applied sciences

MDPI

Article

A New Algorithm for Real-Time Scheduling and Resource Mapping for Robot Operating Systems (ROS)

Khaled Chaaban

College of Computer and Information Systems, Umm Al-Qura University, Makkah 21955, Saudi Arabia; aochaaban@uqu.edu.sa

Featured Application: Real-Time Systems: Architectures, Software and Applications.

Abstract: A new version of a robot operating system (ROS-2) has been developed to address the real-time and fault constraints of distributed robotics applications. However, current implementations lack strong real-time scheduling and the optimization of response time for various tasks and applications. This may lead to inconsistent system behavior and may affect system performance. This article presents a new and efficient heuristic scheduling algorithm for the ROS-2 framework to improve resource mapping and system scheduling for robotics applications. The proposed scheduling design includes grouping callbacks into executors, assigning executor priority, and sequencing callbacks inside executors. The timing constraints and functional properties are expressed in the formal design model, and real-time scheduling is performed with respect to the timing constraints. The benefits of the proposed solution are demonstrated by a set of experimental results. Using this algorithm, it is shown that the number of executors needed to schedule a set of callbacks is minimized and the system schedulability is maximized for loaded and overloaded cases.

Keywords: embedded real-time systems; real-time scheduling; optimization; ROS (robot operating system)

Citation: Chaaban, K. A New Algorithm for Real-Time Scheduling and Resource Mapping for Robot Operating Systems (ROS). *Appl. Sci.* **2023**, *13*, 1532. https://doi.org/10.3390/app13031532

Academic Editors: Jinchao Chen and Chao Chen

Received: 20 December 2022
Revised: 17 January 2023
Accepted: 20 January 2023
Published: 24 January 2023

1. Introduction

The robot operating system (ROS) offers a service-oriented solution for the development of distributed robotic systems. It enables the integration of open-source libraries and tools by standardizing the methodology of software design and the underlying communication layer [1,2].

Most robotic systems are classified as real-time and safety-critical. The predictable execution of the timing under real-time constraints is an important key design for such complex systems. ROS focuses on the fast and functional integration of different applications, while execution control and timing constraints are not adequately addressed by the middleware [3]. Despite the effort in the new ROS-2 distribution to address the support of real-time and fault-tolerance mechanisms [4], there is no clear definition of callback execution order within an executor and there is no control over the execution order between the different nodes. The precedence of execution and the dependencies between different nodes and callbacks cannot be clearly defined, and there are no mechanisms to define the real-time scheduling of running entities [5].

Research on the end-to-end real-time analysis of ROS-2 embedded applications has started recently. Casini et al. [6] proposed a pioneering analysis of end-to-end timing chains in ROS-2. Subsequently, several studies addressed the real-time behavior of the ROS-2 execution model and its performance and security capabilities [3,7]. New scheduling techniques were proposed to improve the conventional ROS-2 scheduler by proposing an alternate scheduler as in [8] or keeping it unmodified as in [9].

All these previous studies consider the number of callbacks and executors to be predefined and assume that the callbacks are already assigned to executors. They then propose solutions for the real-time scheduling of the application. To schedule an embedded application in ROS, we need to map callbacks to an executor in the context of which callbacks are executed on the processor. ROS does not impose any requirements on this step of configuration. The execution model is based on the processes that run the executors to receive messages and invoke callbacks [4]. For large systems, manually setting up an execution order of callbacks inside the executor, as well as assigning priorities to executors, is always error-prone and difficult to predict. In critical safety systems, static memory allocation is crucial. In the initialization phase, the system designer determines the number of callbacks for the embedded application and the necessary dynamic memory to be allocated before the system execution. Then, the system designer maps the callbacks to the executors. The order of callbacks within an executor is important because it may affect the real-time execution and performance of the embedded system in real-time. During the configuration step, the system designer shall define the number of callbacks and their sequence within the executor.

The simplest approach is to bind each callback to one executor. However, mapping each callback to a dedicated executor leads to a lot of context switches and may lead to a deadline violation and significant OS overhead. Similarly, mapping all callbacks of a node to a single executor may leave the system unresponsive if the executor fails. In addition, the internal branching structure of such a process becomes complex to schedule. Having these two extremes in mind, finding a middle solution is an optimization goal. Our problem consists of finding a partitioning of the callbacks set that is schedulable with a minimum number of executors. The number of partitions to be explored for n callbacks is the Bell number [10]. The Bell number is exponential with respect to the number of callbacks n and can be computed by the following recurrence relation:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \quad with\ B_0 = 1 \tag{1}$$

For example, to search for a space of 500 callbacks, we obtain $B_{500} \approx 10,844$. If we group callbacks that have identical periods, the search space can be reduced to $\Pi_{i=0}^{m}(B_{n_i})$, where B_{n_i} is the Bell number of the set i of n callbacks with equal periods and m is the number of sets. However, this number remains exponential. Given the complexity of the problem, we need to use heuristics to reduce the search space. A simple heuristic is to regroup callbacks with the same period (or integer multiples) to be assigned to one executor. Several studies have considered the software clustering problem in the context of real-time systems. Common optimization goals are stack memory usage [11,12] using genetic algorithms and scheduling algorithms, core load balancing [11], inter-communication overhead [13,14] using integer linear programming (ILP), data consistency using ILP and heuristic methods [15], and deadline violation [16,17] using genetic algorithms and the greedy heuristic approach for system resource clustering. Other work proposes regrouping runnables with arbitrary periods on the same task using a greedy heuristic approach combined with linear cost function and schedulability test [18,19]. This approach may lead to a minimum number of executors needed to schedule overloaded systems while minimizing the stack memory usage and context switch overhead. This paper adopts this approach and adapts it to ROS-2 applications.

Contribution: This work adopts a greedy heuristic approach to quickly construct a correct solution to the mapping and resource configuration problem of the ROS embedded system. The heuristic itself is not the central work of this paper. The main contribution is the design of the cost function and the associated schedulability test, which efficiently reduce the clustering complexity. The proposed algorithm assigns a set of callbacks to the executors while preserving the schedulability of the real-time system and simultaneously determines the number of executors required to schedule a set of callbacks, the priority assignment of the executor, and the sequence of callbacks within an executor (Figure 1).

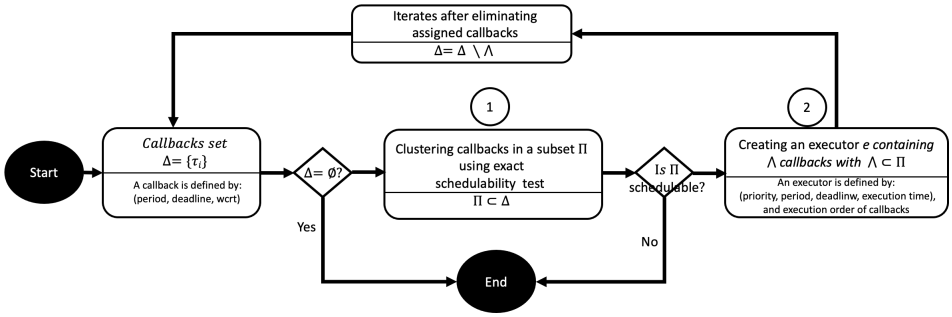

Figure 1. Main steps of the proposed approach.

First, the two-stage approach clusters callbacks using an exact schedulability test before building an executor with priority assignment. The results of this work are as follows:

- A formal model to analyze the system.
- A low-complexity heuristic algorithm to build executors from a set of callbacks.
- A method to assign priority to executors and to define the order of execution of callbacks inside an executor.

Organization: This article is organized as follows: Section 2 reviews the relevant related works. Section 3 introduces the ROS-2 architecture, its main concepts, and the scheduling mechanisms. The formulation of the system model is provided in Section 4. Section 5 presents the new algorithm for real-time scheduling and resource mapping for ROS-2. Section 6 discusses the results of the experiments obtained. Section 7 concludes the work and identifies potential perspectives.

