ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Stress Injection Study on Hard Real-Time Operating Systems

Scientific work to obtain the degree of
Master in Electronic Engineering
at the Department of Electrical, Electronic, and Information Engineering
Alma Mater Studiorum Universita di Bologna.

Submitted By Daniel Mauricio Sepúlveda Flórez
Diploma Electrical Engineer

Supervised By Dr.-Ing. Albrecht Mayer, Infineon Technologies AG
Prof. Andrea Bartolini, University of Bologna

Co-Supervised By M.Sc. Max Brand, Infineon Technologies AG

Submitted In Bologna, Italy, March 2020

Acknowledgements

I would like to express my sincere gratitude to Infineon Technologies AG and all the people who made it possible to do this work. Through this work, I have acquired new skills and improved old ones. Across this work, I faced many challenges and I overcome them, marking a new phase on my personal and professional life.

Initially, I would like to express my deepest gratitude to my family. To my parents, Martha Lucía and Juan, my brother Juan Carlos and my sister Johanna, for their unconditional help and support. I am grateful to my sister for her unlimited advice and encouragement. I thank my parents for their unlimited support and for helping me to face new challenges and experiences. This journey would never have been possible without them.

I am grateful to my supervisor at the University of Bologna, Prof. Dr. Andrea Bartolini, for his support and enrichment feedback in this work. I appreciate his trust and the time invested in my advisory.

I would like to offer my special thanks to the company Infineon Technologies AG and the Technology Platform and Innovation Group at the Automotive Department. It was a fabulous experience plentiful of knowledge and new challenges. I would like to express the deepest appreciation to my supervisor at Infineon, Dr.-Ing Albrecht Mayer who provided straight support and established a solid foundation of my work. His deep experience contributed to the successful realization of this work.

Last but not least, I would like to express my gratitude to my co-supervisor at Infineon M.Sc. Max Brand for his constant support and the very valuable feedback in this work. Our technical discussions allowed me to further optimize this work.

Bologna, Italy, March 2020. Daniel M. Sepúlveda F.

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents a general overview of this thesis. It is composed of five sections. The first section describes the motivation behind the elaboration of the thesis. It describes the challenges of the design of automotive devices. The second section describes the problems and limitations of the current solutions. The goals of the thesis and contributions are presented in the third and fourth sections. Finally, the structure and organization of the thesis is presented in the fifth section.

1.1 Motivation

The automotive industry has been revolutionized by the widespread adoption of new digitalization technologies, such as the electric mobility, the high increase of automation and connectivity, and the integration of smart and autonomous mechanisms (e.g., autonomous driving). While these technologies have the potential to provide a variety of additional services and functions in and around the vehicle, the high criticality of the automotive systems demands a high degree of predictability and resilience. As the degree of vehicle automation increases, so too must the safety and reliability measures.

Real-time constraints demand that the execution of an automotive application task is performed mandatory bounded to hard deadlines. The execution of a task after the deadline may lead to injuries or fatal events. For example, an effective and good steering and braking system depend on the correctness of the computation of different variables and on the execution time. Therefore, high reliability and predictability are imperative in such systems.

High VLSI levels of integration and hardware resource sharing of the multicore System-on-Chips (SoCs), usually designed to implement automotive applications, make it challenging to assurance reliability and predictability. One of the techniques used to reach high levels of reliability is through the system evaluation based on CPU timing analyzing tools. These tools are able to verify and evaluate the deployment of different software tasks through a set of timing performance metrics and constraints. However, these tools present several drawbacks: i) they cannot exactly estimate the CPU performance; ii) they cannot model the timing parameters of the task executed in the system; ii) they cannot estimate the effect of the data and peripherals disturb on the CPU performance. In order to improve and

enhance the evaluation of such systems, improved performance tools are required. These tools should perform a better timing constraints analysis and define an assure boundary for the demanded CPU performance. However, these are challenging tasks. The delineation of the boundary is critical. On one hand, it may lead to a safety risk if it is excessively close to the performance limit of the system. On the other hand, it may lead to the misuse of the system capabilities. Determining the boundary range requires the detection and the analysis of the critical timing chains rather than the lapse of time in which the CPU remains in the idle task. Due to the challenging task of defining this safe boundary, it is imperative to find a symptom or indicator that determines where the system is pushed to its limit. As an analogy, we can think on the human body (SoC), which is a very complex system, composed by different organs (IP cores) which are interconnected by different structures, such as nerves, veins (links) and which react to different input stimuli gathered by the senses (peripherals). The body (SoC) supports different functionalities which should operate in a harmonized way in order to work in a proper manner (reliability) and to react to different situations to preserve life (safety). Arrhythmic breath, heart rate or just delayed reflex can threaten life. The correct functionality of the body should be maintained under different situations, such as repose (zero load operation), normal conditions (common operation) and extreme conditions (corner cases). This evaluation under different stress conditions may help in the diagnosis of possible sickness (fault or erratic behavior). Note that sickness usually has many symptoms. That is before a sickness invades the whole body, the symptoms can be treated to avoid a grave condition and death (failure of the system). In such an analogy, a set of tests is required to evaluate the health of the body (SoC) under different stress conditions (operation conditions). This evaluation allows to identify the symptoms of very bad deceases.

1.2 Problem Description

Critical applications require that the tasks, implemented as embedded software, are able to meet the performance and cost requirements no matter the operation conditions of the system. In order to guarantee such a challenging goal, the embedded software designer can benefit from a deep understanding of the system hardware so to effectively use the capabilities of the SoC. Therefore, it is mandatory to understand and to know the hardware operation boundaries and the possible implications of the pierced boundaries.

The automotive domain is a very challenging environment. The software complexity has increased exponentially in the last 30 years, growing from 0 to over 10 million lines of code [1]. Nowadays, automotive applications are built on top of a hard real-time operating system where thousands of tasks are executed, some of them simultaneously. Approximately 40% of the production costs of a vehicle is due to the software and electronic content infrastructure [1]. To attend the functional safety and security standards dictated in the ISO 26262 and ISO/SAE 21434, the automotive software development normally follows the V-model. This model divides the software development into two main wings: the design wing and the test

wing. The final product is only released if it satisfies all the requirements of the use case. In order to ensure that the electronic devices for automotive applications meet the functional and non-functional concerns, several testing tasks at different abstraction/refinement levels of the software are performed.

The standard ISO 26262 has provided a set of guidelines to develop improved safety systems for vehicles, from simple airbag deployment systems to more complex systems such as the Advanced Driver Assistance Systems (ADAS) capable to predict and avoid accidents. Among the methods for software integration testing described in the ISO 26262 standard, there is the so called *Resource Usage test*. This method states that the test reuse is necessary to ensure the fulfillment of requirements influenced by the hardware architecture with sufficient tolerance. This method includes the measurement of different metrics, such as the average and maximum processor performance, minimum or maximum execution times. However, some aspects of the resource usage test can only be properly evaluated when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests [2]. The ISO 26262 standard also includes tests to determine the robustness of the system. There are two main methods. First, the *Resource Usage test*, also used for software integration testing. Second, the ***Stress Test***, which is performed to drag the system far beyond the capabilities and analyze the behavior of operation. In contrast to a simple load test, the Stress Test uses loads that are far away over the system is expected to handle.

Due to the increase of the VLSI integration levels, the complex requirements, the time-to-market and cost pressures characteristic of the Automotive domain, the software integration and robustness tests should be performed effectively and efficiently. Early silicon debug and in-field diagnostics are key points for a safe and robust system. This has motivated to automate different test processes and to offer different hardware/software support. Different types of trace and debug tools have been developed to enable the software engineer to access the internal functionalities of the SoC. Thus, enabling the information gathering of the system when different types of tests are performed (offline and during runtime). Using more advanced tools for tests and analysis of hard real-time systems is attractive, once they lead to maximize the exploitation of the SoC resources while providing the best safety for the user.

Semiconductor vendors have developed a different type of hardware/software support for these types of tests. In the case of Infineon Technologies AG, the AURIX 2G microcontroller has integrated a novel hardware architecture designed to support the *Resource Usage Test* and the *Stress Test*. Despite the hardware support *Stress Test* based on the stress injection by CPU suspension embedded in the AURIX Infineon microcontroller, it has never been used before. There this is the need to provide an automated method able to perform the stress injection to the *Device Under Test* (DUT), allowing to evaluate the performance and reliability of the system. This tool is capable to offer the observability and controllability of the different conditions of the test for the analysis of hard real-time systems. The tool support allows the software designer to gather a clear and wide understanding of the system behavior, aiding in

the design of safety-critical systems, such as they are designed in the automotive industry.

1.3 Goals of the thesis

The main goal of the thesis is to develop a method that uses stress injection to analyze the performance, robustness values and boundaries of hard real-time systems under different scenarios. The designer is able: i) to configure the embedded debugging hardware architecture in order to efficiently explore different stress scenarios; ii) to gather information; and to quantify different types of performance and robustness metrics. The method is automated and fully parameterizable. The developed tool in this thesis is called ***Galenus***[1]. The tool is integrated into the already existing debugging environment of Infineon AURIX named ChipCoach. To achieve that goal, the following tasks were performed:

1. Identify the load parameters that are of interest for a designer in order to explore different stress injection scenarios.

2. Identify a set of metrics that are of interest for a designer in order to evaluate the performance and robustness of the system.

3. Define a method to perform stress injection and which is able to exploit the current deployed debugging hardware support of the AURIX Infineon Microcontroller.

4. Automate the method for stress injection in order to efficiently evaluate different stress scenarios.

5. Define different case studies that demonstrate the utility of the method. It includes defining a set of tasks that can be deployed in a hard Real-Time Operating System (RTOS).

6. Inject different levels of stress on the previously defined task sets.

7. Collect the timing parameters of the different stress injection levels of the system with the tracing tool.

8. Analyze and evaluate the collected data.

[1] Following the SoC health test analogy, the developed stress injection feature tool in this work is called ***Galenus*** in honor of Claudius Galenus. He was a physician and surgeon in the Roman Empire considered one of the most important figures in the history of Medicine.

1.4 Contribution of the thesis

The following contributions were achieved throughout the work in this thesis:

1. Development of an efficient method that is able to perform stress injection through the exploitation of debugging hardware architecture. This method allows the designer to explore different stress injection (by CPU suspension) scenarios and to evaluate the performance and robustness of critical systems.

2. Development of *Galenus*, the tool that automates the proposed method and that exploits the AURIX Infineon Microcontroller embedded debugging architecture. Galenus is integrated into the ChipCoach debugging tool of Infineon Technologies AG. Galenus is fully parameterizable and allows the efficient evaluation of the SoC.

3. The design of the tool allows that future capabilities are easily and effortless integrated, such as the integration of statistical distribution used to define and model the input parameters of the stress injection capabilities. Further and wider exploration can be easily performed.

4. Definition method to measure the performance and the robustness of the *DUT*. This is done through the technique of *stress injection* by CPU suspension implemented within two types of software application, RTOS and Bare-metal.

5. Utilization for the first time of the embedded debugging hardware architecture of the AURIX Infineon microcontroller for stress injection. The stress injection is based on the reduction of the effective performance of a SoC component (IP hardware core e.g., TriCore within AURIX). The stress injection allows to assess the sensitivity of the SoC under different stress scenarios. These scenarios are defined on the offline initial state using formal methods of scheduling theory. Using the stress injection method, the SoC designer is able to identify possible risk scenarios testing the performance and robustness of the system at runtime. This thesis is based on the stress injection by CPU suspension within two types of software application, RTOS and Bare-metal.

6. Detailed documentation of the method, which allows Infineon hardware and software designers inside Infineon Technologies AG to easily get familiar with the stress injection method and to easily profit from all the capabilities of Galenus and Chipcoach.

1.5 Structure of the thesis

The rest of the thesis is organized as follows.

- **Chapter 2: Main Concepts** This chapter introduces the main concepts that were used throughout this thesis. It is divided into four parts. The first section introduces the concepts related to the RTOS. The second section presents the concepts for task scheduling. The third section introduces the schedulability analysis. The fourth section presents some of the standards of the automotive domain. The fifth section introduces the hard RTOS ERIKA OS [3]. Finally, the sixth section introduces the TriCore AURIX second generation microcontroller [2] developed by Infineon Technologies AG.

- **Chapter 3: State of the Art** This chapter introduces the current works and available tools in the area of stress injection from the academic and industrial contexts. The first section presents the automotive evaluation of robustness and performance. The second section introduces the main robustness and performance testing tools. The third section presents the analysis techniques. Finally, the fourth section presents the stress injection evaluation.

- **Chapter 4: On-Chip Trace and Debugging Architecture** This chapter introduces the on-chip trace and debug solutions developed by Infineon Technologies AG for the AURIX microcontroller. The chapter is divided into six sections. The first section presents the general concepts and architecture of the on-chip trace. The second section introduces the architecture of the on-chip trace solution developed by Infineon Technologies AG. The third section describes the trace targets supported by the on-chip solution. The fourth section describes the Multicore Debug Solution (MCDS) a main component of the on-chip trace solution. The fifth section introduces the Device Access Server (DAS), the interface between the SoC and the trace tool. The sixth section presents the main trace tool of Infineon Technologies AG and the ChipCoach tool, further extended in this thesis with ***Galenus***, to support stress injection.

- **Chapter 5: Stress Injection** This chapter presents the first contribution of the thesis: the design of the stress injection feature for the SoC evaluation. This feature was included in the Infineon Technologies AG design flow for tracing and debugging, more precisely, at the ChipCoach tool. Despite the hardware architecture to support the stress injection that is already embedded in the AURIX 2G Infineon microcontroller, it has never been used before. The work performed in this thesis allows the utilization of such an infrastructure for the first time. This chapter is divided in four sections. The first section presents a general description of the Stress Injection. The second section presents the *Trigger Line Timer* (TLT) of the *OCDS Trigger Switch* (OTGS) that is used to perform periodic suspensions of one or more CPUs. The third section describes the

[2] Referred in this work as AURIX 2G

Stress Injection Trigger Line Timer that is part of OCDS in the central debug interface. The fourth section describes the developed stress injection feature called *Galenus*.

- **Chapter 6: Methodology** In this chapter, the methodology proposed in this thesis is presented. The goal of the methodology is to measure the performance and the robustness of the *DUT* through the technique of *stress injection* by CPU suspension implemented within two types of software application, RTOS and Bare-metal. This chapter is divided into four sections. The first section presents the general description of the methodology. The second section describes the overall methodology for the RTOS test composed by six blocks. The third section describes the overall methodology for the Bare-metal test composed by six blocks. Finally, the fourth section presents a summary of the methodology.

- **Chapter 7: Case Studies and Experiment Results** In this chapter, the experimental work is presented. It describes the experiments performed for the two types of software application: RTOS and Bare-metal. For the RTOS and the Bare-metal experiments, six case studies are designed, three for RTOS and four for Bare-metal. For each case study the performance and robustness evaluation results are presented, following the method proposed in Chapter 6. This chapter is divided into four sections. The First Section describes the general configuration and characteristics of the experimental work. The second section describes the RTOS experiments and results. The third section presents the Bare-metal experiments and results. Finally, a summary of the experimental results is shown.

- **Chapter 8: Conclusions and Future Work** This chapter presents the main conclusions of this thesis and future work.

2 Main Concepts

This chapter introduces the main concepts that were used throughout this thesis. It is divided into four parts. The first section introduces the concepts related to the *Real-Time Operating Systems* RTOS. The second section presents the concepts for task scheduling. The third section introduces the schedulability analysis. The fourth section presents some of the standards of the automotive domain. The fifth section introduces the hard RTOS ERIKA OS [3]. Finally, the sixth section introduces the TriCore AURIX 2G microcontroller developed by Infineon Technologies AG.

2.1 RTOS

Today low-end and high-end cars consist usually of 20 to 110 *Electronic Control Units* (ECUs) [4]. These ECUs are connected with a set of sensors and actuators, running diverse distributed control applications. There are two main safe examples of the ECUs in vehicles. First, the controller of an airbag must be activated in less than two milliseconds when a crash is detected [5]. Second, the brake controller and the steering in the Advanced Driver Assistance Systems (ADAS). In those and other systems, the tasks or processes must be performed in a specific maximum time of execution called the deadline. Due to the time constraints of the system, it is mandatory to prove the logical and temporal correctness of the system to avoid catastrophic consequences to the user.

The automotive application functions performed by the ECUs are built on top of an RTOS where hundreds of tasks are executed. RTOS contains a real-time kernel and other higher-level services such as protocol stacks,file management and other ocomponents [1]. These services are executed within real-time requirements based on the deadlines. There are three types of RTOS:

1. Hard RTOS: If the results are provided after its deadline, it can result in catastrophic consequences on the system under control and fatal injuries on humans [6].

2. Firm RTOS: If the results are provided after its deadline it is useless for the system, but does not cause any damage [6].

3. Soft RTOS: If the results are provided after its deadline it still has some utility for the system, although it causes performance degradation [6].

The hard RTOS are required on automotive applications due to the time constraints. The software within an RTOS is divided into executable units, called tasks.

2.1.1 Task

A task is a block of instructions to be executed by a processor for a specific purpose [7]. Tasks can be triggered periodic or aperiodic. The first appears for example if a sensor must be checked on a regular basis, while the latter is for example applied on tasks triggered by external interrupts caused due to user interaction. The requirements and constraints that regulate the execution of the tasks are constituted by eight relevant attributes [8]:

Period: Describes how often a task is activated. There are three types related to the periodicity of activation. Periodic, the task is activated in an equidistant way with a constant frequency. Aperiodic, the task is activated in a non-periodic way with varying frequency. Sporadic, the task is activated in a non-periodic way in the system with less frequency than the aperiodic task usually corresponds to a user request or an interrupt of the system.

Deadline: Defines a time constraint of a task in which the task has to finish the execution. In general, every task must be finished before the deadline. If it is finished after the deadline, the system became *Infeasible* and can incur severe consequences for the system and the user.

Priority: Is a value that represents the significance of the task in the system. A higher priority task that is ready for execution will be executed prior to a low priority task, which means that the lower priority task will be preempted.

Execution Time: Describes the time it takes for a task to be executed completely. The longest duration of a task is often referred as the Worst-Case Execution Time (WCET) and the shortest as the Best-Case Execution time (BCET)

Arrival Time: Describes the instant when a task is activated and ready for the allocation of the resource of the CPU. For periodic tasks, the arrival time is known before the execution of the system. For aperiodic tasks, the arrival time is only known during the runtime when a specific triggering event occurs.

Offset: Specifies the time between the system starts and the first occurrence of a periodic task. Usually, the offset for the task with higher priority starts as soon as the system is executed.

In an RTOS, a task as a block of instructions with allocated functions, which are executed by the processor for specific purposes. These allocated functions inside the tasks are defined in this thesis defines as *Runnables*. Therefore, a single task is compound by one or more Runnables. When a task is executed, the Runnables are executed as well. In the automotive domain, the Runnables commonly are in charge of reading a sensor, processing data and trigger an actuator [9]. For example, the brake by wire with the anti-lock system in a car, which consists in the control of the breaks. For each wheel, it consists of four Runnables with functions of: i) senses the rotation of the wheel; ii) processes the data; iii) activates the brake according to the processed data; and iv) triggers the corresponding test to the brakes [10]. A set of tasks are called concurrent when they are ready to be executed at the same time or partially. If this concurrent set of tasks must be executed in one processor, the execution order of those tasks has to be defined following a dedicated strategy called scheduling.

2.1.2 Lifecycle of a task

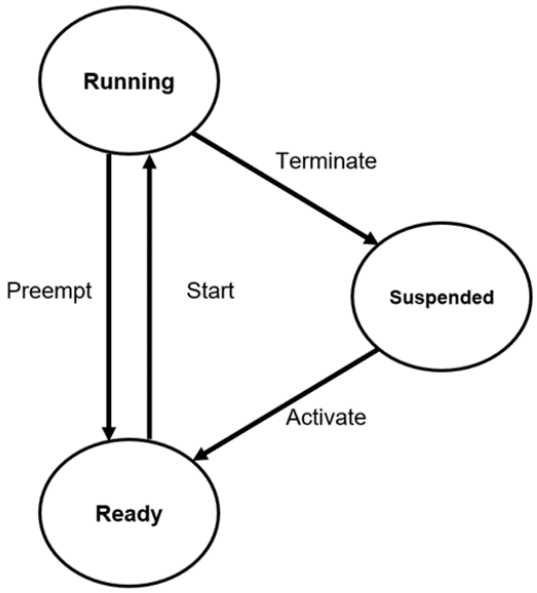

Figure 2.1: Basic task state transitions

During the runtime in the RTOS, each task must be always in one of the following states showed in Figure 2.1. For basic tasks, after the activation, the task is in the ready state. Therefore, the task is ready to be executed and must wait until CPU resources are assigned. The allocation of a task to the processor is based on a defined scheduling that is performed by the scheduler of the RTOS. When the task is set to be executed, it changes the state to Running state. In case the task is preempted, the task goes from running to the ready state. After the execution is finished, the task's state changes to the suspended state.

In case an extended conformance class according to [] is used, tasks can have an additional state. When the task is waiting for an event or a resource, it includes all the previous states plus the waiting state. When the task is in running state and then is waiting for the occurrence of an event, it moves from the running state to the waiting state. After the event occurred, the task changes to ready state showed in Figure2.2.

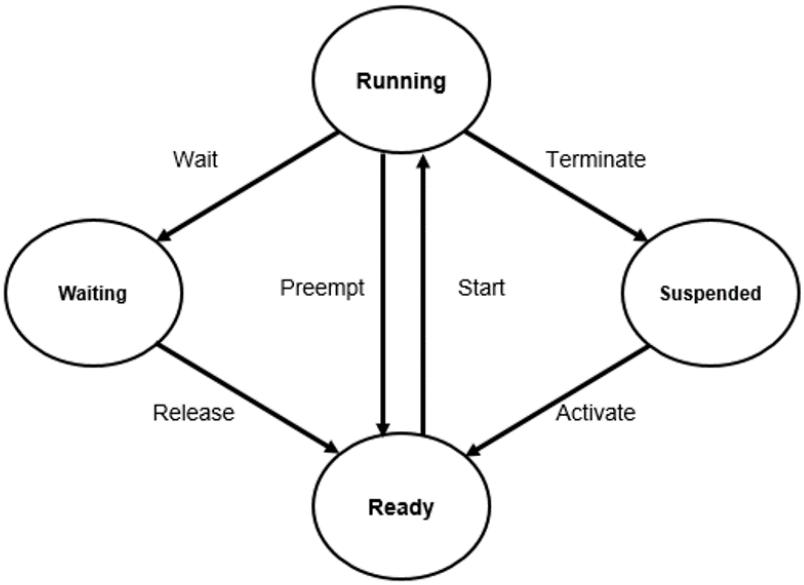

Figure 2.2: Extended task state transitions

1. Suspended: This is the default state of the tasks, where the task is suspended and ready to be activated.

2. Ready: In this state, the task is activated and waiting for the allocation of the CPU. The scheduler of the RTOS manages all the tasks residing in ready state and sorts them by their priority.

3. Running: The task is using resources of the CPU and it is executed. For single-core implementations, only one task per time can be in this state. While for the other states, more than one task can be in those states.

4. Waiting: The task changes from running state to the waiting state due to a requirement of a resource. To continue the operation the task must wait until the resources are acquired. This state is exclusive for the extended task.

2.1.3 Timing Parameters

The timing parameters of a task refers to the period it takes for a task from one initial state until a final state of the real-time behavior of the task. For a basic task, there are five important timing parameters: i) *Response Time*, the period between the task arrival until it is suspended. ii) Execution Time, is the accumulation of time between the *Running* state and the *Suspended* state. iii) Initial Pending Time, the period between the *Ready* and *Running* state that occurs the first time on an instance of a task. iv) Preemption Time, the period between the Ready and Running state that occurs more than one time. Otherwise correspond to the *Initial Pending Time* parameter. v) Slack Time, the interval of time between the Suspended state and the Ready state. In the following Figure 2.3 shows an example of execution of a task set of three basic tasks with periods of 1ms, 5ms, and 10ms using the Rate Monotonic Scheduling (RMS) algorithm, where the 5msTask and the 10msTask are preempted and the total execution time is the sum of the Core Executions.

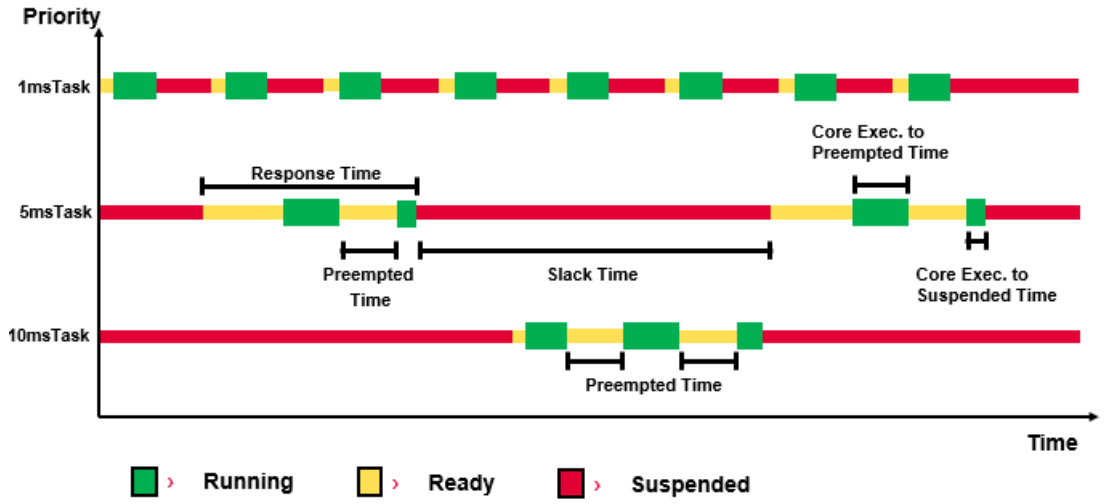

Figure 2.3: Example of execution of three tasks and the respective states with the timing parameters.

2.2 Scheduling

Scheduling is the process of resource allocation of the CPU on a task within a task set. The scheduling produces a schedule, in other words, the order in which the tasks are executed. A schedule is defined as *Feasible* when the complete task set meets its deadlines. The task set is defined as *Schedulable* when there exists a feasible schedule for an arrival pattern. An RTOS can implement different scheduling algorithms with diverse types of scheduling policies for concurrent tasks. The main purposes of the scheduling algorithms are to

guarantee the correctness in the execution between the tasks and minimize the resource starvation.

There are several scheduling algorithms for RTOS and the criteria selection is dependent on the type and the purpose of the system. According to the literature [6], the common way to classify the scheduling algorithms is based on the moment in which the procedure to schedule is taken. Two main types can be found for this criterion. First, off-line algorithms are implemented with static policies, i.e. all the task parameters and instances are defined previous to the execution of the system. Second, on-line algorithms are implemented with dynamic policies, in other words, the scheduling is performed at runtime. Most of the on-line algorithms are preemptive scheduling algorithms, therefore a currently running task can be preempted by a task with higher priority. The preempted task can only continue the execution when the higher prioritized task has completed the execution. The most common classification for the on-line algorithms is based on the task priority parameter criteria. There are two types of on-line algorithms:

Dynamic Priority Scheduling (DPS): The priority of a task can be modified at runtime. One example of DPS is the Earliest Deadline First (EDF) scheduling algorithm, which assigns the highest priority to the task with the closest deadline.

Fixed Priority Scheduling (FPS): The priority is fixed before the start of the system. There are two main examples of FPS. First, the Deadline Monotonic Scheduling (DMS) algorithm, where the priority is assigned based on the deadlines in inverse proportion. Second, the Rate Monotonic Scheduling (RMS) algorithm, where the priority is assigned according to the period, where a smaller period results in a higher priority.

The most common FPS is the RMS, widely used in RTOS safety-critical applications such as in the automotive domain. One example of the use of the (RMS) algorithm in automotive is the admission control for the combustion engine in a car [11]. Due to the importance of this scheduling algorithm on automotive, more detail and description will be done in this thesis. As shown before, RMS is an on-line preemptive scheduling algorithm with fixed priorities for periodic tasks. The priorities are assigned to the tasks based on their periodicity values defined by the software designer engineer [12]. The priority of a task is directly proportional to its frequency. Therefore, a task with lower periodicity value, meaning a greater number of occurrences within a fixed time interval, is assigned a higher priority. The fundamental theory for the RMS is the rate monotonic analysis. This analysis is stated in the following model [12].

1. All processes run on a single CPU, consequently, there is no task parallelism.

2. The context switching time is ignored.

3. The task load remains constant.

4. Tasks are independent of each other.

5. Implicit Deadline, the deadline of an instance of a task occurs at the end of its period.

According to the previous model, the following theorem has been stated in [12]: " *Given a set of periodic tasks to be scheduled using a preemptive priority scheduling, assigning priorities, such that the tasks with shorter periods have higher priorities, yields an optimal scheduling algorithm.*"

2.3 Schedulability Analysis

The schedulability analysis aims to determine whether a task set is schedulable. This is done using certain techniques that use mathematical methods called schedulability tests. For most of the scheduling algorithms, including the (RMS) exist certain schedulability tests [6]. There are two common schedulability tests for the (RMS). First Utilization-Based Analysis (UBA) [13], that provides a sufficient but not necessary condition. Second, Response Time Analysis (RTA) [14].

- **Utilization-Based Analysis (UBA):** In this schedulability test, as used in this work, the task with a shorter period is given the higher priority and the task deadlines are equal to the period, therefore the task set is defined with *Implicit Deadline*. This test defines the following sufficient condition in (2.1) for schedulability for a task set that consists of n periodic tasks $\tau_1, \tau_2, ..., \tau_n$:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(2^{1/n} - 1\right) \tag{2.1}$$

  Where C_i corresponds to the Worst-Case Execution Time (WCET) of the task τ_i with the respective period T_i. This sufficient condition states that if the total utilization *U* of the task set is lower than the utilization bound $n(2^{1/n} - 1)$, the task set is schedulable under the (RMS). However, it is a sufficient condition, a system might be schedulable even if the CPU utilization is higher than the utilization bound. In such cases, it is necessary to use an exact schedulability test such as the Response Time Analysis [13].

- **Response Time Analysis (RTA):** This schedulability test is based in the Worst-Case Response Time (WCRT). This approach uses the following recurrence equation (2.2) to compute the WCRT R_i of a task τ_i:

$$R_i^{(k)} = L_i + \sum_{j=i}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil L_j \tag{2.2}$$

Where the term of the summation is the total interference from the higher priority tasks during R_i and L_i is τ_i own execution time assuming that τ_j has a higher priority than τ_i if $j < i$. The equation (2.2) is iterated until R_i converges at a value. This value is compared against the τ_i deadline in order to determine the schedulability of τ_i. This schedulability test for the (RMS) states a theorem in which takes into account the critical i.e there is no offset between the tasks, therefore the tasks are released simultaneously. The theorem states, if all the tasks meet their first deadline after a critical instant, they will also meet all the subsequent once since the critical instant is considered as the worst case [14].

