Chain-Aware Runnable Scheduling in AUTOSAR: A Case Study in Automotive Systems

Hong Gi Park* and Hyunjong Choi†
* Hyundai Motor Company
† San Diego State University
hgpark@hyundai.com, hyunjong.choi@sdsu.edu

***Abstract*—The increasing complexity of functional operations, combined with the automotive industry's extensive collaboration with multiple vendors, presents significant challenges in managing software components within automotive systems. While AUTOSAR (AUTomotive Open System ARchitecture) aims to address these issues by enhancing software modularity and reusability, comprehensive studies on end-to-end latency, real-time capability, and timing analysis for safety-critical applications remain limited. This study proposes a *chain-aware runnable scheduling* framework to enhance real-time performance and reduce end-to-end latency in AUTOSAR-based automotive systems. Experimental results, incorporating priority assignment and runnable allocation, and a case study on real vehicle control units (VCU) of commercial vehicles, conducted using practical scenarios, indicate that the proposed framework reduces the end-to-end latency of the most safety-critical chain by 82% compared to the existing AUTOSAR schedulers.**

***Index Terms*—AUTOSAR, end-to-end latency, runnable scheduling, automotive systems, chain-aware scheduling**

I. Introduction

The growing complexity of software functions, along with more frequent update cycles, has made software management increasingly difficult in the automotive industry. Moreover, the extensive collaboration with various vendors challenges the ability of in-house development teams to ensure timely updates of software and maintain integrity. AUTOSAR [1] aims to address these challenges by enhancing the modularity, reusability, interoperability, and security of automotive software components.

Many safety-critical functions in the automotive industry, such as perception pipeline and emergency stop, form specific types of processing chains under the AUTOSAR environment. These processing chains consist of sequences of tasks (runnables) that must be executed in a precise order and timely manner to ensure the correct functional operation. Therefore, the end-to-end latency from the recognition of a sensor event to the corresponding action of an actuator is one of crucial metrics for the system safety.

Despite the widespread adoption of AUTOSAR in the automotive industry, comprehensive studies on end-to-end latency analysis, real-time capability, and timing analysis for this framework remain limited. The existing literature often falls short of providing in-depth examinations of how AUTOSAR handles runnable scheduling, particularly in complex, safety-critical automotive applications while extensive research has been conducted in other middleware frameworks such as ROS and ROS 2. The absence of robust timing analysis frameworks and methodologies for AUTOSAR makes it challenging to ensure that such systems can meet stringent real-time performance criteria, potentially leading to unforeseen issues in the industrial environment.

This paper aims to address this gap by proposing new runnable scheduling approaches with analysis frameworks for AUTOSAR-based automotive systems. To the best of our knowledge, there has been no recent work on formally analyzing the scheduling architecture of the AUTOSAR platform, especially with respect to end-to-end latency of processing chains. The main contributions of this paper are shown as follows:

- We present chain-aware runnable scheduling framework which is inspired by the existing work [5], yet specifically tailored for AUTOSAR environment. This includes priority assignment for runnables and the allocation of runnables to OS-tasks and available CPU cores.
- We conduct an analysis of end-to-end latency of chains under three different runnable scheduling approaches in AUTOSAR: time-triggered (basic OS-task), event-triggered (extended OS-task), and chain-aware (modified extended OS-task). We compare their performance characteristics using randomly-generated workload sets under various experimental setups.
- We perform a case study on the real vehicle control units (VCU) platforms provided by an automotive company using a practical scenario.

II. Related Work

In the context of AUTOSAR, many studies have been conducted on mapping runnable to OS-tasks. Authors in [10, 11, 14, 15] proposed rules for mapping runnables to OS-tasks based on intra-ECU communication without considering schedulability of the system. Zeng et al. [14, 15] used MILP (mixed integer linear programming) method, considering data consistency to find optimal mapping and tasks priorities. In [10, 11], authors developed heuristic distribution algorithms for runnables across multiple cores to balance the core loads uniformly. However, none of these approaches considers end-to-end latency of processing chains in AUTOSAR.

Analyzing end-to-end latency of chains has been studied extensively in read-execute-write semantics. Authors proposed analysis of tasks with precedence constraints in multi-core

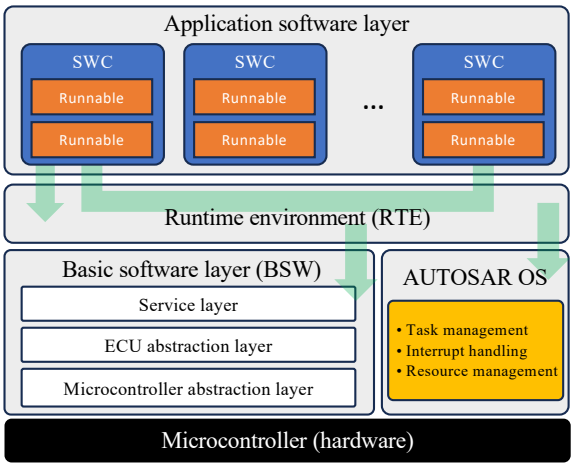


Fig. 1: AUTOSAR architecture

environment in [12, 13]. Becker et al. [2] presented analytical methods to bound the end-to-end latency of a chain under fixed-priority scheduling. Analysis of end-to-end latency of cause-effect chains are addressed in [8]. Choi et al. in [4] proposed chain-based fixed-priority scheduling to improve end-to-end latency for loosely-dependent chain configuration. However, these approaches cannot be directly applicable to AUTOSAR because of discrepancies in scheduling model.

