CHALMERS
UNIVERSITY OF TECHNOLOGY

Integrating OSEK RTOS in L2 Virtual ECU for Simulating Timing Behavior of Automotive Systems

Master's thesis in Embedded Electronic System Design

Pengfei Chen & Wuyang Hao

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's thesis 2025

Integrating OSEK RTOS in L2 Virtual ECU for Simulating Timing Behavior of Automotive Systems

Pengfei Chen & Wuyang Hao

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2025

Integrating OSEK RTOS in L2 Virtual ECU for Simulating Timing Behavior of Automotive Systems
Pengfei Chen & Wuyang Hao

Supervisor: Risat Pathan, Computer Science and Engineering
Company advisor: Christian Axbrink, Volvo Technology AB
Examiner: Per Larsson-Edefors, Department of Microtechnology and Nanoscience

Master's Thesis 2025
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Integrating OSEK RTOS in L2 Virtual ECU for Simulating Timing Behavior of Automotive Systems
Pengfei Chen & Wuyang Hao
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Modern vehicles rely on Electronic Control Units (ECUs) to manage safety-critical functions such as engine control and braking. To reduce hardware dependency and accelerate development, Virtual ECUs (vECUs) are used for early testing, though many lack accurate real-time behavior. This thesis upgrades a Level 2 vECU to Level 3 by integrating Trampoline, an open-source OSEK-compliant real-time operating system (RTOS), enabling fixed-priority preemptive scheduling. The modified vECU runs in a real-time Linux environment and is evaluated against a production ECU using timing metrics like execution time and response time. Two response time analysis (RTA) models—classic and offset-aware—are applied to predict worst-case response times, which are then validated against empirical measurements. Since RTA is essential for ensuring tasks meet deadlines in automotive systems, validating these models enhances their practical utility. The results show that the upgraded vECU can simulate timing behavior with reasonable accuracy, supporting cost-effective, early-stage validation in automotive software development.

Keywords: ECU, vECU, RTOS, OSEK, Real-time, RTA

Acknowledgements

We would like to express our sincere gratitude to our supervisors, Christian Axbrink at Volvo Technology AB and Risat Pathan from the Department of Computer Science and Engineering, for their invaluable guidance, constructive feedback, and continuous support throughout the course of this thesis. Their expertise has greatly contributed to the depth and clarity of our work.
We also extend our thanks to our colleagues at Volvo Technology AB for their assistance, collaborative spirit, and insightful discussions, which helped shape various aspects of this project.
Finally, we acknowledge the contributions of the open-source community, particularly the developers of the Trampoline RTOS and related tools, whose work provided the foundation for our technical implementation.

Pengfei Chen & Wuyang Hao, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

The increasing complexity of modern automotive systems has driven the need for accurate, high-fidelity simulation tools capable of modeling the behavior of Electronic Control Units (ECUs). ECUs are responsible for critical tasks such as engine control, braking, transmission, and infotainment. As vehicles transition toward greater levels of automation, the ability to simulate ECUs reliably is becoming indispensable. Simulations enable iterative development and testing without the need for physical hardware, thereby reducing cost and accelerating design cycles. Virtual ECUs (vECUs) provide a scalable and efficient approach to prototyping and validation, offering flexibility in software development and system integration [2].

vECUs have seen increasing adoption in both industry and academia as a means to accelerate automotive software development and reduce hardware dependence [3]. Commercial tools such as dSPACE VEOS and ETAS ISOLAR-EVE enable software-in-the-loop (SiL) testing and integration, while academic efforts have explored vECU applications in model-based design, real-time simulation, and embedded system verification. These developments highlight the growing role of vECUs in supporting flexible, scalable testing workflows. However, most existing solutions focus primarily on functional correctness and lack support for precise timing behavior, limiting their use in time-critical validation scenarios.

This thesis aims to evaluate the feasibility of enhancing an existing Level 2 (L2) vECU (basic functional simulation without production software) to a Level 3 (L3) vECU (basic software and hardware-independent application) by integrating an open-source *Open Systems and their Interfaces for the Electronics in Motor Vehicles* (OSEK)-compliant real-time operating system (RTOS). Besides, another aim is to determine whether such an upgrade enables more realistic modeling of ECU behavior, particularly in terms of real-time task execution and scheduling. Furthermore, the ultimate goal of this work is to investigate whether an upgraded L3 vECU, when paired with a theoretical scheduling model, can effectively emulate the timing behavior of a production ECU, which could reduce the reliance on more costly and complex L4 vECUs or physical ECUs in development and verification workflows.

To achieve the aims, this thesis implements the upgrade of L2 vECU to L3 vECU by integrating Trampoline, an open-source RTOS. The upgraded L3 vECU is assessed using established real-time metrics, such as execution time, response time, and release jitter. The thesis also deploys two theoretical models grounded in real-time scheduling theory, specifically the rate-monotonic (RM) analysis, which is used to predict the timing behavior of the target ECU system. Measurements from both

hardware and the upgraded vECU are collected and compared against the theoretical predictions. Discrepancies between the model, vECU behavior, and hardware measurements are analyzed to identify the limits of abstraction in virtual simulations.

The results of this thesis have direct applicability in the automotive sector. By improving the real-time accuracy of virtual platforms, developers can use the upgraded L3 vECU for early validation of timing behavior—long before physical ECUs are available. This enables early detection of deadline violations and scheduling issues, reducing the need for costly late-stage fixes. Moreover, the integration of theoretical timing models into the simulation workflow allows developers to predict and reason about system performance analytically. Ultimately, the methods and tools developed in this thesis can support safer, faster, and more cost-effective development of embedded automotive software.

1.1 Related Work

This research builds upon a convergence of three key domains: OSEK-based RTOS implementations, real-time scheduling, and virtualized environments for automotive testing, such as vECUs. Although each area has seen independent progress, their intersection—especially the integration of real-time scheduling policies like RM analysis within vECUs leveraging OSEK RTOS—remains underexplored.

OSEK RTOS in Embedded Automotive Systems

The OSEK RTOS specification [4] has long served as a foundation for embedded software in automotive ECUs. Early implementations like Trampoline [5] provided open-source references for deploying OSEK RTOS on physical hardware platforms. The study [6] further investigated task-switching latency within OSEK RTOS environments, revealing critical timing behavior essential for safety-critical systems. However, these studies predate the rise of virtualization in automotive software testing and lack exploration of how OSEK RTOS behaves in virtualized environments such as vECUs. This thesis addresses that gap by adapting and deploying an OSEK-compliant RTOS within L3 vECUs, evaluating their behavior under constrained real-time requirements.

Virtual ECUs and Software-in-the-Loop Testing

vECUs are increasingly seen as a means to accelerate development cycles and reduce testing costs. The 2024 study by Keil et al. [7] emphasized the benefits of shifting from hardware-in-the-loop (HiL) to software-in-the-loop (SiL) testing, identifying test cases that can migrate to virtual environments. Similarly, the Prostep IVIP white paper [2] outlines the standards and requirements for vECU integration, emphasizing modularity and reuse.

However, these works tend to view vECUs as approximations rather than strict real-time emulators. Ågren et al. [8] highlight impediments such as insufficient timing guarantees and synchronization issues in current virtual verification ecosystems. This thesis builds upon such findings by evaluating whether vECUs, when inte-

grated with an OSEK-compliant RTOS and paired with formal real-time scheduling strategies, can deliver the timing fidelity required for real-world control applications.

Real-Time Scheduling in Automotive ECUs

Real-time scheduling has been pivotal to ensuring system determinism and predictability. Foundational work by Liu and Layland [9] laid the groundwork for fixed-priority RM scheduling, while Stankovic et al. [10] expanded the field with dynamic scheduling via earliest-deadline-first (EDF). Xu et al. [11] introduced harmonic task scheduling and control co-design to optimize processor utilization and reduce jitter, which is particularly relevant for feedback control tasks in automotive systems.

Recent work by Lim et al. [12] compares RM and EDF scheduling strategies within multi-core ECUs, illustrating how each algorithm performs under periodic and aperiodic task sets. However, their study focuses on physical ECUs and omits their applicability in virtualized platforms. Additional studies like [13, 14, 15, 1] extend analysis techniques for worst-case response time, supporting more accurate evaluations of real-time behavior in complex systems. These methods are crucial for assessing scheduling feasibility in both physical and virtual ECUs.

In this thesis, we apply two such models—the classic response time analysis and an offset- and jitter-aware model—to evaluate their predictive accuracy on a real ECU and an upgraded vECU. By comparing model predictions with measured timing behavior, we assess how well these theoretical models reflect actual execution under preemptive scheduling. Our findings not only validate their applicability in virtualized environments but also highlight practical limitations, offering insights for future refinements of model-based timing validation in automotive systems.

1.2 Purpose and Goal

Purpose

The central purpose of this thesis is to evaluate the feasibility of enhancing an existing L2 vECU to an L3 vECU by integrating a preemptive RTOS and deploying a theoretical model to predict real-time task behavior. This work is motivated by the increasing reliance on vECUs for software testing in automotive systems, which reduces development costs and time compared to traditional hardware-based methods. Enhancing vECU fidelity to reflect realistic task execution is critical for advancing autonomous vehicle software validation. Moreover, this integration provides a non-commercial solution for upgrading a vECU from L2 to L3 using an open-source RTOS for industry.

Goal

- **Primary Goal:** Upgrade a basic L2 vECU by integrating a real-time, OSEK-compliant RTOS to enable preemptive scheduling, then evaluate its timing behavior through measurements and compare it with theoretical response-time models to assess accuracy in an OSEK+vECU context.

- **Objectives:**
  - Integrate an open-source OSEK RTOS into the L2 vECU to enable preemptive scheduling based on RM priority assignment. This allows the virtual platform to more accurately reflect real-world ECU scheduling behavior and support time-critical applications.
  - Measure and analyze real-time task behavior in both virtual and physical ECUs. These measurements are essential for validating the realism of the upgraded vECU and establishing a basis for model comparison.
  - Apply existing response-time analysis (RTA) theoretical models including offset-aware variants, to account for task offsets and release jitter. Incorporating advanced models enhances the accuracy of timing predictions under real-world conditions.
  - Evaluate the accuracy of the applied models by comparing their predictions to empirical measurements from both the physical ECU and the vECU. This step verifies whether theoretical models are reliable tools for simulating real-time performance in virtual platforms.
  - Analyze discrepancies between model predictions and measurements to assess model limitations and suggest refinements. Understanding these gaps helps improve simulation fidelity and guides future enhancements in timing analysis and vECU design.

1.3 Thesis Outline

This thesis is organized into seven chapters, addressing introduction, technical background, methods, design, results, discussion, conclusion.

- **Introduction:** Presents a brief introduction to the thesis topic.
- **Technical Background:** Presents an overview of the key concepts and technologies relevant to this thesis.
- **Methods:** Presents approaches adopted to achieve the objectives of this thesis.
- **Design:** Presents system design and implementation for integrating a preemptive RTOS into the vECU, configuring real-time scheduling, and establishing a platform for measurement and validation.
- **Results:** Presents empirical and theoretical timing analyses of periodic tasks on both the ECU and vECU platforms.
- **Discussion:** Interprets the results, analyzes discrepancies between platforms and models, and reflects on the implications and limitations of the study.
- **Conclusion:** Summarizes the main findings, evaluates the feasibility of the RTOS-integrated vECU, and outlines directions for future improvement.

2

Technical Background

This chapter provides an overview of the key concepts and technologies relevant to this thesis, including ECUs, vECUs, OSEK RTOS, and real-time scheduling theory. The overview establishes the foundation for upgrading an L2 vECU to an L3 vECU and developing a theoretical real-time scheduling model to predict ECU behavior.

2.1 ECU in Automotive Systems

Modern vehicles rely on a distributed architecture consisting of dozens to over a hundred ECUs, each responsible for managing specific vehicle functions such as engine control, braking, transmission, body electronics, infotainment, and increasingly, advanced driver-assistance systems (ADAS) [16]. These embedded systems are highly specialized and operate under stringent real-time constraints to ensure functional safety, reliability, and performance in dynamic driving environments.

As vehicle features and software complexity continue to grow—driven by trends such as electrification, connectivity, and automation—the number and interdependence of ECUs have significantly increased [17]. A typical mid-range vehicle today contains between 70 and 100 ECUs, while high-end or electric vehicles may exceed 150. However, this distributed architecture introduces several challenges. These include excessive hardware redundancy, increased vehicle weight and cost due to complex wiring harnesses, integration difficulties across multiple vendors' systems, and growing security vulnerabilities stemming from numerous attack surfaces.

To mitigate these issues, the automotive industry is undergoing a paradigm shift from traditional distributed ECU architectures toward centralized computing architectures. This transition is driven not only by the need to reduce hardware complexity and integration costs but also by the growing computational capabilities of modern processors [18]. In this emerging design, multiple domain-specific functions are consolidated into a smaller number of high-performance computing platforms—referred to as domain controllers or central compute units—capable of handling diverse workloads. These platforms typically feature multi-core processors, support for virtualization or containerization, and unified software environments such as the AUTOSAR Adaptive Platform or embedded Linux distributions.

This architectural transition enables several key advantages: improved resource utilization through dynamic workload distribution, enhanced security via software partitioning and isolation, simplified wiring and system integration, and greater scalability for software-defined vehicle features. Furthermore, centralized compute units

form the foundation for over-the-air (OTA) software updates, continuous feature enhancement, and deployment of AI-powered functions such as sensor fusion and decision-making in autonomous driving applications [19].

Parallel to this hardware evolution, there is an increasing reliance on vECUs—software emulations of physical ECUs—for early-stage development and testing. Traditional HiL testing involves connecting real ECU hardware to simulated environments, while Software-in-the-Loop (SiL) testing runs embedded software within a virtual setup without real hardware[20]. Although HiL offers high fidelity, it is often costly, time-consuming, and less scalable. In contrast, SiL-based approaches using vECUs allow developers to simulate ECU behavior, validate control algorithms, and detect bugs in a virtual environment long before physical prototypes are available. This approach accelerates development cycles, supports continuous integration workflows, and reduces the dependency on physical test setups, which are often bottlenecks in the development pipeline[8].

2.2 vECUs and Their Classification

vECUs are software representations of ECUs that simulate their behavior in a virtualized environment, enabling early-stage testing without physical hardware. vECUs are categorized into five levels (L0-L4) based on their complexity and functionality [2].

- **L0 vECUs**: Comprise only the controller model itself (e.g. as an MATLAB Simulink model) or the C code generated from the controller model (e.g. as a Functional Mock-up Unit(FMU)). A L0 vECU does not include any production code or any part of the basic software layers. This simplest V-ECU type can only be used to test the control algorithm itself. For other tests, e.g. software interface tests, more complex vECU types are necessary which are closer to the production code of the real ECU.
- **L1 vECUs**: Contain production code of the application software. These software components have to be supplemented with functionalities of lower software layers to enable the application software to run as desired during simulation. These typically do not consist of production code but instead are created or generated specifically for the vECU. In case of AUTOSAR, these are the Run-Time Environment (RTE) and the operating system, as well as, for example, functionality that allows the vECU to send and receive data.
- **L2 vECUs**: Provide simulated basic software (BSW) (i.e., referring to all the hardware-independent software modules that support the application layer in an AUTOSAR system), functionalities in addition to the content of L1 vECUs. These include basic software functionalities that are not on the production software level. They are created specifically for use in the vECU for simulation purposes. Examples are a COM stack, NVRAM functionality, or diagnostic functionalities.
- **L3 vECUs**: Include not only production application software but also production basic software (production BSW) or - for tests of the BSW - only the

production BSW or parts of the production BSW. Application software and basic software have to be hardware-independent. In terms of AUTOSAR, this can be everything above and including the microcode abstraction layer (MCAL layer), the operating system, and those parts of complex device drivers that are hardware independent.

- **L4 vECUs**: Contain production code compiled for the real target ECU. It is also possible to have hardware dependencies included. This allows all the software layers (e.g., MCAL, OS, and complex device drivers) to be included. An instruction set simulator for a given target machine is required to execute an L4 vECU on a simulation platform.

L3 vECUs are particularly valuable as they enable the testing of ECU software and network communication without requiring physical ECU hardware. They can also simulate the behavior of other ECUs in a vehicle network, known as 'rest bus models'(i.e., simulation of the remaining ECUs), within HiL simulations. This allows a real ECU to interact with virtual counterparts, bridging the gap between pure software simulations and full hardware implementations [2]. This thesis aims to upgrade an existing L2 vECU to L3 capability by integrating an open-source, OSEK-compliant RTOS, thereby enhancing its ability to support realistic software-in-the-loop and hardware-in-the-loop testing scenarios.

2.3 OSEK RTOS

OSEK (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug), which translates to "Open Systems and their Interfaces for the Electronics in Motor Vehicles," is an RTOS standard specifically developed for use in automotive ECUs [4]. Initially established by a consortium of German automotive manufacturers, including BMW, Daimler-Benz, and Volkswagen, OSEK was designed to improve software reusability, interoperability, and development efficiency across different ECU platforms. The standard was later adopted internationally as ISO 17356 and serves as the foundational runtime environment for the AUTOSAR Classic Platform.

Due to its deterministic behavior and static configuration model, OSEK is particularly well-suited for embedded automotive systems that require predictable timing and resource-constrained execution. It has been widely deployed across a broad range of ECUs, including body control modules (BCMs), transmission control units (TCUs), engine control units (ECUs), and other systems that demand reliable, real-time operation.

A key feature of the OSEK RTOS is its priority-driven preemptive task scheduling. Tasks are assigned fixed priorities at compile-time, and higher-priority tasks are allowed to preempt lower-priority ones. This ensures that critical tasks—such as those responsible for braking, throttle response, or airbag deployment—can be executed with minimal latency, thus satisfying stringent real-time constraints.

For robust resource management, OSEK implements the Priority Ceiling Protocol (PCP)[21], a synchronization technique designed to prevent priority inversion, a condition where a low-priority task blocks a high-priority one from accessing shared

resources. By temporarily raising the priority of a task holding a shared resource to the ceiling level, the system ensures predictable and safe access to critical resources, preserving the system's real-time guarantees.

OSEK also supports a set of inter-task communication mechanisms, including events, messages, and semaphores, which are used for task synchronization and data exchange. These primitives facilitate reliable coordination between concurrent tasks, enabling efficient control logic and consistent system behavior under varying workloads.

2.4 Real-Time Scheduling

Real-time scheduling ensures that time-sensitive tasks are executed within predefined deadlines, which is critical in automotive systems. Two of the most commonly used real-time scheduling algorithms in this domain are Rate Monotonic (RM) scheduling and Earliest Deadline First (EDF) scheduling [9]. RM is a fixed-priority algorithm in which tasks with shorter periods are assigned higher priorities. It is well-suited for periodic tasks with known execution times, but is less effective for handling dynamic task behavior efficiently [9]. In contrast, EDF follows a dynamic priority assignment approach in which the task instance with the earliest deadline in time is given the highest priority. Although EDF offers greater flexibility than RM, it introduces higher computational overhead and increased implementation complexity, making it less ideal for resource-constrained automotive systems [10].

RTA is a fundamental technique in real-time systems used to determine whether all tasks in a system can meet their deadlines under a specific scheduling policy. It is particularly useful for fixed-priority scheduling schemes such as RM scheduling. Figure 2.1 illustrates the response time of each task in a real-time system that consists of three tasks with different priorities. The rectangular box represents the execution of a task instance. The number inside each box represents the index of the task instance. The response time is calculated as the difference between the instance's arrival time and its completion time. Small rectangular boxes with the same index indicate that a low-priority task was preempted by one or more higher-priority tasks during its execution. The worst-case response time is defined as the maximum response time observed across all task instances. In this example, it corresponds to the first instance of Task 3, with a measured value of 8 time units.

2.4.1 Classic RTA Model

Equation 2.1 illustrates the classic way of doing RTA. The parameters of the equation include:

- $R_i^{(k)}$: The k-th iteration of the response time estimate for task τ_i
- C_i: Worst-case execution time (WCET) of task τ_i
- T_j: Period of higher priority task τ_j
- C_j: WCET of higher priority task τ_j

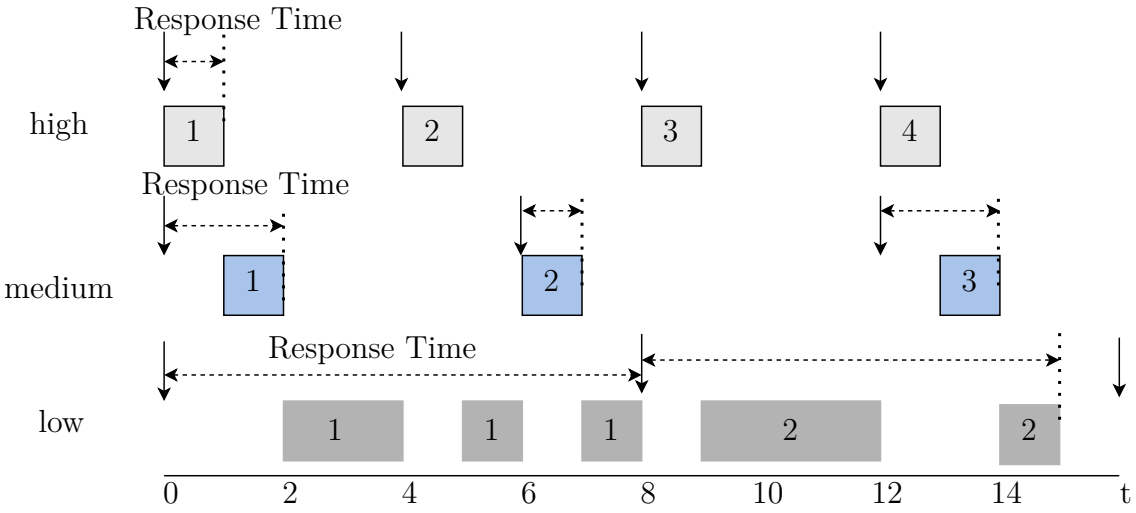


Figure 2.1: Example of Response Time

- $hp(i)$: Set of tasks with higher priority than τ_i
- B_i: Blocking from a task with priority lower than τ_i.

