INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

mecanismos baseados em reservas para clusters de instrumentação de criticalidade múltipla

JOEL OLIVEIRA PINTO

Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Reservation-based mechanisms for Mixed-Criticality Two-Wheeler Instrumentation Clusters

Joel Pinto

A dissertation submitted to obtain the degree of Master of Science, Specialisation Area of Computational Systems

Supervisor: Dr. Cláudio Maia

Porto, October 15, 2021

Dedicatory

To my family, girlfriend, friends and colleagues, for all the love, support, help and advises.

Abstract

Electronics completely transformed the automotive industry as early vehicles were purely composed by mechanical components but the current reality is quite different. The growing acceptance for embedded electronics devices led to a significant increase in the number of microcontroller-based functions embedded in vehicles. With this increase, customer's safety concerns raised.

To ensure customers safety from the use of Electrical and Electronic (E/E) automotive equipment and systematic failures, Original Equipment Manufacturers (OEMs) and their suppliers must comply with standards such as ISO 26262, the road vehicles functional safety standard. ISO 26262 provides regulations and recommendations for the product development process.

When the critical road functionalities are regarded as hard real-time, that shall complete within the defined time boundaries, coexist in an environment with soft and non real-time tasks (e.g., multimedia and connectivity activities) the system designer must use an approach to ensure that no critical activity is jeopardized in order to avoid hazardous events.

To cope with the coexistence of activities with different time boundaries and criticality within the same system, this work proposes the implementation of uniprocessor reservation-based mechanisms, namely the Constant Bandwidth Server (CBS) and the Capacity Sharing and Stealing (CSS), in a real-time operating system for scheduling non-critical activities without jeopardizing the *apriori* guarantee of critical activities. Both schedulers use the concept of server, a task holder where a fraction of the processor bandwidth is reserved for tasks, thus relaxing the need for knowing certain properties of the tasks such as the WCET. Both implementations are detailed and compared through the implementation of task sets where both types of tasks coexist.

Keywords: E/E automotive equipment, ISO 26262, Real-Time Systems, Reservation-based mechanisms

Resumo

A eletrónica transformou por completo a indústria automotiva, os primeiros veículos eram puramente compostos por componentes mecânicos, mas atualmente a realidade é significativamente diferente. O aumento da aceitação de dispositivos eletrónicos levou a um crescimento exponencial do número de funções baseadas em microcontroladores embutidos em veículos. E com este aumento, as preocupações relativas à segurança por parte dos clientes aumentaram.

Para garantir a segurança de falhas sistemáticas e de falhas provenientes do uso excessivo de componentes Elétricos e Eletrónicos (E/E) de um veículo, tanto os Original Equipment Manufacturers (OEMs) como os seus fornecedores tem que cumprir com standards como por exemplo o ISO 26262, standard referente à segurança funcional de veículos rodoviários. O ISO 26262 apresenta os regulamentos e recomendações presentes em todo o processo de desenvolvimento do produto.

Quando as funcionalidades críticas tambem são consideradas como hard real-time, que tem que dar resposta a estimulos externos dentro dos limites temporaris definidos, coexistem no mesmo ambiente com tarefas soft e non real-time (por exemplo, atividades de multimídia e conectividade), o system designer tem que usar abordagens especificas para continuar a garantir que nenhuma atividade hard seja comprometida, evitando assim possiveis consequencias catastróficas.

Para fazer face à coexistência de atividades com difrentes niveis de criticalidade e limitações temporais dentro do mesmo sistema, este trabalho propõe a implementação de mecanismos baseados em reservas de partes de utilização do processador, nomeadamente o Constant Bandwidth Server (CBS) e o Capacity Sharing and Stealing (CSS), num sistema operativo de tempo-real para escalonar atividades não críticas sem comprometer a garantia *apriori* de tarefas criticas. Ambos os escalonadores usam o conceito de servidores dedicados, onde uma fração da largura de banda do processador é reservada para tarefas, relaxando assim a necessidade de conhecer certas propriedades das tarefas, como o WCET. Ambas as implementações são detalhadas e comparadas através da implementação de um conjunto de testes onde os dois tipos de tarefas coexistem.

Keywords: E/E automotive equipment, ISO 26262, Real-Time Systems, Reservation-based mechanisms

Acknowledgement

One of hardest moments in my life, develop a project and dissertation of this magnitude on my free time. Long nights, lost weekends and the desire to give up were my routine throughout the three years that took me to conclude this master's degree.

I owe it to my parents Fernanda Pinho and Manuel Pinto, my sister Claudia Amaral, my aunt Estrela Oberson and my girlfriend Adriana Pinho for all the affection and help. I can not forget my friends and colleagues for all the advises and brainstorming sessions, in particular to Renato Oliveira. All the institutions that helped me throughout these years, ISEP, CISTER, Bosch, InnoWave and NOS, a short path full of adventures.

And at last but not least, to a colleague, mentor and friend, Cláudio Maia, for all the opportunities, lessons and reprimands when needed, a multitude of discussions that made me grow as a person and as an engineer.

Contents

List of Figures xv

List of Tables xvii

List of Algorithms xix

List of Source Code xxi

List of Acronyms xxiii

1 Introduction 1
- 1.1 Context 2
- 1.2 Problem 3
- 1.3 Objective 4
- 1.4 Value Analysis 5
- 1.5 Work Methodology 6
- 1.6 Document structure 6

2 Background 9
- 2.1 System Model 9
- 2.2 Real-Time Systems 9
- 2.3 Real-Time Scheduling 10
- 2.4 Reservation-based Approaches 11
- 2.5 Constant Bandwidth Server 12
- 2.6 Capacity Sharing and Stealing 13
- 2.7 Summary 15

3 Value Analysis 17
- 3.1 Innovation process 17
  - 3.1.1 New Concept Development 18
- 3.2 Value 20
  - 3.2.1 Customer Value 20
  - 3.2.2 Perceived Value 21
- 3.3 Value Proposition 21
- 3.4 Summary 21

4 RTEMS 23
- 4.1 Task Manager 23
  - 4.1.1 Task Priority 24
  - 4.1.2 Task Modes 25
- 4.2 Memory Management 26
- 4.3 Communication and Synchronization 26

4.4 Rate Monotonic Manager . . . 26
4.5 Scheduling . . . 26
4.5.1 Uniprocessor Scheduling . . . 27
Deterministic Priority Scheduler . . . 27
Simple Priority Scheduler . . . 27
Earliest Deadline First Scheduler . . . 27
Constant Bandwidth Server Scheduler . . . 27
4.5.2 Symmetric Multiprocessing Scheduling . . . 28
4.6 Summary . . . 28

5 Implementation 31
5.1 Constant Bandwidth Server . . . 31
5.1.1 Release . . . 32
5.1.2 Unblock . . . 34
5.1.3 Block . . . 34
5.1.4 Budget Allocation . . . 35
5.1.5 Budget Callout . . . 36
5.1.6 Example . . . 37
5.2 Capacity Sharing and Stealing . . . 38
5.2.1 Budget Allocation . . . 38
5.2.2 Budget Reclaiming . . . 39
5.2.3 Budget Stealing . . . 40
5.2.4 Budget Callout . . . 42
5.2.5 Schedule . . . 42
5.2.6 Release . . . 42
5.2.7 Unblock . . . 42
5.2.8 Block . . . 42
5.2.9 Example . . . 43

6 Experimentation and Assessment 45
6.1 Tardiness evaluation . . . 46
6.2 Overhead evaluation . . . 46
6.3 Discussion . . . 47

7 Conclusions and Future Work 49
7.1 Conclusion . . . 49
7.2 Future work . . . 49

Bibliography 51

A Appendices 53
A.1 CBS Structures . . . 53
A.2 CBS Release . . . 55
A.3 Tick Entry . . . 56
A.4 CBS Budget Callout . . . 57
A.5 CSS Budget Allocation . . . 58
A.6 CSS Budget Reclaiming . . . 60
A.7 CSS Budget Stealing . . . 61
A.8 CSS Budget Callout . . . 62
A.9 CSS Schedule . . . 63

A.10 CSS Server Activation 64

List of Figures

1.1 Risk classification of automotive components. **Image credit:** *APTIV* . . . 2
1.2 Two-wheeler instrumentation cluster. **Image credit:** *BMW press group* . . 4
1.3 Cluster partitions 5
1.4 Business value. 6

3.1 Innovation process. **Image credit:** *(Dimitrijevic 2014)* 17
3.2 New Concept Development model. **Image credit:** *(Koen et al. 2001)* . . . 18

4.1 RTEMS internal architecture. **Image credit:** *(*RTEMS Classic API Guide *n.d.)* 24
4.2 RTEMS task state transitions. **Image credit:** *(*RTEMS Classic API Guide *n.d.)* 25

5.1 CBS job release 32
5.2 CBS expired job release 33
5.3 CBS update heir 34
5.4 CBS unblock 34
5.5 CBS block 35
5.6 CBS schedule 35
5.7 CBS budget allocation 36
5.8 CBS scheduling example 37
5.9 CSS budget allocation 39
5.10 CSS budget reclaiming 40
5.11 CSS budget stealing 41
5.12 CSS unblock 43
5.13 CSS block 43
5.14 CSS scheduling example 44

6.1 CSS vs CBS. 46

List of Tables

3.1 Benefits and Costs of the proposed solution 20

6.1 Context Switch average length (milliseconds) 47

List of Algorithms

List of Source Code

A.1 CBS server . . . 53
A.2 CBS server context . . . 54
A.3 CBS scheduler node . . . 54
A.4 CBS release . . . 55
A.5 Tick entry . . . 56
A.6 CBS budget callout . . . 57
A.7 CSS budget allocation . . . 58
A.8 CSS budget reclaiming . . . 60
A.9 CSS budget stealing . . . 61
A.10 CSS budget callout . . . 62
A.11 CSS schedule . . . 63
A.12 CSS server activation . . . 64

List of Acronyms

ABS Anti-lock Braking System.
ADAS Advanced Driver-Assistance Systems.
AMP Asymmetric Multiprocessing.
API Application Programming Interface.
ARAS Advanced Rider-Assistance Systems.
ASIL Automotive Safety Integrity Level.
ASR Asynchronous Service Routines.
AUTOSAR AUTomotive Open System ARchitecture.

CAN Controller Area Network.
CBS Constant Bandwidth Server.
CGI Computer-Generated Imagery.
CPU Central Processing Unit.
CSS Capacity Sharing and Stealing.
CUS Constant Utilization Server.

DSS Dynamic Sporadic Server.

E/E Electrical and Electronic.
ECU Electronic Controller Unit.
EDF Earliest Deadline First.

FFE Fuzzy Front End.
FIFO First In, First Out.

GPOS General Purpose Operating System.

HARA Hazard Analysis and Risk Assessment.
HIL Hardware-In-the-Loop.
HMI Human-Machine Interface.

I^2C Inter-Integrated Circuit.
IPC Inter Process Communication.

LIN Local Interconnect Network.

NCD New Concept Development.
NPD New Product Development.

OEM Original Equipment Manufacturer.

PPM	Parts Per Million.
QM	Quality Management.
QoS	Quality of Service.
RAM	Random Access Memory.
RBE	Rate-Based Execution.
RCB	Resource Control Block.
RM	Rate Monotonic.
RTEMS	Real-Time Executive for Multiprocessor Systems.
RTOS	Real-Time Operating System.
RTS	Real-Time System.
SMP	Symmetric Multiprocessing.
SPI	Serial Peripheral Interface.
TBS	Total Bandwidth Server.
TCB	Task Control Block.
V2V	Vehicle-to-Vehicle.
WCET	Worst-Case Execution Time.

Chapter 1

Introduction

Nowadays, technology is omnipresent in our lives, being employed in and for almost everything. This widespread use of technology occurred due to the trend of embedding computational capacity (embedded computing systems) into everyday products. Embedded computing systems can be seen as a microprocessor/microcontroller-based computer system with both specific hardware and software intended to perform dedicated functions (usually for monitoring or controlling purposes) within larger electrical or mechanical systems.

Embedded systems components can range from a single chip microcontrollers to a set of processors with connected peripherals and network, and can be used in an enormous variety of application domains both traditional and emerging, such as:

1. Home appliances - Household appliances such as refrigerators, washing machines and microwaves;
2. Consumer electronics - Electronic devices for entertainment, communications and recreational uses e.g. televisions, digital cameras, computer printers and video game consoles;
3. Industrial automation - Assembly lines and data collection systems;
4. Military and aerospace applications - Control, navigation and guidance systems;
5. Telecommunication and data communication industries - Communication devices such as routers and satellite phones;
6. Medical Equipment - Electronic medical devices e.g. scanner and ECG machines.
7. Automotive Industry- In today's market, any motor vehicle (e.g., bus, car, truck and motorcycle) can contain, depending on the market segments, up to 5 to 100 embedded electronics systems, also known as Electronic Controller Units (ECUs);

This work focuses on the embedded development for the automotive industry, where according to (Nicolas and Françoise 2009), over the last two decades the growing consumer acceptance for embedded electronic devices and the own nature of these electronic components (e.g., real-time operation, versatility, flexibility, low cost and power) led to an increase in the number of microcontroller-based functions embedded in vehicles.

This chapter firstly discusses the domain and scope of this dissertation, followed by the presentation of the problem and a possible solution for the studied problem. Then, an explanation on how the solution is tested and which are the assessment metrics. It ends by introducing a primary value analysis of the solution, the work methodology and finally the structure of the document.

1.1 Context

Electronics completely transformed the automotive industry. Early vehicles were purely composed by mechanical components but the current reality is different. ECUs now have a tremendous role in this industry as modern vehicles have up to 2500 signals (i.e., elementary information such as the speed of the vehicle and diagnostics) exchanged through up to 100 electronic control units (ECUs) on different types of low-level networks, such as: (i) Local Interconnect Network (LIN); (ii) Controller Area Network (CAN); (iii) FlexRay; (iv) Serial Peripheral Interface (SPI); (v) and Inter-Integrated Circuit (I^2C).

To ensure passengers safety from the use of Electrical and Electronic (E/E) automotive equipment and systematic failures, Original Equipment Manufacturers (OEMs) and their suppliers must comply with standards such as ISO 26262, the road vehicles functional safety standard. ISO 26262 is a risk-based safety standard derived from IEC 61508 that provides regulations and recommendations throughout the product development process. Moreover, it details how to assign an acceptable risk level to a system or component and document the overall testing process.

Concerning safety risks, ISO 26262 specifies how Hazard Analysis and Risk Assessment (HARA) should be performed on automotive component (software/hardware) to establish safety goals and a safety-criticality level. This level can be assigned according to either Quality Management (QM) or Automotive Safety Integrity Level (ASIL).

Severity	Exposure	Controllability		
		C1 (Simple)	**C2** (Normal)	**C3** (Difficult, Uncontrollable)
S1 LIGHT AND MODERATE INJURIES	**E1** (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	QM
	E3 (Medium)	QM	QM	A
	E4 (High)	QM	A	B
S2 SEVERE AND LIFE THREATENING INJURIES – SURVIVAL PROBABLE	**E1** (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	A
	E3 (Medium)	QM	A	B
	E4 (High)	A	B	C
S3 LIFE THREATENING INJURIES, FATAL INJURIES	**E1** (Very low)	QM	QM	A
	E2 (Low)	QM	A	B
	E3 (Medium)	A	B	C
	E4 (High)	B	C	D

QM (Quality Management)
Development supported by established Quality Management is sufficient.

A **lowest ASIL** Low risk reduction necessary
B
C
D **highest ASIL** High risk reduction necessary

Figure 1.1: Risk classification of automotive components.
Image credit: *APTIV*

ASIL is a risk classification scheme assigned to a system or component by performing risk analysis of potential hazards. There are three factors that influence this analysis, namely:

the severity (extend of harm to one or more individuals that can occur in a potentially hazardous event), exposure (likelihood of an hazardous event) and controllability (if the system fails, what is the ability to avoid harm or damage through the timely reactions of the persons involved, possibly with support from external measures - Advanced Driver-Assistance Systems (ADAS) and Advanced Rider-Assistance Systems (ARAS)). There are four ASILs levels identified by the ISO 26262 standard: ASIL-A, ASIL-B, ASIL-C, ASIL-D. ASIL-A dictates the lowest safety requirements that shall be specified on the product and ASIL-D the highest. There is another level called QM that represents hazards that do not dictate any safety requirements, Figure 1.1 displays the ISO 26262 risk classification matrix.

Beside the functional safety criticality levels and the typical ECUs system function classification (e.g., engine management system, Anti-lock Braking System (ABS) and ADAS), embedded automotive ECUs can also be defined as Real-Time Systems (RTSs) [1] and categorized according to their real-time constraints. So, according to their time constraints, ECUs can be categorized as:

1. Hard real-time system - It is a system that must operate within the limit of a stringent deadline. Any deadline miss may be considered as a root cause for system failures, which may result in loss of life or property. In current motor based vehicles, the ECUs related with the engine management, transmission, chassis and ABS are some examples of this type of system;
2. Soft real-time system - Soft RTSs are systems which have tolerant time requirements. As opposed to hard systems, soft systems can tolerate some deadline misses, diminishing the computational output value according to the tardiness, as long as the value is not zero, Vehicle-to-Vehicle (V2V) communications and instrumentation clusters are some examples of this type of system.

This dissertation, made to obtain the master's degree in computational systems engineering, focuses on an ASIL-B and real-time ECUs, a two-wheeler motor-based vehicle instrumentation cluster (example of this type of ECU is depicted in Figure 1.2).

1.2 Problem

With the increase in demand for technological and innovative progress along with safety in the automotive domain, OEMs developed a new two-wheeler instrumentation cluster category, the full digital rider cockpit. These clusters are no longer just an instrumentation cluster, as now, almost all operations in a two-wheeler vehicle are sensed with actuators and displayed in digital format in the cluster itself which provides to the user real-time access to data. These digital cockpit are considered mixed-criticality systems as they compute high ASIL critical activities that have strict time boundaries (e.g., telltales and smart rear mirrors streamed on the cluster) with activities with less stringent constraints regarded as soft real-time activities (e.g., multimedia and connectivity activities).

When working with mixed-criticality systems, the system designer must guarantee some sort of temporal isolation between tasks (i.e., the capability for each process to complete within its timing constraints do not depend on the temporal behavior of other unrelated processes running on the same system). Without it, the system may not be capable of guaranteeing the completion of hard real-time activities due to the properties of the soft

[1] A system that is developed and analyzed to guarantee a worst-case response time to critical events, as well as acceptable average-case response time to non-critical events (Stefan 2008)

Figure 1.2: Two-wheeler instrumentation cluster.
Image credit: *BMW press group*

activities, such as: (i) their dynamic behaviour (ii) no *apriori* guarantee can be achieved due to their non-deterministic minimum inter-arrival time; (iii) Worst-Case Execution Time (WCET) estimations are extremely complicated to obtain for them and thus, significant upper bounds are used which can lead to waste of CPU resources.