2.4 Standards

2.4.1 ISO 26262 - Automotive Safety Standard

Security standards are concerned about protecting a system from attackers, but safety standards aim to prevent any possible harm to the user arising from hazards. There are many safety standards that depend on the industry sector [15]. The automotive industry is constantly improving the safety of the user on the vehicles, implementing a different type of safety system such as the Advanced Driver Assistance Systems (ADAS) that allows to predict and avoid accidents. These types of functions are performed by electronics. Nowadays a car usually consists of multiple different ECUs, these are connected with a set of sensors and actuators. In response to the increase of electronic systems in vehicles and the recognition of the safety-critical functions that they performed, the ISO 26262 standard has been created [16].

The safety standard ISO 26262 provides a detailed guideline to produce all the safety-critical or not equipment and software for automotive systems. This standard determines a risk-management approach and four risk classes (A-D), called Automotive Safety Integrity Levels (ASILs). This levels of risk, specify the required safety measures in order to avoid an unreasonable residual risk, where D is the most rigorous level [16]. Figure 2.4 shows an example of some of the electronic systems in a vehicle and their respective ASIL according to the ISO 26262 standard.

The software development guideline is also contemplated in the ISO 26262. The development of software normally follows the V-model showed in Figure 2.5 that is divided into two main branches, the design, and the test branch. The final product must satisfy all the initial

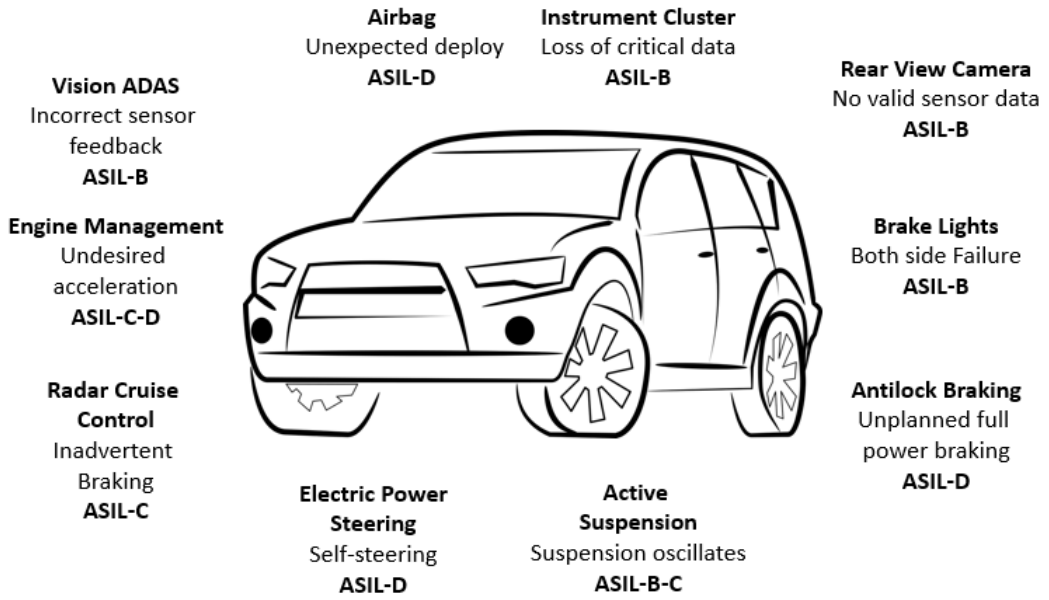

Figure 2.4: Example of electronic systems in a car and the respective ASIL.

requirements of ASIL determined by the standard ISO 26262. These requirements are warranty by performing the testing at several levels that address functional and non-functional requirements. Functional requirements are formalized by a function hierarchy such as Unit or Integration Testing. Non-functional requirements are quality process requirements for the final product such as performance. load, stress, and reliability [1]. Some of the software tests are: i) Performance Test that validates the requirements of the system; ii) System Test in charge of validates the specified software architecture; iii) Integration Test that validates the software requirements specifications; and iv) Unit Test, responsible to validate the component design.

The software testing methods are standardized depending on the industry as shown on the Table 2.1. The safety standard ISO 26262 contemplate the methods for software integration testing such as the *Resource Usage Test*, for this test, the ISO 26262 estates that the Resource Usage Test it is necessary in order to ensure the fulfillment of requirements influenced by the hardware architectural design with some degree of tolerance. Properties such as average and maximum processor performance, minimum or maximum execution times must be determined. Some aspects of the resource usage test can only be evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests [2]. ISO 26262 contemplates the level of robustness as well by the *Resource Usage* but also by the method of the stress test. The standard states that the stress test verifies the test system for correct operation under high operational loads or high demands from the environment [2].

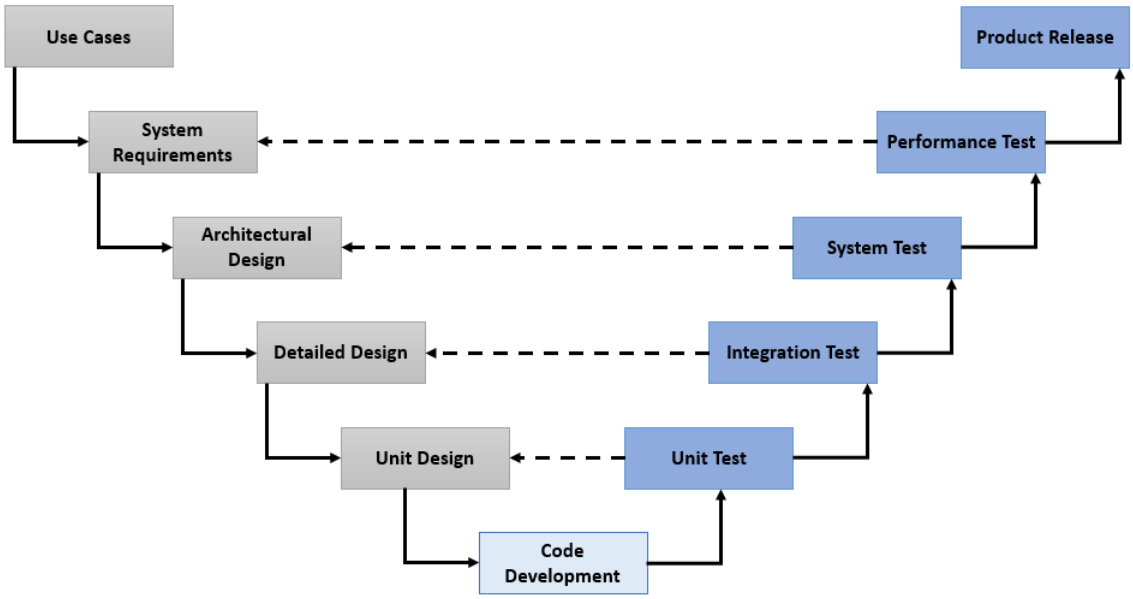

Figure 2.5: V-model for software developments.

2.4.2 OSEK/VDX Standard

OSEK/VDX is a standard for most of the Operating Systems used in the Automotive Industry, based on the ISO 17356 standard. The standard of the software architecture for the ECU in the automotive field was a consequence in one hand by the complexity of the vehicles due to the increase of the networked subsystems and on the other hand the ambition of cost reduction. The basic idea of the standards is to have an architecture characterized by components with easy interchangeability and re-usability. The principal standard for the RTOS it is called OSEK-VDX by the German “Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen” in English “Open Systems and their Interfaces for the Electronics in Motor Vehicles”. This standard provides provide a standard software architecture for real-time systems in vehicles with different interfaces, where the most relevant are [17].:

1. OSEK-OS: Service of the real-time Kernel that declares an operating system based on priorities.

2. OSEK-OIL: OSEK implementation language, kernel configuration.

3. OSEK-RTI: Declares the internal operation of an OSEK OS and a debugger.

4. OSEK-COM: Specify the communication environment.

	Automotive	Medical	Avionics	Railway
Testing Methods	**ISO 26262**	**IEC 62304**	**DO-178B/C**	**EN 50128**
Static Analysis	✓	✓	✓	✓
Requirements Based Tests	✓	✓	✓	✓
Data / Control Flow Interfaces	✓		✓	✓
State Transition			✓	
Resource Usage Test	✓	✓	✓	✓
Timing Tests	✓	✓	✓	✓
Boundary Value Analysis	✓		✓	✓
Error Guessing	✓			✓
Fault Injection / Error Seeding	✓		✓	✓
Structural Coverage Testing	✓	✓	✓	✓

Table 2.1: Software Testing Methods by Safety Standard Industry.

2.4.3 OSEK-OS

The OSEK-OS was designed thinking on embedded systems with a specific purpose. OSEK-OS has been used as a basis in ECUs in the automotive domain. This standard provides several features to build and describe task-based real-time systems. This standard defines the implementation language called OIL that represents the standard system configuration this includes the specifications of interrupts, tasks, resources and many others. OSEK-OS is a static operative system where the tasks, priority of task, memory used, events and other characteristics are assigned at design time before compiling the code. Priorities are defined with an integer number from 0 to 255, where a higher number implies a higher priority. In the case where two tasks have the same priority, they will be executed according to the order of activation. To avoid priority inversion and deadlocks, OSEK-OS uses the priority ceiling protocol [17]. OSEK-OS differentiates two types of tasks with different behaviors: i) Basic Task, is configured with no events therefore there is no waiting state on it. ii) Extended Tasks are configured with events and these tasks can wait for the activation of one or more events. Some of the open-source RTOS with OSEK/VDX certificate are FreeOSEK [18], AUTOSAR [19], ERIKA Enterprise [20], and Trampoline RTOS [21].

2.5 ERIKA Enterprise RTOS

ERIKA Enterprise is an open-source Hard RTOS, certified and designed for the automotive domain with the OSEK/VDX certification aimed for multi-core devices. This RTOS provides an environment with the support of advanced real-time scheduling algorithms. The main features offered of ERIKA enterprise RTOS are: i) Hard real-time support with fixed priority scheduling; ii) Immediate priority ceiling; iii) Support for periodic alarm activation. iv) Support for shared resources; and v) Support for four Conformance Classes called: BCC1, BCC2,

ECC1, ECC2. The classes starting with the letter B only support *Basic Tasks*, instead those starting with the letter E support both *Basic Task* and *Extended Task*. The classes ECC1 and ECC2 have been designed to be the most appropriate support classes for synchronization primitives which involve the use of separate stacks for the exchange rate mechanism stack [22]. The RTOS selected to use in this thesis for the AURIX Second Generation TC39B is ERIKA Enterprise RTOS. The main reasons for choosing this RTOS are:

1. Well known in the automotive industry.

2. Open Source.

3. OSEK-VDX certified.

4. TriCore AURIX microcontroller support.

2.6 TriCore AURIX 2G Microcontroller

The 32-bit TriCore AURIX 2G is a hexa-core microcontroller developed by Infineon Technologies AG that combines safety architecture with high performance. These characteristics are ideal for safety-critical automotive applications. The AURIX is used by many automotive brands in several applications. Ranging from the airbag, power steering and braking systems to more complex systems such as the fail-operational systems that are supported by sensor systems using radar or camera technologies. The AURIX 2G is manufactured in a 40 nm embedded flash technology with six cores running at 300 MHz and with four additional checker cores delivering 4000 DMIPS and 16 MB of internal flash memory. The AURIX is designed to support the standard ISO 26262 up to ASIL-D [23].

Nowadays, vehicles are designed to meet more complex demands for the multiple systems controlled by the ECUs. Therefore, Infineon Technologies AG has developed the Multicore Debug Solution MCDS in order to design and optimize the performance of these systems. The key features of MCDS are the tracing of CPUs, buses, events and peripherals with powerful trigger capabilities. Together with this multicore trigger and trace system, Infineon has developed several tools for its AURIX microcontrollers. This work describes in detail the main trace tool known as the MCDS Trace Viewer (MTV) tool and the internal developed ChipCoach tool. The ChipCoach tool was extended in this thesis, in order to develop a new feature capable of performing stress injection in the AURIX microcontrollers through ***Galenus***. The MCDS and the other tools are going to be described in detail in Chapters 4 and 5.

3 State of the Art

This chapter introduces the current works and available tools in the area of stress injection from the academic and industrial contexts. The first section presents the automotive evaluation of robustness and performance. The second section introduces the main robustness and performance testing tools. The third section presents the analysis techniques. Finally, the fourth section presents the stress injection evaluation.

3.1 Automotive evaluation: Robustness and performance

The main purpose of the embedded software is to fulfill the requirements of the application and the respective quality needs according to certain standards. These are achieved by the effective utilization of the capabilities of the available hardware combined with quality software routines. Therefore, the software engineer must be aware of the boundaries of functionality and risks of the hardware together with the development of functions and programs that ensure the dependability of the system. In the automotive domain, as in other industries, a method to verify the dependability of a system is based on the testing of the robustness and defining the usage of the resources.

The *robustness* of a system is defined by the IEEE 610.12 standard as the degree to which a system can function correctly in the presence of invalid inputs or stressful conditions [24]. This characteristic is challenging for embedded software systems in many industries as the execution environment can not be fully recreated at the time of development. Therefore the robustness tests are required to verify not only general considerations but also more specific ones. General considerations include the absence of deadlocks or no run-time errors. Specific considerations include the capability of the system to remain in a nominal state after being degraded and the resources remain available for the tasks with high-priority on the system. The robustness failures are caused by many reasons, some of these failures are due to the performance CPU and memory-related issues [25].

3.2 Robustness and Performance Testing Tools

The robustness and performance testing tools in many industries are in general build in-house or open-source customized in-house. Where the main functionality is to assist in monitoring the visualization of the chain of activities, status reports and generation, and control of large streams of data [25]. These functions allow the tools to provide flexibility in the construction of worst scenarios and the possibility to replay them (reproduce the scenarios). In the automotive domain, one of the main robustness failures is in the timing constraints. Due to the safety-critical application in automotive, the development of more advanced tools is required. The tools are tied to the hardware architecture functionalities of the SoC that it can provide to support these tests such as tracing and debugging solutions.

In the automotive domain, there are some performance tools based on the timing constraints analysis of the system [26], [27]. The Timing Suite T1 was developed by GLIWA GmbH [26]. This tool is able to perform timing measurements using the hardware trace capabilities of the SoC. The Timing Suite T1 performs analysis and verification tasks. The chronVIEW and chronVAL suite of tools are developed by INCHRON GmbH [27]. They offer to perform statistical analysis of large hardware traces in order to execute verification of timing requirements together with the visualization of the RTOS scheduling.

3.3 Analysis Techniques

According to [28], the techniques for analysis of the performance tools that carry out schedulability analysis on real-time systems can be divided into two complementary groups:

- Approaches based on real-time scheduling theory, which are characterized by the estimation of the schedulability of a set of tasks. It is based on the analytical models that use formulas and theorems that normally assume corner cases such as the WCET and the WCRT. The result of these approaches can be too conservative due to the lack of accuracy in estimating the time values of the worst-cases.

- Model-based approaches to schedulability analysis, which are based on more refined models/representations of the system. The schedulability analysis of a system model seizes the specifications and details of the real-time tasks.

3.4 Stress Injection Evaluation

Previous works present different techniques to evaluate the robustness and performance of the system. The works related to real-time system evaluation through stress testing can be divided into two categories:

- **Offline Initial State:** The schedulability analysis is performed using formal methods based on the scheduling theory or using models such as genetic algorithms. A lower bound on timing parameters of the task set is defined in order to enforce the reachability in an initial state of the stress test.

- **Runtime Stress:** At runtime a framework plugged on the RTOS observes the execution of the system and the stress is injected

Most of the previous works deal with only a single stress testing technique of time-critical hardware-software systems [29], [30], [31]. In the works of [29] and [30], the authors propose different techniques for finding worst-case scenarios with respect to deadline misses. The work of [29] uses genetic algorithms to perform the search, while in [30] the search is based on constraint programming to generate the test cases that most likely will generate misses. The stress inputs are modeled as a combination of sequences of aperiodic tasks in the RTOS of the DUT. These works show that the search for these test scenarios is a complex and demanding task that requires large execution times. In addition, it was demonstrated that on large and complex systems constraint programming outperforms genetic algorithms in terms of efficiency. While genetic algorithms are able to generate very wide and heterogeneous test scenarios, it is not possible to guarantee the coverage, due to the search nature of the genetic algorithms. Further research should be performed on this issue.

In [31] the authors present a parametric model that enforces a specific test scenario under different time behavior of the system. It is composed by two steps. During the offline step, a set of delays is computed. Model-based approaches are used. During the online step, the system is monitored and delays are injected in order to enforce the desired behavior (e.g., simulate longer execution of a set of tasks on RTOS). This is implemented on the Cortex-M4 running Trampoline RTOS with a stress testing strategy based on the seeding time of aperiodic and sporadic tasks. After the first stage of *Offline Initial State*, the delays are injected at runtime to enforce the bounds established on the Initial State. The delays are defined with a timer on the RTOS and then with a framework plugged to the system to observe the behavior of the DUT. This technique allows to accurately perform a test. However, it is limited to enforce reachability properties. Complex properties are not considered (e.g., liveness).

This thesis developed a feature tool that performs a new method for the robustness and

performance analysis using the stress injection method by the CPU suspension technique. To date, no work or study has been developed about this new technique. This is due to the new hardware architecture embedded on the Tricore AURIX 2G, which supports the stress injection by three techniques: i) Artificial Reads; ii) CPU Interrupts; and iii) CPU Suspension. Despite the stress injection hardware architecture is already embedded in the AURIX Infineon microcontroller, it has never been used before. The work performed in this thesis allows the utilization of such an infrastructure for the first time. The stress injection is based on the reduction of the effective performance of a SoC component (IP hardware core) during short and periodic time intervals at runtime. The stress injection allows to assess the sensitivity of the SoC under different stress scenarios. These scenarios are defined on the *Offline Initial State* using formal methods of scheduling theory. Using the stress injection method, the SoC designer is able to identify possible risk scenarios testing the performance and robustness of the system at runtime. This thesis implements the stress injection by CPU suspension within two types of software application, RTOS and Bare-metal.

4 On-Chip Trace and Debugging Architecture

This chapter introduces the on-chip trace and debug solutions developed by Infineon Technologies AG for the AURIX microcontroller. The chapter is divided into six sections. The first section presents the general concepts and architecture of the on-chip trace. The second section introduces the architecture of the on-chip trace solution developed by Infineon Technologies AG. The third section describes the trace targets supported by the on-chip solution. The fourth section describes the Multicore Debug Solution (MCDS) a main component of the on-chip trace solution. The fifth section introduces the Device Access Server (DAS), the interface between the SoC and the trace tool. The sixth section presents the main trace tool of Infineon Technologies AG and the ChipCoach [1] tool, further extended in this thesis with ***Galenus***, to support stress injection.

4.1 On-Chip Trace Architecture

System-on-Chips (SoCs) are widespread in different markets, e.g., Internet-of-Things (IoT), avionics and automotive industries. Nowadays, SoCs in the automotive domain plays an important role, not only for the vast quantity of them integrated inside of a car but especially for the hard real-time constraints. Tracing and debugging capabilities are critical to design a SoC that meets all these constraints. However, for automotive applications, the use of traditional debug and trace methods is not recommended. Usually, these methods require the system to halt. These intrusive methods present two main drawbacks: i) the system behavior is interrupted, and ii) they perform only a static observation rather than a runtime system observation. Thus, a non-intrusive on-chip trace and debug solution able to analyze the system without perturbing the runtime behavior of the SoC is needed. Many SoC vendors already integrate such on-chip tracing and debugging capabilities, allowing at runtime the tracing of the SoC.

To perform the real-time tracing, an on-chip tracing and debugging hardware infrastructure is used. It allows the observation and annotation of the software execution on the SoC. This infrastructure is supported by circuits embedded on the SoC (target device) which are able

[1] Tool developed internally at Infineon Technologies AG

to: i) monitor different on-chip events; ii) generate a stream of trace messages containing the information of the events (e.g., timestamp, type of traced data, operation, address, source); and iii) output the generated trace outside of the SoC through a dedicated output port. Such information is gathered and analyzed by specialized software tools.

It is well-known that a single-core SoC may produce more than 10 gigabits per second of raw trace data [32]. The management of such a vast amount of data is still a challenge. In order to address this trace flow problem, three mechanisms are employed:

- **Trace compression**, that shrinks the generated information into smaller chunks of data.
- **Trace qualification**, that selects only the meaningful data.
- **Trace storage**, which saves the different on-chip events to the cost of the integration of massive memories.

The general architecture for the real-time tracing and debugging is shown in the Figure 4.1. As presented in [32], the architecture is composed by six-layers (from *L1* to *L6*).

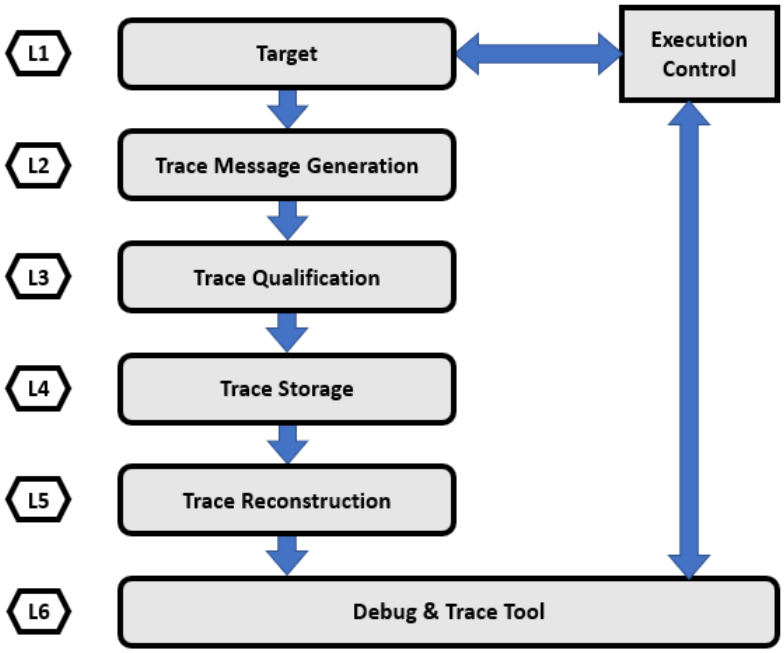

Figure 4.1: Trace and debug architecture

Usually, for the traditional debugging and tracing, the first three layers (*L1* to *L3*) are integrated on-chip, while the remaining layers depending on the used methodology (*L4* to *L6*) are performed off-chip. However, their implementation may differ according to the semiconductor vendor.

The tracing architecture is made by data streams obtained from different trace targets (e.g., CPUs, buses and memories). This data is reduced by the use of a trace qualification technique to be stored in a trace buffer. Finally, the stream is reconstructed and then analyzed through a debugger and trace tool. The details of each layer of the tracing and debugging architecture are described below:

- **L1-Target:** In this layer the trace and debugging targets are defined. It includes the type of the component (e.g., CPUs, buses, memories, interfaces), the type of information (e.g., value, transition, duration). Each trace target is connected through an adaptation logic to an observation point. Depending on the target, different types of trace information can be gathered.
- **L2-Trace Message Generation:** In this layer the data control is performed. The data controller gathers all the trace and debugging messages from the different observation points. It compiles all the information and creates a single trace stream, which keeps the exact temporal order of all the trace messages.
- **L3-Trace Qualification:** In this layer is performed a reduction of the amount of trace and debugging data, defining which kind of trace messages are eligible to be stored. This reduction is known as the trace qualification process and is performed according to certain parameters (e.g., detail level of trace, address range, memory operations). This information allows to identify, for example, when a function is called by a particular task or a specific state of the SoC.
- **L4-Trace Storage:** After the trace qualification, in this layer, the trace data is stored into a memory. Usually, this layer is implemented off-chip but it may differ according to the chip vendor.
- **L5-Trace Reconstruction:** In this layer, the trace data is reconstructed. Usually, it includes several additional information (e.g., timestamps, data, address, operation).
- **L6-Debug and Trace Tool:** The final layer is in charge of setting several parameters, such as the trace qualification, the buffer size, tracing duration and trace targets.

4.2 Infineon Technologies AG AURIX On-Chip Trace solution

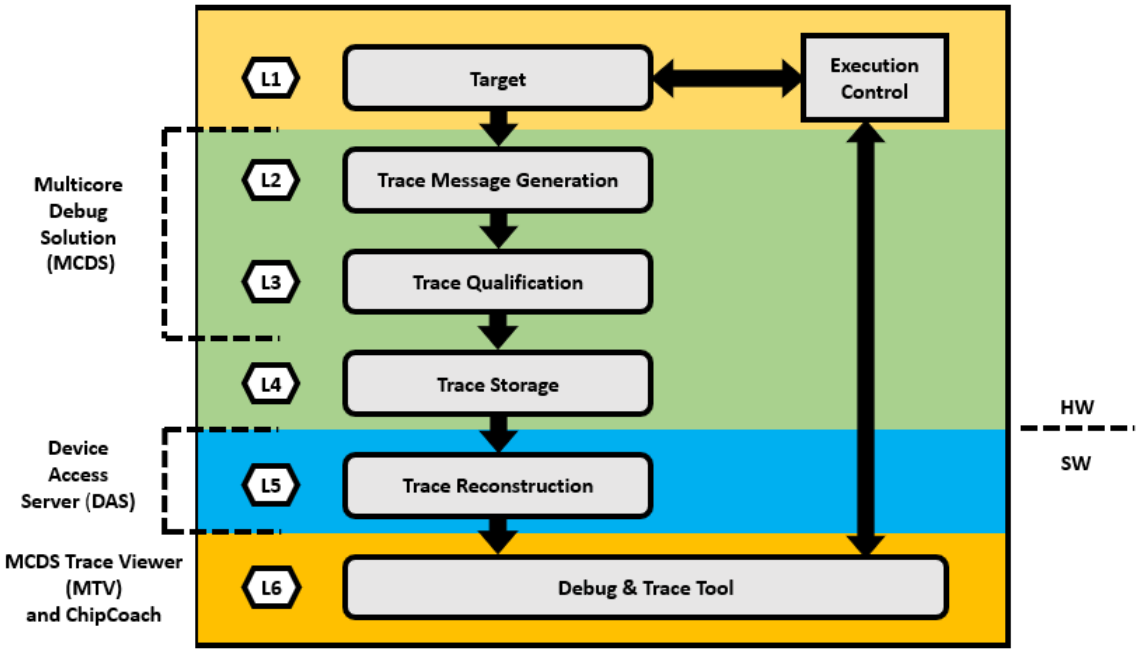

Figure 4.2: Infineon Technologies AG on-chip trace solution

Infineon Technologies AG developed an on-chip trace solution based on the trace architecture described in the Section 4.1. The on-chip trace solution of Infineon Technologies AG for the 32-bit AURIX microcontroller is shown in the Figure 4.2. The trace layers are divided in four groups: i) the trace target of the microcontroller (which implements *L1*); ii) the Multicore Debug Solution (MCDS) and the trace storage (which implement *L2*-*L3* and *L4*, respectively); iii) the Device Access Server (DAS) (which implements *L5*); and iv) the MCDS Trace Viewer (MTV) together with the tool ChipCoach (which implements *L6*). In this thesis, the capabilities of the ChipCoach tool were extended in order to be able to perform the stress injection. The feature is referred as ***Galenus***. In the following sections, each one of the layer groups will be further described.

4.3 Trace Target

The on-chip trace solution of Infineon Technologies AG supports two types of trace targets: CPUs and Buses. Depending on the trace target type, different information is available for tracing. Table 4.1 presents the five types of information that can be gathered by target. It includes: i) Process ID or task ID, that is usually an 8-bit to 16-bit information that allows the process identification; ii) Instruction pointer, which refers to the program flash address

that stores the instructions; iii) Data, which refers to the information that is written or read by the trace target; iv) Status, that indicates the state of the information from the target; and v) Watchpoint, that provides information of the watchpoint events produced by the matching between the instruction addresses or data and the programmed by the debug tool. Each trace target is connected to the MCDS through an Adaptation Logic (AL) which performs the signal adaptation between the trace targets and MCDS. Moreover, the *AL* block synchronizes the signals from the clock domain of the trace target to the MCDS clock domain. The design of the *AL* blocks depends on the type of the target. Each *AL* block connects the trace target custom interface (e.g. address, data or instruction pointer) that is connected to a generic standardized interface used by the MCDS [33].

Trace Information	**CPU**	**Bus**
Process ID	X	
Instruction Pointer	X	
Data	X	X
Status	X	X
Watchpoint	X	X

Table 4.1: Trace information according with the trace target

4.4 Multicore Debug Solution (MCDS)

The Multicore Debug Solution (MCDS) is the main component of the on-chip trace solution of Infineon Technologies AG. MCDS is an on-chip multicore trigger and trace system. It is composed by several configurable IP building blocks capable to provide the following functionalities: i) complex cross-target triggering; ii) trace qualification; iii) trace compression, and iv) timestamping. MCDS is available on the special Emulation Devices (ED) used during the development stage for tracing, profiling and verification. It allows a parallel recording of multiple trace targets into a single trace stream with the correct temporal order. Thus, capturing relevant data without modifying the application software. The MCDS is shown in the Figure 4.3. It consists of three main blocks: i) Observation block (*OB*); ii) Multicore Cross-Connect (*MCX*); and iii) Debug Memory Controller (*DMC*). Each one of these components is further explained in the next subsections.