2. Related Works

Real-time scheduling and resource allocation have been widely addressed by the real-time systems community, such as [20–22]. In the particular context of ROS embedded systems, most of the research work focuses on the integration of real-time scheduling capabilities into the conventional scheduler of the ROS core system [6,8,23,24]. The authors of [6] have made a significant contribution to the field by proposing a formal model with analytical methods for the analysis of the end-to-end response time of ROS embedded systems. It is the first attempt to analyze the response time of ROS-2 processing chains. In subsequent work, [24] gives a more detailed analysis of end-to-end paths with the assumption that a callback may belong to one chain. In [8], the authors suggested an alternative ROS scheduler to reduce end-to-end latency. They consider both system scheduling techniques and the mapping functions of callbacks to executors for monocore and multicore systems. Their algorithm provides callback priority assignments to improve real-time predictability. However, the proposed solution assumes that the set of executors is defined a priori. The work in [23] proposes a method to prioritize message transmission between ROS nodes with harmonic and non-harmonic periods. The results reveal a reduced response time and low variance between nodes.

Furthermore, several studies have evaluated ROS-2 performance using measurement-based approaches instead of analytical methods, such as [3,5]. They mainly conduct empirical performance measures to propose improvement directions. Security aspects are addressed by [25–28]. In [27], the authors investigated the security plug-ins in ROS-2 to study its ability to mitigate multiple threats. The study considers the impact of both QoS and security on ROS-2 network performance using measurement-based results. In [28], the authors use a probabilistic model checker to validate the real-time performance and reliability of ROS-2. Their method reduces the data loss rate and the response time of the system without considering real-time scheduling. In [25], the authors propose an approach to integrate reliability and QoS services into ROS-2 using DDS middleware services. In [26], Krichen et al. adopted a model-based security testing (MBST) approach to check the security of IoT applications in the context of smart cities. The proposed approach uses

extended timed automata to model security and behavioral requirements, to design and generate security tests, and to execute and evaluate the security tests. The model-based nature of the proposed approach makes it applicable to other real-time systems such as ROS-2.

Other works take advantage of optimization methods to address the NP-hard problem of software resource allocation and configuration. These works adopted another perspective using optimization methods [11–13,19]. In [11], Thomas Wilhelm et al. propose methods to optimize the mapping of runnables to tasks in multicore automotive control units. First, a constraint programming method is used to automate the generation of initial configurations while balancing core utilization. Second, an evolutionary algorithm is used to optimize the mapping of runnables to tasks. Task creation is conducted using a simple heuristic of mapping runnables with the same period (or integer multiple) to the same task. The optimization steps are performed in a serial manner, so the process may become stuck in local minima. In [13], the authors address the problem of mapping tasks in multicore architecture using rate monotonic scheduling (RMS). The solution regroups tasks with the same period on the same core; then, a heuristic method is proposed to minimize the intercommunication overhead between the core and task response time. The authors of [12] address the issues of AUTOSAR model design analysis to minimize memory stack utilization for mixed-criticality systems with preemptive threshold setting that addresses the design synthesis problem. In a prior work [19], the authors developed a heuristic algorithm to configure task mapping for AUTOSAR-based embedded systems. The main goal is to optimize the operating system and task mapping to improve system performance. Their method constructs a set of tasks from a set of runnables with arbitrary periods while guaranteeing real-time requirements.

The problem of grouping/clustering has recently been addressed by several studies with an application to the planning of autonomous robot paths [29–31]. Metaheuristic algorithms are proposed to find optimal collision-free paths between two points for mobile robots. The low complexity of the proposed methods and their efficiency of resource usage have been approved for large-scale and crowded farmlands. Metaheuristic algorithms are suitable for multimodal, multi-objective systems. However, for large systems with multiple objectives, these methods suffer from high time complexity and the local minima trap. Heuristic methods quickly provide a solution to a difficult optimization problem. For the particular resource optimization of this study, we opted for the heuristic method to solve a scheduling problem. Combining heuristic with metaheuristic methods to solve a holistic optimization problem seems to be promising for large-scale systems.

Table 1 summarizes related works by classifying them into three main classes: analytical, heuristic, and metaheuristic methods. The online/offline variable indicates that the developed method runs during a configuration step or at runtime. Some methods are integrated into the ROS-2 core scheduler (Built-in). The set of decision variables and cost objectives is listed for the optimization problem.

To the best of our knowledge, this is the first work that applies resource mapping and scheduling with arbitrary periods to construct and schedule a set of executors given a set of callbacks for an ROS embedded system.

Table 1. Summary of related works

Method Class	References	Online/Offline	Built-in ROS-2	Variables	Optimization Goals
Formal Analysis	[15,20–22]	Offline	No	priority	latency-success ratio, data consistency
	[6,23,24]	Offline	Yes	priority	latency
	[8]	Online	Yes	priority, mapping	latency, core balancing
	[26]	Offline	N/A	QoS profiles, threat generation	loss rate, coverage, latency
Measurement-based	[3,7,27]	Offline	No	QoS policy, security	loss rate, latency, throughput
Heuristic	[28]	Offline	No	priority, mapping	real-time performance, reliability
	[11,19]	Offline	No	mapping, priority, sequencing	real-time performance, reliability
	[12,19]	Offline	No	mapping, scheduling	success ratio, latency
	[13]	Offline	No	mapping, scheduling	inter-cores communication overhead, latency
Metaheuristic	[29–31]	Offline	No	clustering	path cost
	[11]	Offline	No	mapping, core balancing	resp. time analysis

3. Background of ROS-2 Architecture

An embedded application in ROS consists of a set of interconnected nodes (Figure 2). Each node contains one or more callbacks, which are the smallest pieces of code of the application. To ensure the consistent execution of programs written with different specific languages and different underlying platforms, a set of libraries provides the required abstraction at different layers. rmw is the middleware layer that ensures the interface between the data distribution service (dds) [32] layer and the robot client library (rcl) layer. The rcl layer provides the abstraction of application programming interfaces (APIs) to the application to ensure consistent execution between the rmw layer and specific language client libraries such as Python (rclpy) and C++ (rcpcpp) [33]. dds is a vendor-specific implementation of rmw that standardizes real-time communication between nodes.

The nodes communicate using a publisher/subscriber communication paradigm or a client/server communication paradigm. The publisher/subscriber communication paradigm is suitable for data flow streaming and sensor nodes, while the client/server communication paradigm is more suitable for synchronous remote procedural call and control nodes. Nodes use topics to communicate between publishers and subscribers, and use services to communicate between clients and servers. Topics and services are identified by a unique name, and one node can publish data on several topics and may also subscribe to several topics. A server may have multiple clients, but a client must have one server.

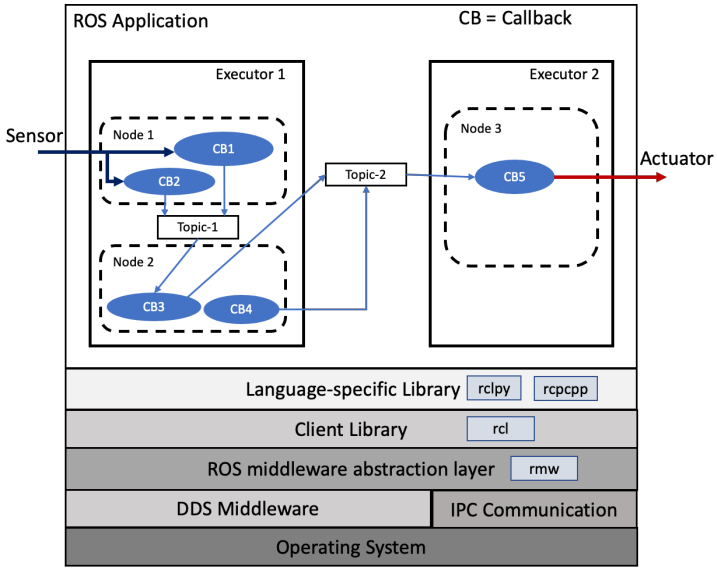

Figure 2. Main concepts of ROS-2.