For a task τ_i with priority i (where lower i implies higher priority), the worst-case response time R_i can be computed iteratively using the following classic recurrence relation:

$$R_i^{(k+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + B_i \tag{2.1}$$

The iteration continues until $R_i^{(k+1)} = R_i^{(k)}$ (convergence) or $R_i^{(k+1)} > D_i$ (deadline miss), where D_i is the deadline of task τ_i. In the equation, the term *blocking* refers to the execution delay experienced by a higher-priority task when it is prevented from executing because a lower-priority task is holding a critical resource that the higher-priority task requires. This analysis enables engineers to validate offline whether a set of periodic tasks scheduled under a fixed-priority policy will meet their timing constraints in the worst-case scenario, which is critical for safety in automotive ECU applications.

2.4.2 RTA Model with Release Jitter and Offset

In addition to the classic RTA method, this thesis adopts an extended analysis model that captures dynamic behaviors such as release jitter, task offsets, and blocking times. *Release jitter* refers to the time variation between a task's arrival and its actual release, while *offset* specifies when the first instance of a periodic task occurs relative to system start. *Blocking time* is the delay a higher-priority task experiences when waiting on a resource held by a lower-priority task.

The model used is based on the work of O. Redell and M. Törngren [1], which incorporates these dynamic factors into the worst-case response time (WCRT) computation. The WCRT of the k-th instance of task τ_i, denoted R_{ik}, is given by Equation

2.2. The k-th instance is used for generalization purposes; in practical scenarios, analysis may focus on the first instance or the critical instance.

$$R_{ik} = B_i + C_i^{\max} + I_0(\tau_{ik}) + I_1(\tau_{ik}) + I_2(\tau_{ik}) + J_i \tag{2.2}$$

where:

- B_i is the blocking time due to lower-priority tasks,
- $C_i^{\max}$ is the maximum execution time of task τ_i,
- J_i is the maximum release jitter of task τ_i,
- I_0, I_1, and I_2 represent the interference caused by higher-priority task instances from three disjoint sets:
  - S_0: task instances arrived before r_{ik} and can only be released before r_{ik}, even with maximum jitter,
  - S_1: task instances arrived before r_{ik} and may be released exactly at r_{ik} due to jitter,
  - S_2: task instances arrived at or after r_{ik} and are released after r_{ik}.

Figure 2.2 illustrates the execution of the k-th instance of task τ_i, denoted as τ_{ik}. The gray boxes represent instances of a higher-priority task τ_j: specifically, the $(p-5)^{\text{th}}$, $(p-4)^{\text{th}}$, and $(p-3)^{\text{th}}$ instances belong to S_0; the $(p-2)^{\text{th}}$ and $(p-1)^{\text{th}}$ instances belong to S_1; and instances from the p^{th} onward belong to S_2.

The horizontal dashed lines extending from the arrival point of each task instance represent the *release jitter*—that is, the delay between when a task arrives (becomes ready) and when it is released (becomes available for execution). These lines help visualize how jitter affects the task release timeline and the classification of instances into S_0, S_1, or S_2. In the figure, a_j denotes the arrival time of each instance, and r_{ik} marks the release time of the task instance τ_{ik}.

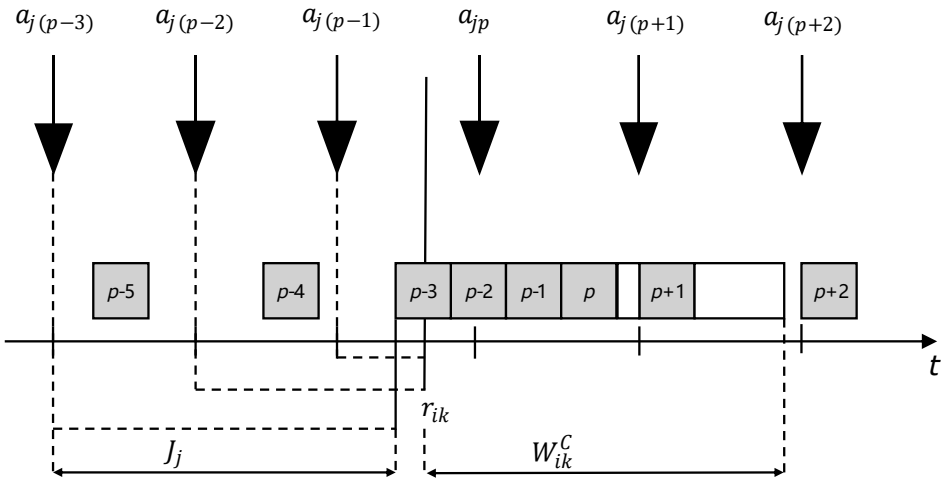


Figure 2.2: Example Execution of τ_{ik} [1]

Additionally, the impact of release jitter on response time is visualized in Figure 2.3, which highlights how task execution and response time shift in the presence of jitter. Compared to the classic RTA model, which assumes synchronous task releases and no jitter, this method provides more accurate response time estimates by accounting for offset, jitter, and interference from specific classes of task instances. The complete derivation and computation procedures for each term in Equation 2.2 are provided in Appendix A.

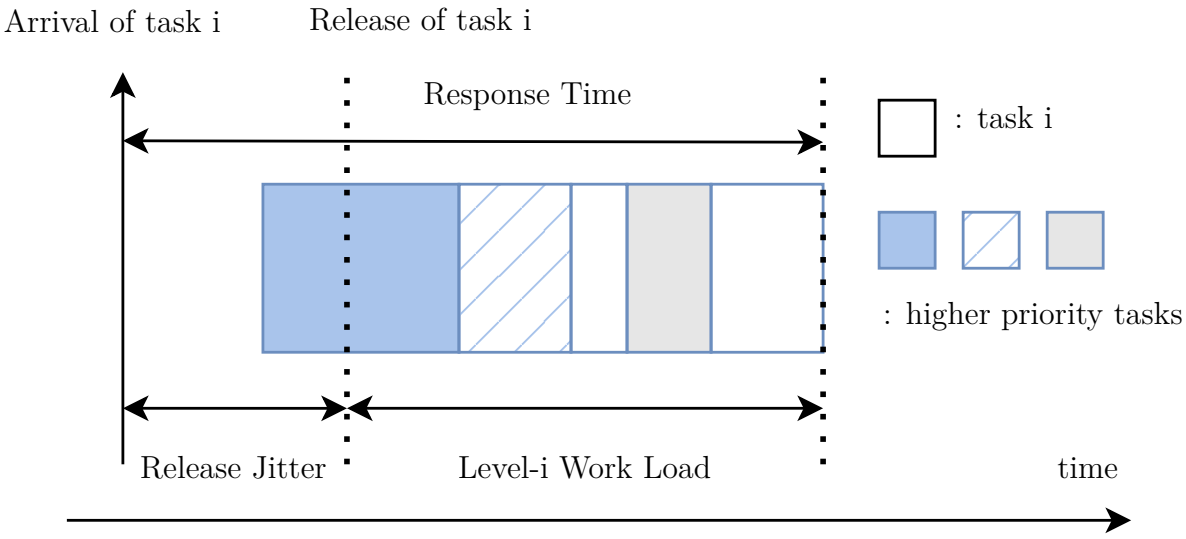


Figure 2.3: Response Time with Release Jitter

2.4.3 Timing Metrics for Real-Time Analysis

In real-time systems, several timing-related metrics are used to evaluate the temporal behavior of task execution. This thesis considers five key time-based metrics for each task instance:

- **Task Arrival Time**: The time at which a task is expected to be released by the scheduler, calculated based on its period and offset.
- **Response Time**: The duration from the task's arrival time to the completion of its execution.
- **Release Jitter**: The deviation between the expected release time and the actual time the task became ready to execute.
- **Interference**: The delay in a task's execution caused by preemption from higher-priority tasks.
- **Blocking Time**: The delay experienced by a task due to lower-priority tasks occupying a critical section that prevents preemption.

These metrics are fundamental for schedulability analysis and are used throughout this thesis for both theoretical modeling and empirical evaluation.

2.4.4 Harmonic Periods

In real-time systems, task periods play a critical role in determining the schedulability and timing behavior of the system. A set of tasks is said to have *harmonic periods* when each task's period is an integer multiple of the periods of all higher-priority tasks. For example, if tasks have periods of 5 ms, 10 ms, and 20 ms, they form a harmonic set. Harmonic periods simplify timing analysis and improve schedulability because task activations align regularly, reducing the likelihood of complex interference patterns and minimizing preemption overhead. Many theoretical models exhibit optimal or near-optimal behavior when task periods are harmonic. Consequently, harmonic task designs are often favored in safety-critical and embedded systems where timing predictability is essential.

2.5 Legacy Scheduler

This section describes the legacy static scheduler used in L2 vECUs. The legacy scheduler is Volvo L2 vECUs' current scheduler that operates based on a static schedule table, where tasks are organized according to their start times. These start times correspond not to clock time, but to periodic interrupt signals known as system ticks. At each system tick, the schedule table specifies which tasks should be activated.

Figure 2.4 illustrates an example of real-time scheduling under the RM priority policy. Both tasks arrive simultaneously at time 0, but since the 5 ms task has higher priority than the 10 ms task, it preempts the latter. During the execution of the first instance of the 10 ms task, the second instance of the 5 ms task arrives and preempts it again. However, the legacy scheduler does not support preemption. Task execution time in the vECU context refers to the time taken to execute the simulated task functions in software, which is significantly shorter than on actual ECU hardware. This is sufficient in the current Level 2 vECU context, where real-time behavior is not required, as the L2 vECUs primarily focus on functional validation rather than precise timing accuracy.

Figure 2.5 shows a sample task schedule generated by the legacy scheduler in a vECU. Due to short execution times and the absence of preemption, task execution is determined solely by arrival order. Once activated, a task runs to completion, regardless of priority or subsequent arrivals. In this example, both tasks arrive at time 0, but the 5 ms task arrives slightly earlier and executes first. At time 10, the third instance of the 5 ms task and the second instance of the 10 ms task arrive simultaneously; however, the 10 ms task arrives just slightly earlier due to the generated static schedule table and executes first—even though it has lower priority—demonstrating the non-preemptive nature of the legacy scheduler. The generated static schedule table and how the legacy scheduler process it will be discussed in detail in Section 4.1.

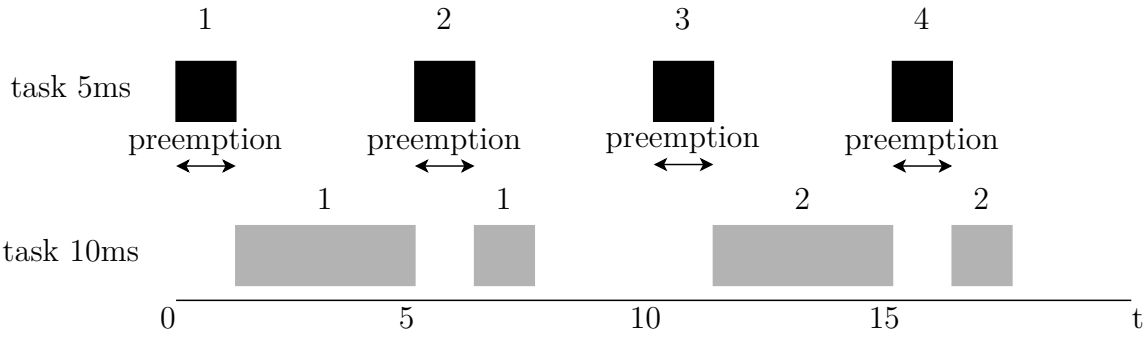


Figure 2.4: Real Time Scheduling with Preemption

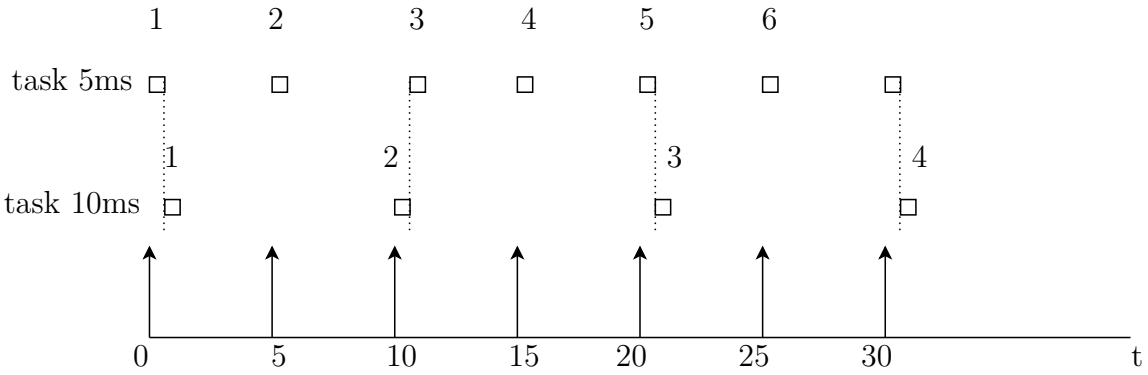


Figure 2.5: Legacy Scheduler Example

2.6 Linux Real-Time Patch: `PREEMPT_RT`

The `PREEMPT_RT` (a real-time Linux kernel patch) is a set of modifications to the standard Linux kernel that transforms it into a real-time capable operating system. While traditional Linux kernels are optimized for throughput and fairness, they lack deterministic timing behavior, making them unsuitable for hard real-time applications. `PREEMPT_RT` addresses these limitations by reducing worst-case latencies and enabling finer control over task scheduling. The Linux kernel patch will be used in replacing selected RTOS's time patch, and it will be introduced in Section 4.1.

2.7 Industrial Tools

Industrial tools provide a robust and efficient foundation for automotive system development. The three representative tools used in this thesis are:

- **Bazel**: An open-source build system developed by Google. It is designed to support large-scale software projects built across multiple languages such as C/C++, Java, and Python. In this thesis, Bazel is used to manage the build process across different environments.
- **ATI Vision**: A real-time calibration and data acquisition tool widely used in automotive ECU development. It is allowed to monitor, log, and modify

internal ECU variables during runtime without stopping the system.

- **Trace32**: A debugging and tracing toolset used for embedded system development. It supports a wide range of microcontrollers and processors via JTAG, SWD, or other debug interfaces, and offers capabilities such as low-level debugging, memory inspection, real-time tracing, and performance profiling.

Together, these tools streamline the development, debugging, and validation of real-time embedded systems in both simulated and physical environments.

3

Methodology

This chapter outlines the methodological approach adopted to achieve the objectives of this thesis, which include enabling preemptive scheduling in the L2 vECU using an OSEK RTOS, analyzing real-time task behavior on both virtual and physical ECUs, applying response-time analysis techniques, including offset-aware models, and evaluating their accuracy through empirical comparison.

The project is structured around two core components: first, the integration of a real-time RTOS into a vECU environment; and second, the measurement of task performance on both the vECU and a hardware ECU. These measurements serve as the foundation for applying two established RTA models aimed at estimating real-time behavior. The two RTA models are primarily based on hardware ECU data, while measurements from the vECU are used to evaluate consistency with both the hardware results and the models' predictions.

3.1 RTOS Integration into vECU

To achieve preemptive real-time scheduling, the legacy static scheduler in the L2 vECU is replaced with *Trampoline* [5], an open-source RTOS compliant with the OSEK standard. Trampoline is selected due to its open-source availability, lightweight footprint, and full compliance with the OSEK standard, making it particularly well-suited for academic experimentation and integration into constrained embedded environments like the vECU. For simplicity, the integrated Trampoline RTOS is configured for single-core operation, replacing the legacy scheduler on Core 1 of vECU. The remaining two cores retain their original scheduling configuration based on the generated static schedule table.

Trampoline's configuration is defined via an OSEK Implementation Language (OIL) file that specifies tasks, alarms, events, system clocks, priorities, and other system parameters. The purpose of this configuration is to tailor the Trampoline RTOS kernel to the specific requirements of the target vECU system. By doing so, we define the structure and behavior of the RTOS to match the timing and execution needs of the vECU application.

Figure 3.1 illustrates the build pipeline of a Trampoline application. The process involves several abstraction levels:

- **Application Level**: This includes the user-defined application source code (e.g., the vECU logic) and an OIL file that describes the required real-time behavior (task set, alarms, time resolution, etc.).

- **Configuration Level**: The OIL file is processed by Trampoline's OIL compiler, `Goil`, which generates configuration-specific C source files. These describe system objects (tasks, counters, alarms, etc.) as expected by the Trampoline kernel.
- **Middleware Level**: This layer includes Trampoline's API implementation files, which provide the standard services specified by OSEK (such as task management, scheduling, and alarm handling). These APIs interact with both the user application and the kernel.
- **Kernel Level**: Finally, all source files—including the application code, Goil-generated configuration code, and API implementations—are compiled and linked with the Trampoline kernel source code to generate the final executable binary tailored to the configured vECU system.

This layered approach ensures that the kernel is not generic but optimized and configured specifically for the real-time needs and structure of the vECU application.

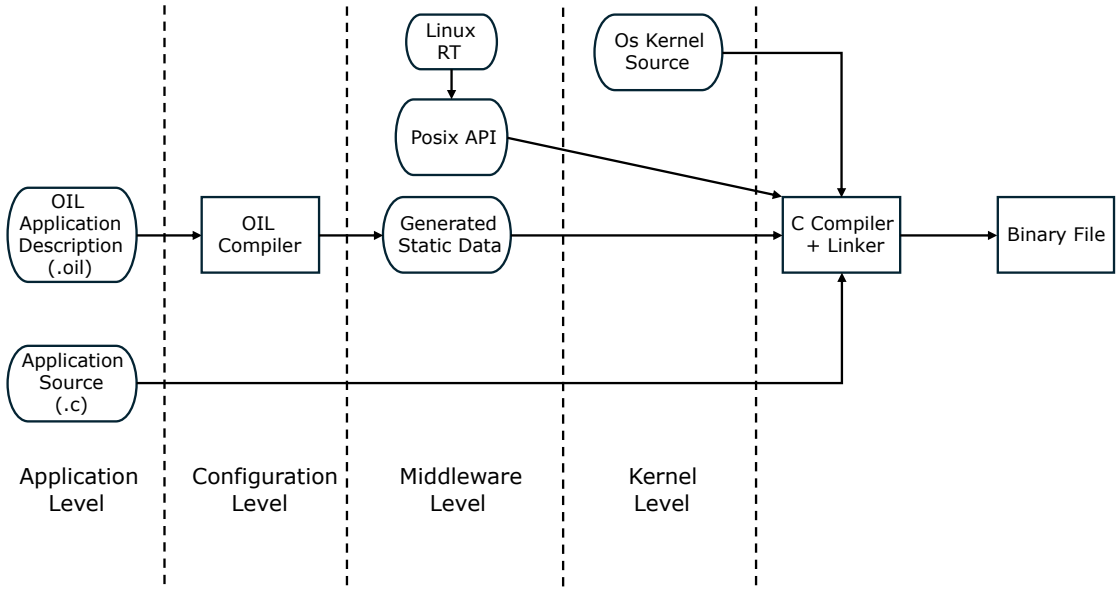


Figure 3.1: Trampoline Build Process

3.1.1 vECU Execution Environment and Timing Emulation

All vECU measurements are conducted on a host system equipped with an Intel Core i7-13850HX processor (`x86_64` architecture), running Ubuntu 24.04.2 LTS with the `PREEMPT_RT` real-time kernel patch applied. This configuration meets the runtime requirements of the vECU, supports real-time scheduling, and provides high-resolution timing capabilities (1 ns), making it suitable for evaluating the integrated RTOS under realistic conditions.

The goal of the measurement is to evaluate the timing behavior of the vECU with the integrated Trampoline-based RTOS. Specifically, the following timing parame-

ters are measured: task start times, inclusive and exclusive execution times, and release jitter. These metrics are used in ECU timing analysis and are essential for assessing the responsiveness and determinism of real-time task scheduling. Such measurements are feasible due to full access to both the application code and RTOS internals in the virtual environment, enabling fine-grained instrumentation not always possible on physical ECUs. By comparing these timing measurements with those collected from a physical ECU, we can assess how accurately the vECU, augmented with an integrated RTOS, can replicate the timing behavior of a real embedded control system.

However, the original vECU implementation includes stub functions in place of hardware-specific code, resulting in negligible task execution times that do not effectively exercise the RTOS's preemptive scheduling. To address this, tasks are augmented with non-blocking delay functions (i.e., calibrated busy loops) that simulate realistic workloads while keeping the scheduler responsive.

