B_k \geq 0$.

Lemma 4. *Let λ be the difference between t_s and the first release instant r^1_i of any $\mathcal{C}_i$ in the problem window, i.e., $\lambda = r^1_i - t_s$. Then, an upper bound of work done by the carry-in job of chain $\mathcal{C}_i$ is given by,*

$$h_i(\lambda) = \begin{cases} 0; & \text{if, } \lambda < T_i - D^v_i \\ E^H_i; & \text{if, } \lambda - T_i + D^v_i \geq E^L_i \\ E^H_i - E^H_{i,n} - \lambda' + E^L_{i,n}; & \text{otherwise} \end{cases}$$

where the auxiliary term λ' is defined as $\lambda' = E^L_i - \lambda + T_i - D^v_i$ and index n is chosen such that $E^L_{i,n} < \lambda' \leq E^L_{i,n+1}$.

Proof. If $\lambda < T_i - D^v_i$, then the job has already completed its execution in the LO-mode. Therefore, no carry-in execution for the job in PW. In the second case, the chain instance can start executing at or after the mode-switch instant, and therefore can execute entirely in PW. The carry-in execution for the last case is illustrated in Fig. 8 for worst-case scenarios. □

Computing $\mathcal{I}^{ST_{<0}}(\mathcal{C}_i)$.

Let us first compute $\mathcal{I}^{ST_{<0}}(\mathcal{C}_i)$ for $i = k$. Since there is only the job under consideration in PW from $\mathcal{C}_k$, the interference $\mathcal{I}^{ST_{<0}}_1(\mathcal{C}_k) = 0$, which is computed for the instances in PW except the one missing deadline. However, the maximum execution of the job can be computed using Lemma 4: $h_k(\ell)$.

Next, we compute $\mathcal{I}^{ST_{<0}}(\mathcal{C}_i)$ for $i \neq k$.

Lemma 5. *Total work done by a carry-in chain $\mathcal{C}_i$ ($i \neq k$), $\mathcal{I}^{ST_{<0}}_2(\mathcal{C}_i)$, is given by,*

$$\mathcal{I}^{ST_{<0}}_2(\mathcal{C}_i) = \begin{cases} \left\lfloor \frac{\ell + e^H_{k,|\mathcal{C}_k|}}{T_i} \right\rfloor \cdot E^H_i + h_i((\ell + e^H_{k,|\mathcal{C}_k|}) \% T_i); \\ \hfill \text{if } D_i \leq D_k \wedge S_i \geq e^H_{k,|\mathcal{C}_k|} \\ \left(\left\lfloor \frac{\ell - E^H_i}{T_i} \right\rfloor + 1 \right) \cdot E^H_i + h_i((\ell - E^H_i) \% T_i); \\ \hfill \text{if } D_i \leq D_k \wedge S_i < e^H_{k,|\mathcal{C}_k|} \\ h_i(\ell); \hfill \text{if } D_i > D_k \end{cases}$$

where $S_i = D_i - E^H_i$.

Proof. If $D_i \leq D_k$, the interference computation for $\mathcal{C}_i$ is the same as Lemma 4. For $D_i > D_k$, only non-preemptive blocks by $\mathcal{C}_i$ can occur in PW, which is upper bounded by $h_i(\ell)$. □

So, the carry-in interference of $\mathcal{C}_i$ in PW for $B_k < 0$ is computed as,

$$\mathcal{I}^{ST_{<0}}(\mathcal{C}_i) = \begin{cases} 0 & \text{if } \ i = k \\ \mathcal{I}^{ST_{<0}}_2(\mathcal{C}_i) & \text{if } \ i \neq k \end{cases} \tag{9}$$

Computing $\mathcal{I}^{ST_{\geq 0}}(\mathcal{C}_i)$.

Compute $\mathcal{I}^{ST_{\geq 0}}(\mathcal{C}_i)$ for $i = k$, which is given by:

$$\mathcal{I}^{ST_{\geq 0}}_1(\mathcal{C}_k) = \left\lfloor \frac{B_k}{T_k} \right\rfloor \cdot E^H_k + h_k(B_k \% T_k) \tag{10}$$

Now, we compute $\mathcal{I}^{ST_{\geq 0}}(\mathcal{C}_i)$ for $i \neq k$, which given by the following lemma.