The recent work of Choi et al. [5] focused on improving end-to-end latency of chains by proposing priority-driven chain-aware scheduling framework for ROS 2 middleware. They re-designed the ROS 2 callback scheduling structure and proposed resource allocation policies. While their approach inspires our work, it is not directly applicable to AUTOSAR due to the architectural differences between ROS 2 and AUTOSAR platform. We review their work in more details and explain how to adapt their ideas in the Section V.

III. Background and System Model

A. AUTOSAR architecture

The AUTOSAR architecture is structured into three primary software layers running on a microcontroller: the application layer, the runtime environment (RTE), and the basic software (BSW) as illustrated in Fig 1. Application software layer is hardware-independent, containing the software components (SWCs) that define the application logic and functionality. The RTE faciliates communication between software components and provides access to the BSW. It serves as the interface for applications, ensuring that components can interact seamlessly. The BSW is subdivided into three major layers such as service-, electronic control unit (ECU) abstraction-, and microcontroller abstraction-layer. These sub-layers provide core services such as communication, diagnostics, memory management, and operating system services, aiming to create a standardized environment for automotive applications. The BSW layer also incorporates the AUTOSAR OS based on OSEK [7][1] as part of its operating system services.

[1]OSEK is a standard for a real-time operating system (RTOS) designed specifically for automotive applications. Now, it is referred to as AUTOSAR OS

B. Key components in AUTOSAR

There are several fundamental key components that play a pivotal role in real-time scheduling in AUTOSAR-based automotive systems:

Event. Events in AUTOSAR are used to trigger actions, signal state changes, and facilitate the coordination of tasks within the system. *Timing events* are triggered based on a predefined time interval and used to periodically activate tasks or runnables at regular intervals, which is mainly used in automotive systems. *Data receive events* are triggered when new data is received on a particular port or communication channel. It is also used to activate a task or runnable when new data arrives, ensuring that the data is processed promptly, although it is not widely used in automotive systems' setting.

Runnable. A Runnable is the minimal schedulable entity in AUTOSAR. It represents a piece of functional behavior within a software component and is associated with a specific event defined by a designer. The execution order of runnables can be configured, and they are allocated to OS Tasks by system designers. However, there are no official guidelines for resource allocation.

OS Tasks. OS Task is an OS-level scheduling entity provided by OSEK in AUTOSAR. OS Task can contain one or multiple runnables and executes them by their predefined order in a non-preemptable manner. OSEK manages the execution of OS Tasks by their priorities on the assigned CPU cores, which corresponds to a thread in typical operating systems. OS Tasks are categorized into basic- and extended OS Tasks: (1) *Basic OS Task* is suitable for periodic activities that do not require complex synchronization or waiting, such as polling sensors at regular time intervals, because it cannot be put into a waiting state. (2) *Extended OS Task* can enter waiting states, allowing them to wait for external events, resources, or alarms. This makes them more flexible and capable of handling complex synchronization scenarios, which is ideal for event-driven activities.

RTE. RTE plays a crucial role in ensuring seamless communication and integration between software components and the underlying BSW. The key functions of the RTE are the inter-component communication, scheduling and task management, and resource management such as memory, CPU time, and I/O bandwidth to different software components.

C. System Model

In this section, we introduce our system model for runnables, OS Tasks, and chains under AUTOSAR environment. Our system model incorporates a fixed same clock frequencies across all CPU cores. This model provides a robust framework for developing and deploying real-time automotive applications.

1) Runnable model: The system consists of real-time runnables, each triggered by either a timer event using an alarm or various external events, e.g., the completion of its precedent runnable in a chain. Each runnable can be associated

with more than one chain. A runnable r_i is characterized as follows:

$$r_i := (C_i, D_i, T_i)$$

- C_i: The worst-case execution time of job of r_i.
- D_i: The relative deadline of a runnable r_i, which is equal to the deadline of its associated chain.
- T_i: The period of a runnable r_i, equal to the period of its associated chain ($Di \leq Ti$).

We denote the priority of runnable r_i within the OS Task it belongs to as π_{r_i}.

2) OS-task model: We represents a set of OS-tasks as below:

$$\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_j, ..., \tau_E\}$$

The priority of j-th OS-task is denoted by π_{τ_j} and $\mathcal{T}$ is sorted in descending order of priority, i.e., $\pi_{\tau_j} > \pi_{\tau_{j+1}}$, and each OS-task has a unique fixed priority during the design phase. The scheduling of OS-tasks is conducted in a preemptive manner. To avoid confusion with a typical workload (e.g., a task), we use the term "OS-task" specifically to represent OS task within the AUTOSAR environment. The period of an OS-task, denoted as T_τ is determined by the greatest common divisor (GCD) of the periods of all runnables assigned to that OS-task.

3) Chain model: Each chain consists of one or more runnables. A chain, defined as Γ^c, is denoted as below:

$$\Gamma^c := [r_s, r_{m1}, r_{m2}, ..., r_e]$$

- r_s: The start runnable a chain Γ^c.
- r_{m*}: The intermediate runnables of a chain Γ^c.
- r_e: The end runnable of a chain Γ^c.