Non-blocking delays are initially configured with varying durations to verify the correct behavior of preemptive scheduling under the integrated RTOS. Once preemptive behavior is confirmed, the delay durations are calibrated to match the WCETs measured on the physical ECU. This adjustment ensures that the vECU task set exhibits computational characteristics comparable to those of the ECU, enabling a meaningful comparison of response times. The final configuration preserves full preemptive capability and schedulability under fixed-priority preemptive scheduling, allowing for accurate evaluation of timing behavior in a controlled virtual environment.

To collect timing data, Linux system time APIs (e.g., `clock_gettime`) are used to measure key runtime metrics such as task start and stop times. While each API call introduces a small overhead, the average cost of this overhead is measured separately and subtracted during post-processing, ensuring that the recorded timing metrics closely reflect true execution behavior.

3.2 Hardware ECU Selection and Measurement

A production ECU with three cores, running the configuration `P_Engine_EU6_T2`, is chosen for performance benchmarking. This specific configuration was chosen for several reasons. First, it is a mature and well-developed platform, which reduces the likelihood of hardware-related issues. Although it is not the latest model, it uses the same processor and a similar architecture as newer versions. Additionally, the corresponding hardware is readily available in stock, ensuring convenient access. Furthermore, while the ECU supports multicore operation, this configuration runs all periodic tasks on a single core, simplifying real-time analysis to a single-core scenario.

The `P_Engine_EU6_T2` configuration represents a well-established setup for combustion engine control. Due to confidentiality constraints, specific implementation details of this configuration cannot be disclosed. However, its use in production environments ensures realistic and representative workload conditions for evaluating timing performance. The combustion engine was chosen because this project is

conducted in collaboration with Volvo Trucks, and the platform for electric engine control is still under development. Importantly, the work presented here is not dependent on whether the underlying system is a combustion or electric engine, so the choice of configuration does not affect the generality of the results.

Ten periodic tasks are scheduled by the scheduler and executed on the ECU. Table 3.1 summarizes the timing characteristics of the task set for both the ECU and vECU platforms. The Lauterbach *Trace32* debugger [22], along with Command Macro Language (CMM) scripts, is used to capture the start time, inclusive execution time, and exclusive execution time of each instance of every task within a 25-second measurement window. The inclusive execution time refers to the duration from the start time to the completion of the task instance. The exclusive execution time is the actual CPU time consumed by the task, excluding any delays due to interference or blocking. The 25-second measurement window starts at the point when monitoring begins using Lauterbach Trace32 and custom CMM scripts, during which the execution behavior of all periodic tasks is captured.

Table 3.1: Task Configuration

Task ID	Period (ms)	Offset (μs)	RM Priority
Task 1	1.25	0	1
Task 2	2.5	1,250	2
Task 3	5	0	3
Task 4	10	2,500	4
Task 5	20	7,500	5
Task 6	40	17,500	6
Task 7	80	37,500	7
Task 8	160	77,500	8
Task 9	320	157,500	9
Task 10	1,000	497,500	10

Note that the start time measured does not represent the release time of the instance, but rather the actual moment the task instance begins execution on the processor (i.e., *start time = release time + interference + blocking*). The inclusive execution time measured refers to the duration from the start time to the completion of the task instance. The exclusive execution time measured is the actual CPU time consumed by the task, excluding any delays due to interference or blocking.

3.3 Theoretical Modeling

This project applies two established response-time analysis models to evaluate the timing behavior of real-time tasks. The first is the classic response-time analysis for RM scheduling, shown in Equation 2.1, which assumes synchronous task releases and uses worst-case execution times (WCETs) derived from preliminary measurements. To accommodate practical factors such as release jitter and task offsets, a second model is employed—an offset- and jitter-aware analysis method based on the approach described in [1]. These models produce theoretical timing predictions, which are compared against empirical results obtained from both the vECU and the

physical ECU. The comparison serves to assess how accurately the vECU replicates the real-time behavior of the physical system and to validate the applicability of theoretical analysis under practical conditions.

In summary, the methodology combines system-level integration, precise measurement, and formal timing analysis to evaluate the real-time capabilities of the vECU enhanced with a Trampoline-based RTOS. A practical measurement framework was developed to capture per-instance timing behavior on both virtual and physical ECUs. Through instance index reconstruction, response time decomposition, and isolation-based jitter estimation, key timing metrics—such as release jitter, blocking, and interference—were extracted. These metrics enable the application of both classic and offset-aware response-time analysis models. The resulting theoretical predictions will be compared (in later chapters) with empirical measurements to assess the timing fidelity of the vECU and to validate the analytical models in practical scenarios.

4

Design

This chapter first explains how the legacy static scheduler was replaced with a priority-based preemptive RTOS to enable realistic real-time execution in the vECU. It then describes the deployment approach used to ensure a consistent and reproducible testing environment across different host systems. Following that, the instrumentation setup for capturing detailed timing behavior on both virtual and physical platforms is introduced, along with strategies to minimize measurement overhead. Finally, the chapter outlines how empirical data were used to apply response-time analysis models and evaluate their predictive accuracy under practical execution conditions.

4.1 vECU and RTOS Scheduler Design

The original L2 vECU employed the legacy scheduler, which is a static scheduler using a tick interval of 1250 μs as introduced in section 2.5. Lacking preemption, it enforced a deterministic but unrealistic execution model. To better approximate hardware scheduling, the legacy scheduler and the original static schedule table are now replaced with Trampoline, as shown in Figure 4.1. The entry functions of each task are now activated by the Trampoline scheduler instead of the legacy scheduler. The Trampoline scheduler manages task release and preemption based on their priorities, periods, and offsets. Once a task is activated, the mechanism for executing its activities remains unchanged. The *nodemanager* module continues to execute activities by calling the corresponding APIs. The primary replacement procedure involves:

- **Porting task metadata to an OIL configuration:** Each task's attributes, including its periodicity, priority level as per the RM assignment, stack size requirements, and associated C function entry point, are translated into Trampoline's OIL format. This includes defining `TASK` objects with `PRIORITY`, `SCHEDULE`, and `STACKSIZE` attributes, and creating corresponding `COUNTER` and `ALARM` objects to manage timing behavior.
- **Enabling periodic task execution via alarm callbacks and time interrupts:** For each periodic task, a corresponding `ALARM` is configured in the OIL file to trigger at specified intervals. These alarms are mapped to `ActivateTask()` calls, enabling time-driven task activation. Instead of relying on a hardware timer, the system uses high-resolution clock signals provided by

the `PREEMPT_RT` Linux kernel patch. This kernel-level support allows Trampoline to simulate timer-based scheduling with microsecond-level precision, using Linux time facilities to emulate OSEK's timer and counter abstractions.

- **Linking task entry functions with Trampoline scheduling hooks:** In the application source code, each task function is implemented using the OSEK-compliant signature `TASK(TaskName)`. These functions are linked to Trampoline's internal task scheduler by ensuring they match the identifiers declared in the OIL configuration. Additionally, the startup routine initializes Trampoline's kernel and activates the initial alarms and tasks as required by the application logic.

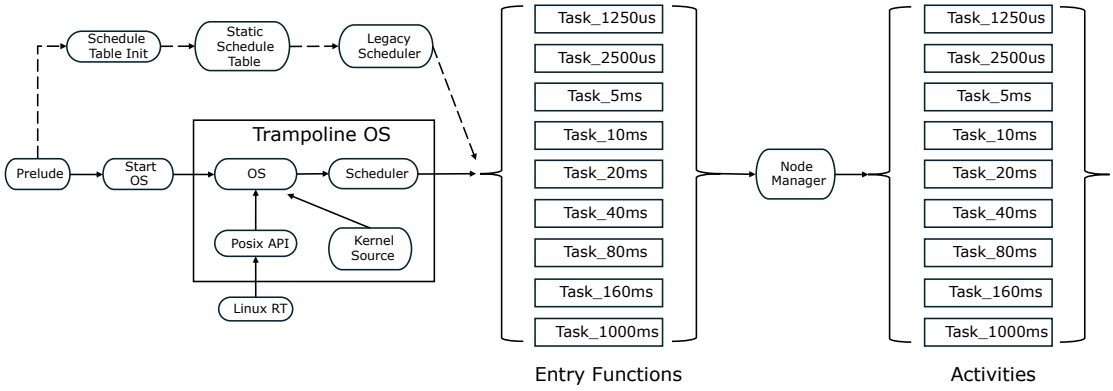


Figure 4.1: vECU Platform and Scheduler Design

With Trampoline integrated into the vECU, priority-based scheduling under the RM policy is now supported, enabling realistic preemptive execution. The next step is to configure and deploy the vECU in a way that allows for consistent and accurate timing measurements.

4.2 Docker-Based Deployment

To ensure portability, reproducibility, and isolation of the test environment, the vECU was deployed inside a Docker [23] container configured with a real-time Linux kernel (`PREEMPT_RT` patch applied). Docker provides a consistent runtime environment that abstracts away host-specific differences, allowing the vECU to be executed on various Linux-based machines without requiring manual reconfiguration.

The container includes all necessary dependencies for building and running the Trampoline-integrated vECU, such as compiler toolchains, system libraries, and build scripts. This encapsulation simplifies the deployment process and ensures that timing measurements are not influenced by variations in user environment setups.

The use of Docker is particularly beneficial for projects involving real-time systems. Without containment, subtle inconsistencies across development machines, such as mismatched library versions, kernel configurations, or timing behavior due to background processes, could compromise the accuracy and repeatability of measure-

ments. Docker eliminates these concerns by maintaining a controlled and identical environment for all executions, thereby improving the credibility of the evaluation and facilitating collaboration, version control, and future reproducibility of the results.

4.3 Performance Instrumentation

Dedicated logging mechanisms were implemented in both the vECU and physical ECU platforms to enable precise performance monitoring. In the vECU, timestamps for task activation, preemption, and termination were recorded and written to structured output files. For the physical ECU, equivalent timing data was collected using automated scripts interfaced with the *Trace32* debugger, ensuring consistency in measurement across both environments. These logs supported the analysis of inclusive and exclusive execution times, as well as preemption behavior, and were also used to verify the correctness of the integrated RTOS's fixed-priority preemptive scheduling.

Instrumentation overhead is an important consideration in timing analysis. Although logging introduces additional runtime cost, this overhead does not compromise the accuracy of individual timing values (e.g., start time, execution duration). A 25-second measurement window was selected to ensure that even the slowest task (with a 1000 ms period) is captured with sufficient instance count. Additionally, high-resolution timestamping (based on Linux system time APIs) was used to measure fine-grained timing metrics while maintaining a balance between resolution and performance overhead.

4.3.1 Arrival Time Reconstruction and Instance Mapping

To calculate response times, it is necessary to determine the arrival time T_a for each task instance. Since task periods and offsets are known (see Table 3.1), and the system uses RM fixed-priority scheduling, we can analytically express the theoretical arrival times of task instances using integer index variables n_1 through n_{10}. These indices count the number of activations for each periodic task since the system started.

$$T_a(1250\mu s) = n_1 \cdot 1250, \;\; n_1 \in \mathbb{Z} \tag{4.1}$$

$$T_a(2500\mu s) = (2n_2 + 1) \cdot 1250, \;\; n_2 \in \mathbb{Z} \tag{4.2}$$

$$T_a(5ms) = 4n_3 \cdot 1250, \;\; n_3 \in \mathbb{Z} \tag{4.3}$$

$$T_a(10ms) = (8n_4 + 2) \cdot 1250, \;\; n_4 \in \mathbb{Z} \tag{4.4}$$

$$T_a(20ms) = (16n_5 + 6) \cdot 1250, \;\; n_5 \in \mathbb{Z} \tag{4.5}$$

$$T_a(40ms) = (32n_6 + 14) \cdot 1250, \;\; n_6 \in \mathbb{Z} \tag{4.6}$$

$$T_a(80ms) = (64n_7 + 30) \cdot 1250, \;\; n_7 \in \mathbb{Z} \tag{4.7}$$

$$T_a(160ms) = (128n_8 + 62) \cdot 1250, \;\; n_8 \in \mathbb{Z} \tag{4.8}$$

$$T_a(320ms) = (256n_9 + 126) \cdot 1250, \;\; n_9 \in \mathbb{Z} \tag{4.9}$$

$$T_a(1000ms) = (800n_{10} + 398) \cdot 1250, \;\; n_{10} \in \mathbb{Z} \tag{4.10}$$

All arrival times are thus represented as integer multiples of the base time unit $1250\,\mu s$, but with task-specific coefficients and offsets, as shown in Equations 4.1 to 4.10. These equations assume that the scheduler begins executing at system time zero, providing a consistent temporal reference for all task activations. While a theoretical least common multiple (LCM) of all task periods exists, it is too large to be observable in any practical measurement window. Therefore, the instance index tuple $(n_1, \ldots, n_{10})$ within any finite window is effectively unique, enabling unambiguous mapping of log entries to instance indices.

4.3.2 Solving Instance Indices via Measured Start Times

In practice, the task arrival times T_a are not directly measurable. Instead, the actual start time of each task instance is captured, denoted as T_s. Since the scheduler begins execution at an unknown time offset T_{bias} relative to system time zero, and assuming each task begins execution within its period after arrival, we can relate the observed start times to the corresponding instance indices using the known task periods and offsets. This relationship is formalized in the following equations:

$$(T_s(1250\,\mu s) - T_{\text{bias}}) \,//\, 1250 = n_1, \quad n_1 \in \mathbb{Z} \tag{4.11}$$

$$(T_s(2500\,\mu s) - T_{\text{bias}}) \,//\, 1250 = 2n_2 + 1, \quad n_2 \in \mathbb{Z} \tag{4.12}$$

$$(T_s(5\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 4n_3, \quad n_3 \in \mathbb{Z} \tag{4.13}$$

$$(T_s(10\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 8n_4 + 2, \quad n_4 \in \mathbb{Z} \tag{4.14}$$

$$(T_s(20\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 16n_5 + 6, \quad n_5 \in \mathbb{Z} \tag{4.15}$$

$$(T_s(40\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 32n_6 + 14, \quad n_6 \in \mathbb{Z} \tag{4.16}$$

$$(T_s(80\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 64n_7 + 30, \quad n_7 \in \mathbb{Z} \tag{4.17}$$

$$(T_s(160\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 128n_8 + 62, \quad n_8 \in \mathbb{Z} \tag{4.18}$$

$$(T_s(320\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 256n_9 + 126, \quad n_9 \in \mathbb{Z} \tag{4.19}$$

$$(T_s(1000\,\text{ms}) - T_{\text{bias}}) \,//\, 1250 = 800n_{10} + 398, \quad n_{10} \in \mathbb{Z} \tag{4.20}$$

In these expressions, the operator // denotes integer division, which discards the remainder and retains only the integer part of the result. This reflects the fact that the measured start time must correspond to a discrete instance number on the timeline of periodic task releases.

It is safe to assume that the difference $(T_s - T_a)$ is less than the period of the task. If this were not the case, it would imply that tasks miss their deadlines and remain queued for execution across multiple periods, which would ultimately lead to system instability or crashes. However, such behavior is not observed in practice. Nonetheless, this assumption is verified after calculating the theoretical arrival times using the inferred instance numbers.

Given the integer constraints of the indices in Equations 4.11 to 4.20, and the previously established uniqueness of the instance index combinations within any finite measurement window (due to the large LCM of the task periods), the system of equations can be solved. This enables us to determine the values of n_1 to n_{10} along with the unknown time offset T_{bias}.

Once these indices are known, the corresponding theoretical arrival times T_a can be computed using Equations 4.1 to 4.10. These arrival times are then used to accurately compute the per-instance response times, as well as other key timing metrics such as release jitter, blocking time, and interference.

4.3.3 Timing Metric Decomposition

To apply theoretical RTA using the models introduced in Sections 2.4.1 and 2.4.2, it is essential to extract a set of timing metrics from each task instance. These include response time, release jitter, blocking time, and interference. Figure 4.2 illustrates the temporal relationships between these metrics during task execution.

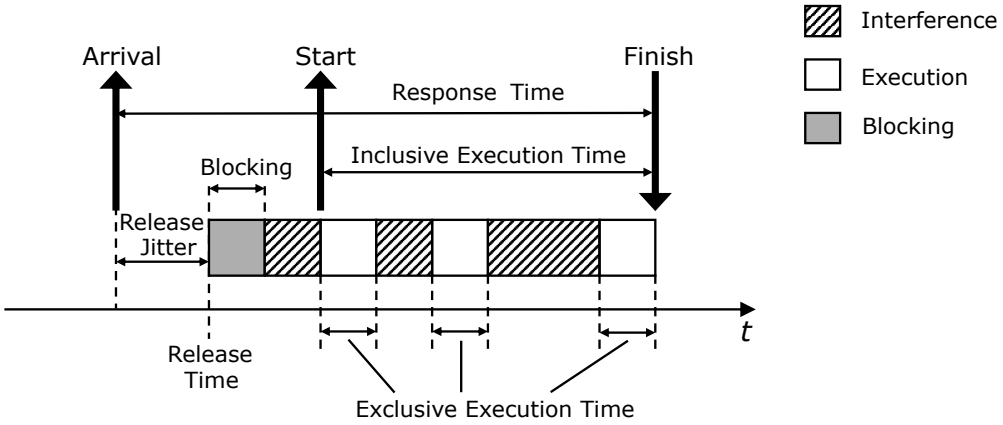


Figure 4.2: Visualization of Time Metrics in Execution

The response time R of a task instance is defined as the time from its arrival to the completion of its execution. It can be expressed as:

$$R = T_{\text{start}} + C_{\text{incl}} - T_a \tag{4.21}$$

where:

- T_a is the theoretical arrival time of the task,
- T_{start} is the actual start time,
- C_{incl} is the inclusive execution time (including preemptions).

Interference is a key contributor to response time and can be divided into two parts. The first occurs before the task begins execution and is caused by higher-priority tasks that continue to run after the current task instance has arrived. The second

occurs during the task's execution in the form of preemptions. The preemption-related interference is quantified by subtracting the exclusive execution time from the inclusive execution time, both of which are directly measured. The pre-start interference is identified by analyzing measurement logs to determine how long the task's execution was delayed by higher-priority tasks following its arrival.

From the timeline visualization (Figure 4.2), it is evident that the interval between a task's arrival and its start includes pre-start interference, release jitter, and possible blocking. Since the pre-start interference can be isolated through cross-task analysis, the remaining delay must be due to a combination of release jitter and blocking. To decouple these two effects, a dedicated execution mode was implemented in which tasks were run individually. Although this setup is functionally incomplete due to inter-task dependencies, it is analytically valuable because it eliminates interference, preemption, and blocking, thus exposing the task's intrinsic release behavior.

This isolated setup was realized through a runtime selection flag embedded in each task's code. Using the *Trace32* debugger, this flag was configured at runtime to activate only the task under study, while all other tasks were effectively skipped. However, the complete task set remained defined in the system, and all original release parameters (period and offset) were preserved. As a result, each task experienced a release pattern consistent with concurrent execution. The observed delay between the task's theoretical arrival and its actual start could thus be attributed solely to release jitter.

In parallel, a non-intrusive method was employed to detect blocking behavior during concurrent execution. Rather than inserting instrumentation in critical sections, blocking was inferred from timing traces. A blocking event was flagged whenever a higher-priority task's arrival coincided with the execution of a lower-priority task, and the higher-priority task did not start immediately. This forensic-style analysis enabled the identification and quantification of blocking durations without modifying or annotating shared resource usage.

Once all metrics were extracted—response time, release jitter, interference, and blocking—they were applied in both classical and comprehensive RTA models to assess schedulability and verify timing guarantees under realistic conditions.

4.4 Model Application Framework

The collected measurements were used to compute the response times of individual task instances on the hardware ECU, as detailed in Section 3.2. These empirical results were then compared against theoretical predictions obtained from established RTA models, enabling an evaluation of model accuracy under real-world execution conditions. Initial evaluations using a classic model assuming fully synchronous releases resulted in pessimistic estimates. To improve prediction accuracy, measured release jitters and task offsets were incorporated into the enhanced RTA model, as described in [1]. The pseudo code of the applied RTA algorithm is shown in Algorithm 1, and the corresponding equations are detailed in Section 2.4.

The following steps were carried out to apply and assess the analytical response-time

Algorithm 1 Calculate Worst-Case Response Time

for each task τ_i, $i = 1$ to n **do**
  $R_i \leftarrow 0$
  Compute L_i^U ▷ Eq. A.28
  $LCM_i \leftarrow \text{LCM}(T_1, ..., T_i)$
  $K \leftarrow LCM_i / T_i$
  for each instance τ_{ik}, $k = 0$ to $K - 1$ **do**
    $r_{ik} \leftarrow a_{ik} + J_i$ ▷ Eq. A.4
    Compute $\phi_{ik,j}$, $\delta_{ik,j}$, $N_{ik,j}$ ▷ Eqs. A.7, A.11, A.18
    Compute $I_0(\tau_{ik})$ ▷ Eqs. A.22, A.26
    Compute W_{ik}^C by iteration ▷ Eqs. A.2, A.12, A.14
    $R_{ik} \leftarrow W_{ik}^C + J_i$ ▷ Eq. A.1
    if $R_{ik} > R_i$ **then**
      $R_i \leftarrow R_{ik}$
    end if
  end for
end for

models in the context of the studied system:

- Extract release jitter and task offsets from empirical measurements to capture practical variations in task activation behavior.
- Apply the offset- and jitter-aware RTA model to estimate the WCRTs of tasks under realistic timing conditions, including release jitter and offsets.
- Compare model-predicted response times with the actual measurements collected from both the vECU and the hardware ECU to evaluate the model's predictive accuracy and to assess whether the vECU provides a safe and faithful simulation of the physical ECU's real-time behavior.
- Analyze and interpret any discrepancies between the theoretical predictions and empirical results to understand the limitations or assumptions of the analytical model.
- Refine and tune timing parameters when necessary to better reflect real-world execution behavior and improve model fidelity.