Scheduling in ROS-2

This section gives a brief description of the ROS scheduling mechanism and scheduling-related artifacts. The ROS scheduling model is implemented using executors. An executor selects pending callbacks from the ready list to be executed. Basically, two implementations are defined in ROS: a single executor and a multithreaded executor. In this study, we consider a single-threaded executor.

Callback This is the smallest executable code scheduled in ROS-2. ROS-2 defines five types of callbacks: timers that are time-triggered by system-level timers, subscribers that are triggered by the reception of new messages at a topic, services, and clients callbacks that are triggered by service requests and responses, respectively.

Node The node is the minimal self-contained unit of behavior. A node contains a set of callbacks organized by application programmers for the modularity and logical partitioning of functions. All callbacks from the same node are executed by the same executor.

Executor The executor reads the incoming messages from a ready list and executes, in a non-preemptive manner, the corresponding callback according to their type and activation instant in this order: timer callbacks first, then subscriber callbacks, service callbacks, and, finally, client callbacks, as illustrated in Figure 3. Unlike common real-time priority-based scheduling algorithms, the executor does not always execute callbacks in their activation instances. Instead, the executor updates the ready status of non-timer callbacks in their respective queues when all queues are empty (called a polling point), and such a delayed update of callback readiness makes the priority assignment of non-timer callbacks ineffective and lets chains run in a round-robin-like manner.

Although the conventional scheduling semantic of ROS-2 does its job well for most applications, there are some limitations that are not compatible with safety-critical embedded systems, such as well-defined execution order, priority inversion, and the high and complex scheduling overhead in terms of CPU and memory usage.

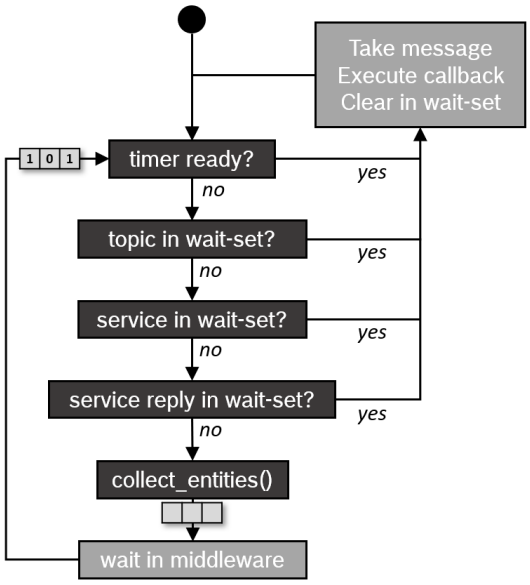

Figure 3. Scheduling semantics in ROS-2 (source: ros.org).

4. System Model

We assume a periodic, fixed priority with constrained deadlines to represent real-time embedded systems. A steady-state system is considered, i.e. callbacks are defined at the initialization phase, and the configuration is still static during run-time since we consider safety-critical systems in the context of this work. The response time of a callback is the duration from the release instant to the completion of the job execution. Each callback must have a relative deadline. If the worst-case response time of the callback is shorter than or equal to its relative deadline, we say that the callback is schedulable. In the following, some useful definitions are given, which are used later in this article.

Definition 1 (GCD). *The greatest common divisor (GCD) of two positive integers (a, b), denoted by GCD(a,b), is defined as the greatest positive number that is a common factor of both positive integers (a, b). Example: GCD(3,4) = 1.*

Definition 2 (LCM). *The least common multiple (LCM) of two positive integers (a, b), denoted by LCM(a,b), is defined as the smallest positive integer that is divisible by both a and b. Example: LCM(3,4) = 12.*

Let us denote by Δ, a set of m callbacks $\Delta = \{\tau_0, \tau_1, \ldots \tau_m\}$. A callback τ_i is the smallest piece of code that runs in the context of an executor, denoted e_j. A callback τ_i can be formalized as follows:

$$\tau_i(c_i, p_i, d_i, k_i) \tag{2}$$

- c_i: The worst-case execution time (WCET).
- p_i: The callback period.
- d_i: The deadline for a callback.
- k_i: A positive integer that defines the order of callback execution within the executor.

Let us denote by ξ a set of n executors $\xi = \{e_0, e_1, \ldots, e_n\}$. To model an executor, a multi-frame task model [34] is adopted where the execution of an executor e_j consists of a sequence of N frames characterized as follows:

$$(\vec{E}_j, P_j, D_j, T_j) \tag{3}$$

- D_j: The executor deadline.
- P_j: The executor priority.

- T_j: The executor period.
- $\vec{E_j} = [C_{j,0}, C_{j,1}, \ldots, C_{j,s}, \ldots, C_{j,N-1}]$: A vector of N_j components. Each component $C_{j,i}$ represents the execution time of e_j in frame j.

The execution time $C_{j,s}$ of the s^{th} frame should be less than the executor period T_j.

The period T_j of an executor e_j that contains m callbacks can be determined as the greatest common divisor (GCD) of the periods:

$$T_j = \underset{1 \le i \le m}{GCD}\{p_i\}. \tag{4}$$

The frames of the executor e_j periodically repeat with a major cycle of π_j equal to the least common divisor (LCM) of the callback periods:

$$\pi_j = \underset{1 \le i \le m}{LCM}\{p_i\}. \tag{5}$$

The number of frames N_j can be determined by dividing the major cycle by the executor period as follows: $N_j = \pi_j / T_j$.

Each execution time $C_{j,s}$ of the s^{th} frame $(s = 0, 1, \ldots, N_{j-1})$ is given by the following equation:

$$C_{j,s} = \sum_{i=1}^{n} h_i(s) * c_i \tag{6}$$

where the function $h_i(s)$ indicates that the i^{th} callback is present or not in the s^{th} frame as follows:

$$h_i(s) = \begin{cases} 1 & if\ f_i(s) = s \\ 0 & if\ f_i(s) \neq s \end{cases} \tag{7}$$

with

$$f_i(s) = \left\lfloor \frac{s}{\gamma_i} \right\rfloor \gamma_i \tag{8}$$

$f_i(s)$ computes a location for callback τ_i in frame s ($f_i(s) \le s$). $\gamma_i = p_i / T_j$ represents the repetition factor of τ_i in e_j. $((n\gamma_i + \delta_i)\ mod\ N)$ indicates the position of τ_i. Table 2 summarizes the main notation of the system model.

Table 2. Main notations used in this study.

Symbol	Description
τ	Callback
c	Worst-case execution time (wcrt) of callback
p	Callback period
d	Callback deadline
k	Execution order of callback within executor
e	Executor
$\overline{E_j}$	Maximum execution time of executor
T	Executor period
D	Executor deadline
P	Executor priority
π	Major cycle of the executor
GCD	Greatest common divider
LCM	Least common multiple

Hereafter, an example with the callbacks set '$\{\tau_1(1, 10, 8, 1), \tau_2(1, 15, 10, 2), \tau_3(1, 15, 12, 3)$ and $\tau_4(1, 30, 19, 4)\}$' is assigned to one executor e_1.

Table 3 illustrates an example of how to compute the parameters of the executor model, and Figure 4 shows a visual representation of the executor and its callbacks.

