Software Development of Reconfigurable Real-time Systems: From Specification to Implementation

Dissertation
zur Erlangung des Grades des Doktors der Ingenieurwissenschaften der Naturwissenschaftlich-Technischen Fakultät der Universität des Saarlandes

und der

Tunisia Polytechnic School, Carthage University

von

Wafa Lakhdhar

Saarbrücken 2021

Tag des Kolloquiums: 12. Juli 2021
Dekan: Prof. Dr. Jörn Walter
Vorsitz: Prof. Dr. Kathrin Flaßkamp
Berichterstatter: Prof. Dr. Georg Frey
Prof. Dr. Mohamed Khalgui
Prof. Dr. Luis Gomez
Akademischer Beisitzer: Dr. Paul Motzki

ABSTRACT

Real-time systems run under real-time constraints that determine their reliability and accuracy. Besides, real-time constraints reconfigurable real-time systems induce additional constraints which are constraints for reconfiguration. This thesis deals with reconfigurable real-time systems in mono-core and multi-core architectures. The development of these systems faces various challenges particularly in terms of reconfiguration capacity, real-time aspects, and resource constraints. In addition, the costs of those systems development are strongly impacted by wrong design choices made in the early stages of development. The focus in this thesis is on providing guidelines, methods, and tools for the synthesis of feasible reconfigurable real-time systems in mono-core and multi-core architectures. To address the given challenges, we propose in this work a new strategy of 1) function assignment 2) placement and scheduling of tasks to execute real-time applications on mono-core and multi-core architectures, 3) optimization step based on Mixed integer linear programming (MILP), and 4) a decision aiding solution (i.e., guidance tool) that assists designers to implement a feasible multi-core reconfigurable real-time from specification level to implementation level. We apply and simulate our contributions to a case study and compare the proposed results with related works.

Keywords: Real-time system, Reconfiguration, Scheduling, Mono-Core, Multi-Core, MILP.

KURZZUSAMMENFASSUNG

Echtzeitsysteme laufen unter harten Bedingungen an ihre Ausführungszeit. Die Einhaltung der Echtzeit-Bedingungen bestimmt die Zuverlässigkeit und Genauigkeit dieser Systeme. Neben den Echtzeit-Bedingungen müssen rekonfigurierbare Echtzeitsysteme zusätzliche Rekonfigurations-Bedingungen erfüllen. Diese Arbeit beschäftigt sich mit rekonfigurierbaren Echtzeitsystemen in Mono- und Multicore-Architekturen. An die Entwicklung dieser Systeme sind verschiedene Anforderungen gestellt. Insbesondere muss die Rekonfigurierbarkeit beachtet werden. Dabei sind aber Echtzeit-Bedingungen und Ressourcenbeschränkungen weiterhin zu beachten. Darüber hinaus werden die Kosten für die Entwicklung dieser Systeme insbesondere durch falsche Designentscheidungen in den frühen Phasen der Entwicklung stark beeinträchtigt. Das Hauptziel in dieser Arbeit liegt deshalb auf der Bereitstellung von Handlungsempfehlungen, Methoden und Werkzeugen für die zielgerichtete Entwicklung von realisierbaren rekonfigurierbaren Echtzeitsystemen in Mono- und Multicore-Architekturen. Um diese Herausforderungen zu adressieren wird eine neue Strategie vorgeschlagen, die 1) die Funktionsallokation, 2) die Platzierung und das Scheduling von Tasks, 3) einen Optimierungsschritt auf der Basis von Mixed Integer Linear Programming (MILP) und 4) eine entscheidungsunterstützende Lösung umfasst, die den Designern hilft, eine realisierbare rekonfigurierbare Echtzeitlösung von der Spezifikationsebene bis zur Implementierungsebene zu entwickeln. Die vorgeschlagene Methodik wird auf eine Fallstudie angewendet und mit verwandten Arbeiten verglichen.

Schlüsselwörter: Echtzeitsystem, Rekonfiguration, Scheduling, Mono-Core, Multi-Core, MILP.

RESUME

Cette thèse traite des systèmes en temps réel reconfigurables dans une architecture mono-coeur et multi-coeur. L'objectif principal de cette thèse est de fournir des directives, des méthodes et des outils pour la synthèse de systèmes temps réel reconfigurables réalisables dans des architectures mono-coeur et multi-coeur. Le développement de ces systèmes est confronté à divers défis, notamment en termes de capacité de reconfiguration, d'aspects temps réel et de contraintes de ressources. En outre, les coûts de développement de ces systèmes sont fortement influencés par les mauvais choix de conception faits dans les premières étapes du développement. Pour relever ces défis, nous proposons dans ce travail une nouvelle stratégie de i) attribution de fonctions, ii) placement et planification des tâches pour exécuter des applications en temps réel sur des architectures mono-coeur et multi-coeur, iii) étape d'optimisation basée sur la programmation linéaire (MILP), et iv) une solution d'aide à la décision qui aide les concepteurs à mettre en œuvre un système temps réel multi-coeur reconfigurable du niveau de spécification au niveau de la mise en œuvre. Nous appliquons et simulons nos contributions à une étude de cas, et nous comparons les résultats proposés avec des travaux connexes afin de montrer l'originalité de cette méthodologie.

Mots clés: Système temps réel, Reconfiguration, Ordonnancement, mono-core, multi-core, MILP.

I dedicate this work to My dear parents Abdel Malek and Yamna, My two brothers Zouhair and Amin, My friends and family, for their endless love, support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acronyms

WCET Worst Case Execution Time

WCRT Worst CaseResponse Time

QoS Quality of Service

RM Rate Monotonic

DM Deadline Monotonic

EDF Earliest Deadline First

LLF Least Laxity First

PIP Priority Inheritance Protocol

SRP Stack Resource Policy

PCP Priority Ceiling Protocol

FMLP Flexible Multiprocessor Locking Protocol

MPCP Multiprocessor Priority Ceiling Protocol

DoD Department of Defense

RT-Java Real-Time Java

POSIX Portable Operating System Interface,

MP Mathematical Programming

LP Linear Programming

MILP Mixed Integer Linear Programming

GA Genetic Algorithm

DP Dynamic Programming

DVFS Dynamic Voltage and Frequency Scaling

CCAS Car Collision Avoidance System

AV Autonomous Vehicles

SW Software

HW Hardware

CHAPTER 1

Introduction

1.1 Thesis Context

Real-time systems are in widespread use in many sectors such as automotive electronics, avionics, telecommunications, consumer electronics, etc [111]. A real-time system is a computed environment that must respond to events timely, to ensure the correctness of the system behavior [82]. In real-time systems, the computation results must be delivered within a time bound, called *deadline*. A real-time system is composed by a set of tasks. Each real-time task has an associated deadline that must meet it. In order to verify that no deadline misses occur, designers perform schedulability analysis [126], [13] (.i.e, processor utilization test or response time analysis). These analysis use the Worst Case Execution Time (WCET) [20] and take place in a very pessimistic case.

Real-time systems must constantly be adapted to their environment evolution and provide reconfiguration techniques according to user requirements [82]. A reconfiguration is an operation allowing the system to transform its working process in order to adapt to changes [118]. Based on this definition, a reconfigurable real-time system is considered as a set of implementations where an implementation is the scenario executed by the system in a particular time or under a particular condition. The synthesis of reconfigurable real-time systems is an ongoing research topic, however it is not a trivial task as the predictability is needed to guarantee certain security and safety requirements [118].

Nowadays, almost all reconfigurable real-time systems become more and more complex and they need additional computational power. Thus, the necessity of multi-core technology [8]. The latter gives the same performance of a single faster processor at lower power consumption (by lowering frequency and voltage [16]) through handling more tasks in parallel [31]. The multi-core architectures offer an ideal platform for complex real-time computations and provide an important boost in processing capacity under relatively low power and price [117].

Even if multi-core technology may offer several benefits to reconfigurable real-time sys-

tems, these latter become much harder to implement due to the interferences between tasks when accessing shared resources. Moreover, the energy and real-time constraints become more and more difficult to satisfy owing to the high increasing demand for new functionalities in current and future reconfigurable real-time systems. Therefore, many researchers are moving toward proposing optimization approaches to provide a feasible implementations of such systems while addressing the mentioned challenges (i.e., real-time and reconfiguration constraints, energy, as well as the expansion of functionalities).

The multi-core real-time systems research community has developed a large number of methodologies/tools such as [4], [87], and [71] by proposing approaches ensure an efficient synthesis. In fact, the implementation multi-core real-time systems under reconfiguration constraints comprises three levels i) specification level in which the designer specifies the functional and non-functional properties of a system, ii) design level in which the designer models the functional requirements at higher abstraction level with the aim to satisfy the non-functional ones note that the analysis may be performed at this level to verify the non-functional requirements , and iii) implementation level in which the design model is transformed into code. In the last level, programming real-time system is generally considered the most difficult kind of programming. So that, real-time languages have been designed to facilitate the coding task [25]. POSIX [55], RT-Java [98], and Ada [89] are considered the most real-time languages being used for programming real-time system [25]. Figure 1.1, sum up the process of the synthesis of reconfigurable real-time systems which involves as we mentioned above three level: i) specification level in which, the designer specifies its specification model [29] (function set, core set (in the case of multi-core architecture), etc.), ii) design level [19], which is composed of the main steps involved in the synthesis process (e.g., function to task assignment, task to core partitioning in the case of multi-core architecture, scheduling, optimizing, real-time and reconfiguration constraints, and guidance to help designer to make good decisions.), and iii) implementation level [3] which deals with the final code implementation.

1.2 Problematic

As outlined in the previous subsection, due to the complexity of modern real-time systems the computing requirement increases as a result those systems may be specified as a large number of scenarios and time constrained functionalities, thus

- the implementation of these systems often consists of a huge number of implementations and tasks. Switching from an implementation to another generates an important reconfiguration time,

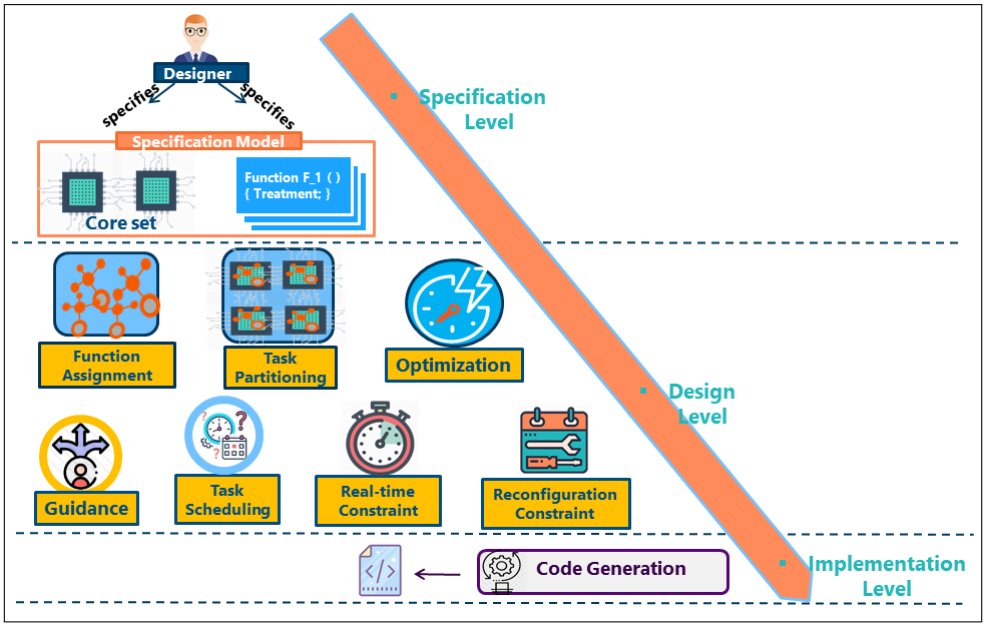

Fig. 1.1 Reconfigurable Real-tme Systems Synthesis Process.

- an important time overhead may be induced due to the large number of tasks as a result it may increase the response time as well as the energy consumption,
- many redundancies may be induced due to the large number of tasks and implementations which produces a complex system code,
- the system stability is affected by the moving time (i.e., the time spent to migrate from a core to another when switching from one implementation to another [68]), the more the latter increases, the more stability will be impaired,
- many conflicts may occur between dependent tasks which increase the blocking time,
- carrying out certain design steps (i.e., how to assign functions to tasks or how to partition or schedule tasks) is not a trivial task. Also, it is not easy to make a decision in the case of non-feasible system. In fact, to look for a feasible solution, various decisions may be taken related to designer experience. This time-consuming approach may extend development time and increase thereby the time to market.

1.3 Contributions

The main research contributions of this thesis are as follows.

- A reconfigurable real-time models in both mono-core and multi-core architecture that describe how a real-time component(function, implementation, task, thread, core, etc.),

supporting various aspects and hidden information, could strongly be designed and implemented as well.

- In order to overcome the time overhead, reconfiguration time, response time and energy increase as well as system code complexity, we propose an approach which addresses initially the mono-core architecture (i.e., first contribution). It performs

  - function to task assignment by proposing a technique to minimize the number of tasks while preserving timing constraints and schedulability.
  - energy-aware optimization solution, a minimized response time solution, while reducing task number solution as a result, it allows to minimize the code complexity, the redundancy between implementation set, the context switch, and the preemption.

- The second contribution which is defined by a set of combinatorial optimization solutions based on integer programming have been developed.

  - For an efficient partitioning task to core, a Mixed Integer Linear Program is proposed. It aims to compute a feasible partitioning while minimizing the blocking time and /or the moving time [68] which enables to maximize the stability of the system.
  - After obtaining a feasible solution base, we execute a second local optimization based on a mixed integer program in each core. Note that in this step the first contribution is applied.

- A methodological guidance framework named MO^{2}R^{2}S (Multi-Objective Optimization approach for Reconfigurable Systems) that assists designers to implement a feasible mono-core/multi-core reconfigurable real-time system from specification level to implementation level. It takes as inputs, at the specification level, the functions, the reconfiguration conditions, and the core set in the case of multi-core architecture. At the design level, the proposed tool generates i) implementations from the reconfiguration conditions, and ii) tasks and resources from the function sets and their dependencies. Then, the tool tries to assign tasks to cores by executing the first optimization step. If it founds at least one feasible solution, it applies the second optimization (.i.e, first contribution) which aims to minimize either the energy consumption, the response time or the number of tasks. Otherwise, it suggests to modify the hardware architecture by increasing the number of cores or to generate a degraded solution where some soft tasks may miss their deadlines. Finally, at the implementation level, it transforms the task model into POSIX code using transformation rules.

- We apply and simulate the contribution to two case studies, and compare the proposed results with related works in order to show the originality of this methodology.

1.4 Publication

The outcomes of this thesis are published in the hereafter list of publications:

1.4.1 Journals (published):

- **Wafa Lakhdhar**, Rania Mzid, Mohamed Khalgui, Georg Frey, Zhiwu Li, MengChu Zhou: *A guidance framework for synthesis of multi-core reconfigurable real-time systems.* **Inf. Sci. 539: 327-346 (2020), Q1, IF= 5,91.**

- **Wafa Lakhdhar**, Rania Mzid, Mohamed Khalgui, Zhiwu Li, Georg Frey, Abdulrahman Al-Ahmari: *Multiobjective Optimization Approach for a Portable Development of Reconfigurable Real-Time Systems: From Specification to Implementation.* **IEEE Trans. Syst. Man Cybern.Syst. 49(3): 623-637 (2018), Q1, IF= 5.13.**

1.4.2 International Conferences (published)

- **Wafa Lakhdhar**, Rania Mzid, Mohamed Khalgui, Georg Frey: *A New Approach for Optimal Implementation of Multi-core Reconfigurable Real-time Systems.* ENASE 2018: 89-98, **Class B**

- **Wafa Lakhdhar**, Rania Mzid, Mohamed Khalgui, Nicolas Trèves: *MILP-based Approach for Optimal Implementation of Reconfigurable Real-time Systems.* ICSOFT-EA 2016: 330-335, **Class B**

1.4.3 Selected Paper (published)

- **Wafa Lakhdhar**, Rania Mzid, Mohamed Khalgui, Georg Frey: *Portable Synthesis of Multi-core Real-Time Systems with Reconfiguration Constraints.* **ENASE** (Selected Papers) 2018: 165-185.

- **Wafa Lakhdhar**, Rania Mzid, Mohamed Khalgui, Nicolas Trèves: *A New Approach for Automatic Development of Reconfigurable Real-Time Systems*. **ICSOFT** (Selected Papers) 2016: 22-44

1.5 Thesis Outline

In Chapter 1, we present the research context, the problems and the contributions of the thesis. The outline and the publications are also introduced here.

In Chapter 2, we present the state of the art in several areas on which we work throughout this thesis. We recall basic definitions and properties of reconfigurable systems and modeling formalisms. We present the current problems and challenges, highlighting the contributions of this thesis related to the state of the art in the development and optimization of reconfigurable real-time systems in both mono-core architecture and multi-core architecture.

Chapter 3 designs a MILP-based approach for the synthesis of reconfigurable real-time in mono-core architecture by optimizing response time, task number and energy consumption.

In Chapter 4, we address the problem of partitioning tasks into cores by minimizing either the stability of system or the blocking time. We propose a guided tool that assists designers to implement a feasible multi-core reconfigurable real-time systems from specification level to implementation level.

In Chapter 5, we present the visual tool that implements the proposed methodology. Also, two case studies are elaborated. Simulations and different tests will also be presented at the end of this chapter.

In Chapter 6, we show and discuss the results and present the conclusion of the presented work. Future improvements that could enrich the work developed during this dissertation are proposed.

CHAPTER 2

State of the art

Introduction

This chapter is devoted to review several generalities and related work required for understanding this dissertation. Initially, some basic concepts concerning real-time systems are presented, e.g., reconfiguration, task models, sharing resources, scheduling algorithms, schedulability analysis in both mono-core and multi-core architecture, and real-time programming languages. Then, we discuss existing optimization methods in the literature and their applications. The objective of this thesis is to enable the synthesis of feasible reconfigurable real-time systems in both mono-core and multi-core architecture. So that, Section 2.4 is devoted to give an overview about existing approaches and methods in the literature that treat the problem of the synthesis. We conclude this section by a comparative study.