The superscript c uniquely identifies the chain Γ^c. The chain initiates with the start runnable r_s, progresses through the intermediate runnables denoted as r_{m*}, and culminates with the end runnable r_e. We consider chain models in two ways as typically used in automotive systems: (1) time-triggered, where all runnables are triggered by a timer event, and (2) event-triggered, where the start runnable is time-triggered, and all subsequent runnables in the chain are triggered by the completion of the preceding runnable. We assume that the priority of a chain, π_{Γ^c}, is assigned by the system designer based on its criticality or importance within the system. Chains can also share a mutual (joint) runnable, which our proposed framework supports.

End-to-end latency. Based on the above the chain model, end-to-end latency refers to the duration between the release of the first element in the chain(r_s) and the completion of the last element(r_e).

IV. Challenges

In this section, we elaborate on the challenges of the current AUTOSAR-based scheduling behavior as identified by practitioners in the automotive industry.

Challenge 1. Shortcomings of resource allocation policy. One of the primary challenges is the inadequacy of the resource allocation policy in AUTOSAR. The current AUTOSAR standard does not provide comprehensive guidelines for the resource allocations such as mapping runnables to OS-tasks and assigning OS-tasks to available CPU cores. Without clear policies, system designers often struggle to optimally utilize the available resources, leading to potential underutilization or overutilization of system resources. The manual process is time-consuming for system designers and prone to human errors, which can compromise system reliability and performance. Resource allocation becomes even more challenging when multiple runnables that belong to different functional chains need to be assigned to multiple OS-tasks with various priority levels.

Challenge 2. Unpredictable timing behavior of the runnables. Automotive functions necessitate that designers comply with stringent timing requirements, particularly for safety-related functions governed by standards such as ISO 26262. However, several factors contribute to the unpredictability and unique scheduling of runnable execution in AUTOSAR systems. These factors include varied activation times triggered by external events and complex data dependencies within runnable chains. A comparison of these scheduling behaviors will be examplified in Section VI-C .

V. Revisiting Priority-Driven Chain-Aware Scheduling for ROS 2

In this section, we review the previously proposed chain-aware scheduling framework for ROS 2 [5]. Since the framework cannot be directly applied to the AUTOSAR environment, we revisit and verify the fixed concepts to adapt them for use in the AUTOSAR environment.

Lemma 1 in [5]. ROS 2 was re-designed in [5] based on the following two principles: (1) higher-priority chains should execute earlier than low-priority chains, and (2) for a single chain, a prior instance of the chain should be completed before the newly released instance starts its execution. As explained earlier in Section III, callbacks and executors in ROS 2 correspond to runnables and OS-tasks in AUTOSAR, respectively. Therefore, we employ these principles of Lemma 1 in [5] directly to our work.

A. Runnable scheduling strategies

While timer callbacks and regular callbacks are managed separately from the system-level definition to their operation across ROS 2 abstraction layers, all runnables in AUTOSAR can be handled in the same manner except for the setting of the RTE event, which can be either time-triggered or event-triggered. Hence, we simplify the six callback scheduling strategies from [5] to four runnable scheduling strategies as follows:

We first describe two runnable scheduling strategies within an OS-task.

- **Runnables from a single chain.** For an OS-task containing runnables from a single chain Γ^c, the priorities of the runnables are assigned in reverse order of their sequence in the chain.
- **Runnables from multiple chains.** For an OS-task containing runnables from multiple chains, Γ^c and $\Gamma^{c'}$, where

$\pi_{\Gamma^c} < \pi_{\Gamma^{c'}}$, all runnables of $\Gamma^{c'}$ should be assigned higher priorities than those of Γ^c.

Then, runnables across OS-tasks can be executed as the following two strategies.

- **A single chain on one CPU.** When a CPU has runnables from only a single chain Γ^c, the OS-task containing the lower-index runnables should have the same or lower priority than the OS-tasks on the same CPU that execute the higher-index runnables of Γ^c.
- **Multiple chains on one CPU.** when a CPU has runnables from multiple chains, Γ^c and $\Gamma^{c'}$, where $\pi_{\Gamma^c} < \pi_{\Gamma^{c'}}$, the OS-task that contains the runnables of $\Gamma^{c'}$ should have at least the same or higher priority than those containing the runnables of Γ^c

VI. Proposed runnable scheduling frameworks

We propose two runnable scheduling frameworks for the AUTOSAR environment based on our modified scheduling strategies: (1) *Event-Triggered Runnable Scheduling*, and (2) *Chain-Aware Runnable Scheduling*.

A. Event-triggered runnable scheduling

Event-triggered runnable scheduling leverages AUTOSAR's extended OS-task configuration. The initial runnable in a chain is time-triggered, releasing and executing periodically using a timer. All subsequent runnables are triggered by the completion of their preceding runnable within the same chain.

Given the lack of official guidelines for resource allocation and execution order of schedulable entities such as runnables and OS-tasks, we propose a comprehensive resource allocation scheme. This scheme aims to execute chains as independently as possible, thereby reducing interference between chains. It also ensures efficient utilization of resources and improves the predictability of task execution.

Note that this approach does not require any modification of the existing AUTOSAR framework, making it compatible with current systems while enhancing their performance. However, it cannot strictly satisfy scheduling principles described in Section V. We address this issue by proposing a new chain-aware scheduler in the next subsection.