This dissertation focuses on enhancing two-wheeler mixed-criticality instrumentation clusters safety since a typical general-purpose linux kernel without temporal isolation is commonly employed. Without temporal isolation, instrumentation cluster's hard real-time activities are not guaranteed, and consequently the customers safety, as soft real-time activities can interfere on their completion within the defined constraint and lead to an hazardous event.

1.3 Objective

A digital instrumentation cluster is generally partitioned into two partitions, see Figure 1.3. The first partition, is developed using the standardized software framework AUTomotive Open System ARchitecture (AUTOSAR) - which in simple terms is an OEM and suppliers consortium that aims to standardize software architecture for this industry, and, a second partition that focuses on connectivity and the Human-Machine Interface (HMI), which it is typically managed by a general-purpose linux kernel. This proposed work aim is to cope with the coexistence of real-time mixed-criticality elements within the second partition.

As current General Purpose Operating System (GPOS) commonly do not provide temporal isolation, the best approach to resolve the problem of interference between tasks with different time constraints and determinism would be to employ either a Real-Time Operating System (RTOS) that already has some type of temporal isolation and *apriori* guarantee of hard real-time tasks and adjust into this use case or implement in a GPOS real-time mechanisms to guarantee the completion of hard real-time activities and ensure temporal isolation within the mixed-criticality tasks.

Following this mindset, this work proposes an alternative to the GPOS with temporal isolation mechanisms by demonstrating the adjustment of an open-source RTOS, Real-Time

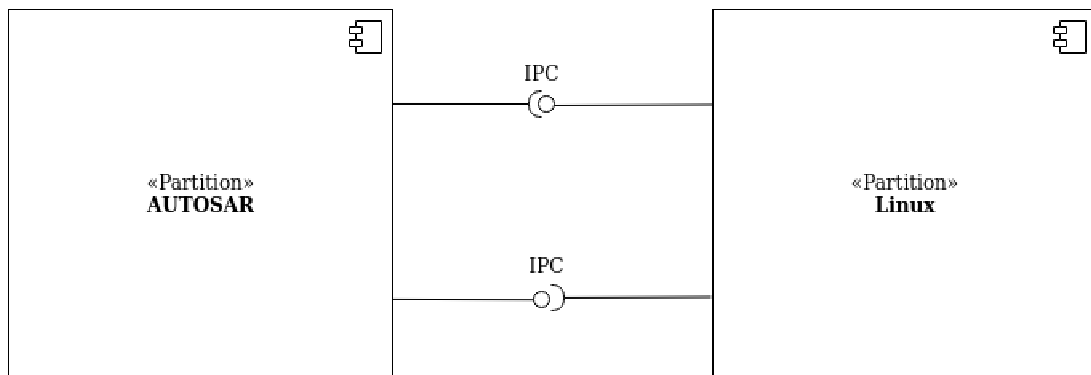

Figure 1.3: Cluster partitions

Executive for Multiprocessor Systems (RTEMS), to mixed-criticality systems. To apply RTEMS into this use case and guarantee that hard real-time tasks are not jeopardized by the soft activities, this work proposes the implementation of uniprocessor reservation-based mechanisms, namely the Constant Bandwidth Server (CBS) (Abeni and G. Buttazzo 1998) and the Capacity Sharing and Stealing (CSS) (Nogueira and Pinho 2007).

These mechanisms allow critical tasks to be scheduled by their absolute deadline, by using an algorithm known as the Earliest Deadline First (EDF), and the remaining tasks with either CSS or CBS, which use the concept of dedicated servers where a fraction of the Central Processing Unit (CPU) bandwidth is reserved for tasks, thus relaxing the need for knowing certain properties of the tasks such as the WCET.

Concerning the technical work, what is proposed regarding the CBS is an improvement of the already existing algorithm in RTEMS, since, the current implementation is considered as a *Hard-CBS* implementation (further explained in Chapters 2.5 and 4). On the other hand, the CSS implementation in RTEMS is made from scratch and follows the rules defined in (Nogueira and Pinho 2007), allowing one to create isolated and non-isolated servers for both periodic and aperiodic tasks in RTEMS (concept further explained in Chapter 2.6).

The performance of both algorithms is evaluated and compared through a set of generated task sets executed over both an emulator and a real platform (trying to simulate Hardware-In-the-Loop (HIL) tests). The task sets encompasses hard, soft real-time periodic tasks, and soft aperiodic tasks and were repeatedly executed for a significant period of time to ensure consistency in the results. The metrics used to measure the algorithms performance were the mean tardiness and average deadline misses, computed over all soft real-time tasks, and the average length of context switch operations.

1.4 Value Analysis

Integrate a hard RTOS with reservation mechanisms on digital mixed-criticality instrumentation clusters would be a solution with great perspectives within the current automotive market. Despite the enormous initial costs and work load necessary to develop support for required Application Programming Interfaces (APIs), teams mindset adjustment, graphical support and integrate Computer-Generated Imagery (CGI) tools. The presented solution, not based on a GPOS, would mainly lead to significant increase of product safety and reliability.

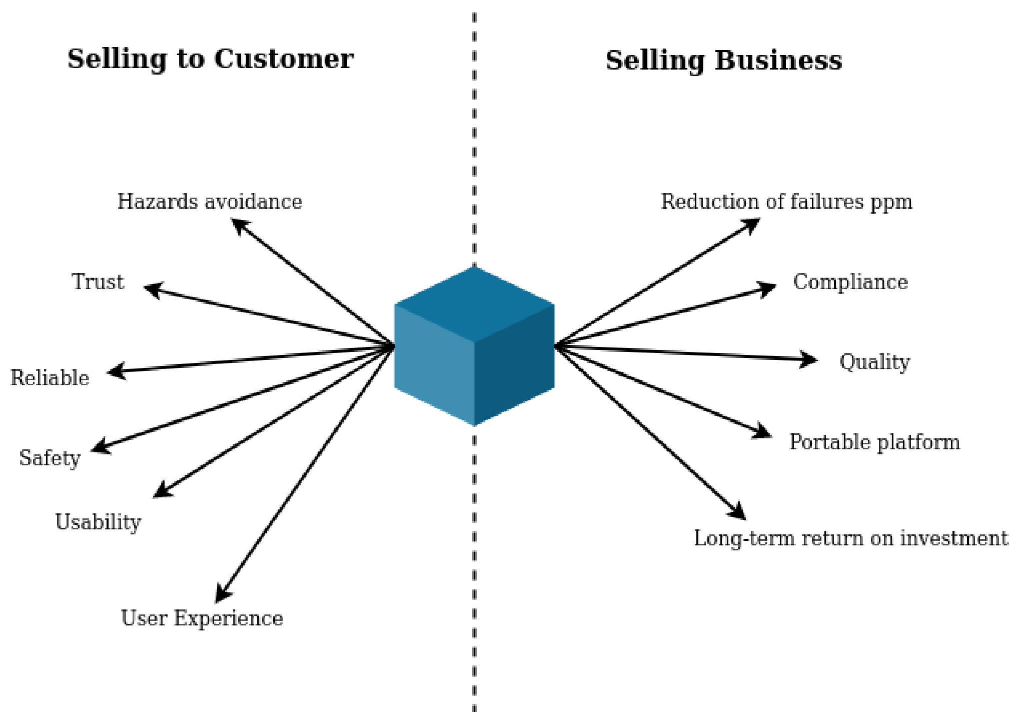

Figure 1.4: Business value.

Figure 1.4 represents the value of an instrumentation cluster developed with the proposed solution. The right side points out the value for the OEMs, meanwhile the left side represents the value for the customer. For the customer, this solution based on temporal isolation would mainly increase his safety as it reduces the possibility of the occurrence of hazardous events. Additionally, having guarantee the isolation property, OEMs could focus themselves on improving cluster's HMI, thus increasing user usability and experience. On the other hand, for the OEM it would essentially improve the quality of the product and consequently reduce the number of failures Parts Per Million (PPM). A further detailed value analysis is presented in section 3.

1.5 Work Methodology

Due to the lack of flexibility and the own nature of the project himself, it has been agreed that a waterfall approach would be best suited for the development of this solution. On an initial stage, the problem and context are identified and assessed. And following the New Concept Development (NCD) mindset, the idea generation and selection follow to chose the most adequate solution. With the reservation based scheme and the RTOS selected, a thorough analysis on RTEMS occurs and a selection on which schemes are to be implemented. With all requirements considered, the development phase follows. Finalizing the solution with the validation, tests and conclusion phases.

1.6 Document structure

The presented dissertation is composed by the following seven Chapters: Introduction, Background, Value Analysis, RTEMS, Implementation, Experimentation and Assessment, and at

last the Conclusions and Future Work.

Chapter one, **Introduction**, has the aim of contextualizing the context and the problem in which the solution suits. Moreover, a brief value analysis with the business value for both the customer and the organization and the steps followed for the development of the solution are presented.

The second chapter, **Background**, is divided into seven parts. The first one presents the assumptions assumed for the development of the solution. The second and third part gives a more extended overview of the context by detailing further concepts of real-time systems and real-time scheduling. Section four, discusses some reservation-based approaches existing in the literature, presents the ideas and goals of each approach and the major blockers of some of these less successful scheduling policies. The fifth section, specifies the employed system model. At the end, sections six and seven displays an overview and specifies the requirements of both the CBS and the CSS.

Chapter three, **Value Analysis**, an extended value analysis is performed, where value of the solution is identified and discussed through multiple techniques.

Chapter four, **RTEMS**, presents the employed real-time operating system and later focuses on some important concepts of RTEMS, namely task management and the existing scheduling policies.

On the fifth chapter, **Implementation**, the implementation of both schedulers in RTEMS is fully described. Additionally, the implementation of some application tests is also presented.

On the sixth chapter, **Experimentation and Assessment**, the testing and metrics assessment processes are fully explained. It starts by demonstrating the task set and the implementation of some developed tests, followed by a presentation of the emulator and hardware employed for the executing and data extraction steps. At last, the data is assessed according to the defined metrics.

Chapter seven, **Conclusions and Future Work**, summarizes the work and additionally presents the difficulties and limitations of this project, and provides the next steps.

Chapter 2

Background

This chapter is divided into six sections, the initial section presents the system model employed and some assumptions made for the development of the solution. The second and third sections aim at presenting some further details concerning some technical concepts to fully understand the proposed solution. The fourth section presents some reservation based approaches existing in the literature. Sections five and six present the CBS and CSS correspondingly. These sections and chapter 4 additionally correspond to the requirements and design analysis, since from the original articles and operating system study the overall specification and system design are given and analysed.

2.1 System Model

For this proposed solution of having a hard real-time operating system with capabilities of offering Quality of Service (QoS) (execution time) to soft real-time tasks without jeopardizing hard critical activities, it is assumed a single-process multiple-threaded system based on tasks. This assumption is made by the chosen RTOS requirements themselves, where a single-process is admitted and the tasks are considered the smallest thread of execution that are able to compete on its own for system resources.

Due to RTEMS limitation to define constraint deadlines, only implicit deadlines are employed. Deadlines can be described as explicit or implicit, an implicit deadline exists when the deadline of job $J_{i,j}$ is equal to its periodicity T_i. On the other hand, explicit deadlines are characterized by having a relative deadline shorter than the task periodicity.

Furthermore, a system consisting of two types of task, hard and soft real-time tasks, is considered. Hard tasks, τ_i, are defined as periodic where each job $J_{i,j}$ of the task is characterized by an arrival time $r_{i,j}$ and by two additional parameters, (C_i, T_i), whereas C_i represents the WCET of each job and T_i is the minimim inter-arrival time between each job, such that $r_{i,j+1} = r_{i,j} + T_i$. For soft tasks, both periodic and aperiodic tasks are assumed to match the soft multimedia and connectivity activities of the cluster and are also represented with an arrival time, average case execution time and a soft deadline.

2.2 Real-Time Systems

The application of RTSs is seen in several industries, from defense and space systems up to automotive, medical devices, consumer electronics and even home appliances. They are mainly used within these domains as a control device in a dedicated application where time requirements are rigid (Sha et al. 2004). As briefly mentioned in section 1.1, these systems

are characterized by its predictability and determinism, not only on the logical correctness. On these systems, the maximum instant in which a system shall respond to an external stimulus is typically imposed by the environment and is called deadline.

RTSs classification is dictated by the value/utility of the result upon deadline misses, if there is still value after it, the system is classified as soft (e.g., multimedia and network systems), otherwise it is firm. If a catastrophe could occur from a deadline miss, the system is regarded as hard (e.g., earthquake alerts and air traffic control systems). Hard real-time systems guarantee that all critical tasks will be completed on time and thus all overhead shall be bounded (e.g., memory operations - fetch and store, Inter Process Communication (IPC), operating system tasks overhead and I/O operations), on the other hand, soft real-time systems are much less restrictive concerning their time boundaries and can tolerate late results as long as the value of the result does not diminishes to zero. Soft RTS are mainly employed in multimedia and common communications systems.

Moreover, the capability of managing a considerable number of concurrent activities is commonly expected from the environment in which the system is integrated. This becomes a problem in typical synchronous software systems, thus requiring that RTSs shall have mechanisms for the processing of internal synchronous activities joined with the external events (e.g., interrupts). This challenge enhances when dealing with multiprocessors. Despite the theoretical processing benefit of multiprocessor systems there are new aspects that need to be considered due to their implication on the predictability of real-time systems, such as: tasks allocation; inter processor communication channels and global resources that may to be shared between competing processors.

An essential component of any RTS is its real-time operating system. That besides being a software responsible for managing hardware resources, it serves as an interface between the hardware components and applications. RTOS are additionally designed to execute applications with precise timing and reliability. This timing precision is one of the key elements that distinguishes RTOS and typical general purpose operating systems. While GPOSs have random execution pattern, not guarantee response times, dynamic memory mapping and are non-preemptive [1]. RTOS are reliable, predictable, deterministic and have a priority based preemptive scheduling which is the key to provide accurate real-time responses to external events.

2.3 Real-Time Scheduling

Scheduling is a process performed by a component named scheduler in which virtual computation elements such as tasks, threads and processes are assigned to hardware elements (e.g., CPU) in order to be executed and complete their work. In GPOSs the following three types of schedulers can be found:

1. Long-term scheduler - Also known as the job scheduler, it selects the processes from the storage pool in the secondary memory and loads them in the main memory for execution;
2. Medium-term scheduler - Responsible for performing the swapping out and in process, which is basically the removal of processes from Random Access Memory (RAM) and placing them on secondary storage (e.g., disk) or vice versa;

[1]An executing thread can only yield upon completion, I/O operation or volunteer operation.

3. Short-term scheduler - Also referred as CPU scheduler. It selects from the computation elements located in main memory that are ready to execute and allocates one of them to the CPU.

An important concept involved in the scheduling process is the dispatcher. The dispatcher main function is to perform context switches, which is the act suspending and state storing of a computation element running in a specific CPU to allocate another computing element chosen by the scheduler to execute on that CPU. This allocation requires loading the context of the selected thread or task.

In RTSs, scheduling is regarded as the most important process. The scheduler is generally a short-term task scheduler and is typically preemptive (i.e., a task which is executing can be replaced by a task with higher priority). Scheduling can be categorized into static and dynamic scheduling. With static scheduling, a run-time schedule table is made offline based on the *apriori* knowledge of the task set (such as WCET, deadlines, priorities and precedence) thus allowing the decisions to be made at compile time and, hence, reducing run-time overhead (Stankovic, Ramamritham, and M. Spuri 1998). Static scheduling is not recommended to unpredictable or dynamic systems as scheduling table can not be modified online.

Dynamic scheduling is much more flexible and adaptive but it displays a significant overhead as decisions are made at run time. An example of a dynamic scheduling algorithm is the EDF, a dynamic preemptive algorithm based on dynamic priorities in which the tasks with earliest deadline are assigned with the highest priority.

2.4 Reservation-based Approaches

One of the approaches employed to tackle mixed criticality systems are schemes on which fractions of the CPU are reserved for the soft and hard real-time activities, known as the reservation-based schemes. Several solutions based on these schemes were proposed in the literature to guarantee QoS for soft real-time tasks without jeopardizing the execution of hard real-time tasks. Next, a few examples of reservation based schemes are presented.

In (K. Jeffay, Stone, and Smith 1992), the authors presented a hard real-time system used as a test bed for video conferences and scheduled by EDF. Although real-time tasks could be guaranteed based on their WCET and minimum inter-arrival time, the inter-arrival time had an unreasonable upper bound due to the network unpredictability. To tackle this bounding issue, a new task model totally independent from the minimum inter-arrival time was presented in (Kevin Jeffay and Bennett 1995), the Rate-Based Execution (RBE) task model. With this model, the schedulability of hard real-time activities were not jeopardized though network activities were not guaranteed to complete within the expected response time.

In (Kaneko et al. 1996), Kaneko et al. introduced a scheme based on a periodic process dedicated to the service of soft real-time tasks in order to integrate soft tasks with hard real-time tasks. However it only proposes one server to schedule the served tasks. Another scheme, based on CPU capacity reserves, is presented by Mercer, Savage and Toduka in (C. W. Mercer, S. Savage, and H. Tokuda 1993). This solution reserves a fraction of the CPU bandwidth to each task by establishing an upper bound on the computational time C_i in each period T_i, thus removing the need of knowing *apriori* the WCET of each task. However, it presents a serious issue when handling overload situations.

In (Deng and Liu 1997), the authors presented a hierarchical scheduling model, which requires the *apriori* knowledge of the WCET of all tasks and that allowes the coexistence of hard, soft and non real-time activities within the same system. With this solution, each task is handled by a dedicated server, which can be the Constant Utilization Server (CUS) (Deng, Liu, and Sun 1997) for tasks that do not use non-preemptive sections or global resources, and the Total Bandwidth Server (TBS) (Spuri and Buttazzo 1994) for the remaining tasks.

In (Abeni and G. Buttazzo 1998), Abeni and Butazzo proposed the CBS. This scheme reserves a fraction of the CPU bandwidth to each task while assuring that task overloads do not occur by allocating each task to a dedicated server. For that, hard real-time tasks have a WCET and minimum inter-arrival time, while soft real-time tasks are served assuming average-case values for execution-time and period. Soft tasks execute in the context of a server and both hard tasks and servers are executed following the EDF scheduling policy (further details concerning CBS are provided in Section 2.5).

To increase resource utilization of reservation-based schemes, other works proposed reclaiming capacity not used by dedicated servers, they exploit early completions of tasks executing in the context of a server. For instance, Caccamo and Buttazzo presented CASH (Caccamo, G. Buttazzo, and Lui Sha 2000), a scheduler that allows servers to utilize unused capacities, that are originated from early completions, before using their own budget. CASH stores all the unused budget in a global queue, ordered by deadline.