Table 3. An example of computing executor model parameters.

s^{th} Frame	$f_i(s)$				$h_i(s)$				$C_{1,s}$
	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4	
0	0	1	0	5	1	0	1	0	2
1	0	1	0	5	0	1	0	0	1
2	2	1	0	5	1	0	0	0	1
3	2	4	3	5	0	0	1	0	1
4	4	4	3	5	1	1	0	0	2
5	4	4	3	5	0	0	0	1	1

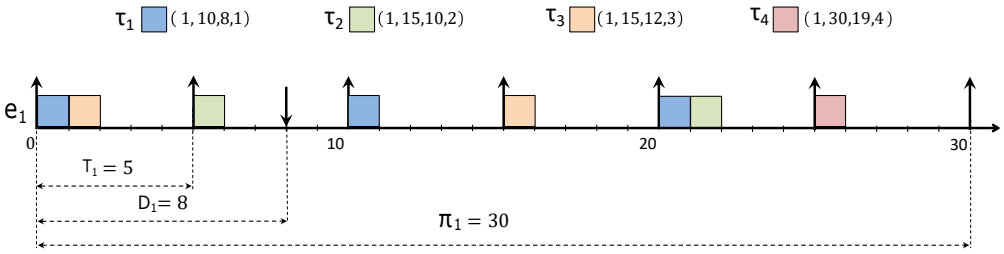

Figure 4. Example of assigning callbacks to the executor e_1. $\pi_1 = LCM(10,15,15,30) = 30$. $T_1 = GCD(10,15,15,30) = 5$. $D_1 = \min\{8,10,12,19\} = 8$. *Number of frames* $N = 30/5 = 6$.

5. Heuristic Scheduling Algorithm

We try to assign a callback τ_i from a set Δ to a unique executor e_i of a set ξ. The problem consists of finding a feasible assignment using a minimal number of executors while meeting the timing requirements. The system is declared schedulable if and only if any executor e_j of ξ is schedulable and satisfies the deadline constraint.

To achieve this, we propose the following algorithm that assigns n callbacks from Δ to m executors of ξ (Algorithm 1).

Algorithm 1: Assigning callbacks to executors.

Input: $\Delta = \{\tau_1, \tau_2, \ldots, \tau_n\}$
Output: $\xi = \{e_1, e_2, \ldots, e_m\}$

```
1  ξ = ∅ ;                                                    //
2  j = 1 ;                          // Starting from the lowest priority
3  while Δ ≠ ∅ do
4      Π_j = Assignment(Δ) ;                                  // Algo. 2
5      if Π_j ≠ ∅ then
6          [Λ_j, e_j] = CreateExecutor(Π_j) ;                 // Algo. 3
7          ξ = ξ ∪ {e_j} ;                                    //
8          Δ = Δ\Λ_j ;                                  // Remove Λ_j from Δ
9          j = j + 1;
10     else
11         ξ = ∅ ;                                  // Set empty output
12         break ;                          // System is not scdedulable!
13     end
14 end
```

To establish the output list of executors ξ, the algorithm invokes two other algorithms in its main loop. The first algorithm, 'Assignment' (Algorithm 2), establishes a subset of callbacks Π_j to be assigned to the same executor e_j by starting from the lowest prior-

ity level (j = 1). In the case of not empty Π_j, a second algorithm, called 'CreateExecutor' (Algorithm 3), is called to define the subset of callbacks Λ_i from Π_j to be assigned to the executor e_j and determine its parameters. By iteration, the set of executors $\xi = \{_1, \ldots, _j, \ldots, _m\}$ is constructed until all callbacks ($\Delta = \emptyset$) are assigned to the executors. If, in any iteration, Algorithm 2 does not find a feasible subset Π_j, then the entire system is declared unschedulable. In the following, we detail Algorithms 2 and 3 used by the main Algorithm 1.

5.1. Algorithm 2: Assignment

This section details Algorithm 2, the objective of which is to identify a subset of callbacks that could be regrouped with one executor while satisfying real-time requirements.

Theorem 1. *Suppose a set Δ of callbacks and an executor e_j. There is a subset $\Pi_j \subseteq \Delta$ that can be assigned to the same executor e_j if and only if each element τ_i of Π_j is schedulable when assigned a lower priority and other elementsΔ -$\tau_i\}$ are assigned higher priorities.*

Proof. The proof can be obtained by contradiction. Let us suppose two callbacks τ_1 and τ_2 with d_1 and d_2 as deadlines, respectively. Suppose that the set of callbacks $\{\tau_1, \tau_2\}$ is feasible with the lowest priority. Two distinct cases may appear: (1) τ_1 has a higher priority than τ_2, or (2) τ_2 has a higher priority than τ_1. The response times of τ_1 and τ_2 when assigned the lowest priority are identical for the two cases. This means that τ_1 and τ_2 must always complete the execution before the shortest deadline. Therefore, the set of callbacks τ_1, $\tau_2\}$ can be assigned to an executor with a deadline D =min$\{d_1, d_2\}$. Consequently, if a set of callbacks is feasible when all callbacks are assigned the lowest priority, then this set can be assigned to only one executor. On the other hand, if $\{\tau_1, \tau_2\}$ is not schedulable with the lowest priority, there is no priority assignment that allows scheduling callbacks. Therefore, if a set of callbacks can be assigned to one executor, each callback is feasible when assigned to the lowest priority. □

As stated in the above theorem, the subset Π_j can be obtained if each callback of Δ satisfies the following equation:

$$R_i = c_i + \sum_{k \in \Delta - \{\tau_i\}} \left\lceil \frac{R_i}{p_k} \right\rceil c_k \leq d_i \tag{9}$$

R_i is the worst-case response time of callback τ_i when assigned the lowest priority and all callbacks are released at the same time (critical instant). p_k and c_k are, respectively, the period and the worst-case execution time for the k^{th} callback. This results in the longest response time to callback [21]. As all callbacks have the same wcrt when assigned to the lowest priority, R_i can be simplified as:

$$\forall \tau_i \in \Delta : R = \sum_{k=1}^{|\Delta|} \left\lceil \frac{R}{p_k} \right\rceil \epsilon_k \leq d_i \tag{10}$$

and R can be solved by the following recurrence relation:

$$R_{t+1} = \sum_{k=1}^{|\Delta|} \left\lceil \frac{R_t}{p_k} \right\rceil c_k \tag{11}$$

where $R_0 = \sum_{k=1 \to |\Delta|} c_k$. Equation (11) surely converges if the processor utilization factor does not exceed 100%. The schedulability condition is given by the following condition:

$$R_{t+1} > \max_{k=1}^{|\Delta|} \{d_k\} \tag{12}$$

where d_k is the deadline for the k^{th} callback.

The Algorithm 2 finds, from a set Δ, a subset of callbacks Π that can be assigned to the same executor with a time complexity of $\mathcal{O}((\max(p_i)/\min(p_i))n)$.

Algorithm 2: Mapping feasibility test

Input: $\Delta = \{\tau_1, \tau_2, \ldots, \tau_n\}$
Output: Π

$\Pi = \emptyset$; //
$R = \sum_{i=1}^{|\Delta|} c_i$; //
$D_{max} = \max_{i=1}^{|\Delta|}\{d_i\}$; //
cont = True ; //
while *cont* **do**
  $R_t = \sum_{i=1}^{|\Delta|} \left\lceil \frac{R}{p_i} \right\rceil c_i$; //
  if $R == R_t$ **then**
    cont = False
  end
  if $R_t > D_{max}$ **then**
    return $\emptyset$; // Δ is not schedulable
  end
  $R = R_t$;
end
for $i = 1$ *to* $|\Delta|$ **do**
  if $(R \leq d_i)$ **then**
    $\Pi = \Pi \cup \{\tau_i\}$;
  end
end

5.2. *Algorithm 3: Executor Creation*

This section presents the algorithm *CreateExecutor*, whose objective is to select a subset of callbacks to be assigned to one executor and to determine its scheduling parameters.

The Algorithm 3 takes as input a set of callbacks Π and then determines as output a subset Λ to be assigned to an executor e. The algorithm also defines the sequence order of callbacks within the executor and computes its timing parameters: vector of execution times $\vec{E}$, period T, and deadline D.

We say that there is a set of callbacks $\Lambda \subseteq \Pi$ that can be assigned to the same executor if and only if:

Condition 1: $\forall i \in \Lambda : T = GCD_{i=1}^{|\Lambda|} p_i > 1$
Condition 2: $\overline{E} = \sum_{s=0}^{N-1} C_s \leq T_j$
Condition 3: $R \leq \min_{i=1}^{|\Lambda_j|} d_i$

Condition 3 is ensured by the feasibility test, which guarantees that R is less than or equal to the shortest callback deadline. Moreover, the executor deadline may be longer than its period, and consequently, buffering instances may occur for the execution of an executor. The number of buffered instances of an executor e_j is bounded by $\left\lceil R_{\tau_j}/T_j \right\rceil$. The first condition prevents the maximum buffering of an executor instance by creating subsets of callback periods.