2.1 Reconfigurable Systems

This section presents an overview of the basic concepts in reconfigurable systems. First, it briefly introduces the principles of reconfiguration. Next, it presents the different types of reconfiguration. Finally, it discusses a number of state of the art approaches that are closely related to our work.

2.1.1 Definition

Computing systems are employed to assist humans by performing tasks that require an interaction with the physical world. With the increasing complexity of these systems, their continuous availability has become a critical requirement. Due to the various conditions in which these systems have to operate, modification is unavoidable in them. Thus, designers have to optimize the costs and risks of adapting these systems by respecting real-time constraints. Consequently, reconfiguration has been widely admitted as an essential capability of many systems. In the literature, reconfiguration is defined as the capacity to modify system behavior to adapt to a changing environment by i) adding/removing hardware/software components, ii) modifying logic relations between

the components, or iii) updating particular system data [125]. This adaptation allows the software system to move from a current version to a new one. Based on this definition, we consider the reconfiguration as the transition from one implementation to another under well-defined conditions called reconfiguration conditions. An implementation is the scenario implemented by the system at particular time when the reconfiguration condition is true.

2.1.2 Types of reconfiguration

As we mentioned above, the reconfiguration is attributed in a large range of systems [58] [47]. The classification of reconfiguration arises from three questions: "what" can be altered, "when" the alteration is triggered, and "how" the alteration is executed? In response to i) "what" can be modified, the reconfiguration can affect the software level, the hardware level, and both levels, ii) "when" the adaptation takes place, it is triggered when a change is detected in either environment, system-itself, or combination of them [18], and iii) "how" the configuration is performed, configurations can be either created iii.1) dynamically at run-time in which, the reconfiguration may applied manually (.i.e., by user) or automatically (.i.e., by intelligent agents) [22], or iii.2) statically at the design phase by pre-preparing all possible cases of implementation (.i.e., scenario).

In the literature, a huge amount of contributions aim to the modeling, design and development of reconfigurable real-time systems. In fact there are various studies which deal with software reconfigurations applied to real-time applications. In [126], the authors propose an analysis and control process for reconfigurable manufacturing systems. The work reported in [81] presents a general framework to manage the reconfigurability of manufacturing systems. There are also other works in which reconfiguration affect the hardware level such as [85][2]. In [85], the authors propose a scrubbing strategy and a dynamic reconfiguration mechanism within an FPGA-based system-on-chip, to be applied as reconfigurable processor for space applications. The work reported in [2] presents a reconfiguration design approach of FPGA-based logic controller for electromechanical system. Many researchers from academia and industry adopt reconfiguration in both software and hardware levels. The authors in [22] propose a framework supporting the development of real-time systems that exploit hardware accelerators developed through FPGAs based on hardware and software reconfiguration process. In [78], the authors propose a framework for smart grid modeling and simulation based on multi-agent reconfigurable system. This thesis deals with software and hardware reconfiguration. Despite the importance of the works dealing with both software and hardware reconfiguration, they are not a general solution and cannot synthesize all systems (.i.e., smart grid and FPGA-based systems). Concerning the third classification mentioned above (.i.e., dynamic and static reconfiguration), various studies have addressed

Table 2.1 Thesis Position vis-à-vis Exisiting Reconfiguration Types.

Works	What		When		How	
	SW Level	HW level	Environment	System-itself	Dynamic	Static
[126],[81]	✓	-	✓	✓	-	✓
[85],[2]	✓	✓	✓	-	✓	-
[6],[120]	✓	-	✓	✓	✓	-
Our Thesis	✓	-	-	✓	-	✓

the dynamic reconfiguration such as [6][120]. The reconfiguration in these works aims to actively adjust system behavior according to changed environment or user requirements and it is fault-tolerance. Mostly, the dynamic reconfiguration changes the current configuration of a system not the system. However, works dealing with static reconfiguration such as [17]and [9] aim to alter physical equipment or to integrate new techniques in order to largely improve or modify the original system. In Table 2.1, we place this thesis to the existing reconfiguration types. As seen in Table 2.1, this dissertation deals with static software reconfiguration. In addition, it is interested to configuration that is triggered when a change is detected in system-itself.

2.2 Real-time Systems

A real-time system is a computing system which must respond timely to events [75]. ”Real-time systems are those systems in which the correctness of their execution depends not only on the logical results of computation but also on the time at which the results are produced“ [104]. This means that guaranteeing the response within the imposed timing constraints is the most important goal, when developing real-time systems. According to their criticality real-time systems could be [2][115][116]:

- Soft: in which performance is degraded but not destroyed by failure to meet response time constraints [59],
- Hard: in which consequences of missing a deadline can be catastrophic [95],
- Firm: in which a few missed deadlines will not lead to total failure, it may degrade QoS (e.g, financial forecast systems) [90].

This thesis deals with firm real-time systems.

2.2.1 Real-Time System Structure

In this section we define the basic real-time terminology that is used in the thesis. Figure 2.1 illustrates the generic architecture of a real-time system. The real-time software gets

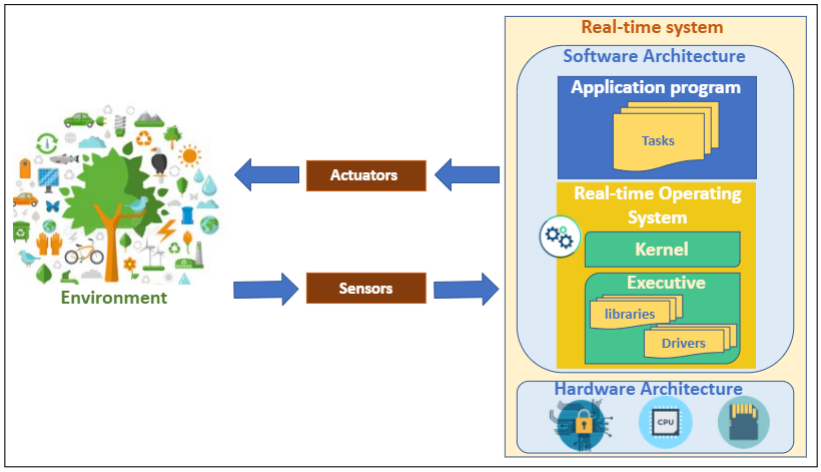

Fig. 2.1 A generic architecture of a real-time system.

information about the environment by the sensors and acts through actuators. A real-time system is composed of a software architecture executing on a hardware platform.

2.2.1.1 Hardware Architecture

The hardware structure presents all the required material resources such as processors, memories, networks, input/output cards, etc. Furthermore, the different types of architectures are related to the interaction between the hardware elements. There are three categories of architecture [15]:

- Mono-processor architecture: all the tasks are executed by a unique processor,
- Multi-processor architecture: all the tasks are divided among different processors sharing a central memory,
- Distributed architecture: all the tasks are distributed over different processors communicating via networks without sharing memory.

Based on number of cores, processors (CPUs) are now split up in three types:

- Mono-core or single core CPU: consists of one core, it can currently only be found on low-power solutions [113],
- Multi-core CPU: consists of two to eight cores which are embedded in the same die [99]. It can execute multiple instructions (See Figure 2.2),
- Many-core CPU: consists of more than eight cores, it designed for a high degree of parallel processing [113].

This dissertation deals with multi-core processors.

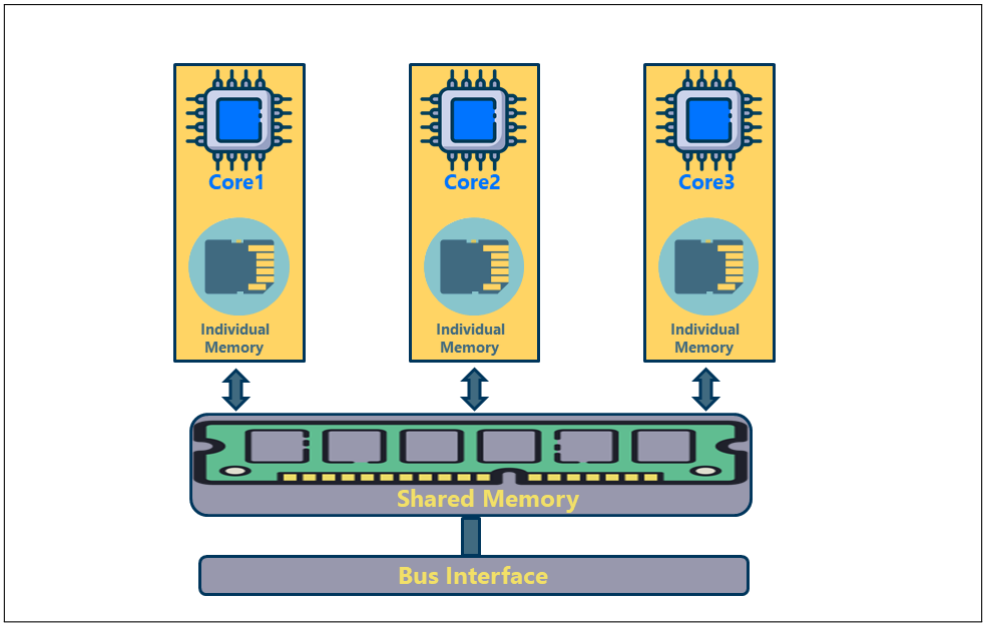

Fig. 2.2 Multicore Processor Architecture.

2.2.1.2 Software Architecture

The software structure consists of a real-time operating system and an application program (see Figure 2.1). The real-time operating system links the application program to the hardware structure. It is composed of [110] i) a real-time kernel which manages the hardware resources access, and the scheduling features, and ii) a executive which consists of a set of drivers modules and libraries facilitating the files management, the communication management. The application program presents the software that runs the system functionalities. It is defined by a set of tasks where each task ensures a sequence of instructions in order to perform a specific treatments.

2.2.2 Real-time Development

The development of real-time systems involves a step of designing. In this level, it is important to introduce the notion of task model as well as the real-time scheduling. One of the basic term used in real-time system theory is that of a task. A task is an abstraction of a piece of software that implements a basic functionality in a real time system [103]. So that a real-time system is defined by a set of control software tasks.

2.2.2.1 Task Model

In order to analyze the feasibility of a real-time system, we need to construct a mathematical model describing the relevant aspects of the system. Let *Sys* be the real-time system. *Sys* is defined by N task τ_i. The characterization of a periodic task τ_i may differ from a scheduling model to another and according to the related nature [72], [97] [100].

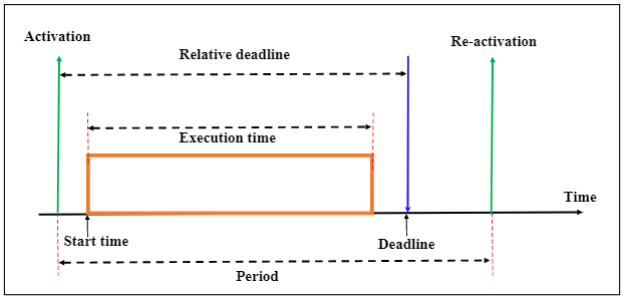

Fig. 2.3 Real-time tasks characterization.

The most used parameters (See Figure 2.3) are: This latter shows a part of the main properties, like:

- r_i (activation or Release time): The date when the task τ_i may start implementing,
- s_i (Start time): The date when the task τ_i starts executing on the processor,
- C_i (execution time): The computing time of task τ_i. This parameter is considered in the majority of work on the real-time scheduling as the worst case execution time (WCET for worst-case execution time) [42] of a task on the processor. The WCET is an upper bound of the execution time that the task can be completed earlier. To be valid, the value of this parameter should not be too overstated and should be never exceeded [43],
- D_i (Deadline): The time devoted to task τ_i to finish its execution,
- T_i (Period): The execution frequency of task τ_i,
- L_i (Laxity): It represents the remaining time before the occurrence of the start date of execution or recovery at the latest [33]. When (Li = 0) is zero at a given time, then, the corresponding task should be strictly performed at this time, uninterrupted otherwise.

2.2.2.2 Tasks Classification

Tasks are classified according to three axes.

Periodic/Aperiodic/Sporadic tasks

Depending on the way of task triggering, real-time tasks are classified as follow [119]:

- periodic tasks which are repeated indefinitely and their instances are separated by a constant period (See Figure 2.4) [121],
- aperiodic tasks which respond to randomly arriving events. They must run at least once and do not repeat necessarily indefinitely, the minimum separation between two consecutive instances can be 0 (See Figure 2.4). In addition, its deadline is expressed as either an average value or is expressed statistically [53],

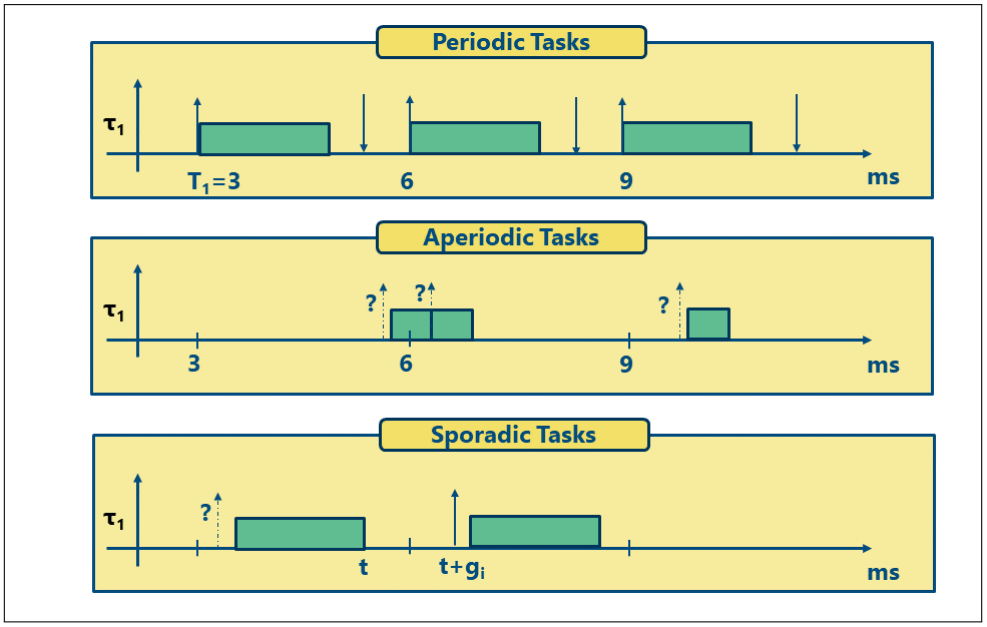

Fig. 2.4 Periodic/Aperiodic/Sporadic Task Characteristics.

- sporadic tasks which present a special case of aperiodic tasks where a minimum time period g_i between two successive activations is fixed implies that once an instance of a sporadic task occurs, the next instance cannot occur before g_i time units have elapsed (See Figure 2.4) [127].

Dependent/Independent tasks

In a typical real time systems, if tasks need to cooperate to complete their missions we call them dependent tasks otherwise they are called independent tasks. Dependent tasks interact in two ways including:

- Precedence dependency: i.e, if a task τ_i has a precedence dependency with τ_j it means that ether the execution of τ_i precedes τ_j or the execution of τ_j precedes τ_i.
- Sharing resource: shared resources are accessed by several tasks in a mutual exclusive manner [103].

Note that a resource is any software structure that can be used by a task to advance its execution [28]. Normally, a resource may be either i) a data structure, ii) a piece of program, iii) a set of variables, or iv) a main memory area.

We consider in this thesis dependent tasks by sharing resources. A piece of task's code executed by shared resource is called a critical section as exhibited in Figure 2.5. As shown in Figure 2.5, in order to access a resource *R* the task τ_1 must first lock the resource *R* and after using it, *R* is unlocked.

In multi-processor and multi-core architectures, resources are divides into two distinct types: local and global resources [68]. A local resource is only used by tasks of

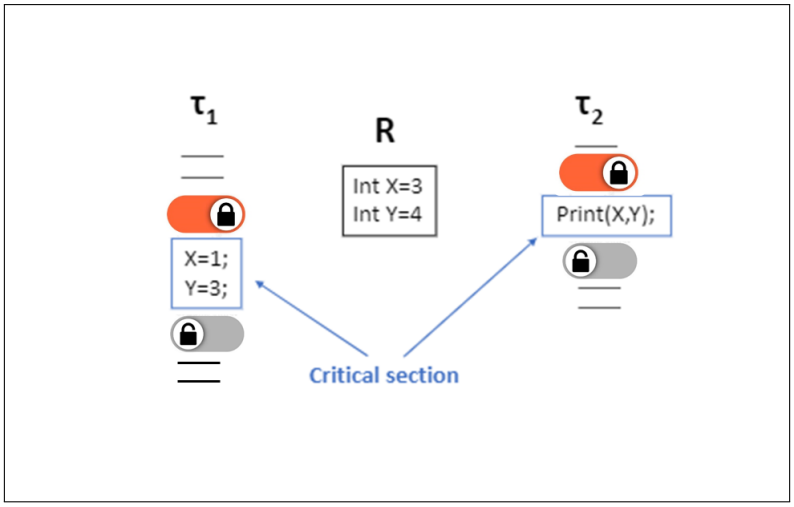

Fig. 2.5 Two tasks sharing two variables.