This systematic process allows for a critical evaluation of how well existing RTA models, especially those accounting for jitter and offsets, can represent actual task behavior in both virtualized and physical embedded environments. The analysis not only supports the validation of these models under realistic conditions but also provides insight into their practical applicability for timing verification in automotive software systems.

5

Results

This chapter presents the timing analysis results of periodic tasks executed on both the ECU and the vECU. A detailed description of the two platforms is provided in the chapter 2. The analysis includes empirical measurements obtained through runtime instrumentation, and theoretical response time calculations based on analytical models. Each platform is evaluated independently to assess task behavior in terms of execution time, response time, and release jitter. A comparative analysis is included to report observed differences in timing behavior between the ECU, vECU, and the theoretical predictions. These differences are further interpreted in the chapter 6.

The chapter is organized as follows:

- Section 5.1 provides an overview of the task set, scheduling configuration, and measurement setup.
- Sections 5.2 and 5.3 present detailed empirical results from the ECU and vECU platforms respectively.
- Section 5.4 presents the analytically derived worst-case response time estimates, which serve as upper bounds for evaluating observed task performance.

5.1 Overview of Task Set and Measurement Setup

To evaluate the real-time behavior of the system, runtime measurements were collected from both platforms. For each task instance, three key parameters were recorded:

- **Task Start Time**: The actual time at which a task instance begins execution.
- **Inclusive Execution Time**: The total duration from task start to completion, including any delays due to preemption and blocking.
- **Exclusive Execution Time**: The cumulative time the processor actively executed the task, excluding preemption and blocking.

Each periodic task was monitored independently, with its corresponding data saved in a separate CSV file containing hundreds of instances. During the measurement process, all tasks were active and scheduled concurrently using a preemptive, fixed-priority scheduler. Based on the raw trace data, five time-based metrics, including arrival time, response time, interference, release jitter, and blocking, as introduced in Section 2.4.3, were extracted for each task instance.

In addition to these empirical measurements, theoretical response time bounds were computed using an analytical model described in Chapter 3. These results provide a basis for comparison with observed behavior.

The evaluation uses a set of ten periodic tasks that are actually executed on the real ECU hardware, making the analysis directly applicable to real-world automotive systems. These tasks span a broad range of execution frequencies and are scheduled according to the Rate Monotonic (RM) policy, where shorter-period tasks receive higher priority. Table 3.1 in Section 3.2 summarizes the timing characteristics of the task set for both the ECU and vECU platforms. The remainder of this chapter presents comparative plots, summary tables, and statistical analyses. Each task is analyzed individually, and differences between ECU, vECU, and theoretical predictions are highlighted to assess timing behavior, predictability, and model accuracy.

The purpose of this comparison is twofold: comparing ECU and vECU executions helps identify discrepancies and evaluate how accurately the vECU can simulate the real ECU's real-time behavior; comparing both platforms against model-predicted timing boundaries assesses the safety and reliability of using the vECU for early-stage validation in place of the actual hardware.

5.2 ECU Measurements

This section presents the empirical timing results obtained from the physical ECU platform. Each periodic task was executed under a fixed-priority preemptive scheduler, and detailed runtime measurements were collected during a controlled 25-second observation window. The recorded data includes response time, inclusive and exclusive execution times, release jitter, and various forms of interference. These results provide insight into the real-time behavior, variability, and scheduling dynamics of the system under realistic multi-tasking conditions. The results are organized by timing metric, starting with response time, followed by execution time and interference characteristics.

5.2.1 Response Time Results

This section presents the response time distributions for a subset of the ten periodic tasks, focusing on those with the most notable or varied timing behavior. For each task, a plot visualizes response time per instance, with the goal of identifying timing consistency, variability, and any anomalies. Due to the similarity in behavior among many tasks—particularly those with narrow and tightly clustered response time distributions—plots for only the most representative and analytically interesting tasks are shown here. The complete set of plots is provided in Appendix B for reference.

Figure 5.1 presents the response time plot for the 5 ms task. Across approximately 5000 instances, the response times range from 0.567 ms to 1.164 ms, resulting in a total spread of about 0.597 ms. Notably, the data points form two distinct bands: one between approximately 0.98 ms and 1.17 ms, and another between 0.56 ms and

0.80 ms. This bimodal distribution may be attributed to preemption by higher-priority tasks, different execution paths, or variations in release jitter.

Each task consists of multiple functional activities (such as engine speed control or cooling system regulation), and these activities may execute different code paths depending on real-time input signals and operating conditions. As a result, even the same task can follow multiple distinct execution paths with varying computational demands.

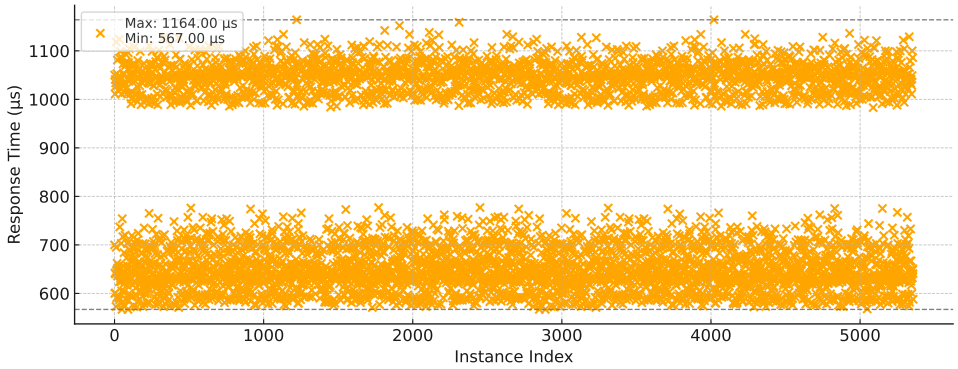


Figure 5.1: Response time per instance for the 5 ms task

As shown in Figure 5.2, the 10 ms task exhibits the widest range of response times among the higher-frequency tasks, with values spanning from 0.375 ms to 1.626 ms. Despite this broad range, the majority of instances are concentrated between 0.375 ms and approximately 0.5 ms, indicating generally consistent behavior.

The occasional large deviations are caused by a submodule called *NodeManager*, whose state machine is both initialized and executed within the 10 ms task. The initialization process involves multiple steps and depends on several input signals, which may not all be satisfied within a single execution cycle. As a result, the task may temporarily include additional computational load until the initialization completes.

Moreover, during normal operation, the state machine can reside in different internal states, each triggering different sets of operations. These state-dependent execution paths vary in complexity and execution time. While one main state is most frequently active—resulting in the dominant cluster of short response times—transitions to less common states occasionally occur, producing higher response times and resulting in multiple outliers.

Figure 5.3 illustrates the response time distribution for the 20 ms task. The measured response times range from 5.725 ms to 7.188 ms, yielding a spread of about 1.463 ms. The plot reveals three distinct bands of concentration, suggesting that the task is commonly executed under multiple timing scenarios. These may be due to varying interference patterns or conditional execution paths. Such structured variation highlights the importance of considering timing behavior beyond just worst-case values. Despite the variability, all observed response times remain comfortably within the task's deadline.

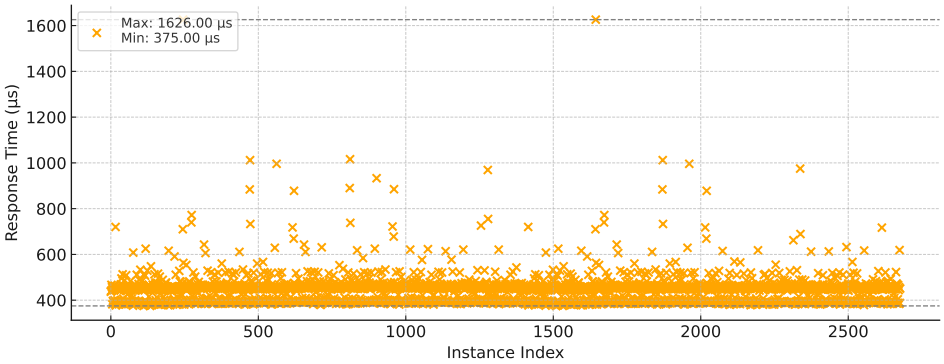


Figure 5.2: Response time per instance for the 10 ms task

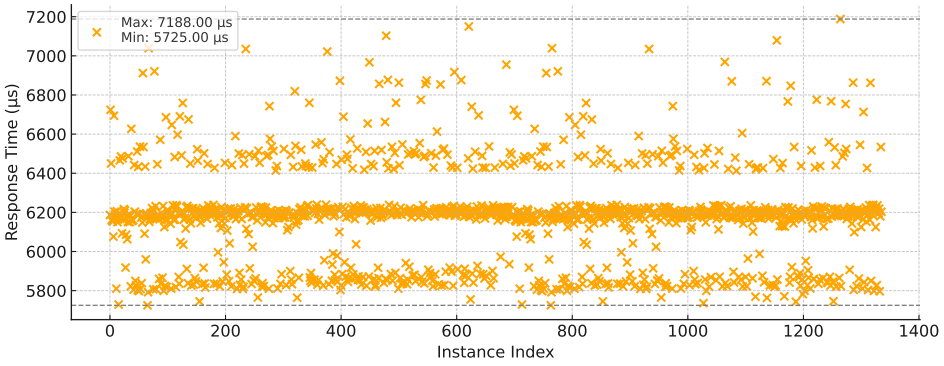


Figure 5.3: Response time per instance for the 20 ms task

Figure 5.4 presents the response time distribution for the 320 ms task, based on 80 recorded instances. Despite the long period and limited sample size, the data show a clear two-band pattern, suggesting distinct execution modes. The consistent appearance of these bands across multiple runs indicates stable underlying causes rather than random noise.

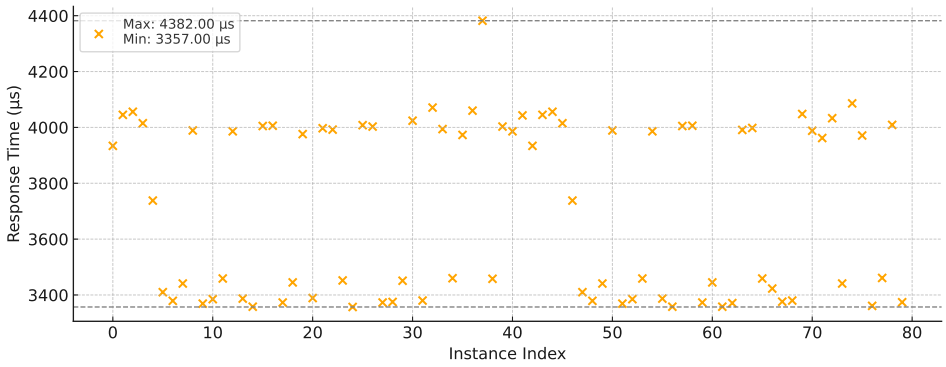


Figure 5.4: Response time per instance for the 320 ms task

The remaining tasks (1250 μs, 2500 μs, 40 ms, 80 ms, 160 ms, and 1000 ms) exhibited highly consistent response time distributions with minimal variation and no significant anomalies. Their plots are omitted from the main text for brevity and can be found in Appendix A. These tasks demonstrate predictable execution and are not the primary focus of variability analysis.

It is important to note that the data for each task was collected during a fixed 25-second measurement window. Although the measurement process was automated, it relied on inserting breakpoints into the running program to start and stop data collection. This introduced a non-negligible overhead in the total time needed for data collection, but it does not affect the accuracy of individual response time measurements. The variation in instance counts across tasks reflects only the differences in task periods.

Table 5.1 provides summary statistics for the response times of all periodic tasks executed on the ECU platform. Each entry includes the minimum, maximum, mean, and standard deviation across all recorded instances. These statistics complement the individual plots by quantifying the overall timing behavior of each task. As shown, all tasks meet their timing constraints, with response times remaining well below their respective periods. The degree of variability differs across tasks, reflecting their interaction with the fixed-priority scheduler and the level of interference experienced during execution.

5.2.2 Exclusive Execution Time Results

This subsection focuses on the exclusive execution time of each periodic task on the ECU platform. Exclusive execution time represents the duration for which the processor actively executed a given task, excluding all periods of preemption and blocking. This metric reflects the pure computational cost of the task and is par-

Table 5.1: Summary of Response Time Statistics for ECU Tasks

Task	Min (μs)	Max (μs)	Mean (μs)	Std Dev (μs)
1250 us	123	352	172	31
2500 us	141	217	174	11
5 ms	567	1,164	809	199
10 ms	375	1,626	445	70
20 ms	5,725	7,188	6,181	243
40 ms	419	519	444	15
80 ms	352	447	397	15
160 ms	5,552	6,712	5,949	216
320 ms	3,357	4,382	3,835	267
1000 ms	483	563	498	10

ticularly relevant for worst-case execution time (WCET) analysis and schedulability evaluations.

The reason WCET is of particular interest is that it serves as a key input for theoretical model estimations, such as response time analysis under fixed-priority scheduling. Accurate WCET values are essential for determining whether all tasks can meet their deadlines under worst-case conditions, thus ensuring system predictability and real-time correctness.

Although inclusive execution time was also recorded, it is not presented here. As shown in Figure 4.2, inclusive execution time is defined as the time between the task's actual start and its completion. This measurement excludes release jitter, blocking, and the portion of interference caused by the simultaneous release of higher-priority tasks. It only captures the interference that occurs after the task has started executing, specifically, any preemption by higher-priority tasks during its execution window.

Inclusive time was measured primarily because it, along with the exclusive execution time, can be directly observed from task start and completion timestamps. These two values are essential for deriving other time metrics, such as interference, that cannot be measured directly. However, as a performance metric on its own, inclusive time is limited: it does not reflect delays that occur before the task begins execution and therefore under-represents the total impact of scheduling. These effects are more comprehensively captured by the response time metric discussed earlier. As a result, exclusive execution time is presented here as the most meaningful measure of execution cost for real-time analysis. Table 5.2 summarizes the exclusive execution time statistics across all tasks. Each value is reported in milliseconds and includes the minimum, maximum, mean, and standard deviation across all observed task instances.

Figure 5.5 shows the distribution for task 5 ms, which is clearly bimodal. This indicates two dominant execution paths, likely triggered by input-dependent branching or distinct activity configurations within the task. Figure 5.6 illustrates a similar bimodal distribution for the 160 ms task, with one dominant execution path and a second, less frequent variant. This suggests conditional logic or feature toggles that activate additional computation under specific conditions.

Table 5.2: Summary of Exclusive Execution Time Statistics for ECU Tasks

Task	Min (μs)	Max (μs)	Mean (μs)	Std Dev (μs)
1250 us	127	354	174	32
2500 us	15	40	18	5
5 ms	391	460	418	21
10 ms	233	1,254	259	58
20 ms	4,047	4,222	4,072	26
40 ms	262	284	266	5
80 ms	193	218	200	5
160 ms	3,819	4,070	3,899	62
320 ms	2,322	2,360	2,334	9
1000 ms	40	46	41	1

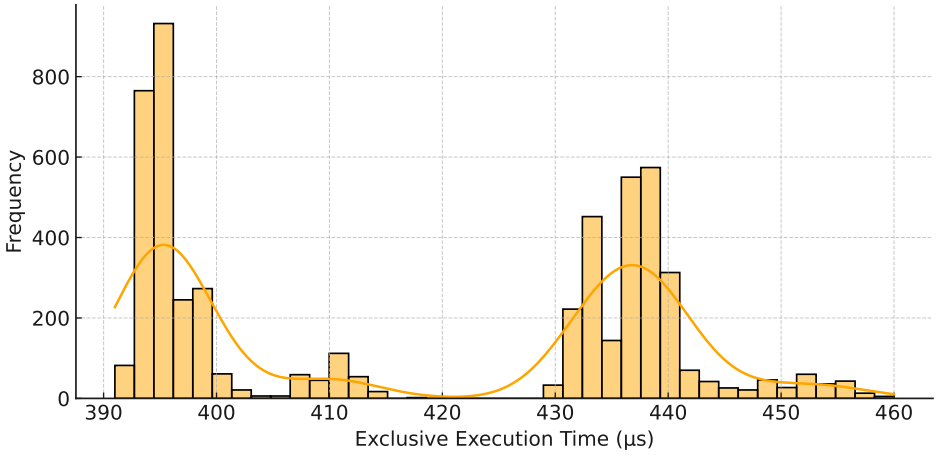


Figure 5.5: Exclusive Execution Time for the 5 ms task

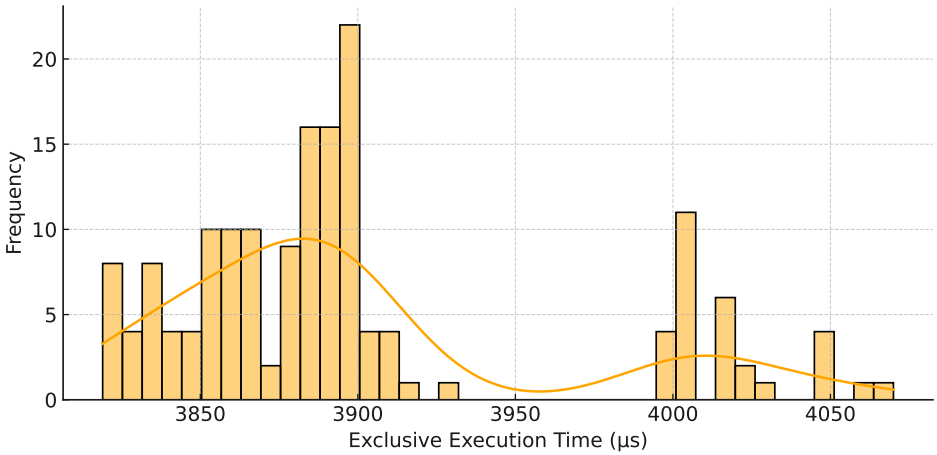


Figure 5.6: Exclusive Execution Time for the 160 ms task

Figure 5.7 presents the distribution for the 10 ms task. It is unimodal and right-skewed, reflecting consistent execution with some variation, likely due to the state-dependent execution of a submodule, as discussed earlier. Figure 5.8 shows the distribution for the 20 ms task. The shape is unimodal but asymmetric, with a longer right tail, indicating consistent execution interspersed with less frequent, longer-duration cases. The remaining task distributions (1250 μs, 2500 μs, 40 ms, 80 ms, 320 ms, 1000 ms) exhibited tightly clustered unimodal distributions with minimal variability and no notable features. These plots are provided in Appendix C for completeness.

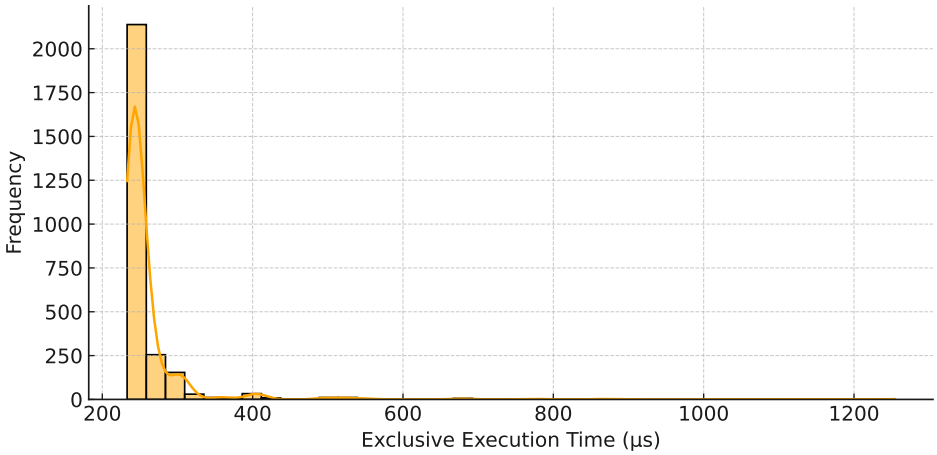


Figure 5.7: Exclusive Execution Time for the 10 ms task

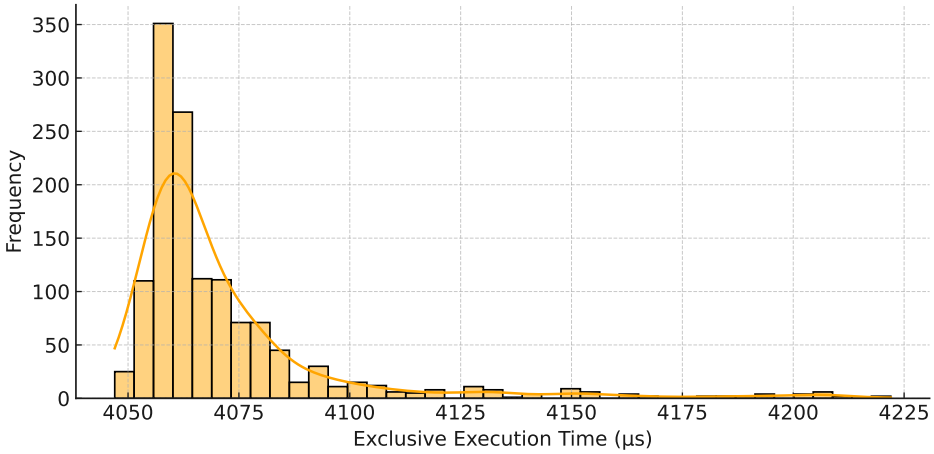


Figure 5.8: Exclusive Execution Time for the 20 ms task

5.2.3 Release Jitter and Blocking Results

Based on the design introduced in Section 4.3.3, isolated execution runs were used to compute each task's release jitter. The measured release jitter was tightly bound across all tasks, ranging from $-4\,\mu$s to $9\,\mu$s.