In (Mercer, Savage, and Tokuda 1994) Mercer, Savage and Tokuda presented GRUB, a greedy scheduling model that minimize tasks preemption by allocating all excess bandwidth to the current executing server and postponing the servers' deadline before the arrival of a new job. Nogueira and Pinho proposed in (Nogueira and Pinho 2007) a new way to handle overloaded servers with CSS, a mechanism based on the assignment of residual bandwidth to overloaded servers. Moreover, by admitting the coexistence of two bandwidth server types, this model is capable of diminish the mean tardiness of guaranteed jobs. CSS diverges from CASH and GRUB by suspending budget recharging and deadline update until a specific time (further details concerning CSS are provided in Section 2.6).

2.5 Constant Bandwidth Server

The Constant Bandwidth Server (Abeni and G. Buttazzo 1998) is a reservation-based scheduling algorithm that works on top of EDF and is based on both the TBS (Spuri and Buttazzo 1994) and the Dynamic Sporadic Server (DSS) (Ghazalie and Baker 1995). It handles hard and soft real-time tasks by providing temporal protection between tasks through the means of servers. In this model each task with soft requirements is served by a server. CBS specifies that a server represented by S_s has two parameters (Q_s, T_s), where Q_s corresponds to the maximum budget and T_s is the server period (or also called the reservation period). The server bandwidth is given by the ratio $U_s = Q_s/T_s$. Additionally, the server has a fixed deadline $d_{s,k}$.

Temporal isolation is regarded by Abeni and Buttazzo as the most important one as it allows activities to run without interference from each other concerning their temporal constraints, and it expressed with the following theorem:

Theorem 1 *Given a set of n periodic hard tasks with processor utilization U_p and the sum of all servers processor utilization U_s the whole set is schedulable by EDF if and only if*

$$U_p + U_s \leqslant 1$$

With theorem 1 and its respective proof, (Abeni and G. Buttazzo 1998) demonstrated that the isolation property allowed the system to allocate a faction of the processor utilization to the soft tasks. Hence, guaranteeing that soft real-time tasks can be scheduled together with hard tasks without affecting the *apriori* guarantee of hard tasks. In the original paper, CBS is defined according to the following rules:

- When a job $J_{i,j}$ arrives and is served by an active server S_s the request is enqueued in a First In, First Out (FIFO) queue.
- $J_{i,j}$ is assigned with a deadline equal to the server deadline $d_{s,k}$, such as $d_{i,j} = d_{s,k}$.
- At any finite interval of time $[t_a, t_b]$, the budget c_s is different from 0. When $c_s = 0$, it is recharged to $c_s = Q_s$ and a new deadline is generated $d_{s,k+1} = d_{s,k} + T_s$.
- If there are pending jobs on the server queue at time t, the server is considered active, otherwise the server is idle and said be inactive.
- When a job $J_{i,j}$ arrives and the respective server is inactive, if $c_s \geq (d_{s,k} - r_{i,j})U_s$ then the CBS server generates a new deadline $d_{s,i} = r_{i,j} + T_s$ and the budget is recharged to the maximum value Q_s. Otherwise the job is served using the current deadline and budget.
- When a server serves a job $J_{i,j}$ of τ_i for a period of time δ, the budget c_s is decreased by δ : $c_s = c_s - \delta$.
- When a job finishes, if there is any pending job, it is served with the current budget and deadline.
- When a job $J_{i,j}$ arrives and the server is active the request is enqueued in a queue of pending jobs according to a given non-preemptive discipline.

According to the above rules, CBS offers quality of service (QoS) to each served task allowing it to execute during the reserved computation time, which is guaranteed by the server, without compromising the guarantees of any hard real-time task.

2.6 Capacity Sharing and Stealing

The Capacity Sharing and Stealing (Nogueira and Pinho 2007) is a bandwidth-based scheduler created to efficiently handle overloaded servers and reduce the mean tardiness of soft real-time jobs through an efficient management of unused capacities.

CSS recognizes the coexistence of two types of servers: *isolated servers*, used to schedule periodic guaranteed tasks and are ensured a specific amount of resource every period; and *non-isolated servers*, for aperiodic and sporadic tasks that can be served in a best-effort manner. Both types of servers are characterized by a tuple (Q_s, T_s), where Q_s represents the reserved capacity and T_s the server period. Each server S_s has a current capacity c_s, a deadline d_s, a recharging time r_s, an activity state, residual capacity c_r and a pointer that

points to the server from which the budget is going to be consumed, or also referred as budget accounting (the pointed server is selected using a budget allocation mechanism[2]).

With this scheduler, an isolated or non-isolated server S_i can be in an *active* or *inactive* state at time t. Thus, a server is said to be *active* if: one of the served tasks is ready to execute; or is executing; the server is supplying capacity to other servers until it reaches the ongoing deadline. A server is *inactive* if there are no pending jobs to serve; or the server has no capacity to be reclaimed by other servers. State transitions are determined by the arrival of a new job, capacity exhaustion, or the non-existence of pending jobs at replenishment time. An important aspect of non-isolated servers is that it is not guaranteed that a task τ_j served by a non-isolated server S_j can execute Q_j for every period T_j since a portion of the reserved capacity Q_j can be stolen by one or several active overloaded servers (see Inactive non-isolated capacity steal below).

When a job $J_{i,j}$ served by a server S_s is released at time t the scheduler executes one of the following steps:

- When a job $J_{i,j}$ arrives and is attached to an inactive server S_s, S_s becomes active and is inserted into the ready queue. If the job $J_{i,j}$, arrived at $a_{i,j}$, and before the server's deadline $d_{s,j}$, such that $a_{i,j} < d_{s_j}$, the job is served with the ongoing server deadline $d_{s,j}$ and using the current capacity c_s. If the $J_{i,j}$ is released after the server's deadline $d_{s,j}$, the capacity is replenished to the maximum budget $c_s = Q_s$, a new deadline is generated $d_{s,j} = max\{a_{i,j}, d_{s,j-1}\} + T_i$, the replenishment time is updated $r_s = d_{s,j}$ and the residual capacity is set to $r_c = 0$.
- if S_s is active and already executing pending work, the new job is inserted in the server's queue and served later.

An active server (S_s) has the following set of rules to perform budget allocation:

- *Residual capacity reclaim* - S_s points to the earliest deadline server S_p from the set of eligible active servers A_r for capacity reclaiming. S_s uses the residual capacity r_k^p, running with the deadline of the pointed server. When reaching exhaustion or S_p deadline and there is pending work, S_s disconnects from the pointed server and selects the next available server.
- *Dedicated capacity consumption* - when all residual capacity eligible from the set of active servers A_r is exhausted and the job is not completed, S_s consumes its own reserved capacity c_k^s either until job completion or c_k^s exhaustion.
- *Inactive non-isolated capacity steal* - when the capacity c_k^s is exhausted and there is still pending work to do, S_s connects to the earliest deadline server S_p from the set of eligible inactive non-isolated server I_s to steal its capacity c_k^p, running with its own deadline d_k^s.

When S_s completes its pending work and its capacity is not exhausted $c_s > 0$, S_s releases its remaining budget as residual capacity $r_s = c_s$ and sets c_s to zero. The released residual capacity can then be reclaimed by eligible active servers either until S_s current deadline or r_s exhaustion.

[2]Budget allocation mechanism is a algorithm responsible for assigning a server where the capacity is going to be consumed.

2.7 Summary

This chapter described the required context and theoretical details to allow the reader to comprehend the implementation presented in chapter 5. With this context, the reader can understand that there are systems which are developed to guarantee worst-case response times and average-case response times to critical and non-critical events, respectively.

Additionally, an introduction to the temporal isolation mechanisms that exist to cope with mixed criticality systems is presented, in particular, an analysis of both CBS and CSS is specified. These algorithms implement the concept of reservation based scheduling, where a fraction of the CPU is allocated for the scheduling of non-critical activities and the remaining reserved for critical activities (ensuring the temporal non-interference between reserved fractions).

Chapter 3

Value Analysis

The term value analysis was originally introduced as an accident of necessity during World War II, when the administration of the American company General Electric noticed that, while seeking substitutes for their shortages of skilled labour, raw materials, and component parts, these substitutions often reduced costs, improved products or even both. In simple and objective terms, value analysis is a managerial decision-making process to assess how to increase the value of a product or service at the lowest cost, without sacrificing quality. This systematic process is divided into the following phases: (i) orientation; (ii) information; (iii) innovation; (iv) evaluation; and (v) implementation

This chapter specifies the techniques used throughout this systematic process, the business value and the value generated by the solution presented in this dissertation.

3.1 Innovation process

The value analysis innovation process can be divided into three domains: the Fuzzy Front End (FFE), the New Product Development (NPD) and the commercialization, see Figure 3.1.

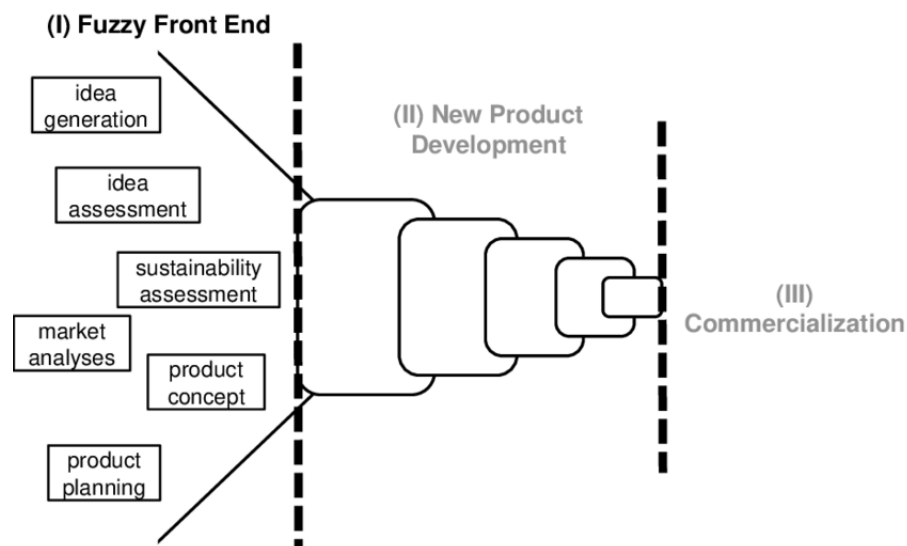

Figure 3.1: Innovation process.
Image credit: *(Dimitrijevic 2014)*

Despite the positive outcome of the FFE innovation method, the FFE stymied due to the difficulty of comparing this method across companies. There was a lack of standardized language and vocabulary, Nonaka et al. even alleged in (Krogh, Ichijo, and Nonaka 2000) that it could be impossible to acquire new knowledge and create distinctions between different phases of the process due to the lack of standardization. To solve this defect, Koen et al. presented in (Koen et al. 2001) a new theoretical construct to provide insight and common language for the FFE, the NCD.

3.1.1 New Concept Development

This new model emerged as the solution for the non-standardized language of the FFE model. The NCD model, Figure 3.2, specifies common terms and definition of the key elements of the FFE. It follows a circular and interactive flow as ideas and concepts are expected to go over all defined elements. The arrows pointing to the model represent starting points and illustrates that projects begin at either opportunity identification or idea generation and enrichment. The exiting arrow represents how concepts leave the model and enter the NPD process. NCD is divided into three key parts:

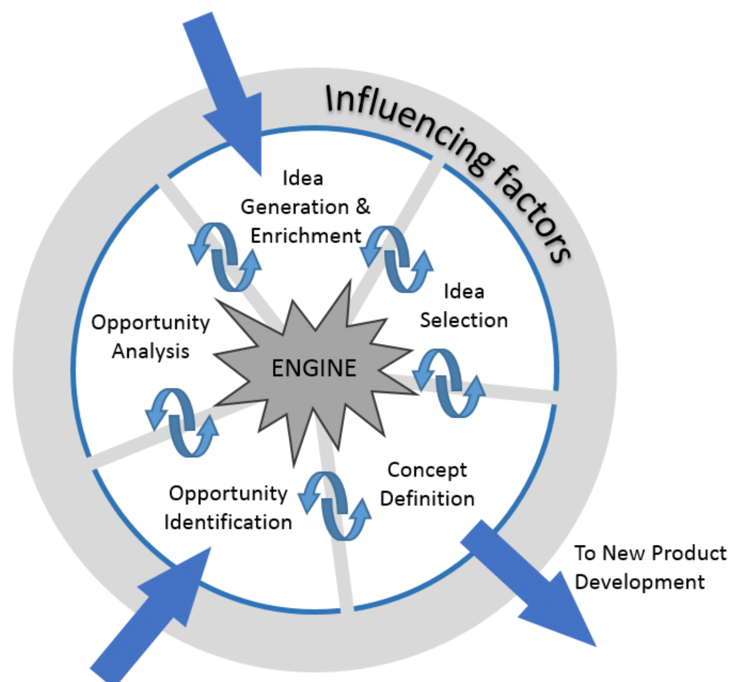

Figure 3.2: New Concept Development model.
Image credit: *(Koen et al. 2001)*

1. The engine - Is the center of the model, it accounts for the leadership, culture, business strategy, and overall management issues that drive the five elements of the front end;
2. Inner spoke - Specifies the FFE five controllable elements:
   (a) Opportunity identification - Identification of any business or technical that may solve obstacles and generate market advantages. The opportunities are identified according to the need of resources, technology on business processes;
   (b) Opportunity analysis - Analysis and assessment of the opportunities previously identified;

(c) Idea generation and enrichment- Evolutionary process were opportunities are transformed into mature and concrete product ideas. This phase includes brainstorming sessions and idea banks;

(d) Idea selection - Assessment of the generated ideas and selection of the ones with higher prospect according to the organization criteria. Typical criteria are: financial risk; product market share; development risks and future revenue;

(e) Concept definition - Final stage of the NCD and only exit to the NPD. It consists on the specification of an investment case with quantitative and qualitative information (e.g., market potential, needs of the client and the market, commercial and technical risk aspects, objectives and other) to present for the decision of pursue and invest into the new idea.

3. Influencing factors - Factors that that may affect the innovation process, such as: organization capabilities, distribution channels, law, government policies, economic climate and technological capabilities and internal know-how.

Applying the five key elements of the NCD process to the solution proposed:

1. Opportunity identification - The digital instrumentation clusters of two wheeler vehicles have seen a recent breakthrough and with improvements in connectivity, control and information. Despite all the safety concerns and validation, the new features being developed for are always increasing and safety must follow. The opportunity identified is to develop a safe solution expanding the critical functionalities, such as ADAS features;

2. Opportunity analysis - The analysis focused on how a solution could increase the responsibilities of the cluster without jeopardizing the safety of the driver and how OEMs could guarantee that the system does not overload. The full system and known similar technologies have been studied to ease the idea generation phase;

3. Idea generation and enrichment - This step started by identifying on which partition a new solution could add more value to the final product. Based on the current author knowledge, no other technology could easily and swiftly replace AUTOSAR on the first partition, so ideas were created based on how to boost safety and throughput into the second partition and still be able to expand the existing functionalities. Two main ideas emerged: (i) update general purpose linux kernel to schedule and guarantee hard real-time tasks; (ii) identify a hard RTOS and adjust it to schedule the multimedia and connectivity features (non-critical activities);

4. Idea selection - After an analysis on both ideas, the selected one is to adapt an hard RTOS. This idea can be safer on the long run as intensive testing and validation already exists and the selected RTOS may already have proven itself on different critical products;

5. Concept definition - On this phase, the operating system and the schedulers are chosen. As this is simply a proof of concept, the remaining operating system support, namely video and audio frameworks, are out of this project scope.

3.2 Value

In (S. Nicola and Ferreira 2012), the authors stated that value creation is key to any business and that all types of business activities are about exchanging some tangible and/or intangible good or service and having its value accepted and rewarded by customers or clients. This high praised business key is the fundamental foundation for a positive economical growth and sustainability. Value can be seen as the ratio between the benefits and the costs, (Holbrook 1999), where the benefits might include monetary revenue and sales increase, competitive advantage and even intangible benefits such as employee and client safety, morale and loyalty. On the other hand, the costs can represent potential risks, investments, intangible costs, and direct and indirect costs.

Table 3.1 represents the benefits and costs of the proposed solution for both the customer and selling entity.

Benefits	Safety; Usability; Reliability; Hazards avoidance; Reduction of failures ppm; Platform with enormous portability and margin to expand;
Costs	Software tools readjustment; Initial investment on team training; High acquisition cost as it is not an individual product; Effort on schedulability analysis and static code analysis; Communication and multimedia latency increase in overload corner cases

Table 3.1: Benefits and Costs of the proposed solution

3.2.1 Customer Value

Customer value can be seen as the potential rating of a product or service for the needs and satisfaction of the client as compared to the possible alternatives. In (Woodall 2003) Woodall stated that the value for the consumer is the customer personal perception of advantage acquired with a product or service, and this advantage results from any weighted combination of costs and benefits over time.

The advantage perceived by the customer for the proposed solution occurs from the ratio between its benefits and costs or sacrifices for himself. As the cluster is directly sold as a component of the motorcycle, the initial cost can be relatively high and the positive outcome of the value ratio might be difficult to be achieved at the acquisition moment and initial uses. The essential positive outcome will arise on long term uses, as crucial features (such as ADAS) will be safer and have higher responsibilities and thus ease the dangers for the customer.

3.2.2 Perceived Value

The meaning of perceived value can somewhat be extremely similar to the meaning of customer value, perceived value is the customer's own perception of a product's benefit or desirability to them, especially in comparison to a competitor's product. To reinforce this mindset, (Zeithaml 1988) stated that the "*... perceived value is the consumer's overall assessment of the utility of a product based on perceptions of what is received and what is given.*".

The solution proposed by this dissertation may diverge customers opinions. Many of them will not see the advantages of having such thorough development and safety concerns, and the major blocker will be the increased cluster price. Nevertheless, other customers will realize the safety improvement, specially on the long run, and thus approve this solution which enhances the possibility of ADAS features and consequently reduce the chances of rider induced accidents.

3.3 Value Proposition

Value proposition can be seen as how products and services as well as complementary value-added services are packaged and offered to fulfil the customer needs (Kambil and Bloch 1997). A value proposition can be defined by answering the following questions:

1. **What is the product ?** - An enhanced RTOS with reservation based schedulers to be integrated on two wheeler digital instrumentation clusters for increased safety. This integration offers higher safety to drivers and a portable platform with capabilities to increase ADAS features;

2. **Who is the target customer ?** - Concerning the end customer, the target is essentially the drivers opened to digital clusters. If this solution would be implemented on the suppliers side, OEMs searching for higher safety standards could also be included as customers;

3. **What value does the product provides ?** - The value added by the proposed solution is the safety increase. The solution has lower probabilities to miss any time deadlines and, thus, the cluster may be used to share of responsibilities on processing and displaying critical features and data.