Listing 1 describes the extended OS-task code template for the event-triggered runnable scheduler. This OS-task retrives the OS events, which are defined as a bit pattern in a data type `EventMaskType`. Each bit represents a specific event, enabling the OS to execute all ready runnables in order after checking their respective bits by masking flags, such as e.g., `r1_flag` within the loop. After executing runnables, it clears all OS events and conducts the termination process.

```
TASK (ExampleTask)
{
    EventMaskType evt;
    while(1)
    {
        OSWaitEvent();
        OSGetEvent(&evt);
        OSClearEvent(evt);

        // Execution of Runnable1
        if (evt & r1_flag) == r1_flag
        {
            Runnable1();
        }
        // Execution of Runnable2
        if (evt & r2_flag) == r2_flag
        {
            Runnable2();
        }
        // Terminate task to let the OS schedule the next
            task
        evt = RTE_CLEAR;
        TerminateTask();
    }
}
```

Listing 1: Extended OS-task code

Priority assignment of runnables. The priority assignment for runnables adheres to the scheduling strategies explained in Section V. First, chains are sorted in ascending order of their semantic priorities. For each chain, runnables are then assigned priorities such that a runnable with a lower index gets a lower priority. Thus, the start runnable of a chain gets the lowest priority within the chain, while the end runnable gets the highest priority.

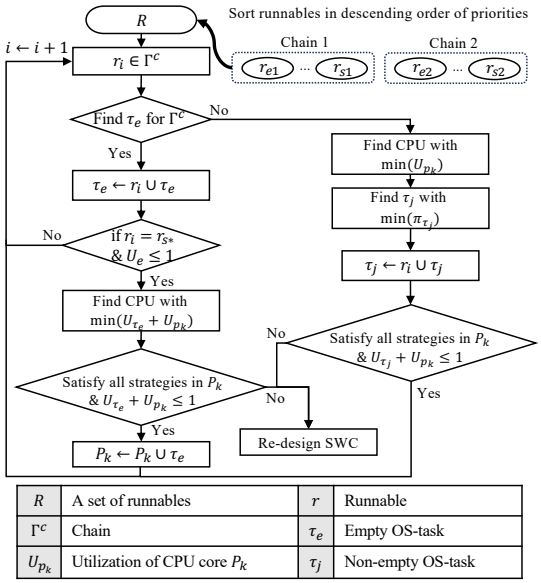
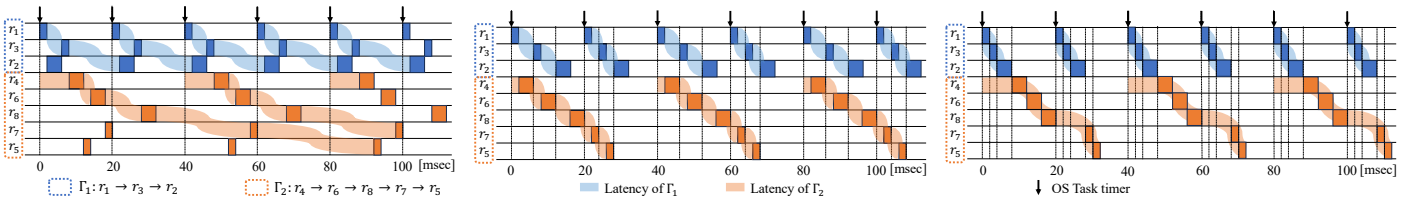


Fig. 2: Diagram of runnable allocation algorithm

Runnable allocation We present a runnable allocation scheme for AUTOSAR. The proposed algorithm allocates runnables into OS-tasks and available CPU cores, taking into account the scheduling strategies described in Section V. Fig 2 illustrates the diagram of the proposed runnable allocation algorithm. The allocation is performed in a manner that separates the chains as much as possible to reduce interference between them. This scheme allocates higher-priority runnables of critical chains first to available OS-tasks, then maps them onto the CPU core where the utilization is the minimum. When an OS-task is assigned to a CPU core, the scheme verifies the scheduling strategies and the sum of the utilization, i.e., the utilization should be less than 1. If the conditions are not met, system designers are guided to re-design software components. When there are no empty OS-tasks available, the scheme finds a non-empty OS-task from the CPU core with the minimum utilization and allocates the runnable to the OS-task with the lowest priority, ensuring compliance with the scheduling



(a) Time-triggered runnable scheduling (b) Event-triggered runnable scheduling (c) Chain-aware runnable scheduling

Fig. 3: Examples of runnable scheduling

TABLE I: Runnable sets

	Chains				
Chain 1	Γ^1 =: $[r_1, r_3, r_2]$				
Chain 2	Γ^2 =: $[r_4, r_6, r_8, r_7, r_5]$				
Runnables	T [ms]	C [ms]	Runnables	T [ms]	C [ms]
r_1	20	2	r_5	40	2
r_2	20	4	r_6	40	4
r_3	20	2	r_7	40	2
r_4	40	4	r_8	40	4

strategy requirements. It is worth noting that mutual runnables can be considered part of the highest priority chain.