Lemma 6. *Total work done by a carry-in chain $\mathcal{C}_i$ ($i \neq k$), $\mathcal{I}^{ST_{\geq 0}}_2(\mathcal{C}_i)$, is given by,*

$$\mathcal{I}^{ST_{\geq 0}}_2(\mathcal{C}_i) = \begin{cases} \left\lfloor \frac{B_k + D_k}{T_i} \right\rfloor \cdot E^H_i + h_i((B_k + D_k) \% T_i); \\ \hfill \text{if } D_i \leq D_k \wedge S_i \geq e^H_{k,|\mathcal{C}_k|} \\ \left(\left\lfloor \frac{\ell - E^H_i}{T_i} \right\rfloor + 1 \right) \cdot E^H_i + h_i((\ell - E^H_i) \% T_i); \\ \hfill \text{if } D_i \leq D_k \wedge S_i < e^H_{k,|\mathcal{C}_k|} \\ \left[\!\left[\left\lfloor \frac{B_k - 1}{T_i} \right\rfloor \cdot E^H_i \right]\!\right]^0 + \left[\!\left[e^H_{i,1} + \sum_{l=2}^{|\mathcal{C}_k|} (e^H_{i,l} - 1) \right]\!\right]_{S_k} \\ \quad + h_i((B_k - 1) \% T_i); \\ \hfill \text{if } D_i > D_k \wedge |\mathcal{C}_i| \geq |\mathcal{C}_k| \\ \left\lfloor \frac{\ell'}{T_i} \right\rfloor \cdot E^H_i + \left[\!\left[E^H_i - |\mathcal{C}_i| \right]\!\right]_{D_k - E^H_k} + h_i(\ell' \% T_i); \\ \hfill \text{if } D_i > D_k \wedge |\mathcal{C}_i| < |\mathcal{C}_k| \end{cases}$$

where $S_l = D_l - E^H_l$, $\forall \mathcal{C}_l$, and
$\ell' = \left[\!\left[\ell - \left(E^H_i + \sum_{j=|\mathcal{C}_k| - |\mathcal{C}_i| + 1}^{|\mathcal{C}_k| - 1} e^H_{k,j} - |\mathcal{C}_i| + 1 \right) \right]\!\right]^{B_k}$.

The interference computed by Lemma 6 is the same as Lemma 2 except for the carry-in job computed by Lemma 4.

So, the carry-in interference of $\mathcal{C}_i$ in PW for $B_k \geq 0$ is computed as,

$$\mathcal{I}^{ST_{\geq 0}}(\mathcal{C}_i) = \begin{cases} \mathcal{I}_1^{ST_{\geq 0}}(\mathcal{C}_i) & \text{if } i = k \\ \mathcal{I}_2^{ST_{\geq 0}}(\mathcal{C}_i) & \text{if } i \neq k \end{cases} \tag{11}$$

Theorem 2 (Schedulablity test for transient state)**.** *A set of* HI*-critical processing chains is schedulable in the transient state under FPP-GEDF-VD on m processors, if any chain $\mathcal{C}_k$ and for any B_k, both the conditions are satisfied:*

$$\sum_{\forall \mathcal{C}_i \in \Gamma^H} \mathcal{I}^{ST<0}(\mathcal{C}_i) \leq (\ell - h_k(\ell)) \cdot m, \qquad \text{if } B_k < 0$$

$$\sum_{\forall \mathcal{C}_i \in \Gamma^H} \mathcal{I}^{ST\geq 0}(\mathcal{C}_i) \leq (\ell - E_k^H + e_{k,|\mathcal{C}_k|}^H) \cdot m, \quad \text{if } B_k \geq 0$$

where $\ell = D_k + B_k - e_{k,|\mathcal{C}_k|}^H + 1$.