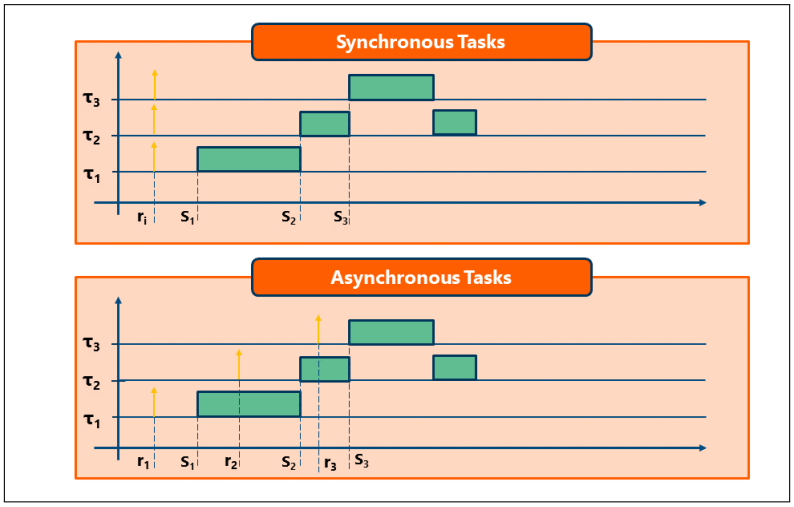

Fig. 2.6 Real-time Synchronous/Asynchronous Tasks.

one application while a global resource is shared by tasks from multiple applications [128]. In our case, as this thesis deals with multi-core systems a local resource in a core ζ_i can be accessed just by local tasks to that core while the global resource is shared among multiple tasks mapped to different cores.

Synchronous/asynchronous tasks

We distinguish two classes of task regarding to the offsets:

- Synchronous tasks: in which the first release of tasks (i.e., offsets) are defined, and all of them are equals [107] $\forall\ i, j,\ \ r_i\ =\ r_j$,
- Asynchronous tasks: an hypothesis activation of tasks is counted [32]. As shown in Figure 2.6, in synchronous case all tasks have the same release time r_i, contrary to the asynchronous case.

  Hypothesis: *In this dissertation, the considered real-time tasks are periodic, dependent, and synchronous.*

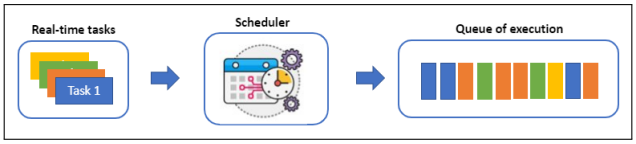

Fig. 2.7 Scheduler Model.

2.2.2.3 Real-time Scheduling

Scheduling is the act of choosing which task to allow execution and when to preempt it [28]. We call scheduling algorithm the set of rules that determines the order in which tasks are executed. The entity that ensures these scheduling policies is called a scheduler [106]. As shown in Figure 2.7, the scheduler dispatches the tasks based on a scheduling policy. Many scheduling techniques exist and each of which is performed for a specific task model or an environment in which a real-time system operates. We list the most important categories in the following. Note that we will present the classification based on the type of the target architecture i.e., mono and multi-core architecture.

Mono-core Scheduling

We present in this part the different types of mono-core scheduling algorithms found in the literature.

Depending on how the priorities are assigned, scheduling algorithms in real-time systems can be classified as follows [39]:

- Fixed Priority: when each task is assigned a fixed priority that does not change at runtime, both Rate Monotonic (RM) scheduling [72] and Deadline Monotonic (DM) [70] use this type of scheduling. In RM algorithm, tasks priorities are assigned according to their periods, i.e., the smaller the period, the higher the priority. DM algorithm is similar to RM but the priority depends on the task relative deadlines instead of periods, i.e., the smaller the deadline, the higher the priority,
- Dynamic Priority: tasks priorities may be reviewed, if necessary over time. Two well-known scheduling algorithms are Earliest-Deadline-First (EDF) [72] and Least-Laxity First (LLF) [57]. In EDF algorithm, the task with earlier deadline among all tasks will execute first. The priority of the task is dynamic and can be changed during run-time depending on the deadline of the task instant and other released tasks ready for execution [72]. LLF algorithm is based on the laxity [14]. LLF elects the task whose the laxity is lowest. In contrary to EDF, LLF is a job-level dynamic priority scheduling algorithm (i.e., the priority of a job may vary with time).

Table 2.2 Comparison of real-time scheduling algorithms.

Algorithm/ Criteria	Rate Monotonic (RM)	Deadline Monotonic (DM)	Earliest Deadline First (EDF)	Least Laxity First (LLF)
Implementation Complexity	Simple	Simple	Difficult	Difficult
Processor Utilization	Less	More than RM	Full Utilization	Full Utilization
Priority Assignment	Static	Static	Dynamic	Dynamic
Scheduling Criteria	Task Period	Relative Deadline	Deadline	Laxity
Preemption	Preemptive	Preemptive	Preemptive	Preemptive
Context switching	More	More	Less	Less
Predictability	More	More	Less	Less
Overhead	Less	Less	More	More

Moreover, depending on whether task instance can be preempted or not, scheduling algorithms can be classified as follow:

- Preemptive: a task can be interrupted by some high priority task during the execution at any time,
- Non-Preemptive: no interruption by other task is allowed

Table 2.2 presents a comparative study between the mentioned scheduling algorithms that emphasis our choice for RM. Comparing to EDF, RM is characterized by simpler implementation [27]. In addition, it is more predictable as tasks response time are constant comparing to dynamic priority algorithms.

Multi-core Scheduling

After having discussed the mono-core scheduling algorithms, this subsection focusses on multi-core scheduling algorithms [100]. The latter may also be classified according to the allocation problem (.i.e., on which a core, a task should be executed). This problem was formulated for first time by Liu in 1969 [72]. There are three approaches to multi-core scheduling: global, partitioned and semi-partitioned [72].

Global scheduling

In this algorithm, tasks are dynamically allocated to cores and they can migrate from a core to another [34]. Moreover, different instances of the same task can execute on different cores. In the case of global scheduling, a single scheduler is used thus, a single scheduling policy is applied to all the cores (See Figure 2.8).

Partitioned scheduling

It consists of applying on each core a scheduling policy which may be different from others. Compared to the global scheduling, it is an off-line scheduling

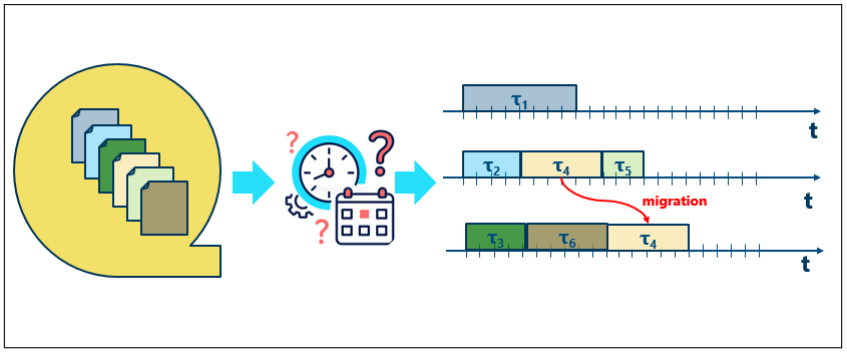

Fig. 2.8 Global Scheduling.

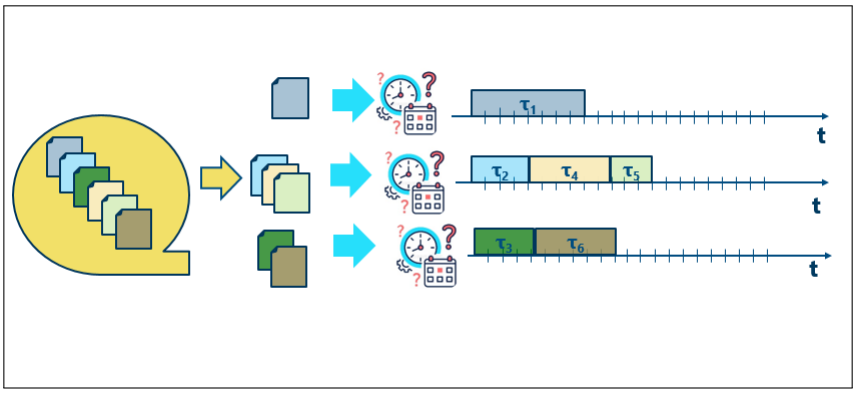

Fig. 2.9 Partitioned Scheduling.

approach allowing each task to be statically assigned to only one core. In this approach, tasks are not allowed to migrate from one core to another (See Figure 2.9).

Semi-partitioned scheduling

The semi-partitioned approach presents an improvement of the partitioning scheduling allowing the controlled tasks migration. It is a hybrid between partitioned and global scheduling [66]. Within this approach, only few tasks can be executed on more than one core (See Figure 2.10). We adopt in this thesis the partitioned approach because it is easier to implement and to analyze by reducing the problem of multi-core scheduling to a set of mono-core problems [30] and treats each core

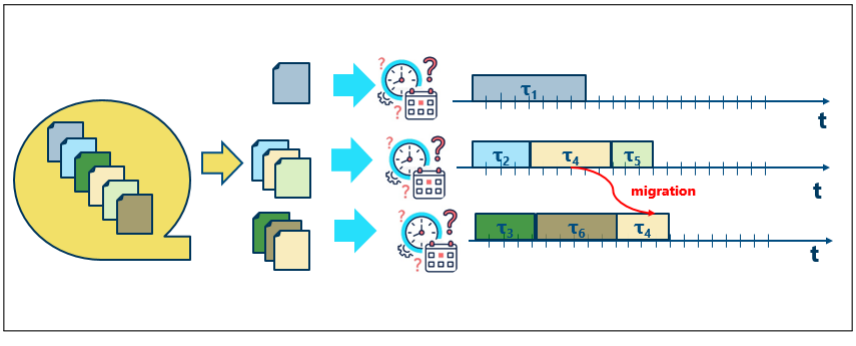

Fig. 2.10 Semi-partitioned Scheduling.

Table 2.3 Comparison of Multi-core scheduling algorithms.

Algorithm/ Criteria	Global Scheduling (GS)	Partitioned Scheduling (PS)	Semi-Partitined Scheduling (SS)
Implementation Complexity	Difficult	Simple	Difficult
Migration	More	No migration	Less
Preemption	More	Less	More as compared to PS
Run-time Overhead	More	Less	More as compared to PS

independently. Thus, it allows to reduce task migration and consequently time overheads as well. Table 2.3 presents a comparative study between the different multi-core scheduling that highlights our choice for partitioned scheduling. In the partitioning scheduling, the designer has to allocate the tasks on the different cores before stating their execution. Allocation tasks on cores (Partitioning) is considered as bin-packing problem which is known to be NP-hard problem [48]. Several heuristic algorithms have been developed to perform the distribution of tasks on the different cores. The most widespread rules are [109]:

- First-fit: mapping task τ_i on the first core able to contain it, in the order of their indexes,
- Best-fit: mapping task τ_i on the first core with the highest utilization factor able to receive it

Those heuristics have been designed to reduce the number of cores for the task partitioning problem. We propose in this thesis an heuristic that aims to optimize the mapping of tasks into core in term of time overhead. Once the partitioning step is done, the tasks will be scheduling using the classic algorithms which is RM in our case.

The different classes of the real-time scheduling algorithms presented in this thesis are summarized in Figure 2.11. To sum up, in this thesis, we consider for both architecture mono-core and multi-core a preemptive static-priority scheduling algorithm which is RM. For the mapping of tasks into cores in multi-core architecture, we adopt the partitioning scheduling.

2.2.3 Real-time Verification

Real-time verification has two aspects: functional and non-functional aspects. For the first aspect, verification consists in checking if the system produces a correct

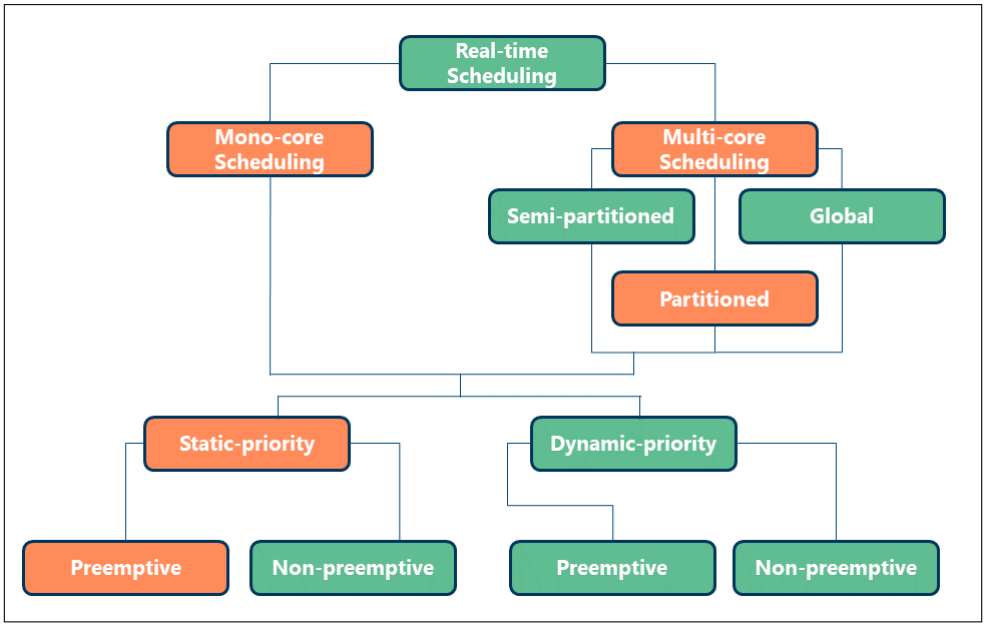

Fig. 2.11 Real-time scheduling algorithms.

result for each input data set [5]. Regarding the non-function aspect, the verification consists in ensuring that the system respects non-functional constraints such as time constraints, memory consumption or power constraints [5]. The verification techniques of non-functional aspect are divided into: i) static techniques which are based on analysis methods and ii) dynamic techniques which are based on simulation, test, and model checking [5]. Note that in this dissertation, we are interested in the verification of the non-functional aspect using static analysis technique which is scheduling analysis. The scheduling analysis determines for a given task model under certain scheduling algorithm, whether all deadlines are met. This analysis is performed under some system hypothesis (.e.g., worst system behavior that executes tasks with worst-case execution time) [40]. Utilization-based scheduling test and Worst Case Response Time (WCRT) [82] are among the most widely used schedulability tests. Note that in this thesis, all the schedulability tests are applied under RM scheduling algorithm [38].

Based on task dependency, we distinguish schedulability analysis for [105] i) independent tasks, and ii)dependent tasks.

2.2.3.1 Schedulability Analysis For Independent Tasks

Utilization-based Schedulability Test For Independent Tasks

This method consists in computing the time that the processor spends to execute the tasks (.i.e., the CPU utilization factor denoted by U). In order to ensure the system feasibility, the CPU utilization factor must be under a specific utilization

threshold. A real-time system composed of N tasks is schedulable if the total utilization does not exceed $N\ (2^{\frac{1}{N}} - 1)$ under RM scheduling algorithm [102]:

$$U\ =\ \sum_{i\in\{1...N\}} \frac{C_i}{T_i} \leq\ N\ (2^{\frac{1}{N}} - 1) \tag{2.1}$$

Where U is given by the following equation:

$$U\ =\ \sum_{i\in\{1...N\}} (\frac{C_i}{T_i} + \sum_{j\in\{1...i-1\}} \frac{C_j}{T_j}) \tag{2.2}$$

WCRT analysis For Independent Tasks

This analysis produces a necessary and sufficient schedulability test. It consists in computing the worst response time R_i of each task τ_i. It is given by [41]:

$$R_i = C_i + I_i \tag{2.3}$$

Where I_i is the maximum interference of task τ_i by higher priority tasks [122]. The interference is build upon i) the number of times the task is interfered, and ii) the execution time of the task interfering it [80]. The maximum interference I_i of task τ_i by the the set of tasks with higher priority than τ_i ($hp(i)$) is given by the formula [73]:

$$I_i = \sum_{j\in hp(i)} \lceil \frac{R_i}{T_j} \rceil\ C_j \tag{2.4}$$

The response time R_i can be calculated as [73]:

$$R_i = C_i + \sum_{j=hp(i)} \lceil \frac{R_i}{T_j} \rceil\ C_j \tag{2.5}$$

The schedule is feasible if: $R_i \leq D_i$ [70].

2.2.3.2 Schedulability Analysis For Dependent Tasks

In order to perform a schedulability analysis of dependent tasks, the blocking time has to be taken into account as sharing resources may cause a priority inversion problem. The latter happens when a task with high priority can not access to a shared resource which is using by another lower priority task. So the higher priority task will be blocked until the lower priority one releases the shared resource. Figure 2.12 illustrates a priority inversion situation between two tasks τ_1 and τ_2 where τ_1 has a higher priority than τ_2. Task τ_1 and τ_2 shared a resource φ_1.

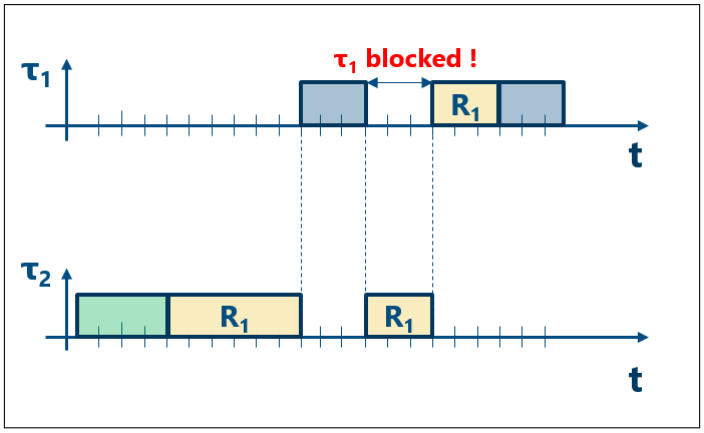

Fig. 2.12 Priority inversion situation between two tasks.