B. Chain-aware runnable scheduling

Now, we propose a chain-aware runnable scheduling framework for AUTOSAR. To realize Lemma 1 and runnable scheduling strategies within the AUTOSAR environment, modifications to the extended OS-task are necessary because the default extended OS-task cannot execute runnables in their priority orders, as will be exemplified in Section VI-C. Listing 2 briefly describes the modified extended OS-task structure. Compared to the default extended OS-task, the modified structure executes the highest-priority runnable first and then conducts a termination process, clearing the register bit associated with the runnable that just executed. This enables the second highest-priority runnable to be executed in the next loop. Hence, this structure ensures the correct execution order and strictly satisfies Lemma 1, thereby improving the latency of safety-critical chains by preventing unwanted interference.

```
...
    // Execution of Runnable1
    if (evt & r1_flag) == r1_flag
    {
        Runnable1();
        OSClearEvent(evt & r1_flag);
        TerminateTask();
    }
    // Execution of Runnable2
    if (evt & r2_flag) == r2_flag
    {
        Runnable2();
        OSClearEvent(evt & r2_flag);
        TerminateTask();
    }
...
```

Listing 2: Modified extended OS-task code

C. Example of runnable schedulings

We run three different runnable scheduling approaches: time-triggered (3a), event-triggered (3b), and chain-aware runnable scheduling (3c). As described in Table I, runnable set comprises two chains with 3 and 5 runnables, respectively. The chains have periods (T) of 20 or 40 ms and execution time (C) of 2 or 4 ms.

We assume that all runnables are allocated to a single OS-task and the index of a runnable represents the order of execution within the task, i.e., r_i has higher priority than r_j where $i < j$. Besides, chain 1 is more critical than chain 2.

As shown in Fig 3a, chains 1 and 2 experience the maximum end-to-end latencies of 26 ms and 92 ms, respectively, when all runnables are scheduled by a default time-triggered runnable scheduling. Under the event-triggered runnable scheduler, latency of chains is 16 ms and 28 ms, respectively, and 8 ms and 32 ms under the chain-aware runnable scheduler. This is because the proposed chain-aware scheduler strictly prioritizes safety-critical chains, significantly improving latency of chain 1, e.g., reducing by 69% compared to time-triggered runnable scheduler (AUTOSAR default).

VII. Analysis of End-to-End Latency

This section presents the end-to-end latency analysis of a chain for three different runnable scheduling approaches: time-triggered, event-triggered, and chain-aware schedulers.

A. Latency analysis for time-triggered runnable scheduler

The analysis of end-to-end latency for time-triggered runnable scheduler can be conducted in two steps: (i) calculating the lower bound of the start time ($LB^s_c(i)$) and the upper bound of the finish time ($UB^f_c(i)$) of a runnable r_i as described in [6], and (ii) identifying the chain instance that has the maximum end-to-end latency based on the time bounds obtained in the previous step.

In a first step, we re-write the lower bound of the start time and the upper bound of the finish time of a runnable r_i of Γ^c to fit our system model as below:

Lower bound of start time of r_i in [6].

$$LB^s(i) = \sum_{k=1}^{i-1} C_k + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{\delta_h}{T_{\tau_h}} \right\rceil \times \mathcal{C}_h \tag{1}$$

where $hp(\tau_c) = \{\tau_h | m_h = m_c, \pi_h > \pi_c\}$, with m_h representing the CPU core to which τ_h is allocated, $\mathcal{C}_h = \sum_{k=1}^{|\tau_h|} C_k$ denoting the total execution time of all runnables in τ_h, and $\delta_h = max(0, LB^s(i) - (T_{\tau_h} - \mathcal{C}_h) + 1)$.

Upper bound of finish time of r_i in [6].

$$UB^f(i) = \sum_{k=1}^{i} C_k + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{UB^f(i)}{T_{\tau_h}} \right\rceil \times \mathcal{C}_h \tag{2}$$

Note that the first terms of the above Equations represent the execution time of runnable r_k of chain Γ^c.

Algorithm 1 Maximum end-to-end latency of Γ^c

```
1: Input: Γ^c, ∀τ ∈ Γ^c
2: Output: max(latency)
3: latency ← []                          ▷ An empty array to save latency.
4: s_flag ← true
5: cnt ← 0
6: for all r_i ∈ Γ^c do                  ▷ Check a chain in a single OS Task.
7:     if r_i ∉ τ_c then
8:         s_flag ← false
9:     end if
10: end for
11: if s_flag then
12:     for all ∀(r_i, r_j) ∈ Γ^c, i < j do
13:         if π_{r_i} < π_{r_j} then
14:             cnt ← cnt + 1
15:         end if
16:     end for
17:     latency ← UB^f(|Γ^c|) + cnt × T_τ
18: else
19:     s ← LB^s(1)
20:     for t = s to UB^f(1) − C_{r_1} do
21:         f ← t + C_{r_1}
22:         for all r_i, 1 < i < |Γ^c| do
23:             f ← find_finish_time(f, r_i)
24:         end for
25:         latency ← latency ∪ f
26:     end for
27: end if
```

In a second step, we propose an algorithm to find the maximum end-to-end latency among all possible chain instances. Algorithm 1 describes the process of finding the maximum end-to-end latency of a chain under time-triggered runnable scheduling. It begins by determining whether all runnables of the target chain are allocated into a single OS-task or not (line 6 to 10). If a chain uses only a single OS-task, the algorithm counts the number of reversed orders of runnables by comparing their execution order (i.e., priorities of runnables π_{r_i}) with the indices of any two runnables within the chain (line 14). The latency is then calculated by adding the upper bound of the finish time of the last runnable in the chain and the `count` multiplied by the period of the OS-task to which the runnables belong (line 17). If the runnables of a chain are distributed across multiple OS-tasks, the algorithm investigates all possible end-to-end latencies based on the time bounds of a runnable obtained in the first step (line 19 to 25). The function `find_finish_time` iteratively finds the nearest completion time (f) of the next runnable (r_i) by adding its period. This process returns a possible latency after searching through all runnables of Γ^c (line 22 to 24). Finally, the maximum latency is chosen from all the potential latencies that the chain could experience.