Bounding B_k. When $B_k < 0$: the mode-switch can occur as soon as the first callback's LO-WCET and as late as $\mathcal{C}_k$'s virtual deadline D_k^v, therefore $B_k \in [-D_k^v, -e_{k,1}^L]$. When $B_k \geq 0$: following lemma provides the bound:

Lemma 7. *For a set of processing chains with utilization $U(\Gamma^H) < m$, if the condition ($B_k \geq 0$) in Theorem 2 is to be violated for any B_k, then it is violated for some B_k that satisfies the condition below:*

$$B_k < \frac{2 \times \sum_{\forall i} E_i^H + m \cdot (E_k^H - e_{k,|\mathcal{C}_k|}^H)}{m - U(\Gamma^H, H)} - D_k + E_k^H - 1$$

Proof. Similar to Lemma 3. □

Theorem 3. *A set of mixed-criticality ROS 2 processing chains is schedulable under FPP-GEDF-VD on m processors if it is schedulable in both steady state (satisfying Theorem 1) and transient state (satisfying Theorem 2).*

Algorithm 2: Deadline shrinkage parameter x

```
Input: System workload Γ = {Γ^L, Γ^H} and precision acc ε
if Γ^H with {E_i^H, D_i} fails Theorem 1 then return FAIL;
  // unschedulable for steady-state HI-mode
δ ← 0.5; x ← δ ;                    // step size of search
while δ ≥ ε do
    δ ← δ/2 ;                        // updating step size
    for each τ_i ∈ Γ^H do  D_i^v ← x · D_i ;
    C_A = Check (Condition in Theorem 1 for Γ)
    C_B = Check (Conditions in Theorem 2 for Γ^H)
    if C_A ∧ C_B then  return x ;
    else if C_A ∧ ¬C_B then  x ← x − δ ;
    else if ¬C_A ∧ C_B then  x ← x + δ ;
    else return FAIL;                // no x can be found
return −1 ;                          // try with smaller ε
```

Obtaining Deadline Shrinkage Parameter. We use a common deadline shrinkage parameter x for HI-chains in LO-mode, and a binary search algorithm (provided in Algorithm 2) to find a suitable x for schedulability. The binary search algorithm returns either x for a schedulable taskset or 'failure' for an unschedulable taskset under FPP-GEDF-VD.

VI. Schedulability with Callback Groups

We now handle blocking times due to mutually exclusive callback groups under the global OMLP, which can be accounted for by methods similar to [14]. Such an accounting method allows *suspension-oblivious* schedulability analysis, as in Theorem 3, by inflating callback WCETs. The analysis in [14] assumes preemptive job-level fixed priority scheduling, which ensures that a waiting callback $c_{i,j}$ of a callback group $\mathcal{G}_{i,j}$ is delayed by a lock-holding callback for at most the duration of its execution. However, with non-preemptive callbacks, such a lock-holding callback's execution may be further delayed due to non-preemptive execution of a lower-priority callback not in $\mathcal{G}_{i,j}$. Therefore, while waiting on each lock-holding callback to finish, the execution of a waiting callback $c_{k,\ell}$ can be delayed by at most $\max_{c_{i,j}|\mathcal{G}_{i,j} \neq \mathcal{G}_{k,\ell}}\{e_{i,j}\}$ time (due to non-preemptive blocking) and at most $\max_{c_{i,j}|\mathcal{G}_{i,j} = \mathcal{G}_{k,\ell}}\{e_{i,j}\}$ to complete. Moreover, since mode changes can cause priority changes under FPP-GEDF-VD, the WCET inflation for a HI task requires consideration of blocking times both before and after the mode switch point. Considering the worst-case blocking time in both before and after the mode switch point, the blocking time of a mutually exclusive callback can be accounted for by inflating the callback's WCET according to the following theorem.

Theorem 4. *Under the global OMLP, blocking time incurred by a callback $c_{i,j}$ in a mutually exclusive callback group can be accounted for by inflating its* LO*-WCET and* HI*-WCET by $b_{i,j}^L$ and $b_{i,j}^H$, respectively, where*

$$b_{i,j}^L = (2m-1) \cdot \left(\max_{c_{k,\ell}|\mathcal{G}_{i,j} \neq \mathcal{G}_{k,\ell}} \{e_{k,\ell}^L\} + \max_{c_{k,\ell}|\mathcal{G}_{i,j} = \mathcal{G}_{k,\ell}} \{e_{k,\ell}^L\} \right),$$

$$b_{i,j}^H = m \cdot \left(\max_{c_{k,\ell}|\mathcal{G}_{i,j} \neq \mathcal{G}_{k,\ell}} \{e_{k,\ell}^L\} + \max_{c_{k,\ell}|\mathcal{G}_{i,j} = \mathcal{G}_{k,\ell}} \{e_{k,\ell}^L\} \right) + (2m-1) \cdot \left(\max_{c_{k,\ell}|\mathcal{G}_{i,j} \neq \mathcal{G}_{k,\ell}} \{e_{k,\ell}^H\} + \max_{c_{k,\ell}|\mathcal{G}_{i,j} = \mathcal{G}_{k,\ell}} \{e_{k,\ell}^H\} \right).$$