As shown in Figure 2.12, task τ_1 is blocked by τ_2 which is lower priority than τ_1. Preventing higher priority tasks from executing, is needed in almost all meaningful real-time systems [112]. So that, the access to the shared resource should be arbitrated by a synchronization protocol. Many synchronization protocols have been proposed to solve the priority inversion problem for i) mono-core real-time system, such as such as the Priority Inheritance Protocol (PIP) [101], the Stack Resource Policy (SRP) [12] and the Priority Ceiling Protocol (PCP) [94], and ii) multi-core real-time system, such as Flexible Multiprocessor Locking Protocol (FMLP) [23] and Multiprocessor Priority Ceiling Protocol (MPCP) [92]. For mono-core system, PCP is the used synchronization protocol in this thesis and for the multi-core we use MPCP as they are suitable for RM scheduling algorithm.

Blocking time computation under PCP synchronization protocol

In order to compute the blocking time using PCP synchronization protocol, it is necessary to introduce some notions. Let $Use(\varphi_q)$ be the set of tasks sharing the resource φ_q, C_{k,φ_q} be the WCET of task τ_k using the resource φ_q, and ς_q be the ceiling of resource φ_q which is the highest priority of all tasks that can use φ_q. The blocking time B_i of task τ_i, when the PCP protocol is used, is given by

$$B_i = max_{\tau_i,\ \tau_k \in Use(\varphi_q)} C_{k,\varphi_q} \tag{2.6}$$

with $P(\tau_k) < P(\tau_i) \leq \varsigma_q$

Blocking time computation under MPCP synchronization protocol

The tasks scheduled on multi-core architectures would encounter distant blocking other than local blocking due to the global resources. We present the various types of blocking which must be taken into account in the computation of the blocking time. Before giving the expression of the blocking time under MPCP, it was necessary to introduce the following definitions which are explaining by Rajkumar

Table 2.4 Some notations used in this subsection

Symbole	Defenition
n_i^G	The Number of global critical sections of task τ_i
NL_{ir}	The number of tasks with priority lower than the priority of τ_i executing on core ζ_r
$\{\tau'_{ir}\}$	The set of tasks on core ζ_r (other thanτ_i's core) with global critical sections having higher priority than global critical sections of tasks that can directly block τ_i.
NH_{irj}	The number of global critical sections of task $\tau_j \in \{\tau'_{ir}\}$ having higher priority than a global critical section on core ζ_r that can directly block τ_i.
$\{GR_{ij}\}$	The set of global resources that will belocked by both
τ_i and τ_j.	
NC_{ij}	The number of global critical sections of τ_j in which it request a global resource in $\{GR_{ij}\}$.
B_i^{local}	The longest local critical section among tasks with a priority lower than τ_i executing on the same core as τ_i which can block τ_i.
BL_{ij}^{global}	The longest global critical section of task τ_i with a priority lower than τ_j executing on a different core than τ_i's core in which τ_j requests a resource in $\{GR_{ij}\}$.
BH_{ij}^{global}	The longest global critical section of task τ_j with a priority higher than τ_i executing on a different core than τ_i's core. In this global critical section, τ_j requests a resource in $\{GR_{ij}\}$.
$B_{ij}^{\prime global}$	The longest global critical section of $\tau_j \in \{\tau'_{ir}\}$ having higher priority than a global critical section on core ζ_r that can directly block τ_i.
B_{ij}^{lg}	The longest global critical section of a lower priority τ_j on the τ_i's host core.

[93]. We note that each global resource has a ceiling priority equal to the highest priority among all tasks that can use this resource. Thus, the global resource can only be blocked by an other global resource. A task that uses a global resource is executed at the priority of that resource. In the case of local resources, MPCP operates in the same way as PCP. Table 2.4 shows some important notations used to compute the blocking time B_i of the task τ_i. The blocking time B_i is given by the sum of the 5 terms presented below [93].

1. $B_{i,1}$: Local blocking time: it is computed as $n_i^G B_i^{local}$
2. $B_{i,2}$: Direct blocking time: when a task τ_i is blocked on a global resource which is locked by a lower priority task executing on another core and it is computed as $n_i^G BL_{ij}^{global}$.
3. $B_{i,3}$: Direct blocking time: when a task τ_i is blocked on a global resource which is locked by a higher priority task executing on another core and it is

computed as $\sum_{P_i \leq P_j} NC_{ij} \lceil \frac{T_i}{T_j} \rceil BH_{ij}^{global}$ where τ_j is not on τ_i's core.

4. $B_{i,4}$: Indirect preemption delay: when the global resource of lower priority tasks on core ζ_r (different from τ_i's core) are preempted by higher priority global resource of $\tau_j \in \{\tau'_{ir}\}$ it is computed as $\sum_{\tau_j \in \{\tau'_{ir}\}} NH_{irj} \lceil \frac{T_i}{T_j} \rceil B_{ij}^{\prime global}$ where $\zeta_r \neq \tau_i$'s core.

5. $B_{i,5}$: Indirect preemption delay: when τ_i is blocked on global resources and suspends a local task τ_k can execute and enter a global section which can preempt τ_i when it executes in non-global resource sections. $B_{i,5} = \sum_{P_i \leq P_k} min(n_i^G + 1, n_i^G) B_{i,k}^{lg}$.

$$Bi = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} + B_{i,5} \tag{2.7}$$

Utilization-based Schedulability Test For Dependent Tasks

In the case of blocking times the CPU utilization factor U becomes [45]:

$$U = \sum_{i \in \{1...N\}} (\frac{C_i + B_i}{T_i} + \sum_{j \in \{1...i-1\}} \frac{C_j}{T_j}) \tag{2.8}$$

WRCT Analysis For Dependent Tasks

Using the blocking times, the maximum response time of τ_i can be described by the following recurrence equation [73]:

$$R_i = C_i + B_i + \sum_{j=hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j \tag{2.9}$$

2.2.3.3 Energy Consumption Model

The energy consumption is of critical importance to real-time systems. Optimizing it may extend the battery life of the system. In this subsection, we introduce a mathematical model presenting the energy consumption. This latter is denoted by *E*, it consists of two parts [67]: (i) dynamic energy E_d due to the circuit switching activity in the system [69], and (ii) the static energy consumption E_s of the platform without the given application execution. As the dynamic energy E_d is usually dominant and for simplicity reason, the energy model is built based on E_d. Similarly for the multi-core system, the consumed energy in each core is based on the dynamic energy which is given according to [69] by

$$E = E_d = \alpha f V_{dd}^2 \theta \tag{2.10}$$

where V_{dd} is the supply voltage, f is the number of clock cycles, i.e., the processor frequency, θ is the execution period and α is a constant [69].

2.2.4 Real-time Implementation

First, it is necessary to give a basic definition of thread. A thread is defined as a stream of instructions that can be scheduled to run as such by the operation system [123]. At the implementation level, programming real-time application is widely recognized as the hardest kind of programming [26]. Thus, real-time languages have been intended to make coding task easier.

2.2.4.1 Real-time programming language

We review in this subsection some of the languages that are currently used in programming real-time systems.

Ada

It is originally designed by the US Department of Defense (DoD) as a language for huge safety critical systems such as Military systems [26]. Ada is useful because of strong typing, run-time checking, parallel processing, exception handling, and generics. Ada supports concurrent programming via tasks. It is more powerful in large scale programs than in small ones. Ada is a Pascal-like language but it contains additional features of modern programming languages.

RT-Java

Real-time Java is a modification of the standard Java language, it is used in implementing soft, firm and even hard real-time systems, whereas the standard Java is basically used for only soft real-time systems. Contrary to Ada, RT-Java supports real-time threads which implement tasks. It aims to integrate into Java features that are already present in Ada. But in RT-Java thread facility is more dynamic and flexible comparing to Ada's task. In addition, Ada's tasking model is more sophisticated than the Java/RTSJ thread facility [88].

POSIX

POSIX denotes Portable Operating System Interface, it is a set of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems [26]. It deals with thread extensions. POSIX threads are generally known as *Pthreads*.

Table 2.5 presents a comparative study between the aforementioned real-time programming languages. We choose POSIX as a real-time language. The possibility of compiling a code writing in C for almost every hardware platform, the resource management functions, its simplicity, and its wide utilization constitute excellent reasons for choosing this programming language.

Table 2.5 Comparison of real-time programming languages.

Language/ Criteria	Ada	RT-Java	POSIX
Implementation Complexity	Difficult	Less as compared to Ada	Simple
Standardization	Mostly	No	Yes
Flexibility	Yes	Yes	Partial
Resource Management	Mostly	Partial	Yes
Popularity	No	Partial	Yes
Portability	Yes	Yes	No
Completness of Implementation	Partial	Yes	Yes

2.2.4.2 POSIX Threads Programming

We focus in this subsection on the link between POSIX threads (*Pthread*) on Linux kernel and the aforementioned periodic task model. The standard POSIX 1003.1c provides three scheduling policies:

- SCHED_FIFO: It implements Fixed priority, preemptive, First-In First-out scheduling,
- SCHED_RR: It implements Fixed priority, preemptive, round robin scheduling,
- SCHED_OTHER: Not specified (but it often implements the default time-sharing scheduler)

SCHED_FIFO and SCHED_RR can be used to implement RM scheduling policy.

Pthread Pthread is C-based language. It is implemented with the header “pthread.h”, there are more than 100 Pthread procedures which can be classified into four groups [26]:

- **Thread management:** which deal with threads creating, joining, etc. They also may deal with methods that query or set thread attributes (.e.g.,scheduling parameters, scheduling policy, etc.),
- **Mutexes:** which deal with synchronization using mutual exclusion "mutex",
- **Condition variables:** that address communications between threads that share a mutex.,
- **Synchronization** with locks and barriers.

We summarize in table 2.6 all the used Pthread primitives in this thesis.

Table 2.6 Used Pthreads API.

Primitive	Definition
pthread_reate	create a thread
pthread_oin	join a thread
pthread_ttr_nit	intialize pthread attributes
pthread_$t$tr_etschedpolicy	define scheduling policy
pthread_ttr_etschedparam	define scheduling parameters
pthread_mutex_ock/ pthread_mutex_nlock	use a mutex semaphore to protect/ unlock a critical section

2.3 Optimization Methods

In this section, we give an overview on the existing optimization methods that could be used to optimize and solve the decision problems for feasible synthesis of real-time systems.

2.3.1 Mathematical Programming

Mathematical Programming (MP) [54] is used to solve complex problems that can be modeled as i) an objective function to be optimized (see 2.11), and ii) a set of conditions or constraints (see 2.12)

$$Min\ (or\ Max): \quad f(X_1, X_2, \ldots, X_n) \tag{2.11}$$

$$\begin{aligned} & f_1(X_1, X_2, \ldots, X_n) \leq c_1 \\ & f_2(X_1, X_2, \ldots, X_n) \leq c_2 \\ & \ldots \\ & f_m(X_1, X_2, \ldots, X_n) \leq c_m \end{aligned} \tag{2.12}$$

Where $\{X_1, X_2, \ldots, X_n\}$ is the set of optimization variables that characterizes the problem. $\{f_1, f_2, \ldots f_m\}$ and $\{c_1, c_2, \ldots c_m\}$ are respectively the constraint functions and problem parameters that specify the objective.

2.3.1.1 Linear Programming

Linear Programming (LP) [54] is a Mathematical Programming MP where the objective function and all constraints are linear. An LP can be solved by multiple methods: i) algebraically such as the simplex method [108] and the interior point method [10], or ii) graphically [96]. These methods can solve very big problems with a huge number of variables and constraints. But the behavior of real-world

problems can only be approximated [54], since all variables must to have real number and the constraints and the objective function have to be linear.

2.3.1.2 Mixed-integer Linear Programming

An LP aims to optimize (maximizes/minimizes) a linear objective function subject to a set of constraints. Mixed integer programming has an additional condition that at least one of the variables can only take on integer values. With integer or binary variables we can provide a more detailed representation of real-time systems than with LP. In Mixed-integer Linear Programming, solvers use a combination of algorithms, such as branch-and-bound, cutting plane and heuristics [63]. MILP solvers are able to solve large problems in term of number of variables and constraints. The time for solving a MILP is hard to be estimated as it depends on the specific structure (.i.e., formulation) of the problem, and its complexity.

2.3.2 Genetic Algorithm

A genetic algorithm (GA) is a population based meta-heuristic inspired by biological evolution [77]. GA [84] consists on four steps which are then repeatedly applied to the population until the last population meets a certain stopping criteria:

1. Random creation of population of candidate solutions (.i.e., individuals) for the optimization problem.
2. Based on the value of the objective function (fitness), we evaluate each candidate solution
3. Creation of new generation based on the selected candidate using crossover and mutation mechanisms [84].
4. Evaluation of the new generation population.

Generally, GA is not convenient for finding the solutions to complex problems.

2.3.3 Dynamic Programming

Dynamic Programming (DP) is a technique for formulating problems in which decisions are to made in stages (multistage) [74]. It is based on Bellman's principle of optimality which states that a sub-policy of an optimal policy for a given problem must itself be an optimal policy of the sub-problem [44]. DP deals with

Table 2.7 Comparative Study of Optimization Methods.

Optimization Method	Advantages	Drawbacks
Linear programming (LP)	– Scales to big problems – Global optimum attainable	– Very limited expressions available – Difficult to deal with complex interactions
Mixed-integer linear programming (MILP)	– Scalable for big problems – Handles complex interactions – Possibility of the evaluation of solutions' quality	– Bad worst-case complexity
Genetic algorithm (GA)	- High freedom of expression	– Difficult to scales big problems – Impossibility of the evaluation of solutions' quality
Dynamic programming (DP)	– High freedom of expression – Global optimum attainable	– Difficult to scale big problems – Possiblity of suboptimal solution due to discretization of continuous variables.

problem including non-convex, non-continuous, non-differentiable and black-box functions and it is able to find the global optimal solution of a problem [61]. Table 2.7 summarizes the optimization methods with their advantages and disadvantages. This table emphasis our choice for MILP. In fact, MILP provides very accurate models comparing with the other optimization methods.

2.4 Synthesis of Reconfigurable Real-time systems

This section presents a global picture of the related works to our approach for the synthesis of reconfigurable real-time systems. *Note that we mean by synthesis the derivation of a system from its specification.*
First, we present software function to task assignment works. Next, we cite reconfiguration-aware approaches. Then, we introduce optimization-based approaches. Finally, we review works that deal with task partitioning (i.e., task to core) problem. The goal of this discussion is not to introduce a detailed survey of the related work, but to underline that each of the approaches is proposed with limited subset of constraints. We present in Table 2.8, a set of requirements and their gratification by the cited approaches.

Many approaches in the literature aim to design architectural models that need to meet timing requirements, by transforming the functional specification to tasking architecture. In the work related in [83], the authors, provide a framework for real-time embedded system synthesis. Using Prelude language, designers write a functional specification with this framework. Then, the latter generates real-time tasks using one to one assignment strategy. The main drawback of this strategy is

the time overhead caused by the large number of produced task. In the literature, different alternatives exist in order to solve the optimization problem related to this assignment strategy. The work reported in [21] proposes a clustering technique that allows the assignment of more than one function to the same task. Contrasting the clustering method proposed by [24], the clustering is restricted to tasks with identical periods. Beside the clustering strategy, the authors of [24] define multi-criteria design exploration methods as it involves several performance criteria to be optimized while performing scheduling analysis. However, this approach is intended only for a mono-core architecture. In addition, it does not deal with both task/function dependency constraint and reconfiguration constraint.

Some research contributions have been dedicated to develop reconfigurable real-time systems in various areas. The work reported in [4] presents a framework for design and implementation of reconfigurable real-time systems by providing techniques for runtime reconfiguration, and dynamic intercircuit communication and synchronization for FPGAs. Providing guidance and support to the designer is out of scope of this work. This problem is solving by the work reported in [91] which proposes a guided platform to help application developer in the implementation of hardware-software solutions for reconfigurable systems(FPGA-based system). The work aims to translate high-level functions into FPGA-accelerated kernels by matching software functions to appropriate architectural templates.

Task partitioning problems are very common in multi-core real-time systems. Mostly, they are proven to be special instances of the general bin-packing problem. Finding an optimal solution in polynomial time is not a trivial task so that, many heuristics have been proposed in the literature in order to solve the allocation problem in an acceptable computational time.
For multi-core real-time systems without considering shared resources, several solutions are proposed, e.g., [56]. The work reported in [56] proposes an heuristic (HYDRA) for assigning security tasks into multi-core real-time systems. The authors aim to insert security tasks into this schedule without changing it and without breaking the system's real time constraints. The work reported in [7] differs from [56] by considering the impact of shared resources when performing task allocation. It presents an optimal MILP-based partitioning strategy for fixed priority scheduling with shared resources. Several important works have focused on energy-aware partitioning. In [1], the authors describe and evaluate real-time task assignment heuristics for optimizing the global deadline success ratio. This work aims to determine a partitioning that guarantees absence of both energy starvation and deadline missing under EDF scheduling.

Metaheuristic algorithms (e.g., genetic algorithm) and Mathematical program-

Table 2.8 Related work overview.

Work	Task Dependency		Reconfiguration Aware	Function Assignment	Task~ Partitioning	Optimization Aware	Guided Strategy
	Dependent	Independent					
[83]		✓	-	✓	-	-	-
[21]	-	✓	-	✓	-	✓	-
[24]	-	✓	-	✓	-	✓	✓
[4]	-	✓	✓	-	-	-	-
[91]	-	✓	✓	-	-	✓	✓
[56]	✓	-	-	-	✓	✓	-
[7]	✓	-	-	-	✓	✓	-
[1]	-	✓	-	-	✓	✓	-
[87]	✓	-	✓	-	✓	✓	-
[60]	✓	-	-	-	✓	✓	-
[71]	-	-	-	-	-	✓	-
Our Thesis	✓	-	✓	✓	✓	✓	✓

ming (e.g., MILP, ILP, and dynamic programming) have increasingly become the focus of attention in solving many issues of optimization in the design of real-time systems. A genetic algorithm approach to minimize energy consumption and schedule length in a multiprocessor system with two homogeneous processors with shared memory architecture is presented in [87]. This work proposed an ontology-based agent to design system reconfigurations according to user requirement. For MILP-based optimization approach, several works are proposed, e.g., [60]. In this approach, the author focus on the problem of static scheduling multiple periodic systems composed of both hard and soft periodic tasks in multiprocessor systems for energy consumption minimization. A dynamic programming (DP) has been employed [71] to develop an optimal real-time system with real-time energy management. This work aims to find an optimal control mode in term of energy consumption. The above described approaches ([87], [60], and [71]) share a common real-time aspect: schedulability of the system in the scope of optimization techniques. However, none of these approaches deals function to task assignment problem, also they does not provide decision aiding solutions.