B. Latency analysis for event-triggered and chain-aware runnable scheduling

In this section, we first propose an end-to-end latency analysis method for chain-aware scheduling. Then, we utilize this technique to analyze the latency of a chain in the event-triggered scheduler by reconsidering the interference affecting the target chain.

Chain-aware scheduling. To prevent undesirable latency increases, our chain-aware runnable scheduler strictly adheres to the priority conditions of runnables and OS-tasks under the chain configuration, as outlined in Lemma 1 of [13] for callbacks and executors in ROS 2. Therefore, we adapt the end-to-end latency analysis of [5] directly by adjusting the system model for AUTOSAR.

The analysis of end-to-end latency for the chain-aware runnable scheduler can be conducted in two steps as described in [5]: (i) computing the worst-case response time (WCRT) of each segment (i.e., a consecutive subset of a chain Γ^c on one CPU core) of a chain as per Lemma 3 of [5], and (ii) summing up all WCRTs of all segments of a chain as per Theorem 1 of [5].

We first re-write Lemma 3 from [5] to align with the system models used in our work:

Lemma 1 (WCRT [5])**.** *The worst-case response time of a segment $\Phi_i \subset \Gamma^c$, denoted by $R^n_{c,i}$, is bounded by the following recurrence:*

$$R^{n+1}_{c,i} \leftarrow B_i + \sum_{\forall j: r_j \in \Phi_i} C_j + \sum_{\substack{\forall k: r_k \in \tau(\Phi_i) \vee \\ r_k \in \tau_{HP}}} \eta_i(R^n_{c,i}, r_k) \times C_k \tag{3}$$

where B_i is the blocking time from lower priority runnable (Equation (2) in [5]), η_i is the maximum number of arrivals of a runnable that causes interference to a target segment Φ_i (Lemma 2 in [5]), and τ_{HP} is a set of OS-tasks with higher priority than the OS-task of Φ_i. The recurrence starts with $R^0_{c,i} = B_i + \sum_{\forall j: r_j \in \Phi_i} C_j$.

Then, the end-to-end latency of a chain Γ^c is computed by:

Theorem 1 (End-to-end latency [5])**.**

$$L_{\Gamma^c} \leftarrow \sum_{\Phi_i \subset \Gamma^c} R^n_{c,i} + S(\Gamma^c) \tag{4}$$

where Φ_i is a segment of the chain Γ^c and $S(\Gamma^c)$ is the maximum blocking delay caused by a prior instance of the chain provided in Lemma 4 of [5].

Due to the nature of overwriting the mask flag in the extended OS-task structure for runnable execution (as explained in Section VI-B), when the worst-case response time of a segment of a chain exceeds the period of the chain, the next instance can be delayed by at most one cycle of the chain's period. This behavior is similar to the overload handling mechanism of ROS 2. Therefore, Theorem 1 remains valid for our proposed chain-aware runnable scheduler.

Event-triggered scheduling. Under the default extended OS-task structure, runnables that belong to any prior instance of all chains can interfere with the execution of a runnable for the target segment, which enables Lemma 1 not valid for the event-triggered scheduler. Therefore, we modify the Lemma 2 in [5] as below:

Corollary 1.1. *The maximum number of arrivals of a runnable $r_k \in \Gamma^{c'}$ that causes interference to a target segment $\Phi_i \subset \Gamma^c$*

during an arbitrary time window t is bounded by:

$$\eta_i(t, r_k) = \begin{cases} \left\lceil \frac{t}{T_{\Gamma^{c'}}} \right\rceil & , \textit{if } \pi_{\Gamma^{c'}} \geq \pi_{\Gamma^c} \wedge P(\Phi_i) = P(\tau_k) \\ 0 & , \textit{otherwise} \end{cases} \quad (5)$$

where $T_{\Gamma^{c'}}$ is a period of chain $\Gamma^{c'}$.

Proof. Since any runnable (r_k) in a higher- or the same priority OS-task on the same CPU core can interfere with the target segment, they are considered sources of interference. Thus, the proof is done. □

VIII. Evaluation

This section evaluates our proposed runnable scheduling frameworks by comparing them with existing AUTOSAR runnable scheduling approaches. We begin with an industrial case study inspired by current automotive systems. Then, we perform an end-to-end analysis using randomly generated workloads to investigate the performance characteristics across various experimental setups.

Comparison of approaches. We compare three runnable scheduling approaches. The first approach is AUTOSAR-default Runnable Scheduling (ADRS), which uses only a time-triggered chain configuration between runnables, equipped with the latency analysis framework explained in Section VII-A. The second approach is Event-Triggered Runnable Scheduling (ETRS), which utilizes the extended OS-task of AUTOSAR to enable the triggering of runnables by external events, with the latency analysis framework explained in Section VII-B. Lastly, Chain-Aware Runnable Scheduling (CARS) is our proposed scheduling approach, with the latency analysis described in Section VII-B.

A. Case study

A case study was conducted on a commercial Vehicle Control Unit (VCU) equipped with Infineon TC3xx microcontroller commonly used in automotive industry. We used the Matlab AUTOSAR toolbox to configure the software component (SWC) layers and employed the Mobilgene AUTOSAR platform tool provided by Hyundai AutoEver to edit other layers, such as RTE and BSW, and flashed the generated binary onto the VCU. To measure the latency of chains, we embedded timestamp code at the beginning and the end of all runnable executions. These timestamps were transmitted to the host laptop via the VCU's CAN communication interface.