To implement this, the bucket select algorithm (BSA) is used to divide n callbacks into several buckets $\{L_i\}$ using the divisibility test by prime numbers. The bucket select Algorithm 4 described in the following and illustrated by an example in Figure 5 selects one bucket from n callbacks with a time complexity of $\mathcal{O}(kn)$.

Algorithm 3: CreateExecutor

Input: $\Pi = \{\tau_1, \tau_2, \ldots, \tau_n\}$
Output: Λ and executor $e(\vec{E}, T, D)$

$\Lambda = \emptyset$; //
$D = \infty$; //
let K be the list of k first prime numbers ; //
$[L_b, T_b] = BucketSelectAlgorithm(\Pi, K)$; // Algorithm 4
Sort L_b by increased period order ; //
$T_w = p_1$; //
$\pi = p_1$; //
for *i in* L_b **do**
  $T_w = LCM(\pi, p_i)$; ; //
  $\overline{E} = max_{s=0}^{T_w/T} C_s$; // Computed using Equation (6)
  if $\overline{E} \leq T$ **then**
    $\Lambda = \Lambda \cup \{r_i\}$; //
    $D = \min\{D, d_i\}$; //
    $\pi = T_w$; //
  end
end
Sort Λ in ascending deadline order ; // Define callbacks execution order k
return Λ, e

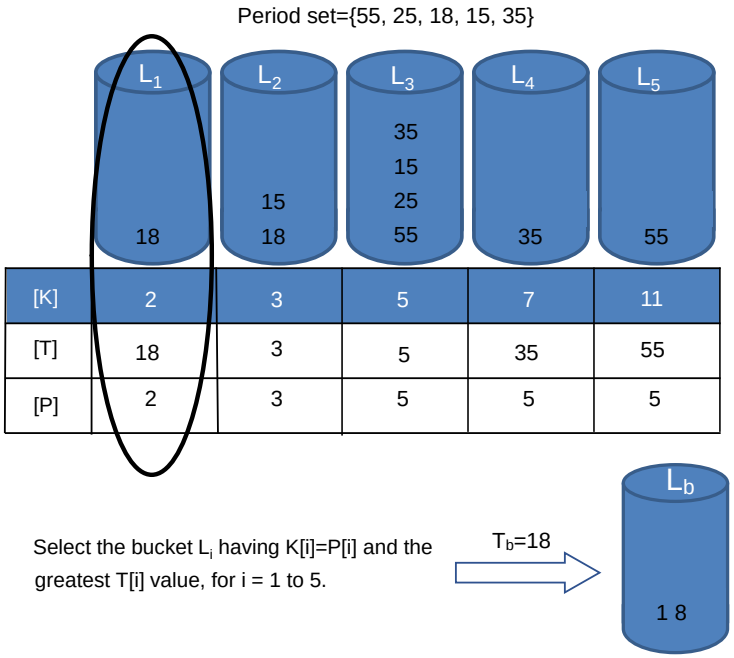

Figure 5. Example of the bucket select algorithm (BSA).

Algorithm 4: Bucket select algorithm

Input: $\Pi = \{\tau_1, \tau_2, \ldots, \tau_n\}$, $K = \{x_1, x_2, \ldots, x_k\}$
Output: L_b, T_b
/* Creating k empty list $\{L\}$ and two list of k sise [T] and [P] */
for $i = 1$ *to* k **do**
  $L_i = \emptyset; T[i] = 0; P[i] = 0;$
end
/* Creating buckets $L_1, L_2, \ldots, L_k$ */
for $i = 1$ *to* n **do**
  for $j = 1$ *to* k **do**
    if $mod(p_i, x_j) = 0$ **then**
      $L_j = L_j \cup \tau_i$; /* */
    end
  end
end
/* Computing [T] and [P] */
for $i = 1$ *to* k **do**
  $T[i] = GCD(L_i)$; //
  for $j = 1$ *to* k **do**
    if $mod(T[i], x_j) = 0$ **then**
      $P[i] = x_j$; // This is the first prime number, where T_i is divisible by it
      break
    end
  end
end
/* Determining L_b and T_b */
for $j = 1$ *to* k **do**
  if $x_j = P[j]$ *and* $T[j] > T_b$ **then**
    $T_b = T[j]$; //
    $L_b = L_j$; // Select the bucket that has the greatest period value
  end
end

Let [K] be a list of first prime numbers. For each prime number x_i (i = 1 to k), the algorithm regroups in a list L_i (i = 1 to k) the callbacks whose periods can be divided by x_i. Therefore, each bucket L_i contains a subset of callback periods characterized by two timing parameters: $T[i]$, which corresponds to the GCD of the callback periods of the bucket L_i, and $P[i]$, which is the minimum prime number, where $T[i]$ is divisible by it. Finally, the algorithm returns the bucket L_b with $x_i = P[i]$ and the highest period value $T[i]$, denoted by T_b.

To avoid overloading the executor, a sequencing of callbacks within the executor is necessary. For that, we adapt the lowest peak (LP) method developed in [35]. The algorithm chooses the callback offset through a larger frame window T_w, which is equal to the LCM of the callback periods. The maximum execution time $\overline{E_j} = max_{s=0}^{T_w/T_j} C_s$ must be less than or equal to the executor period T_j.

The Algorithm 3 creates an executor e from n callbacks with a time complexity of $\mathcal{O}((k+N)n+n)$, where N is the frame number ($N = T_w/T$) and k is the number of prime numbers.

6. Experimental Results

This section presents and discusses the evaluation of the proposed algorithm. The evaluation is carried out in a uniprocessor case study with periodic and independent callbacks. The proposed solution is compared with three scheduling methods: RMS, GBFS, and greedy LL scheduling algorithms.

The rate monotonic scheduling (RMS) algorithm was introduced by Liu and Layland in 1973 [21] and is considered a baseline for real-time scheduling. It is a preemptive fixed-priority scheduling algorithm suited for periodic independent and synchronous tasks. The priority of a task is inversely proportional to its period, i.e., the highest priority task will correspond to the task with the shortest period. The algorithm clusters n callbacks on m executors in two steps:

- In the first step, callbacks that have the same periods are grouped in the same executor with a time complexity of $\mathcal{O}(mn)$;
- In the second step, a merge sort algorithm is used to define callback priority by assigning high priority to the shortest deadlines with a time complexity of $\mathcal{O}(m^2)$.

In the following, a pseudocode of the RMS algorithm is given (Algorithm 5). The algorithm iterates to regroup elements of S that have the same period while satisfying a linear schedulability test (Schedule() function) defined in [36] and considering the following assumptions:

Clustering two elements τ_i and τ_j in S $\{tau_i \cup \tau_j\}$ produces a new element with the following parameters: $c_{ij} = c_i + c_j$, $d_{ij} = min(d_i, d_j)$, and $p_{ij} = p_i = p_j$.

Algorithm 5: Rate monotonic scheduling (RMS) algorithm.

Input: $S = \{\tau_1, \tau_2, \ldots, \tau_n\}$
Function `clustering(`S`)`**:**
  $S^{'} = \emptyset$; //
  Sort S in ascending order of priorities first, and deadlines second ; //
  for $i = n - 1 \rightarrow 0$ **do**
    if $p_i = p_n$ **then**
      $S^{'} = \{S \setminus \{\tau_i\} \cup \{\tau_{in}\}$; `/* Clustering` p_i `and` p_n `*/`
    if *Schedule* $(S^{'})$ **then** ; `// Test the schedulability of the system`
      $S_{min} = S^{'}$
    else
      $p_i = p_i + 1$; `/* Increase the priority of` p_i `*/`
      $S_{min} = S$; `/* Dot not cluster` τ_i `with` τ_n `*/`
  if S_{min} *not* $\emptyset$ **then**
    return clustering (S_{min}) ; `/* Iterate for the next element */`
  else
    return S

Note that the number of executors in this case is equal to at least the number of distinct periods in the set of callbacks. In industry, the system designer still uses the algorithm for embedded critical systems, as it guarantees upper bounds to the response time and is simple to implement. However, it is generally too conservative (pessimiste) for practical use and does not optimize the use of system resources [37].