Blocking events were identified in concurrent runs by detecting timing overlaps where a higher-priority task's arrival was delayed due to the execution of a lower-priority task. This analysis revealed blocking in only four tasks, summarized in Table 5.3. All other tasks experienced zero blocking during the measurement window.

Table 5.3: Tasks with Non-Zero Blocking

Task	**Max Blocking (μs)**
1250 μs	39
2500 μs	35
5 ms	451
10 ms	68

5.3 vECU Measurements

This section presents timing analysis results obtained from the vECU after integrating an RTOS and simulating realistic execution durations. Each task was configured with a non-blocking delay corresponding to its WCET as measured on the physical ECU. In the vECU environment, many functions, especially those involving hardware interaction, are implemented as stub functions that return immediately without meaningful computation. As a result, the raw execution profile of the vECU differs significantly from that of the real ECU.

To address this discrepancy and to maintain preemptive behavior within the RTOS, we introduced non-blocking delays that emulate the execution time of each task without obstructing the RTOS scheduler. These delays allow higher-priority tasks to preempt as they would on real hardware. This approach strikes a balance between preserving schedulability realism and simplifying implementation. Before collecting timing data, a verification step was performed to confirm that preemption was functioning correctly in the modified vECU environment. The following results evaluate task timing behavior under fixed-priority preemptive scheduling and are intended to support direct comparison with the corresponding ECU measurements.

5.3.1 Verification of Preemptive Scheduling

To confirm that the integrated RTOS correctly enforced preemptive scheduling, artificial non-blocking delays were added to each task to create sufficient execution length for preemption to occur. A trace of execution intervals was collected, capturing the start and stop times, inclusive execution time, and exclusive execution time of each task instance.

Representative cases were selected where the inclusive execution time significantly exceeds the exclusive execution time, indicating that the task was interrupted and later resumed. Table 5.4 shows that instance 0 of the 5 ms task was preempted by instance 9 of the 1250 μs task and instance 4 of the 2500 μs task. The inclusive execution time of this 5 ms task instance is 1221 μs, which is slightly greater than the sum of the exclusive execution times of all three involved instances: 793 μs for

Table 5.4: Representative Task Instances Demonstrating Preemption (vECU)

Task	Instance	Start (μs)	Stop (μs)	Inclusive (μs)	Exclusive (μs)
5 ms	0	941761	942982	1221	793
1250 μs	9	942545	942946	401	401
2500 μs	4	942967	942974	7	7

the 5 ms task itself, 401 μs for the 1250 μs task, and 7 μs for the 2500 μs task, totaling 1201 μs.

Ideally, the inclusive execution time of a preempted task should equal the total exclusive execution time of all overlapping tasks, which includes the task itself and any higher-priority tasks that interrupted its execution. The observed discrepancy of 20 μs highlights a measurable overhead introduced by the RTOS during context switching. This overhead is inherent to task preemption, as the scheduler incurs additional cost when saving and restoring task states.

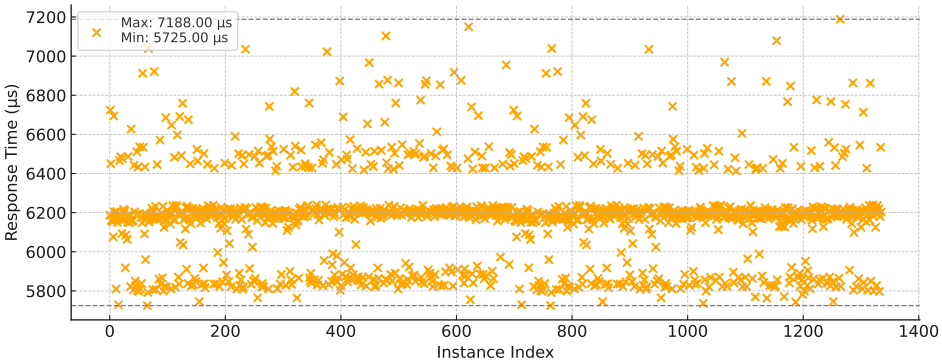
This case and many similar ones demonstrate that fixed-priority preemptive scheduling is functioning correctly in the modified vECU. The presence of preemption-related overhead and interference patterns in the measured data confirms that the RTOS is correctly implementing fixed-priority preemptive scheduling at runtime.

5.3.2 Response Time Results

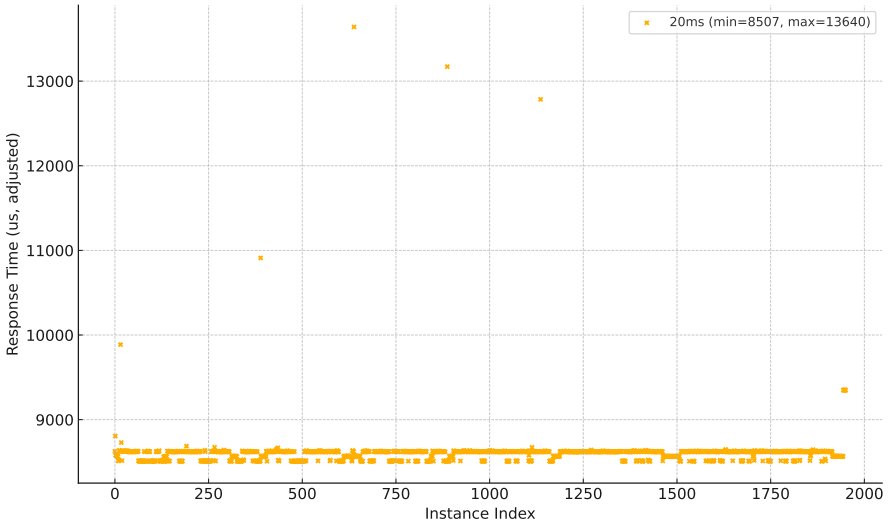
To analyze the behavior of the vECU under the integrated RTOS, the response times of all periodic tasks were measured during execution. These measurements were obtained using custom instrumentation functions developed to read Linux system time and were post-processed to isolate true response time characteristics. Specifically, the response time of each task instance was computed as the elapsed time between its arrival and completion. To highlight key observations, only the 20 ms and 320 ms task response time plots are presented here alongside their corresponding ECU results. The remaining plots for vECU response times are included in Appendix D.

Figure 5.9 shows the response time distribution of the 20 ms task on both the ECU and vECU. At first glance, the vECU plot appears to exhibit only a single narrow horizontal band, in contrast to the ECU plot, which displays a three-band pattern. However, this discrepancy is primarily due to the presence of high-magnitude outliers in the vECU data, which stretch the y-axis and visually compress the main distribution.

Upon closer inspection or zooming in, the vECU plot also reveals a similar three-band pattern, though with slightly different magnitudes compared to the ECU. This similarity confirms that the execution timing structure is preserved across both platforms. The differences in band heights are expected, as the vECU uses the WCET values measured from the ECU to simulate task execution durations. This results in a slightly more uniform task profile, but the underlying preemption behavior and release conditions remain consistent. The high-magnitude outliers in the vECU data are mainly caused by external interference, as the vECU runs as an application on the host system. This issue is explored further in the chapter 6.



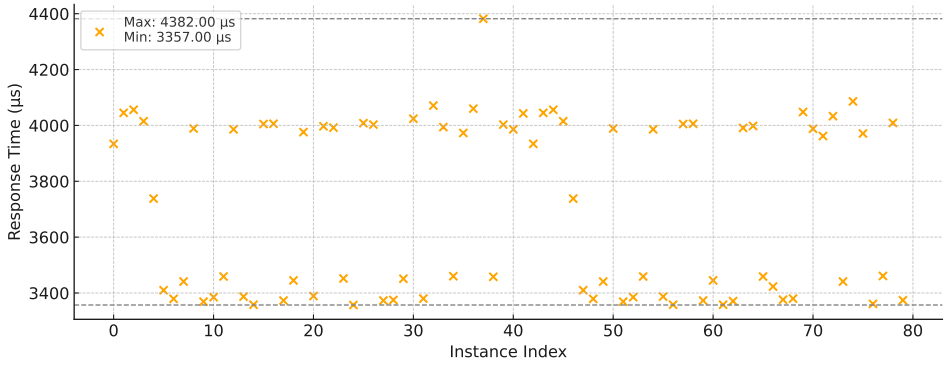
(a) ECU



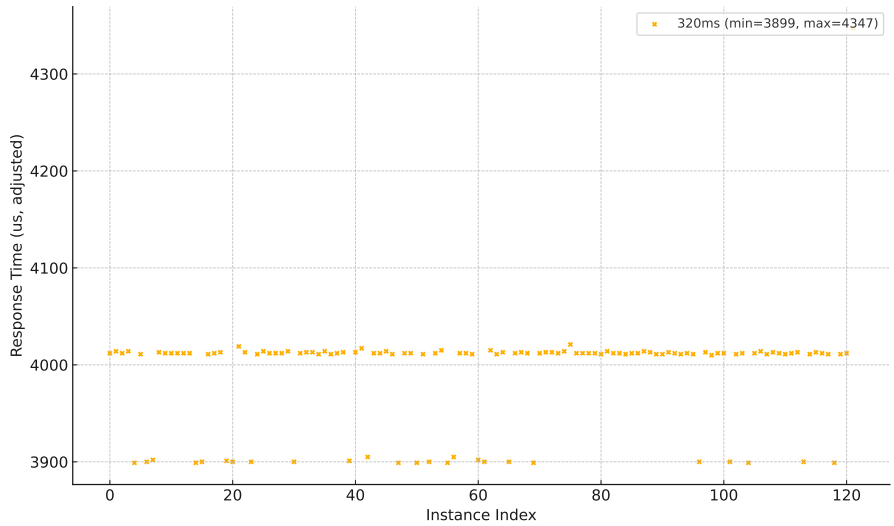
(b) vECU

Figure 5.9: Comparison of ECU and vECU Response Time for the 20 ms task

Figure 5.10 presents the response time distribution of the 320 ms task. On the ECU, two horizontal bands are observed, suggesting different execution paths or varying interference conditions. The vECU plot shows a similar pattern, with a comparable number of data points. The relatively low instance count for both platforms is due to the task's long period and the fixed 25-second measurement window, which naturally limits the number of observable activations. The close match in timing behavior between the ECU and vECU further confirms that the RTOS integration in the vECU effectively reproduces key aspects of real-time scheduling.



(a) ECU



(b) vECU

Figure 5.10: Comparison of ECU and vECU Response Time for the 320 ms task

Table 5.5 summarizes the response time statistics for all periodic tasks measured on the vECU. The table reports the minimum, maximum, mean, and standard deviation of the measured response times in microseconds. As expected, the highest-priority tasks (e.g., 1250 μs and 2500 μs) exhibit low and consistent response times, while lower-priority tasks (e.g., 10 ms and 20 ms) show increased variability due to higher susceptibility to preemption.

Table 5.5: Summary of Response Time Statistics in vECU

Task	Min (μs)	Max (μs)	Mean (μs)	Std Dev (μs)
1250 μs	345	457	346.33	1.74
2500 μs	320	496	376.27	9.68
5 ms	683	1,226	769.28	42.08
10 ms	1,771	8,275	1,871.25	128.55
20 ms	8,507	13,640	8,607.01	200.96
40 ms	508	713	600.63	44.61
80 ms	443	580	539.02	41.92
160 ms	8,065	9,130	8,351.27	74.25
320 ms	3,899	4,347	3,992.20	55.78
1000 ms	551	670	658.82	25.10

5.3.3 Execution Time, Release Jitter and Blocking

Even though we used a non-blocking delay in each task in vECU to simulate the ECU, we still observed variation in exclusive execution time of each task, which are summarized in Table 5.6. Release jitter was measured for all periodic tasks in the vECU to evaluate temporal consistency under the integrated RTOS. The observed jitter values ranged from 0 to approximately 90 μs, with no consistent pattern or periodicity across tasks or instances. This suggests that the jitter is likely due to nondeterministic factors in the underlying execution environment, such as OS scheduling delays or timer resolution.

Despite this variability, the magnitude of jitter remains within acceptable bounds for tasks with periods in the millisecond range. No instances of task blocking were detected during measurement, indicating that all tasks executed without waiting for shared resources under the tested conditions. The observed variation in exclusive execution time, differences in release jitter compared to the ECU, and the absence of task blocking are discussed in more detail in the chapter 6.

Table 5.6: Summary of Exclusive Execution Time for All Tasks in the vECU

Task	Min (μs)	Max (μs)	Mean (μs)	Std Dev (μs)
1250 μs	345	457	346.33	1.74
2500 μs	30	76	31.65	1.26
5 ms	171	479	451.52	3.74
10 ms	895	4,971	1,246.17	61.29
20 ms	2,145	4,261	4,099.75	95.29
40 ms	275	356	276.43	2.93
80 ms	210	218	211.26	0.99
160 ms	3,482	6,829	3,908.07	200.53
320 ms	2,206	2,397	2,360.52	31.74
1000 ms	40	48	41.77	1.27

5.4 Theoretical Response Time Analysis

This section presents the theoretical WCRT estimates for each task, computed using two different analytical approaches. The first approach is the classic RTA, as illustrated in Section 2.4.1, which assumes zero release jitter and fully synchronous task activation. The second approach applies the more comprehensive offset-aware model developed by Redell and Törngren [1], as described in Section 2.4.2, which accounts for task offsets and release jitter.

Table 5.7 summarizes the computed WCRT values for all ten tasks under both models using timing parameters derived from the physical ECU. The jitter-aware (offset-aware) approach yields more conservative bounds for several tasks, particularly those influenced by offset and jitter interactions. This increased conservatism arises because, in the offset-aware model, task instances that were released before the release of the current task instance can still interfere with it, due to the assumption of worst-case release jitter. In contrast, the classic model does not account for such early-released instances in its interference calculations, resulting in tighter but potentially less safe estimates.

Table 5.7: Theoretical WCRT Estimates Using ECU-Derived Parameters

Task	**Period (μs)**	**Classic RTA (μs)**	**offset-aware Model (μs)**
1250us	1,250	393	396
2500us	2,500	429	432
5ms	5,000	1,659	1,662
10ms	10,000	2,924	2,073
20ms	20,000	9,388	10,603
40ms	40,000	9,672	19,921
80ms	80,000	9,890	20,777
160ms	160,000	19,126	29,659
320ms	320,000	33,730	42,373
1000ms	1000,000	34,130	60,933

Table 5.8: Theoretical WCRT Estimates Using vECU-Derived Parameters

Task	**Classic Model (μs)**	**offset-aware Model (μs)**
1250 μs	457	547
2500 μs	533	623
5 ms	1,012	1,026
10 ms	9,899	-
20 ms	-	-
40 ms	-	-
80 ms	-	-
160 ms	-	-
320 ms	-	-
1000 ms	-	-

Table 5.8 presents the corresponding WCRT computations using timing parameters extracted from the vECU. The same two analytical models are applied. Both the

classic model and the offset-aware model yield results only for the higher-priority tasks, while response time computations for lower-priority tasks do not converge within the allowed maximum number of iterations. To better understand the practical applicability of the offset-aware model, Table 5.9 shows WCRT estimates using adjusted vECU execution times that exclude rare, high-magnitude outliers. These outliers are attributed to external interference from the host system, not to the vECU itself or the Trampoline scheduler, and are therefore safe to discard. This adjustment provides a more realistic upper bound for typical execution scenarios and demonstrates that the model can converge when input parameters reflect consistent platform-level behavior.

Table 5.9: Theoretical WCRT Estimates Using vECU Adjusted Parameters

Task	**Classic Model (μs)**	**offset-aware Model (μs)**
1250 μs	457	547
2500 μs	533	623
5 ms	1,012	1,026
10 ms	3,264	2,342
20 ms	13,604	12,444
40 ms	14,417	24,609
80 ms	14,635	27,435
160 ms	36,202	49,677
320 ms	54,463	67,481
1000 ms	54,511	89,909

In addition, the scheduled arrival times of tasks in the vECU were observed to exhibit a small cumulative drift over the measurement interval. While the individual drifts are minor, their accumulation over time can lead to deviations from the intended periodic schedule, potentially affecting synchronization with other tasks or external events. The underlying cause of this phenomenon, along with its implications for temporal accuracy and scheduling stability, is examined in detail in the chapter 6.

6

Discussion

This chapter discusses the timing analysis results presented in Chapter 5, with a focus on interpreting the empirical measurements and evaluating the theoretical predictions. While the ECU and vECU platforms share the same task set and scheduling policy, notable and informative differences in execution behavior, response time patterns, and jitter characteristics were observed. Rather than being problematic, these differences provide valuable insights into how each platform handles timing behavior under varying conditions. They are examined concerning system architecture, scheduling dynamics, and measurement methodology, helping to assess the strengths and limitations of the vECU as a tool for early-stage timing validation.

Before performing the comparative analysis, a validation step was carried out to confirm that the integrated open-source RTOS within the Level 2 vECU enabled preemptive, real-time scheduling as intended. This was verified by observing expected preemption patterns in the trace data, based on known task periods, offsets, and configured execution times. Two types of preemption were identified: one where a task's execution was delayed due to simultaneously arriving higher-priority tasks, and another where a task was interrupted by higher-priority tasks during execution and later resumed. The presence of both behaviors in the measurement data confirms that the RTOS operated correctly within the vECU environment.

Finally, the accuracy and practicality of the classic and offset-aware theoretical response time analysis models are assessed by comparing their predictions to empirical measurements collected from both the ECU and the vECU. In parallel, the vECU's ability to reproduce the timing behavior of the physical ECU is evaluated by analyzing differences in execution time, response time, and jitter characteristics. Through this discussion, key insights are drawn regarding system predictability, model limitations, and the applicability of the integrated vECU for real-time system validation and early-stage analysis.

6.1 Comparison of Timing Behavior Between ECU and vECU

This section compares the timing behavior of the ECU and vECU platforms across three dimensions: exclusive execution time, response time, and release jitter. The goal is to evaluate how closely the vECU replicates the real-time behavior of the physical ECU and to identify sources of divergence that may affect model fidelity or timing predictability.

6.1.1 Exclusive Execution Time Comparison

Table 5.2 and Table 5.6 summarize the exclusive execution time statistics for all tasks measured on the ECU and the vECU, respectively. While the values appear broadly comparable in terms of order of magnitude, it is important to note that the underlying execution environments and task implementations differ significantly between the two platforms.

On the ECU, each task consists of multiple activities, some of which perform direct hardware interactions, such as sensor reads or actuator control, which contribute to execution time variability. In contrast, the vECU implementation inherently replaces all hardware-dependent components with stub functions. For the purposes of this thesis project, a custom non-blocking busy-wait loop was manually inserted into each task to emulate execution time and approximate the durations observed on the ECU. This was done to enable a meaningful comparison of timing behavior across platforms. This results in a different internal workload distribution, where timing is no longer governed by hardware latency but by code execution on a general-purpose Linux system running on an Intel i7 processor.

Table 6.1: Comparison of Exclusive Execution Time Between ECU and vECU (all values in μs)

Task	ECU Max	vECU Max	ECU Std Dev	vECU Std Dev
1250us	354	457	32	1.74
2500us	40	76	5	1.26
5ms	460	479	21	3.74
10ms	1,254	4,971	58	61.29
20ms	4,222	4,261	26	95.29
40ms	284	356	5	2.93
80ms	218	218	5	0.99
160ms	4,070	6,829	62	200.53
320ms	2,360	2,397	9	31.74
1000ms	46	48	1	1.27

Table 6.1 provides a side-by-side comparison of maximum and standard deviation values of exclusive execution time from both platforms. Although the vECU execution times were manually tuned to resemble the ECU's worst-case execution time, observed variation still exists. This variability is likely due to timing imprecision in the busy-wait delay used to emulate execution time, compounded by indirect effects from the host environment. In particular, the higher standard deviation observed in the vECU stems from the fact that it runs as a user-level application on a Linux host system. As such, it is susceptible to interference from other workloads running concurrently on the host. These occasional interruptions can cause high-magnitude outliers in execution time, thereby inflating the standard deviation. Although the Trampoline RTOS inside the vECU enforces deterministic task scheduling, it cannot fully shield task execution from host-level CPU scheduling and resource contention. This effect could potentially be reduced by assigning a higher scheduling priority to the vECU process on the host system. However, due to the scope and limited time of the project, this optimization was not explored.