3.4 Summary

In this chapter, the reader can perceive both the benefits and costs for the costumer, selling entity and domain of the proposed solution. All benefits and costs are displayed and measured by synthesizing several value analysis concepts and models, such as: (i) the NCD which promotes and evaluates the innovation process; (ii) the value specification, where the added benefits and costs and presented and calculated with costumer and perceived value; and at last (iii) the value proposition, in which, the product is grouped and promoted to fulfill costumer needs.

Chapter 4

RTEMS

This chapter aim is to present an overview of the real-time operating system employed on the presented solution. As introduced on chapter 1.3, RTEMS (*RTEMS Classic API Guide* n.d.) is a real-time executive that supports open standard APIs such as POSIX and ITRON, and provides to specific applications domains (e.g., space, flight, medical, networking and many more) a high performance environment with the following dedicated features:

1. Multitasking capabilities;
2. Homogeneous and heterogeneous multiprocessor systems (i.e., Asymmetric Multiprocessing (AMP) and Symmetric Multiprocessing (SMP));
3. Priority-based preemptive scheduling;
4. Intertask communication and synchronization;
5. Priority inheritance;
6. Interrupt management;
7. Dynamic memory allocation;
8. High level of user configurability.

Moreover, as a result of its single address space implementation RTEMS is also regarded as a closed RTS, whereas there is no separation between user-space and kernel-space, it consists of a single process multi-threaded environment. Considering its internal architecture, RTEMS can be regarded as a set of layered components that provides services to a real-time application. The interface presented to the application is formed by joining directives into logical sets labelled resource managers. RTEMS Core depends on a small set of processor dependent routines, being part of the executive core functions such as scheduling, dispatching and object management, that are used by several managers. The co-working of all components, as displayed in Figure 4.1, generates a powerful run time environment that promotes the development of efficient real-time application systems. In the following sections, further explanation of some essential managers and mechanisms is presented.

4.1 Task Manager

As referred in section 2.1, RTEMS environment is characterized by being single process multi-task based. RTEMS employs tasks as the smallest thread of execution able to compete on standalone manner for the system resources and are characterized by the Task Control Block (TCB). RTEMS TCBs are C data structures allocated upon task creation and released to

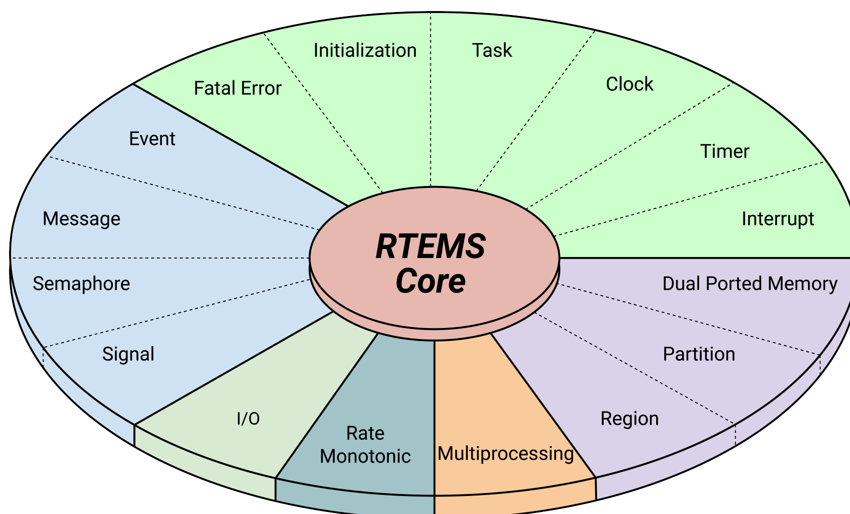

Figure 4.1: RTEMS internal architecture.
Image credit: *(*RTEMS Classic API Guide *n.d.)*

the TCB free list upon deletion. Each TCB allocated to a task contain all the information related with the task execution (e.g., task name, ID, current priority, current and starting states, execution mode, scheduling control structures).

As seen in Figure 4.2, RTEMS tasks can be in one of the following five states:

1. Executing - Tasks that are currently allocated to processors are in the executing state;
2. Ready - Tasks that are currently on the scheduler ready queues (according to the scheduling policy) and can be allocated to a processor at any time;
3. Blocked - Tasks that can not be currently allocated to any CPU due to blocking operations, namely I/O operations, blocking resource acquisition calls and synchronous communication;
4. Dormant - Tasks created that are still waiting for starting directive invocation;
5. Non-existent - Not created or deleted task.

An active task may occupy the executing, ready, blocked or dormant states, otherwise the task is considered non-existent.Tasks can not reference tasks in the non-existent state as they do not have a TCB and ID allocated to it. Although dormant tasks already have TCB and ID, they can not compete for resources and must remain in this state until a specific start directive is invoked, upon this invocation, dormant tasks are allowed to transit into the ready state and thus made available for processor allocation and resources competition. Executing tasks may block themselves (e.g., I/O operations and blocking resource acquisition calls) or blocked by other tasks in the system (e.g., using a specific directive given by RTEMS api).

4.1.1 Task Priority

In RTEMS, tasks priorities are represented by the built-in data type *rtems_task_priority* and can range from 1 to 255, where 1 is considered the highest priority level and 255 the lowest. An initial priority is given upon task creation, and can be dynamically updated according to the scheduling policy or upon specific RTEMS API invocation.

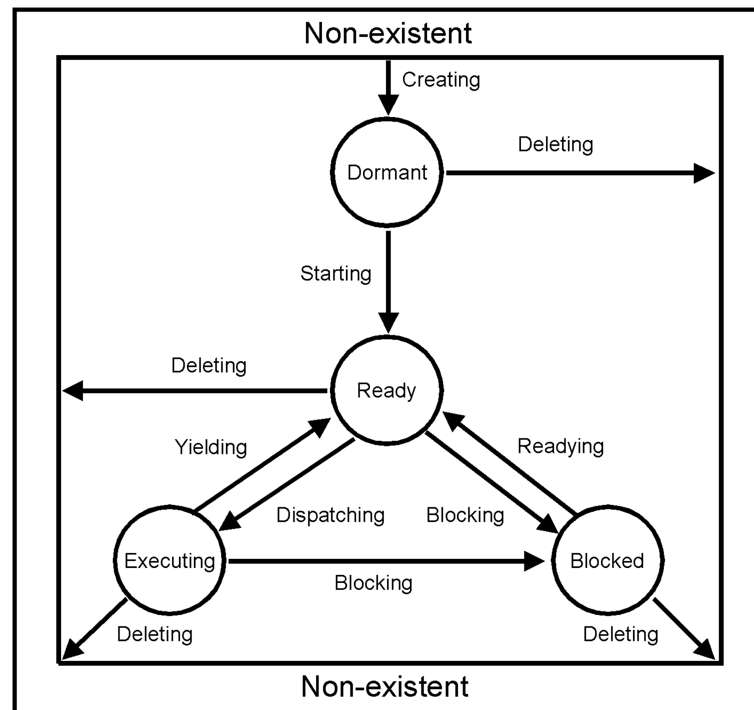

Figure 4.2: RTEMS task state transitions.
Image credit: *(*RTEMS Classic API Guide *n.d.)*

4.1.2 Task Modes

RTEMS tasks are additionally characterized by an execution mode that allows the application to modify the scheduling process and the task run time environment. The execution mode is specified through the combination of the following components:

1. Preemption - Allows a task to determine when control of the processor is relinquished. If preemption is disabled, the task will retain control of the processor as long as it is in the executing state. If preemption is enabled, then the task only controls the processor until a higher priority task is made ready;

2. Time slicing - Used by the RTEMS scheduler to determine how the processor is allocated to tasks of equal priority. If time slicing is enabled, RTEMS will limit the amount of time the task can execute before the CPU is allocated to another ready task of equal priority. If disabled, the task will be allowed to execute until a task with higher priority is made ready;

3. Asynchronous Service Routines (ASR) processing - Used to determine when received signals are to be processed by the task. If ASR processing is enabled, the signals sent to a specific task shall be processed the next time the task executes. If disabled, all signals received by the specified task shall remain posted until signal processing is enabled;

4. Interrupt level - Used to specify which interrupts are allowed when the task is running. In RTEMS, 256 interrupt levels are supported and mapped to the target processor's interrupt levels.

4.2 Memory Management

In RTEMS (*RTEMS Classic API Guide* n.d.), memory management operations can be grouped into dynamic memory allocation and address translation, whereas dynamic memory allocation is employed by user applications whose memory requirements vary dynamically. Alternatively, address translation is used by applications which share memory with another processing target.

For these memory operations, three managers are provided by the operating system. The first one, **partition manager**, provides directives to manage pools of fixed size entities such as Resource Control Blocks (RCBs). The second manager, the **region manager**, gives a general memory allocation scheme in which variable size blocks of memory can be dynamically obtained and freed by the application. At last, the **dual-ported memory** manager provides support for address translation between internal and external dual-ported RAM address space.

4.3 Communication and Synchronization

In the real-time domain, communication and synchronization mechanisms are crucial to attain some of the system requirements. In RTEMS, a vast majority of provided managers are capable of providing a basic scheme of communication and/or synchronization, though, some managers are dedicated for this purpose and grant mechanisms to match the required needs, namely:

1. Semaphore - This manager allows the creation and management of semaphores, both binary and counting semaphores are supported;
2. Message Queue - The message manager supports both communication and synchronization by passing messages among tasks and interrupts;
3. Event - Provides a high performance synchronization mechanism;
4. Signal - Commonly employed for exception handling, it provides ASR communication.

4.4 Rate Monotonic Manager

To create and use periodic tasks in RTEMS, users have to employ the Rate Monotonic (RM) manager which has as only goal the handling of the periodic behaviour of tasks. This includes information about task execution which can then be used to collect data that allows the user to analyze and tune the application. To clarify, RM manager is not the scheduler *per se*, it only provides the means that allows one to handle periodic requests. RTEMS provides a plugin framework which supports multiple scheduling algorithms, allowing the user to choose one to use in their application at link-time.

4.5 Scheduling

As referred in section 2.3, scheduling is an essential component of any real-time system as it dictates the capability of the system to provide responses to external stimulus within given timing requirements. As part of the vast user configurability provided by RTEMS,

several scheduling algorithms are available for user selection in both uniprocessor and SMP[1] architectures. Even if the given extended scheduling support is not most suitable for the user use case requirements, RTEMS additionally facilitates through its plugin framework the implementation and configuration of custom schedulers (*RTEMS Classic API Guide* n.d.). Sections 4.5.1 and 4.5.2 present an overview of the current RTEMS schedulers.

4.5.1 Uniprocessor Scheduling

In uniprocessor platforms, RTEMS presents four priority based schedulers: the Deterministic Priority scheduler, Simple Priority scheduler, EDF scheduler, and the CBS scheduler.

Deterministic Priority Scheduler

The Deterministic Priority scheduler is regarded as the default uniprocessor scheduler, used if no other is selected by the user at link-time. It has always been in RTEMS and was recently modified to suit into the operating system plugin framework. This preemptive priority based scheduler uses an array of FIFO queues in which each queue corresponds to one of the 256 priority levels (maximum priority level range, it can be configured to less upon user choice). The queues are employed to buffer the ready tasks and order them according to their priorities. Similar to all priority based schedulers, it selects the highest priority task to execute.

Simple Priority Scheduler

The Simple Priority scheduler behaviour is extremely similar to the Deterministic Priority scheduler, though it diverges as it only uses one queue to manage all ready tasks. When a task transits into the ready state, the scheduler performs a linear search on the queue to determine where to insert the task. (*RTEMS Classic API Guide* n.d.) cites that this algorithm uses much less memory resources when compared to the Deterministic Priority scheduler even though its space complexity is $O(n)$, where n is the number of ready tasks.

Earliest Deadline First Scheduler

This dynamic priority based scheduler is an alternative scheduling policy for single-core applications with a CPU utilization that can theoretically reach 100%. In RTEMS, EDF assumes the following two distinct types of task priority: deadline-driven priorities for periodic tasks (tasks which employ the RM manager to create and manage the period), and background priorities for aperiodic tasks (the application defined priority for the task is used). Aperiodic tasks have a lower importance than the deadline-driven tasks.

Constant Bandwidth Server Scheduler

As mentioned in (*RTEMS Classic API Guide* n.d.), RTEMS implements a version of the CBS scheduler that works as a budget aware extension of EDF with the intention of reserving computation time for all jobs of a given task. The aim of this implementation is to guarantee temporal isolation of tasks meaning that a task's execution in terms of meeting deadlines must not be influenced by other tasks. To attain this goal, RTEMS current implementation specifies that each task can be assigned a server, where the server is characterized by

[1] multiprocessor architectures in which all the processors are homogeneous and treated equally by the operating system

period, which is equal to its deadline, and a computational time (budget). Furthermore, (*RTEMS Classic API Guide* n.d.) defines rules for the current version, task cannot exceed their registered budget and can not be scheduled when the ratio between remaining budget and remaining deadline is higher than declared bandwidth.

The developer can interact with the scheduler through a special API allowing tasks to indicate their scheduling parameters through the following CBS directives:

- Initialize the scheduler: *rtems_cbs_initialize*.
- Create servers: *rtems_cbs_create_server*.
- Attach a task to a server: *rtems_cbs_attach_thread*.
- Detach a task from a server: *rtems_cbs_detach_thread*.
- Destroy a server and subsequently detach all the associated tasks: *rtems_cbs_destroy_server*.
- Set the server S_s parameters (Q_s, T_s) : *rtems_cbs_set_parameters*.

Current RTEMS implementation has an unexpected way to handle situations where a job exceeds the budget which *does not follow* the original paper (Abeni and G. Buttazzo 1998). In the original paper, when a job $J_{i,j}$ served by a server S_s exceeds the allowed computational time c_s the server must replenish the budget and postpone its deadline. However this is not the case for the CBS implementation in RTEMS since it neither replenishes the budget nor updates the deadline. Instead, it puts the priority of the task to background execution which unfortunately affects the job's execution time, and consequently delays its completion. Another limitation of the current implementation is that a server can only have one task attached to it.

4.5.2 Symmetric Multiprocessing Scheduling

For SMP platforms, RTEMS presents four priority based schedulers: the Deterministic Priority SMP scheduler, Simple Priority SMP scheduler, EDF SMP scheduler, and the Arbitrary Processor Affinity Priority SMP scheduler. The Deterministic Priority SMP scheduler and the Simple Priority SMP scheduler will not be further detailed as the concept remains the same on both SMP and uniprocessor targets.

The Earliest Deadline First SMP scheduler is considered as the default scheduling policy when more than one CPU is configured. In this policy, task processor affinities of one-to-one and one-to-all are supported (i.e., task can execute on exactly one processor or all processors managed by the scheduler instance). When choosing an one-to-all affinity, the set of processors shall contain all online CPUs.

The Arbitrary Processor Affinity Priority SMP scheduler is a fixed-priority scheduler that uses one queue per priority level for the ready tasks (similar to the Deterministic Priority policy). It supports arbitrary task CPU affinities, allowing a task to execute only on certain processors, depending on the processors set configuration (i.e., a task can be configured to execute only on a specific set of processors, one-to-n affinity where $1 \leqslant n \leqslant MAX_CPU$).

4.6 Summary

This chapter presented the domains in which RTEMS is employed and its key managers. The most important managers are the task manager as it manages task states, the rate

monotonic manager since it specifies how periodic tasks are created and managed, and finally the scheduling mechanisms as it specifies the RTEMS priority based schedulers.

Chapter 5

Implementation

The main objective of this work is to present a reliable solution for mixed-criticality systems that differ from GPOS with enforced real-time capabilities. Aligned with this objective, it has been assumed that adapting a hard RTOS that has already been qualified and tested for some of the most critical environments (*RTEMS SMP Qualification* n.d.) into automotive mixed-criticality systems could have a considerable positive outcome. The domain adaptation is performed by implementing two reservation based schemes, CBS and CSS, that act as an upper scheduling layer of EDF and allows multimedia and communication activities to coexist with critical tasks.This chapter is divided into two sections, where each section presents a technical overview and scheduling example for one of the mechanisms.

5.1 Constant Bandwidth Server

For CBS, the proposed implementation focuses on adjusting the current upstream CBS implementation, regarded as HARD-CBS, to behave as described in the original paper (Abeni and G. Buttazzo 1998) by correcting the following: (i) number of tasks served by a server; and (ii) the scheduler behaviour when a job exceeds its reserved capacity. These adjustments enhance the QoS for soft real-time activities, allowing them to run freely without interfering on the correct completion of hard real-time activities as long the fraction of CPU time dedicated for such, noted as U_s, and the fraction reserved for hard real-time tasks, U_p, are schedulable:

$$U_s + U_p \leqslant 1$$

On the current upstream implementation, a server can only be attached to one task as it does not define a data structure to buffer the released jobs from multiple tasks. To solve this limitation, this presented solution proposes a redefinition of the server structures and scheduler node (scheduler specialization for per-thread data) to encompass a binary searching red-black tree to buffer the released jobs. Furthermore, this use of a red-black tree improves efficiency as jobs are ordered based on their absolute deadline, though no job preemption can occur on server context, i.e., if a new job released and enqueued into the server red-black tree becomes the server's heir (head of the tree), the server will only schedule it if there is no job being currently served, if there is, the new heir will wait until either the completion or blocking of the current served job. Appendix A.1 displays the redefined CBS structures. The following subsections present the proposed primary CBS scheduling mechanisms.

5.1.1 Release

On the original implementation, when a job of a given task is released by the RM manager [1], it is directly released into EDF context, meaning that no further instructions were given on CBS context. The proposed implementation redefines this release behavior. Figure 5.1 displays the proposed behavior, in which, when a job of a given task is released, it will only be released into EDF context if it has no CBS server attached to it, otherwise, the release is purely handled by the CBS scheduler.