The GBFS algorithm developed by [17] uses the greedy heuristic approach to gather the largest number of callbacks to the same executor. Priorities are assigned by deadline, i.e., the callbacks with the shortest deadline have the highest priority. The time complexity of the algorithm is of $\mathcal{O}(n^4)$, where n is the initial number of callbacks to be clustered (Algorithm 6).

Algorithm 6: GBFS algorithm.

```
   Input: S = {τ1, τ2, ..., τn}
1  Function Clustering(S):
2      S' = ∅ ;                                                    //
3      minSet = ∅ ;                                                //
4      minSumTests = ∅ ;                                           //
5      Sort S in ascending order of deadlines ;                    //
6      for i=n→0 do
7          for j = i − 1 → 0 do
8              if p_i = p_j then
9                  if (c_i + c_j) ≤ min(d_i, d_j) then ; // Test laxity
10
11                     S' = {S \ {τ_i, τ_j}} ∪ τ_ij if Schedule (S') then
12                         if cost(S) < minSumTests then
13                             m
14                         inSumTests = cost(S) ;                  //
15                         S_minSet = S' ;                         //
16 if S_min not ∅ then
17     return Clustering (minSet)
18 else
19     return S
```

For each clustering iteration, the algorithm uses a sufficient schedulability test and a cost function to select the best result in each clustering iteration (Equation (13)).

$$cost(S) = \sum_{k=0}^{|S|} \frac{c_k}{d_k} \tag{13}$$

The third algorithm used for comparison is the least loaded (LL) heuristic approach, used in [18]. The approach applies a deadline test schedulability similar to the one used in this paper along with a heuristic approach of least load to design the cost function, and defines the sequencing order of callbacks inside the executor with a complexity of $\mathcal{O}(n^2)$.

6.1. Workload Generation

A set of callbacks is generated randomly using the UUnifast algorithm developed by [22] in order to accurately assess the proposed solution in different execution scenarios. The algorithm splits the CPU usage factor U into smaller usage factors u_i for n callbacks. For each callback τ_i, the deadline d_i is determined by the following equation:

$$d_i = (p_i - c_i)rand(a, b) + c_i \tag{14}$$

where $rand(a, b)$ generates values from a uniform distribution in the interval $[a, b]$ with $0 \leq a \leq b$. Using these equations, several sets of callbacks are randomly generated with a cardinality varying from 10 to 10,000. The callback period is chosen randomly in the range of [10, 275] ms and the deadline is set equal to the period.

6.2. Performance Metrics

The comparison study considers four performance metrics: success ratio, number of executors, execution time, and average response time.

Success ratio: Consider a set $\Sigma = \{\Delta_1, \Delta_2, \ldots, \Delta_i, \ldots, \Delta_N\}$ composed of N subsets of callbacks Δ_i, and A as a scheduling algorithm. We say that Δ_i can be scheduled by the

scheduling algorithm A if all callbacks from Δ_i meet their deadlines, denoted by $A(\Delta_i) = 1$, and if not, we denote it by $A(\Delta_i) = 0$. As a result, the success rate SR of the scheduling algorithm A on Σ is defined by the following:

$$SR = 100 * \frac{\sum_{i=1}^{N} A(\Delta_i)}{N} \tag{15}$$

Number of executors: This metric represents the number of executors provided by the algorithm to schedule all callbacks while meeting real-time constraints. We try to minimize this metric.

Runtime: This metric measures the time it takes the algorithm to obtain a solution.

Average response time: Average response time is the time elapsed between the activation and completion of a callback. The following equation is defined to calculate the average response time rate TRm for a set of callbacks with cardinality n:

$$TRm = 100 * \frac{\sum_{i=1}^{n} \frac{c_i}{d_i}}{n} \tag{16}$$

6.3. Schedulability Test

Figure 6 shows the success ratio of all algorithms when varying callback deadlines. For each deadline interval $[a, b]$, we run all algorithms on sets of 1000 callbacks with a cardinality of 100 callbacks per set and a CPU utilization factor of 90%.

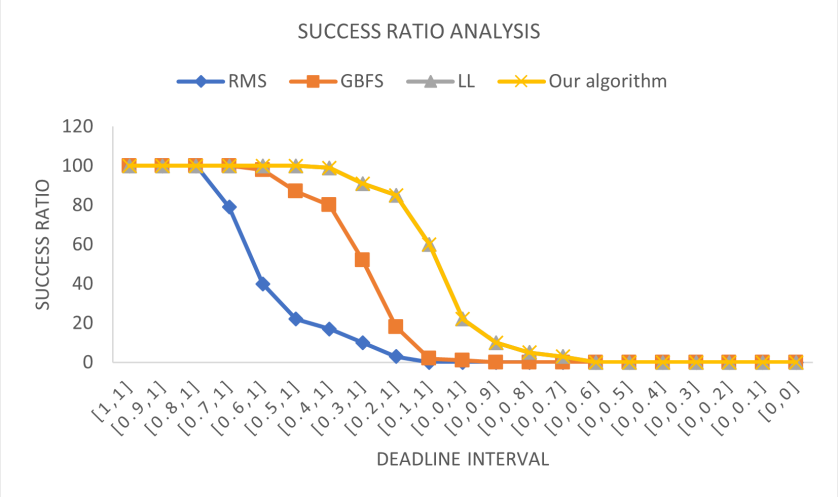

Figure 6. Success ratio vs. deadline.

Figure 6 shows that the proposed algorithm significantly outperforms both the RMS and the GBFS algorithms in terms of the success ratio. We observe that the ratio decreases as the deadlines become shorter (stressed system). RMS algorithms quickly fail to schedule callbacks when the proposed algorithm continues to provide feasible solutions under tight deadlines. However, LL provides the same success ratio as the proposed algorithm, since it applies the same schedulability test.

6.4. Number of Executors

In this experiment, we compare the number of executors obtained by all algorithms. A utilization factor of 60% is used and 10 callback sets are randomly generated with a cardinality of 100 and a deadline on request.

We then successively run all algorithms on all workloads and determine the maximum number of executors among the 10 sets of callbacks.

As shown in Figure 7, the proposed algorithm outperforms RMS, LL, and GBFS in terms of the number of executors. For both RMS and GBFS, the algorithms try to regroup callbacks with the same period to the same executor, while LL regroups periods to the same executor. The proposed algorithm builds executors from callbacks with arbitrary periods. This leads to a minimal number of executors with low variance while guaranteeing system schedulability even in very stringent deadline situations. On the other hand, in RMS and GBFS, the variance is greater and the number of executors may be twice the number of distinct periods. This may have a major influence on the system stack memory. For example, suppose that all callbacks consume the same stack memory size of 512 bytes. The stack of an executor is the maximum size of all callbacks assigned to it. Therefore, the stack memory used to implement MPS for a deadline interval [0.2, 1] is 20 executors*512 bytes = 10 kb. However, in the proposed algorithm, it is only 2 executors*512 bytes = 1 kb.