2.4.1 Discussion (comparative table)

An overview of related work is presented in Table 2.8, summarizing the previous works and techniques covered in this work. None of these solutions simultaneously considers the real-time constraints, reconfiguration property, function assignment, partitioning, optimization aware and guidance aspect. In this dissertation, we propose a guided strategy for modeling, designing, and implementing feasible reconfigurable real-time systems in multi-core architecture with dependent tasks using multi-objective optimization techniques.

Conclusion

In this chapter, we overviewed the approaches that have been used in the literature to solve the problem of real-time systems synthesis, including scheduling, placement, and optimization problem in mono-core and multi-core architecture. We have presented the real-time systems, their different classes. We have introduced the concept of real-time tasks and the related timing constraints. We have talked about the schedubility analysis, energy consumption model, real-time programming languages as well as the optimization methods. We have studied the solving approaches and existing methods in the literature. Few researches have been successfully done to solve some synthesis problems but still present some defects. None of these solutions considers simultaneously the real-time constraints, reconfiguration property, partitioning and scheduling problem, and user guidance. Thus, this work is motivated to deal with all of these constraints. More details on our contributions will be given in the following chapters.

CHAPTER 3

Implementation of Mono-core Reconfigurable Real-time systems

Introduction

The previous chapter dedicated to introducing basic concepts concerning reconfigurable real-time systems in mono-core and multi-core architecture. It further surveyed existing methodology in the literature for synthesis reconfigurable real-time systems. In this chapter, we present an initial version of MO^2R^2S approach which is a multi-objective optimization approach for the development of a reconfigurable real-time system with mono-core architecture. It is organized as follows: in Section 3.1, we give the motivation of this work and summarize our contributions. In Section 3.2, we propose the formalization and terminology used to model the approach. The proposed methodology is well detailed in Section 3.3. This chapter is closed by a proposal of a formal case study presented in Section 3.4. Note that this chapter has been published in the international journal IEEE transactions on Systems, Man, and Cybernetics(t-SMC).

3.1 Motivation

Despite significant advances in the development of real-time systems, the synthesis of such system from specification level to implementation is not a trivial task. The main phase of the synthesis is called deployment which is done at the design level and it concerns the

- software architecture exploration (i.e., assigning functions to tasks),
- scheduling of tasks. Besides, the challenges produced by real-time system, reconfiguration induces additional difficulties such as having a huge number of implementations may induces a big number of redundancies between them [64].

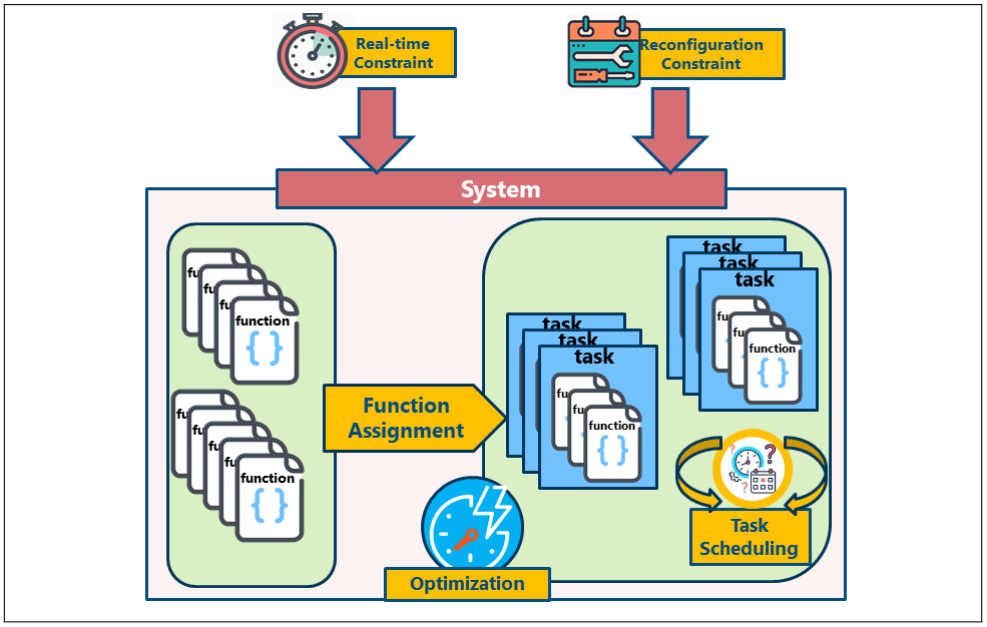

Fig. 3.1 Challenges of Reconfigurable Real-tie System Implementation under Mono-core Architecture.

In addition, such system is generally specified by a large number of applicative functions which may

- causes an important time overhead,
- causes many reconfigurations between the different implementations which increases the reconfiguration time [64],
- increases response time,
- produces complex system code.

Moreover, as it is important for these systems that tasks meet their real-time constraints, the need increasingly oriented towards powerful processors, without forgetting the importance of the trade-off between system performance and energy efficiency for those battery-based systems. The problem we deal with in this chapter can be summarized as shown in Figure 3.1

- function to task assignment for a given functional specification,
- task scheduling,
- multi-criteria optimization process,
- real-time constraints,
- reconfiguration constraint.

In order to overcome these limitations, we propose the MO^{2}R^{2}S approach which addresses initially the mono-core architecture. This approach aims to:

- Reduce the number of tasks,
- Minimize reconfiguration time between implementation set,
- Minimize response time,
- Minimize energy consumption
- Guarantee the respect of deadlines after each reconfiguration scenario
- Optimize the system code,

3.2 Formalization

We introduce in this section a formal description of a reconfigurable real-time system in mono-core architecture. It presents the formalization of common notions used throughout this chapter i) system model, ii) real-time analysis i.e., blocking time, CPU utilization and response time, iii) reconguration time model, and iv) energy model proposed for mono-core real-time system to adequate models under reconfiguration constraints.

3.2.1 System Model

In specification level, we present the reconfigurable real-time system as a set of

- p function denoted by $\xi_F = \{F_1, \dots F_p\}$, each function is characterized by
  - r_{F_k} the release time (.i.e., each function cannot begin executing before r_{F_k}),
  - T_{F_k} which is the activation period of function F_k,
  - C_{F_k} which is an estimation of the worst case execution time (WCET) of F_k,
  - Cn_{F_k} the computation time at normalized processor frequency,
  - $Cond$ under which the function must be executed,
  - $Type$ which can be normal when during its execution it does not depend on other functions or critical when it is a critical section in other functions,
  - $\mathscr{F}_k$ which represents the set of functions having dependency on F_k,
- m conditions defining the execution modes of the considered system [64].

The function and condition sets present an entry point to the design level. In this latter, we present the reconfigurable real-time system as a set of m implementations. Each implementation Π_i is composed of i) N_i periodic tasks which

implement normal functions, and ii) R_i shared resources which implement the critical functions. Note that a task can implement one or more normal functions. In the case of just one normal function, the task is characterized by the same parameters of the implemented normal function. In the other case, the task must implement harmonic normal function i.e., $\forall i, j \in \{1..p\}$ two functions F_i and F_j are harmonic if and only if T_{F_i} *mod* T_{F_j} = *q*, where *q* in an integer. If a τ_j in Π_i implements more than one normal function it would be characterized by

- its release time r_{ij},
- T_{ij} which is equal to the smallest period of the implemented harmonic functions,
- C_{ij} the WCET of τ_j, which is equal to the sum of the WCETs of the normal functions executed by this task in implementation Π_i,
- Cn_{ij} the computation time at normalized processor frequency,
- the deadline D_{ij} which we assume that is equal to its period $D_{ij} = T_{ij}$,
- priority P_{ij} which we assume that is inversely proportional to period T_{ij} according to the RM policy,
- $C_{\varphi_{q_j}}$ the time required by task τ_j to access to shared resource φ_q such that $C_{\varphi_{q_j}} \leq C_{ij}$,
- B_{ij} the blocking time,
- E_{ij} presents the energy consumption.

Note that a task may belong to more than one implementation, so it may have different values for its parameters. We denote by ξ_{φ_i} the set of R_i resources that can be shared between the tasks in implementation Π_i $\xi_{\varphi_i} = \{\varphi_1, \ldots, \varphi_{R_i}\}$. We denote by Sys a reconfigurable real-time system which is defined by $Sys = (\xi_{imp}, C)$ (See Figure 3.2), where:

- ξ_{imp}: presents *m* implementations that define the system where $\xi_{imp} = \{\Pi_1, \ldots \Pi_m\}$. Each implementation Π_i is characterized by a N_i tasks and R_i shared resources where $N_i \leq N$ (respectively $R_i \leq R$),
- C represents the controller which manages the moving from one implementation to another under well defined reconfiguration conditions.

3.2.2 Real-time Analysis

The analysis results may compute three parameters: The blocking time, the processor utilization and WCRT analysis.

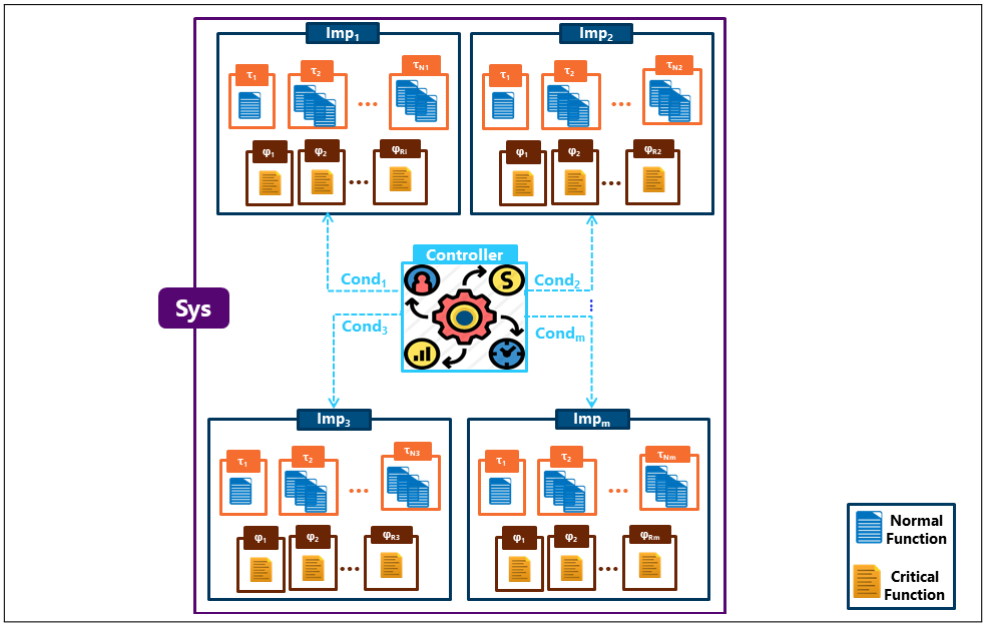

Fig. 3.2 Reconfigurable Real-time system Model.

3.2.2.1 Blocking time

The MO^{2}R^{2}S approach is based on *PCP* for managing the access to shared resources. We define ceiling ς_q of resource φ_q as the highest priority of all tasks that can use φ_q [64]. Thus, we define the blocking time B_{ij} of the task τ_j in implementation Π_i is given by

$$B_{ij} = max_{\tau_l \in \chi_{\tau_i}, \varphi_q \in \xi_\varphi} \{C_{\varphi_{q_l}}\} - 1 \tag{3.1}$$

with $pri(\tau_l) < pri(\tau_j) \leq \varsigma_q$. Namely the priority of τ_l is lower than τ_j.

3.2.2.2 CPU Utilization

In this thesis, we propose the computation of CPU utilization factor U as the sum of the CPU utilization factor of each implementation U_i where $i \in \{1..m\}$. The expression of U is given by

$$U = \sum_{i \in \{1...m\}} U_i \tag{3.2}$$

Where U_i is defined by

$$U_i = \sum_{j \in \{1...N_i\}} \frac{C_{ij} + B_{ij}}{T_{ij}} \tag{3.3}$$

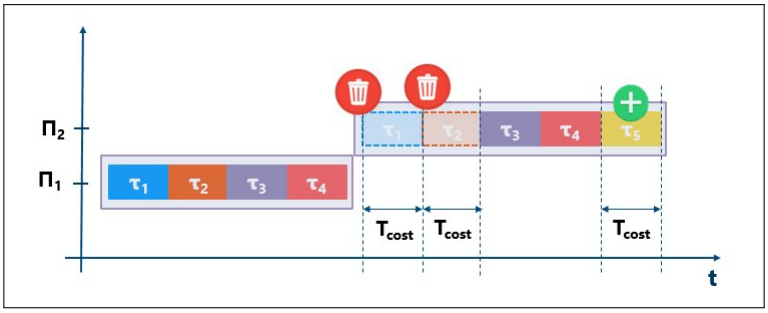

Fig. 3.3 Reconfiguration scenario.

3.2.2.3 WCRT analysis

As we mentioned in previous chapter, Worst Case Response Time (WCRT) analysis is one of the most widely used schedulability tests. We aim in this thesis to optimized it. It is defined as the time spent between the start time of a task and its finish time. We denote by $\mathscr{R}_{ij}$ the response time of task τ_j in implementation Π_i. The proposed expression of $\mathscr{R}_{ij}$ is given by [64]

$$\mathscr{R}_{ij}^0 = 0, \quad \mathscr{R}_{ij}^h = C_{ij} + B_{ij} + \sum_{l \in Hp(j)} \lceil \frac{\mathscr{R}_{ij}^{h-1}}{T_{il}} \rceil C_{il} \tag{3.4}$$

3.2.3 Reconfiguration Time Model

The reconfiguration scenario corresponds to either adding or removing tasks. We introduce in this thesis the reconfiguration time $\Re_i$ related to Π_i as the time spent by the system to jump to Π_i (i.e., required time for loading Π_i) [65]. It is given by [65]

$$\mathscr{T}_i = A * T_{cost} + B * T_{cost} = (A + B) * T_{cost} \tag{3.5}$$

where *A* (resp, *B*) denotes the number of suspended tasks (i.e., deleted tasks) (resp, activated tasks), T_{cost} is the time spent to suspend/activate a task. We aim in this thesis to minimize the reconfiguration time. Figure 3.3 shows an example of reconfiguration scenario which aims to compute $\Re_i$ in each implementation Π_i. The system is composed of two implementation Π_1 and Π_2, the first implementation Π_1 is defined by $\Pi_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ to move to the second implementation Π_2 we need to suspend two tasks τ_1 and τ_2 and activate τ_5. So the reconfiguration time $\Re_2$ related to Π_2 is given by: $2 * T_{cost} + 1 * T_{cost}$, so $\Re_2 = 3 * T_{cost}$. If we assume that T_{cost} is equal to $5\ ms$, then $\Re_2$=$2 * 5$=$10ms$.

3.2.4 Energy consumption Model

Energy consumption is one of the most challenging issues in current real-time systems. During tasks execution, the processor needs a significant amount of energy due to the important amount of data to be calculated. In order to get over this problem, we consider a real-time system with multi-core architecture supporting the Dynamic Voltage and Scaling frequency DVFS [37] capabilities. Real-time voltage and frequency scaling can potentially save energy, while they meet real-time constraints [46]. Let's f_n and V_n be the normalized frequency and the voltage of the system. Note that, the execution time of the task is extended when the voltage is decreased to save the energy. The fact is that the processor frequency is roughly linearly proportional to the voltage supply [37]. Thus, reducing voltage cuts down the energy dissipation. We can see that the task execution time is inversely proportional to the voltage [37]. In order to minimize the energy consumption while respecting system performance and temporal requirements, the supply voltage should be scaled as low as possible. As we mentioned in the previous chapter the energy consumption is given by

$$E = \alpha f V_{dd}^2 \theta \tag{3.6}$$

where V_{dd} is the supply voltage, f is the number of clock cycles, i.e., the processor frequency, θ is the execution period and α is a constant. We assume that $\alpha = 1$. We suppose that each function F_k is described by two parameters: (i) the function's frequency f_{F_k}, and (ii) the function's voltage V_{F_k} [64]. The energy consumption for the execution of function F_k that we denote by E_{F_k} is computed as $E_{F_k} = f_{F_k} V_{F_k}^2 C_{F_k}$ [64]. We introduce in this thesis the energy consumption E_{ij} of task τ_j in Π_i as to the sum of the implemented functions $E_{ij} = \sum_{k \in \{1 \ldots p_j\}} E_{F_k} = \sum_{k \in \{1 \ldots p_j\}} f_{F_k} V_{F_k}^2 C_{F_k} = \sum_{j \in \{1 \ldots N_i\}} f_{ij} \quad V_{ij}^2 C_{ij}$ where (f_{ij}, V_{ij}) are two parameters characterizing task τ_j in implementation Π_i [64]. We assume that $f_{ij} = \sum_{k \in \{1 \ldots p_j\}} f_{F_k}$ and $V_{ij} = \sum_{k \in \{1 \ldots p_j\}} V_{F_k}$. Thus the total energy consumption of implementation Π_i is given by [64]

$$E_i = \sum_{j \in \{0, N_i\}} E_{ij} = \sum_{j \in \{0, N_i\}} f_{ij} V_{ij}^2 C_{ij} \tag{3.7}$$

Let's η_j be the reduction factor of voltage when τ_j is executed, $V_{ij} = \frac{V_n}{\eta_j}$ and $f_{ij} = \frac{f_n}{\eta_j}$. So the WCET is equal to $C_{ij} = C_{n_{ij}} \eta_j$. Thus the total energy consumption of

implementation Π_i according to [37] is given by

$$E_i = \sum_{j \in \{0,N_i\}} \frac{f_n * V_n^2 * C_{n_{ij}}}{\eta_j^2} = K \sum_{j \in \{0,n\}} \frac{C_{n_{ij}}}{\eta_j^2} \tag{3.8}$$

where $K = V_n^2 f_n$.