Proof. Provided in an Appendix available online [32]. □

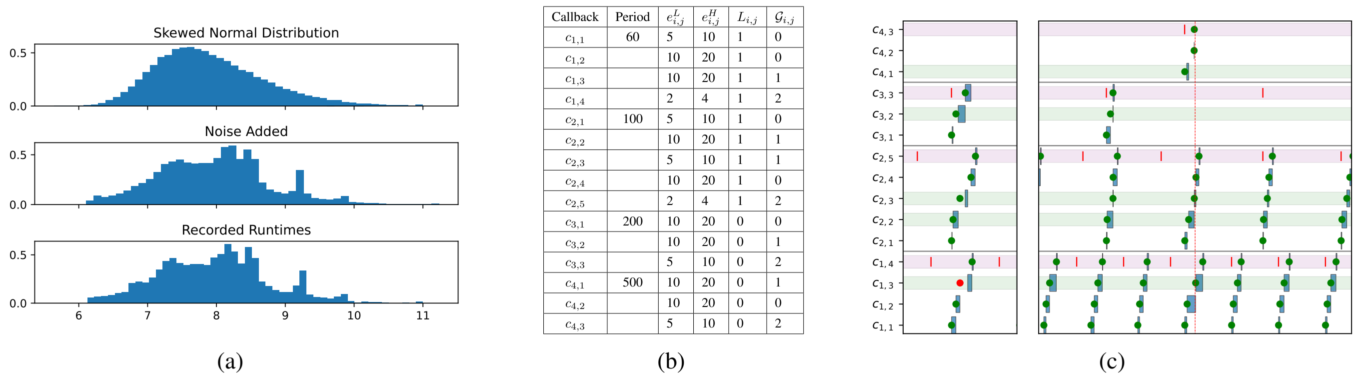
Using the above theorem, schedulability analysis of ROS 2 chains in presence of mutually exclusive callback groups can be done by replacing LO-WCET and HI-WCET of a callback $c_{i,j}$ with $e_{i,j}^L + b_{i,j}^L$ and $e_{i,j}^H + b_{i,j}^H$, respectively.

VII. Evaluation

A. Case Study

We implemented a dynamic-priority based MC scheduler (FPP-GEDF-VD), and OMLP locks for mutually exclusive callbacks on the ROS 2 Foxy [33] multi-threaded executor using the `readyQueue` as described in Sec. IV. We ran the case study on two cores of an Intel i7-14700, locked to 4 GHz.

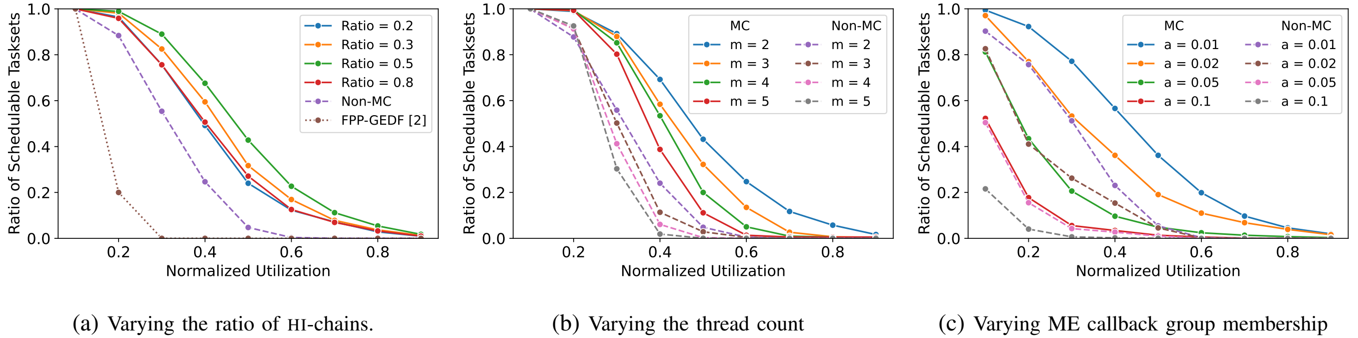
Emulating Task Behavior. [34] proposed that the runtimes of complex tasks can be modeled with distributions which can be used to estimate the worst-case runtime. To simulate the varying runtime of callbacks, we assign to each callback a skewed normal distribution. We add noise to each distribution