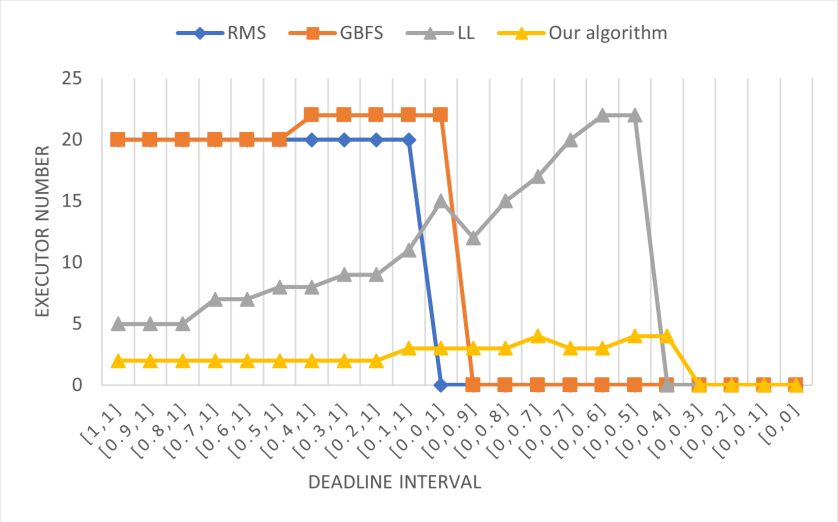

Figure 7. Executor number vs. deadline.

Let us also note that the RMS and GBFS algorithms do not succeed in scheduling the system, since the deadline becomes tight where the proposed algorithm still finds a schedulable system.

6.5. Runtime Analysis Comparison (in ms)

Runtime analysis of algorithms is performed by running each algorithm on different sets of callbacks with a cardinality ranging from 1 to 1000. For each cardinality value, ten sets of callbacks are randomly generated with a deadline on request (i.e., the deadline is equal to the period), and a CPU utilization factor of 60%. All algorithms are run successively on all callback sets, and the average execution time is computed for each. The runtime is obtained using a PC i7-2640-2.8 GHz and 8 GB of memory.

As shown in Table 4, the GBFS algorithm quickly fails to find an acceptable solution (n/a) when the number of callbacks increases. However, the proposed algorithm outperforms RMS in finding a schedulable solution in a reasonable time. The lower time complexity of the algorithm proposed by $\mathcal{O}((k+N)n+n)$ leads to a minimum runtime, compared to the RMS and LL algorithms that have a time complexity of $\mathcal{O}(m^2+mn)$ and $\mathcal{O}(n^2)$, respectively, and the GBFS algorithm that has a polynomial time complexity of $\mathcal{O}(n^4)$.

Table 4. Runtime results (in ms).

Callback Cardinality	GBFS	RMS	LL	Proposed Algorithm
50	12,000	1345	1744	30
60	n/a	1862	732	160
80	n/a	2668	1260	924
100	n/a	5316	n/a	769
200	n/a	16,705	n/a	2483

6.6. Response Time Analysis

This experiment considers 10 sets of callbacks. A set contains 100 callbacks randomly generated with a CPU utilization factor of 60%.

For each deadline interval $[a, b]$, we run all algorithms on 10 sets of callbacks and determine the maximum average response time. Figure 8 shows the average response time versus the deadline. The proposed algorithm finds a mapping of callbacks to executors that gives better response times than those obtained by other algorithms. This is due to the tuning of the callback sequencing and activation. However, the LL algorithm underperforms compared with the RMS and GBFS algorithms by causing the highest response times. Furthermore, we note that when the deadlines tighten after the interval [0.2, 1], both the LL and the proposed algorithm continue to be schedulable with a gradual increase in response time up to the interval [0, 0.5].

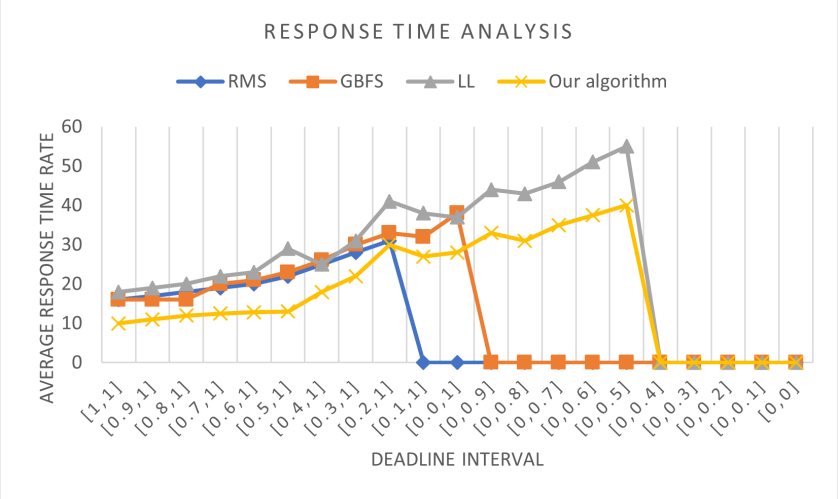

Figure 8. Response time for different algorithms.

7. Limitations, Extensions, and Conclusions

This paper provides a new method to optimize the allocation and configuration of software resources for ROS embedded systems. The proposed solution builds a set of executors, maps callbacks to executors, assigns executor priority, and defines the execution sequencing of callbacks. The experimental results reported in this article demonstrate the efficiency of the new proposed algorithm in improving system scheduling and performance. For overloaded systems with tight deadlines, the proposed algorithm continues to provide a feasible solution while minimizing the number of executors to schedule the embedded application, whereas other algorithms do not find a solution.

The proposed solution is applied to critical embedded real-time systems with periodic executions. Formal analysis and optimization should be performed in the configuration phase and before system execution. The study considers a monocore system with a priority-driven scheduling policy. In extension to this study, future work should extend the system

model to consider multicore architecture with resource dependencies. A trade-off between response time, stack memory usage, and data consistency should be considered. In addition, end-to-end system analysis can be performed by extending the model with the notion of end-to-end chains for distributed systems. Thus, the analysis considers dependencies among callbacks when computing the bound of end-to-end response time.

Funding: The author would like to thank the Deanship of Scientific Research at Umm Al-Qura University for supporting this work by Grant Code: (22UQU4361048DSR04).

Conflicts of Interest: The author declares no conflict of interest.