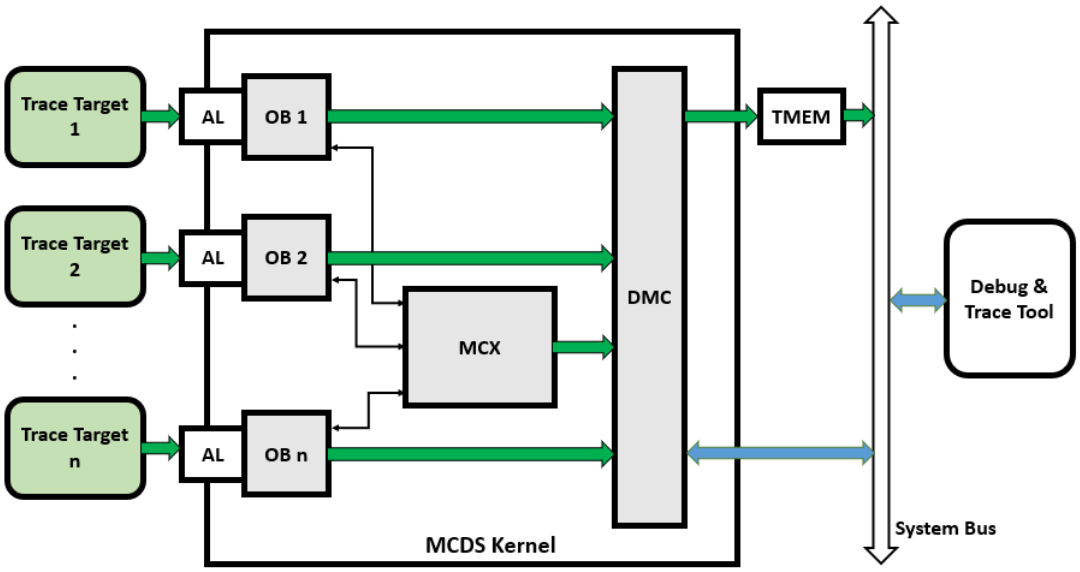

Figure 4.3: MCDS Architecture

4.4.1 Observation Block (*OB*)

The *OBs* are IP blocks that are connected to each trace target through an Adaptation Logic (*AL*). At each *OB* block, the trace qualification and trace message generation takes place. The interconnection of the different *OBs* with the remaining MCDS components and the trace targets is shown in Figure 4.3. In addition, the internal architecture of the *OBs* is shown in Figure 4.4. *OBs* are composed by three main building blocks: Trace Units, Trace Qualification Unit (*TQU*) and the Message Sequencer Unit (*MSU*).

- **Trace Units**, correspond to IP blocks able to encapsulate the gathered data from the *AL* into a trace message. They are identified as the ① in Figure 4.4. The integration of a certain type of *Trace Units* IPs is determined by the trace requirements. Table 4.2 presents the five most common Trace Units and the type of gathered information.
- **Trace Qualification Unit (TQU)**, that corresponds to an IP block programmed by the trace tool and which contains a filter mechanism able to control the trace message that is sent to the Message Sequencer Unit (MSU). They are identified as the ② in Figure 4.4.
- **Message Sequencer Unit (MSU)**, which is an IP block in charge of sorting the trace messages from the different Trace Units. This sorting is performed based on the trace time tags. They are identified as the ③ in Figure 4.4.

The information gathered from the Trace Target is properly interfaced through the *AL* and after being processed by the *OB*, the sorted information is sent to the Debug Memory Controller (DMC) block.

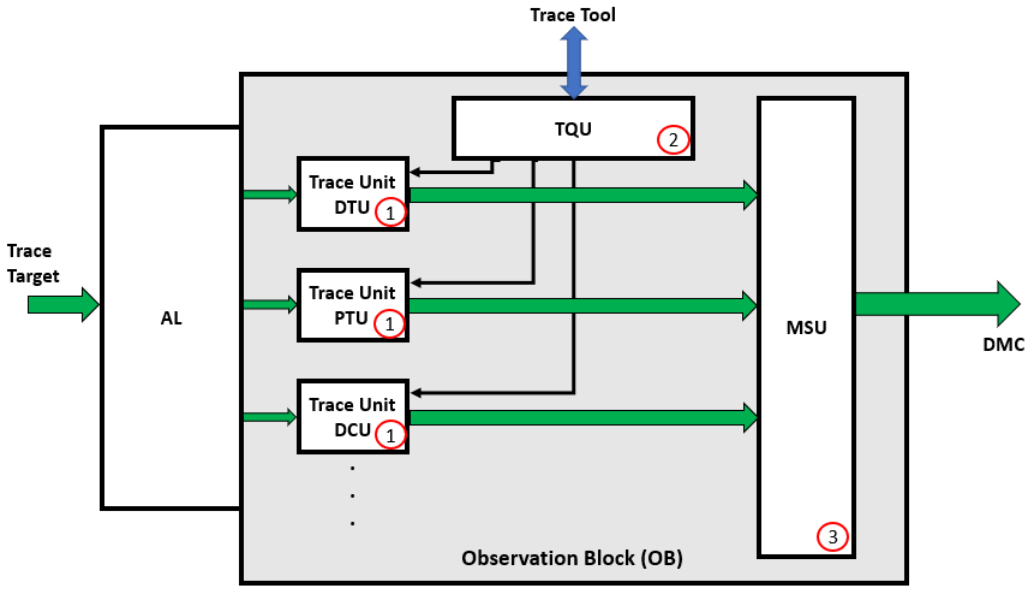

Figure 4.4: Main blocks of the Observation Block

Trace Unit Type	Target Information
Data Trace Unit (DTU)	Data
Program Trace Unit (PTU)	Instruction Pointer
Debug Status and Control Trace Unit (DCU)	Status
Ownership Trace Unit (OTU)	ID Process
Watchpoint Trace Unit (WTU)	Watchpoints

Table 4.2: Trace unit types according with the target information

4.4.2 Multicore Cross Connect (*MCX*)

This block is connected with all the *OB*s through several input and output trigger lines (cross triggers). The MCX is responsible for the distribution of the cross triggers. The cross triggers are programmable and can be configured through the Trace Qualification Units embedded at the *OB*s (as shown in subsection 4.4.1). Additionally, the MCX provides a central timestamp for all the trace messages. The output of the MCX feeds the Debug Memory Controller (DMC) block.

4.4.3 Debug Memory Controller (*DMC*)

This block collects the trace messages generated from the different *OB*s and the timestamp generated by the *MCX*. In addition, DMC has the functionality of sorting the trace messages according to the time tags added by the *Trace Units* IPs inside the *OB*s and the MCX timestamp. In case several trace messages present equal time tags, they are combined into a single trace message. This technique ensures the trace message ordering, which is mandatory for the trace reconstruction at the trace tools.

The collected trace messages by DMC are written into one single trace stream and then stored in an on-chip trace memory (TMEM), usually dual-ported. It allows simultaneously the read and write of the gathered trace messages. A debug interface allows the transmission of the on-chip trace data to the off-chip trace tool for analysis purposes. Typically, the trace memory size is in the range of several kilobytes. In order to better profit and enhance the trace capabilities of the system, only relevant trace data is captured and compressed. This process highly depends on the Trace Unit type used in the *OB*s as discussed in subsection 4.4.1. For example, for the Program Trace Unit (PTU), the data compression is performed through the storage of only the difference between two successive instruction pointer values. However, for the Data Trace Unit (DTU), the complete addresses and data must be stored.

4.5 Device Access Server (*DAS*)

Infineon Technologies AG has developed the Device Access Server (DAS), a software that is able to perform the interface between the SoC (Device Under Test) and the different debugging and trace tools. DAS is used for multicore systems with high demanding emulation requirements. The DAS Application Programming Interface (API) is implemented on software through generic Dynamic Link Libraries (DLLs). DAS is executed on the host PC together with the debug and trace tools and it allows that these tools access the On-Chip Debug Support (OCDS) and the Multicore Debug Solution (MCDS) through the USB port, as shown in Figure 4.5. Alternatives for accessing the hardware includes a miniWiggler, a converter between USB and DAP/JTAG.

4.6 Infineon Technologies AG AURIX Debug and Trace Tools (MTV and ChipCoach)

For several complex systems in the automotive domain, such as the power-train control, it is mandatory to analyze the system behavior under all the possible operational scenarios.

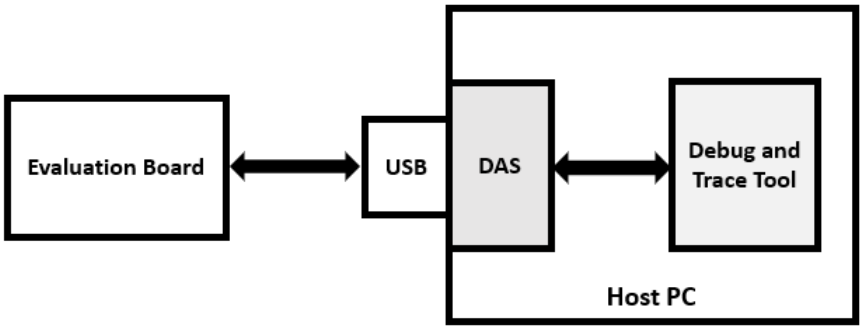

Figure 4.5: Device Access Server(DAS) Configuration

This analysis allows the evaluation of the reliability of the system, that is, the capability of SoC is able to meet the functional and performance constraints under any scenario. As discussed in Section 4.4, the MCDS infrastructure is used to perform on-chip tracing and debugging functionalities. It includes on-chip trigger generation, trace data compression, and trace storage. Due to the memory limitation, only relevant trace data is provided to the debug and trace tools. Therefore, even under complex scenarios, it is possible to design reliable systems. Infineon Technologies AG has developed several tools for their AURIX microcontrollers. In this work is described in detail the main trace tool known as the MCDS Trace Viewer (MTV) tool and the internal developed ChipCoach tool. The ChipCoach tool was extended in this thesis in order to support the stress injection feature through ***Galenus***. The extension of the tool is described in detail in Chapter 5.

4.6.1 MCDS Trace Viewer (MTV)

MTV is a non-intrusive trace tool developed on top of the DAS interface with two main functions: i) the MCDS trace configuration; and ii) the trace decoding. The MTV tool uses the MCDS trace qualification and triggering capabilities to monitor and trace many trace targets with different range of detail. In order to perform this tracing, it is necessary to configure some MCDS registers. The MTV tool provides an easy-to-use and friendly interface in which the user can configure the trace and analyze the decoded data trace in an efficient manner. The MCDS Trace Configuration and the Trace Decoding are further detailed below.

- **MCDS Trace Configuration:** It sets the values of the main parameters of a trace using MCDS. The parameters that are configured include the trace targets, the on-chip trace buffer mode (by default the buffer stops the trace once it is completely filled) and many other parameters. Furthermore, it is possible to configure the *Trace Units* of the *OB* blocks, defining the trace qualifiers and the triggers. After the parameters are set, MTV generates automatically the proper configuration information which is then

transmitted to the MCDS.

- **Trace Decoding:** To get full information of the trace data, it is possible to load the Executable and Linkable Format (ELF) file. After the trace is performed, the trace data is stored in the TMEM. Then, MTV reads the trace information along with the associated *ELF* file. The trace data is the information gathered using the trace qualifier and trace triggers, while the *ELF* file contains the instructions that were executed on the micro-controller. MTV uses this information and displays it to the user through a chart. It includes the timestamps, the address, the data, and many others.

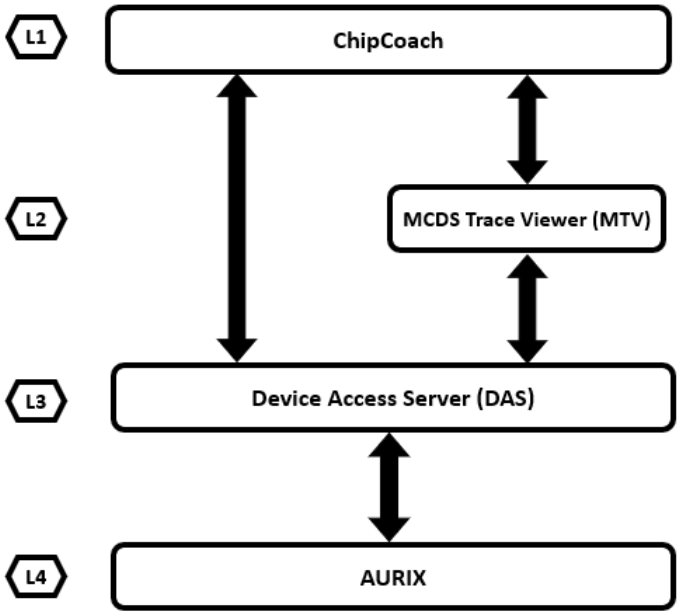

Figure 4.6: ChipCoach Layers

4.6.2 ChipCoach

The ChipCoach is a tool developed by Infineon Technologies AG and is based on the MCDS hardware tracing functionalities embedded in the AURIX microcontrollers. ChipCoach is able to configure the MCDS and gather the trace messages in order to perform a different types of analysis of the system. This tool runs on Windows and uses DAS and MTV libraries. ChipCoach is programmed on top in Java and in the lower layers in C/C++. Figure 4.6 shows the four layers for the tracing and debugging flow for the ChipCoach tool.

This tool allows to perform different types of post-processing and runtime system diagnosis using the trace and debug functionalities of the AURIX microcontrollers. The architecture of

this tool allows to develop advanced features based on the trace scope configuration and the trace messages gathered. Therefore, by exploiting these tracing control functionalities, in this thesis, the tracing control functionalities were used in order to develop a new feature capable to perform stress injection in the AURIX microcontrollers. The new feature called ***Galenus*** allows to gather and filter the trace messages in order to analyze the performance and robustness of the system at runtime when the stress injection is performed. This feature is explained in detail in Chapter 5.

5 Stress Injection

This chapter presents the first contribution of the thesis: the design of the stress injection feature for the SoC evaluation. This feature was included in the Infineon Technologies AG design flow for tracing and debugging, more precisely, at the ChipCoach tool. Despite the hardware architecture to support the stress injection that is already embedded in the AURIX 2G Infineon microcontroller, it has never been used before. The work performed in this thesis allows the utilization of such an infrastructure for the first time. This chapter is divided in four sections. The first section presents a general description of the Stress Injection. The second section presents the *Trigger Line Timer* (TLT) of the *OCDS Trigger Switch* (OTGS) that is used to perform periodic suspensions of one or more CPUs. The third section describes the Stress Injection Trigger Line Timer that is part of OCDS in the central debug interface. The fourth section describes the developed stress injection feature called *Galenus*.

5.1 Stress Injection

In this Section, the general concept of the *stress injection* is presented. The first subsection describes the goal of the stress injection in a SoC and the second subsection describes the types of stress injection.

5.1.1 General description

The stress injection is a technique for performance and robustness evaluation of a SoC that must meet hard real-time constraints[1]. It is based on the reduction of the effective performance of one or more CPUs by periodic intervals. The stress injection allows to assess the sensitivity of the SoC under different stress scenarios. A robust system is able to meet the hard real-time constraints even under a high level of stress injection (in this

[1] The evaluation of the performance and robustness of the SoC usually is called as the *SoC health test*. By following this medical analogy, the SoC stress injection is similar to a human breath test. The breath test is a **non-invasive method** to help doctors to **diagnose a number of conditions** in a human being. By **analyzing** the breath, the amount of certain gases is **measured**, allowing doctors to arrive in a diagnosis regarding the health of the patient **quickly and accurately**. In this case, the patient is the SoC.

work a robust system is also called as *healthy system*). The stress injection is a key tool for SoC diagnosis and debugging. By using stress injection, the SoC designer is able to identify possible risk scenarios (where the system is not able to meet the hard real-time requirements) and to trigger possible mechanisms for solving/improving and supporting the reliable operation of the SoC.

5.1.2 Types and Requirements

According to the stress injection method and the IP hardware block target for the performance degradation, three types of stress injection techniques can be identified: i) stress injection by artificial reads; ii) stress injection by CPU interrupts; and iii) stress injection by CPU suspension. These techniques of stress injection are periodically performed for a brief interval of time to analyze the overall effect on the SoC. In order to analyze the effect of the stress injection, the SoC under test must support access to the trace and debug infrastructure. By using this, a designer is able to configure the system and to gather the information required to analyze the SoC behavior.

5.2 Infineon Technologies AG AURIX On-Chip Debug and Suspend Generation

The On-Chip Debug Support (OCDS) is the on-chip debug solution for the Infineon AURIX microcontroller family. The OCDS infrastructure is based on an on-chip network of coupled *debug units*, which also are interconnected to the processing and communication components of the SoC (e.g., CPUs, bus controllers, peripherals, interrupt requester). This network enables to gather the different debug events of the system. The general structure of the OCDS is shown in Figure 5.1. OCDS is a complex structure that integrates many components. For performing the stress injection (goal of this thesis) four main elements of the OCDS are highlighted: i) Cerberus; ii) OCDS Trigger Switch (OTGS); iii) Trigger Lines (TL), and iv) Trigger Line Timer (TLT). Further details of these components are given below.

1. **Cerberus**: It is the central debug interface for the set of on-chip *debug units*. It integrates three main components. First, the OCDS Trigger Switch (OTGS), which is the central component for run-controlling. It controls the propagation of the signals *suspend* and *halt* through the so-called *Trigger Lines*. These transmission lines are linked to all the trace targets (e.g., CPUs, peripherals). Furthermore, the *OTGS* monitors the trace targets and guarantees that the collected data is forwarded to the *MCDS* for their later analysis at the debug and trace tools.

2. **OCDS Trigger Switch (OTGS)**: It is embedded in the Cerberus and its function is to control the operation between the trigger sources and trigger targets. That is, OTGS is able to route the triggers from the MCDS (trigger source) to the CPU (trigger target). The trigger routing (transmission) is performed through the *trigger lines*. This allows to route one or more trigger sources to several trigger targets. The process (software component) that defines the state of the *OTGS*'s (*trigger lines*) is the *Multicore Break Switch (MCBS* and is executed on the OTGS. In case of a CPU suspension request (assertion of *suspend* signal), the CPU pipeline is either stalled or delayed through the injection of NOP (no operation) instructions. This state is kept until the *suspend* signal is deasserted. Then, the CPU can resume the operation.

3. **Trigger Lines**: These lines correspond to the transmission wires used to route the triggering signals. These lines link the OCDS with the different trigger sources and trigger targets. The trigger lines are especially important because they allow the synchronous suspension of the trigger targets (e.g., CPUs, peripherals). Once a trigger line is set, a suspension is performed in all the IP blocks that are sensitive to this trigger line, that is, the suspension signal is broadcasted to all the sensitive IP blocks. A suspension will generate two effects on the IP blocks: i) halting of new bus transactions, and ii) completing the ongoing computation/processes, but without executing new processes (also called *delayed suspension*). Note that as a default, the IP blocks of the SoC are not sensitive to a suspend request. The configuration of the IP block sensitivity to a trigger line is explained in Section 5.3. The only trigger line that is able to implement the *delayed suspension* is the trigger line 1 (*TG Line 1*). This trigger line will be used in this work to implement the stress injection.

4. **Trigger Line Timer (TLT)**: It is a hardware component embedded within the OTGS and that can be used to trigger periodic suspensions in the trigger targets (e.g., CPUs and peripherals). The timer is based on a synchronous counter which decrements with every clock. The suspension is triggered when the counter value reaches 0 and remains in suspension by 12 CPU clock cycles. The configuration of the timer is done by modifying the initial value of the counter (the new value is stored in the counter register). This modification is performed through the MCDS and the debug tool. This timer is used in this thesis in order to perform the stress injection. This feature will be further described in detail in the Section 5.3 of this Chapter.

The generation of suspension signals is performed using these blocks. Note that the MCDS (trigger source) is configured by a debug and trace off-chip tool. The process of suspension/activation is executed through four steps. The first step takes place when the MCDS triggers the *suspend* signal in the Cerberus OTGS block. When a periodic suspension is required, the MCDS also configures the TLT of the OCDS. In the second step, the execution of the MCBS on the OTGS component is performed. As a result, a suspension is created. In the third step, this signal is transmitted through the *TG Line 1* to the sensitive trigger targets (e.g., CPUs, peripherals). Then, these blocks are suspended. The consequence of the suspension of the IP hardware blocks of the SoC is the immediate halting of new

transactions and the completion of the remaining pending transactions. In the fourth step, the *suspend* signal is cleared after 12 CPU clock cycles, then the CPUs are able to resume the operation.

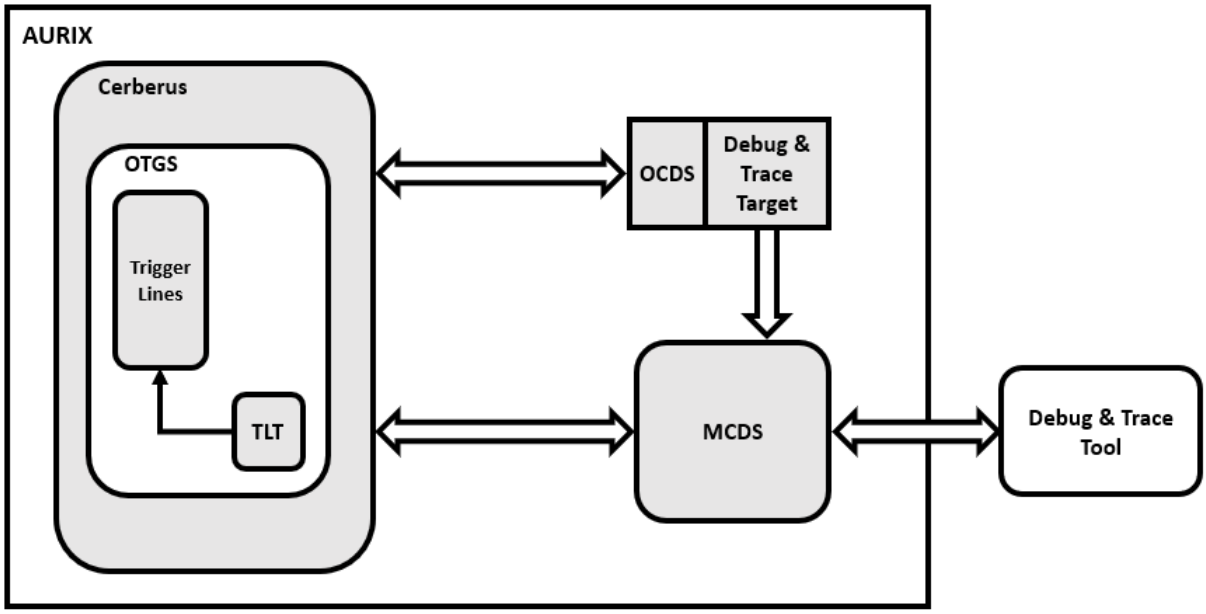

Figure 5.1: OCDS and Suspend Generation

5.3 Stress Injection Trigger Line Timer

The *Stress Injection Trigger Line Timer* and the *Injector for Faults and Stress*(IFS) are Cerberus functionalities that evaluate the performance and the robustness of the DUT in a systematic and repeatable way. This work uses the*Stress Injection Trigger Line Timer* to apply the stress to the DUT and *IFS* only will be explained. *IFS* performs a predictable fault injection for registers and controlled periodic stress injection. By using the stress injection, two scenarios can be evaluated: i) to determine the amount of stress that causes the first performance and robustness degradation symptoms of the system, and ii) to determine the sensitivity of a specific performance metric in the system. For example, the CPU time required to execute a task under a defined amount of stress.

The method of stress injection intentionally reduces the effective performance of a SoC resource (e.g., CPU, memory) in order to analyze the overall effect on the performance and robustness of the system. There are three possible types of stress injection: i) Stress injection by artificial reads; ii) stress injection by CPU interrupts, and iii) stress injection by CPU suspension.

1. **Stress Injection by Artificial Reads**: Defines a periodic number of consecutive read accesses to the shared resources. It uses the OTGS Trigger Line Timer (TLT).
2. **Stress Injection by CPU Interrupts**: Defines a periodic interrupt request for a CPU. It employs the TLT and the interrupt routing.
3. **Stress Injection by CPU Suspension**: Defines a periodic suspension of the CPU. It uses the TLT of one or all CPUs. The number of clock cycles during which the suspension is performed should be defined by the SoC designer.

Among all these stress injection techniques, in this thesis the *stress injection by CPU suspension* method was selected using *Stress Injection Trigger Line Timer*. The suspension of the CPU has the greatest impact on the SoC performance when compared to the remaining two techniques. Furthermore, for critical systems, the CPU timing parameter is the most important performance component in a hard real-time system. For example, if the break by wire with an anti-lock system in a car is starved of CPU time due to a CPU overload, a deadline miss or a response miss can be provoked. Thus, producing a potentially serious consequence in the safety of the user. In order to perform the stress injection by CPU suspension, the process described in Section 5.2 is followed. The trace tool configures the sensitive list of the suspension targets (e.g. the CPUs) through the MCDS and OCDS components. Furthermore, the trace and debug tool additionally configures the trigger line timer in order to perform a periodic suspension. These two configuration processes are explained below:

- **Suspension Targets Configuration:** In order to configure the suspension targets to perform the stress injection it is necessary to turn the CPUs sensitive to the *suspension* signal. This process is done through the trigger line with the functionality of suspension, as explained in the Section 5.2. By modifying the 32-bit register of the central debug interface (Cerberus) that corresponds to the *Trigger Line 1* of the OTGS, the sensitivity to suspension can be activated. Note that a single or multiple CPUs can be sensitive to suspension. The *Trigger Line 1* broadcast the suspend signal to all the CPUs and peripherals that are sensitive to this line.

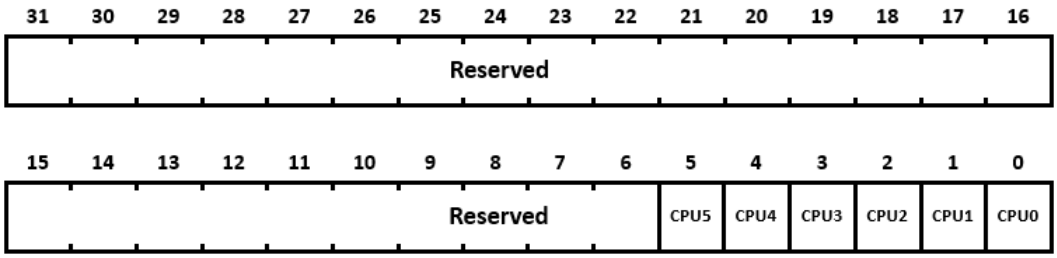

Figure 5.2: 32-bit Register of the Trigger Line 1 Suspension Targets

- **Timer Configuration:** It is used to perform a periodic suspension of the CPUs for stressing the system and analyzing its behavior. The OTGS of the central debug interface (Cerberus) has a timer function that allows to implement periodic suspension with a fine resolution (from 12 to 780000 CPU clock cycles). The timer configuration requires the setting of four parameters (by modifying the value stored in the 32-bit register of Cerberus): timer value, reload timer (RL), trigger line value (VTZ) and timer to trigger value (TGL). The structure of the register is shown in the Figure 5.3. The parameters are discussed below.
  - **Timer Value:** The timer value represents the number of clock cycles between each suspension. It is represented as a 16-bit value which is automatically decremented each 12 clock cycles until it reaches the zero value.
  - **RL:** The Reload Timer generates a periodic suspension. It is represented as a 1-bit. When the *RL* is set to high (logic 1), the *Timer Value* is reloaded to the initial 16-bit number after the counter reaches zero. Otherwise, when the RL is set to low (Logic 0), the *Timer value* field of the register is not reloaded after the zero is reached.
  - **VTZ:** The *Trigger Line* value stores the value that will be routed through the *Trigger lines* after the counter reaches zero. To activate the suspension of the sensitive suspension targets, *VTZ* should be set to high (logic 1).
  - **TGL:** It controls the Timer to Trigger Line routing. It is represented as a 4-bit value, where each value correspond to the trigger line selected. The activated line will transmit the value stored at *VTZ*. In order to transmit the suspension, the *TG Line 1* is used. It is able to activate the sensitive trigger targets.

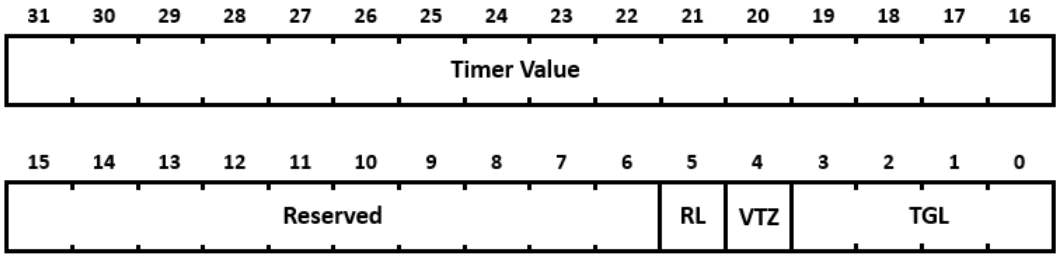

Figure 5.3: 32-bit Register of the Trigger Line Timer

The stress injection based on the CPU suspension technique is performed by turning the CPUs (sensitive trigger targets) sensitive to the suspension. The periodicity of the suspension is achieved through the timer function of the central debug interface. The schematic of the single and periodic suspensions are shown in the Figure 5.4. The periodic suspension constitutes the so-called stress injection. By modifying the 16-bit data stored at the *Timer Value* the interval between t_0 and t_S is configured, as shown in Figure 5.4 (a). The CPU

suspension is fixed to at least 12 CPU clock cycles and it is represented as the interval from t_S to t_R. Note that the CPU suspension can be greater than 12 clock cycles. It will depend on the pipeline state of the CPU at the instant of time when the CPU suspension is activated. In order to achieve a periodic stress injection, the *RL* should be set to high (logic 1). In this way is performed periodically the stress injection to the system.

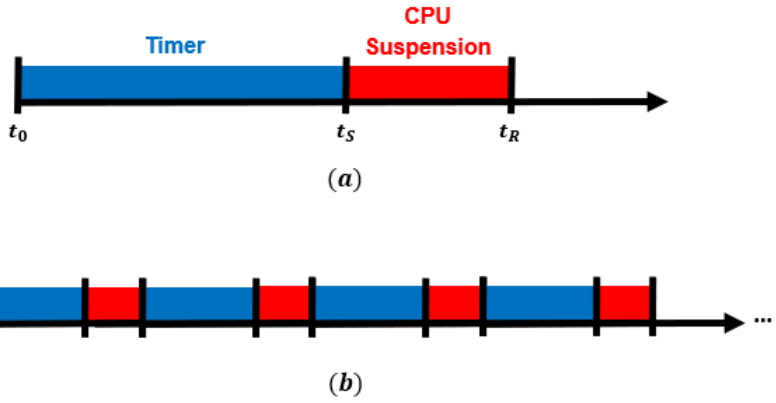

Figure 5.4: Stress injection (a) Single suspension (configurable *Timer value* and fixed *CPU suspension*); (b) Periodic suspension.

5.4 Developed Stress Injection Feature in ChipCoach

Following the international standards for functional safety of electronics systems for the automotive domain (ISO 26262), Infineon Technologies AG has developed a new generation of AURIX microcontrollers. These devices have the capability to perform resource usage tests when the software integration tests are executed on the hardware. To perform this new functionality using the embedded debug and trace infrastructure, Infineon Technologies AG has developed ChipCoach, a tool introduced in the Chapter 4. This thesis further extends ChipCoach by integrating a new feature that enables stress injection. In this thesis, the new feature will be called as ***Galenus***.

Galenus is an additional feature of ChipCoach, programmed in Java and fully configurable by the user. The *Galenus* is capable to implement the stress injection through CPU suspension within two types of software applications (RTOS and Bare-metal). Furthermore, it quantifies a set of metrics based on the gathered information of the SoC that allows the analysis of performance and robustness of the system. *Galenus* has four main functions: i) configure the trace; ii) configure the stress injection; iii) sort and map the trace, and iv) generate the evaluation metrics. Depending on the application, all or a set of these functions can be used. For instance, RTOS applications employ all the functionalities of the *Galenus*. In

contrast, Bare-metal applications do not require the Sort and Map Trace functionality. The four functionalities of *Galenus* are described below.