The scenario of the case study is inspired by the highest-level controller, which is responsible for managing motor torque, vehicle start, and various other functional controls of a vehicle, as listed in Table II. It involves 40 runnables, with a total utilization (U) of 0.52, organized into 8 real-time chains where lower-indexed chains have higher priorities, such as torque control (Γ^1) and emergency stop (Γ^2). Each chain comprises 5 runnables. We assume the availability of 5 OS-tasks and a single CPU core. Since ADRS does not have a specific resource allocation guideline, we used the worst-fit decreasing (WFD) approach to allocate runnables to OS-tasks. For ETRS and CARS, the first 4 chains areß assigned to higher-priority OS-tasks separately, while the remaining chains are all assigned to the last OS-task based on our proposed allocation scheme.

TABLE II: Case study chain configuration

	Chains
Chain 1	Γ^1 =: $[r_1, r_3, r_2, r_4, r_5]$
Chain 2	Γ^2 =: $[r_9, r_6, r_7, r_6, r_{10}]$
Chain 3	Γ^3 =: $[r_{15}, r_{11}, r_{12}, r_{14}, r_{13}]$
Chain 4	Γ^4 =: $[r_{17}, r_{18}, r_{16}, r_{20}, r_{19}]$
Chain 5	Γ^5 =: $[r_{23}, r_{21}, r_{22}, r_{24}, r_{25}]$
Chain 6	Γ^6 =: $[r_{26}, r_{28}, r_{27}, r_{29}, r_{30}]$
Chain 7	Γ^7 =: $[r_{34}, r_{32}, r_{33}, r_{31}, r_{35}]$
Chain 8	Γ^8 =: $[r_{37}, r_{38}, r_{36}, r_{40}, r_{39}]$

Runnables	T [ms]	C [ms]	Runnables	T [ms]	C [ms]
$r_1 \sim r_5$	10	0.128	$r_{21} \sim r_{25}$	10	0.096
$r_6 \sim r_{10}$	10	0.128	$r_{26} \sim r_{30}$	10	0.192
$r_{11} \sim r_{15}$	10	0.096	$r_{31} \sim r_{35}$	10	0.192
$r_{16} \sim r_{20}$	10	0.12	$r_{36} \sim r_{40}$	10	0.08

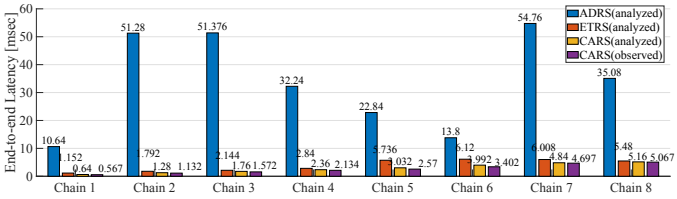


Fig. 5: Case study

Fig 5 shows the results of observed and analyzed latencies of 8 chains. As expected, CARS significantly outperforms the other approaches in terms of end-to-end latency, providing tight upper bounds for most chains. It is worth noting the release time of chain instances only can be validated under CARS. This is because the first runnable of chain 1, which is allocated the highest-priority OS-task, always starts execution at its release time, which is not guaranteed by other approaches.

B. End-to-end latency experiments

In this subsection, we conduct all experiments on a machine equipped with an Intel Xeon W-2295 3.0 GHz processor and 32GB of memory, running on Ubuntu.

Workload generation. We use 1,000 randomly generated runnable workload sets for each experimental setting, with the utilization of each workload set ranging from 0.2 to 0.85. For a workload set, each runnable's utilization is obtained using the UUniFast algorithm [3]. Motivated by current commercial automotive systems and a prior benchmarks work [9], the runnable period is chosen randomly from the set 10, 20, 50, 100, 1000 ms. Each runnable workload set consists of 6 chains, with the lower-indexed chain being more critical than the others, and each chain comprising 3 runnables. The order of runnables in each chain is randomly selected.

Comparison of the average end-to-end latency of chains. The results of the average end-to-end latency for 6 chains at each utilization are shown in Fig 4. In this experiment, we assume that all runnables are allocated into a single OS-task and mapped onto a single CPU core. The average end-to-end latency increases as the utilization increases for all approaches. CARS significantly outperforms the other two approaches,