One notable case is the 10 ms task, where an extreme outlier was observed in each measurement run, with values reaching nearly 5000 μs. While these spikes vary slightly between runs, they consistently appear across executions. The majority of instances, however, cluster around the expected value near 1254 μs, closely matching the ECU behavior. The cause of this isolated but repeatable anomaly remains unclear, but it does not significantly affect the overall profile of the task's execution time. As such, even though the mean execution times in many tasks closely match those on the ECU, the standard deviation tends to be larger in the vECU, particularly in tasks with longer delays (e.g., 10 ms and 160 ms tasks).

These discrepancies highlight that while the vECU is capable of approximating computational load at a coarse level, its timing fidelity is inherently limited by the host environment and the simplified execution structure. Nonetheless, the overall magnitude and pattern of task execution durations remain reasonably aligned, supporting the use of the vECU for approximate timing validation in early development phases.

6.1.2 Analysis of Response Time Discrepancies

The response time behavior observed across the ECU and vECU platforms shows several notable differences, primarily stemming from platform-specific characteristics and the measurement methodology used in each environment, as shown in Table 6.2. The maximum response times recorded in the vECU are consistently higher than those of the ECU. A key reason for this is that, in the vECU, each task's exclusive execution time was manually configured to approximate the WCET measured on the ECU. Although some variation still occurred during vECU runs, all tasks executed near their configured WCETs. Consequently, the level of interference from higher-priority tasks, caused by preemption, also approached the worst-case scenario. Since response time is the sum of a task's exclusive execution time and cumulative interference, the resulting response times in the vECU naturally reflect a worst-case configuration.

In contrast, on the ECU, the worst-case response time observed for a task does not necessarily occur under a system-wide worst-case condition in which all tasks execute at their WCETs simultaneously. Additionally, ECU tasks are composed of multiple internal activities, each treated as a critical section. During the execution of an activity, hardware interrupts are globally disabled until the activity completes. This results in non-preemptive behavior within each activity, effectively limiting the amount of interference a task can experience.

By comparison, the vECU replaces hardware-dependent operations with stub functions, and the remaining code segments are not treated as critical sections. As a result, no interrupt disabling, mutexes, or resource locking mechanisms are employed, allowing full preemption throughout task execution. This leads to greater opportunities for interference in the vECU than in the ECU.

Although the ECU may experience occasional blocking due to resource access (see Table 5.3), the associated delays are relatively small compared to the interference caused by higher-priority tasks. Therefore, the absence of blocking in the vECU does not significantly reduce its response times. Overall, it is expected that the vECU

exhibits longer maximum response times due to its conservative execution configuration, higher degree of preemptability, and deliberate approximation of worst-case behavior. Nevertheless, the vECU remains a valuable tool for early timing analysis, providing a safe and efficient environment to evaluate task response times under controlled, near-worst-case conditions.

Table 6.2: Comparison of Max Response Time and Std Dev (in μs)

Task	Max ECU	Max vECU	Std ECU	Std vECU
1250us	352	457	31	1.74
2500us	217	496	11	9.68
5ms	1,164	1,226	199	42.08
10ms	1,626	8,275	70	128.55
20ms	7,188	13,640	243	200.96
40ms	519	713	15	44.61
80ms	447	580	15	41.92
160ms	6,712	9,130	216	74.25
320ms	4,382	4,347	267	55.78
1000ms	563	670	10	25.10

On the ECU, response time traces were collected using runtime instrumentation that captured actual task start and completion times on an Infineon automotive microcontroller. As a real embedded system with deterministic scheduling and hardware-controlled interrupts, the ECU produced consistent response time profiles. For most tasks, values were tightly clustered. In cases with noticeable variability (e.g., 5 ms and 10 ms tasks), this could be attributed to expected effects of preemption and blocking. Outliers were rare and typically linked to specific scheduling scenarios involving heavy interference.

In contrast, the vECU operates as a user-level application on a Linux host and uses the Trampoline RTOS for deterministic task scheduling. However, the surrounding host environment does not enforce strict real-time behavior. Time measurements rely on Linux system clock APIs, which introduce some imprecision. Moreover, part of the task execution is emulated using busy-wait loops, making the system susceptible to external influences such as CPU load and host OS scheduling. For instance, the 10 ms task in the vECU exhibited an extreme outlier in response time that consistently appeared across different measurement runs with slightly varying magnitudes. This suggests a recurring but rare timing artifact rather than a measurement error. The majority of instances remained tightly grouped around 1,254 μs, which corresponds to the configured delay used to emulate the task's WCET through a busy-wait loop.

In general, response time variability in the vECU is more pronounced for mid- and low-priority tasks, especially those with longer configured execution times. This is because higher-priority tasks experience less interference from other tasks and are less sensitive to potential overestimation of their WCETs. In contrast, lower-priority tasks are more vulnerable to cumulative interference and timing inaccuracies. The lack of hardware-level determinism, combined with imprecise emulation and limited measurement granularity, further contributes to the observed variability. Nonethe-

less, response time patterns in the vECU generally align with those on the ECU in terms of task priority ordering and response magnitude. This indicates that the vECU, despite its limitations, remains a useful tool for early-stage timing validation.

6.1.3 Release Jitter and Blocking Effects

Release jitter was measured for all periodic tasks on the vECU to assess temporal consistency under the integrated RTOS. The observed jitter values ranged from 0 to approximately 90 μs, with no apparent pattern or periodicity across instances. This suggests that the jitter likely stems from nondeterministic factors in the underlying software environment, such as instrumentation overhead or variations in host scheduling behavior, rather than from the RTOS scheduler itself. In comparison, the ECU exhibited significantly lower jitter, as expected from a hardware-based real-time system with accurate timers and minimal background interference. The tighter timing precision on the ECU contributes to more consistent task activation and scheduling behavior.

These variations in the vECU can be reduced or even eliminated by increasing the execution priority of the vECU process on the host system, thereby minimizing interruptions from other host workloads. Additionally, employing more efficient instrumentation tools with lower runtime overhead would further improve temporal consistency by reducing measurement-induced perturbations.

Blocking was also investigated. On the ECU, some tasks experienced measurable blocking times due to the system's internal handling of critical sections. Each task consists of multiple activities, and during the execution of an activity, interrupts are disabled to ensure atomicity, preventing preemption by higher-priority tasks. Table 5.3 lists the tasks where nonzero blocking was observed. How these blockings were computed is demonstrated in Section 3.2. No blocking was recorded on the vECU. This is consistent with its software architecture, in which critical sections are absent, and all tasks execute in a fully preemptible environment. Most hardware interactions are stubbed out, and no resource-locking mechanisms were used in the measurement setup.

In addition to random jitter, a small but accumulating drift was observed in the scheduled arrival times of tasks in the virtual ECU (vECU). This drift originates from the way the Trampoline RTOS handles periodic task activation. Each task is linked to an `Alarm` object, which, when triggered, activates the task and then resets itself for the next period. Ideally, the alarm reset should occur immediately upon triggering to maintain a consistent periodic schedule. However, the reset operation incurs a small execution delay. As a result, each successive trigger occurs slightly later than the ideal time point, leading to a gradual drift in task activation times. This drift can be further exacerbated by CPU load, instrumentation overhead, and timing inaccuracies inherent in the host simulation environment. By contrast, the physical ECU uses high-precision hardware timers that provide deterministic behavior and avoid such drift.

In the vECU, alarms are triggered based on a unit called a *date*, representing the smallest clock tick in the Trampoline OS. To compensate for the reset overhead

and maintain accurate periodic timing, one approach would be to set the alarm interval slightly shorter, specifically, to the desired task period minus the measured reset overhead. This would effectively counteract the drift by accounting for the delay introduced during each reset. However, due to the limited time resolution and the measurement overhead in our current setup, implementing such fine-grained adjustments was not feasible.

Overall, while the vECU eliminates blocking through full preemptibility, it exhibits slightly higher and more irregular jitter. These results underscore key differences in platform-level timing behavior, which must be considered when using the vECU for real-time validation or model-based analysis.

6.2 Theoretical Model Evaluation

This section evaluates the applicability and accuracy of theoretical RTA models introduced in Chapter 3. Both the classic RTA model and the offset-aware model were used to compute WCRT estimates for the task set, based on measured execution times from the ECU and vECU platforms. The goal of this evaluation is to compare the theoretical predictions to the empirical response times observed in Chapter 5, assess the models' accuracy and robustness, and provide guidance on their practical use in real-time system validation.

6.2.1 Comparison with Empirical Measurements

Tables 5.7 and 5.9 summarize the WCRT values predicted by both the classic and offset-aware RTA models, based on timing parameters derived from the ECU and vECU. These theoretical estimates are compared to the measured maximum response times reported in Table 6.2.

For the ECU, both analytical models produce response time bounds that are safely above the measured maximums. However, these bounds are relatively pessimistic, particularly for low- and mid-priority tasks. A key reason for this overestimation is that the task set on the ECU is carefully constructed with harmonic periods and strategically selected offsets. As a result, simultaneous arrivals of multiple tasks—especially those that could cause significant interference—are largely avoided. The classic RTA model, based on the foundational assumptions by Liu and Layland [9], considers the worst-case scenario where all higher-priority tasks are released simultaneously with the task under analysis. Similarly, the offset-aware model, while incorporating offsets and jitter, still assumes the most adverse phasing between tasks to compute worst-case interference [1]. These conservative assumptions help ensure safety but often do not reflect the more optimized and asynchronous behavior observed in practical, well-engineered systems.

From Equations 4.1 to 4.10 in Section 3.2, we observe that task 1250 μs arrives at the same time as all lower-priority tasks in every instance. Beyond that, only task 40 ms is occasionally aligned with task 1000 ms. While this pattern may not be immediately obvious, simultaneous arrivals occur only when two task activation equations have a common solution with integer instance indices. Among all the

task pairs, only the 1250 μs task satisfies this condition consistently with all lower-priority tasks, and the 40 ms task does so intermittently with the 1000 ms task. This carefully designed timing structure eliminates most synchronous release scenarios, thereby significantly reducing cumulative interference from higher-priority tasks.

To explain this further, for two tasks to arrive simultaneously, their respective arrival time equations must be equal and yield integer solutions for their instance indices. From the listed equations, it is evident that only the 1250 μs task satisfies this condition with all lower-priority tasks, and the 40 ms task does so only with the 1000 ms task.

Nevertheless, the classic RTA model is based on the worst-case assumption that all higher-priority tasks release synchronously with the task under analysis. This results in a pessimistic WCRT estimate that does not reflect the actual runtime conditions observed on the ECU. Similarly, although the offset-aware model includes offsets and jitter in its formulation, it still considers highly conservative release scenarios in which the worst-case phasing of task activations is assumed, as illustrated in Section 2.4.2. As a result, it too provides bounds that are looser than the empirical response times.

For the vECU, additional factors come into play. When using raw execution time measurements (including outliers), the offset-aware model fails to converge for several tasks due to high variability and extreme values (e.g., the 10 ms task reaching nearly 5000 μs). These outliers are not representative of the task's actual WCET, as execution in the vECU is simulated using non-blocking delays intended to mimic the WCET observed on the physical ECU. The outliers arise from the fact that the vECU runs as a user-space application on a general-purpose host system, where interference from background processes and OS scheduling can lead to occasional, artificial timing spikes. Therefore, removing these outliers is considered safe, as they reflect host-level artifacts rather than intrinsic task behavior. Once the parameters are adjusted accordingly, the models regain convergence and the estimated WCRT values align more closely with empirical results.

These findings demonstrate that theoretical models can produce valid safety bounds, but their conservatism grows in the presence of harmonic task sets with asynchronous releases. Careful task offset design, as seen in the ECU setup, can significantly mitigate interference and improve system behavior in practice. However, this beneficial effect is not fully captured by worst-case analytical models. In particular, offset-based analysis is inherently complex, and no exact schedulability test is known for the general case. As a result, existing models often resort to conservative approximations that may not reflect the optimized timing behavior achievable through well-engineered task phasing.

6.2.2 Model Strengths and Limitations

The classic RTA model offers a lightweight and computationally efficient approach to worst-case response time estimation. It assumes fully synchronous task releases and zero release jitter, which simplifies analysis but may lead to pessimistic bounds—especially in systems with offsets or asynchronous task behavior. For high-

priority tasks, this model often suffices, as they are less affected by interference or offset chains.

The offset-aware model improves upon the classic approach by accounting for task-specific offsets and release jitter. This enables more accurate modeling of real-world behavior, particularly in systems using offset-aware scheduling or exhibiting non-synchronous task activation patterns. However, this increased conservatism and modeling granularity come at the cost of higher computational complexity and greater sensitivity to input data.

In the vECU, where execution time variation is larger and some tasks experience outliers or timing irregularities, the offset-aware model may fail to converge or produce overly pessimistic bounds. This is particularly evident for lower-priority tasks, where accumulated interference from higher-priority tasks—combined with jitter and complex offset interactions—can introduce feedback cycles during analysis. The model assumes worst-case phasing between tasks, which, while safe, often does not reflect typical runtime behavior. To improve its accuracy, one should consider using tighter, statistically-informed execution time bounds, filtering out extreme outliers, and adjusting the model assumptions to account for strategically configured task offsets that minimize alignment and reduce actual interference. Additionally, hybrid approaches that calibrate model parameters using runtime traces can help bridge the gap between conservative predictions and observed behavior. In early-stage virtual environments where timing stability is limited, these refinements are especially important to ensure the model remains both practical and informative.

In summary, while the classic model is simple and robust under idealized assumptions, it may over-predict WCRT. The offset-aware model is theoretically stronger but more sensitive to input variability, such as outliers in execution time. Choosing between them requires a tradeoff between conservatism, prediction accuracy, and the stability of input parameters such as execution times, offsets, and jitter.

6.2.3 Interpretation and Practical Utility

Model-based timing analysis can offer valuable insights, but its reliability depends on how well the assumptions align with real system behavior. Models tend to be reliable when execution times are well-characterized, scheduling behavior is deterministic, and the hardware platform matches the modeled environment—for example, using fixed-priority scheduling with bounded interference on validated HIL setups. Nonetheless, caution is needed when applying these models to virtual platforms, systems with high jitter, or configurations with dynamic behaviors that are difficult to model accurately. Discrepancies may arise from optimistic assumptions about synchronization, communication delays, or interrupt handling.

In practice, we recommend using models primarily for design-time validation, where conservative assumptions help verify feasibility and guide architectural decisions. For runtime verification, models should be complemented with trace analysis, instrumentation, or monitoring to validate assumptions and detect timing anomalies under real operating conditions. A hybrid approach ensures that insights from modeling remain grounded in practical observability and system evolution.

While commercial platforms offer similar modeling capabilities, this work demonstrates that accurate response-time analysis can also be integrated into an open-source vECU environment. This provides a flexible and cost-effective alternative for exploring real-time behavior in early development phases. The successful application of theoretical models—even with their limitations—shows the potential of combining model-based analysis with open virtual platforms to support timing validation in both academic research and industry use cases, especially where hardware access is limited or timing predictability must be assessed early.

6.3 Limitations and Recommendations for Future Work

While this thesis demonstrates the feasibility of integrating an open-source RTOS (Trampoline) into an existing L2 vECU platform to enable preemptive real-time scheduling, several limitations remain that impact both the accuracy of the vECU and the robustness of the analytical models used for validation.

One key limitation is the timing precision of the vECU, which runs on a general-purpose Linux host with the PREEMPT_RT patch applied. While the real-time patch improves scheduling determinism, some variability persists due to residual OS overhead, timer granularity, and background system activity. This is particularly noticeable in short-duration tasks, where even microsecond-level jitter can affect measurements. Additionally, cumulative drift in periodic task releases remains a concern. Further improvements could involve hardware-assisted tracing, finer-grained timer calibration, or implementing time-drift compensation mechanisms within the Trampoline scheduler.

Second, the vECU currently simulates task workloads using busy-loop delays, which fail to capture the distribution and structure of actual ECU computational behavior. Replacing these with instruction-level load modeling or trace-replay mechanisms would provide a more accurate representation of ECU workloads.

Third, the lack of sophisticated intra-task resource management in the vECU prevents it from modeling non-preemptive sections and resource locking effects that are common in real ECUs. While it is technically possible to implement blocking delays for certain task segments, a non-blocking delay approach was used in this thesis for simplicity. Blocking delays could be incorporated, but doing so would require a more thorough understanding of the ECU's workload distribution and careful tuning of delay durations to reflect realistic execution timing. Implementing mutex-based or priority ceiling protocols in the vECU would enable simulation of priority inversion scenarios and more realistic preemption behavior.

Fourth, the current Trampoline integration in the vECU is configured for single-core operation for simplicity, although the RTOS itself supports multi-core configurations. Expanding the integrated vECU to utilize multi-core capabilities in future work would allow for more realistic modeling of modern ECU architectures and enable analysis of inter-core scheduling and synchronization effects.

Fifth, in the current vECU setup, Trampoline is configured to operate in single-core mode and replaces the legacy scheduler only on core 1. However, the legacy scheduler continues to run on the remaining cores. As a result, the Trampoline initialization sequence is somewhat coupled with that of the legacy scheduler, due to shared platform state and initialization order dependencies. This hybrid setup was chosen for simplicity, but it introduces architectural complexity and potential timing inconsistencies. For instance, during testing, there were cases where the state machine of the `NodeManager` submodule was not initialized correctly due to race conditions during startup. As a result, some task activities were not triggered as expected, leading to incomplete or misleading execution traces. Since accurate task activation is critical for both timing measurements and response-time analysis, such initialization issues directly affect the reliability of the validation process. In future work, this could be addressed either by configuring Trampoline in multi-core mode to fully replace the legacy scheduler across all cores, or by refactoring the initialization process to decouple the schedulers cleanly. This would improve modularity, reduce cross-scheduler interference, and better reflect modern ECU deployment scenarios.

Lastly, the response-time analysis models, while useful for bounding worst-case behavior, remain conservative and often diverge from empirical measurements, particularly in systems with designed offset strategies and harmonic task sets. While jitter can capture some dynamic delays in task release, current analytical models often do not explicitly account for OS-level phenomena such as context-switch overhead, interrupt latency, or scheduler-induced delays. These effects can compound in complex systems and reduce the accuracy of worst-case response time predictions, especially in early-stage or non-deterministic environments. Future work should explore model enhancements that incorporate these effects, as well as hybrid approaches that calibrate analytical models using runtime trace data.

Together, these improvements will increase the fidelity of the vECU as a timing validation platform and enhance the practical relevance of analytical methods in real-world embedded system design.

6.4 Ethical Considerations

This project adhered to standard ethical practices throughout the development and reporting phases. All performance data were obtained from system-internal instrumentation on test platforms provided by Volvo Group. No personal or user-specific information was collected at any stage. Any data or implementation details subject to confidentiality have either been omitted from the report or anonymized using alias names in agreement with Volvo's internal policies.

Measurement logs were stored locally in structured formats and analyzed using controlled tools to ensure responsible data handling and traceability. In addition, while generative AI tools (e.g., ChatGPT) were occasionally used for language refinement or clarification of general technical concepts, all analytical content, experiments, and findings were developed independently by the author. These measures helped ensure the integrity, transparency, and compliance of the work with both academic and industrial ethical standards.

7

Conclusion

This thesis investigated the feasibility and effectiveness of integrating an open-source RTOS into an existing Level 2 vECU to enable fully preemptive, real-time scheduling. By doing so, the vECU is augmented with time-aware simulation features that enable it to replicate the functional behavior of automotive applications, thereby supporting early-stage software development and validation. The work also evaluated how accurately the modified vECU reflects the timing behavior of a physical ECU and how well theoretical RTA models align with both empirical platforms.

The integration of the open-source Trampoline RTOS into the vECU demonstrated the feasibility of achieving preemptive real-time scheduling using entirely open-source software, offering an alternative to proprietary commercial solutions. While real-time scheduling in vECUs is not an entirely new idea and is already supported by several industrial platforms, this work responds to a need for greater flexibility, reduced licensing costs, and deeper control over the scheduling infrastructure. The integration was configured for single-core operation, replacing the legacy scheduler only on one core, while other cores continued to run the legacy scheduler. This setup introduced mild coupling between initialization sequences across multiple cores, highlighting the need for either multi-core expansion or further architectural decoupling in future iterations. Trace-based verification confirmed that the expected preemption behavior occurred as configured, aligning with fixed-priority preemptive scheduling semantics.

Comparative analysis between the vECU and ECU revealed that while the vECU approximates the overall structure and magnitude of task execution and response times, important discrepancies remain. These stem from the use of busy-wait loops for task execution emulation, limitations in host-based timing precision despite the `PREEMPT_RT` patch, and the absence of resource locking or non-preemptive sections that are present in the real ECU. In particular, the integrated vECU exhibited higher response time variability and occasional timing drift, especially for mid- and low-priority tasks.

Theoretical RTA models—both classic and offset-aware—were found to provide safe upper bounds on response times but tended to be overly conservative, especially in systems designed with harmonic task sets and offset strategies that mitigate worst-case interference. Moreover, neither model currently captures OS-level phenomena such as context-switch overhead, interrupt latency, or scheduler behavior, all of which affect timing predictability in real systems.

This platform has strong potential to support software development teams in the

automotive sector by enabling early simulation of timing behavior during control algorithm design, integration, and testing. Developers can explore task interactions, identify timing bottlenecks, and adjust configurations before hardware is available—reducing costly iterations later in the cycle. Moreover, timing parameters derived from the vECU, such as response times, execution times, and scheduling traces, can guide not only validation but also system-level tuning, task restructuring, and safety argumentation in functional safety contexts.