1. **Configure the Trace:** It sets the values of the parameters that define the trace characteristics. The trace is configured by setting the trace target, the observation block with the trace units, the record buffer size, the qualifier and the record type.
2. **Configure the Stress Injection:** It defines the characteristics of the stress injection. The stress injection is configured by setting the parameters of the registers for the timer and suspension targets configuration as described in the Section 5.3.
3. **Sort and Map the Trace:** It configures the sorting and mapping characteristics of the stress injection. After the trace stream is gathered from the DUT, the trace is sorted and mapped according to the requirements of the test.
4. **Generate the Evaluation Metrics:** According to the sort, the timing parameters are obtained and the evaluation metrics of the performance and robustness of the system are generated.

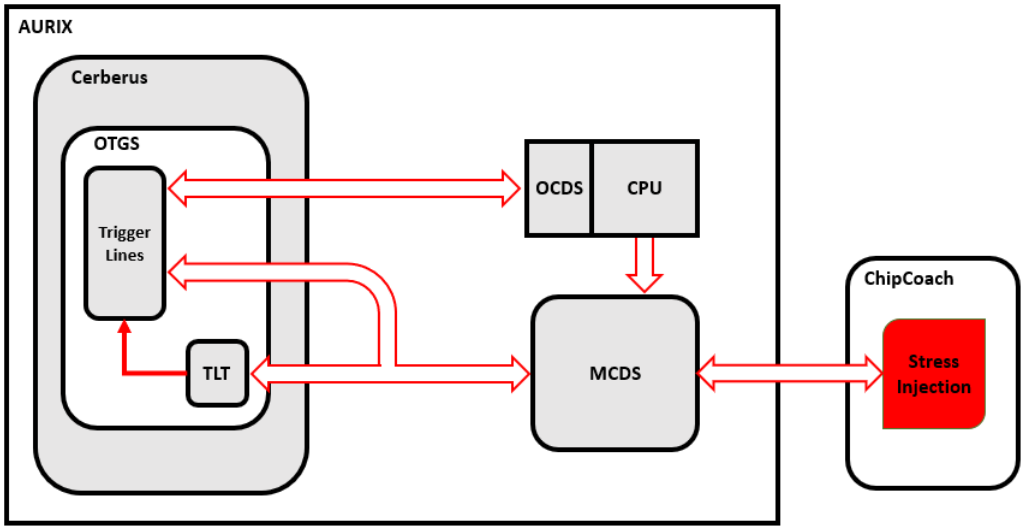

Figure 5.5: Data flow diagram of the stress injection feature *Galenus*.

The data flow of *Galenus*, is shown in the Figure 5.5. *Galenus* configures the MCDS in order to set the trace configuration and to apply the stress injection. MCDS configures the register of the central debug interface, then set the timer and the suspension targets. Afterwards, the suspension is done using the OCDS component in the CPU. While the stress is being performed, MCDS records the trace and sends it to the Host-PC running ChipCoach. Then, *Galenus* gatherers and sorts the trace stream to quantify the set of evaluation metrics that allow the evaluation of the performance and robustness of the system.

The block diagram of *Galenus* is presented in the Figure 5.6. It is composed by four blocks, one for each function of the feature. The enumerated circle identifies the functionality previously described. In the following subsections, each one of the blocks will be further described in detail.

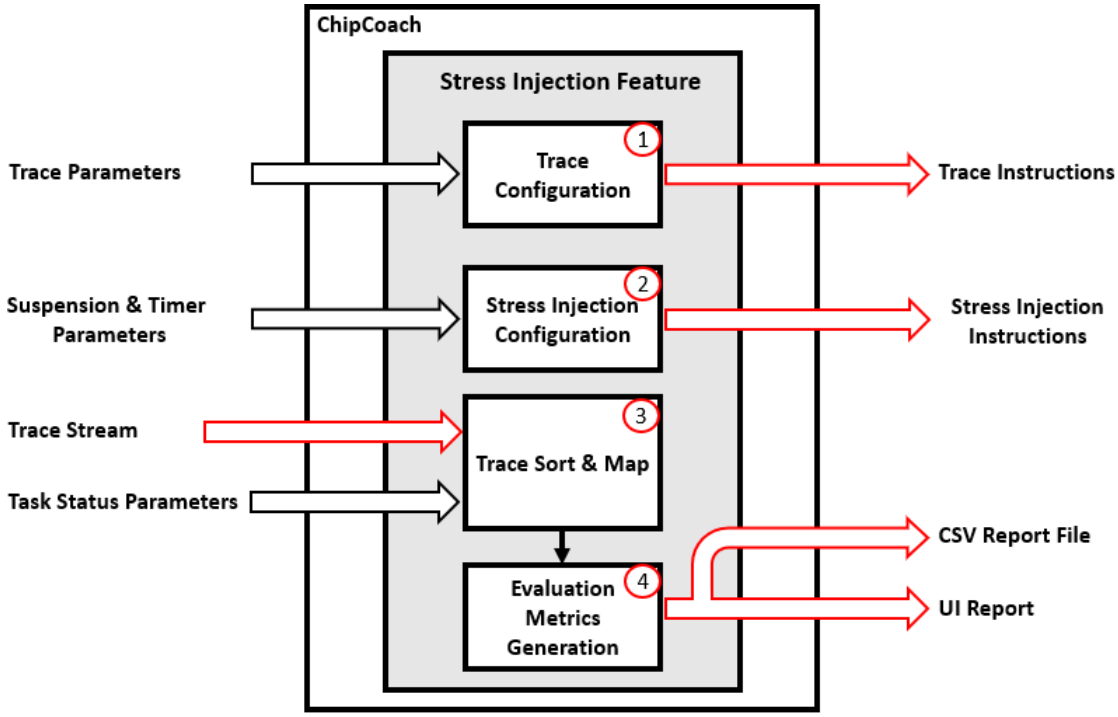

Figure 5.6: Block diagram stress injection feature.

5.4.1 Tracing Configuration

The trace configuration block is responsible to define the setup of the trace that will be performed in the DUT when the system is stressed. This configuration is defined according to the six input parameters:

- **Trace Buffer Parameters:** Refers to three parameters of the on-chip trace buffer: i) The size of the buffer that is used to store the recorded trace data (e.g., 16kB); ii) The buffer record mode (e.g., record Until full mode or the circular tracing stopped by a trigger mode); iii) The trigger position type, that indicates the amount of data to be traced after the trigger conditions are met (e.g., 30%, therefore 70% of the buffer is filled with the trace data after the trigger conditions are met).
- **Trace Target OB:** This parameter defines the Observation Block (OB) in the trace target (e.g., CPU1).

- **Timestamp Type:** This parameter defines the type of timestamp of the trace message of the MCDS (e.g., Ticks enabled).
- **Data Trace Unit Parameters:** Refers to three parameters that should be set in order to configure the DTU. These parameters are i) The trace Target (e.g., CPU1); ii) The type of range of the qualifier (e.g., in-range or out-range), that define in which regions of the program memory the trace must be performed. In case the qualification is in-range, the trace will be performed only if the *Instruction Pointer* points to an address inside the defined ranges; iii) The type of trace data to be captured (e.g., the data and the address of all the write and read operations).
- **Qualifier Parameters:** Refers to two parameters that should be set in order to configure the qualifier. These parameters are i) The type of range of the qualifier (e.g., in-range); ii) The start and the end address for the qualifier.
- **Continuous Trace Parameters:** Refers to three parameters that should be set when the continuous trace is used. These parameters are: i) The use of the reset before the trace is started (e.g., enable reset); ii) The maximum tracing time until the trace stops (e.g., 3 seconds); iii) The maximum amount of data until the trace stops (e.g., 10kB).

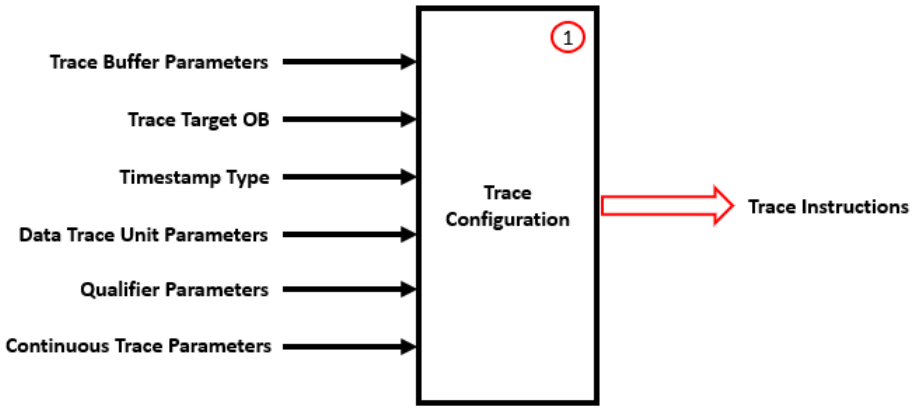

Figure 5.7: Block diagram of the trace configuration function.

The value of these parameters configures the trace in order to extract the desired trace data. It defines the trace parameters, which include: the trace target, the used trace unit and the region of the memory in which the system should be traced. By applying this configuration to the MCDS, we can now start to trace the DUT. The next functionality, the stress injection configuration, is then activated.

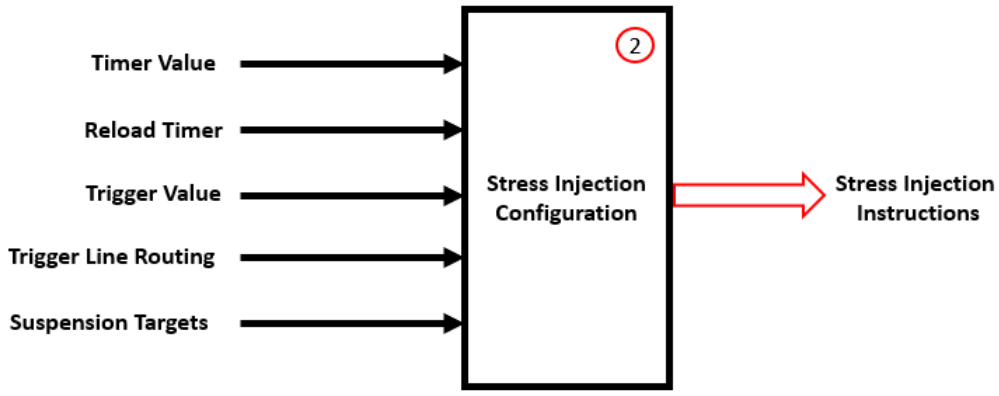

Figure 5.8: Block diagram of the stress configuration function.

5.4.2 Stress Injection Configuration

The stress injection configuration block is responsible for defining the settings to perform the stress injection. The stress injection is performed through the CPU suspension technique. Therefore, the tracing CPUs targets should be sensitive to suspension and the configuration of the trigger line timer, to perform a periodic suspension, should be configured. In order to perform such a task, the stress injection configuration block requires five inputs: i) Timer Value, a 16-bit value that represents the number of clock cycles between each suspension; ii) Reload Timer, a 1-bit value used to enable a periodic timer; iii) Trigger Value, a 1-bit value that represents the value set to the trigger when the timer reaches the zero; iv) Trigger Line Routing, a 4-bit value that defines the Trigger Line in which the function of the timer is communicated; and v) Suspension Targets, which defines the IP blocks that will be suspended.

According to the value of the aforementioned parameters, the stress injection configuration block communicates to the MCDS a set of values that will be written down into the MCDS registers. The write access is performed immediately after the request and the functionality of the MCDS is then configured. Once the trace and stress configuration is done, the tracing is started and the trace data is retrieved by the Trace Sort and Map block.

5.4.3 Trace Sort and Map

The Trace Sort and Map block is responsible for filtering the trace stream gathered from the SoC. Afterward, when an RTOS application is implemented, the mapping is performed. Otherwise, in the case of a Bare-metal application, the trace data is automatically redirected to the *Evaluation Metrics Generation* block. To perform the sorting and mapping process,

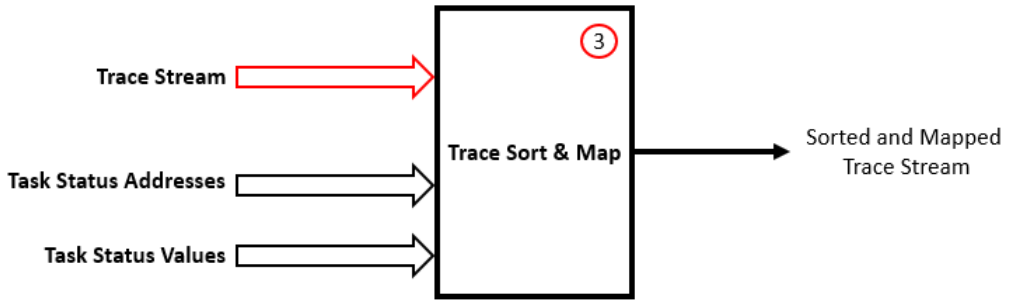

Figure 5.9: Block diagram of the trace sort and map function.

the *Trace Sort and Map* block requires three information: i) Trace stream, which refers to the data stream acquired from the SoC. The trace is gathered after the trace and stress injection configuration are performed and after the tracing process takes place; ii) Task Status Addresses, which refers to the memory addresses that contain the status of the task set of the RTOS; and iii) Task Status Values, which refers to the values that represent the states of a task as explained in Chapter 2 (e.g., Suspended (3), Ready (2), Running (0)). According to the values of these three inputs, the sort and mapping process are performed in three stages. Figure 5.10 shows the three-stage filtering process for a single trace message. This process is applied to all the trace stream data. A further detailed description of the stages is presented in the next paragraphs.

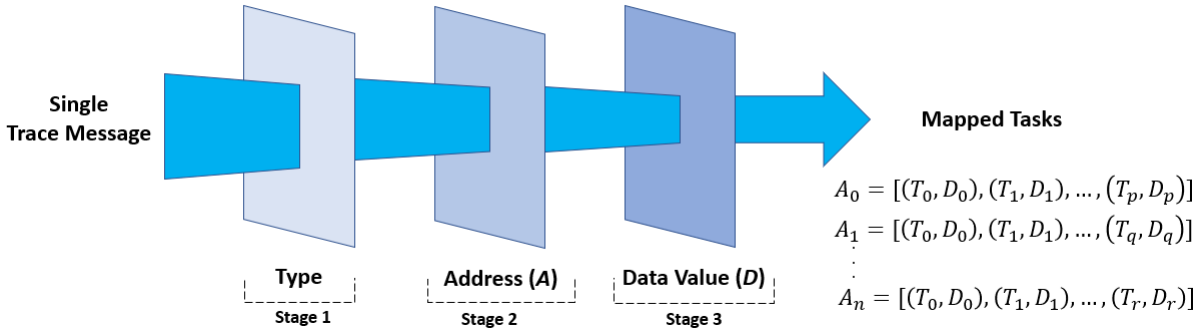

Figure 5.10: Single trace message stages of mapping.

- **Stage 1:** It filters the trace message according to the type. Only the trace message type *Data* can continue to stage 2. The remaining types (e.g., Trace Gap, End of Trace, FIFO Overflow) are discarded.
- **Stage 2:** It filters the trace message according to the address. Only the trace messages with the memory address equal to the defined Task Status Addresses input information can continue to the stage 3.

- **Stage 3:** It filters the trace message according to the value of the data. Only the trace messages with the data value equal to the defined Task Status Values input information can continue to the mapping process.

After the single trace passed through all the three stages, the map is performed following the data structure shown in the right side of the Figure 5.10. This structure is composed by two elements called *Key* and *Value*. The *Key* is unique for the Task Status Address. Therefore, the number of keys is equal to the number (*n*) of tasks in the RTOS. The *Value* is a list of tuples composed by the timestamp *T* and the Task Status Values *D*. The possible values of *D* are Suspended (3), Ready (2), Running (0). These values may vary according to the selected RTOS.

5.4.4 Evaluation Metrics Generation

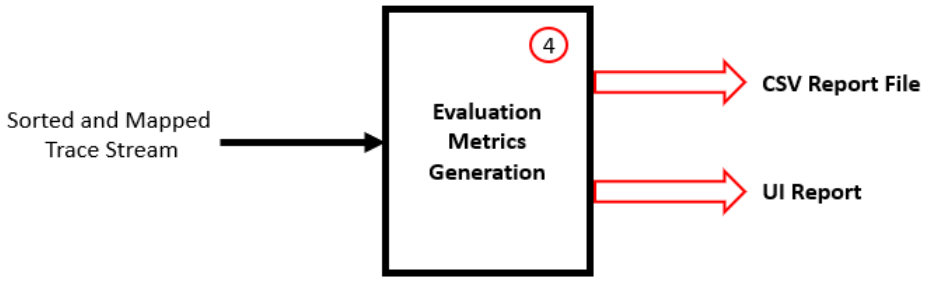

Figure 5.11: Block diagram of the evaluation metrics generation function.

The Evaluation Metrics Generation block is responsible for calculating the metrics used for evaluating the DUT. The set of metrics may vary according to the software application test. In the case of an RTOS test, the input of the block is the mapped tasks generated by the Sort and Map Trace block. Otherwise, for a Bare-metal test, the input is directly the trace message without mapping. Each case will be further described in the following paragraphs:

- **RTOs Test:** To quantify the metrics for evaluating the performance and robustness of the system under stress injection and when an RTOS software application test is used, three stages should be performed.
  - **Stage 1 - Timing Parameters Trace:** This stage is responsible for measuring the timing parameters based on the monitored task-set, generated by the block of Trace Sort and Map. The timing parameters we can derive from our trace match the parameters that were already introduced in Chapter 2. Each mapped task is described by their *Value*, a list of tuples (T_i, D_i), where *T* refers to the

timestamp and *D* is the *Task Status Values*. To obtain the task timing parameters, a Mealy Finite State Machine (FSM) was designed and implemented. The high-level representation of the FSM is shown in Figure 5.12. This FSM is composed by five states, which corresponds to the initial state (Init), three *Task Status Values* (Suspended, Running, Ready) together with the preempted state. The way that the FSM is activated is used to calculate three aspects of the task: timing information, number of preemptions and number of instances.

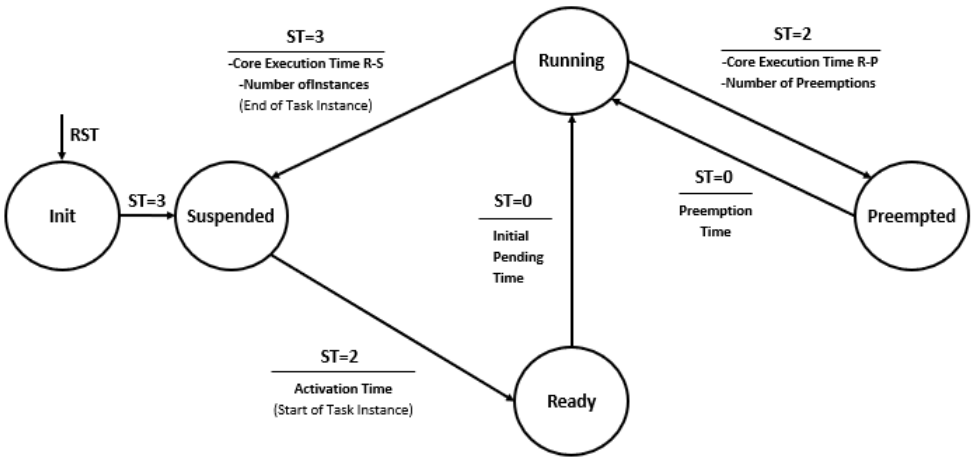

Figure 5.12: Designed Finite State Machine and the timing parameters.

The FSM is represented by a 6-tuple $(\Sigma, \Lambda, S, s_0, F, \delta)$, where Σ is a finite non-empty set of symbols called the input alphabet, Λ is a finite non-empty set of symbols called the output alphabet, S is a finite non-empty set of states, s_0 is the initial state of S, F is the final state of S and δ is the state transition function. The designed FSM is described as follows:

$\Sigma = \{0, 2, 3\}$ refers in this work as the *Status Values of the Task* (*ST*).

$\Lambda = \{$*AT*, *IPT*, *CET/R-P*, *PT*, *CET/R-S*, *NP*, *NI*$\}$, where the output alphabet corresponds to the timing parameters. They are (in order) the Activation Time, Initial Pending Time, Core Execution Time R-P, Preemption Time, Core Execution Time R-S, Number of Preemptions and the Number of Instances.

$S = \{Init, Suspend, Ready, Running, Preempted\}$ refers to the states of the FSM.

$s_0 = \{Init\}$, refers to the initial state of the FSM. It corresponds to the Init state.

$F = \{Suspended\}$, refers to the final state of the FSM. It corresponds to the suspended state.

$delta$, the transition function of the FSM is shown in the Table 5.1.

Current State	Input ST	Next State	Output
Suspended	0	Suspended	-
Suspended	2	Ready	$ActivationTime = T_n - T_{n-1}$
Suspended	3	Suspended	-
Ready	0	Running	$InitialPendingTime = T_n - T_{n-1}$
Ready	2	Ready	-
Ready	3	Ready	-
Running	0	Running	-
Running	2	Preempted	$CoreExecutionTimeR - P = T_n - T_{n-1}$ $NumberofPreemptions = n + 1$
Running	3	Suspended	$CoreExecutionTimeR - S = T_n - T_{n-1}$ $NumberofInstances = n + 1$
Preempted	0	Running	$PreemptionTime = T_n - T_{n-1}$
Preempted	2	Preempted	-
Preempted	3	Preempted	-

Table 5.1: Transitions function of the FSM.

The output of the FSM corresponds to the timing parameters of the system. The values are acquired by the difference between the current timestamp T_n and the previous timestamp T_{n-1}. Figure 5.13 presents an example of the use of the FSM to gather the timing parameters. The input information is a mapped task (performed by the previous block), where the address of task 1 is mapped together with a list of tuples of the timestamp and the *Status Value of the Task* (*ST*).

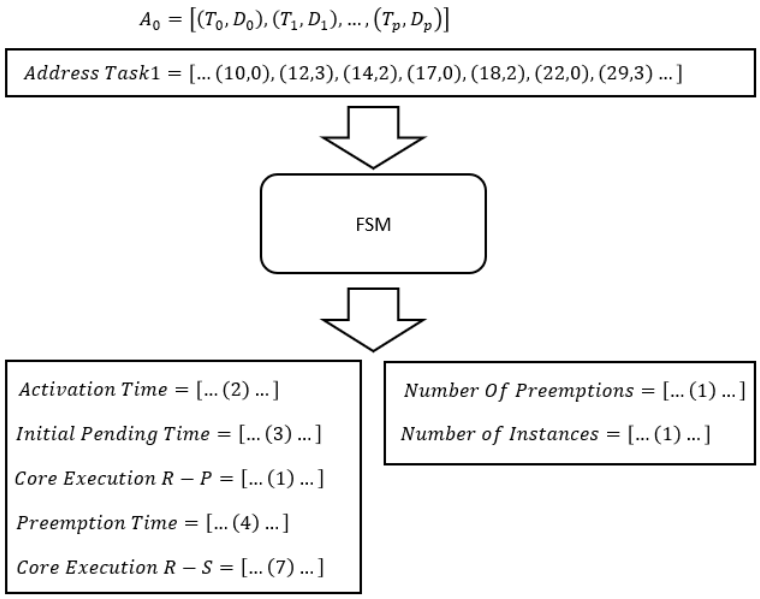

Figure 5.13: Example of the stage 1 with the developed FSM

For the sake of simplicity, in this example only seven tuples of a preempted task are shown: (10,0), (12,3), (14,2), (17,0), (18,2), (22,0), (29,3). Note that integer numbers are used to represent the magnitude of the timestamp, and milliseconds are used as units of measurement. After the initialization, the FSM is initiated at the Init state.

For the first tuple (10,0), $ST = 0$, therefore the FSM does not perform any transition. A change of state takes place only when *ST* is equal to three. This fact ensures the measurement of a full instance of a task instead of a partial instance. Following this process, for the second tuple (12,3), $ST = 3$, and then the first transition takes place. The current state is $Suspended$.

For the third tuple (14,2), $ST = 2$, then the next state is $Ready$. The output timing parameter ***Activation Time*** is equal to the difference between the timestamp of the current tuple and the timestamp of the previous tuple. Therefore, the *Activation Time* is equal to $AT = T_n - T_{n-1} = 14 - 12 = 2\,ms$. The result is then added into a list that records the timing parameter.

For the fourth tuple (17,0), $ST = 0$, then the next state is $Running$. The output timing parameter ***Initial Pending Time*** is equal to the difference between the current timestamp and the previous timestamp. Therefore, the value of $IPT = T_n - T_{n-1} = 17 - 14 = 3\,ms$. For the fifth tuple (18,2), $ST = 2$, meaning that the task has been preempted by higher priority task. Then, the next state is $Preempted$ which results to the timing parameter ***Core Execution Time R-P*** to be equal to $CET/R - P = T_n - T_{n-1} = 18 - 17 = 1\,ms$ and the ***Number of Preemptions*** $NP = 1$.

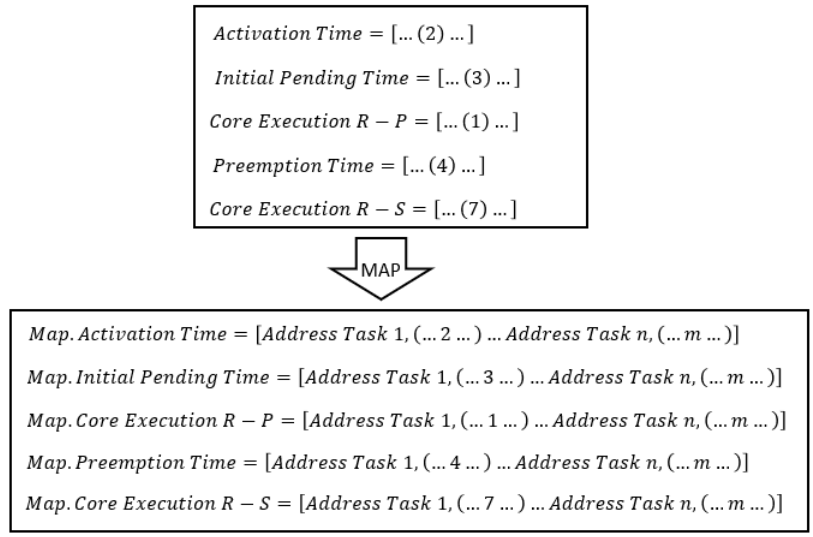

Figure 5.14: Mapping example of the timing parameters in the Stage 1

For the sixth tuple (22,0), $ST = 0$, then the next state is $Running$. The timing parameter ***Preemption Time*** $PT = T_n - T_{n-1} = 22 - 18 = 4\,ms$. For the

seventh and final tuple of the example (29,3), $ST = 3$, then the next state is $Suspended$, which signals the end of a task instance. The value of the timing parameter ***Core Execution Time R-S*** $CET/R-S = T_n - T_{n-1} = 29 - 22 = 7\,ms$ and the ***Number of Instances*** $NI = 1$.

This process is repeated until the complete trace data is processed. In case there is a non-complete instance of a task, e.g., in the presence of a gap in the trace when a continuous trace is performed, then the interrupted timing parameters are not considered. This guarantees data reliability. When the process is finished for a single task, all the list values of each timing parameters are mapped for the respective task, as shown in Figure 5.14. The process is done for every task.

□ **Stage 2 - Complex Timing Parameters:** This stage is responsible for computing more complex timing parameters (e.g., the ***Execution Time, Response Time and the Period***) based on the gathered values mapped at Stage 1. The three complex timing parameters are described below and an example is given, which follows the example presented at Stage 1.

The *Execution Time* is defined as the time required for the implementation of a task in the CPU, as described in Chapter 2. To compute the Execution Time, the *Core Execution Time R-P* and the *Core Execution Time R-S* of an instance of a task are added. This ensures that the metric considers the preemption of a task. The values obtained at Stage 1 are used and the process is performed for all the task set. The results are included in the Execution Time map for each task as shown in the Figure 5.15

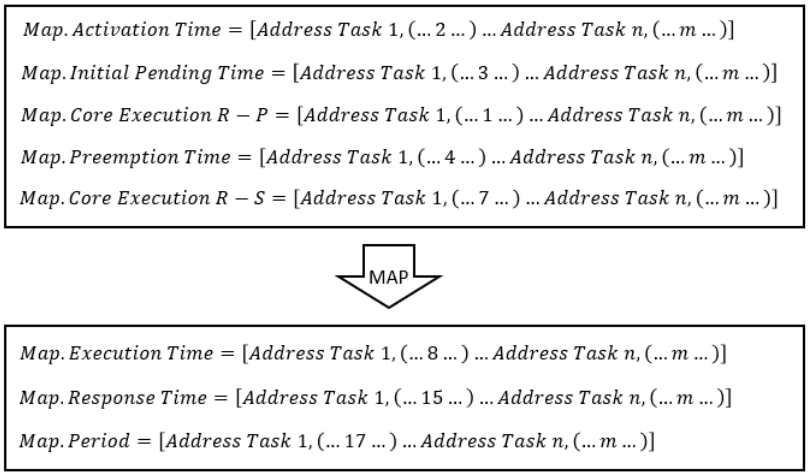

Figure 5.15: Mapping example of the complex timing parameters in the Stage 2

The *Response time* is defined as the time required for a task to be completed. The value of this parameter is equal to the addition of all the timing parameters since the task is in *Ready* state. Therefore, the response time involves the summation of all the timing parameters excluding the Activation Time from the Stage 1. The addition is performed when the instance of a task is completed and then is mapped for the respective task, as shown in Figure 5.15. The process is repeated for all the tasks.

The *Period of a task* is defined as the interval the task is repeatedly activated, as explained in Chapter 2. It is necessary to add all the timing parameters of the respective task in order to quantify the *Period*. That is, it is equal to the addition of the *Response Time* and the *Activation Time*. Afterwards, the value is mapped for the respective task and repeated for all the tasks. The Figure 5.15 continues with the proposed example for Stage 1. It shows the mapping process for the Execution Time, Response Time and the Period for the Address Tasks 1 (with values) and other address tasks. For the mapped Execution Time, the value stored at the list is $ExecutionTime = CET/R-P - CET/R-S = 1 + 7 = 8\,ms$ (addition of the Core Execution R-P and the Core Execution R-S). For the mapped Response Time, the value stored at the list is $ResponseTime = IPT + CET/R-P + PT + CET/R-S = 3 + 1 + 4 + 7 = 15\,ms$(addition of all the timing parameters excluding the activation time). Finally, for the mapped Period, the value is$Period = AT + IPT + CET/R-P + PT + CET/R-S = 2 + 3 + 1 + 4 + 7 = 17\,ms$.