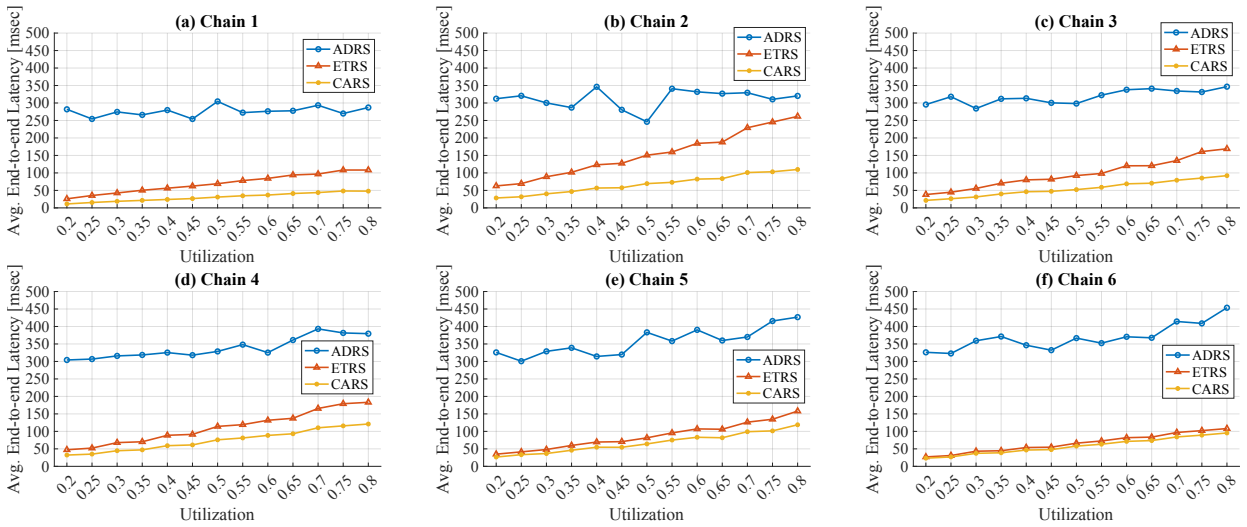


Fig. 4: Average end-to-end latency of chains by utilization

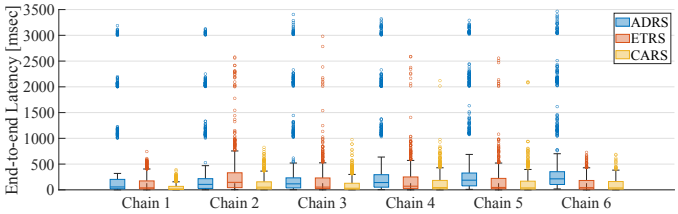


Fig. 6: Distribution of latencies ($U = 0.8$)

specifically reducing latency by 82% for the most critical chain (i.e., chain 1) when the utilization is 0.8. Additionally, CARS strictly prioritizes chains based on their criticality, resulting in a diminishing difference between ETRS and CARS as the criticality of chains decreases, i.e., from chain 1 to chain 6.

Fig 6 shows the distributions of end-to-end latency of 1,000 runnable sets when each set has a utilization of 0.8. As expected, CARS significantly reduces the end-to-end latency compared to the other approaches.

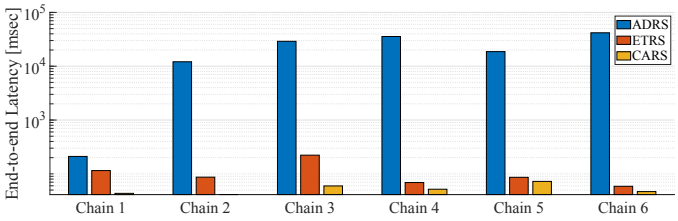


Fig. 7: Multiple OS-tasks and CPU cores setup

Multiple OS Tasks and CPU Cores. In this experiment, we generate a workload set with a utilization of 1.3 and use two OS-tasks and two CPU cores. Since ADRS does not have any official resource allocation guidelines, we used the worst-fit decreasing (WFD) approach, while ETRS and CARS used the proposed resource allocation scheme. As shown in Fig 7, CARS significantly outperforms the other two approaches in terms of end-to-end latency for all 6 chains. This is because the proposed resource allocation framework distributes runnables based on their chain configuration, thereby minimizing interference between them. On the other hand, ADRS exhibits very high latency results compared to the other two approaches because WFD does not consider the chains' structure.

IX. Conclusion and Future Work

In this paper, we addressed the challenge of managing end-to-end latency performance in AUTOSAR-based automotive systems. We introduced a novel chain-aware runnable scheduling framework that enhances the efficiency of runnable execution by prioritizing runnables and optimizing their allocation to OS-tasks and CPU cores. Our approach, inspired by existing methodologies, was tailored specifically for the AUTOSAR environment. As future work, we aim to explore further refinements and extensions of our framework, including its integration with emerging self-driving technologies and related applications.

References

[1] Home Autosar. https://www.autosar.org, accessed June 2024.

[2] M. Becker et al. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *RTCSA*. IEEE, 2016.

[3] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-time systems*, 30(1):129–154, 2005.

[4] H. Choi, M. Karimi, and H. Kim. Chain-based fixed-priority scheduling of loosely-dependent tasks. In *ICCD*. IEEE, 2020.

[5] H. Choi, Y. Xiang, and H. Kim. Picas: New design of priority-driven chain-aware scheduling for ros2. In *RTAS*. IEEE, 2021.

[6] J. Choi, D. Kang, and S. Ha. End-to-end latency analysis of cause-effect chains in an engine management system. In *2018 IEEE DATE*.

[7] O. Group et al. Osek/vdx operating system specification 2.2. 3, 2005.

[8] M. Günzel et al. Timing analysis of asynchronized distributed cause-effect chains. In *RTAS*. IEEE, 2021.

[9] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *WATERS*, volume 130, 2015.

[10] A. Monot et al. Multisource software on multicore automotive ecus - combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 2012.

[11] N. Navet et al. Multi-source and multicore automotive ecus-os protection mechanisms and scheduling. In *ISIE*. IEEE, 2010.

[12] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *IEEE RTSS*.

[13] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *RTSS*. IEEE, 1998.

[14] H. Zeng and M. Di Natale. Efficient implementation of autosar components with minimal memory usage. In *SIES*. IEEE, 2012.

[15] H. Zeng, M. Di Natale, and Q. Zhu. Optimizing stack memory requirements for real-time embedded applications. In *ETFA*. IEEE, 2012.