Beyond technical validation, this thesis also opens up practical opportunities for broader use. Timing information—when interpreted correctly—can serve as a "spice rack" for embedded development: adding just enough "salt" (realism) and "sugar" (predictability) to strike a balance between performance and safety. Other industries such as industrial automation or robotics could benefit from this work by adapting the methodology to simulate and analyze real-time behavior in their own domains. The ability to extract real-time characteristics from a virtual platform without full hardware access supports more agile, scalable, and safety-aware development workflows.

Overall, this thesis shows that a modified vECU with an integrated RTOS can serve as a viable platform for early-stage real-time validation, especially when used in conjunction with analytical models. However, further improvements are needed to increase the fidelity and applicability of this approach. Key future directions include enhancing the vECU's workload modeling, supporting multi-core configurations, decoupling the initialization sequence, and refining analytical models to better reflect system-level timing effects. Together, these enhancements will strengthen the bridge between virtual validation environments and real-world embedded system behavior.

Glossary

dSPACE A company of tools and solutions for developing and testing embedded control systems, especially in the automotive and aerospace industries. Founded in Germany, dSPACE offers both hardware and software platforms for rapid prototyping, Hardware-in-the-Loop (HIL) simulation, and virtual validation.

earliest-deadline-first A dynamic priority scheduling algorithm used in RTOS to place processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process is the next to be scheduled for execution.

ETAS A German company that provides tools and solutions for the development of automotive embedded systems, with a strong focus on software engineering, AUTOSAR, real-time systems, and security. It is a subsidiary of the Bosch Group and supports the entire embedded software development process.

harmonic task A harmonic task set is a collection of periodic tasks whose periods are integer multiples of each other.

ISOLAR-EVE A tool developed by ETAS for virtual ECU (vECU) simulation and Software-in-the-Loop (SiL) testing in the automotive domain. It enables developers to execute and test embedded software without physical hardware, accelerating the development and integration process.

preemptive scheduling A type of CPU scheduling in which the operating system can interrupt (preempt) a currently running task in order to assign the CPU to another task with higher priority. This approach allows real-time systems to respond promptly to time-critical tasks.

rate-monotonic A priority assignment algorithm used in RTOS with a static-priority scheduling class. The static priorities are assigned according to the cycle duration of the task, a shorter cycle duration results in a higher task priority.

real-time It means the system must respond to events or inputs within a specific and predictable time limit.

response-time The response time of a task is the time interval between when the task is released (ready to run) and when it finishes execution..

software-in-the-loop A testing technique used in embedded systems and model-based development where software components are tested in a simulated environment, before any real hardware is involved.

VEOS A tool developed by dSPACE. VEOS stands for Virtual Electronic Control Unit Simulation. It is a PC-based simulation environment that executes models of ECUs (electronic control units) and entire vehicle systems, enabling virtual validation.

Bibliography

[1] O. Redell and M. Torngren, "Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter," 2002. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=c65fe6c7-a4ea-3cd8-a702-d71dcce50265

[2] Prostep IVIP Association, "Smart Systems Engineering Requirements for the Standardization of Virtual Electronic Control Units (v-ECUs)," White paper, 03 2020, prostep IVIP. [Online]. Available: https://www.ps-ent-2023.de/fileadmin/prod-download/WhitePaper_V-ECU_2020_05_04-EN.pdf

[3] D. Von Wissel, Y. Jordan, A. Dolha, and J. Mauss, "Full Virtualization of Renault's Engine Management Software and Application to System Development," *arXiv preprint arXiv:1802.06841*, 2018.

[4] *OSEK/VDX Operating System Specification*, 2nd ed., OSEK Group, July 2005, standard specification for automotive real-time operating systems. [Online]. Available: http://www.osek-vdx.org/

[5] J.-L. Bechennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification," in *2006 IEEE Conference on Emerging Technologies and Factory Automation*, 2006, pp. 62–69.

[6] S.-h. Seo, S.-w. Lee, S.-h. Hwang, and J. W. Jeon, "Analysis of Task Switching Time of ECU Embedded System ported to OSEK(RTOS)," in *2006 SICE-ICASE International Joint Conference*, 2006, pp. 545–549.

[7] R. Keil, J. A. Tschorn, J. Tümler, and M. E. Altinsoy, "Evaluation of SiL Testing Potential—Shifting from HiL by Identifying Compatible Requirements with vECUs," *Vehicles*, vol. 6, no. 2, pp. 920–948, 2024, electronic resource, English. Accession Number: edsdoj.710606783d744debb1eb709f57ed8895. [Online]. Available: https://doaj.org/article/710606783d744debb1eb709f57ed8895

[8] S. Ågren, E. Knauss, P. Giusto, G. Soremekun, R. Heldal, and D. Damian, "The Automotive Virtual Verification Ecosystem: Impediments and Enablers," *IEEE Access*, 2020, dOI: 10.1109/ACCESS.2020.3043665. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=d0737324-9b6c-3c89-b87a-f04a2dbf47e5

[9] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20,

no. 1, pp. 46–61, 1973. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=17caa7dc-73ae-348d-a2a7-1f12d80e7594

[10] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline scheduling for real-time systems: EDF and related algorithms*. Springer Science & Business Media, 1998, vol. 460.

[11] Y. Xu, A. Cervin, and K. Årzén, "Harmonic Scheduling and Control Co-Design," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2016, pp. 182–187. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=4e317c7a-220e-364a-851e-78c5549b9b5d

[12] S. J. Lim, J. H. Kim, and J. W. Jeon, "Analysis of EDF and RM scheduling algorithms for periodic and aperiodic tasks in multi-core ECU," in *2023 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC)*, 2023, pp. 1–6.

[13] T. H. C. Nguyen, W. Grass, and K. Jansen, "Exact Polynomial Time Algorithm for the Response Time Analysis of Harmonic Tasks with Constrained Release Jitter," 2019. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=621f7879-eb2d-3027-916b-aaab2e12ba29

[14] D. V. R. Sudhakar, K. Albers, and F. Slomka, "Generalized and Scalable Offset-Based Response Time Analysis of Fixed Priority Systems," *Journal of Systems Architecture*, vol. 112, no. null, 2021. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=7a4f3f7c-8173-3c05-beef-7b6acc4d0032

[15] L. Du and G. Xu, "Worst Case Response Time Analysis for CAN Messages with Offsets," in *2009 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. IEEE, 2009, pp. 41–45, accessed: 2025-01-27. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=4c24401f-7c2b-3873-b916-7590e7741bb4

[16] N. Navet and F. Simonot-Lion, Eds., *Automotive Embedded Systems Handbook*, 1st ed. Boca Raton: CRC Press, 2009. [Online]. Available: https://doi.org/10.1201/9780849380273

[17] V. Bandur, G. M. K. Selim, V. Pantelic, and M. S. Lawford, "Making the Case for Centralized Automotive E/E Architectures," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1230–1245, 2021.

[18] L. Mauser and S. Wagner, "Centralization potential of automotive E/E architectures," *arXiv preprint arXiv:2409.10690*, 2024. [Online]. Available: https://arxiv.org/abs/2409.10690

[19] L. Wen, M. Rickert, F. Pan, J. Lin, Y. Zhang, T. Betz, and A. Knoll, "Virtualization & Microservice Architecture for Software-Defined Vehicles: An Evaluation and Exploration," *arXiv preprint arXiv:2412.09995*, 2024. [Online]. Available: https://arxiv.org/abs/2412.09995

[20] Z. Fei, M. Andersson, and A. Tingberg, "Correlation of Software-in-the-loop Simulation with Physical Testing for Autonomous Driving," 2024. [Online]. Available: https://arxiv.org/abs/2406.03040

[21] L. Sha and J. B. Goodenough, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks," also published in *ACM SIGAda Ada Letters*, Vol. VIII, No. 7, 1988.

[22] L. GmbH, "TRACE32: JTAG Debugger & Trace Solutions," n.d., accessed: 2025-06-27. [Online]. Available: https://www.lauterbach.com

[23] A. Gupta and A. Gupta, *Docker: Practical Guide for Developers and DevOps Teams*. Apress, 2020.

A

Appendix 1

This appendix presents the detailed derivation of the extended response time analysis model based on the work of Redell and Törngren [1]. The goal is to compute the worst-case response time (WCRT) of a task instance τ_{ik} using the following extended formulation:

$$R_{ik} = W_{ik}^{C} + J_i \tag{A.1}$$

$$W_{ik}^{C} = B_i + C_i^{max} + I_0(\tau_{ik}) + I_1(\tau_{ik}) + I_2(\tau_{ik}) \tag{A.2}$$

The terms are defined as follows:

- B_i: Blocking time,
- $C_i^{\max}$: Maximum execution time of task τ_i,
- J_i: Maximum release jitter of τ_i,
- I_0, I_1, I_2: Interference from higher-priority tasks in sets S_0, S_1, and S_2.

Phase and Task Set Classification

Let r_{ik} be the release time of task instance τ_{ik}. The set of interfering higher-priority task instances are partitioned as:

- S_0: Arrive before r_{ik} and can only be released before r_{ik}.
- S_1: Arrive before r_{ik} but may be released at r_{ik} due to jitter.
- S_2: Arrive at or after r_{ik}.

$$\phi_{ik,j} = r_{jp} - r_{ik}, \quad 0 \leq \phi_{ik,j} < T_j \tag{A.3}$$

$$a_{jp} = r_{jp} = p \cdot T_j + O_j \tag{A.4}$$

$$0 \leq p \cdot T_j + O_j - r_{ik} < T_j, \quad p \in \mathbb{Z} \tag{A.5}$$

$$p = \left\lceil \frac{r_{ik} - O_j}{T_j} \right\rceil \tag{A.6}$$

$$\phi_{ik,j} = \left\lceil \frac{r_{ik} - O_j}{T_j} \right\rceil \cdot T_j - (r_{ik} - O_j) \tag{A.7}$$

Interference from S_1

$$J_{j(p-N)} + \phi_{ik,j} = N \cdot T_j \tag{A.8}$$

$$J_{j(p-N)} \leq J_j < J_{j(p-N)} + T_j \tag{A.9}$$

$$\frac{J_j + \phi_{ik,j}}{T_j} - 1 < N_{ik,j} \leq \frac{J_j + \phi_{ik,j}}{T_j} \tag{A.10}$$

$$N_{ik,j} = \left\lfloor \frac{\phi_{ik,j} + J_j}{T_j} \right\rfloor \tag{A.11}$$

$$I_1(\tau_{ik}) = \sum_{j=1}^{i-1} \left\lfloor \frac{\phi_{ik,j} + J_j}{T_j} \right\rfloor \cdot C_j^{\max} \tag{A.12}$$

Interference from S_2

$$N_{S_2} = \left\lceil \frac{W_{ik}^C - \phi_{ik,j}}{T_j} \right\rceil \tag{A.13}$$

$$I_2(\tau_{ik}) = \sum_{j=1}^{i-1} \left\lceil \frac{W_{ik}^C - \phi_{ik,j}}{T_j} \right\rceil \cdot C_j^{\max} \tag{A.14}$$

Interference from S_0

$$r_{j(p-N_{ik,j}-1)} = r_{ik} + \phi_{ik,j} - N_{ik,j} \cdot T_j + J_j \tag{A.15}$$

$$\delta_{ik,j} = r_{ik} - r_{j(p-N_{ik,j}-1)} \tag{A.16}$$

$$\delta_{ik,j} = \left(\left\lfloor \frac{\phi_{ik,j} + J_j}{T_j} \right\rfloor + 1 \right) \cdot T_j - (\phi_{ik,j} + J_j) \tag{A.17}$$

$$\delta_{ik,j} = T_j - ((\phi_{ik,j} + J_j) \bmod T_j) \tag{A.18}$$

Windowed Interference Computation

$$n_j(\tau_{ik}, t) = \left\lfloor \frac{t - \delta_{ik,j}}{T_j} \right\rfloor + 1, \quad t \geq \delta_{ik,j} \tag{A.19}$$

$$W^P(\tau_{ik}, t) = \sum_{j=1}^{i} n_j(\tau_{ik}, t) \tag{A.20}$$

$$\Delta_{ik}(t) = \max(W^P(\tau_{ik}, t) - t, 0) \tag{A.21}$$

$$I_0(\tau_{ik}) = \max_{t>0} \Delta_{ik}(t) \tag{A.22}$$

$$w^{n+1} = \sum_{j=1}^{i} n_j(\tau_{ik}, w^n) \cdot C_j^{\max} \tag{A.23}$$

$$t = \delta_{ik,j} + l \cdot T_j, \quad \forall l \geq 0 \tag{A.24}$$

$$l_j^{\max} = \left\lfloor \frac{w^* - \delta_{ik,j}}{T_j} \right\rfloor \tag{A.25}$$

$$\begin{aligned} I_0(\tau_{ik}) &= \max_{t \in \Lambda(\tau_{ik})} \left(\Delta_{ik}(t)\right), \\ \text{where} \quad \Lambda(\tau_{ik}) &= \left\{ \delta_{ik,j} + l \cdot T_j \;\middle|\; j = 1, 2, \ldots, i;\ l = 0, 1, \ldots, l_j^{\max} \right\} \end{aligned} \tag{A.26}$$

Upper Bound of Busy Period

$$\phi_{t_c,j} = \left\lceil \frac{J_j}{T_j} \right\rceil \cdot T_j - J_j \tag{A.27}$$

$$L_i^U = \sum_{j=1}^{i} \left(\left\lceil \frac{J_j}{T_j} \right\rceil + \left\lceil \frac{L_i^U - \phi_{t_c,j}}{T_j} \right\rceil \right) \cdot C_j^{\max} + B_i \tag{A.28}$$

$$L_i^0 = \sum_{j=1}^{i} \left(\left\lfloor \frac{J_j}{T_j} \right\rfloor + 1 \right) \cdot C_j^{\max} + B_i \tag{A.29}$$

B

Appendix 2

This appendix provides the complete set of response time scatter plots for all periodic tasks. Tasks included here showed minimal variation or no significant anomalies and were therefore omitted from the main discussion in Section 5.2.1 for brevity.

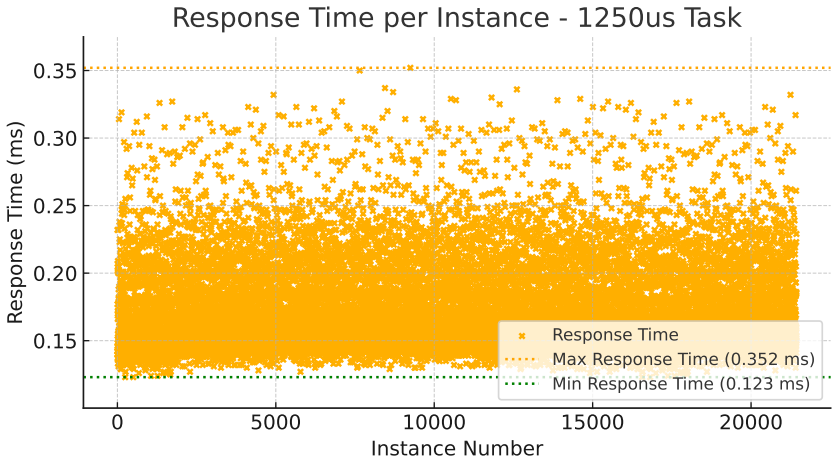


Figure B.1: Response time per instance for the 1250 us task

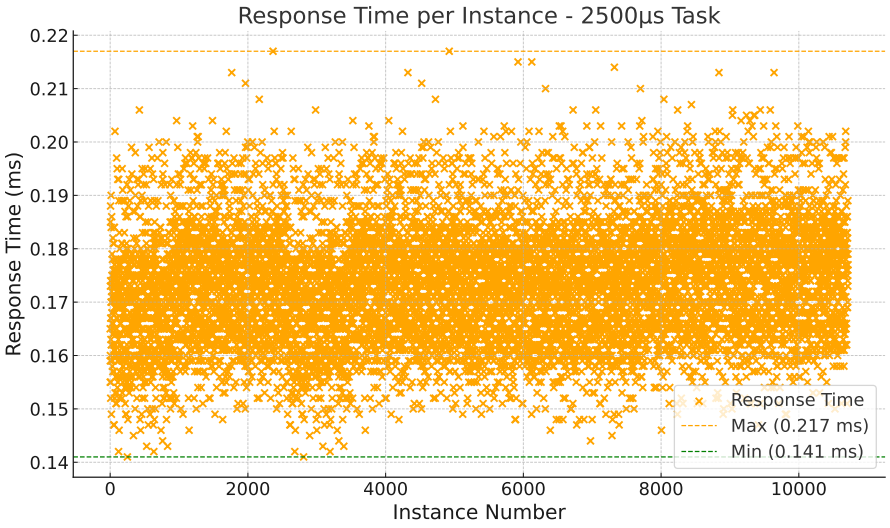


Figure B.2: Response time per instance for the 2500 us task

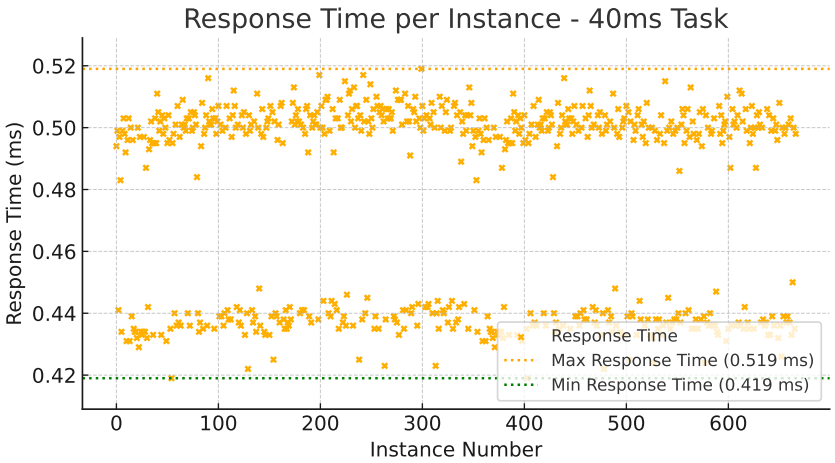


Figure B.3: Response time per instance for the 40 ms task

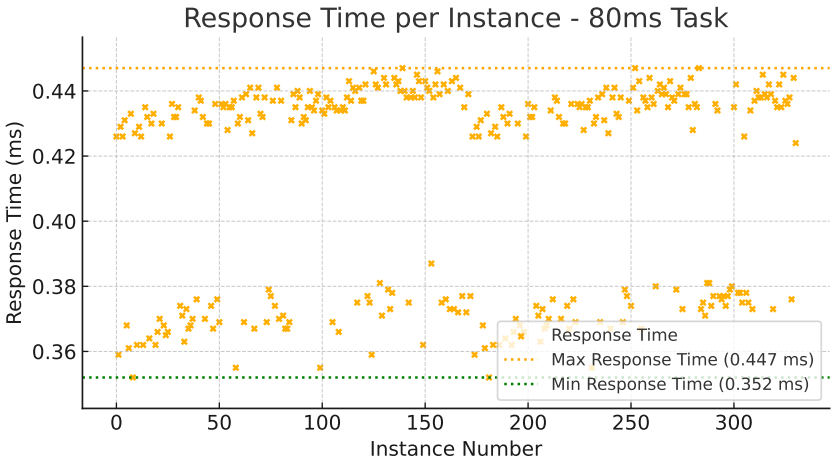


Figure B.4: Response time per instance for the 80 ms task

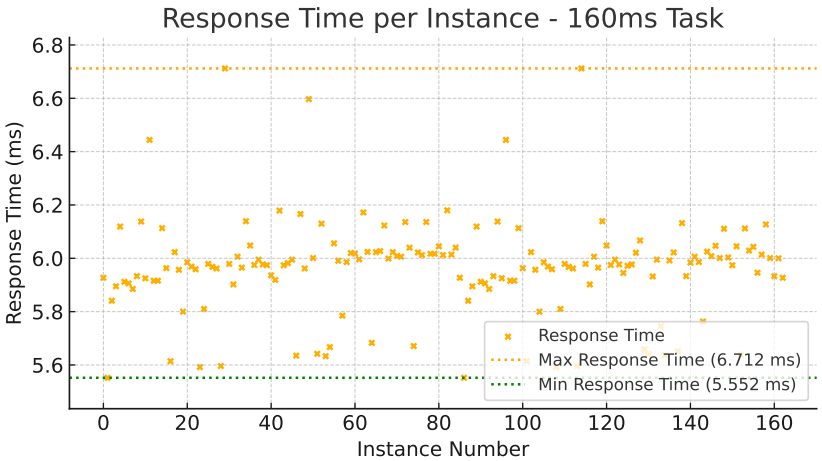


Figure B.5: Response time per instance for the 160 ms task

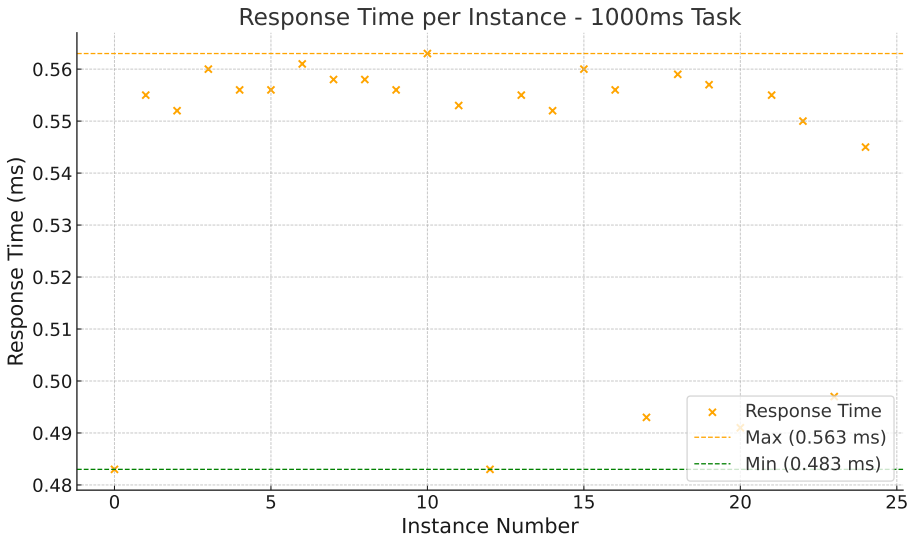


Figure B.6: Response time per instance for the 1000 ms task

C

Appendix 3

This appendix contains the exclusive execution time distribution plots for the remaining tasks not shown in the main results section. These tasks exhibited simple unimodal distributions with very low variation, indicating consistent computational behavior across instances.