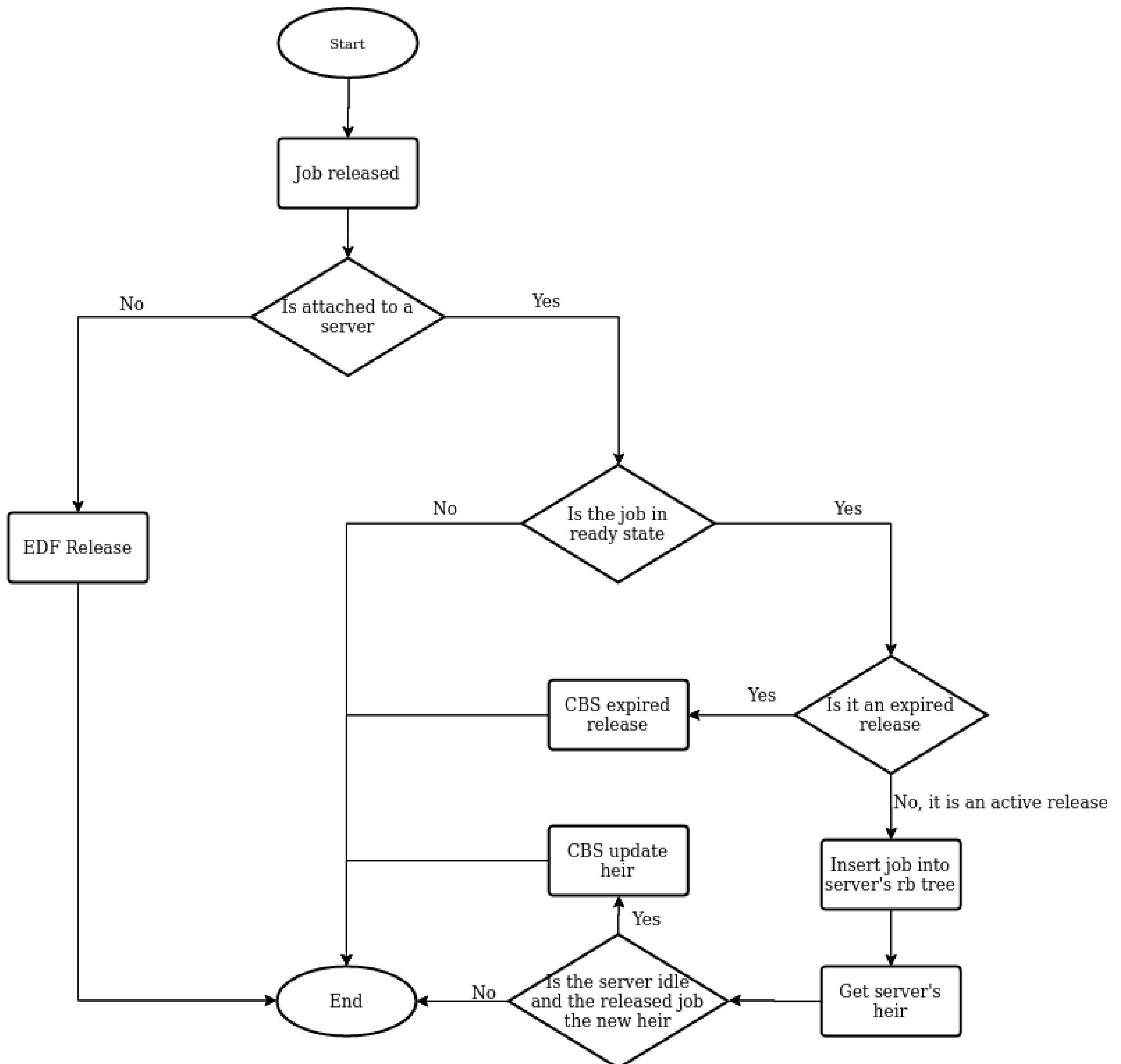

Figure 5.1: CBS job release

When a task releases a job with a server attached, the scheduler behavior depends on the task state, if the task is on blocked state[2] the scheduler only caches the new job absolute

[1] RTEMS manager responsible for the management of deadline-driven tasks
[2] State after a regular rate-monotonic release

deadline and further processing is performed on the unblock operation. On the other hand, if the task is in ready state (occurs when the rate-monotonic period is immediately initiated by either being expired[3] or never initiated[4]) the scheduler performs either an expired release or an active release. Appendix A.2 displays the listing for the CBS release.

On an expired release (see Figure 5.2[5]), where the server's current served task is the owner of the released job, the scheduler needs to update the task's red-black node key. In order to update the node key, it extracts and re-enqueues the red-black node into the server's red-black tree. Once the server's red-black tree is updated, the scheduler retrieves the heir (through the *_RBTree_Minimum*() call) and verifies if it is the expired task, if so, it performs the budget allocation operation (see subsection 5.1.4), otherwise, if there is a new heir, the scheduler performs the schedule operations and blocks the current task.

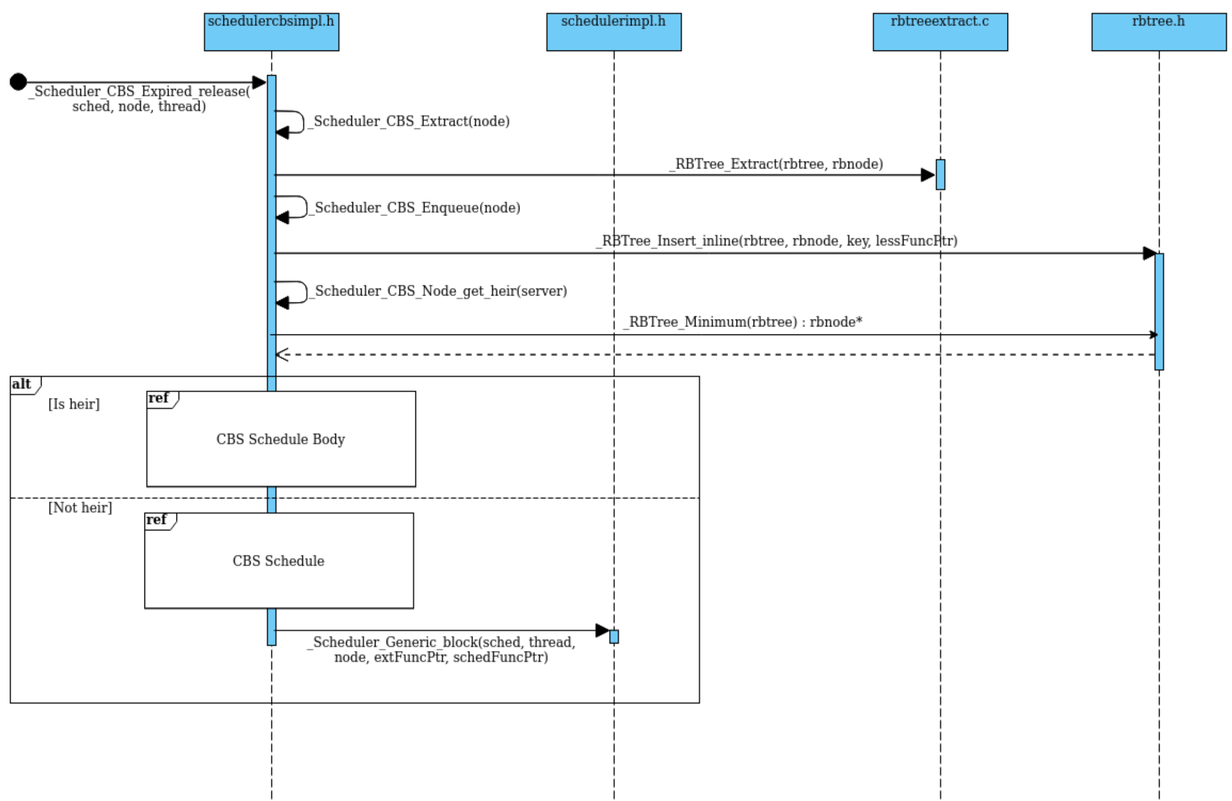

Figure 5.2: CBS expired job release

As depicted in Figure 5.1, on an active release the red-black tree node is enqueued and if the task's job is the new heir, the scheduler performs the update heir operation. The Update heir operation, represented in Figure 5.3, invokes the budget allocation mechanism (denoted as *CBS Schedule Body*) and performs the EDF unblock if the task is still to be enqueued in EDF ready red-black tree.

[3] Job release after a deadline miss occurred
[4] First RM period creation
[5] *alt*, *opt* and *ref* are sequence fragments employed to create and maintain accurate sequence diagram, and respectively describe alternative scenarios of a workflow, an optional step in workflow, and an interaction defined on another representation.

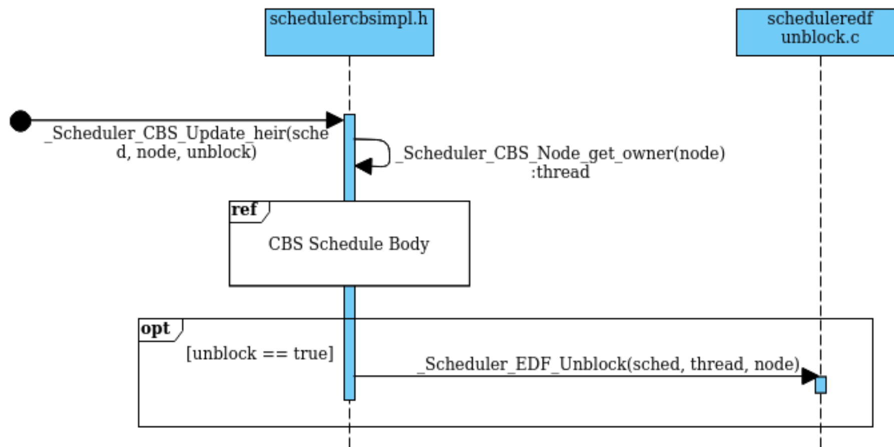

Figure 5.3: CBS update heir

5.1.2 Unblock

When a task state changes from any blocking state into the ready state, the unblock operation is executed. In other schedulers, such as EDF, this operation is used to enqueue the task into their chain policy and update the heir if required, following this approach, CBS unblock implementation performs the same operations. As seen in Figure 5.4, if a job with a server attached to it is unblocked, it is enqueued into the server red-black tree and if the server is IDLE and the job is the new server heir, the update heir mechanism is executed.

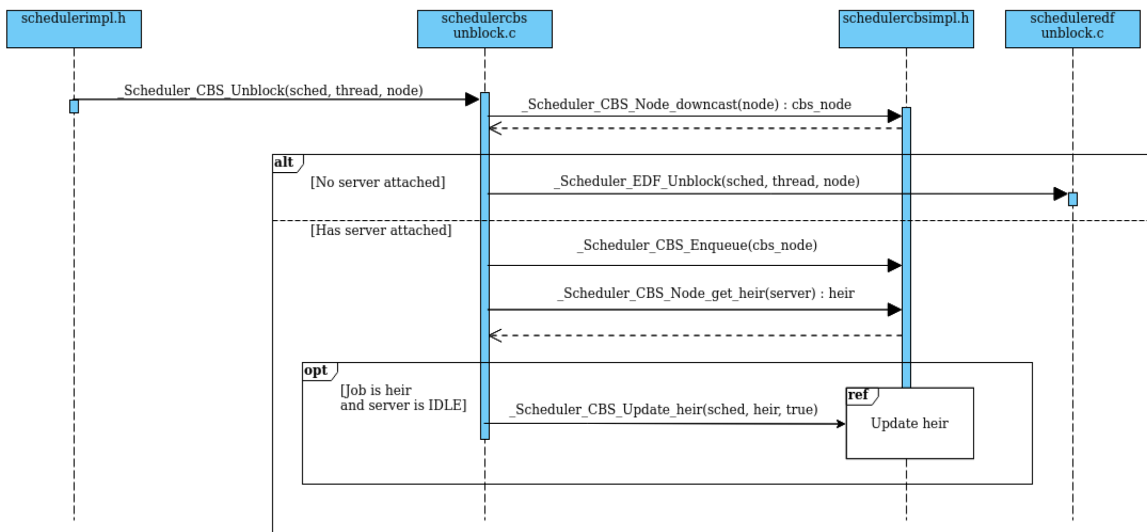

Figure 5.4: CBS unblock

5.1.3 Block

The block operation is executed when a task state change occurs and the previous state was the ready state. The proposed implementation, depicted in Figure 5.5, forwards directly hard real-time tasks into the scheduler generic block, which extracts the task from EDF red-black tree and schedules EDF new heir. Tasks with a server attached have extra steps before the generic block. These steps include the extraction of the task's red-black node from the

server's red-black tree and if the task's job being blocked was the current one being served, the scheduler caches the server remaining budget (the fraction of CPU time consumed by the job is removed from the server remaining budget) and the schedule mechanism is invoked.

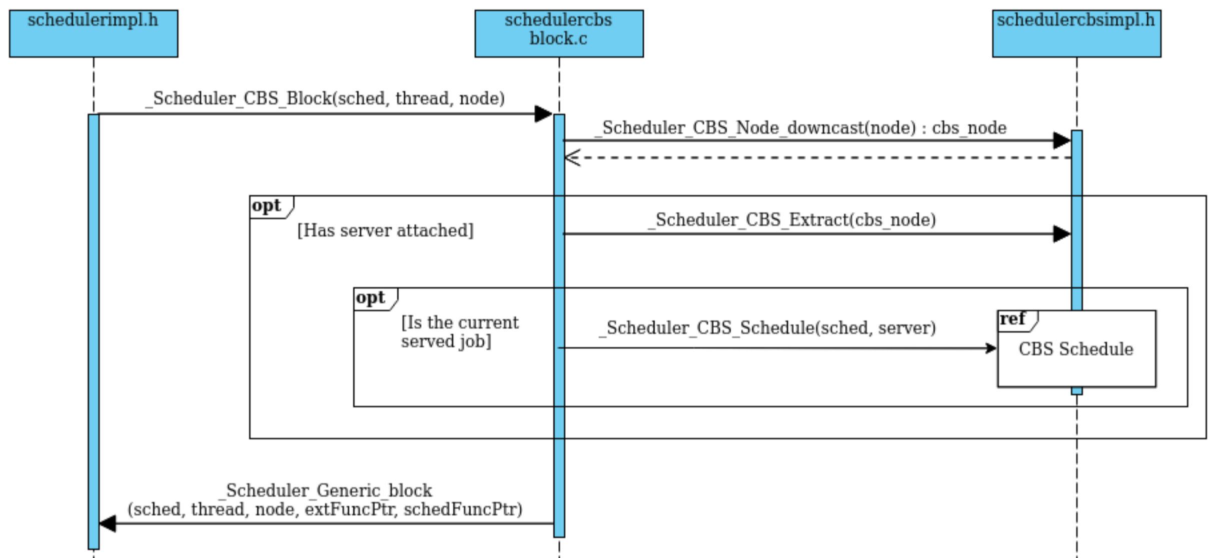

Figure 5.5: CBS block

The schedule operation is a simple mechanism invoked when a served job is blocked, completed or can not be served (not heir in the expired release case) where the scheduler retrieves the heir and performs the update heir operation, see Figure 5.6.

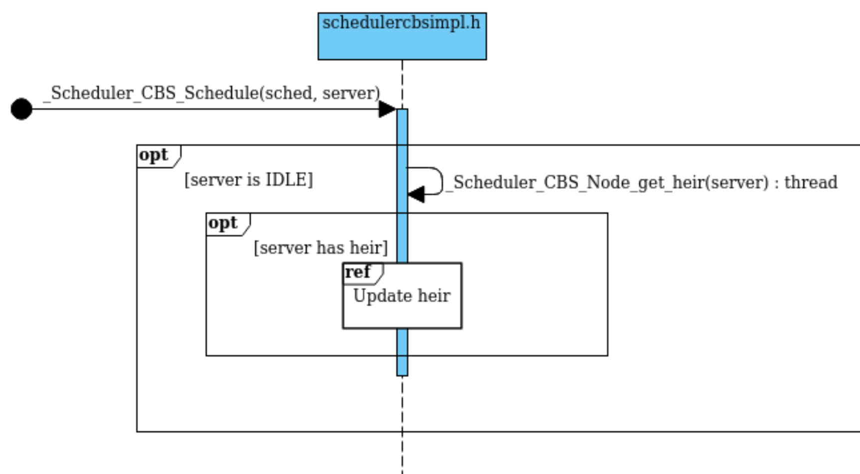

Figure 5.6: CBS schedule

5.1.4 Budget Allocation

As depicted in the flowchart displayed in Figure 5.7, the proposed CBS budget allocation mechanism is essentially responsible for the following operations:

1. Verify if the server needs to be replenished. A CBS replenishment generates a new server's deadline and restores the its capacity, it occurs when the server's capacity is

exhausted $c_s = 0$ or when the deadline has been passed $d_s \leq T$ (given time since boot in ticks);

2. Assignment of the server's current budget as the job time quantum[6];
3. Assign server's deadline as the new job priority and propagate it.

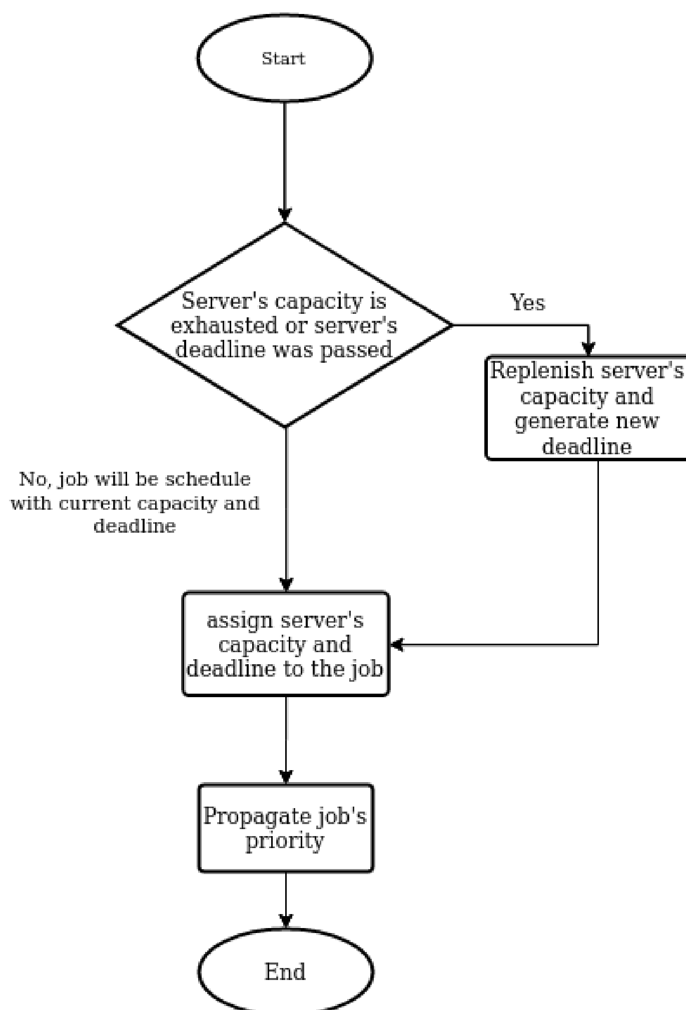

Figure 5.7: CBS budget allocation

5.1.5 Budget Callout

In RTEMS, the basic unit of time is known as a clock tick or simply tick. The tick interval defines the basic resolution of all interval and calendar time operations, and it is defined by the application configuration. At each tick, a tick handler is invoked (see appendix A.3), where if a budget algorithm is employed, the time quantum that the task is able to consume is decreased and, in the case of CBS (*THREAD_CPU_BUDGET_ALGORITHM_CALLOUT* is declared as budget algorithm) a budget callout is invoked when the time quantum is exhausted.