□ **Stage 3 - Descriptive Statistics and Utilization:**

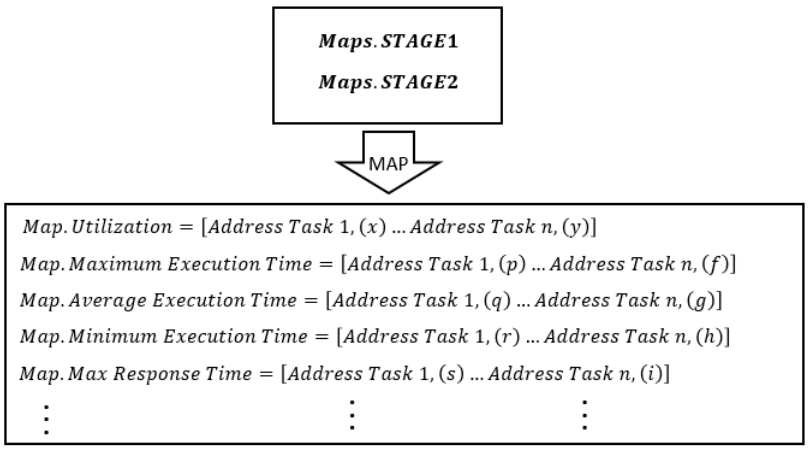

Figure 5.16: Mapping example of the descriptive statistics in the Stage 3

This stage is responsible for calculating further statistics regarding the behavior of the mapped timing parameters from *Stage 1* and *Stage 2* and for mapping the results for the task set. The calculated statistics include the maximum, average

and the minimum of each timing parameter. Moreover, the designer can use these values to further calculate elaborated measurements (e.g., deviation, distribution, utilization). Afterwards, the descriptive statistics are mapped for each task as shown in Figure 5.16. The presented values follow the example started in Stage 1.

Once the descriptive statistics are quantified, it is possible to calculate the **Utilization** for each task. As described in the Chapter 2, the utilization of a task is defined as the quotient of the maximum execution time and the period.

- **Bare-metal Test:**

  In order to analyze the instruction dependency against the stress injection, a Bare-metal test for different types of instructions was designed. It uses a flow control variable and the System Timer Module (*STM*). The *Galenus* is able to measure the time it takes for a defined amount of instructions to be executed under different levels of stress injection.

  The algorithm implemented in the Bare-Metal environment is shown in the Figure 5.17. It is defined a flow control variable (called *Block*) in a *while* loop and can be modified by the *Galenus*. When the ***Block*** variable is changed to false, the Start Time is taken from the STM module. Afterwards, the test instructions are executed. Finally, it is acquired the elapsed time of execution after all the instruction of the same type are executed. The variable *Duration* stores the difference between the current value of the STM and the Start Time value.

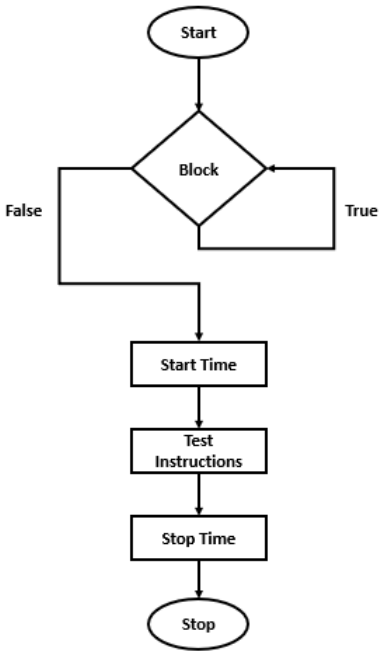

Figure 5.17: Algorithm implemented for the Bare-metal Test of instruction dependency

For this type of test, the stress injection feature is responsible for: i) modifying the value of the flow control variable called *Block*; and ii) gathering the interval of time required to execute a defined amount of instructions.

By using the tool framework to decode symbolic data and addresses of the ELF file, it is possible to obtain the respective address for a selected variable of the system. Therefore, once the ELF file is loaded to the tool framework it is possible to write and read the values of the variables contained in the registers. In order to perform a stress injection test and evaluate the set of metrics for a Bare-metal application in *Galenus*, the following steps should be performed: i) load the ELF file of the Bare-metal system to the tool framework; ii) perform the stress injection configuration by the *Stress Injection Configuration* block explained in detail in the subsection 5.4.2; iii) modify the flow control variable *Block*; iv) gather the value of the *Duration* variable, which registers the elapsed time of execution of the instructions.

With this test, it is possible to quantify the exact time it takes a defined amount of instructions to be executed when the stress injection is performed. As the flow control variable is modified after the stress injection is activated, it is guaranteed that the gathered time is related only to the interval for the instruction set execution. By using different types of instructions and applying the same amount of stress injection, it is possible to identify the presence of certain instruction dependency for the stress injection.

5.4.5 Report of Evaluation Metrics

Once the evaluation metrics are generated for the two supported software application tests, *Galenus* is capable to provide the evaluation metrics of the stress injection to the software developer. Two methods are used to represent the quantified values: i) to display the evaluation metrics through a table in the User Interface (UI) of the tool framework, and ii) to generate a Comma-Separated Values (CSV) file with the quantified evaluation metrics and the descriptive statistics for further analysis. These two methods are described in detail below:

- **UI Report**

  The designed User Interface (UI) for *Galenus* is shown in the Figure 5.18. The designer can use the on-chip tracing capabilities with and without stress injection and can obtain a set of evaluation metrics. One of the goals of the UI is the better visualization and understanding of the system behavior, as well as better control of the stress injection capabilities. Note that this UI can be easily modified in order that the designer has a refined control over the stress injection. The first release of *Galenus* was designed to provide the basic functionalities of the stress injection. To expose and turn available the different parameters of the stress injection is a decision of the

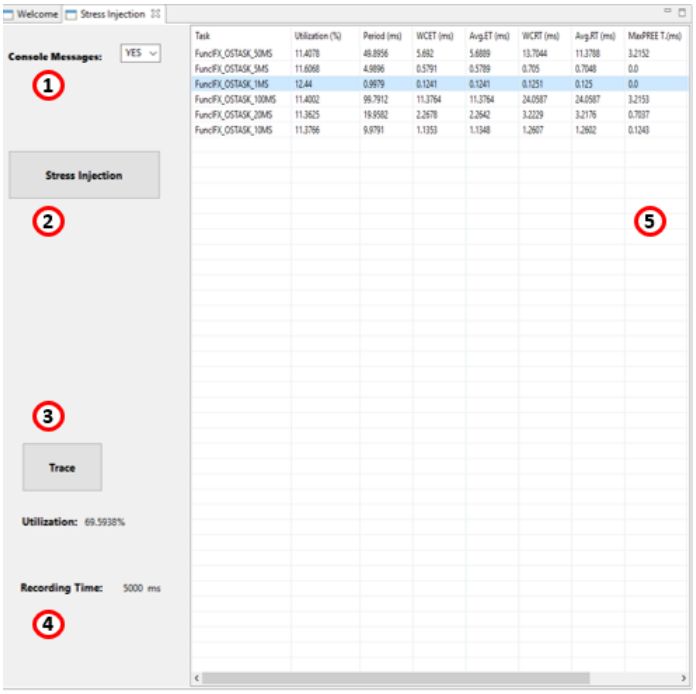

Figure 5.18: User Interface of *Galenus*

chip vendor. Further parameters can be opened in the future to the designer. These basic and minor changes are left as future work. The User Interface is composed by five main elements indicated with an enumerated circle in the Figure 5.18. These elements are further described below.

(1) **Console Messages Option:** This feature offers the designer a very refined information regarding all the on-chip trace processing and evaluation. It includes the trace gathering, the several filtering steps and concludes with the quantification of a set of metrics. The software developer can observe step-by-step the data flow transformation, allowing a better understanding of the system. The information is shown in the tool console.

(2) **Trace with Stress Injection Button:** This feature is represented as a button that allows the execution of the stress injection test. Once the button is activated, the system is traced with the functionality of the stress injection. Once the test is finished, a CSV file is automatically generated and the result of a set of metrics is obtained.

(3) **Trace without Stress Injection Button:** This feature is represented as a button that executes the tracing of the system without the stress injection. This feature is used to gather the initial trace of the system without the stress injection. The trace values are shown in the table of the UI.

(4) **Recording Time and Total Utilization data:** This feature shows the results of two main information, named the recording time and the total utilization data. The

information is generated every time that the trace test is finalized.

(5) **Evaluation Metrics Report Table:** Once the test is finalized the main evaluation metrics are displayed in the table of the UI. This information provides a brief report of the test to the software developer.

The User Interface allows the software developer to perform the stress injection and to visualize a brief report of the evaluation metrics. This report is also generated as a CSV file, which contains all the trace data and the evaluation metrics obtained through the stress injection test.

■ CSV Report File

Each time a stress injection test is performed, a CSV file is generated automatically. The CSV is a common data exchange format that is widely supported by different types of programs (e.g., MATLAB, Excel). By using this format, the software developer can quickly create a program to import the files, to plot the information (using different colors and styles) and to analyze the data. The CSV file usually contains the following information:

- ☐ The timing parameters of a task for RTOS and Bare-metal tests. For an RTOS test metrics such as *Response Time* and *Execution Time* are obtained. For a Bare-metal test, the metric such as the interval of time of execution by the instruction set can be obtained.
- ☐ The number of instances and the utilization of a task.
- ☐ The recording time of the trace.
- ☐ The number of gaps in case of continuous tracing.
- ☐ The amount of performance degradation.
- ☐ The parameters of the test (e.g., the period of the timer of the stress injection).
- ☐ The descriptive statistics for all the evaluation metrics (e.g., maximum, average, minimum).

6 Methodology

In this chapter, the methodology proposed in this thesis is presented. The goal of the methodology is to measure the performance and the robustness of the *DUT* through the technique of *stress injection* by CPU suspension implemented within two types of software application, RTOS and Bare-metal. This chapter is divided into four sections. The first section presents the general description of the methodology. The second section describes the overall methodology for the RTOS test composed by six blocks. The third section describes the overall methodology for the Bare-metal test composed by six blocks. Finally, the fourth section presents a summary of the methodology.

6.1 General Description

The performance and robustness of an embedded system vary according to the type of load and the operating conditions. This is especially critical for Systems-on-Chip (SoCs) that are subject to hard real-time constraints. Therefore, to guarantee the meeting of the time requirements of the DUT, stress testing is required. Software designers must analyze and understand the behavior of the SoC under a wide variety of scenarios. Usually, the load of the SoC is described by a set of parameters, which when set to a selected set of values, are able to create different load scenarios. Once the DUT is stressed by a load scenario, the system behavior can be observed. To execute a system analysis of hard real-time applications, an efficient, non-intrusive and parallel system observation structure is indispensable. Therefore, the stress test analysis must be done through an on-chip trace structure embedded on the DUT. The trace of the system is performed at run-time while the system is stressed. Among the different types of stress, in this work has been selected the stress injection through CPU suspension, which was described in detail in the Chapter 5. This technique has been selected for its huge impact on the DUT, achieving a system degradation on a larger scale when compared with the other stress techniques. In order to gather and evaluate the behavior of the DUT, a trace stream generated by the DUT is captured. Based on such information, a set of metrics are quantified. This type of test can be implemented for two types of software application, RTOS and Bare-metal.

6.2 RTOS Test Methodology

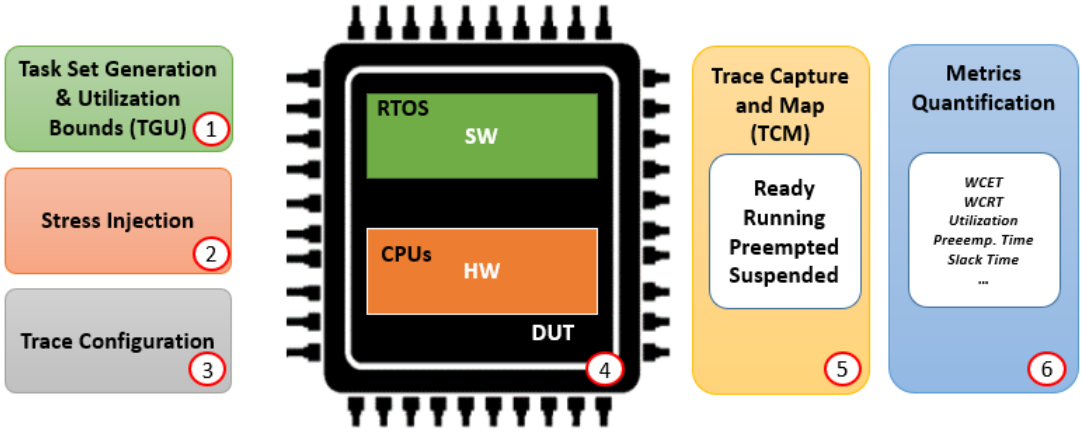

Figure 6.1: General description of the RTOS test methodology.

The high-level representation of the method proposed in this work for an RTOS test is presented in 6.1. It is composed by six blocks, indicated with an enumerated circle:

(1) **Task Set Generation and Utilization Bounds (TGU):** Generates a *feasible* task set and the utilization bounds. A feasible task set is defined when every task in the set meets its deadlines (Subsection 6.2.1).

(2) **Stress Injection:** Generates the stress injection parameters for the debug interface (Subsection 6.2.2).

(3) **Trace Configuration:** This block configures the trace that will be performed in the DUT when the system is stressed (Subsection 6.2.3).

(4) **Design Under Test (DUT):** It is composed by software and hardware components. It schedules the task set along with the stress injection. The DUT generates a stream of trace data (Subsection 6.2.4).

(5) **Trace Capture and Map (TCM):** Gathers and sorts the stream of trace data of the task set. (Subsection 6.2.5).

(6) **Metrics Quantification:** Quantifies a set of metrics able to evaluate the performance and robustness goals (Subsection 6.2.6).

In the following subsections, each one of the blocks will be further described.

6.2.1 Task Set Generation and Utilization Bounds (TGU)

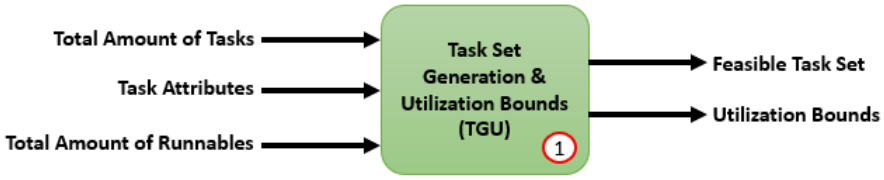

Figure 6.2: Task Set Generation and Utilization Bounds (TGU) block.

The *Task Set Generation and Utilization Bounds* (TGU) is the first block of the proposed methodology as shown in Figure 6.1 as (1). The TGU is in charge of performing three functions. The first function of TGU is to generate a feasible task set *S*. It corresponds to a collection of *n* tasks *J*, where each task *J* is a collection of *m* runnables *B*. The task set *S* and the task *J* are defined as in (6.1) and (6.2), respectively.

$$S = [J_1, J_2, ...J_n] \tag{6.1}$$

$$J = [B_1, B_2, ...B_m] \tag{6.2}$$

The second function of TGU is to distribute the task load among the runnables *B*. The third function of the TGU is to define the utilization boundaries of the task set.

The *TGU* has three parameters as an input: i) the total amount of tasks *n*, which constitutes the feasible task set; ii) the task attributes *A* as in (6.3), that includes the period *T*,defined as the regular interval of time of activation of the task, offset *F*, defined as the range of time between the starting of the system and the first occurrence of the task, priority *P*, which represents the relative importance among the tasks that are executed in the system. Tasks with higher priority are executed before the tasks with lower priority and the activation of shared resources *H* in a task, defined as a binary value that represents the activation of shared resources of a task; and iii) the total number of runnables *m* per task *J* to be assigned to the task set.

$$A = [T, F, P, H] \tag{6.3}$$

TGU quantifies the lower bound for the total utilization U_b of the system based on the number of tasks *n* and on the UBA schedulability test (described in the Chapter 2, where the value of U_b is defined as in (6.4).

$$U_b = n(2^{1/n} - 1) \tag{6.4}$$

Once the lower bound is quantified, afterward, for each task, the task load L is quantified as in (6.5) using the period T of the task and the total number of tasks n. It assumes an even distribution of the utilization of the tasks.

$$L = \frac{U_L * T}{n} \tag{6.5}$$

The upper bound of total utilization of the system is calculated using the exact test *RTA* approach explained in the Chapter 2. It employs the task load L (quantified previously) and the recursive equation (6.6). As a result, the WCRT R_i of the task J_i can be quantified as in (6.7).

$$R_i^{(0)} = L_i \tag{6.6}$$

$$R_i^{(k)} = L_i + \sum_{j=i}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil L_j \tag{6.7}$$

In the previous equations, L_i is the task load of the task J_i. Assuming without loss of generality, that the tasks have an implicit deadline (Deadline *D* = Period *T*) and that the task J_j has a higher priority than J_i. The equation (6.7) can be solved iteratively till R_i converges. In order to determine the schedulability of J_i the converged value is then compared against the implicit deadline of J_i. If $R_i \leq D_i$ for the complete task set, then the task set is schedulable. Otherwise, the task set is not schedulable.

By using the equations 6.5 till 6.7 and increasing the value of U_L for the task load iteratively in 6.5 until the task set is no longer schedulable in 6.7, it is possible to determine the upper bound value of the total utilization of the system. Hence, the first output of *TGU* is generated, that is, the utilization bounds of the system.

Finally, after the feasible task set is obtained, the load per task is distributed evenly between the number of runnables. In this way, *TGU* generates a feasible task set and the utilization bounds for a non-faulty analysis of the DUT that which is stressed. The generated feasible task set is scheduled by the RTOS in the CPUs of the *DUT* and the utilization bounds will support the analysis once the evaluation metrics are acquired.

6.2.2 Stress Injection

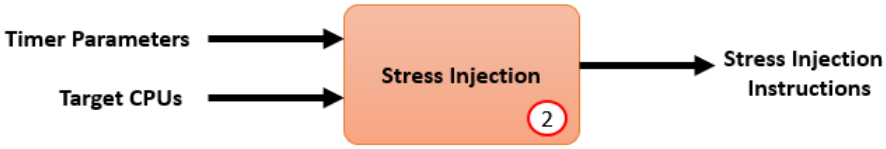

Figure 6.3: Stress Injection block.

The Stress Injection block, indicated in the Figure 6.1 as (2), generates the parameters for the on-chip trigger and trace interface of the *DUT*. The stress injection block has two inputs: i) the timer parameters (i.e., timer period, timer routing, initial value and reload, further explained in Chapter 5); and ii) the stress injection target CPUs of the DUT. According to these variables, this block generates a set of values that will configure a set of registers of the central debug support interface. These values are transferred through the on-chip trigger and trace interface in the DUT. The alterations in the registers differ according to the architecture of the DUT. These values are known in this work as the stress injection instructions and are used to perform the stress injection in the DUT.

6.2.3 Trace Configuration

Figure 6.4: Trace Configuration Block.

The Trace configuration block is shown in the Figure 6.1 as the (3). This block is responsible for defining the setup of the trace that will be performed. The trace configuration is defined according to the input *Trace Parameters*. These parameters may differ according to the semiconductor vendor. However, usually the trace configuration is characterized by five main parameters: i) trace buffer (e.g., size, wide, flow control); ii) trace target; iii) timestamp type; iv) data trace (e.g., type of captured data); v) qualifier (e.g., range and type); vi) type of trace (e.g., continuous trace, trigger trace). The setting of the trace may differ according to the architecture of the DUT. The trace configuration is performed by sending the value of the selected parameters to the on-chip trace and debug solution of the DUT. The RTOS is scheduling the task set on the DUT. Based on the three outputs generated from the Stress Injection block and the trace instructions it is possible to apply the stress injection and to trace the DUT.

6.2.4 Device Under Test (DUT)

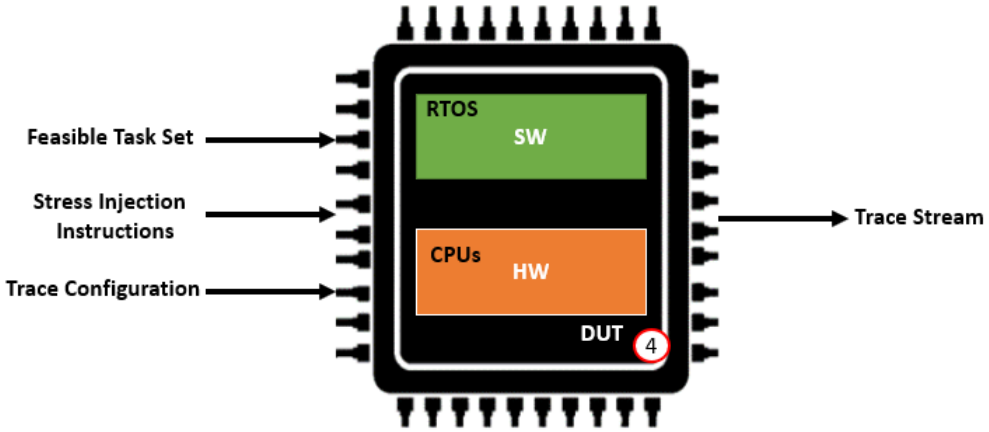

Figure 6.5: Device Under Test (DUT) block for the RTOS test methodology.

The DUT is indicated in the Figure 6.1 as (4). It is composed by a software (RTOS) and hardware components. In the DUT the task set is scheduled in the CPUs and the stress is injected. The DUT considered in this methodology is an embedded system with hard real-time constraints that is time-triggered with periodic task set. This block is compound by three inputs: i) the feasible task set generated by the TGU block; ii) the stress injection instructions, generated by the stress injection block; and iii) the trace configuration, generated by the *Trace Configuration* block. According to the value of these inputs, the task set is scheduled by the RTOS into the CPUs of the *DUT*. Then, the stress injection instructions are gathered by the on-chip trigger and trace interface. As a result, the registers of the central debug support interface are configured to inject the stress in the target CPUs. The output of the DUT is the Trace Stream, which is sent to the TCM through the on-chip trigger and trace interface. This trace extraction is performed simultaneously with the non-interrupted operation of the system.

6.2.5 Trace Capture and Map (TCM)

The TCM block is indicated in the Figure 6.1 as the (5). It captures the trace stream of the stressed task set scheduled by the *DUT*. After the trace configuration is applied and the stress injection to DUT is performed, the trace stream is captured. The stream of trace data is captured, filtered and mapped according to two characteristics. First, the task status addresses, that refers to the addresses of the task scheduled by the RTOS and which contain the status of each task of the task set. Second, the task status data values, which refer to the state values of each task that belongs to the task set allocated in the DUT by the RTOS. These states are described in detail in the Chapter 2. In general, there are four states:

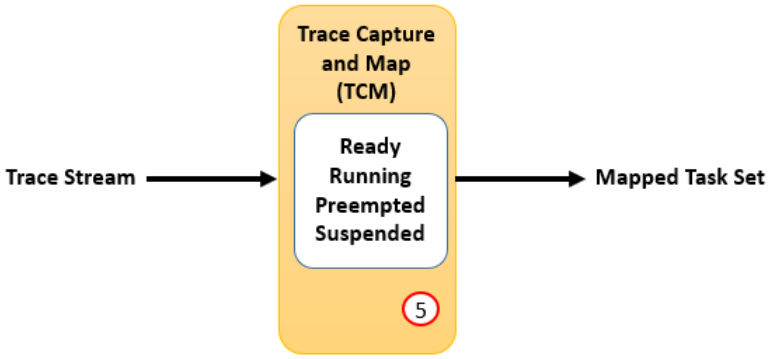

Figure 6.6: Trace Capture and Map (TCM) block.

Ready, Running, Waiting and Suspended. Afterward, the mapping contains the task identifier with the state and the respective timestamp (obtained during the trace for all the task set). The TCM block generates the mapped task set, that is sent to the metrics quantification block.

6.2.6 Metrics Quantification

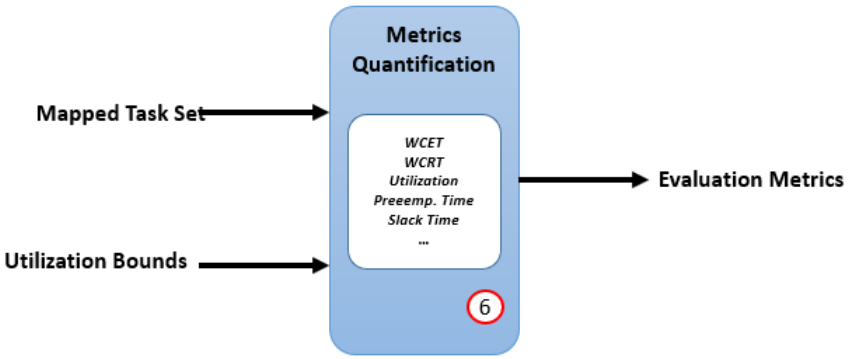

Figure 6.7: Metrics Quantification block.

The Metrics Quantification block, indicated in the Figure 6.1 as (6), quantifies a set of metrics for the performance and robustness evaluation of the DUT. This block has two inputs: i) the constructed map with the trace data of the task set provided by the capture TCM block; and ii) the feasible utilization bound, given by the TGU block. According to the mapped trace of the task set, the metrics quantification block generates all the timing parameter explained in the Chapter 2. The timing metrics, for instance, WCET, WCRT, Slack Time and others are calculated having the identifier of the task. This process is followed by the insertion of the utilization bounds for the stress injection provided by the TGU block. Therefore, the final

generated evaluation metrics will be centered in this range of utilization for the analysis of the correct data.

The output of this block is a set of evaluation metrics for the performance and robustness of the DUT. Note that the methodology can be extended and can further quantify several metrics defined by the designer. In this work, the two main metrics used for the stress injection analysis are i) the ratio between the WCRT and the minimum Slack time; and ii) the ratio between the WCRT and the Deadline. For further system analysis, two complementary metrics are used in this work, named Utilization and the Maximum Preemption Time. These metrics are the most common values used by designers. Note that further metrics can be quantified. These metrics are evaluated in terms of the Performance Degradation (*PD*) of the system utilization. The *PD* is expressed as a percentage value which represents the total utilization added to the system when the stress injection is performed by the technique of CPU suspension. The *PD* is calculated as the difference between the total utilization of the system when the stress injection U_S is performed and the initial total utilization without stress injection *U* as shown in the Equation 6.8:

$$PD = U_S - U \tag{6.8}$$

The selected metrics to analyze the system under stress injection were chosen for the meaningful description of the system behavior. In this subsection are introduced the main symptoms of the system (following the SoC health test analogy): the WCRT and the minimum Slack Time Ratio Symptom (SRS). These values indicate when the allocation of the task set is close to the infeasibility. Furthermore, the ratio between the WCRT and the Deadline is relevant in this work to find the so-called Infeasible Point (IFP). The remaining of the metrics helps to describe the system behavior before and after the system pass through the SRS and the IFP.

- **WCRT and Minimum Slack time Ratio Symptom (SRS):** This ratio combines the WCRT and the minimum Slack Time. The Slack time refers to the interval of time between the completion of the task and its deadline. This metric is used as a symptom of the system, once the ratio represents how close the task is to reach the deadline. Therefore, when the value of this metric is greater than one, the system falls into a *Danger Region*, where it is no longer feasible. The regions of operations are further described in the Subection 6.2.7.
- **Ratio of the WCRT and the Deadline:** This ratio shows how close the system is to reach the Infeasible Point (IFP). In such a region, the whole system becomes infeasible. When the value of this ratio is greater than one, the task set becomes infeasible and therefore the system can no longer operate correctly.
- **WCET:** This metric represents the maximum interval of time spent by the task actively using the CPU resources. As the stress injection adds load to the system, performing

the stress injection in the system will affect the WCET. This metric helps the designer to evaluate the system behavior under different load conditions.

- **WCRT:** This metric represents the maximum interval of time between the activation and termination of a task. Using this metric, it is possible to characterize the system, where the maximum interference is considered. By using this metric combined with other metrics, it is possible to define a risk symptom of the system such as the SRS or a specific point where the task set of the system is no longer feasible.
- **Utilization:** This metric represents the CPU utilization of a task. By using this metric it is possible to evaluate the operation of the system and to identify hot spots. The utilization is a key value to identify the symptom (SRS) and to determine when the Infeasible Point (IFP) is reached.
- **Maximum Preemption Time:** This metric represents the maximum preemption time of a task. This metric shows the behavior of the task of the task set when the stress injection is performed.

6.2.7 Worst-Case Response Time Analysis

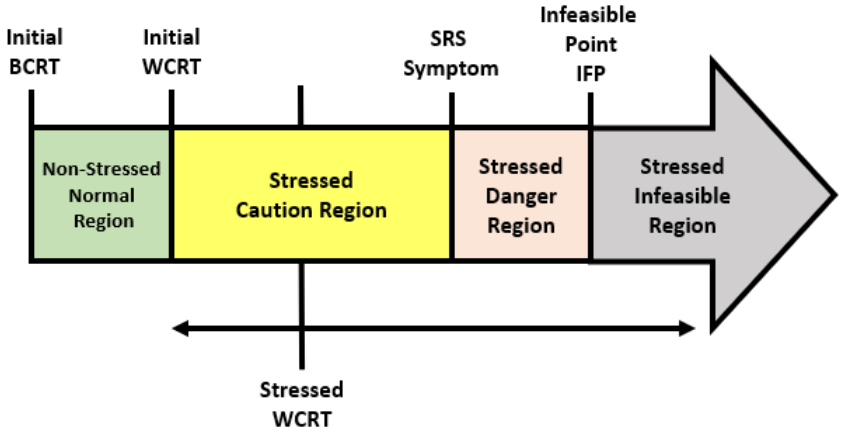

Figure 6.8: Response time regions of analysis.

A task-set is considered infeasible when it violates the temporal constraints (e.g., deadline). Therefore, the most intuitive time-based approach to identify a malfunction of the system uses the deadline as a reference point for identifying abnormal behavior. However, for hard real-time systems, the deadline miss is insufficient considering that awaiting until the task violates the deadline may be dangerous. The miss of the deadline could lead into catastrophic events. Therefore, is required to define a temporal boundary to delineate the off-normal behavior of the task set of the system, as shown in the Figure 6.8.