3.3 MO^{2}R^{2}S Approach

In this section, first we present an overview of our design methodology. Then, we detail the structure of the different modules involved in this work.

3.3.1 Methodology description

As shown in Figure 3.4, the entry point of our approach is the specification model provided by the designer. This specification defines i) functions of the system, their temporal parameters and dependencies between them, and ii) reconfiguration conditions. Then, from this specification model, an initial task model is proposed such a one to one assignment solution, i.e., each normal function (respectively each critical function) is assigned to a single task (respectively to a shared resource). Note that the feasibility concerns are not considered at this level. Once the initial task model is defined, we come to the multi-objective design and optimization step based on MILP formulation which aims to produce a feasible task model while optimizing task number and either energy consumption or response time depending on the designer choice. When this step fails to find a feasible solution then the timing parameters of the functions must be adapted. Finally,a POSIX code is generated from the optimized task model.

3.3.2 Initial Task Model Generation

MO^{2}R^{2}S needs an initial solution to start the search procedure in the next step (i.e., optimization step). In this step, the specification model is considered as input. This step is performed by an heuristic (Algorithme 1) that generates:

- implementation set for different reconfiguration conditions,
- shared resource set such as for each critical function F_k, it generates a shared resource φ_q,
- task set, for each normal function F_k it generates a task τ_j that has the same parameters of F_k

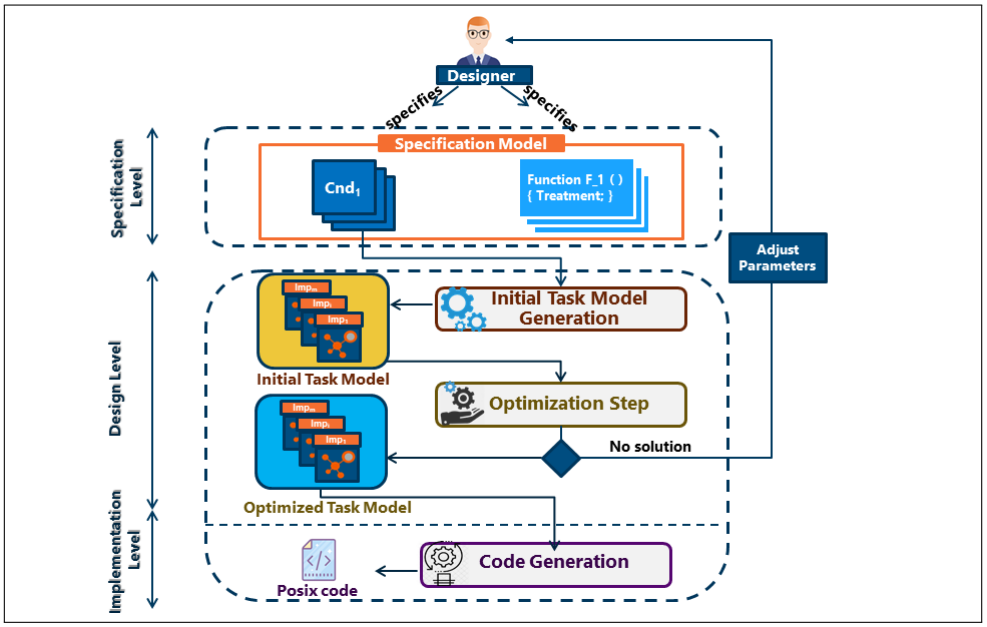

Fig. 3.4 Approach description.

Algorithm 1: INITIAL TASK GENERATION [64]

Input:
- F : Functions set
- ReconfCnd : Reconfiguration condition set

Output:
- InitTask : Initial Task Model
- imp : Implementation set

Notations:
- Reconf_Cnd_Func: Correlation table between the reconfiguration conditions and the functions.
- φ : Resource set

/*** Resource's number ***/
$nbr\_R \leftarrow 0$
$k \leftarrow 0$
/*** Generation Of Implementations ***/
for $i \leftarrow 0$ ***to*** $SizeOf(ReconfCnd)$ **do**
  for $j \leftarrow 0$ ***to*** $SizeOf(F)$ **do**
    if $(F[j] \in Reconf\_Cnd\_Func[i])$ **then**
      $imp[i][k] = F[j]$
    $k++;$
  $k \leftarrow 0$
/*** Generation Of Shared resource ***/
$i \leftarrow 0$
for $j \leftarrow 0$ ***to*** $SizeOf(F)$ **do**
  if $F[j]$ *is a critical function* **then**
    $\varphi[nbr\_R] = F[j]$
    if $F[j] \in imp[i]$ **then**
      $\varphi[nbr\_R] \in imp[i]$
    $i++;$
    nbr_R++
/*** Generation Of Task Model ***/
for *each implementation* imp_i **do**
  for $each function F_j$ **do**
    /* We create a task and we initialize its parameters with function F_j parameters */
    $ReleaseTimeOf(InitTask[j]) = ReleaseTimeOf(F[j])$
    $WcetOf(InitTask[j]) = WcetOf(F[j])$
    $PeriodOf(InitTask[j]) = PeriodOf(F[j])$
    $DeadlineOf(InitTask[j]) = DeadlineOf(InitTask[j])$
    $NormalizeWCETOf(InitTask[j]) = NormalizeWCETOf(F[j])$
    $imp[i][j] = InitTask[j]$
return *imp*

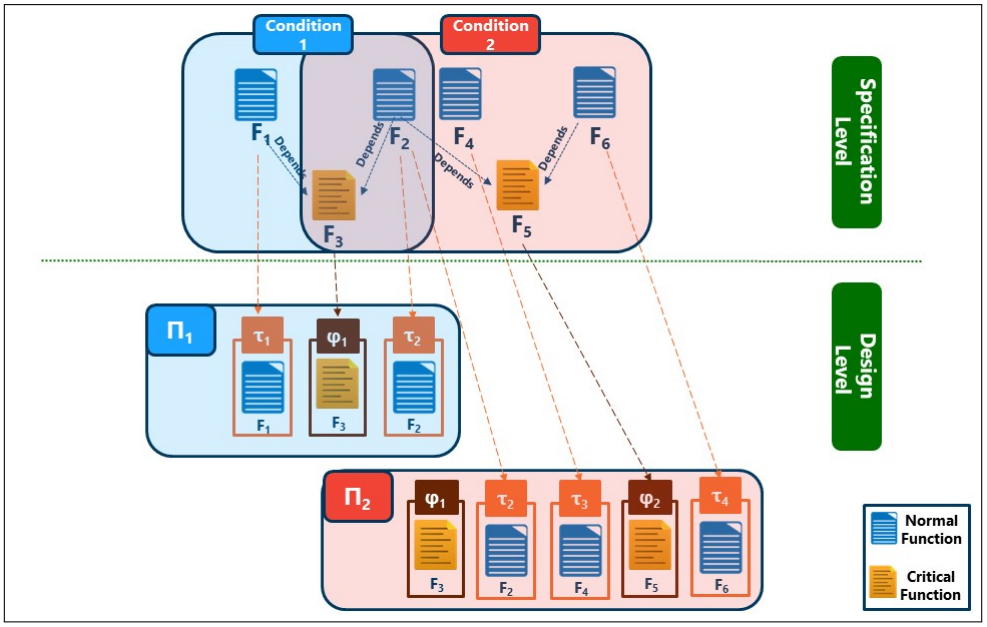

Fig. 3.5 Running Example of an Initial task Model Generation.

Note that in this step the feasibility concerns are not inspected. The complexity of this step is $\mathcal{O}(N^2)$. For example a system *Sys* specified by a designer as a set of six functions. Let us suppose that F_1, F_2, F_4, and F_6 are normal functions and F_3 and F_5 are critical functions depicted in Figure 3.5. F_1, F_2, and F_3 are executed in condition Cnd_1, while the rest are executed in reconfiguration condition Cnd_2. By executing, Algorithm 1 we obtain an initial task model with two implementations Π_1 and Π_2 defined as follows: $\Pi_1 = \{\tau_1, \tau_2, \varphi_1\}$ and $\Pi_2 = \{\tau_2, \tau_3, \tau_4, \varphi_2\}$. Note that in this step, the number of tasks is equal to number of normal functions, no clustering technique is applied in this step. As the next step (i.e., Multi-objective Design and Optimization Step) needs an initial solution to start the search procedure, this initial model presents the input of this step.

3.3.3 Multi-objective Design and Optimization Step

This step is performed by a MILP formulation to define a multi-criteria design exploration as it involves several performance criteria to be optimized while performing the scheduling. As we mentioned previously, this step aims to i) reduce the number of tasks which leads to reduce the number of threads in implementation level thus optimize the code complexity, ii) reduce reconfiguration time between implementation sets, iii) minimizing either the energy consumption or response time. Since the third objective consists in optimizing one of two metrics depending on the designer choice (i.e., energy consumption or response time) two MILP formulations are provided. These latter share common points such

as i) the two objectives which are number of task and reconfiguration time minimization, and ii) temporal constraints. We combined the first objective which aim to reduce the number of tasks with the second objective which attempts to decrease the reconfiguration time between implementations, into just one objective by proposing to merge harmonic tasks while ensuring the respect of timing properties. We remind that two tasks are harmonic if and only if they implement harmonic functions [64]. Thereby, if τ_j and τ_k are two harmonic tasks (i.e., T_{ij} mod $T_{rk} = q$) so the result of their merging produces τ_l with the following temporal parameters: $z \in \{i, r\}, \quad \forall i, r \in \{1 \dots m\}$

$$\tau_j = (C_{zj}, T_{zj}, D_{zj}) = \begin{cases} C_{zj} = \begin{cases} C_{ij} + C_{rk} \; if \; i = l \\ C_{ij} \; in \; \Pi_i \; And \; C_{rk} \; in \; \Pi_r \; otherwise. \end{cases} \\ T_{zj} = min(T_{ij}, T_{rk}) \\ D_{zj} = min(D_{ij}, D_{rk}) \end{cases}$$

Note that the execution time of the merged task τ_j depends on the task that will be executed. Let us consider an initial task model which is composed by two implementations Π_1 and Π_2. Each implementation is characterized by a set of task as shown in Table 3.1. We can notice that i) task τ_1 is harmonic with τ_2 in Π_1 and τ_3 in Π_2, and ii) τ_4 is harmonic with τ_5 in Π_2. By applying merging technique for this particular example, we obtain a new SW architecture which is presented in Table 3.2. By applying the merge technique τ_1 absorbs τ_2, and τ_3,

Table 3.1 Example Of Initial Task Model

Π_i	τ_j	T_{ij}	C_{ij}	D_{ij}
Π_1	τ_1	5 ms	2 ms	5 ms
	τ_2	10 ms	1 ms	10 ms
Π_2	τ_3	5 ms	1 ms	5 ms
	τ_4	21 ms	2 ms	21 ms
	τ_5	7 ms	1 ms	7 ms

Table 3.2 Example: Resulting Task Model

Π_i	τ_j	T_{ij}	C_{ij}	D_{ij}
Π_1	τ_1	5 ms	3 ms	5 ms
Π_2	τ_1	5 ms	1 ms	5 ms
	τ_4	7 ms	3 ms	7 ms

and τ_4 absorbs τ_7. So obtain just two task τ_1 and τ_4, such as their WCETs depends on which implementation they are executed.

3.3.3.1 Models Parameters *and* Variables

The two models parameters and variables are depicted in the following table 3.3

Table 3.3 Models Parameters and Variables.Constants

Constants	
Concepts	Defintion
Hm_{jk}	A boolean variable used to mention if two tasks are harmonic. Thus if the value of $Hm_{sjk}$is equal to 1, then the corresponding tasks $\tau_j$and $\tau_k$have harmonic rates.
C_{ij}	Task's WCET
D_{ij}	Tasks' deadline
M	Big constant
N	Number of tasks
Variables	
Concepts	Defintion
Mg_{jk}	A boolean variable used to mention whether two tasks $\tau_j$and τ_k are merged such that $Mg_{jk}$is equal to 1 if task $\tau_j \in imp_i$ and task $\tau_k \in imp_r$are merged, the merge corresponds to the situation in which $\tau_j$absorbs τ_k, to be deleted from the model
NewTask	The resulting task model after merging the different tasks (i.e., optimized task model)
$C_{new_{ij}}$	The new task's WCET
$D_{new_{ij}}$	The new task's deadline
$T_{new_{ij}}$	The new task's period
μ_{ijk}	A binary variable where μ_{ijk} = 1 when τ_j is executed before τ_k
U	CPU utilization factor
$B_{new_{ij}}$ \ B_{ij}	The new blocking time \The old blocking time (i.e., before merging technique)
$\mathscr{R}_{ij}$	The reponse time of τ_j
y_{ijk}	Number of possible interference of τ_k on τ_j.
x_{ijk}	Number of possible interference of τ_k on τ_j. if μ_{ikj} = 1
η_j	Scaling factor of τ_j
η_{min}	η_{min} = max (ηj), j = 1...N
$C_{new_{n_{ij}}}$	The new normalized WCET of τ_j

3.3.3.2 General Objective Function

We define in expression (3.9) the shared objective function. It aims to maximize the number of merges while minimizing the metric *Metric* which could be either the total response time or the energy consumption.

$$Maximize \sum_{i \in \{1..m\}} \sum_{j,k \in \{0,N_i\}} Mg_{jk} \; - \; Metric \tag{3.9}$$

In the following, we present first the common constraints (i.e., the constraints related to merging situations and real-time constraints), then we define the constraints specific to each metric.

3.3.3.3 Common constraints

A we mentioned above, the two formulations share common constraints that we define in this section.

Merging situation constraints.

In order to avoid the merge of non-harmonic tasks we define constraint (3.10) [64]

$$\forall j,k \in \{1..N_i\} \quad Mg_{jk} = 0 \quad if \quad (Hm_{jk} = 0) \tag{3.10}$$

If Hm_{jk} = 0 this means that task τ_j and τ_k are not harmonic, therefore Mg_{jk} will be equal to zero and the two tasks could not be merged. Constraint (3.11) is defined to avoid merging task which is already merged i.e., $\forall\ k\ \in \{1..N_i\}$

$$\sum_{j \in \{1..N_i\}} Mg_{jk} \; \leq \; 1, \; \forall\ j, \; k, \; z \; \in \; \{1..N_i\}, \; k, z \neq \; j, \; Mg_{jk} \; + \; Mg_{zj} \; \leq \; 1 \tag{3.11}$$

This constraint ensures that the task τ_j can be absorbed by just one task τ_i. Constraint (3.12) is defined to create the new obtained model i.e.,

$$\forall\ j \; \in \; \{1 \ldots N_i\}, \; NewTask_k \; = \; 1 - \sum_{j \in \{1...N_i\}} Mg_{jk} \tag{3.12}$$

We define a new boolean variable $NewTask_j$ which presents the new task model after merging. The two constraints 3.11 and 3.12 ensure that if Mg_{jk} = 1 than $NewTask_k$ = 0 and $NewTask_j$=1, it means that the task τ_k is absorbed by τ_j.

Real-time constraints.