Callback	Period	$e^L_{i,j}$	$e^H_{i,j}$	$L_{i,j}$	$\mathcal{G}_{i,j}$
$c_{1,1}$	60	5	10	1	0
$c_{1,2}$		10	20	1	0
$c_{1,3}$		10	20	1	1
$c_{1,4}$		2	4	1	2
$c_{2,1}$	100	5	10	1	0
$c_{2,2}$		10	20	1	1
$c_{2,3}$		5	10	1	1
$c_{2,4}$		10	20	1	0
$c_{2,5}$		2	4	1	2
$c_{3,1}$	200	10	20	0	0
$c_{3,2}$		10	20	0	1
$c_{3,3}$		5	10	0	2
$c_{4,1}$	500	10	20	0	1
$c_{4,2}$		10	20	0	0
$c_{4,3}$		5	10	0	2

(a) (b) (c)

Fig. 9: **9a**: Illustration of the distribution, noisy variant, and corresponding task runtimes. Top: An example skewed normal distribution, centered around 7 ms with a skew shape of 3 and a scale of 1.2. Any time above the 99th percentile is clipped. Middle: The same distribution with noise added. Bottom: Recorded runtimes of a task. **9b**: Workloads used in the autonomous driving case study. All time values are in ms. We set the deadline shrink factor $x = 0.7$. **9c**: Two time lines from our autonomous driving-inspired case study. Each row represents a callback. Each chain is separated by gray lines. The callback executions are shown as blue blocks. The green and purple lanes indicate callbacks that are members of callback groups. The filled circles in each lane represent the release times of each task - a green circle indicates that the released job was placed in FQ, while a red circle indicates that a job was placed in PQ. The solid red lines represent the chain deadlines. The left side shows a time instant where a job of $c_{1,3}$ was added to PQ. This happened because $c_{2,3}$ and $c_{3,2}$ had been released shortly before. Since $c_{1,3}$ is in PQ, it has to wait until $c_{2,3}$ and $c_{3,2}$ leave FQ before it can be run, even though it has a shorter deadline than $c_{1,3}$. The right side shows a time instant where $c_{1,2}$ exceeded $e^L_{1,2}$ and caused a mode switch, marked with a dashed red line.



(a) Varying the ratio of HI-chains. (b) Varying the thread count (c) Varying ME callback group membership

Fig. 10: Schedulability ratios of groups of tasksets with varied parameters. Each point in the graphs is the percentage of schedulable tasksets out of 10000. In (a), we compare our (steady-state) schedulability test with existing analysis of the ROS 2 executor with EDF [7].

to account for possible execution paths of each callback that are taken in response to non-deterministic factors such as sensor readings or probabilistic algorithms. Fig. 9a shows how we start with a skewed normal distribution (top) and add noise (middle). We ran the same distribution with the dummy workload (bottom) to validate that the dummy task system behaves as expected. Each callback has its own distribution.

Case Study with Mixed Criticality and OMLP. This task set is inspired by an autonomous driving workload run on an F1Tenth car. There are two shared resources that require mutually exclusive callback groups: a GPU and a serial port. The first chain is the main driving chain. The first callback polls a LIDAR sensor, the second pre-processes the LIDAR data, the third performs some kind of model inference on a GPU, and the fourth callback passes control decisions over serial output. The second chain serves a similar purpose, but retrieves images from a camera, and over two callbacks, processes the images with a CNN using the GPU. It also outputs results over the shared serial port. The last two chains represent an arbitrary workload that also uses the GPU and serial port. All callbacks that use the GPU are placed in callback group 1, and all callbacks that use the serial port are in callback group 2.