The WCRT metric is important to characterize the system. However, to base the analysis only in the response time metric does not provide a complete system information. The WCRT should be analyzed together with other metrics. The Slack time is defined as the interval of time between the termination of a task and its deadline. This interval of time can be used to understand which kind of task can be executed on the system without missing the deadline. When the interval of the response time is higher than the minimum slack time, the task is dangerously close to miss the deadline. Therefore, following the SoC health test analogy described in the Chapter 5, this metric can be used as a symptom of the malfunction of the system. The WCRT and Minimum Slack Time Ratio Symptom (SRS) represent the lower boundary of the so-called Stressed Danger Region (as shown in the Figure 6.8). The upper boundary of this region is the Infeasible Point (IFP), which refers to the point in which the task misses the deadline.

The defined temporal boundary that delineate the performance of the response time of a task when the stress injection is performed, it is shown in the Figure 6.8. It is composed by four regions:

1. **Non-Stressed Normal Region:** This region is bounded by the Best-case Response Time (BCRT) and the Worst-Case Response Time (WCRT) of the system without the stress injection. It represents the normal behavior of the system before the stress injection. When the system is stressed, the stressed WCRT passes to the Stressed Caution Region.
2. **Stressed Caution Region:** This region is bounded by the initial WCRT, before the stress injection, and the SRS.
3. **Stressed Danger Region:** This region is bounded by the SRS Symptom and the Infeasible Point (IFP). It represents a task that is dangerously close to violate the deadline. Thus, the task set may become infeasible.
4. **Stressed Infeasible Region:** Over the Infeasible Point (IFP) the system became infeasible and therefore, may cause possible catastrophic consequences for the user.

Following the analogy of the breath test to the Stress Injection, a non-invasive method to diagnose a possible disease is very important. It is desirable to identify the symptoms of the disease (infeasibility of the system) in an early stage. This may lead to apply less aggressive treatment methods to the patient (SoC). That is, instead of a total reboot of the system (aggressive treatment), a simple rescheduling of a task at design time may save the operation of the SoC. In addition, it may guarantee that the system continues operating in a stable state. Therefore, using early symptoms, such as the SRS , will lead in less aggressive correction methods and will maintain the feasibility of the hard real-time system.

6.3 Bare-metal Test Methodology

To characterize the system behavior when the stress injection is applied at the instruction-level, the stress is directly applied to a set of instructions in a bare-metal software application.

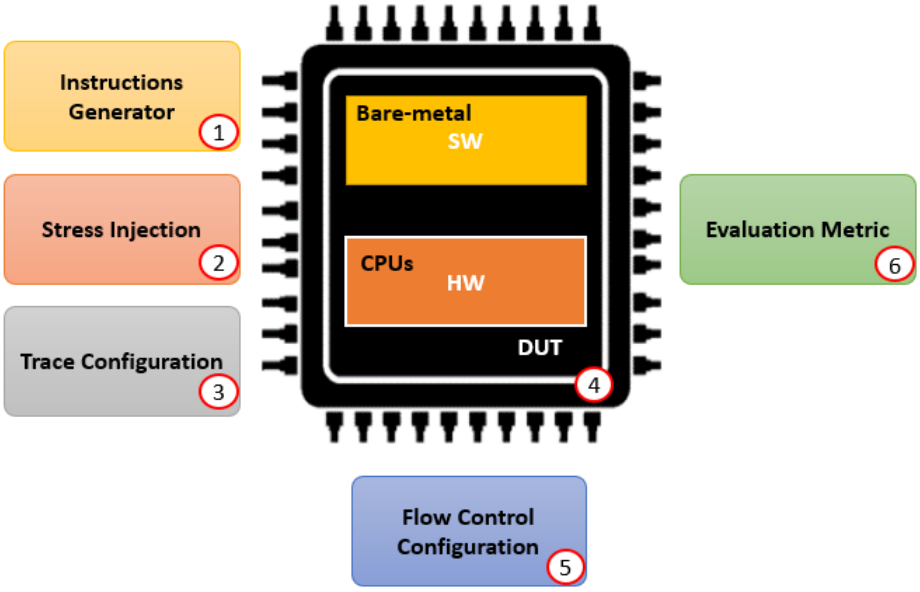

Figure 6.9: General description of the Bare-metal test methodology.

It is based on gathering the execution time for different types of instructions. This method allows the identification of possible dependencies of a certain type of instruction against the stress injection performed in the DUT. The high-level representation of the method adopted in this work for a Bare-metal test is presented in 6.9. It is composed by six blocks, indicated with an enumerated circle:

(1) **Instructions Generator:** This block generates the set of instructions that will be executed on the Bare-metal environment. (Subsection 6.3.1).

(2) **Stress Injection:** Generates the stress injection parameters for the debug interface (Subsection 6.2.2).

(3) **Trace Configuration:** This block configures the trace that will be performed in the DUT when the system is stressed (Subsection 6.2.3).

(4) **Design Under Test (DUT):** It is composed by a software and hardware components. (Subsection 6.3.2).

(5) **Flow Control Configuration:** This block modifies the flow control variable of the bare-metal system in order to perform the stress injection to the test set of instructions (Subsection 6.3.3).

(6) **Evaluation Metric:** Quantifies the execution time metric. It evaluates the instruction dependency to the stress injection in the DUT (Subsection 6.3.4).

Two of these six blocks used for the bare-metal test (Stress Injection and Trace Configuration blocks) are also used for the RTOS test and were previously described. Therefore, they are not described again. The following subsections further describe the remaining four blocks.

6.3.1 Instructions Generator:

Figure 6.10: Instructions Generator block.

The instructions Generator block, shown in Figure 6.9 as the (1). It generates the set of instructions that are executed on the bare-metal implementation. The *Instructions Generator* block has two inputs: i) amount of instructions; and ii) type of instructions. According to these variables, this block generates a set of equal type of instructions using the respective registers of the DUT. Afterward, the set of test instructions are implemented in the Bare-metal environment to apply the stress injection and to implement the trace configuration.

6.3.2 Device Under Test (DUT):

The DUT is indicated in the Figure 6.9 as the (4). It is composed by a software (bare-metal) and hardware components. This block requires four inputs: i) set of test instructions generated by the *Instructions Generator* block; ii) stress injection instructions generated by the *Stress Injection* block; iii) trace instructions generated by the *Trace Configuration* block; and iv) flow control instructions generated by the *Flow Control* block. The set of instructions is implemented in the bare-metal environment of the DUT and the stress injection is performed. Once the flow control instructions unblock the flow control variable, the set of instructions are executed. The execution time, limited to the set test instructions are measured using System Timer module as a reference clock for the measurement and the value is stored in a variable. Afterwards, the data stream generated by the DUT is sent to the Evaluation Metric block.

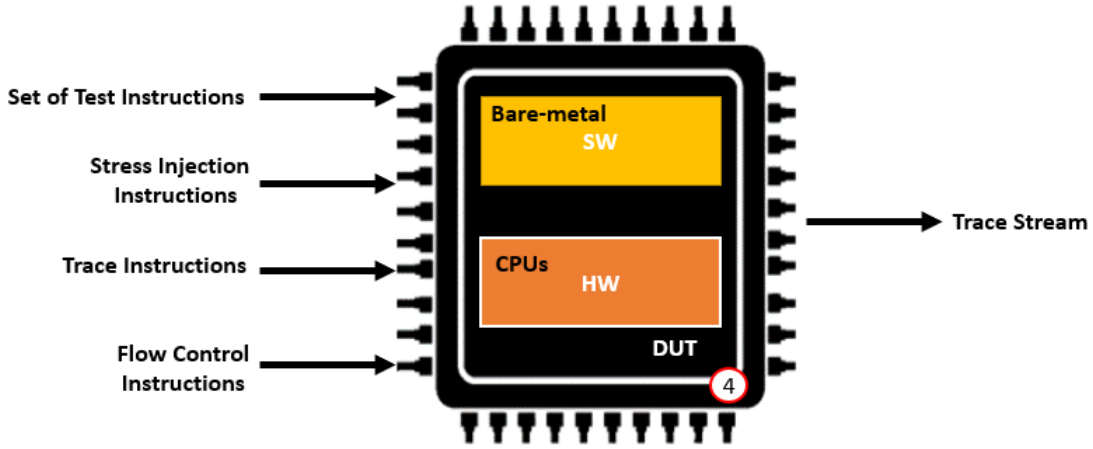

Figure 6.11: Design Under Test block.

6.3.3 Flow Control Configuration:

Figure 6.12: Flow Control Configuration block.

The *Flow Control Configuration* block is indicated in the Figure 6.9 as the (5). This block is responsible for modifying the value of the flow control variable implemented in the Bare-metal system. The *Flow Control Configuration* Block is composed by two inputs: i) ELF file, which contains binary and symbolic information; and ii) Unblock signal, which controls the flow control variable. The *Flow Control Configuration* block decodes the ELF file in order to acquire the binary and symbolic information of the flow control variable. Once the unblock signal is set, the *Flow Control Configuration* block generates the Flow Control Instructions that modifies the register that stores the flow control variable. This process ensures that the execution time gathered from the DUT belongs exclusively to the set of instructions generated by the Instructions Generator block and implemented in the DUT.

Figure 6.13: Evaluation Metric block.

6.3.4 Evaluation Metric:

The *Evaluation Metric* block, shown in the Figure 6.9 as the (6) is responsible for acquiring the execution time of the instructions. This block has two inputs: i) trace stream generated by the DUT; ii) ELF file of the system. According to these parameters, this block decodes the variable that contains the execution time of the instructions. This process is achieved by using the trace stream of the DUT and by linking the trace with the binary and symbolic information available in the ELF file. Afterward, the result is displayed to the software developer.

6.4 Summary

Block	Inputs	Outputs
Task Set Generation & Utilization Bounds (TGU)	-Total Amount of Tasks -Task Attributes -Total Amount of Runnables	-Feasible Task Set -Utilization Bounds
Stress Injection	-Timer Parameters -Target CPUs	-Stress Injection Instructions
Trace Configuration	-Trace Parameters	-Trace Instructions
Design Under Test (DUT)	-Feasible Task Set -Stress Injection Instructions -Trace Configuration	-Trace Stream
Trace Capture and Map (TCM)	-Trace Stream	-Mapped Task Set
Metrics Quantification	-Mapped Task Set -Utilization Bounds	-Evaluation Metrics

Table 6.1: Summary I/O of the RTOS test methodology blocks.

In this chapter was presented the methodology to apply the stress injection. First, the methodology of stress injection in an RTOS system was described. The method is summarized in the Table 6.1. In this system were implemented a feasible periodic task with implicit deadline and balanced distribution of the loads within the task set in order to perform a reliable analysis.

Block	Inputs	Outputs
Instructions Generator	-Amount of Instructions -Type of Instructions	-Set of Test Instructions
Stress Injection	-Timer Parameters -Target CPUs	-Stress Injection Instructions
Trace Configuration	-Trace Parameters	-Trace Instructions
Design Under Test (DUT)	-Set of Test Instructions -Stress Injection Instructions -Trace Instructions -Flow Control Instructions	-Trace Stream
Flow Control Configuration	-ELF File -Unblock Signal	-Flow Control Instructions
Evaluation Metric	-Trace Stream -ELF File	-Evaluation Metrics

Table 6.2: Summary I/O of the Bare-metal test methodology blocks.

Second, the methodology of stress injection in a Bare-metal system was described. It is summarized in the Table 6.2. The goal of this approach is to analyze the instruction dependency when the stress injection is performed. In this system were implemented a set of instructions on a Bare-metal environment to apply a stress injection on the DUT and obtain the execution time of the set of instructions. In order to acquire the exact execution time of the set of instructions when the stress injection is performed. This methodology uses a flow control mechanism where the data acquired from the DUT is ensured to belongs exclusively to the set of instructions implemented in the system.

7 Case Studies and Experiment Results

In this chapter, the experimental work is presented. It describes the experiments performed for the two types of software application: RTOS and Bare-metal. For the RTOS and the Bare-metal experiments, six case studies are designed, three for RTOS and four for Bare-metal. For each *Case Study* the performance and robustness evaluation results are presented, following the method proposed in Chapter 6. This chapter is divided into four sections. The First Section describes the general configuration and characteristics of the experimental work. The second section describes the RTOS experiments and results. The third section presents the Bare-metal experiments and results. Finally, a summary of the experimental results is shown.

7.1 General Description

The method presented in this thesis allows the SoC designer to evaluate the performance and robustness of an AURIX Infineon microcontroller with two types of software application, RTOS and Bare-metal and under different stress injection scenarios. For both applications the stress injection is performed as explained in Chapter 5 and using the methodology described in Chapter 6.

For the RTOS experiments, a feasible periodic task with implicit deadline and an even distribution of the load within the task set is implemented. These experiments are designed to evaluate the behavior of the system implemented within an RTOS application under critical performance and robustness cases for the AURIX microcontroller. The used RTOS is the ERIKA OS executed on the AURIX microcontroller. The case studies for the RTOS experiments implement three configurations of task sets, described and analyzed in detail in the Section 7.2.

For the Bare-metal experiments, a set of instructions in a Bare-metal environment is implemented. These experiments have the purpose to study the instruction dependency when the stress injection is performed. In each *Case Study* the stress injection is implemented within four different types of instructions, explained and analyzed in detail in the Section 7.3.

The stress injection is performed in the AURIX 2G TC39 using *Galenus*, which is executed in the ChipCoach framework and was programmed in Java. The full description of *Galenus* is described in Chapter 5. In order to guarantee the reliability of the initial conditions of the experiments, the setup of each experiment is verified using the Multicore Debug Solution Trace Viewer (MTV). Using *Galenus* the stress injection is executed, a set of metrics are quantified and the results are finally plotted for both cases (RTOS and Bare-metal). The most relevant plots for each experiment are shown in this work.

7.2 RTOS Experiments

The single-processor RTOS experiments use the RTOS ERIKA OS. It uses preemptive RMS as scheduling strategy. Each *Case Study* implements a feasible task set of six periodic tasks with implicit deadline ($D = T$) and an even distribution of the loads on the task set. Following the methodology of stress injection described in the Chapter 6, six parameter values must be defined:

1. **Total Amount of Tasks:** For all the case studies the total number of tasks implemented in the RTOS is six.
2. **Task Attributes:** For each *Case Study* the task attributes are modified as shown in Table 7.1
3. **Total Runnable per Task:** For each *Case Study* the total number of runnables per task is modified as shown in Table 7.1
4. **Timer Parameters:** For all case studies the fourth timer parameters are modified in the same way: i) *Timer Value* varies in the range from 1 to 125, which in clock cycles represents a variation from 12 to 1500 CPU cycles.This means, when the *Timer Value* takes the value of 1, it represents that every 12 CPU clock cycles it is performed the CPU suspension. In case of a value 125, the stress injection is performed every 1500 CPU cycles. Therefore at value of 1, the stress injection is maximum. The range between 12 to 1500 CPU cycles, was selected after detecting that from 1500 CPU cycles on, it is possible to detect a performance degradation over 1%; ii) *Reload Timer Value* is fixed to one. Therefore, the Timer Value is reloaded when the countdown timer reaches zero; iii) *Trigger Line Value* is fixed to one. Therefore, the trigger line is set to one when the Timer Value reaches zero. Thus, activating the suspension to all the sensitive suspension targets, and iv) *Timer to Trigger Line Value* is fixed to one. Therefore, the broadcast capabilities of the *Trigger Line 1* can be used to communicate the suspension signal to all the sensitive targets.
5. **Target CPUs:** The target CPU in which the stress injection is performed is fixed for all the case studies as *CPU0*.

6. **Trace Parameters:** For all the case studies the six parameters are modified similarly: i) *Trace Buffer Parameters* are fixed to 16 kB buffer size, with FIFO (First-in-First-out) IFTG (If-Full-Trace Gap) as a recording type and a continuous trace of the DUT; ii) *Trace Target OB* is fixed to the CPU0; iii) *Timestamp Type* is fixed to trace the events based on the DUT ticks; iv) *Data Trace Unit (DTU) Parameters* are set for the CPU0 with an in-range qualifier and to capture the address and data of all the write operations; v) *Qualifier Parameters* records the start and the end address of the qualifier. The ERIKA OS stores the state of each task in an array variable called *EE_th_status*. Therefore, the qualifier is set in the range of this variable for the implemented task set; and vi) *Continuous Trace* is set to five seconds of a continuous trace.

Case	Total Utilization (%)	Load Balance (%)	Period (ms) / Runnables (#)	Harmonic Task Set	Independent Tasks
1	70	11.6	1 / 4 5 / 1 10 / 1 20 / 4 50 / 3 100 / 1	No	Yes
2	70	11.6	1 / 4 5 / 1 10 / 1 20 / 4 60 / 3 120 / 1	Yes	Yes
3	70	11.6	1 / 4 5 / 1 10 / 1 20 / 4 50 / 3 100 / 1	No	No

Table 7.1: Summary RTOS experiments.

Table 7.1 shows the task set parameters implemented in the RTOS for the three case studies. The wider variety of case studies is advantageous in the analysis of the system. It increases the coverage of different load scenarios and thus favors the reliability and performance evaluation of the SoC. In the following subsections, each *Case Study* is described with the respective distribution of the loads within the task set. The results are presented and analyzed with the respective plots. Moreover, the evaluation metrics quantified by *Galenus* are evaluated in terms of the *Performance Degradation (PD)* of the total utilization of the system, as explained in the Chapter 6. The *PD* is a percentage value that represents

the utilization added to the system when the stress injection is applied. The stress injection is increased for each study case. The stress injection is inversely proportional to the Timer Value. That is, when the Timer Value gets smaller, a higher CPU suspension is performed and therefore, the stress injection gets larger.

7.2.1 Case Study 1: Synchronous task set with Total Utilization of 70% and non-harmonic periods

The first *Case Study* of stress injection on an RTOS software application is designed to start the stress injection with the worst independent task set attributes in terms of schedulability, using the *Rate Monotonic Scheduling (RMS)*. This behavior occurs when the task set has non-harmonic periods, synchronous task set and the total utilization is at the bound of the sufficient condition for the utilization of the system. The task periods are considered harmonic when each task period is an exact integer multiple of the next short period. In a synchronous task set, the offset of the first release of each task is equal to zero. In an independent task set, the task does not depend on the completion of requests of other tasks.

The designed task set is composed of six independent tasks with non-harmonic periods, as shown in Table 7.1 for this *Case Study*. Where the 50 msTask disturbs the harmonic behavior of the remaining tasks. With the non-harmonic relation among the task set periods using *RMS*, the schedulability bound downgrades.

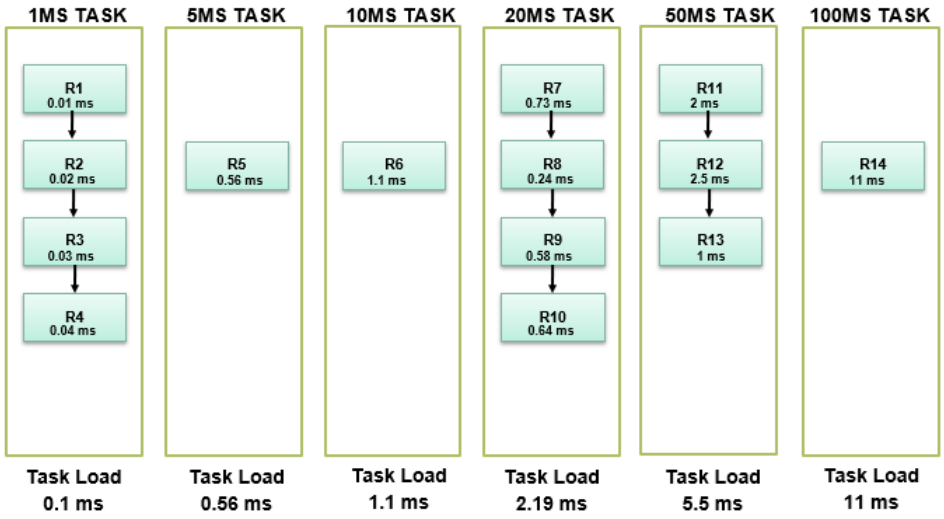

Figure 7.1: Case-Study 1 Task Set: Six synchronous and independent tasks with non-harmonic periods and a total utilization of 70%.

The designed task set is synchronous. In addition, the bound of the sufficient condition of the utilization of the system for the six tasks at RMS is set at 70% of CPU utilization. The utilization percentage was calculated based on the Utilization-Based Analysis presented in Equation 6.4. It provides the lower bound of the utilization for the RTOS experiments with a task set of six tasks. Once the lower bound is quantified, the load per task is evenly distributed on the task set considering the periodicity of each task quantified based on the Equation 6.5. The load per task is evenly distributed between the number of runnables per task, as shown in Figure 7.1.

The upper bound of the total utilization of the system is quantified based on the Response Time Analysis (*RTA*). It provides an exact test using the recursive Equation 6.7. The upper bound is equal to the necessary condition for achieving the 100% of utilization. Therefore, it guarantees that the system remains schedulable even when applying a stress injection that forces the 100% of the total utilization. This kind of load is very similar to the real automotive domain scenario, where the utilization of the system is very close to 100% and the SoC should guarantee the schedulability of the task set.

The experiment is executed according to the methodology described in this thesis. It can be summarized as follows. First, the task set is implemented in the RTOS and then the stress injection is performed. The tracing information is gathered and filtered in order to quantify a set of metrics through *Galenus*. Results are plotted using MATLAB.

For this *Case Study*, six graphs are presented and analyzed in terms of the *Performance Degradation*:

i) CPU Utilization graphs, it is identified the operation point on which the injected degree of stress makes that the system losses the even distribution of the load for each task; ii) WCRT and Minimum Slack Time Ratio symptom graph, which is used to identify the SRS symptom. iii) Maximum Preemption Time graph, which shows the preemption time for each task; iv) WCET graph, which shows the effect on the core execution when the synthetic load is added by the stress injection test; and vi) WCRT graph, which shows the maximum time that a task takes since its activation until its completion when different degrees of stress injection are applied. With this graph, it is identified the Infeasible Point (IFP).

■ **CPU Utilization:**

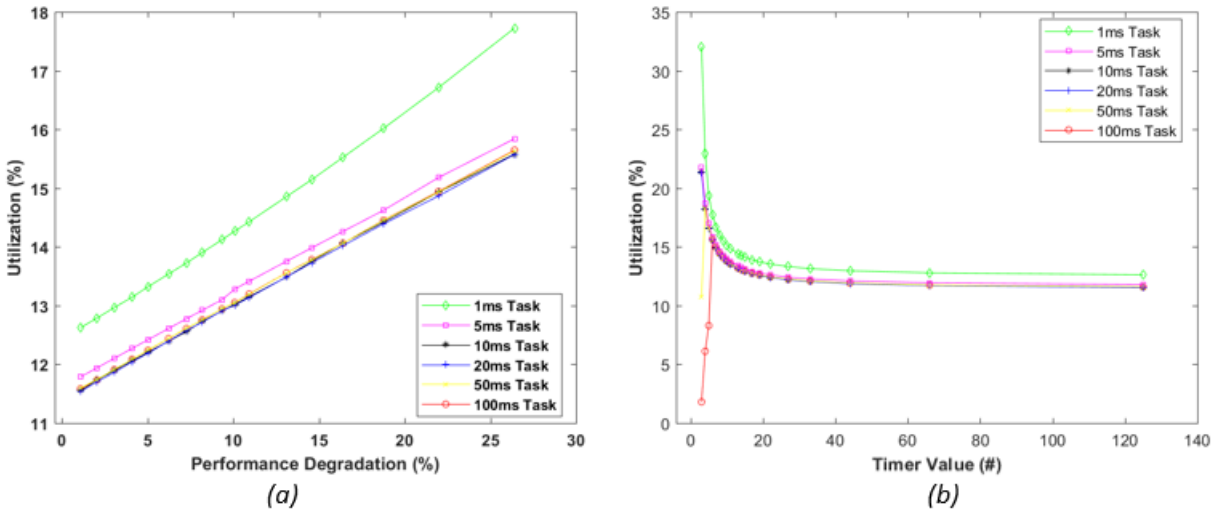

Figure 7.2: Case-Study 1: CPU utilization when the stress injection is performed.

Figure 7.2 shows the behavior of CPU utilization of each task executed on the DUT with different degrees of stress injection. Figure 7.2 (a) on terms of *Performance Degradation* and Figure 7.2 (b) on terms of the *Timer Value*. Figure 7.2 (a) shows how the CPU utilization increases directly proportional to the amount of stress injected on the DUT. This behavior is maintained until the *Performance Degradation* reaches the 26%.

Considering that the initial total utilization of the system is 70% and that the *Performance Degradation* is defined as a percentage of the total utilization added to the system when the stress injection is applied. Therefore at 26% of *Performance Degradation* the total utilization of the system is at 96%.

To have a clear view after the 26% of *Performance Degradation* Figure 7.2 (b) at the x-axis the *Timer Value* shows the resource starvation of the higher priority task i.e. 1 ms Task, showed with color green starve the CPU time for the lower priority task i.e. 100 ms Task, showed in Figure with the color red. While the lower priority decreases the CPU utilization from 15.5% to 8.3% while the other tasks increase their CPU utilization. The higher priority task increased from 17.7% to 19.3%.

The starvation of CPU resources from the higher priority over the lower priority and the total utilization of the system at that moment 96% of utilization leads in a miss of a deadline of the lower priority task. Therefore, the system becomes infeasible above 96% of utilization. This will be shown in the following evaluation metric.

■ **WCRT and Minimum Slack time Ratio Symptom (SRS):**

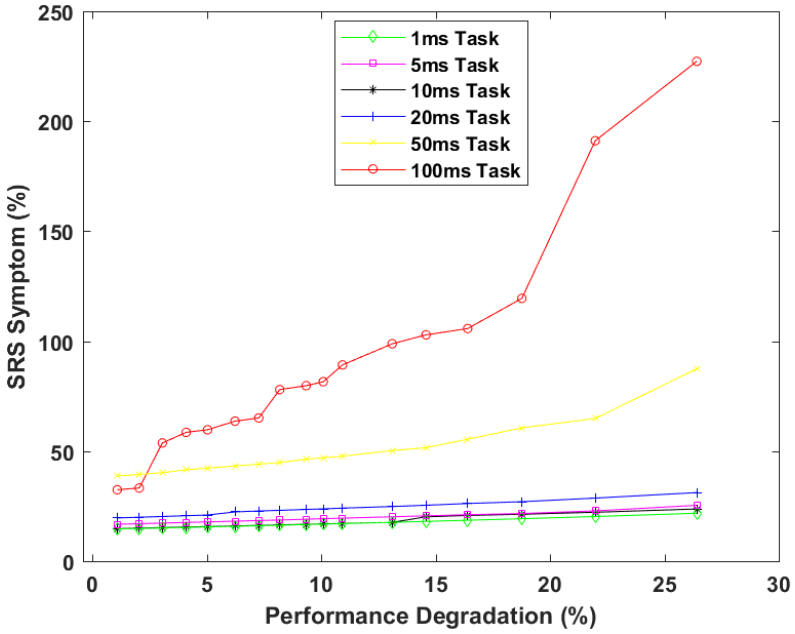

Figure 7.3: Case 1: WCRT and Minimum Slack time Ratio Symptom (SRS) when the stress injection is performed.

Figure 7.3 shows the percentage ratio between WCRT and the minimum Slack time. This ratio represents how close is the task to the Infeasible Point. Using the SoC health analogy, this ratio is a symptom of the system that is detected using the non-invasive stress injection test to help in this case the software developer to diagnose the system before it became infeasible. In this work the WCRT and Minimum Slack time Ratio symptom is called SRS, once the percentage is over 100% it is considered as a symptom of infeasible system. Therefore, performing a stress injection with a larger amount of CPU suspension over the SRS symptom is found will lead soon in the infeasibility of the system.

Figure 7.3 shows on the one hand how the four tasks with high priority in the system i.e. 1 msTask, 5 msTask, 10 msTask, 20 msTask, are in the range of 15% to 25% of the SRS during the stress injection. This means that these four tasks are still distant to reach the 100% of the SRS symptom and be considered as a symptom of infeasible system. On the other hand, the remaining tasks i.e. 50 msTask and 100 msTask present a larger degree of increase of SRS overall the task with lower priority showed in Figure 7.3 with the color red the 100 msTask. This task surpasses the 100% of SRS when the system is at 14% of performance degradation ergo the system is at 84% of total utilization. This task reaching the 100% of SRS, the system is now in the Danger Region defined in the Chapter 6 as the region between the SRS Symptom and the

Infeasible Point (IFP). The system became infeasible after the 26% of performance degradation and the SRS is detected at 14% of performance degradation. Therefore, applying the stress injection on the system helps the software developer to identify these thresholds of operation of the RTOS application on the DUT and diagnose it before it became infeasible. Considering the time constraints of the hard real-time systems, detect only when the system is infeasible is insufficient, therefore performing the stress injection on the DUT will lead in the detection of the SRS before the system reaches the Infeasible Point.

- **Maximum Preemption Time:**

  Figure 7.4 shows the maximum Preemption Time for each task of the task set when the stress injection is performed on the system and therefore the performance of the DUT is degraded. As shown in Figure 7.4 the two tasks with higher priority are never preempted i.e. 1 msTask and 5 msTask. The following three tasks on priority i.e. 10 msTask, 20 msTask, and 50 msTask present a linear increase, directly proportional to the performance degradation. However, for the task with lower priority on the system i.e. 100 msTask presents a non-linear behavior. This occurs due to the addition of the synthetic load added to the system when the stress injection is performed triggering the increase in the preemption of the other tasks over the lower task.

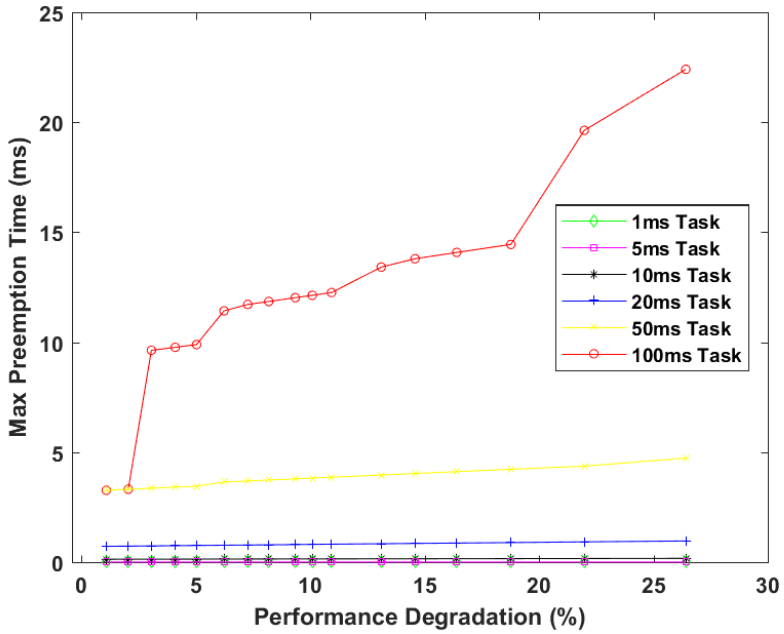

Figure 7.4: Case 1: Maximum Preemption Time when the stress injection is performed.