The constraints defined in this section are related to real-time requirements. First, we define the model obtained after applying the merge technique. The new WCET $C_{new_{zl}}$ is given by

$\forall j \in \{1..N_i\},\ \forall k \in \{1..N_l\}, \forall i,l \in \{1 \dots m\}$

$$C_{new_{zl}} = \begin{cases} NewTask_j * (C_{ij} + C_{rk})\ if\ (r = k) \\ (NewTask_j * C_{ij}\ \in\ \Pi_i, NewTask_k * C_{rk}\ \in\ \Pi_r) \text{ otherwise} \end{cases} \tag{3.13}$$

As we mentioned previously, if two harmonic task are in the same implementation then the execution time of the $C_{new_{zl}}$ resulting task will be equal to the sum of the execution time of the merged task otherwise it will have different execution time depending on implementation in which it is executed. The constraint 3.14 computes the new period $T_{new_{zl}}$ which is equal to the minimum period between the merged tasks.

$$T_{new_{zl}} =\ min(T_{ij},\ T_{rk}) \tag{3.14}$$

The new priority P_{zl} is defined by the maximum priority between merged tasks.

$$P_{new_{zl}} =\ max(P_{ij},\ P_{rk}) \tag{3.15}$$

The CPU utilization factor is an important term in scheduling analysis. In order to ensure that the design model meets the timing constraints the constraint 3.17 must be verified

$$U\ = \leq \sum_{i \in \{1...m\}} \sum_{j \in \{1...N_i\}} N_i(2^{\frac{1}{N_i}} - 1) \tag{3.16}$$

Where U is defined by

$$U\ =\ \sum_{i \in \{1...m\}} U_i = \sum_{i\ \in \{1...m\}} \sum_{j\ \in \{1...N_i\}} \frac{C_{new_{ij}}\ +\ B_{new_{ij}}}{T_{new_{ij}}} \tag{3.17}$$

Where $B_{new_{ij}}$ presents the new blocking time which is defined by:

$$B_{new_{ijs}} = \begin{cases} -B_{ij}\ if\ \sum_{jk \in \{1...N_i\}}\ Mg_{jk} =\ 0 \\ -max\{B_{ij}, B_{ik}\} \text{ otherwise} \end{cases} \tag{3.18}$$

Where B_{ij} in (3.19) represents the local blocking time of task τ_j in implementation Π_i which is defined by $\forall\ \tau_k \in Hp_j,\ \forall \varphi_q \in \varphi$

$$B_{ij} = max\{C_{\varphi_{q_k}}\} - 1 \tag{3.19}$$

3.3.3.4 Response Time Optimization Model

In order to optimize the response time besides the minimization of the number of tasks, we define the following objective function

$$maximize \sum_{i\in\{1..m\}} (\sum_{j,k\in\{0,N_i\}} Mg_{jk} - \sum_{j\in\{0,N_i\}} \mathscr{R}_{ij}) \tag{3.20}$$

Response time $\mathscr{R}_{ij}$ of τ_{ij} is given by

$$\mathscr{R}_{ij} = C_{new_{ij}} + B_{new_{ij}} + \sum_{k\in Hp(j)} I_{ikj} * C_{new_{ik}} \tag{3.21}$$

where I_{ikj} the number of interference of τ_{ik} on τ_{ij} during its response time.

$$I_{ikj} = \lceil \frac{\mathscr{R}_{ij}}{T_{ik}} \rceil \tag{3.22}$$

Constraint 3.22 is non linear so in order to compute this constraint we start by adding the following variables:

$y_{ijk} = \begin{cases} N_i \text{ number of possible interference of } \tau_k \text{ on } \tau_j \text{ in } \Pi_i \\ 0, \text{ otherwise} \end{cases}$

y_{ijk} is equal to 0 if there are no interference of τ_k on τ_j in Π_i otherwise it is equal to number of interference of τ_k on τ_j in Π_i. The possible number of interference is defined as function of the response time and period by the following constraint

$$0 \leq y_{ijk} - \frac{\mathscr{R}_{ij}}{T_{ik}} \leq 1 \tag{3.23}$$

We define an additional variable x_{ijk} by

$x_{ijk} = \begin{cases} N_i \text{ number of possible interference of } \tau_k \text{ on } \tau_j \text{ in } \Pi_i \; if \; \mu_{ikj} = 1 \\ 0, \text{ otherwise} \end{cases}$

x_{ijk} is defined in constraints 3.24 and 3.25 in terms of y_{ijk} and μ_{ijk} by introducing the big M formulation (i.e., M is a big constant [129]).

$$y_{ijk} - M(1 - \mu_{ikj}) \leq x_{ijk} \leq y_{ijk} \tag{3.24}$$

$$0 \leq x_{ijk} \leq M * \mu_{ikj} \tag{3.25}$$

M is a constant larger than any other quantity involved in the constraint and it is typically used to encode alternative constraints that depend on a binary variable (the value of μ_{ikj} makes one of the constraints trivially true). μ_{ikj} is a binary variable, μ_{ijk} = 1 when τ_j is executed before τ_k it is equal to 0 otherwise. Finally,

the response time of task τ_j in Π_i can be computed as

$$\mathscr{R}_{ij} = C_{new_{ij}} + B_{new_{ij}} + \sum_{k \in \{1..N_i\}} x_{ijk} * C_{new_{ik}} \tag{3.26}$$

To sum up the full model of the response time optimization is given by:

$$\begin{cases}
Maximize \ \sum_{i \in \{1..m\}} (\sum_{j,k \in \{0,N_i\}} Mg_{jk} - \sum_{j \in \{0,N_i\}} \mathscr{R}_{ij}) \ (3.20) \\
\\
\forall j,k \in \{1..N_i\} \quad Mg_{jk} = 0 \quad if \quad (Hm_{jk} = 0) \ (3.10) \\
Mg_{jk} \leq 1, \ z \in \{1..N_i\}, \ k, z \neq j, \ Mg_{jk} + Mg_{zj} \leq 1 \ (3.11) \\
\forall \, j \in \{1 \dots N_i\}, \ NewTask_k = 1 - \sum_{j \in \{1 \dots N_i\}} Mg_{jk} \ (3.12) \\
\forall \, i,r,z \in \{1 \dots m\}, \ j,k,l \in \{1 \dots N\} \\
if \ (r = k) \ then \ C_{new_{zl}} = NewTask_j * (C_{ij} + C_{rk}) \ (3.13) \\
if \ (r <> k) \ then \ C_{new_{zl}} = \ (NewTask_j * C_{ij} \ in \ \Pi_i, NewTask_k * C_{rk} \ in \ \Pi_r) \ (3.13) \\
\forall \, i,r,z \in \{1 \dots m\}, \ j,k,l \in \{1 \dots N\} \ T_{new_{zl}} = \ min(T_{ij}, \ T_{rk}) \ (3.14) \\
\forall \, i,r,z \in \{1 \dots m\}, \ j,k,l \in \{1 \dots N\} \ P_{new_{zl}} = \ max(P_{ij}, \ P_{rk}) \ (3.15) \\
B_{new_{ij}} = B_{ij} \ if \ \sum_{jk \in \{1 \dots N_i\}} Mg_{jk} = 0 (3.18) \\
B_{new_{ij}} = max\{B_{ij}, B_{ik}\} \text{ otherwise} (3.18) \\
B_{ij} = max\{C_{\varphi_{q_k}}\} - 1 \ (3.19) \\
\forall \, j \in \{1 \dots N_i\}, \ NewTask_k = 1 - \sum_{j \in \{1 \dots N_i\}} Mg_{jk} \ (3.12) \\
0 \leq y_{ijk} - \frac{\mathscr{R}_{ij}}{T_{ik}} \leq 1 \ (3.23) \\
y_{ijk} - M(1 - \mu_{ikj}) \leq x_{ijk} \leq y_{ijk} \ (3.24) \\
0 \leq x_{ijk} \leq M\mu_{ikj} \ (3.25) \\
\mathscr{R}_{ij} = C_{new_{ij}} + B_{new_{ij}} + \sum_{k \in \{1..N_i\}} x_{ijk} * C_{new_{ik}} \ (3.26) \\
U \leq \sum_{i \in \{1 \dots m\}} \sum_{j \in \{1 \dots N_i\}} N_i (2^{\frac{1}{N_i}} - 1) \ (3.17)
\end{cases}$$

3.3.3.5 Energy consumption Optimization Model

For optimizing the energy consumption, we define the following objective function

$$Maximize \sum_{i \in \{1..m\}} (\sum_{j,k \in \{0,N_i\}} Mg_{jk} - \sum_{j \in \{0,N_i\}} E_{ij}) \tag{3.27}$$

As we mentioned previously in Section 3.2.4 in Eq. 3.8 the expression of E_{ij} is given by

$$E_{ij} = K \sum_{j \in \{0,N_i\}} \frac{C_{new_{n_{ij}}}}{\eta_j^2} \tag{3.28}$$

We notice that this equation is fractional due to the fact that the WCET of the task $C_{new_{n_{ij}}}$ is proportional to the reduction factor η_j. Thus, we simplify this program by maximizing the minimum of the reduction factor η_j. Hence, we introduce an

additional variable η_{min} which is equal to the minimum of η_j. The constraint 3.29 establishes that

$$\eta_{min} \ \leq \ \eta_j \tag{3.29}$$

The new normalized WCET $C_{new_{n_{ij}}}$ is given by

$$C_{new_{n_{ij}}} = \begin{cases} NewTask_j * (C_{n_{ij}} + C_{n_{rk}}) \ if \ (r = k) \\ (NewTask_j * C_{n_{ij}}, NewTask_k * C_{n_{rk}}) \text{ otherwise} \end{cases} \tag{3.30}$$

In order to confirm that the obtained model meets the timing constraints the following constraint must be verified:

$$U \ = \sum_{i \ \in \ \{1...m\}} \sum_{j \ \in \ \{1...N_i\}} \frac{C_{new_{n_{ij}}} \ \eta_j \ + \ B_{new_{ij}}}{T_{new_{ij}}} \leq \sum_{i \in \ \{1...m\}} \ \sum_{j \in \ \{1...N_i\}} N_i(2^{\frac{1}{N_i}} - 1) \tag{3.31}$$

The full formulation of blocking time optimization is given by

$$\begin{cases} Maximize \ \sum_{i \in \{1..m\}} (\sum_{j,k \in \{0,N_i\}} Mg_{jk}) \ + \eta_{min} \ \ (3.20) \\ \\ \forall j,k \in \{1..N_i\} \quad Mg_{jk} = 0 \quad if \quad (Hm_{sjk} = 0) \ (3.10) \\ Mg_{jk} \ \leq \ 1, \ z \ \in \ \{1..N_i\}, \ k, z \neq \ j, \ Mg_{jk} \ + \ Mg_{zj} \ \leq \ 1 \ (3.11) \\ \forall \ j \ \in \ \{1 \dots N_i\}, \ NewTask_k \ = \ 1 - \ \sum_{j \in \{1...N_i\}} Mg_{jk} \ (3.12) \\ \forall \ i,r,z \in \{1 \dots m\}, \ j,k,l \in \{1 \dots N\} \\ if \ (r = k) \ then \ C_{new_{n_{zl}}} = NewTask_j * (C_{n_{ij}} + C_{n_{rk}}) \ (3.30) \\ if \ (r <> k) \ then \ C_{new_{n_{zl}}} = (NewTask_j * C_{n_{ij}} \ \in \ \Pi_i \\ NewTask_k * C_{n_{rk}} \ in \ \Pi_r) \ (3.30) \\ \forall \ i,r,z \in \{1 \dots m\}, \ j,k,l \in \{1 \dots N\} \ T_{new_{zl}} = \ min(T_{ij}, \ T_{rk}) \ (3.14) \\ \forall \ i,r,z \in \{1 \dots m\}, \ j,k,l \in \{1 \dots N\} \ P_{new_{zl}} = \ max(P_{ij}, \ P_{rk}) \ (3.15) \\ B_{new_{ij}} = B_{ij} \ if \ \sum_{jk \in \ \{1...N_i\}} \ Mg_{jk} = \ 0 (3.18) \\ B_{new_{ij}} = max\{B_{ij}, B_{ik}\} \text{ otherwise} (3.18) \\ B_{ij} = max\{C_{\varphi_{q_k}}\} - 1 \ (3.19) \\ \eta_{min} \ \leq \ \eta_j \ (3.29) \\ 0 \ \leq \ \eta_j \ (3.30) \\ U \ \leq \ \sum_{i \in \ \{1...m\}} \ \sum_{j \in \ \{1...N_i\}} N_i(2^{\frac{1}{N_i}} \ - \ 1) \ (3.31) \end{cases}$$

3.3.4 Code Generation

The obtained task model from the previous step will be an input for the code generation step. The latter aims to generate the skeleton of the program which is based on POSIX code. Algorithm 2 formalizes this generation step. First, it generates the includes of the POSIX code. Then, for each task we generate a thread

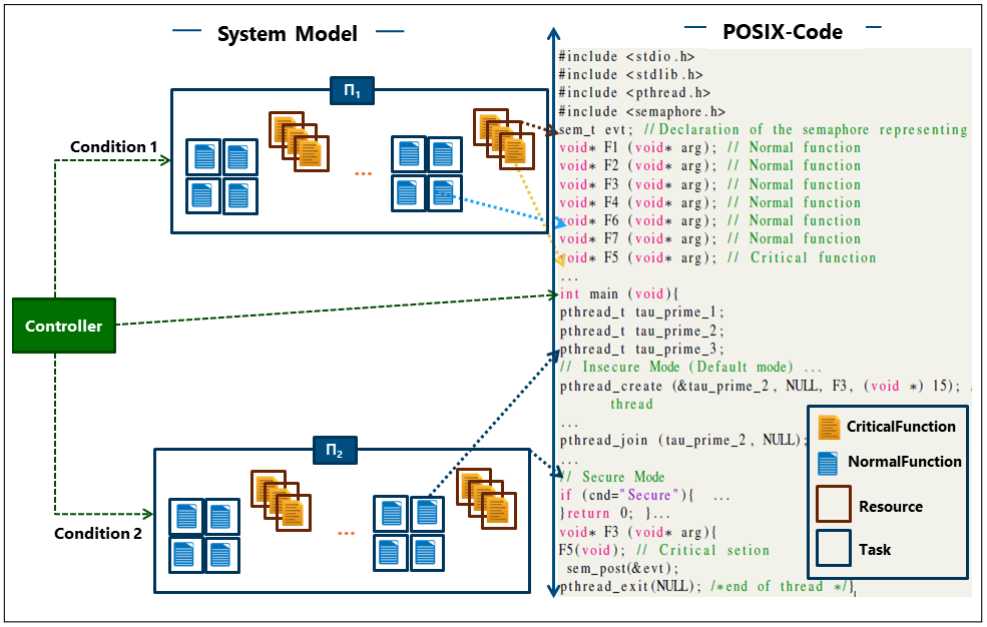

Fig. 3.6 Correspondence between system model and POSIX code.

(*pthread* in POSIX), and each function is implemented by a POSIX function. The controller which has the role of switching from an implementation to another, is implemented by the main function (i.e., int main (void)). The resources are implemented by critical functions, and we use a semaphore for the synchronization process. The complexity of this algorithm is $\mathcal{O}(N)$.

The transformation rules from the task model to POSIX code are depicted in Table 4.3 and in Figure 3.6.

3.4 Formal Case Study

Our methodology will be applied to a case study where the system will be modeled according to the above formalization. We consider a system Sys characterized in the specification level by two reconfiguration conditions and four functions: three normal functions (F_1, F_2, F_4) and one critical function (F_3). As shown in Figure 3.7 section **A**, the function F_1, F_2, and F_3 are executed in the condition $Cond_1$, while F_2, F_3, and F_4 in condition $Cond_2$. Table 3.5 depicts in details the specification model (i.e., function sets with its parameters). In the following subsections, we apply the proposed approach to the considered case study.

Algorithm 2: POSIX-CODE GENERATION

Input:
- Y: Mapping Matrix task to implementation
- X: Mapping Matrix of function to task
- F: List of function
- Cf: Function WCET vector
- m : Implementation number
- N: Task number

Output:
- PC: $POSIX\_Code$

```
1  /*** Generation Of Includes ***/
2  Write("#include <stdio.h>")
3  Write("#include <stdlib.h>")
4  Write("#include <pthread.h>")
5  Write("#include <semaphore.h>")
6  /*** Creation of Semaphore Declaration ***/
7  for each Function F[k] do
8      if F[k].type = critical then
9          Write("Sem_t evt[k]") /*** Declaration of Function ***/
10         else
11             Write("void* F[k] (void* arg);")
12 /*** Generation of the Controller ***/
13 Write("int main (void);")
14 /*** Generation of the Threads Declaration ***/
15 for j = 0 to N do
16     Write("pthread_t tau[j];")
17 for i = 0 to m do
18     Write("if imp[i]") for j = 0 to N do
19         if Y[ij] = 1 then
20             for each Function F[k] do
21                 if X[kj] = 1 then
22                     /
23                 *** if function k in task j ***/ Write ("pthread_Create(and tau[j], NULL, F[k], (void* ) Cf[k]
                       ;)")
24                 Write ("pthread_join(tau[j], NULL);")
25 if (i== m) then
26     Write("else")
27     for j = 0 to N do
28         if Y[ij] = 1 then
29             for each Function F[k] do
30                 if X[kj] = 1 then
31                     Write ("pthread_Create(and tau[j], NULL, F[k], (void* ) Cf[k] ;)")
32                     Write ("pthread_join(tau[j], NULL);")
33                     )
34 /*** Function ***/
35 for each Function F[k] do
36     if F[k].type = normal then
37         Write("void* F[i] (void* arg)") for each Function F[z] do
38             if F[k] depends on F[z] then
39                 Write("F[z];")
40                 Write("sem_post (andevt[k] );")
41 Write("pthread_exit (NULL) ;") return PC
```

Table 3.4 Correspondence between the task model and POSIX specific language.

Task Model	POSIX_Code
Task	pthreadt $task\_name$; pthread_create(); pthread_ join ();
Resource	Facility is provided by mutexes and condition variables. sem_t mutex; /* used for mutual exclusive access to waiting and busy*/ sem_t cond[]; /* used for condition synchronization*/ SEM_WAIT(andmutex); /* lockmutex */ SEM_POST(andmutex); /* release mutex */
Function	void* Function_name (void* arg)
Implementation	Functions set to be executed in each implementation
Controller	int main (void) {pthread_t thread_name if (Cnd$_i$){ pthread_create() . . . pthread_join() . . . return 0 }

Scheduling Policy	$\#define SCHED\_OTHER$
	/* implementation_defined scheduler
	(RM)*/
	int pthread_attr_setschedpolicy();
	int pthread_attr_getschedpolicy();
	/* set/get the contention scope attribute
	for a thread attribute object */

Table 3.5 Case Study Specification.

Function	T_{F_k}	Cn_{F_k}	$Type$	$\mathscr{F}_k$	Cond
F_1	20	1	Normal	F_3	$Cond_1$
F_2	10	2	Normal	F_3	$Cond_1$, $Cond_2$
F_3	15	1	Critical	-	$Cond_1$, $Cond_2$
F_4	20	1	Normal	F_3	$Cond_2$

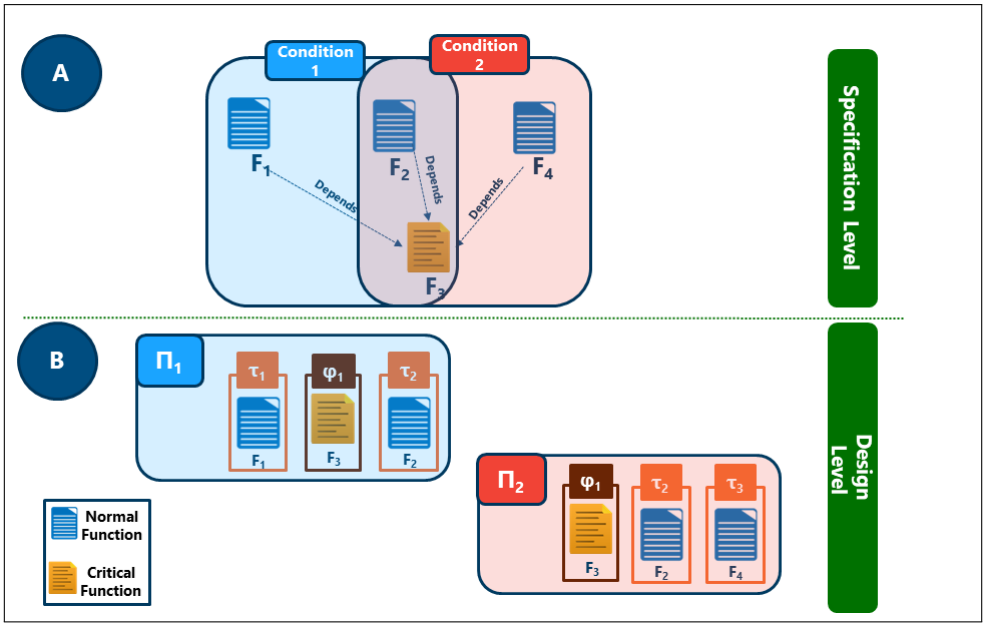

Fig. 3.7 Specification of Formal Case Study.