The taskset parameters are given in Table 9b. For each callback, we created distributions that result in the majority of callbacks running under their respective $e^L_{i,j}$'s. Each distribution is skewed, so some callbacks will run longer than $e^L_{i,j}$, but we prevent any callback from running longer than $e^H_{i,j}$.

We recorded the time between the release of a callback and when the callback begins executing. As expected, callbacks that are members of the ME callback groups had much longer gaps due to the blocking time incurred by OMLP. For tasks not part of any callback group, the longest delay was 0.22 ms, in $c_{2,2}$. In group 1, $c_{4,2}$ had the longest blocking time of 14.87 ms, and in group 2, $c_{1,4}$ was blocked for 2.00 ms.

The average overrun detection latency (the time between

a task exceeding its LO execution time and the watchdog detecting the overrun) was 78 μs. Except for one overrun which took 6ms to detect, the longest overrun detection latency was 752 μs. The very small average detection latency can be explained by the Linux's handling of sem_timedwait. The hardware interrupt scheduled by sem_timedwait (which is allowed to expire when an overrun occurs) triggers a reschedule. Since the watchdog threads have a higher priority than the executor threads, the kernel switches to the watchdog thread as soon as the interrupt ends. Beside some potential non-preemptable kernel work (disk IO, networking - which may explain the outliers), the major contributor to overhead to overrun detection is rescheduling and context switching.

B. Schedulability Evaluation

Workloads. For each taskset and scheduler configuration in the schedulability test, we randomly generate and evaluate 10000 tasksets. Each taskset has up to 10 chains, and each chain can have up to 10 callbacks. Except for in Fig. 10b, each taskset is evaluated with 2 threads. We used UUnifast [35] to distribute utilization across chains and callbacks. Each chain has a randomly selected period between 50 and 200. The stated utilization for each taskset is the utilization of the taskset (normalized by m) as if it were run by a non-MC scheduler. In Fig. 10a, we changed the ratio of HI- and LO-chains, and compare them with non-MC tasksets. In Fig. 10b, we varied the number of threads (m), and kept the ratio of HI- and LO-chains at 0.5. In Fig. 10c, we consider the effect of OMLP locks on callback groups. a is the ratio of tasks added to any callback group. We randomly place those tasks into one of two callback groups.

Observations. When we generated tasksets with different ratios of HI- and LO-chains, and tasksets without MC features, tasksets with an even balance of HI- and LO-chains performed best, while tasksets tested without MC performed the worst.

When adding more threads and keeping the normalized utilization the same, the tasksets run under the MC test performed best, although increasing the threads for the same normalized utilization had a negative effect on schedulability.

Introducing callback groups into the tasksets reduced the schedulability of tasksets in both the MC and non-MC schedulers, but the MC schedulability test accepted more tasksets.

VIII. Conclusion

This work presents the first effort to enable mixed-criticality scheduling and handle ME callback groups in ROS 2 by modifying its executor design. Enabling mode switch mechanisms can provide huge design flexibility for future systems using ROS 2. We provide a detailed system implementation of a proposed virtual deadline-based global scheduler (FPP-GEDF-VD) and OMLP locking protocol, corresponding with schedulability analysis with bounded blocking time. We evaluated the proposed approach via synthetic workloads and a case study simulating an autonomous driving workload. Combining this technique with a preemptive executor and callback-level techniques to support aborting during a critical section in future work will allow us to deploy mixed criticality with shared resources to a real system.

Acknowledgment

We are grateful to anonymous reviewers for their valuable feedback. We thank James H. Anderson from the University of North Carolina at Chapel Hill for providing feedback on a preliminary draft of the paper. This work was supported in part by the National Science Foundation under Grant CMMI 2246672.

References

[1] T. Blaß, D. Casini, S. Bozhko, and B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *42nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 41–53.

[2] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. Brandenburg, "Automatic latency management for ROS 2: Benefits, challenges, and open problems," in *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 264–277.

[3] 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 (ECRTS)*. Schloss Dagstuhl, 2019, pp. 1–23.

[4] 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 *41st IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 231–243.

[5] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-to-end timing analysis in ROS2," in *43rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 53–65.