The many leaps on the behavior of the maximum preemption time on the lower priority task are a consequence of the non-harmonicity of the task set. Therefore, the interval of time in which the lower task is preempted is not regular due to the preemption of

the non-harmonic 50 msTask that preempts the lower priority task. This will be verified on the *Case Study 2*, that is performed with a harmonic task set. As a consequence of the behavior of the preemption time on the task set, this is reflected in the WCRT of the tasks and is shown in the WCRT evaluation metric analysis.

- **WCET:**

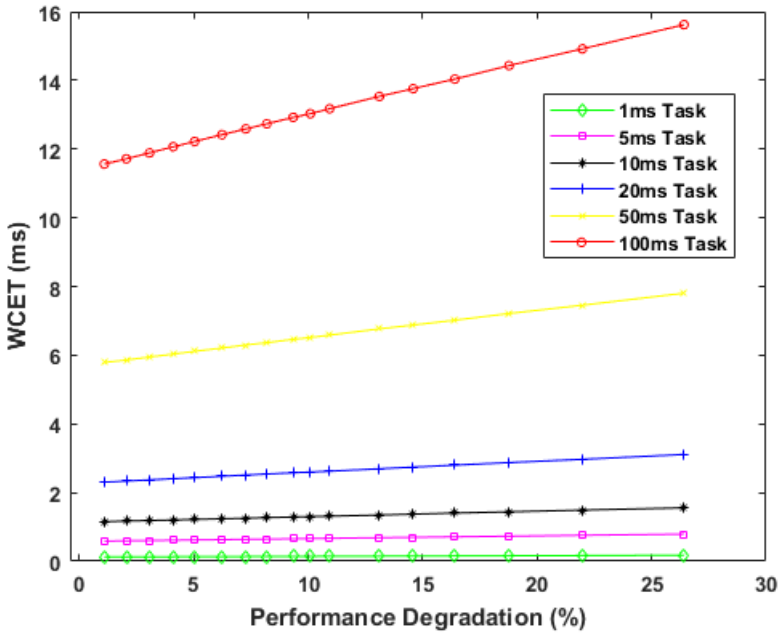

Figure 7.5: Case-Study 1:WCET when the stress injection is applied.

Figure 7.5 shows the WCET for each task when the degree of stress injection is increased. The WCET refers to the maximum consumed time of the core execution of a task. In Figure 7.5 all tasks behave linearly, the WCET is directly proportional to stress injection. The stress injection adds to the system a synthetic load by the CPU suspension on the DUT, therefore increasing the number of suspension on the system will lead to the increased time of the core execution of the tasks as shown in Figure 7.5.

- **WCRT:**

Figure 7.6 shows the WCRT when stress injection is performed on each task of the system. The WCRT represents the maximum time it takes for a task since it is activated until it is completed. When the stress injection is performed, the synthetic load is added to the system affecting the core execution as well the preemption time as analyzed previously. The minimum value of WCRT will be used for the response time analysis showed in Figure 7.7.

Figure 7.6 shows that as the synthetic load increases, it degrades the performance

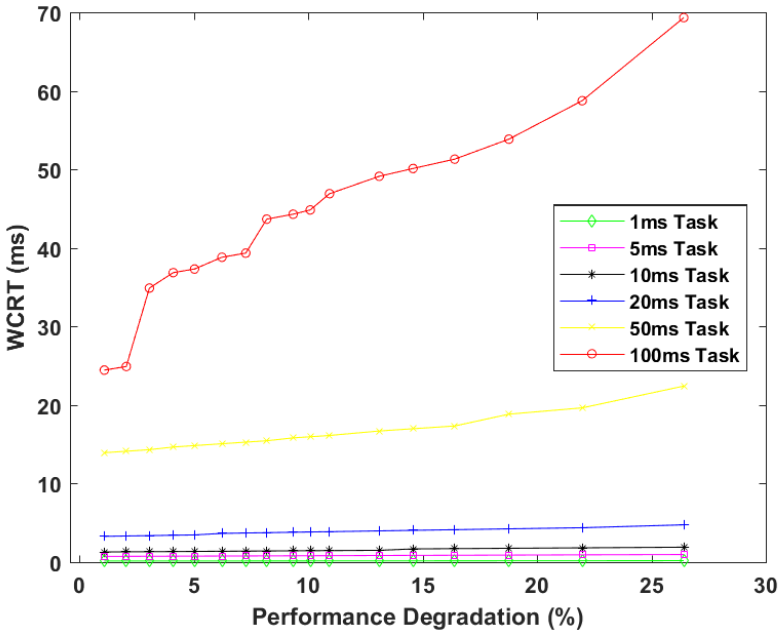

Figure 7.6: Case-Study 1: WCRT when the stress injection is performed.

of the system on a linear behavior for all the tasks except for the lower priority task i.e. 100 msTask. The non-linear behavior is due to the preemption delay that all the five tasks add to the lower priority task when the stress injection is performed. This behavior continues until reaching 26% of performance degradation, as analyzed in the CPU Utilization figure. The higher priority task starves the resources of the lower priority task. The 100 ms Task with less CPU time and higher WCRT will lead in a miss of a deadline after the 26% of performance degradation.

At the point of 26% of performance degradation, the 100 msTask reaches the maximum feasible WCRT, used in this work as the Infeasible Point (IFP). After this point, the system became infeasible, therefore the analysis of the evaluation metrics of this *Case Study* will be done until the IFP.

According to the temporal boundary regions defined in the Chapter 6 to delineate the off-normal behavior of the system. This is defined in four regions showed in Figure 7.7, using the evaluation metrics acquired by *Galenus* when the stress injection is applied. The software developer can use the response time analysis regions to analyze the performance and robustness of the DUT. Figure 7.7 shows the analysis for the lower priority on the system i.e. 100 msTask.

Before performing the stress injection, the *Non-Stressed Normal Region* showed with green color is bounded by the best and the worst-case response time for the 100 msTask between the 24 ms and 24.1 ms. When the stress injection is applied, the WCRT is on the *Stressed Caution Region* bounded by the initial WCRT and the SRS symptom. As

analyzed before, the SRS is detected when the system is at 14% of *Performance Degradation* (PD) with a WCRT of 50.11 ms. Once the SRS is detected, the WCRT pass to the *Stressed Danger Region* that is bounded between the SRS and the Infeasible Point (IFP).

Finally, the detected IFP were found at 26% of performance degradation of the system after this point the complete system is at *Stressed Infeasible Region*. Performing the stress injection was possible to degrade the performance of the system, as shown in the previous analysis for the 100 msTask, the WCRT was degraded from 24.1 ms to 109.5 ms.

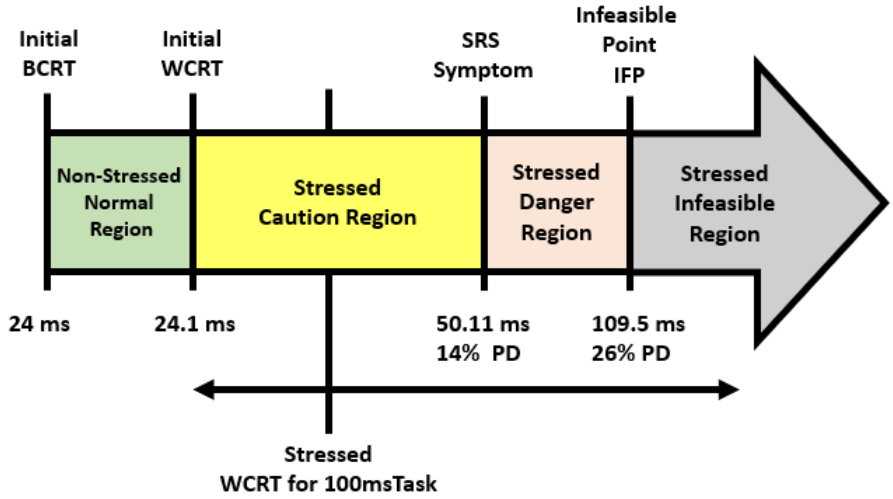

Figure 7.7: Case-Study 1: 100 msTask stress injection response time region analysis.

7.2.2 Case Study 2: Synchronous task set with Total Utilization of 70% and Harmonic periods

This second *Case Study*, differs from the previous one, according to the implementations of a task set with harmonic periods. The task set of this *Case Study* is compound by six independent tasks with harmonic periods as shown in Figure 7.8 where the 50 msTask and 100 msTask of the previous study case were replaced by the 60 msTask and 120 msTask.

This case study was designed in order to study the stress injection on an RTOS application with a task set with harmonic periods. As the previous case, the task set is synchronous and the initial total utilization of the system is 70% with a balanced distribution of the load within the task set as shown in Figure 7.8 where the number of the runnables per task are maintained. The analyzed utilization bounds for the stress injection remains equal to the previous case. For this *Case Study*, four graphs will be introduced and analyzed: i) CPU Utilization. ii) WCRT and Minimum Slack time Ratio Symptom (SRS). iii) Maximum Preemption Time and iv) WCRT.

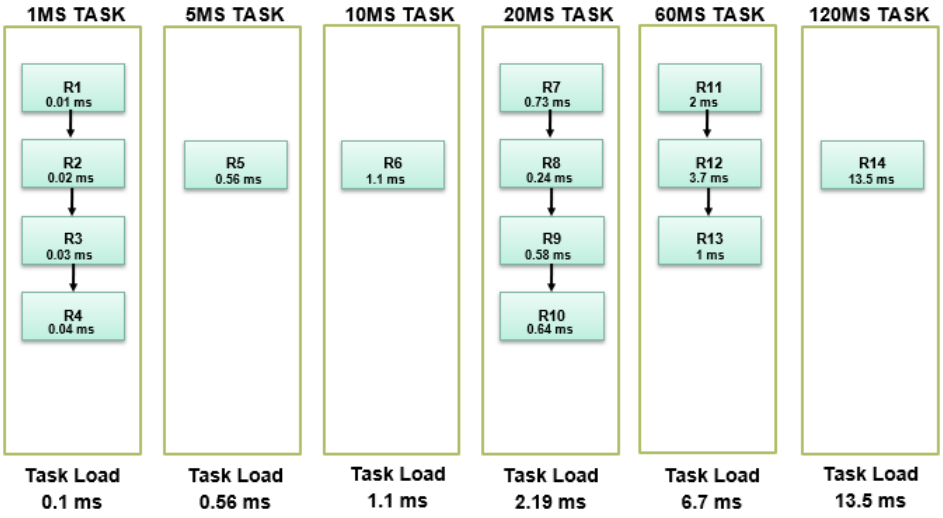

Figure 7.8: Case-Study 2 Task Set: Six synchronous and independent tasks with harmonic periods and a total utilization of 70%.

■ **CPU Utilization:**

Figure 7.9 shows the behavior of the CPU utilization for each task with harmonic periods when the stress injection is performed on the DUT. We can observe, compared with the CPU utilization with non-harmonic periods, in this case, the CPU Utilization for all the task set, the load distribution is sustained during the performance degradation. It is sustained, starting the stress injection until the 26% of performance degradation, where the task with higher priority starve the CPU time of the lower priority and the

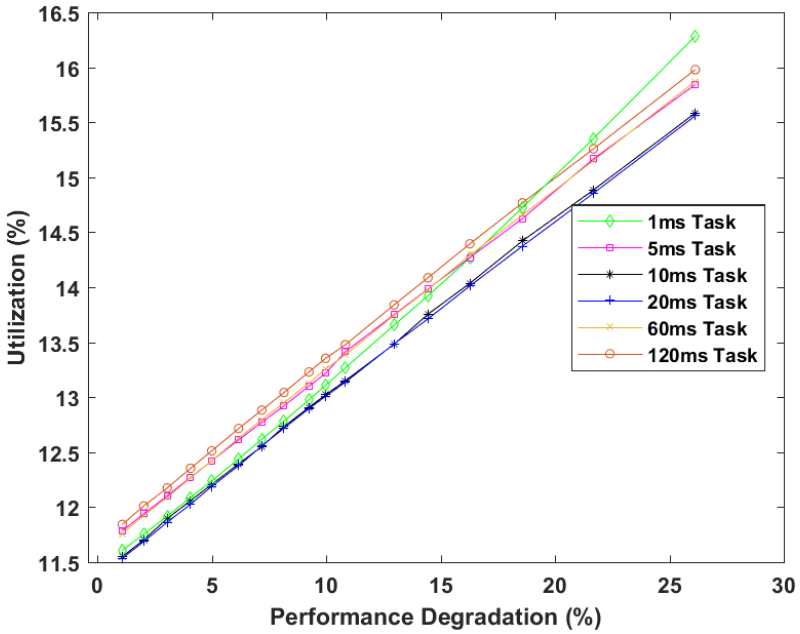

Figure 7.9: Case-Study 2: CPU utilization when the stress injection is performed.

system became infeasible. For the non-harmonic case, on the same type of graph, the task with higher priority has a higher utilization even when the load is distributed evenly. However, for the *Case Study 2* with harmonic periods, the CPU utilization for all the task set start close. This behavior remains during almost all the performance degradation. Until the system is close to became infeasible at 26%.

■ **WCRT and Minimum Slack time Ratio Symptom (SRS):**

Figure 7.10 shows the percentage of the SRS for the task set with harmonic periods. It shows that the symptom in this *Case Study* is detected only until reaching the 18% of performance degradation rather than 14% compared with the case with non-harmonic periods. This difference is due to the period of the lower priority task i.e., 120 ms. The SRS remains nearly stable for the complete task set until the 12% of performance degradation. Only after the 12% of performance degradation, the system starts to present symptoms that the system is starting to be pushed to the boundaries of the performance.

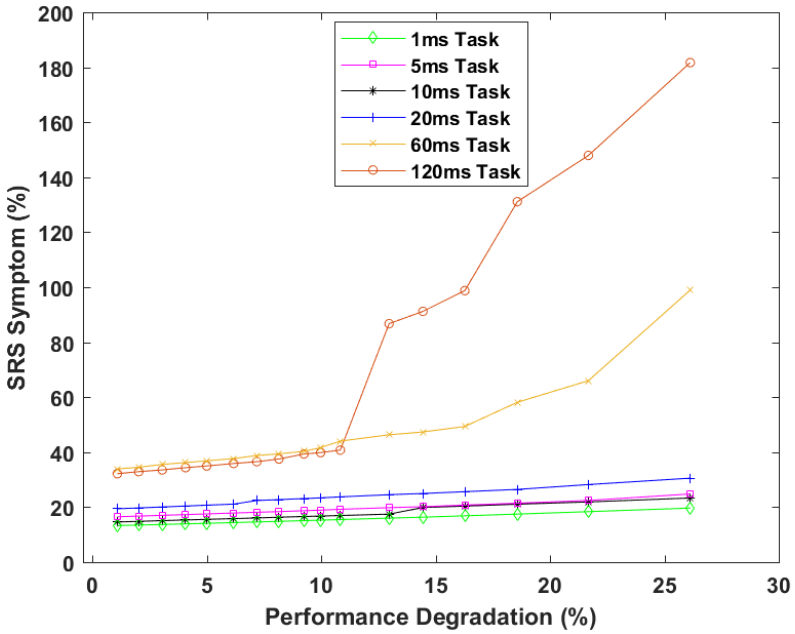

Figure 7.10: Case-Study 2: WCRT and Minimum Slack time Ratio Symptom (SRS) when the stress injection is performed.

■ **Maximum Preemption Time:**

Figure 7.11 shows the maximum Preemption Time for each task of the task set when the stress injection is applied to the system. As shown in Figure 7.4 on the *Case Study 1* with non-harmonic periods and in Figure 7.11 on *Case Study 2* with harmonic periods, the two tasks with higher priority are never preempted i.e., 1 msTask and 5 msTask.

The following tasks on priority i.e., 10 msTask, 20 msTask, 50 msTask and 120 msTask present a linear increase, directly proportional to the performance degradation. This behavior remains during all the performance degradation except for the lower priority i.e. 120 msTask at 12%. At this point, this task shows a big leap in the preemption time. This occurs due to the addition of the synthetic load added to the system when the stress injection is performed, triggering the increase in the preemption of the other tasks over the lower task.

Comparing the maximum preemption, the Figure 7.4 for the *Case Study 1* and Figure 7.11 for the *Case Study 2*, it shows a clear different behavior for the lower priority task. On the one hand, for *Case Study 1*, presents two leaps, one at 2% of performance degradation and the other at 6%. On the other hand, for *Case Study 2*, present only one leap at 12% of performance degradation. This comparison between an harmonic and non-harmonic task set, shows how the period harmonicity of the task set, affect the performance of the system.

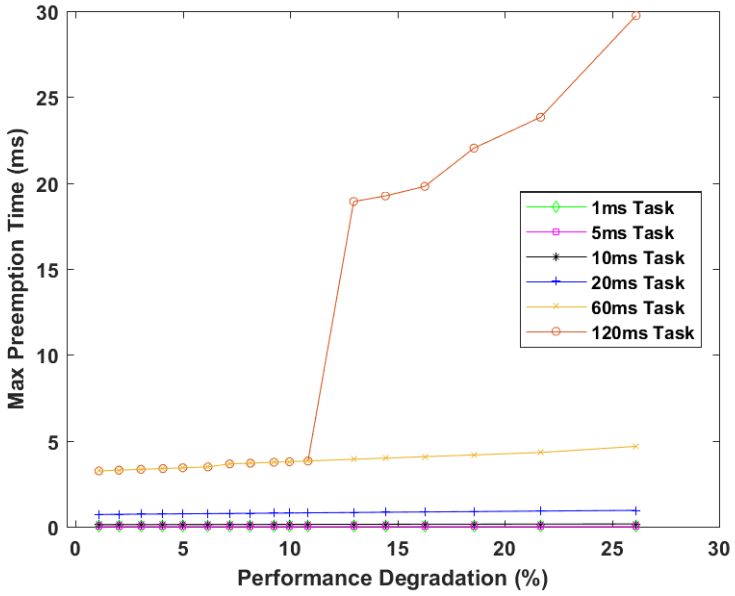

Figure 7.11: Case-Study 2: Maximum Preemption Time when the stress injection is applied.

■ **WCRT:**

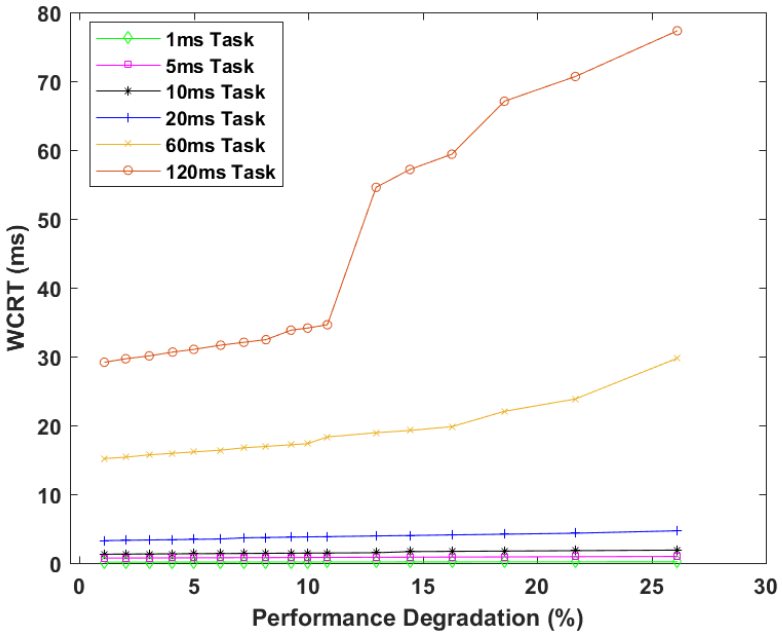

Figure 7.12: Case-Study 2: WCRT when the stress injection is performed.

Figure 7.12 shows the WCRT for each task of the task set when the stress injection is applied. Observing this figure, it is possible to identify the minimum WCRT and the Infeasible Point (IFP), metrics that we will use to perform the response time region analysis that we will observe at Figure 7.13.

On this configuration with harmonic periods, it is possible to observe how all the tasks behave linearly until the 12% of performance degradation. This behavior is due to the preemption time, as discussed in the previous item. For the *Case Study 1* with non-harmonic and this *Case Study 2* have the same behavior after the 26% of performance degradation. In both cases, the system became infeasible after this percentage of performance degradation. This behavior in both cases is due both systems are close to reach the 100% of total utilization, therefore less CPU time will be allocated in the lower priority task, leading in the infeasibility of the task set.

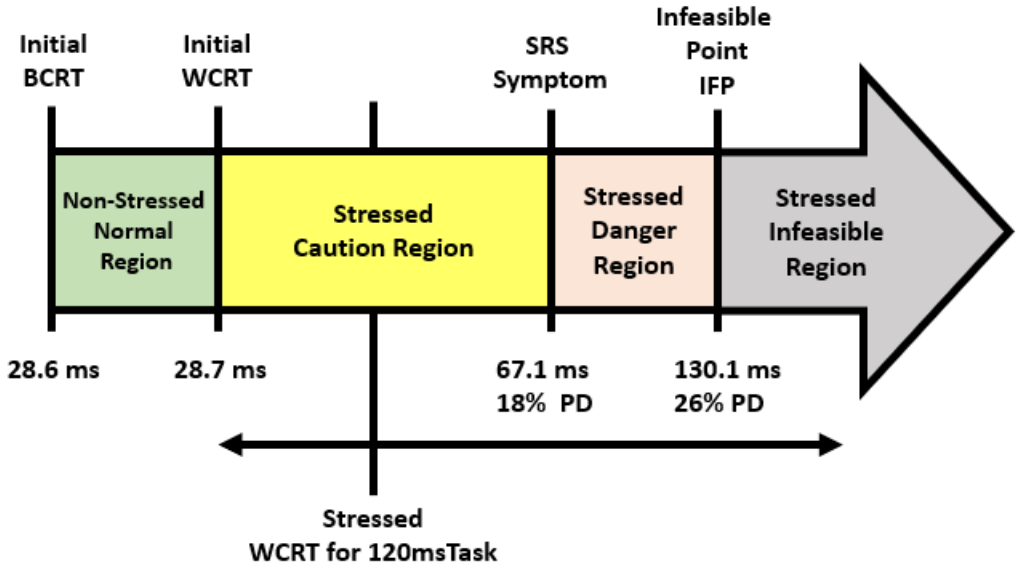

Figure 7.13: Case-Study 2: 120 msTask stress injection response time region analysis.

According to the temporal boundary regions defined in Chapter 6 to delineate the off-normal behavior of the system. Figure 7.13 shows the defined temporal boundaries regions for the analysis of the response time of the system when the DUT is under stress. Once the system is stressed performing an increased number of CPU suspension, the synthetic load added to the system increases the number of preemptions over the lower priority task.

Comparing the response time region analysis between the previous *Case Study 1* in Figure 7.7 and for the *Case Study 2* in Figure7.13. For *Case Study 2* with harmonic periods, it is until 18% of performance degradation when the SRS is activated with a WCRT of 67.1 ms and it is until the 26% of performance degradation that the system reaches the Infeasible Point with a WCRT of 130.1 ms. These two metrics form the

Stressed Danger Region of the range of 63 ms, however for *Case Study 1*, this region is smaller, i.e., 59.3 ms. For the *Stressed Caution Region*, *Case Study 2* has a range of 38.4 ms and for *Case Study 1*, this region has a range of 26.01 ms. For the *Case Study 2* this region is also larger. For the software engineer, it is beneficial to have larger regions because it implies that more stress of the system can be applied without causing infeasibility.

7.2.3 Case Study 3: Synchronous task set with Total Utilization of 70% and non-harmonic periods with Shared Resources

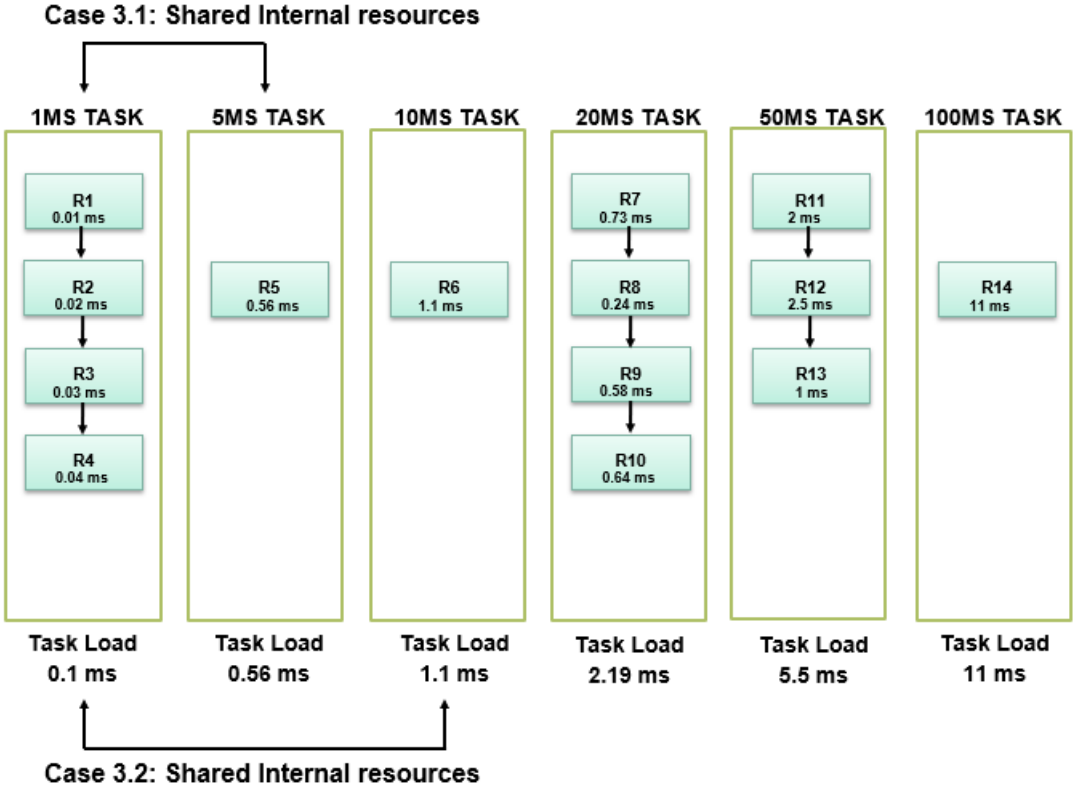

Figure 7.14: Case-Study 3 Task Set: Six synchronous and independent tasks with non-harmonic periods and a total utilization of 70%.

The *Case Study 3* is compound by six synchronous and independent tasks with non-harmonic periods. However, this third *Case Study* will add another degree of complexity by modifying the independent tasks to dependent tasks by sharing internal resources between two tasks of the task set. The sharing resources are performed between two tasks, to have a clear view of the behavior of the task set when this occurs.

For *Case Study 3*, two tests of shared resources between two tasks are performed between: i) 1 msTask and 5 msTask. ii) 1 msTask and 10 msTask. This *Case Study* was designed in this way in order to compare the results with the *Case Study 1*, due that case has the same task set characteristics except for the internal shared resources. The target of internal shared resources is to create groups of cooperative tasks to prevent concurrent access to shared resources.

The internal shared resources are the resources that are allocated for an instance of a task. Therefore, with this mechanism, all the tasks that use the same internal resource will behave as non-preemptive tasks. Considering the priority of the task using the resource, this is automatically changed to the ceiling priority of the resource. For this reason, in this case, the internal shared resources between the highest priority task and a task below the 10 msTask in terms of priority i.e. 20 msTask, 50 msTask and 100 msTask the system became infeasible even without performing the stress injection as sown in Figure 7.15. This figure shows the WCRT in which the 1 msTask and the 20 msTask are sharing internal resources. We can observe for the 1 msTask with the green line, starting the performance degradation, it is already infeasible. The response time is significantly larger than 1 ms. For this reason, the stress injection study is applied in two independents tests between i) 1 msTask, 5 msTask and ii) 1 msTask,10 msTask. These two tests are executed independently, however on the analysis of the evaluation metrics will be analyzed together to perform a clear comparison between both tests. The evaluation metrics evaluated in this *Case Study* are: i) CPU Utilization, ii) WCRT and Minimum Slack time Ratio Symptom (SRS), and iii) WCRT.

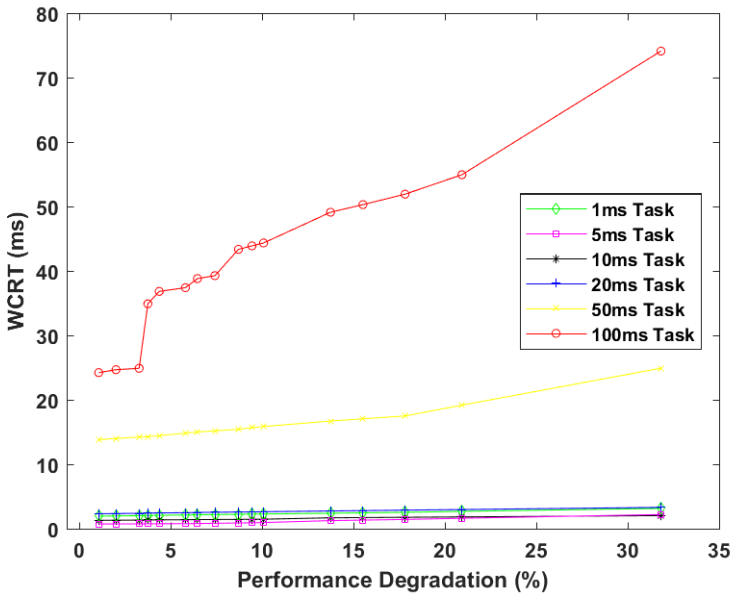

Figure 7.15: WCRT when the stress injection is applied, in case of 1 msTask and 20 msTask sharing internal resources, showing the infeasibility of the system

■ **CPU Utilization:**

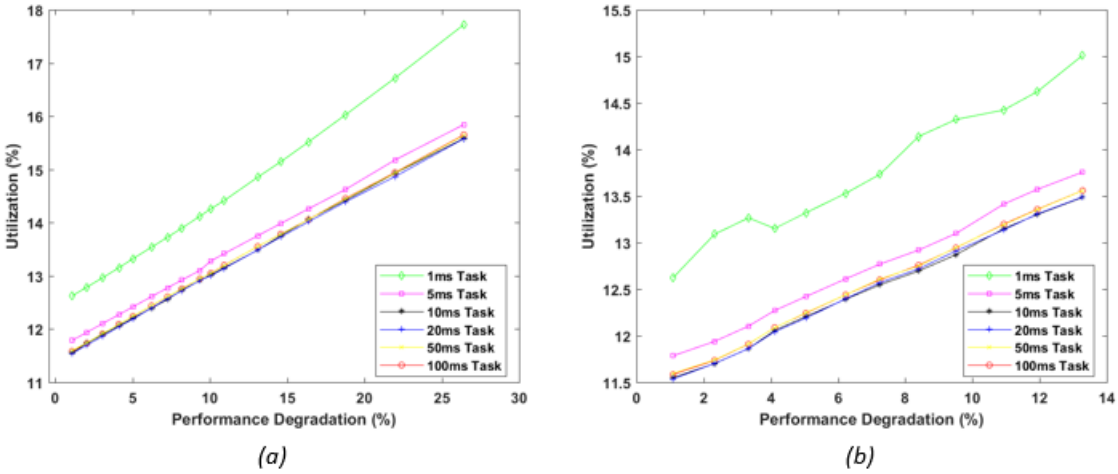

Figure 7.16: Case-Study 3: CPU utilization when the stress injection is applied. Shared internal resources: (a) 1 msTask and the 5 msTask. (b) 1 msTask and the 10 msTask.