[6] *cpu_time_budget* member in the TCB, defines the maximum interval of time a job can execute

The CBS callout function replenishes the server budget, generates a new server deadline based on the previous deadline, allocates the job time quantum based on the server budget, and finally updates the job priority and propagates it to EDF through the implicit call to the scheduler update priority. See appendix A.4 for further details concerning the CBS callout function.

5.1.6 Example

To provide a thorough understanding of the proposed CBS behavior, Figure 5.8 illustrates a possible execution with the following periodic task set:

1. Server S_i characterized by $(Q_i, T_i) = (3, 8)$;
2. Hard real-time task characterized with WCET, minimum interarrival time and release time: $t_z = (1, 6, 0)$;
3. Soft real time tasks characterized with average execution times, minimum interarrival times and release times: $t_j = (1, 4, 2)$, $t_k = (2, 5, 1)$.

A delay is introduced in the release times to simulate RTEMS behavior when creating periodic tasks (tasks are initially aperiodic and acquire a period and a deadline-driven priority when the period is created through the RM manager and the active release performed).

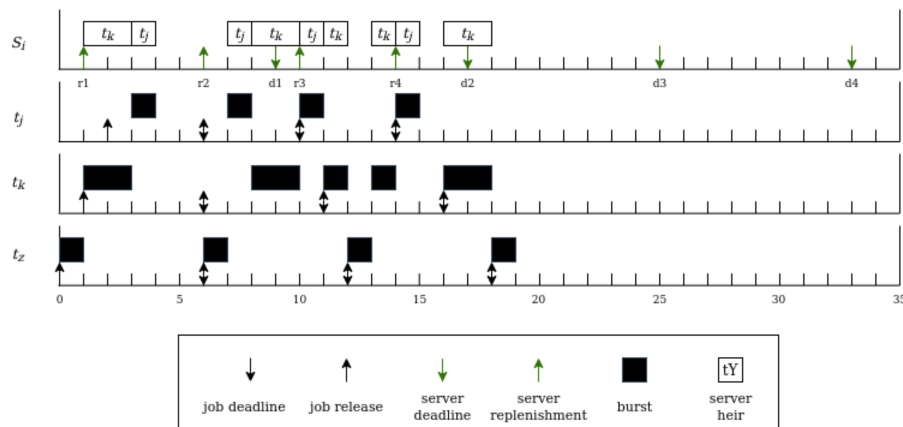

Figure 5.8: CBS scheduling example

At time 4, server S_i has already exhausted all its capacity $c_i = 0$, so when t_j and t_k are released again at time 6, S_i advances its replenishment $c_i = Q_i$ and postpones its absolute deadline $d_{i,s} = d_{i,s-1} + T_i$ (new deadline denoted $d2$ in Figure 5.8) to have budget to serve the demanding jobs, this situation occurs again at time 10 and 14. With this example, it is important to display at time 6 the benefit of employing a deadline ordered red-black tree as job t_j and t_k are released at the same time and t_j will be the first one to be served since it has the shortest job deadline.

Moreover, despite the fact that the hard real-time task t_z has the longest periodicity and consequently deadline (only implicit deadlines are defined in RTEMS) among the task set, soft activities ,t_j and t_k, do not interfere in the completion of t_z since they run under server context (which by designed has a defined period higher than t_z) and due to the early replenishment policy the server deadline is postponed thus reducing the jobs priority and avoiding any interference on the *apriori* guarantee of t_z.

5.2 Capacity Sharing and Stealing

For CSS (Nogueira and Pinho 2007), the goal is the same as the one presented in 5.1, provide QoS for soft real-time activities without jeopardizing any hard real-time guarantee. CSS distinguishes himself from CBS as it enhances its abilities to efficiently handle overloaded servers and reduce the mean tardiness of periodic guaranteed jobs. CSS benefits are achieved by defining two different types of servers (isolated and non-isolated) in which overloaded servers can reclaim and steal capacity from other servers, and finally through the budget recharging suspension until replenishment time.

The proposed solution presents two additional data structures, red-black trees, to manage the set of active server with positive residual capacity and the set of inactive non-isolated servers. These red-black trees are further discussed in subsection 5.2.1 as they are used in the budget allocation mechanism to define the capacity that can be assigned as the job time quantum. Similar to the proposed CBS work, this implementation also employs a red-black tree per server to buffer ready jobs.

The following Subsections present the technical details of some mechanisms implemented for CSS.

5.2.1 Budget Allocation

In the proposed implementation, the CSS budget allocation mechanism is the scheduler core component as it is here that the main contribution of CSS is implemented, the capacity of a overloaded server to acquire extra bandwidth from two additional sources: (i) residual capacity reclaiming, see 5.2.2, when a server performs an early completion; and (ii) capacity stealing, see 5.2.3, from inactive non-isolated servers used to schedule aperiodic jobs on a best-effort manner.

As depicted in Figure 5.9, when a server S_i is selected to schedule a job and performs budget allocation, the scheduler starts by verifying if the replenishment time (equivalent to the deadline $r_i == d_i$) of S_i has been reached, if so, the server budget is recharged and a new deadline is generated. Following the possible replenishment, the scheduler tries to define the job time quantum, for such, it initially performs the residual budget reclaiming to confirm if there is any server on the set of eligible idle active servers (denoted as A_r). If there is, the budget is allocated and the job deadline defined, otherwise the scheduler verifies if it still has capacity $c_i > 0$. If there is none residual capacity from the set A_r and $c_i > 0$, the scheduler uses its own capacity and deadline to execute the job, otherwise, it tries to steal capacity from the set of inactive non-isolated servers (denoted I_s) as last resource.

At last, if there is non eligible servers from the set I_s, the job served by S_i is blocked until S_i replenishment time, moment in which the timer is triggered and the job is unblocked and the budget allocation mechanism performed once again, important to cite that other jobs being released and/or unblocked from the set of tasks attached to this server will not be scheduled but only enqueued into the red-black tree as the server's *current_task* member is assigned to the blocked job (this removes the overhead of having to confirm at every tick if a blocked server reached its replenishment time). If the scheduler successfully defined the time quantum from any of the tree available capacity sources, the job priority is updated based. Appendix A.5 displays the budget allocation listing.

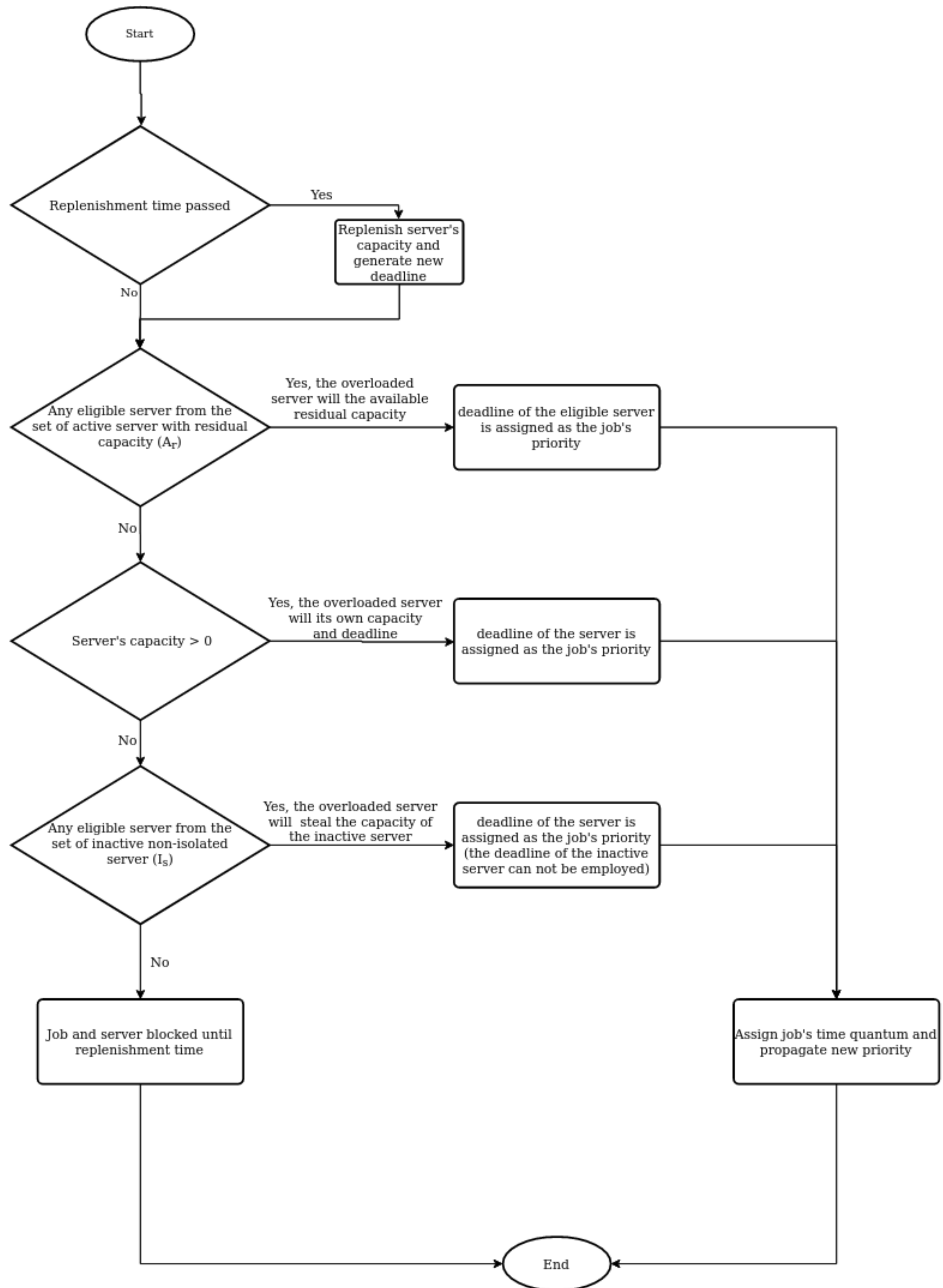

Figure 5.9: CSS budget allocation

5.2.2 Budget Reclaiming

One of the mechanisms that enhances CSS when compared to other reservation based schemes is its capacity to reclaim active server's residual capacity generated from early completions. When a server S_i is performing the budget allocation mechanism, it initially looks in the set of active servers with residual capacity A_r, defined in this implementation

as a red-black tree ordered by deadline and described as $A_r = \{S_r | S_r \in A, d_r > t, c_r > 0\}$, where t is the given time since boot in ticks and A the set of servers defined in the system. As displayed in Figure 5.10, S_i will only reclaim capacity from servers with early completion if in the set A_r exists a server S_r which complies with following rules: (i) has residual capacity and its deadline is still active, if a server does not comply with this rule it is extracted from the red-black tree; (ii) S_r is not currently serving as the accounting server for an active busy server; (iii) S_i has an higher priority (shortest deadline) than the current server retrieved from A_r. Appendix A.6 displays the budget reclaiming listing.

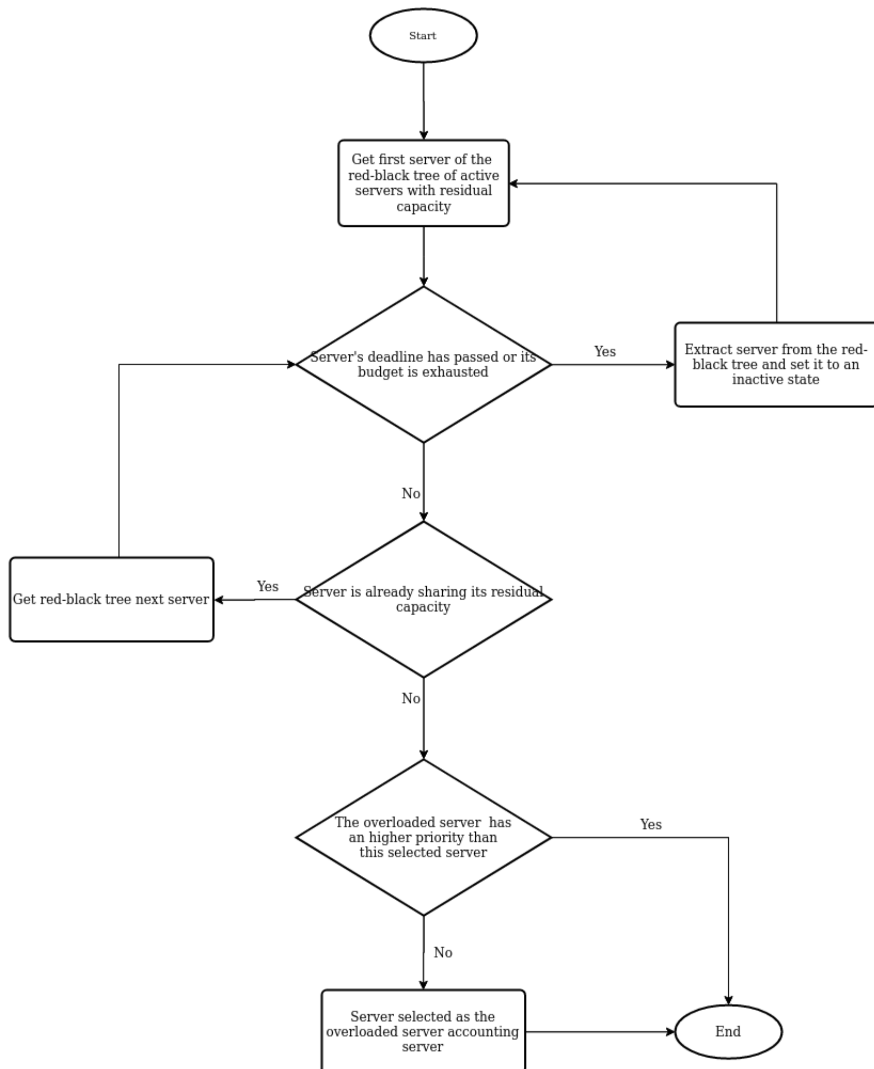

Figure 5.10: CSS budget reclaiming

5.2.3 Budget Stealing

When a server S_i has pending work and there is neither residual capacity from early completion to reclaim nor own capacity available $c_i = 0$, S_i is allowed to steal inactive non-isolated capacity. In RTEMS, the set of inactive non-isolated servers I_s is defined as a red-black tree ordered by deadline where all inactive non-isolated servers are inserted $I_s = \{S_s | S_s \in I, Y_s == 1\}$, I represent the set of inactive servers and Y_s the server's type. As displayed in Figure 5.11, S_i connects to the earliest server eligible in I_s that complies with

the following rules: (i) S_i deadline is shortest (higher priority) than S_s; (ii) S_s is not serving as the accounting server of an active busy server; (iii) if S_s replenishment is required, its new priority shall be lower (longest deadline) than S_i; (iv) shall have positive residual capacity $c_s > 0$. Appendix A.7 displays the budget stealing listing.

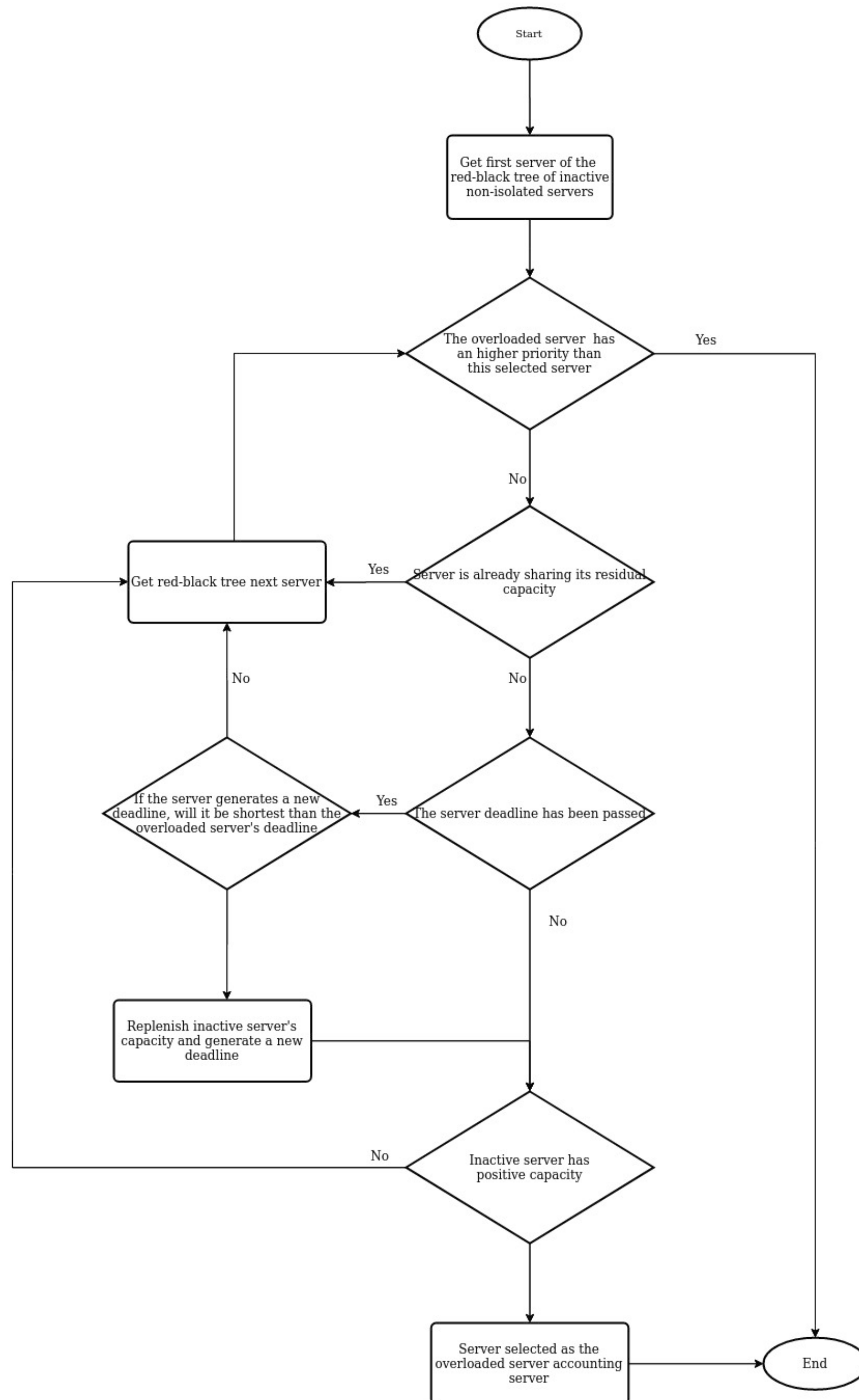

Figure 5.11: CSS budget stealing

5.2.4 Budget Callout

This follows the same principle as 5.1.5, at each tick a function is invoked where the task time quantum is decreased and when it reaches 0 a specified callback is invoked, for CSS, the callback is *_Scheduler_CSS_Budget_callout*. The CSS budget callout function, essentially invokes the budget accounting mechanism (to try to define a new time quantum for the task) and if the server is using an additional source of bandwidth, it manages the accounting server. See CSS budget callout listing in appendix A.8.