References

1. Quigley, M.; Gerkey, B.; Conley, K.; Faust, J.; Foote, T.; Leibs, J.; Berger, E.; Wheeler, R.; Ng, A. ROS: An open-source Robot Operating System. *ICRA Workshop Open Source Softw.* **2009**, *3*, 1–6.
2. Koubaa, A. *Robot Operating System (ROS): The Complete Reference (Volume 3)*; *Studies in Computational Intelligence*; Springer: Cham, Switzerland, 2018; Volume 778. https://doi.org/10.1007/978-3-319-91590-6.
3. Li, Z.; Hasegawa, A.; Azumi, T. A_Perf: A tracing and performance analysis framework for ROS 2 applications. *J. Syst. Archit.* **2021**, *123*, 102341. https://doi.org/10.1016/j.sysarc.2021.102341.
4. Macenski, S.; Foote, T.; Gerkey, B.; Lalancette, C.; Woodall, W. Robot Operating System 2: Design, architecture, and uses in the wild. *J. Sci. Robot.* **2022**, *7*, abm6074. https://www.doi.org/10.1126/scirobotics.abm6074.
5. Yang, Y.; Azumi, T. Exploring Real-Time Executor on ROS 2. In Proceedings of the 2020 IEEE International Conference on Embedded Software and Systems (ICESS), Shanghai, China, 10–11 December 2020; pp. 1–8. https://doi.org/10.1109/ICESS49830.2020.9301530.
6. Casini, D.; Blaß, T.; Lütkebohle, I.; Brandenburg, B. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In Proceedings of the 31th Euromicro Conference on Real-Time Systems (ECRTS 2019), Stuttgart, Germany, 2019. https://doi.org/10.4230/LIPIcs.ECRTS.2019.6.
7. Maruyama, Y.; Kato, S.; Azumi, T. Exploring the Performance of ROS-2. In Proceedings of the 13th International Conference on Embedded Software, EMSOFT '16, Pittsburgh, PA, USA, 1–7 October 2016 https://doi.org/10.1145/2968478.2968502.
8. Choi, H.; Xiang, Y.; Kim, H. PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2. In Proceedings of the 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS), Online, 18–21 May 2021; pp. 251–263. https://doi.org/10.1109/RTAS52030.2021.00028.
9. Blaß, T.; Casini, D.; Bozhko, S.; Brandenburg, B.B. A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance. In Proceedings of the 2021 IEEE Real-Time Systems Symposium (RTSS), Dortmund, Germany, 2021; pp. 41–53. https://doi.org/10.1109/RTSS52674.2021.00016.
10. Rota, G.C. The Number of Partitions of a Set. *Am. Math. Mon.* **1964**, *71*, 498–504. https://doi.org/10.1080/00029890.1964.11992270.
11. Wilhelm, T.; Weber, R. Towards Model-Based Generation and Optimization of AUTOSAR Runnable-to-Task Mapping. In Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 2019; pp. 38–43. https://doi.org/10.1109/MODELS-C.2019.00012.
12. Zhao, Q.; Gu, Z.; Zeng, H. Design optimization for AUTOSAR models with preemption thresholds and mixed-criticality scheduling. *J. Syst. Archit.* **2017**, *72*, 61–68. https://doi.org/10.1016/j.sysarc.2016.08.003.
13. Gupta, P.; Singh, N.P.; Srinivasan, G. An Efficient Approach For Mapping AUTOSAR Runnables in Multi-core Automotive systems to Minimize Communication Cost. In Proceedings of the 2019 Innovations in Power and Advanced Computing Technologies (i-PACT), Vellore, India, 2019; Volume 1, pp. 1–4. https://doi.org/10.1109/i-PACT44901.2019.8960215.
14. Saidi, S.E.; Cotard, S.; Chaaban, K.; Marteil, K. An ILP Approach for Mapping AUTOSAR Runnables on Multi-core Architectures. In Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Amsterdam, The Netherlands, 19–21 January 2015; pp. 6:1–6:8. https://doi.org/10.1145/2693433.2693439.
15. Al-bayati, Z.; Sun, Y.; Zeng, H.; Natale, M.D.; Zhu, Q.; Meyer, B.H. Partitioning and Selection of Data Consistency Mechanisms for Multicore Real-Time Systems. *ACM Trans. Embed. Comput. Syst.* **2019**, *18*, 1–28. https://doi.org/10.1145/3320271.
16. Wozniak, E.; Mehiaoui, A.; Mraidha, C.; Tucci-Piergiovanni, S.; Gerard, S. An optimization approach for the synthesis of AUTOSAR architectures. In Proceedings of the 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA), Cagliari, Italy, 10–13 September 2013; pp. 1–10.
17. Bertout, A.; Forget, J.; Olejnik, R. A Heuristic to Minimize the Cardinality of a Real-time Task Set by Automated Task Clustering. In Proceedings of the 29th Annual ACM Symposium on Applied Computing, Gyeongju, Republic of Korea, 24–28 March 2014; pp. 1431–1436. https://doi.org/10.1145/2554850.2554958.
18. Mixed Harmonic Runnable Scheduling for Automotive Software on Multi-Core Processors. *Int. J. Automot. Technol.* **2018**, *19*, 323–330.
19. Khenfri, F.; Chaaban, K.; Chetto, M. Efficient mapping of runnables to tasks for embedded AUTOSAR applications. *J. Syst. Archit.* **2020**, *110*, 101800.

20. Kim, S. Efficient Exact Response Time Analysis for Fixed Priority Scheduling in Lowest Priority First-Based Feasibility Tests. *IEEE Embed. Syst. Lett.* **2021**, *13*, 69–72. https://doi.org/10.1109/LES.2020.3025600.
21. Liu, C.L.; Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* **1973**, *20*, 46–61.
22. Bini, E.; Buttazzo, G. Measuring the Performance of Schedulability Tests. *Real-Time Syst.* **2005**, *30*, 129–154.
23. Saito, Y.; Azumi, T.; Kato, S.; Nishio, N. Priority and Synchronization Support for ROS. In Proceedings of the 2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA), Nagoya, Japan, 6–7 October 2016; pp. 77–82. https://doi.org/10.1109/CPSNA.2016.24.
24. Tang, Y.; Feng, Z.; Guan, N.; Jiang, X.; Lv, M.; Deng, Q.; Yi, W. Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors. In Proceedings of the 2020 IEEE Real-Time Systems Symposium (RTSS), Houston, TX, USA, 1–4 December 2020; pp. 231–243. https://doi.org/10.1109/RTSS49844.2020.00030.
25. Chaaban, K. A seamless integration of fault-tolerant and real-time capabilities for Robot Operating System (ROS), In Proceedings of the 4th International Conference on Applied Automation and Industrial Diagnostics ICAAID, Hail, Saudi Arabia, 29–31 March 2022. https://doi.org/10.1109/ICAAID51067.2022.9799496
26. Krichen, M.; Lahami, M.; Cheikhrouhou, O.; Alroobaea, R.; Maâlej, A.J., Security Testing of Internet of Things for Smart City Applications: A Formal Approach. In *Smart Infrastructure and Applications: Foundations for Smarter Cities and Societies*; Mehmood, R., See, S., Katib, I., Chlamtac, I., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 629–653. https://doi.org/10.1007/978-3-030-13705-2_26.
27. Fernandez, J.; Allen, B.; Thulasiraman, P.; Bingham, B. Performance Study of the Robot Operating System 2 with QoS and Cyber Security Settings. In Proceedings of the 2020 IEEE International Systems Conference (SysCon), Montreal, QC, Canada, 24 August–20 September 2020; pp. 1–6. https://doi.org/10.1109/SysCon47679.2020.9275872.
28. Lu, Q.; Li, X.; Guan, Y.; Wang, R.; Shi, Z. Modeling and Analysis of Data Flow-Oriented ROS2 Data Distribution Service. *Int. J. Softw. Inform.* **2021**, *11*, 505–520. https://doi.org/10.21655/ijsi.1673-7288.00258.
29. Chen, J.; Du, C.; Zhang, Y.; Han, P.; Wei, W. A Clustering-Based Coverage Path Planning Method for Autonomous Heterogeneous UAVs. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 25546–25556. https://doi.org/10.1109/TITS.2021.3066240.
30. Chen, J.; Ling, F.; Zhang, Y.; You, T.; Liu, Y.; Du, X. Coverage path planning of heterogeneous unmanned aerial vehicles based on ant colony system. *Swarm Evol. Comput.* **2022**, *69*, 101005. https://doi.org/10.1016/j.swevo.2021.101005.
31. Chen, J.; Zhang, Y.; Wu, L.; You, T.; Ning, X. An Adaptive Clustering-Based Algorithm for Automatic Path Planning of Heterogeneous UAVs. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 16842–16853. https://doi.org/10.1109/TITS.2021.3131473.
32. Pardo-Castellote, G. OMG Data-Distribution Service: architectural overview. In proceedings of IEEE Military Communications Conference, Boston, MA, USA, 2003; pp. 242–247, Volume 1. https://doi.org/10.1109/MILCOM.2003.1290110.
33. Cousins, S. Exponential growth of ROS. *Robot. Autom. Mag.* **2011**, *18*, 19–20. https://doi.org/10.1109/MRA.2010.940147.
34. Mok, A.K.; Chen, D. A multiframe model for real-time tasks. *IEEE Trans. Softw. Eng.* **1997**, *23*, 635–645. https://doi.org/10.1109/32.637146.
35. Monot, A.; Navet, N.; Bavoux, B.; Simonot-Lion, F. Multisource Software on Multicore Automotive ECUs Combining Runnable Sequencing With Task Scheduling. *Ind. Electron.* **2012**, *59*, 3934–3942.
36. Audsley, N. *Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times*; Department of Computer Science, University of York: York, UK, 1991.
37. Cottet, F.; Delacroix, J.; Kaiser, C.; Mammeri, Z. *Scheduling in Real-Time Systems*; Wiley: Oxford, UK, 2002; p. 282. ISBN 0-470-84766-2.