Figure 7.16 shows the behavior of the CPU utilization for each test. On the one hand, Figure 7.16 (a) depicts the results for the 1 msTask and the 5 msTask On the other hand, Figure 7.16 (b) shows the results for the 1 msTask and the 10 msTask. Comparing the utilization of the system between the case of 5 msTask Figure 7.16 (a) with internal shared resources and the first *Case Study*, it is possible to observe, the behavior of both systems are different, most of all, in the behavior of the 1 msTask . This behavior is due to the internal resources are automatically taken when the task enters the running state.

In Figure 7.16 (b) when the 10 msTask is using the resources at running state i.e. critical section, 1 msTask it is waiting for the resources that the 10 msTask is using. Comparing with Figure 7.16 (a), this behavior is not observed due to the effects on the CPU utilization are distinct on the highest priority. This behavior is possible to observe on the 1 msTask on Figure 7.16 (b) with color green between 1% and 4% of performance degradation the nonlinear behavior, this due the 1 msTask had a larger blocking time by the 10 msTask.

■ **WCRT and Minimum Slack time Ratio Symptom (SRS):**

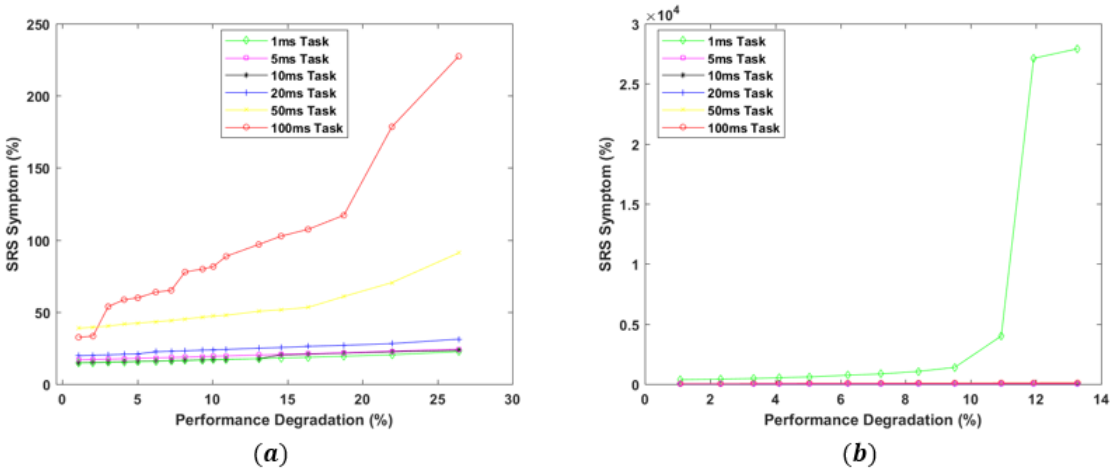

Figure 7.17: Case-Study 3: SRS Symptom when the stress injection is performed. Shared internal resources: (a) 1 msTask and the 5 msTask. (b) 1 msTask and the 10 msTask.

Figure 7.17 shows the percentage of the SRS symptom when the stress injection is performed. On the one hand, Figure 7.17 (a) shows the activation of the symptom at 14% of performance degradation and at 15.6% the system became infeasible. On the other hand for Figure 7.17 (b) the symptom is detected at the start of the test, as will be showed on the WCRT evaluation metric the 1 msTask starts the test close to the Infeasible Point, considering the blocking time of the 10 msTask over the highest priority time.

■ **WCRT**

Figure 7.18 shows the WCRT for the internal shared resources case when the stress injection is applied. Figure 7.18 (a) shows that the internal shared resources between the 5 msTask and the highest priority task produce no impact on the performance on the system if we compare with the same graph for *Case Study 1*.

For the case between 1 msTask and 10 msTask at Figure 7.18 (b). 1 msTask starts the stress injection close to become infeasible. This behavior is due to the blocking time of the lowest priority task, in this case, the 10 msTask is larger compared with 7.18 (a). Besides, the tasks with shared resources behave as non-preemptive, therefore the response time of the highest priority i.e., 1 msTask is close to the deadline starting the stress injection. As a consequence, the system became infeasible at 13% of performance degradation. Rather than 26 % for the 7.18 (a).

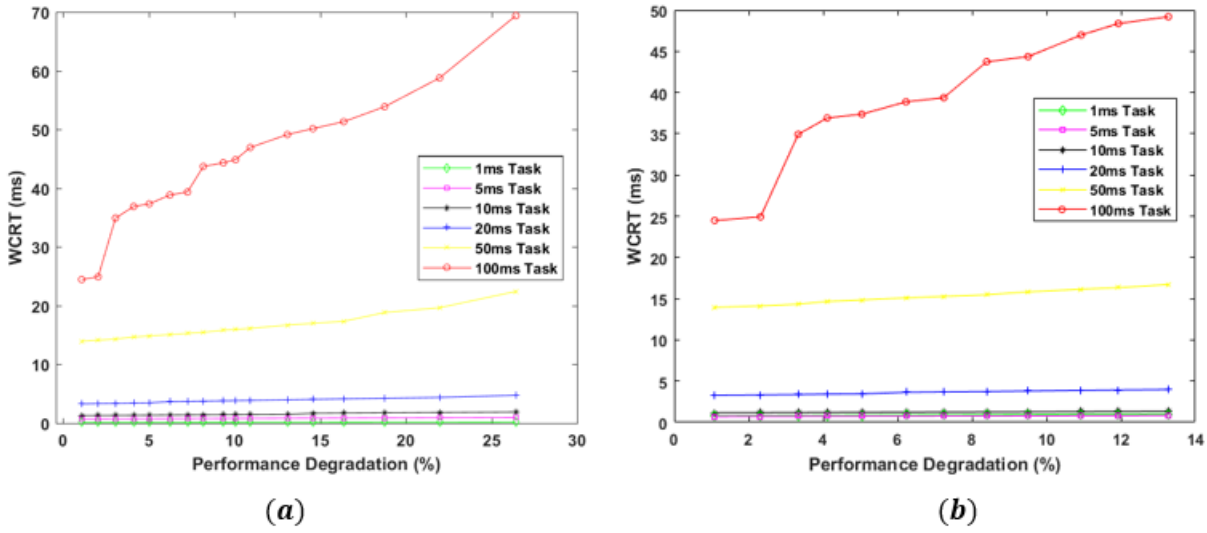

Figure 7.18: Case-Study 3: WCRT when the stress injection is performed. Shared internal resources: (a) 1 msTask and the 5 msTask. (b) 1 msTask and the 10 msTask.

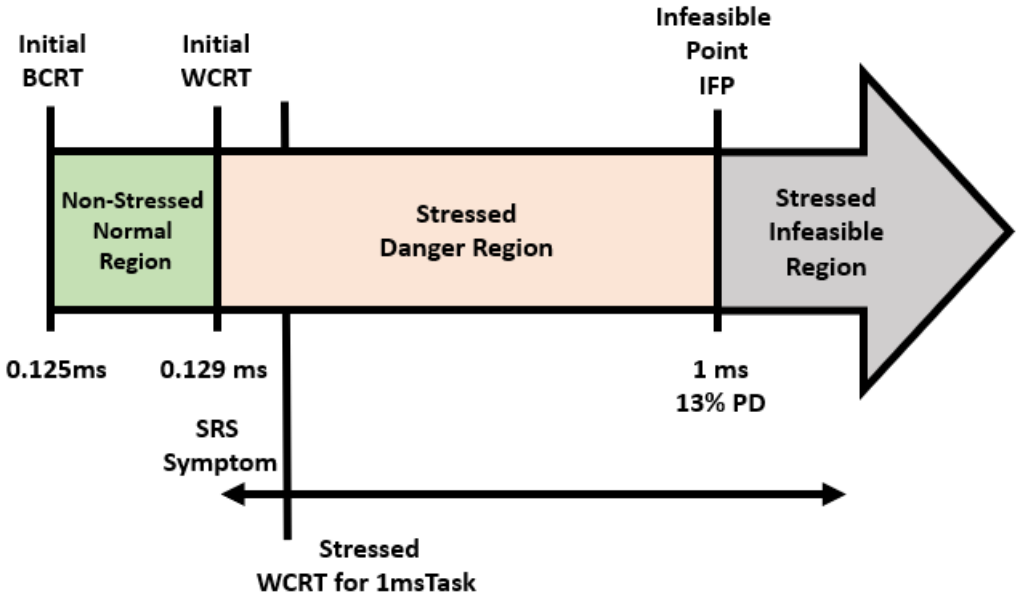

Figure 7.19: Case-Study 3: 1 msTask stress injection response time region analysis for the case of internal shared resources between 1 msTask and 10 msTask.

According to the temporal boundary regions defined in Chapter 6 to delineate the off-normal behavior of the system. The analysis is applied for the 1 msTask in the case of sharing internal resources between the 1 msTask and 10 msTask and shown in Figure

7.19. This case was selected due to the behavior of the WCRT. Before the stress injection, the *Non-Stressed Normal Region* showed with green color is bounded by the best and the worst-case response time for the 1 msTask between the 0.125 ms and 0.129 ms. In this case, the system is already pushed on the edge of the performance, therefore starting the stress injection, the system starts at the *Stressed Danger Region* until the 13% of performance degradation at 1 ms, for the 1 msTask the system turns infeasible.

7.3 Bare-Metal Experiments

The purpose of the Bare-metal experiments is to analyze at instruction level the effects of the stress injection. This is achieved by acquiring the time required to execute a predefined number of instructions on the system when the stress injection is applied. The Bare-metal experiments are implemented on a single-processor of the AURIX microcontroller. These experiments consist on the execution of an instruction of the same type on the Bare-Metal environment several times. Using *Galenus* and the System Timer Module (STM) it is possible to acquire the time of execution of these instructions on the system while the stress injection is performed. Following the methodology for a Bare-metal test of stress injection described in the Chapter 6, six parameter values must be defined:

1. **Total Amount of Instructions:** For all the Bare-metal experiments the total number of instructions executed on the DUT are 2704.
2. **Type of Instructions:** The type of instructions executed on the Bare-metal environment were: i) Arithmetic Instructions. ii) Logic Instructions. iii) Shift Instructions. iv) Data Transfer Instructions. v) Branching Instructions.
3. **ELF File:** Binary information of the software application provided by the software designer in order to obtain the address of the flow control variable and the execution time variable.
4. **Timer Parameters:** For all the Bare-metal experiments, the four timer parameters are modified in the same way: i) *Timer Value* varies in the range from 1 to 125, which in clock cycles represents a variation from 12 to 1500 CPU cycles.
5. **Target CPUs:** The target CPU in which the stress injection is performed on all the Bare-metal experiments is the *CPU0*.
6. **Trace Parameters:** For all the Bare-metal experiments the six parameters are fixed: i) *Trace Buffer Parameters* are fixed to 16 kB buffer size with an On-Chip Trace Buffer Mode as Full, therefore the trace buffer is filled with trace data until it is full. ii) *Trace Target OB* is fixed to the CPU0; iii) *Timestamp Type* is fixed to trace the events based

on the DUT ticks; iv) *Data Trace Unit (DTU) Parameters* are set for the CPU0 with the qualifiers enabled to capture the address and data of all the write and read operations; v) *Program Trace Unit (PTU) Parameters* are set for the CPU0 with an instruction level of trace detail. vi) *Qualifier Parameters* the qualifier is set in the range of all the system considering is a Bare-metal application. and vii) *Mode of Trace* is set to On-chip Trace Buffer Mode.

The Bare-metal experiments are divided into four case studies, according to the type of instruction executed on the DUT as shown on the Table 7.2. Each instruction is executed 2704 times while the system is stressed. Afterwards, the execution time is acquired by *Galenus* and then the degree of stress injection is increased. This process is repeated until the complete range of stress injection is applied, this range is defined by the software designer on the Timer Parameters. Once the complete range of stress is applied, this process is repeated for the next instruction.

Case	Type of Instruction	Instruction	Syntax
1	Arithmetic	-ADD Addition No Saturation -SUB Substract -MUL Multiply Signed -DIV.F Divide Float	-ADD D[a], D[b] -SUB D[a], D[b] -MUL D[a], D[b] -DIV.F D[a], D[b], D[c]
2	Logic	-OR Bitwise OR -XOR Bitwise XOR -AND Bitwise AND	-OR D[a], D[b] -XOR D[a], D[b] -AND D[a], D[b]
3	Branching	-CALL-RET Call-Return from Call -J Jump Unconditional	-CALL foobar; RET -J foobar
4	Data Transfer	-MOV Move	-MOV D[a], D[b]

Table 7.2: Summary Bare-metal experiments.

For example, for the first arithmetic instruction i.e. ADD instruction. The timer value is configured from 12 to 1500 CPU cycles. Therefore, the stress injection is performed every 1500 CPU cycles while the ADD instructions are executed 2704 times, then *Galenus* acquires the execution time and decreases the timer value for the CPU suspension to 1499 CPU cycles. This process is repeated until reaching the 12 CPU cycles of the timer value. This is repeated for each instruction on each case of study.

Once this process is performed for all the instructions of each *Case Study*, the metrics acquired by *Galenus* are plotted using MATLAB. These results are shown in a logarithmic graph, the y-axis is the normalized value of the Cycles Per Instruction. The x-axis is the Stress Injection Factor (SIF). SIF is an integer value from 1 to 125 which refers to the clock cycles of the timer used to perform the suspension of the CPU. Therefore, when SIF takes the value of 1, it represents that every 12 CPU clock cycles it is performed the CPU suspension. In case SIF takes the value 125, the stress injection is performed every 1500 CPU cycles. Therefore when SIF takes the value of 1, the stress injection is maximum.

The analysis is performed jointly for the four case studies to have a more clear understanding of the behavior of the instructions against the stress injection. In the Table 7.3, shows the execution time per instruction set and the CPI without stress and with the maximum stress for each type of instruction. Additionally, this table shows the Ratio of Stress, which represents the ratio of the CPI with maximum stress and the CPI without stress.

Instruction	Execution Time (ns)	CPI	Execution Time Maximum Stress (ns)	CPI Maximum Stress	Ratio of Stress
ADD	9050	1.014	31040	3.479	3.43
SUB	9050	1.014	31040	3.479	3.43
MUL	18160	2.035	54410	6.098	3.00
DIV	54450	6.102	108500	12.159	1.99
OR	8990	1.007	31210	3.498	3.47
XOR	13500	1.513	36400	4.079	2.70
AND	8990	1.007	31210	3.498	3.47
CALL-RET	117080	13.121	216730	24.288	1.85
JUMP	27190	3.047	72480	8.123	2.67
MOVE	13400	1.502	36400	4.079	2.72

Table 7.3: Execution time, CPI and Ratio of Stress for the four study cases with stress-free and maximum stress.

With the data acquired we can observe that the instructions that are executed in a shorter time are the instructions that have a larger ratio of stress. This behavior is due to the characteristic of the stress injection by CPU suspension. In which the stress injection immediately halts new transactions and allows to complete the remaining pending transactions in order to perform the CPU suspension and this state is kept until the suspend signal is deasserted. Then, the CPU can resume the operation. Therefore, with maximum stress, the stress injection is performed every 12 CPU clock cycles and the instructions that are executed in a shorter time have larger stress injection effects.

In order to have a clear understanding of the stress injection at the instruction level, the CPI at stress was normalized as shown in Figure 7.20. This figure shows the six different behaviors that were identified performing the stress injection on the four cases studies. When the stress injection is applied, the SUB, OR, XOR and AND instructions have the same behavior of the ADD instruction, therefore they were omitted in Figure 7.20.

Decreasing the Stress Injection Factor (SIF) the frequency of CPU suspensions increases. Under a SIF of 66, meaning a stress injection every 792 CPU clock cycles, on one hand, the MOV instruction continues on a stable normalized CPI close to 1 until the SIF is equal to 4, this behavior is unexpected and was not further investigated but should. On the other hand, the behavior of the other five instructions remains increasing exponentially until SIF reaches 20 were the stress injection is performed every 240 CPU clock cycles. At this point, the DIV and CALL-RET instructions start to behave differently according to the ADD, MUL and

JUMP instructions. While the DIV instruction has a moderate degree of exponential decay, the CALL-RET instructions present two peaks between the range of 4 and 10 SIF.

Considering that the experiment was performed several times with the same setup, obtaining the same behavior for all the case studies. Hence the behavior for the CALL-RET instructions is due to the execution of Call instruction and Return instruction while in the other cases are only executed one type of instruction.

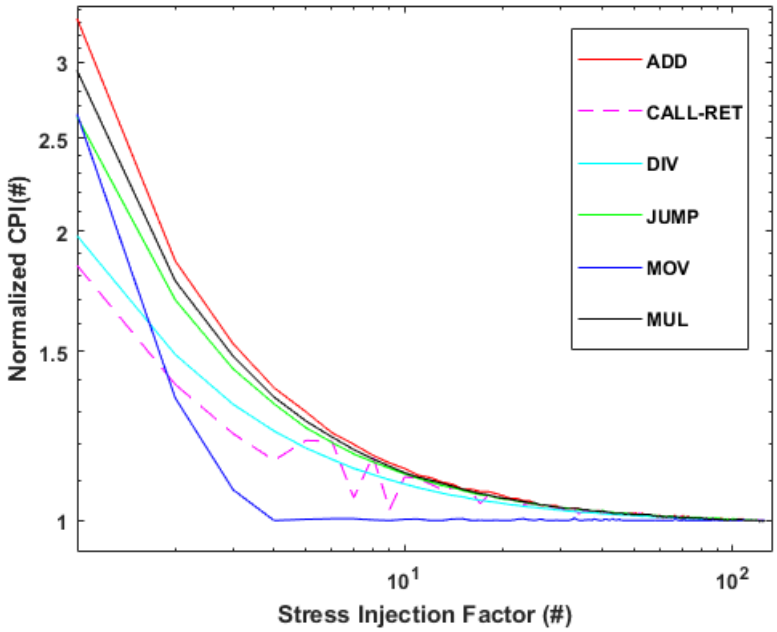

Figure 7.20: Normalized CPI vs Stress Injection Factor.

7.4 Summary

Experimental results of the stress injection by CPU suspension were acquired with *Galenus*, within two types of software application, RTOS and Bare-metal under different stress injection scenarios. For the RTOS experiments were able to evaluate the behavior of the system implemented within an RTOS application under critical performance. Performing the stress injection, the CPU utilization increases directly proportional to the amount of stress injected on the DUT. This is due to the increased core execution time when the CPU suspension is performed. Analyzing the evaluation metrics provided by *Galenus*, it was possible to identify the starvation of CPU resources from the higher priority task over the lower priority task. This leads in a miss of a deadline of the lower priority task becoming infeasible the system reaching the Infeasible Point (IFP).

On the RTOS case studies, the Minimum Slack time Ratio Symptom (SRS) was a clear representation of how close the task is to the IFP. The SRS parameter will help the software developer to identify these thresholds of operation of the RTOS application on the DUT. Using the Soc health analogy, the software developer as the doctor can perform the stress injection as the breath test, in order to early detect symptoms as the SRS in order to apply the less aggressive treatment to the patient as the DUT. Otherwise, a delayed or not identification of a symptom will lead in a more aggressive treatment for the patient. Therefore using early symptoms in the hard real-time such as the SRS Symptom will lead in a less aggressive repercussions in the system in order to maintain the feasibility of the hard real-time system.

For the Bare-metal experiments, the instructions that are executed in a shorter time are the instructions that have a larger ratio of stress. This response is due to the property of the stress injection by CPU suspension. In which the stress injection instantly halts new transactions and permits complete the remaining pending transactions in order to perform the CPU suspension and this state is kept until the suspend signal is deasserted. Then, the CPU can resume the operation until the next CPU suspension is asserted.

8 Conclusions and Future Work

The developed stress injection feature *Galenus* is a reliable and usable new feature on the internally developed tool at Infineon Technologies AG. *Galenus* is suitable to measure the performance and robustness of a hard real-time system. This goal is achieved based on the measurement of the performance and robustness factors such as the CPU utilization, WCET, WCRT and the feasibility of the system against the stress injection. *Galenus* is capable to implement the stress injection through CPU suspension within two types of software applications (RTOS and Bare-metal). The main functionalities of *Galenus* are the configuration of the stress injection and trace, sorting and mapping the trace stream and generating the evaluation metric of the performance and robustness. The evaluation metrics acquired by *Galenus* are crucial for the software designer in order to build adequate evidence to demonstrate that no safety risks are raised against potential CPU overloads.

The experimental results of the case studies of the stress injection by CPU suspension were performed on the AURIX 2G microcontroller following the methodology presented on this thesis, with two types of software application, RTOS and Bare-metal under different stress injection scenarios. On the RTOS experiments, were performed an exploration of system behavior against the stress injection with feasible periodic tasks with implicit deadlines and a balanced distribution of the load within the task set of the RTOS. These experiments were possible to identify and validate the impact of the stress injection in the effective performance of a CPU by periodic suspensions of the CPU. Moreover, the robustness of the system was tested, studying the feasibility of the task set against the stress injection using *Galenus*. The RTOS experiments were able to identify thresholds of operation of the RTOS application on the DUT using the evaluation metrics provided by *Galenus* such as the Minimum Slack time Ratio Symptom (SRS) and the Infeasible Point (IFP).On the Bare-metal experiments was possible to investigate the instruction dependency in four different types of instruction sets against the stress injection.

The methodology presented in this thesis allows the SoC designer to evaluate the performance and robustness of an AURIX 2G under different stress injection scenarios. This methodology together with the Multicore Debug Solution (MCDS), the Infineon internal developed tool (Chipcoach) and the developed feature *Galenus* developed on this thesis. It allows the SoC designer to perform the resource usage and stress tests leading to the maximum exploitation of the resources of the system with the maximum safety of the user.

This work allowed the utilization for the first time of the hardware debugging architecture and thus opening a wide variety of future and interesting works. It includes: i) to perform stress injection on more complex and realistic systems; ii) the integration of statistical properties to model the different scenarios can also already be performed easily with the current *Galenus* configuration; iii) the utilization of AI techniques for further exploration and identification of the parameters of the stress injection; iv) the further identification of complex indicators in stressed systems to determine the stability of them; and v) the exploration of different stress injection types on the AURIX 2G microcontroller, including the artificial reads and the CPU interrupts.

Bibliography

[1] Inga Harris. Chapter 22 - embedded software for automotive applications. In Robert Oshana and Mark Kraeling, editors, *Software Engineering for Embedded Systems*, pages 767 – 816. Newnes, Oxford, 2013. ISBN 978-0-12-415917-4.

[2] ISO. Road vehicles – Functional safety, ISO 26262, 2011. URL `https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en`. Accessed: 2020-01-14.

[3] Paolo Gai, Enrico Bini, Marco Di Natale, and Luca Abeni. Architecture for a portable open source real time kernel environment. 07 2001.

[4] Peter Schiefer. Automotive conference call. URL `https://www.infineon.com/dgdl?fileId=5546d4615ee5d3d6015f02a0a266023f&redirId=57324`. Accessed: 2020-02-27.

[5] National Highway Traffic Safety Administration-NHTSA. Air bags, May 2019. URL `https://www.nhtsa.gov/equipment/air-bags`. Accessed: 2019-11-15.

[6] Giorgio C. Buttazzo. *Hard real-time computing systems predictable scheduling algorithms and applications*. Springer, 2011.

[7] Xiaocong Fan. Chapter 12 - software architectures for real-time embedded systems. In Xiaocong Fan, editor, *Real-Time Embedded Systems*, pages 303 – 338. Newnes, Oxford, 2015. ISBN 978-0-12-801507-0.

[8] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, Jun 1989. ISSN 1573-1383. doi: $10.1007/\mathrm{BF02341920}$. URL `https://doi.org/10.1007/BF02341920`.

[9] R. Long, H. Li, W. Peng, Y. Zhang, and M. Zhao. An approach to optimize intra-ecu communication based on mapping of autosar runnable entities. In *2009 International Conference on Embedded Software and Systems*, pages 138–143, May 2009. doi: $10.1109/\mathrm{ICESS.2009.63}$.

[10] M. Peraldi-Frati, A. Goknil, J. DeAntoni, and J. Nordlander. A timing model for specifying multi clock automotive systems: The timing augmented description language v2. In *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, pages 230–239, July 2012.

[11] S. Lauzac, R. Melhem, and D. Mosse. An efficient rms admission control and its application to multiprocessor scheduling. *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. doi: 10.1109/ipps.1998.669964.

[12] Phillip A. Laplante. *Real-time systems design and analysis*, pages 92–95. Wiley, 2004.

[13] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL https://doi-org.ezproxyegre.uniandes.edu.co:8843/10.1145/321738.321743.

[14] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 01 1986. ISSN 0010-4620. doi: 10.1093/comjnl/29.5.390. URL https://doi.org/10.1093/comjnl/29.5.390.

[15] Kristian Beckers. *Background*, pages 11–35. Springer International Publishing, Cham, 2015. ISBN 978-3-319-16664-3. doi: 10.1007/978-3-319-16664-3_2. URL https://doi.org/10.1007/978-3-319-16664-3_2.

[16] Mark Pitchford. Chapter 15 - embedded software quality, integration and testing techniques. In Robert Oshana and Mark Kraeling, editors, *Software Engineering for Embedded Systems*, pages 441 – 510. Newnes, Oxford, 2013. ISBN 978-0-12-415917-4.

[17] Georg Macher, Muesluem Atas, Eric Armengaud, and Christian Kreiner. Automotive real-time operating systems: A model-based configuration approach. *SIGBED Rev.*, 11(4):67–72, January 2015. doi: 10.1145/2724942.2724953. URL https://doi.org/10.1145/2724942.2724953.

[18] Freeosek. URL http://opensek.sourceforge.net/. Accessed: 2020-02-28.

[19] Automotive open system architecture - autosar. URL https://www.vector.com/int/en/know-how/technologies/autosar/. Accessed: 2020-02-28.

[20] Evidence SRL. Erika enterprise rtos v3, Sep 2019. URL https://www.erika-enterprise.com/. Accessed: 2020-02-28.

[21] Trampoline rtos, Feb 2020. URL https://github.com/TrampolineRTOS/trampoline. Accessed: 2020-02-28.

[22] Erika enterprise manual, 2012. URL http://erika.tuxfamily.org/drupal/documentation.html. Accessed: 2019-12-03.

[23] Infineon Technologies AG. 32-bit tricore aurix– tc3xx. URL https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/. Accessed: 2019-11-22.

[24] Ieee standard glossary of software engineering terminology. *IEEE Std 61012-1990*, page 1–84, Dec 1990. doi: 10.1109/ieeestd.1990.101064.

[25] S. M. A. Shah, D. Sundmark, B. Lindström, and S. F. Andler. Robustness testing of embedded software systems: An industrial interview study. *IEEE Access*, 4:1859–1871, 2016. ISSN 2169-3536. doi: 10.1109/ACCESS.2016.2544951.

[26] GLIWA GmbH. Timing suite-t1, . URL `https://www.gliwa.com/index.php?page=products_T1&lang=eng`. Accessed: 2019-12-14.

[27] INCHRON GmbH. Analyze and verify real-time capability in worst-case scenarios, . URL `https://www.inchron.com/tool-suite/chronval/`. Accessed: 2020-01-09.

[28] Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, pages 759–775, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33666-9.

[29] Lionel Briand, Yvan Labiche, and Marwa Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7:145–170, 06 2006. doi: 10.1007/s10710-006-9003-9.

[30] S. Di Alesio, S. Nejati, L. Briand, and A. Gotlieb. Stress testing of task deadlines: A constraint programming approach. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 158–167, Nov 2013. doi: 10.1109/ISSRE.2013.6698915.

[31] J. Béchennec, S. Faucou, O. H. Roux, M. Brun, and L. Givel. Testing real-time systems with runtime enforcement. *IEEE Design Test*, 35(4):31–37, Aug 2018. ISSN 2168-2364. doi: 10.1109/MDAT.2018.2791801.

[32] A. Mayer, H. Siebert, and K. D. McDonald-Maier. Boosting debugging support for complex systems on chip. *Computer*, 40(4):76–81, April 2007. doi: 10.1109/MC.2007.118.

[33] A. Mayer, H. Siebert, and C. Lipsky. Multi-core debug solution ip, soc software debugging and performance optimization. May 2007.

Glossary

ADAS Advanced Driver Assistance Systems.

AL Adaptation Logic.

API Application Programming Interface.

ASIL Automotive Safety Integrity Levels.

BCET Best-Case Execution time.

BCRT Best-case Response Time.

DAS Device Access Server.

DCU Debug Status and Control Trace Unit.

DLL Dynamic Link Libraries.

DMC Debug Memory Controller.

DMS Deadline Monotonic Scheduling.

DPS Dynamic Priority Scheduling.

DTU Data Trace Unit.

ECU Electronic Control Unit.

ED Emulation Device.

EDF Earliest Deadline First.

ELF Executable and Linkable Format.

FPS Fixed Priority Scheduling.

IFP Infeasible Point.

IoT Internet of Things.

MCDS Multicore Debug Solution.

MCX Multicore Cross Connect.

MSU Message Sequencer Unit.

MTV Multicore Debug Solution Trace Viewer.

OB Observation Block.

OCDS On-Chip Debug Support.

OTGS OCDS Trigger Switch.

OTU Ownership Trace Unit.

PD Performance Degradation.

PTU Program Trace Unit.

RL Reload Timer.

RMS Rate Monotonic Scheduling.

RTA Response Time Analysis.

RTOS Real-Time Operating System.

SIF Stress Injection Factor.

SoC System on Chip.

SRS Slack Time Ratio Symptom.

TGL Timer to Trigger Value.

TGU Task Set Generation and Utilization Bounds.

TLT Trigger Line Timer.

TMEM On-Chip Trace Memory.

TQU Trace Qualification Unit.

UBA Utilization-Based Analysis.

VTZ Trigger Line Value.

WCET Worst-Case Execution time.

WCRT Worst-Case Response time.

WTU Watchpoint Trace Unit.