5.2.5 Schedule

Similar to CBS, the CSS schedule operation is only invoked when a served job is blocked, completed or can not be served (not heir in the expired release case). When invoked, the schedule operation retrieves the heir and schedules it. If there is no heir, the scheduler tries to insert the server into A_r, if not possible, it changes the server's state to inactive and if its a non-isolated server the scheduler inserts it into I_s.

5.2.6 Release

The CSS release implementation only contrasts from the CBS implementation by performing the server activation on active releases (see active release definition in subsection 5.1.1). The server activation, see listing in appendix A.10 for further details, serves two essential purposes: (i) enable a server that is in an inactive state; and (ii) remove a server from any red-black tree of additional bandwidth (idle active servers with residual budget A_r and inactive non-isolated servers I_s). When removing a server from a red-black tree of additional bandwidth, the scheduler needs to confirm if the server's capacity is currently being shared, if it is, the sharing is stopped and budget allocation performed on the task that just lost its accounting server.

Concerning job releases for servers that are blocked until replenishment time, it is impossible to a job to perform an expired release for blocked servers, and for active releases the job is only inserted in the red-black tree as the server's current task is the heir that has been blocked with the server.

5.2.7 Unblock

The major differences between CSS and CBS unblock operation (see Subsection 5.1.2) are the unblocking of jobs that are waiting for the server's replenishment time and server's activation. When a job that was waiting for the server's replenishment time is unblocked (means that the server's replenishment time has arrived), the update heir operation is called to essentially perform the budget allocation mechanism.

5.2.8 Block

Block operation for CSS behaves like the CBS blocking, the differences lie when a server and job are blocked until the replenishment time. When a server and, consequently, the heir job that is still to be inserted into EDF ready context are blocked until timeout, the CSS blocking mechanism invokes directly the EDF schedule body operation (this allows system threads to be executed in case there is none application task as EDF heir). When the job being blocked until server's replenishment time is EDF heir, the generic block is executed.

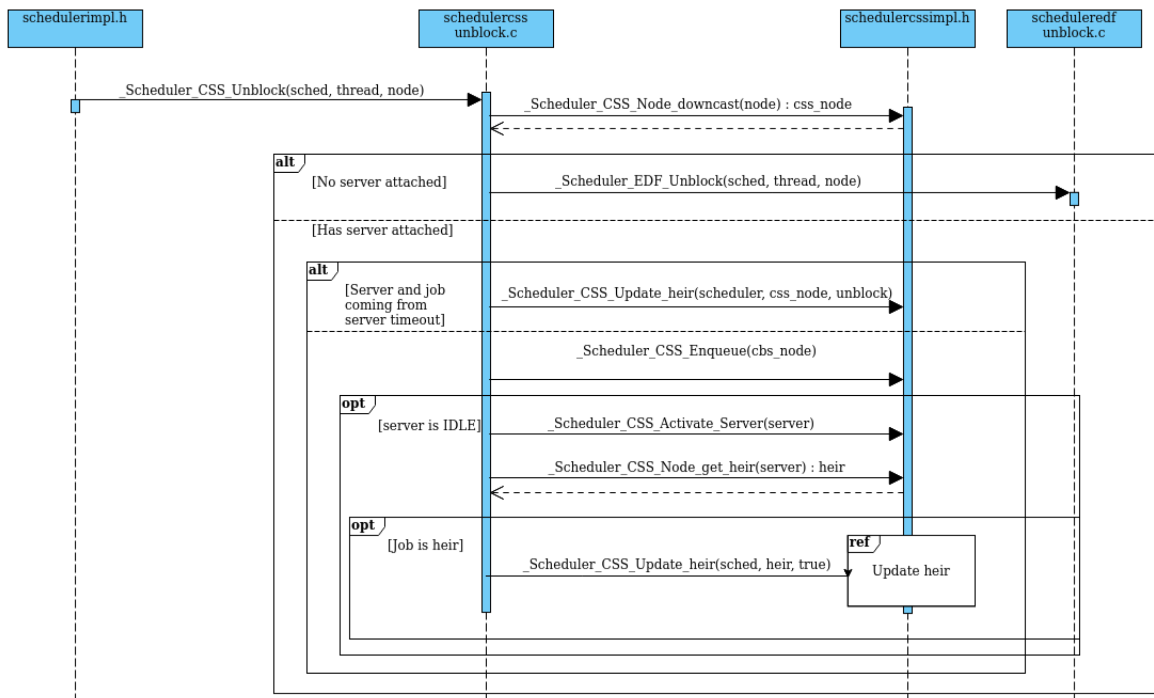

Figure 5.12: CSS unblock

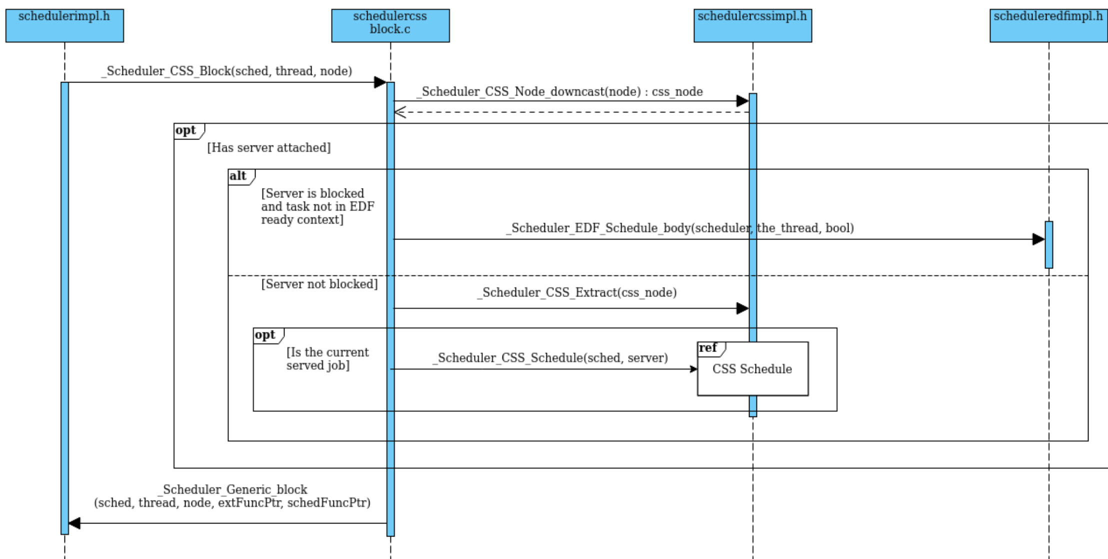

Figure 5.13: CSS block

5.2.9 Example

To fully comprehend CSS behaviour, Figure 5.14 illustrates a possible execution where overloads are handled by either using residual capacities or by stealing capacities of inactive non-isolated servers without postponing deadlines. The task set is the following:

1. Isolated servers characterized by reserved capacity and minimum interarrival times: S_a = (2,9), S_b = (3,9);

2. Non-isolated server characterized with reserved capacity and minimum interarrival times: S_c = (1,4);
3. Soft real-time periodic tasks characterized with average execution times and minimum interarrival times: t_j = (1,7), t_k = (2,7);
4. Hard real-time periodic task characterized with WCET and minimum interarrival time: t_z = (1,6).

A delay is also introduced in these release times to simulate RTEMS behavior when creating periodic tasks.

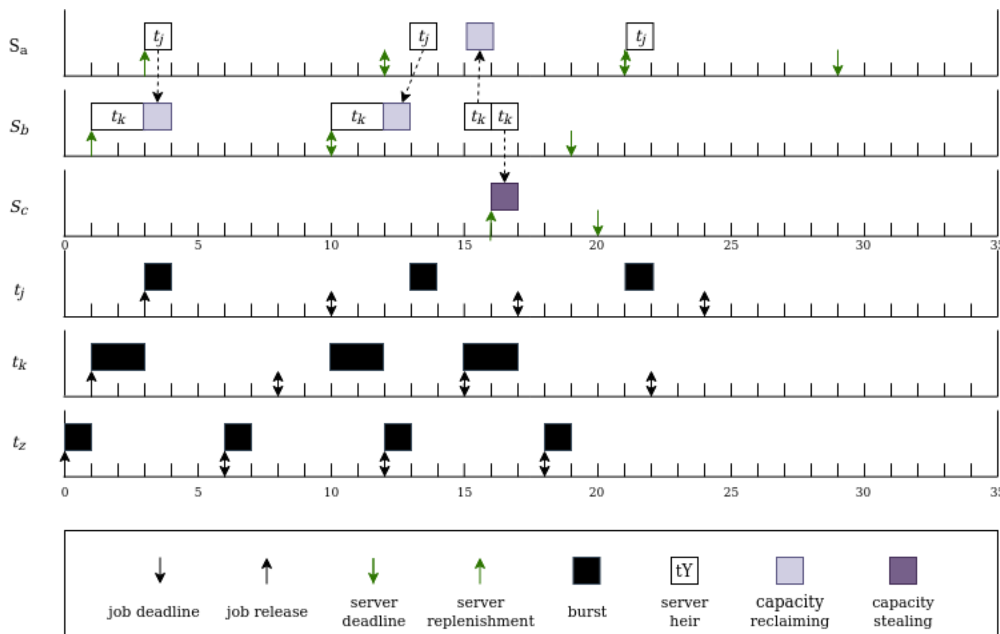

Figure 5.14: CSS scheduling example

At time 3, the server S_a is able to reclaim capacity from S_b to serve the new released job $j_{j,1}$, this available residual capacity originates from the early completion of job $j_{k,1}$. At time 8, a new job released $j_{k,2}$ from task t_k is blocked until the replenishment time of server S_b as there is no available residual capacity, $c_b == 0$ and the rules defined to steal capacity from inactive non-isolated servers are not complied. At time 10, when unblocked, the job $j_{k,2}$ is scheduled with the server's capacity. At time 13, S_a reclaims S_b residual capacity to execute job $j_{j,2}$, when completed, S_a is inserted into the red-black tree of active servers with positive residual capacity ($c_a == 2$). When job $j_{k,3}$ is released, S_b uses the second unit of available capacity to define a time quantum of one unit to the served job (the first unit of S_a residual capacity is decreased at time 15 due to idle CPU). At time 16, the CSS budget callout is invoked to redefine a new time quantum for the served job, as there is no available residual capacity, exhausted capacity $c_b == 0$ and S_b complies with the rules to steal capacity form inactive non-isolated server, S_c is replenished and $j_{k,3}$ is served with S_c capacity.

Chapter 6

Experimentation and Assessment

At the beginning of this dissertation, the performance was to be measured on a real instrumentation cluster with a task set composed of ADAS and communication activities (multimedia tasks would not be implemented due to RTEMS lack of graphical frameworks and support). The use of a cluster would help to analyze the safety increase of this proposed solution and identify any new vulnerability. Unfortunately, when the time came to assess the proposed algorithms, no instrumentation cluster was available, so, the measurement and comparison of both implementations were made on a Raspberry Pi 1 Model B simulating the hard and soft real-time activities with static task sets.

The task set used for simulation purposes is a hybrid task set consisting of the following periodic tasks:

1. Four periodic hard real-time tasks with fixed WCET and minimum interarrival times: $t_a = (8, 80)$, $t_b = (9, 90)$, $t_c = (5, 50)$, $t_d = (10, 100)$. These parameters are fixed to achieve the aimed utilization fraction of $U_p = 0.4$;
2. Dynamic number (up to 5 to 10) of periodic soft real-time tasks with variable mean execution times and minimum interarrival times. These parameters were varied between each test to obtain the desired utilizations $0.4 \leq U_s \leq 0.7$.
3. For CBS, each soft and hard real-time parameters was tested with a dynamic number of servers (up to 3 to 8) and parameters. The variation on the number of servers and, consequently, the number of tasks attached to each server allowed to analyze the performance of ordering the jobs by their deadlines.
4. For CSS, the approach used for CBS was employed. Beside varying the number of isolated servers (up to 3 to 8), each task set was executed with a variation on the number of non-isolated servers (up to 0 to 3) obtain average results for CSS.

For each test application a tick interval of 20000 microseconds was configured. Each task set executed until t= 100000 ticks and was repeated 10 times to ensure the results were consistent. The metrics used to measure the algorithms performance were the mean tardiness and the average deadline misses, computed for all soft tasks, and the average length of context switch operations. The reason for choosing these metrics is motivated by the impact of meeting soft deadlines without jeopardizing hard tasks and to measure the impact of our implementation in terms of overhead when there is the need to compute the additional time required by each algorithm to perform budget allocation to the served jobs.

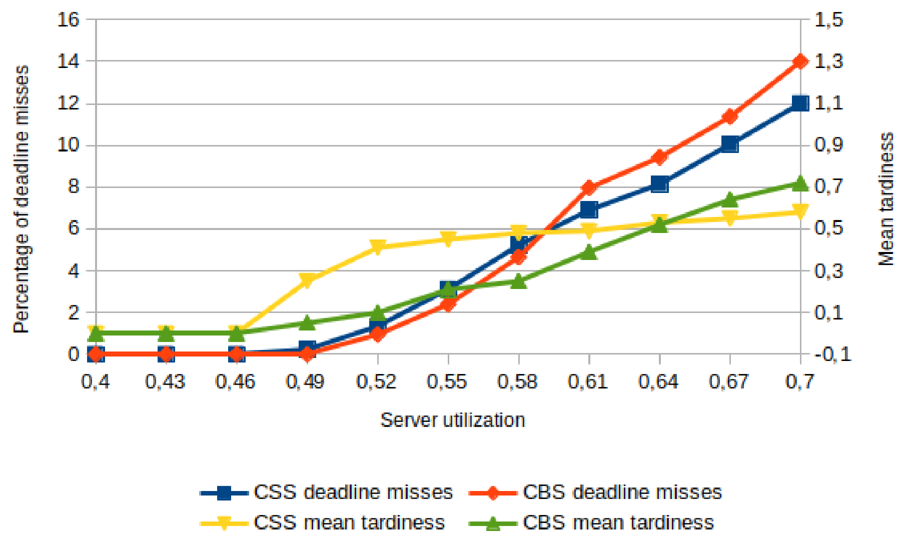

Figure 6.1: CSS vs CBS.

6.1 Tardiness evaluation

Figure 6.1 shows the performance of both schedulers as a function of the system's utilization, measuring the mean tardiness and the average percentage of deadline misses for soft periodic tasks under different utilizations. The results present that for lower soft tasks utilizations CBS as a slightly better performance than CSS and stems from the behaviour of each algorithm when the budget is exhausted. While CSS postpones a task until the server replenishment time, which in the case of no available budget that means that it may lead to a deadline miss, CBS updates its deadline and replenishes the budget providing the maximum reserved capacity to the served job. On extreme cases where soft real-time tasks utilization surpasses 100%, CBS early replenishment leads to a significant increase of deadline misses and mean tardiness when compared to CSS.

6.2 Overhead evaluation

Table 6.1 presents the average context switch length in both schedulers, measuring it for both soft tasks served by servers and hard tasks. These measurements were made on scenarios where the released job would become the new heir in EDF ready red-black tree and, consequently, be executed. The context switch length represents the time difference between the moment in which a job is released and the moment in which EDF updates the heir and forces the dispatch.

Using this metric it is possible to measure the additional overhead created by the budget allocation mechanism of both algorithms and compare it to the overhead of hard real-time tasks. As a result one can see that CSS presents a higher average overhead than CBS (68.01 milliseconds in CSS against 54.84 in CBS). This result is higher in CSS due to the budget allocation mechanism since it looks for remaining capacity in other servers to perform budget accounting.

Table 6.1: Context Switch average length (milliseconds)

Scheduler	Soft jobs	Hard jobs
CSS	68.01	45.04
CBS	54.84	44.45

6.3 Discussion

These results show that CBS performs better than CSS in average cases, CBS performance is worse in extreme cases when the soft real-time utilization surpasses 100%. CSS initial increase in deadline misses and mean tardiness can be explained by the server's replenishment suspension when there is no available capacity (the lack of capacity can derive from the properties of the task set and the capacity stealing defined rules). Additionally, an implementation detail that differs from the original CSS can also justify the initial values. In the proposed implementation, the capacity of all servers inserted in red-black trees is decreased homogeneously since it is dynamically calculated from the moment in which budget accounting is performed from its capacity, while (Nogueira and Pinho 2007) proposes to only decrease the server with the earliest deadline when the CPU is idle. This change ensures system schedulability by guarantying that at time t no server S_s inserted in the set of idle active servers with positive residual capacity and the set of inactive non-isolated servers has a budget that allows a job to overrun the server's deadline: $c_s + t \geq d_s$.

Chapter 7

Conclusions and Future Work

This chapter presents the conclusions of the proposed work and additionally presents the open points that could be implemented in the future.

7.1 Conclusion

This work proposed a solution for mixed-criticality automotive systems, where critical hard real-time tasks coexist with multimedia and communication tasks (considered soft real-time tasks due to their properties) within the same system. The solution's aim was to achieve temporal isolation in a RTOS through the implementation of two reservation-based schemes, namely the Constant Bandwidth Server and the Capacity Sharing and Stealing, where a fraction of the CPU is reserved for soft real-time activities, to ensure QoS for soft activities while guaranteeing critical tasks are not jeopardize. The chosen operating system was RTEMS due to its real-time properties and application in a multitude of domains.

The performance of both algorithms has been compared by measuring the mean tardiness, deadline misses and the context switch overhead. CBS presented better results in average cases, essentially due to its behaviour when the budget is exhausted. In particular, it presents a lower overhead when performing context switch operations as a consequence of its simpler budget allocation mechanism.

Based on the preliminary results, this work could one day be a suitable solution for a considerable number of OEMs as both schedulers reasonably achieved their purpose, temporal isolation (no interference occurred on critical activities from soft real-time tasks). Throughout the performance analysis, it has been guaranteed that no hard real-time deadline miss occurred (as long as the fraction of CPU reserved for hard tasks is no over the theoretical maximum utilization). Despite the security increase, the notable drawback with this proof of concept remains the lack of support and compatibility with multimedia frameworks, the limited board support packages, the effort to rectify these limitations, and essentially the poverty of the results (further experimentation on real instrumentation clusters with a dedicated task set is required to ensure a possible future for this work on the automotive domain).

7.2 Future work

As future work, an engaging work package that emerged from this work would be to implement the reservation-based schemes presented in this work to SMP platforms, and additionally, to increase the number of use cases in which these schemes could be employed,

implement CPU affinity, i.e., a server could be attached to only one or a set of CPUs (can only be executed on the specified set). Furthermore, due to the hazard of not having the possibility to test this proposed solution on a real instrumentation cluster, a future work would be to enhance the task sets and perform the measurements and comparison on a cluster.