[6] H. Teper, D. Kuhse, M. Günzel, G. von der Brüggen, F. Howar, and J.-J. Chen, "Thread carefully: Preventing starvation in the ROS 2 multithreaded executor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3588–3599, 2024.

[7] A. Al Arafat, K. Wilson, K. Yang, and Z. Guo, "Dynamic Priority Scheduling of Multithreaded ROS 2 Executor With Shared Resources," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3732–3743, 2024.

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

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

[10] H. Sobhani, H. Choi, and H. Kim, "Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors," in *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 106–118.

[11] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE international real-time systems symposium (RTSS)*. IEEE, 2007, pp. 239–243.

[12] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–37, 2017.

[13] K. Wilson, A. Al Arafat, J. Baugh, R. Yu, and Z. Guo, "Physics-informed mixed-criticality scheduling for f1tenth cars with preemptable ros 2 executors," in *31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2025, pp. 215–227.

[14] B. Brandenburg and J. Anderson, "Optimality results for multiprocessor real-time locking," in *31st IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2010, pp. 49–60.

[15] H. Teper, T. Betz, M. Günzel, D. Ebner, G. Von Der Brüggen, J. Betz, and J.-J. Chen, "End-to-end timing analysis and optimization of multi-executor ros 2 systems," in *30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024, pp. 212–224.

[16] eProsima, "eProsima fast DDS," https://github.com/eProsima/Fast-DDS. [Online]. Available: https://github.com/eProsima/Fast-DDS

[17] Eclipse Foundation, "Eclipse cyclone DDS™," https://projects.eclipse.org/projects/iot.cyclonedds. [Online]. Available: https://projects.eclipse.org/projects/iot.cyclonedds
[18] GurumNetworks, "GurumDDS," https://gurum.cc/gurumdds_rmw_eng. [Online]. Available: https://gurum.cc/gurumdds_rmw_eng
[19] Y. Tang, N. Guan, X. Jiang, X. Luo, and W. Yi, "Real-time performance analysis of processing systems on ROS 2 executors," in *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 80–92.
[20] X. Jiang, D. Ji, N. Guan, R. Li, Y. Tang, and Y. Wang, "Real-time scheduling and analysis of processing chains on multi-threaded executor in ROS 2," in *43rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 27–39.
[21] Y. Suzuki, T. Azumi, S. Kato, and N. Nishio, "Real-time ROS extension on transparent CPU/GPU coordination mechanism," in *21st IEEE International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2018, pp. 184–192.
[22] 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 *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 93–105.
[23] R. Li, N. Guan, X. Jiang, Z. Guo, Z. Dong, and M. Lv, "Worst-case time disparity analysis of message synchronization in ROS," in *43rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 40–52.
[24] J. Sun, T. Wang, Y. Li, N. Guan, Z. Guo, and G. Tan, "Seam: An optimal message synchronizer in ros with well-bounded time disparity," in *44th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 172–184.
[25] K. Wilson, A. A. Arafat, J. Baugh, R. Yu, X. Liu, and Z. Guo, "Soteria: A formal digital-twin-enabled framework for safety-assurance of latency-aware cyber-physical systems," in *Proceedings of the 28th ACM International Conference on Hybrid Systems: Computation and Control*, 2025, pp. 1–11.
[26] S. Liu, R. Wagle, S. Ahmed, Z. Tong, and J. H. Anderson, "ROS^{RT}: Enabling flexible scheduling in ros 2," in *46th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2025, pp. 42–54.
[27] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time system synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.
[28] Z. Tong, S. Ahmed, and J. Anderson, "Overrun-resilient multiprocessor real-time locking," in *34th Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022, pp. 10–1.
[29] T. Johnson and K. Harathi, "Interruptible critical sections," *Computer Systems Science And Engineering*, vol. 15, no. 4, pp. 241–252, 2000.
[30] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *28th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2007, pp. 119–128.
[31] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *29th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2008, pp. 137–146.
[32] A. Al Arafat, K. Wilson, S. Ahmed, and Z. Guo, "Supporting mixed-criticality and mutually exclusive callback groups in multi-thread ROS 2," https://abdullahaarafat.github.io//files/rtas2026.pdf, 2026.
[33] "ROS 2 Documentation," https://docs.ros.org/en/foxy/index.html.
[34] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 91–101.
[35] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, 2005.