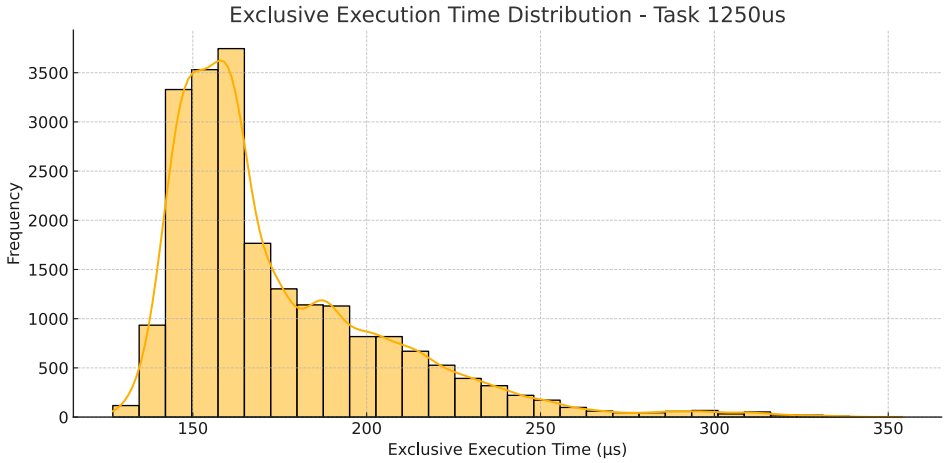


Figure C.1: Exclusive Execution Time for the 1250 μs task

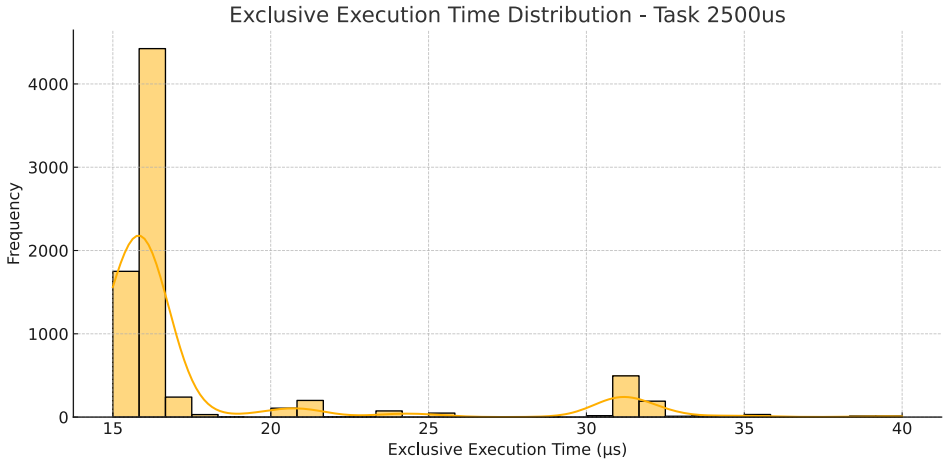


Figure C.2: Exclusive Execution Time for the 2500 μs task

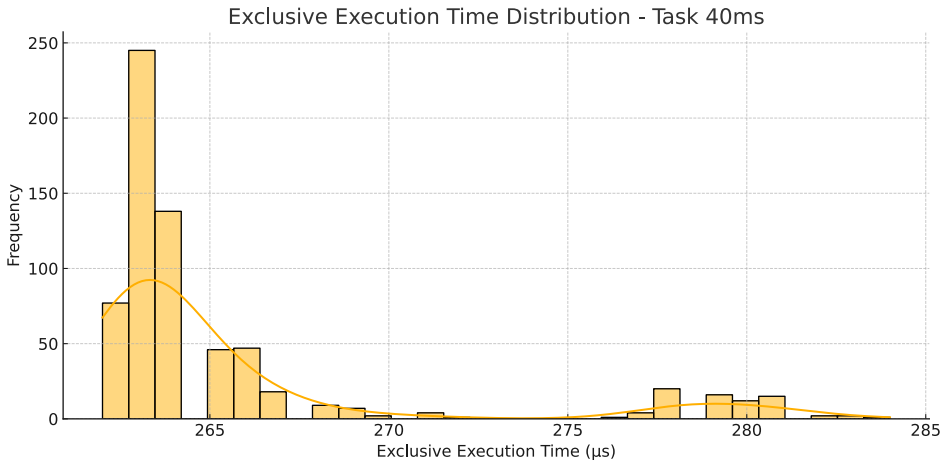


Figure C.3: Exclusive Execution Time for the 40 ms task

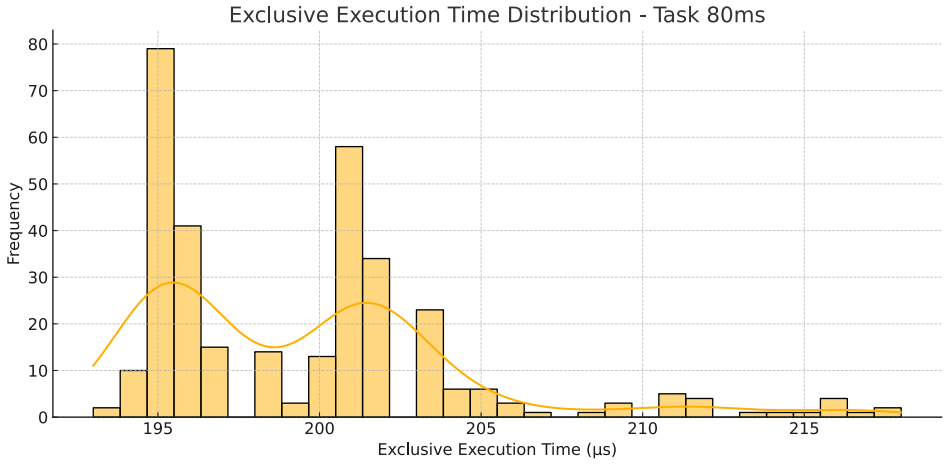


Figure C.4: Exclusive Execution Time for the 80 ms task

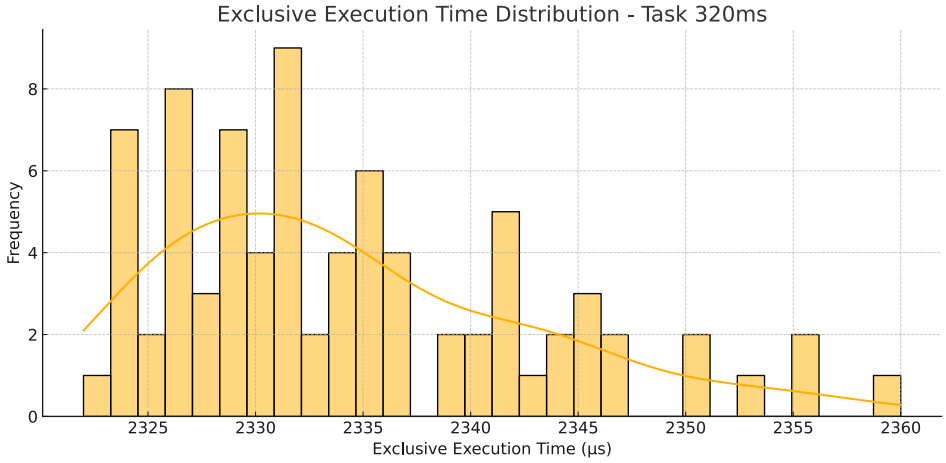


Figure C.5: Exclusive Execution Time for the 320 ms task

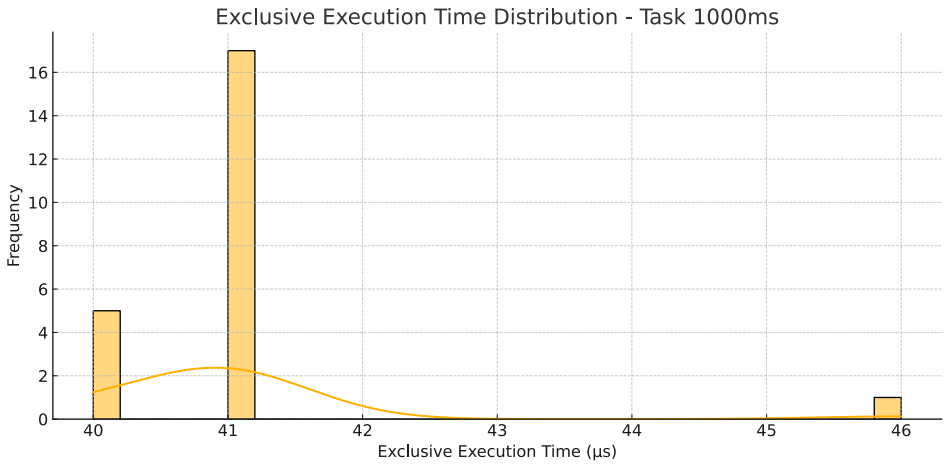


Figure C.6: Exclusive Execution Time for the 1000 ms task

D

Appendix 4

This appendix contains the full set of vECU response time distribution plots not shown in the main section. These plots include tasks with either highly consistent or less analytically distinct behavior, and are included here for completeness.

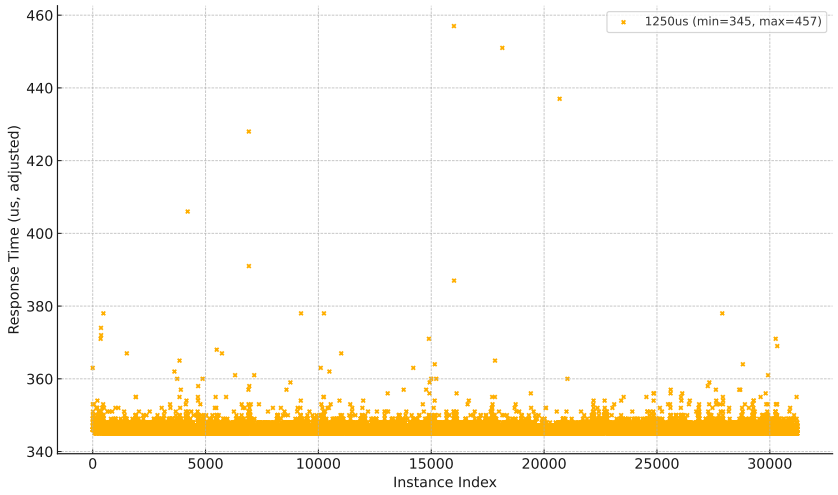


Figure D.1: vECU Response Time Distribution for the 1250 μs Task

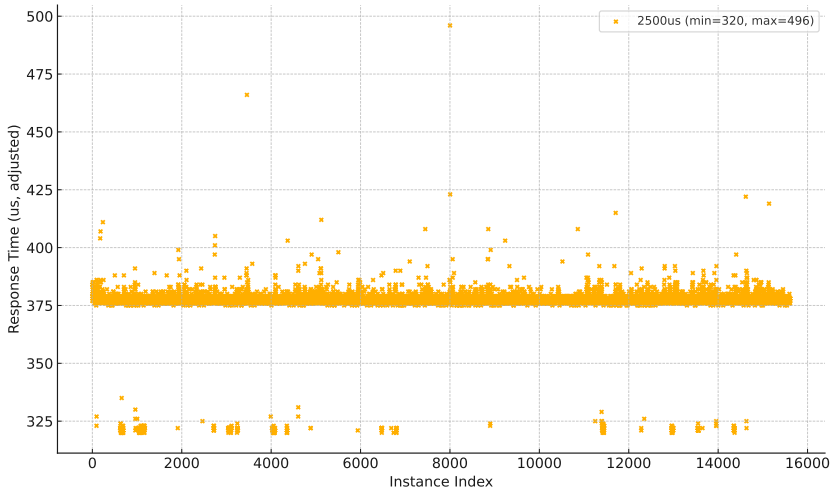


Figure D.2: vECU Response Time Distribution for the 2500 μs Task

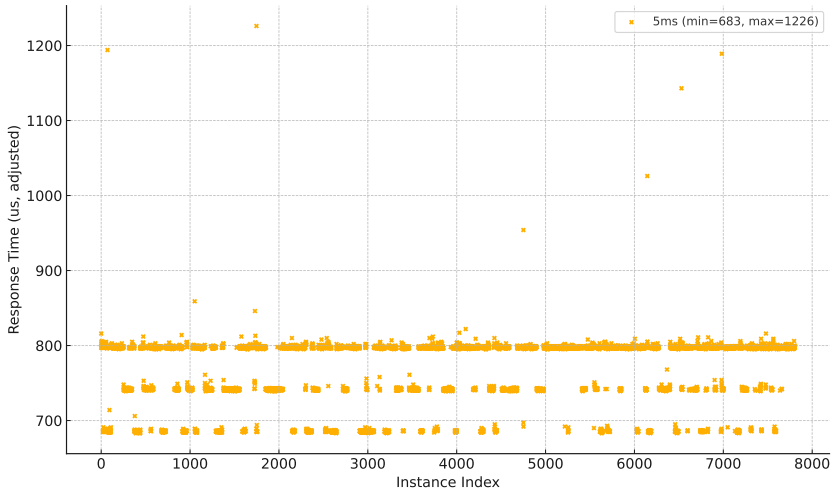


Figure D.3: vECU Response Time Distribution for the 5 ms Task

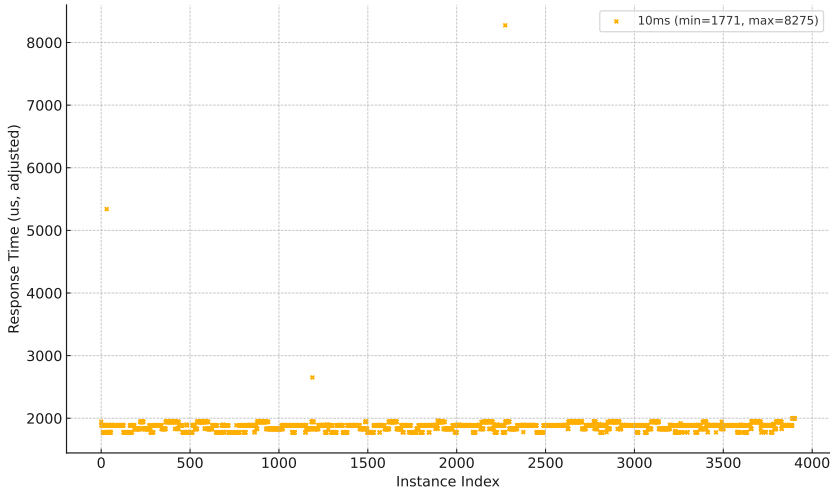


Figure D.4: vECU Response Time Distribution for the 10 ms Task

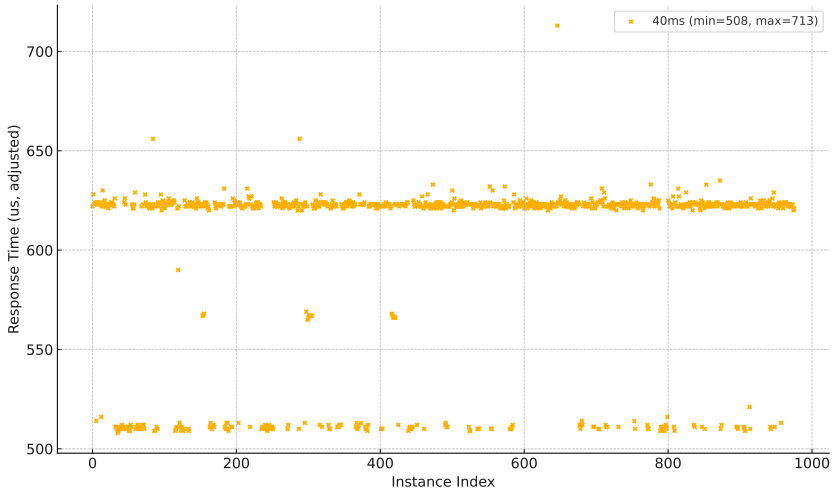


Figure D.5: vECU Response Time Distribution for the 40 ms Task

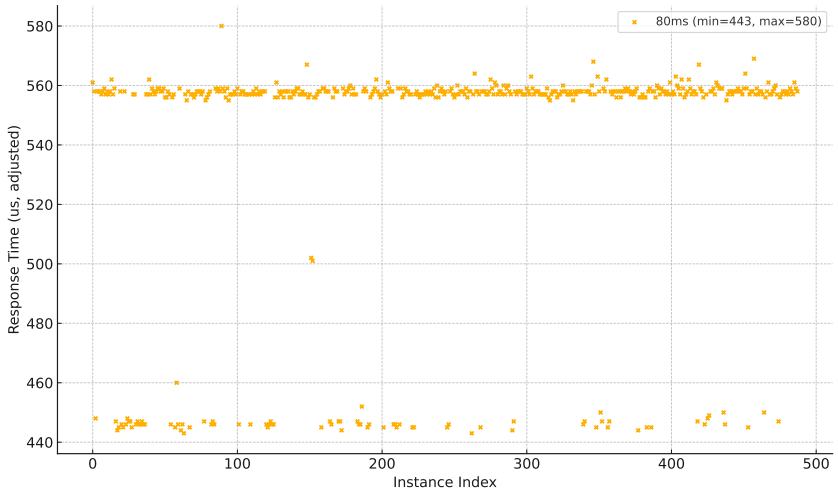


Figure D.6: vECU Response Time Distribution for the 80 ms Task

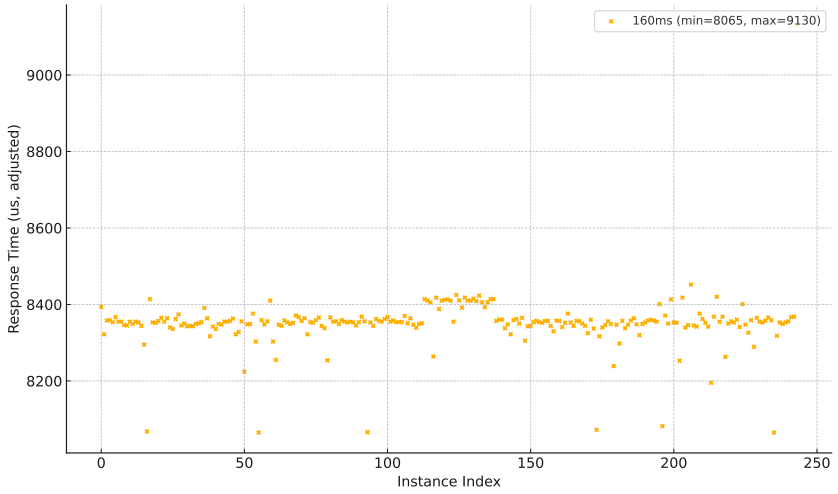


Figure D.7: vECU Response Time Distribution for the 160 ms Task

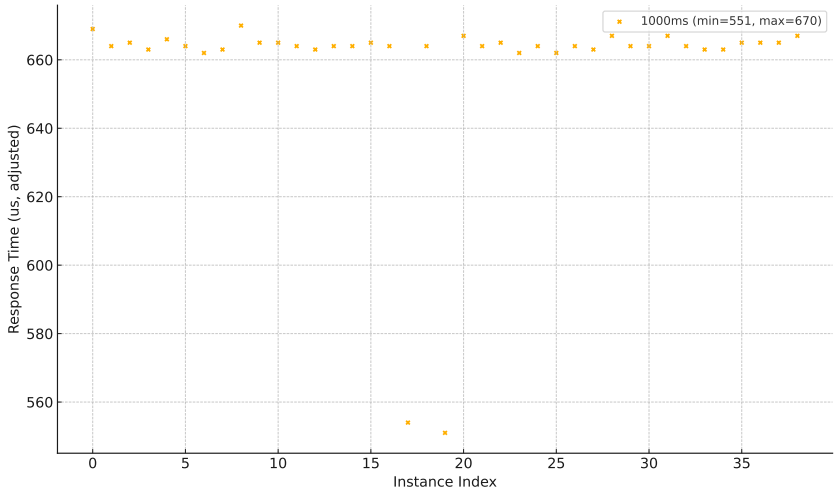


Figure D.8: vECU Response Time Distribution for the 1000 ms Task