Bibliography

Abeni, L. and G. Buttazzo (1998). "Integrating multimedia applications in hard real-time systems". In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pp. 4–13.

Caccamo, M., G. Buttazzo, and Lui Sha (2000). "Capacity sharing for overrun control". In: *Proceedings 21st IEEE Real-Time Systems Symposium*, pp. 295–304.

Deng, Z. and J. W. -. Liu (1997). "Scheduling real-time applications in an open environment". In: *Proceedings Real-Time Systems Symposium*, pp. 308–319.

Deng, Z., J. W. -. Liu, and J. Sun (1997). "A scheme for scheduling hard real-time applications in open system environment". In: *Proceedings Ninth Euromicro Workshop on Real Time Systems*, pp. 191–199.

Dimitrijevic, Katarina (Nov. 2014). "Transgressing Plastic Waste: Designedisposal Strategic Scenarios". In:

Ghazalie, T. and Theodore Baker (July 1995). "Aperiodic servers in a deadline scheduling environment". In: *Real-Time Systems* 9, pp. 31–67. doi: 10.1007/BF01094172.

Holbrook, B. (1999). *Consumer Value: A framework for analysis and research*.

Jeffay, K., D. L. Stone, and F. D. Smith (1992). "Kernel support for live digital audio and video". In: *Computer Communications* 15.6, pp. 388–395.

Jeffay, Kevin and David Bennett (1995). "A rate-based execution abstraction for multimedia computing". In: *Network and Operating Systems Support for Digital Audio and Video*. Ed. by Thomas D. C. Little and Riccardo Gusella. Springer Berlin Heidelberg.

Kambil A., A. Ginsberg and M. Bloch (1997). "Rethinking Value Propositions". In:

Kaneko, H. et al. (1996). "Integrated scheduling of multimedia and hard real-time tasks". In: *17th IEEE Real-Time Systems Symposium*, pp. 206–217.

Koen, Peter et al. (Mar. 2001). "Providing Clarity and Common Language to the Fuzzy Front End". In: *Research-Technology Management* 44, pp. 46–55.

Krogh, G., K. Ichijo, and I. Nonaka (Jan. 2000). *Enabling Knowledge Creation: How To Unlock the Mystery of Tacit Knowledge and Release the Power of Innovation*. doi: 10.1093/acprof:oso/9780195126167.001.0001.

Mercer, C. W., S. Savage, and H. Tokuda (1993). "Processor capacity reserves for multimedia operating systems". In:

Mercer, Savage, and Tokuda (1994). "Processor capacity reserves: operating system support for multimedia applications". In: *1994 Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pp. 90–99.

Nicolas, Navet and Simonot-Lion Françoise (2009). *Automotive Embedded Systems Handbook*.

Nogueira, L. and L. M. Pinho (2007). "Capacity Sharing and Stealing in Dynamic Server-based Real-Time Systems". In: *International Parallel and Distributed Processing Symposium*.

RTEMS Classic API Guide (n.d.). url: https://docs.rtems.org/branches/master/c-user/.

RTEMS SMP Qualification (n.d.). url: https://rtems-qual.io.esa.int/.

S. Nicola, E. Ferreira and J. Ferreira (2012). “A NOVEL FRAMEWORK FOR MODELING VALUE FOR THE CUSTOMER, AN ESSAY ON NEGOTIATION”. In:

Sha, Lui et al. (Nov. 2004). “Real Time Scheduling Theory: A Historical Perspective”. In: *Real-Time Systems* 28, pp. 101–155.

Spuri and Buttazzo (1994). “Efficient aperiodic service under earliest deadline scheduling”. In: *1994 Proceedings Real-Time Systems Symposium*, pp. 2–11.

Stankovic, John A., Krithi Ramamritham, and Marco Spuri (1998). *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Kluwer Academic Publishers. isbn: 0792382692.

Stefan, Petters M. (2008). *Real-Time Systems*. url: https://www.cse.unsw.edu.au/~cs9242/08/lectures/09-realtimex2.pdf.

Woodall, Tony (Jan. 2003). “Conceptualising ’Value for the Customer’: An Attributional, Structural and Dispositional Analysis”. In: *Academy of Marketing Science Review* 12.

Zeithaml, Valarie (July 1988). “Consumer Perceptions of Price, Quality and Value: A Means-End Model and Synthesis of Evidence”. In: *Journal of Marketing* 52, pp. 2–22. doi: 10.1177/002224298805200302.

Appendix A

Appendices

A.1 CBS Structures

```
/**
 * This structure represents a time server.
 */
typedef struct {
  /**
   * @brief ID of current served task.
   */
  rtems_id                    current_task_id;

  /**
   * @brief Server paramenters.
   */
  Scheduler_CBS_Parameters    parameters;

  /**
   * @brief Server deadline unmapped.
   */
  Priority_Control            priority;

  /**
   * @brief Server remaining budget.
   */
  time_t                      remaining_budget;

  /**
   * @brief Scheduler basic context.
   */
  Scheduler_CBS_Server_Context  context;

  /**
   * @brief Indicates if this CBS server is initialized.
   *
   * @see _Scheduler_CBS_Create_server() and
    _Scheduler_CBS_Destroy_server().
   */
  bool                        initialized;
} Scheduler_CBS_Server;
```

Listing A.1: CBS server

```c
typedef struct {
  /**
   * @brief Control tree for the server assigned tasks.
   */
  RBTree_Control ready;
} Scheduler_CBS_Server_Context;
```

Listing A.2: CBS server context

```c
/**
 * This structure handles CBS specific data of a thread.
 */
typedef struct {
  /**
   * @brief EDF scheduler specific data of a task.
   */
  Scheduler_EDF_Node            Base;

  /**
   * @brief CBS server.
   */
  Scheduler_CBS_Server         *Server;

  /**
   * @brief Task RBtree node related to the server context.
   */
  RBTree_Node                   Node;

  /**
   * @brief Task raw deadline - key.
   */
  Priority_Control              task_priority;

  /**
   * @brief Node employed to propagate thread priority
   */
  Priority_Node                *deadline_node;
} Scheduler_CBS_Node;
```

Listing A.3: CBS scheduler node

A.2 CBS Release

```
void _Scheduler_CBS_Release_job(
  const Scheduler_Control *scheduler,
  Thread_Control          *the_thread,
  Priority_Node           *priority_node,
  uint64_t                 deadline,
  Thread_queue_Context    *queue_context
)
{
  Scheduler_CBS_Node   *node;
  Scheduler_CBS_Node   *heir;
  Scheduler_CBS_Server *serv_info;

  node = _Scheduler_CBS_Thread_get_node( the_thread );
  serv_info = node->Server;
  node->deadline_node = priority_node;

  if ( serv_info != NULL ) {
    node->task_priority = deadline;
    /* Currently running thread releases job in server context -
   enqueuing and accounting must be performed */
    if ( _Thread_Is_ready( the_thread ) ) {
      if ( serv_info->current_task_id == the_thread->Object.id ) {
        _Scheduler_CBS_Expired_release( scheduler, node, the_thread );
        return;
      }

      _Scheduler_CBS_Enqueue( node );

      heir = _Scheduler_CBS_Node_get_heir( serv_info );
      _Assert( heir != NULL );

      if ( serv_info->current_task_id == -1 && heir == node ) {
        _Scheduler_CBS_Update_heir( scheduler, heir, false );
      }
    }
  } else {
  /* Hard tasks are directly scheduled by EDF */
    _Scheduler_EDF_Release_job(
      scheduler,
      the_thread,
      priority_node,
      deadline,
      queue_context
    );
  }
}
```

Listing A.4: CBS release

A.3 Tick Entry

```
void _Scheduler_default_Tick(
  const Scheduler_Control *scheduler,
  Thread_Control          *executing
)
{
  (void) scheduler;

  /*
   *  If the thread is not preemptible or is not ready, then
   *  just return.
   */

  if ( !executing->is_preemptible )
    return;

  if ( !_States_Is_ready( executing->current_state ) )
    return;

  /*
   *  The cpu budget algorithm determines what happens next.
   */

  switch ( executing->budget_algorithm ) {
    case THREAD_CPU_BUDGET_ALGORITHM_NONE:
      break;

    case THREAD_CPU_BUDGET_ALGORITHM_RESET_TIMESLICE:
    #if defined(RTEMS_SCORE_THREAD_ENABLE_EXHAUST_TIMESLICE)
      case THREAD_CPU_BUDGET_ALGORITHM_EXHAUST_TIMESLICE:
    #endif
      if ( (int)(--executing->cpu_time_budget) <= 0 ) {

        /*
         *  A yield performs the ready chain mechanics needed when
         *  resetting a timeslice.  If no other thread's are ready
         *  at the priority of the currently executing thread, then the
         *  executing thread's timeslice is reset.  Otherwise, the
         *  currently executing thread is placed at the rear of the
         *  FIFO for this priority and a new heir is selected.
         */
        _Thread_Yield( executing );
        executing->cpu_time_budget =
          rtems_configuration_get_ticks_per_timeslice();
      }
      break;

    #if defined(RTEMS_SCORE_THREAD_ENABLE_SCHEDULER_CALLOUT)
      case THREAD_CPU_BUDGET_ALGORITHM_CALLOUT:
  if ( --executing->cpu_time_budget == 0 )
    (*executing->budget_callout)( executing );
  break;
    #endif
  }
}
```

Listing A.5: Tick entry

A.4 CBS Budget Callout

```
1  void _Scheduler_CBS_Budget_callout(
2    Thread_Control *the_thread
3  )
4  {
5    Scheduler_CBS_Node        *node;
6    Thread_queue_Context       queue_context;
7
8    node = _Scheduler_CBS_Thread_get_node( the_thread );
9
10   _Scheduler_CBS_replanish( node->Server, false );
11   the_thread->cpu_time_budget = node->Server->remaining_budget;
12
13   _Priority_Node_set_priority(
14     node->deadline_node,
15     SCHEDULER_PRIORITY_MAP( node->Server->priority )
16   );
17
18   if ( _Priority_Node_is_active( node->deadline_node ) ) {
19     _Thread_Priority_changed(
20       the_thread,
21       node->deadline_node,
22       false,
23       &queue_context
24     );
25   } else {
26     _Thread_Priority_add( the_thread, node->deadline_node, &
     queue_context );
27   }
28 }
```

Listing A.6: CBS budget callout

A.5 CSS Budget Allocation

```
RTEMS_INLINE_ROUTINE bool _Scheduler_CSS_Schedule_Body(
  Thread_Control      *the_thread ,
  Scheduler_CSS_Node *node
)
{
  Scheduler_CSS_Server *server;
  Per_CPU_Control       *cpu_self;
  Priority_Control       priority;
  Thread_queue_Context  queue_context;
  Watchdog_Interval      interval;

  server = node->Server;
  server->current_task = the_thread->Object.id;

  if ( server->deadline <= _Watchdog_Ticks_since_boot ) {
    _Scheduler_CSS_Replenish_server( server );
    server->share_data.recharged_time = _Watchdog_Ticks_since_boot;
  }

  int32_t rem_budget = _Scheduler_CSS_get_budget( server );

  if ( _Scheduler_CSS_budget_reclaiming( server ) ) {
    priority = server->server_accounting->deadline;
  } else if ( rem_budget > 0 ) {
    server->server_accounting = server;
    priority = server->server_accounting->deadline;
  } else if ( _Scheduler_CSS_budget_stealing( server ) ) {
    priority = server->deadline;
  } else {
    /** Can not be scheduled, must wait for replenhsiment. */
    cpu_self = _Thread_Get_CPU( the_thread );
    interval = server->deadline - _Watchdog_Ticks_since_boot;
    _Thread_Set_state( the_thread, STATES_BLOCKED_UNTIL_RC );
    _Thread_Wait_flags_set( the_thread, THREAD_WAIT_STATE_BLOCKED );
    _Thread_Add_timeout_ticks( the_thread, cpu_self, interval );
    return false;
  }

  the_thread->cpu_time_budget = _Scheduler_CSS_get_budget( server->
    server_accounting );

  _Priority_Node_set_priority(
    node->deadline_node,
    SCHEDULER_PRIORITY_MAP( priority )
  );

  if ( _Priority_Node_is_active( node->deadline_node ) ) {
    _Thread_Priority_changed(
      the_thread,
      node->deadline_node,
      false,
      &queue_context
    );
  } else {
    _Thread_Priority_add( the_thread, node->deadline_node, &
    queue_context );
  }

```

```
    return true;
}
```

Listing A.7: CSS budget allocation

A.6 CSS Budget Reclaiming

```
RTEMS_INLINE_ROUTINE bool _Scheduler_CSS_budget_reclaiming(
  Scheduler_CSS_Server *the_server
)
{
  RBTree_Node          *next;
  Scheduler_CSS_Server *server;

  next = _RBTree_Minimum( &_Active_Servers );

  while ( next != NULL ) {
    server = RTEMS_CONTAINER_OF( next, Scheduler_CSS_Server, Node );
    int32_t remaining_budget = _Scheduler_CSS_get_budget( server );

    if ( server->deadline <= _Watchdog_Ticks_since_boot ||
   remaining_budget <= 0 ) {
      _RBTree_Extract( &_Active_Servers, &server->Node );
      _Scheduler_CSS_set_state( server, false );
      next = _RBTree_Minimum( &_Active_Servers );
      continue;
    }

    if ( server->share_data.serving_task != -1 ) {
      next = _RBTree_Successor( next );
      continue;
    }

    if ( server->deadline > the_server->deadline ) {
      break;
    }

    server->share_data.serving_task = the_server->current_task;
    the_server->server_accounting = server;
    return true;
  };

  return false;
}
```

Listing A.8: CSS budget reclaiming

A.7 CSS Budget Stealing

```
RTEMS_INLINE_ROUTINE bool _Scheduler_CSS_budget_stealing(
  Scheduler_CSS_Server *the_server
)
{
  RBTree_Node          *next;
  Scheduler_CSS_Server *server;

  next = _RBTree_Minimum( &_Inactive_NI_Servers );

  while ( next != NULL ) {
    server = RTEMS_CONTAINER_OF( next, Scheduler_CSS_Server, Node );

    if ( server->deadline > the_server->deadline )
      break;

    if ( server->share_data.serving_task != -1 ) {
      next = _RBTree_Successor( next );
      continue;
    }

    if ( server->deadline <= _Watchdog_Ticks_since_boot ) {
      if ( (_Watchdog_Ticks_since_boot + server->parameters.deadline) >
    the_server->deadline ) {
        next = _RBTree_Successor( next );
        continue;
      }
      _Scheduler_CSS_Replenish_server( server );
      server->share_data.recharged_time = _Watchdog_Ticks_since_boot;
    }

    int32_t remaining_budget= _Scheduler_CSS_get_budget( server );
    if ( remaining_budget <= 0 ) {
      next = _RBTree_Successor( next );
      continue;
    }

    server->share_data.serving_task = the_server->current_task;
    the_server->server_accounting = server;
    return true;
  };

  return false;
}
```

Listing A.9: CSS budget stealing

A.8 CSS Budget Callout

```
void _Scheduler_CSS_Budget_callout(
  Thread_Control *the_thread
)
{
  Scheduler_CSS_Node    *node;
  Scheduler_CSS_Server *server;
  Scheduler_CSS_Server *accounting_server;

  node = _Scheduler_CSS_Thread_get_node( the_thread );
  server = node->Server;
  accounting_server = server->server_accounting;

  if ( accounting_server != server ) {
    accounting_server->share_data.serving_task = -1;

    if ( accounting_server->state ) {
      _RBTree_Extract( &_Active_Servers, &accounting_server->Node );
      _Scheduler_CSS_set_state( accounting_server, false );

      if ( accounting_server->type == 1 ) {
        _RBTree_Insert_inline(
          &_Inactive_NI_Servers,
          &accounting_server->Node,
          &accounting_server->deadline,
          _Scheduler_CSS_Server_Priority_less_equal
        );
      }
    }
  }

  server->server_accounting = NULL;
  _Scheduler_CSS_Schedule_Body( the_thread, node );
}
```

Listing A.10: CSS budget callout

A.9 CSS Schedule

```
RTEMS_INLINE_ROUTINE void _Scheduler_CSS_Schedule(
  Scheduler_Control     *scheduler,
  Scheduler_CSS_Server *server
)
{
  Scheduler_CSS_Node *heir;

  if ( server->current_task != -1 ) {
    return;
  }

  heir = _Scheduler_CSS_Node_get_heir( server );
  if( heir != NULL ) {
    /** Server has heir to schedule. */
    _Scheduler_CSS_Update_heir( scheduler, heir, true );
  } else {
    /** No heir. */
    int32_t remaining_budget = _Scheduler_CSS_get_budget( server );
    if ( server->deadline > _Watchdog_Ticks_since_boot &&
   remaining_budget > 0) {
      /** Server will be inserted into the active server rbt for budget
   reclaiming. */
      _RBTree_Insert_inline(
        &_Active_Servers,
        &server->Node,
        &server->deadline,
        _Scheduler_CSS_Server_Priority_less_equal
      );
    } else {
      _Scheduler_CSS_set_state( server, false );
      if ( server->type == 1 ) {
        /** Server is a NI Server and will be inserted into NIRBtree. */
        _RBTree_Insert_inline(
          &_Inactive_NI_Servers,
          &server->Node,
          &server->deadline,
          _Scheduler_CSS_Server_Priority_less_equal
        );
      }
    }
  }
}
```

Listing A.11: CSS schedule

A.10 CSS Server Activation

```
RTEMS_INLINE_ROUTINE void _Scheduler_CSS_Stop_budget_sharing(
  Scheduler_CSS_Server *server
)
{
  Scheduler_CSS_Node *node;
  Thread_Control     *the_thread;
  ISR_lock_Context    lock_context;

  if ( server->share_data.serving_task == -1 )
    return;

  the_thread = _Thread_Get( server->share_data.serving_task, &
   lock_context);
  if ( the_thread == NULL )
    return;

  _ISR_lock_ISR_enable( &lock_context );
  node = _Scheduler_CSS_Thread_get_node( the_thread );
  node->Server->server_accounting = NULL;
  server->share_data.serving_task = -1;
  _Scheduler_CSS_Schedule_Body( the_thread, node );
}

RTEMS_INLINE_ROUTINE void _Scheduler_CSS_Activate_Server(
  Scheduler_CSS_Server *server
)
{
  if ( server->state && !_RBTree_Is_node_off_tree( &server->Node ) {
    _RBTree_Extract( &_Active_Servers, &server->Node );
    _Scheduler_CSS_Stop_budget_sharing( server );
  } else if ( !server->state ) {
    _Scheduler_CSS_set_state( server, true );
    if ( server->type == 1 ) {
      _RBTree_Extract( &_Inactive_NI_Servers, &server->Node );
      _Scheduler_CSS_Stop_budget_sharing( server );
    }
  }
}
```

Listing A.12: CSS server activation