Table 3.6 Tabular description of the initial task model of the Case Study

τ_j/φ_q	Π_i	T_{ij}	Cij	Function
φ_1	Π_1	15	1	F_3
τ_1		20	1	F_1
τ_2		10	2	F_2
φ_1	Π_2	15	1	F_3
τ_2		10	2	F_2
τ_3		13	1	F_4

3.4.1 Initial Task Model

This step has as input the specification model, it aims to generate an initial task model by executing Algorithm 1. As shown in Figure 3.7 section **B**, each normal function is assigned to task and each critical function is mapped to a shared resource. For each condition, an implementation is generated. As a result, we obtain i) three tasks: τ_1, τ_2 , and τ_3 implement respectively F_1, F_2, and F_4, ii) one shred resource φ_1 implements F_3, and iii) two implementation Pi_1, and Pi_2 which execute respectively the set $\{\tau_1,\ \tau_2,\ and\varphi_1\}$ and $\{\tau_2,\ \tau_3\ \varphi_1\}$. Table 3.6 gives a tabular description of the initial task model describing the case study. As shown in Table 3.6, each normal function is affected to a task and each critical function is executed by a resource. Thus we obtain three tasks and one shared resource.

Table 3.7 Obatined Optimized Task Model in term of Total response time.

τ_j/φ_q	Π_i	T_{ij}	C_{ij}	B_{ij}	Total $\mathscr{R}_{Old}$	Total $\mathscr{R}_{New}$
φ_1	Π_1	20	1	-	4	3
τ_1		10	3	0		
φ	Π_2	20	2	-	-	-
τ_1		10	2	1	4	4
τ_3		13	1	0		

Table 3.8 Optimized Task Model in term of Energy Consumption.

τ_j/φ_q	Π_i	T_{ij}	Cn_{ij}	η_{ij}	$C_{new_{ij}}$	E_{old}	E_{new}
φ_1	Π_1	15	1	-	-	-	-
τ_1		10	3	0.7	2.1	431.5	302.05
φ_1	Π_2	15	1	-	-	-	-
τ_1		10	2	0.7	1.4	430	301
τ_3		13	1	0.7	0.7	250	175

3.4.2 Formal Case Study Optimized Models

The second step of MO^{2}R^{2}S approach consists in generating two optimized task models by executing MILP formulations defined previously. Both proposed linear programs produce the same merge matrix which is given as follow:

$$\begin{bmatrix} & \tau_1 & \tau_2 & \tau_3 \\ \tau_1 & 0 & 1 & 0 \\ \tau_2 & 0 & 0 & 0 \\ \tau_3 & 0 & 0 & 0 \end{bmatrix} \tag{3.32}$$

The MILP formulations allow to merge τ_1, τ_2. The resulting task model is composed of τ_1, τ_3, and φ_1. The result of the first MILP formulation which aims to reduce the total response time is shown in Table 3.7. Table 3.7 shows that the new response time is less than the old one. For the second objective function which aims to minimize energy consumption, the linear program generates the following task model depicted in Table 5.5. We can see from Table 5.5 that this model allows to reduce the energy consumption.

3.4.3 Formal Case Study POSIX Code

In order to implement the obtained task model in POSIX code, we apply Algorithm 2. The skeleton of the code implementation of the considered case study is represented by the following listing.

Listing 3.1: POSIX code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
sem_t evt; //Declaration of the semaphore representing the synchronization event
void* F1 (void* arg); //  Normal function
void* F2 (void* arg); // Normal function
void* F3 (void* arg); // Critical function
void* F4 (void* arg); // Normal function
int main (void){
pthread_t tau_1;
pthread_t tau_2;
pthread_t tau_3;
pthread_create (&tau_1, NULL, F1, (void *) 1);
pthread_create (&tau_1, NULL, F2, (void *) 2);
pthread_join (tau_1, NULL);
if (cnd=cnd1){
pthread_create (&tau_1, NULL, F2, (void *) 2);
pthread_join (tau_1, NULL);
pthread_create (&tau_3, NULL, F4, (void *) 1);
pthread_join (tau_3, NULL);
}return 0; }

void* F2 (void* arg){
F3(void);        // Critical section
 sem_post(&evt);
pthread_exit(NULL);} /* end of thread */
```

The presented skeleton in listing 1 helps the developer to implement the full code and detail the implementation of functions.

Conclusion

The initial version of MO^{2}R^{2}S approach offers a mechanism for reconfigurable real-time system synthesis under real-time constraints. This version addresses initially the mono-core architecture. The proposed approach allows to optimize some metric such as task count, energy consumption and total response time which conduct to reduce time overhead and code complexity as well. In the next chapter, the proposed approach is generalized to support multi-core platforms.

CHAPTER 4

Guided Implementation of Multi-core Reconfigurable Real-time Systems

Introduction

In this chapter, we generalize the MO^2R^2S approach to provide a guidance framework to assist designer in the synthesis of a reconfigurable real-time system with multi-core architecture. This version of MO^2R^2S approach is performed also by mixed-integer linear programming. This chapter is organized as follow: in Section 4.1, we give the motivation of this work and summarize its contributions. Section 4.2 formalizes the contributions and Section 4.3 details the methodology. In Section 5.2, we evaluate the proposed approach using a formal case study. The content of this chapter have been published in the international journal Information Sciences.

4.1 Motivation

Multi-core architectures are becoming more and more common in high performance real-time applications. They introduce additional challenges such as finding efficient solutions to the task allocation and scheduling problem especially when taking into consideration shared resources. Many of the previous works on task allocation in multi-core real-time systems ignored the effect of shared resources. The latter can introduce a significant blocking time. In multi-core reconfigurable real-time system, turning from an implementation to another produces a huge moving time overhead, that may affects the total stability of a system [68]. In addition, such system is generally specified by a large number of applicative functions which may cause as we mentioned in the previous chapter

- an important time overhead,
- many reconfigurations between the different implementations which increases the reconfiguration time [64],

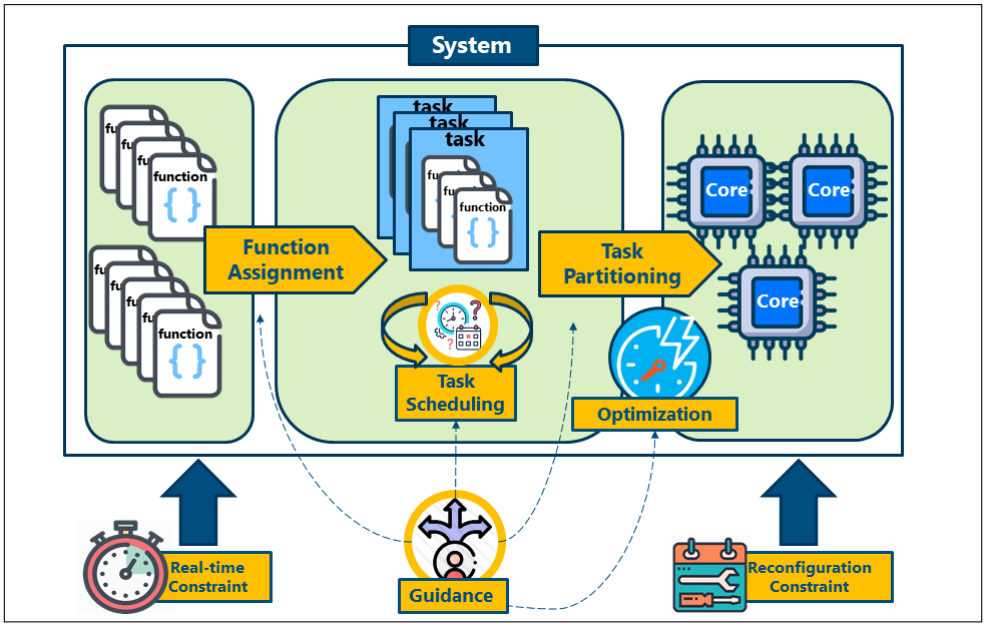

Fig. 4.1 Thesis Challenges.

- increase response time, and energy consumption,
- produce complex system code.

Furthermore, carrying out certain design steps, e.g., mapping functions to tasks, partitioning / scheduling tasks, and decision making in the case of non-feasible systems, is not a trivial task. The problem we deal with in this thesis can be summarized as shown in Figure 4.1:

- function to task assignment for a given functional specification,
- task partitioning into cores,
- task scheduling,
- multi-criteria optimization process,
- real-time constraints,
- reconfiguration constraint,
- guidance strategy.

The considered problems in this work are widely explored in literature but none of the proposed solutions simultaneously considers the real-time constraints, reconfiguration property, software architecture exploration, task partitioning, task scheduling, optimization concept and user guidance [68]. To overcome these limitations, the proposed approach aims to fulfill two main purposes:

- helping designers to deal with the multi-core real-time system design difficulties, and guide them to

    - creating, partitioning and scheduling tasks,
    - processing with the design modification due to reconfiguration actions.
- providing a feasible optimized synthesis of multi-core real-time system in term of
    - blocking time or moving time during a partitioning phase which leads to optimize the time overhead,
    - task count which conduct to minimize the reconfiguration time,
    - response time or energy consumption.

Thus, we propose a methodological guidance framework that assists designers during the synthesis of multi-core real-time systems which leads to optimize development time and reduce thereby the time to market. Note that this chapter generalizes the defined approach in the previous chapter to multi-core architecture. So comparing with the previous chapter i) it introduces the portioning issue and how to obtain an optimized one, and ii) it proposes a guidance framework that helps designer when feasibility problems appear.

4.2 Formalization

Reconfigurable real-time systems formalization introduced in the previous chapter is generalized in this chapter to support multi-core architecture. We extend the system model, real-time analysis (blocking time, CPU utilization, and response time), reconfiguration time model, and energy model proposed in the previous chapter to adequate models for the multi-core platforms.

4.2.1 System Model

In the specification level, the formalization consists of:

- hardware model which consists of *M* identical cores $\{\zeta_1, \zeta_2, \ldots, \zeta_M\}$ that share a common memory.
- software model which is composed of,
    - p functions $\xi_F = \{F_1, \ldots, F_p\}$,
    - m conditions [64] defining the execution modes of the system.

As we mentioned in the previous chapter, each function is characterized by real-time parameters (r_{F_k}, T_{F_k}, C_{F_k}, $Cond$, $Type$, $Nature$, Q_{F_k}, $\mathscr{F}_k$). Compared with the function characteristics presented in the previous chapter, we add

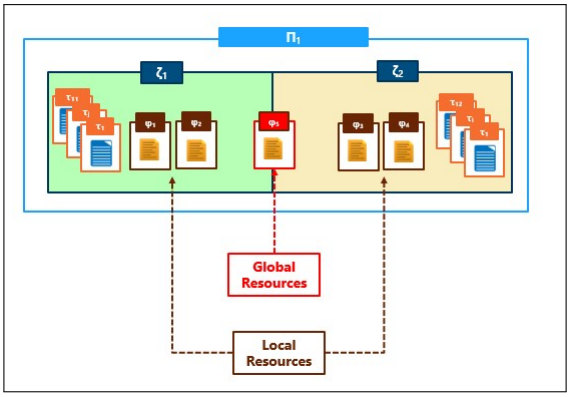

Fig. 4.2 Type of Shared Resources.

$Nature$ which defines the function type, i.e., hard or soft. , and Q_{F_k} which denotes the quality factor which is the degradation rate of scheduling performance (i.e., the percentage of instances of function F_k that do not meet their deadlines). This factor for any hard function should be equal to zero, i.e., Q_{F_k} =0.

The software model presents an entry point in task synthesis approach to provide a software architecture. The latter is composed of i) m implementations generated from conditions, ii) task set, and iii) shared resources among tasks. Each implementation Π_i is composed of

- N_i activated tasks which execute a set of normal functions. Each task is characterized by a tuple $(r_{ij},\ T_{ij},\ C_{ij},\ D_{ij},\ P_{ij},\ C_{\varphi_{q_j}},\ B_{ij},\ Nature,\ Q_{ij})$,
- R_i resources that can be shared among the tasks.

Note that we use a partitioning-based rate monotonic (RM) multi-core scheduling [11], in which no migration is allowed. Thereby, tasks are assigned to a particular core in which they have to be scheduled and executed. Concerning the shared resources, there are two different types: local and global resources (See Figure 4.2). Local resources in core ζ_s can be accessed just by local tasks to that core. Global resources are accessed by tasks assigned to different cores. We denote by Sys a multi-core reconfigurable real-time system which is defined by $Sys = (\xi_{imp}, C)$ (See Figure 4.3), where:

- ξ_{imp}: presents m implementations that define the system where $\xi_{imp} = \{\Pi_1, \ldots \Pi_m\}$. Each implementation Π_i is characterized by a M cores, N_i tasks and R_i shared resources where $N_i \leq N$ (respectively $R_i \leq R$),
- C represents the controller which manages, as depicted in Figure 4.3, the moving from one implementation to another under well defined reconfiguration conditions.

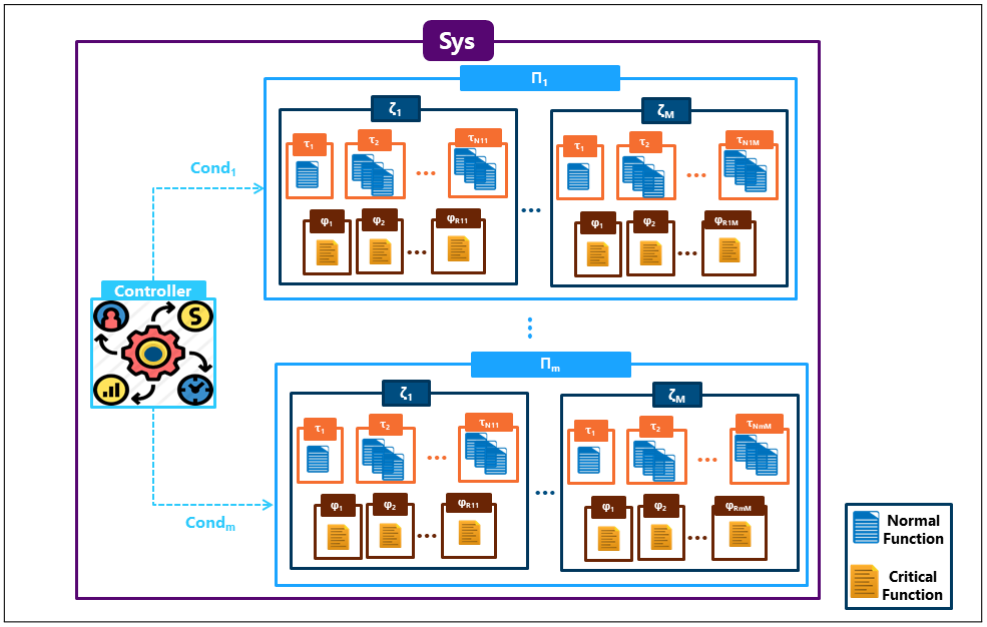

Fig. 4.3 Multi-core Reconfigurable Real-time system Model.

4.2.2 Real-time Analysis

The analysis results may compute three parameters: The blocking time, the processor utilization and response time.

4.2.2.1 Blocking time

The MO^{2}R^{2}S approach in multi-core architecture is based on *MPCP* for managing the access to shared resources. For local resources, *MPCP* performs like PCP [50]. Due to global shared resources, scheduled tasks would encounter distance other than local blocking. According to [124], blocking time B_{ij} consists of

- local blocking time $B_{ij,1}$,
- direct blocking time $B_{ij,2}$,
- indirect preemption delay $B_{ij,3}$,
- local preemption delay $B_{ij,4}$,

$$Bij = \sum_{k=1}^{4} B_{ij,k} \tag{4.1}$$

Note that a task may be blocked by tasks of lower or higher priorities. In contrast, in mono-core architecture, only lower-priority jobs cause blocking. We represent in Figure 4.4 and Figure 4.5 an example scenario of a trio-Core system with tasks accessing local and global resources in which those kind of blockage occur. Where i) L_1 is a local resource shared by τ_1 and τ_3, ii) G_1 is a global resource

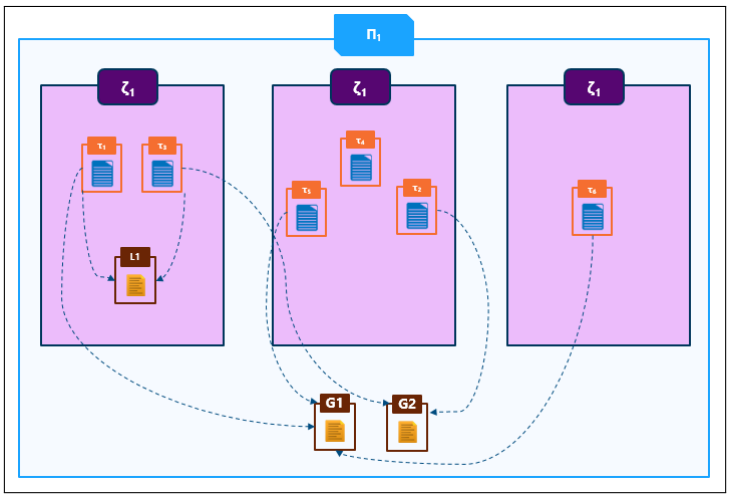

Fig. 4.4 Description of the system.

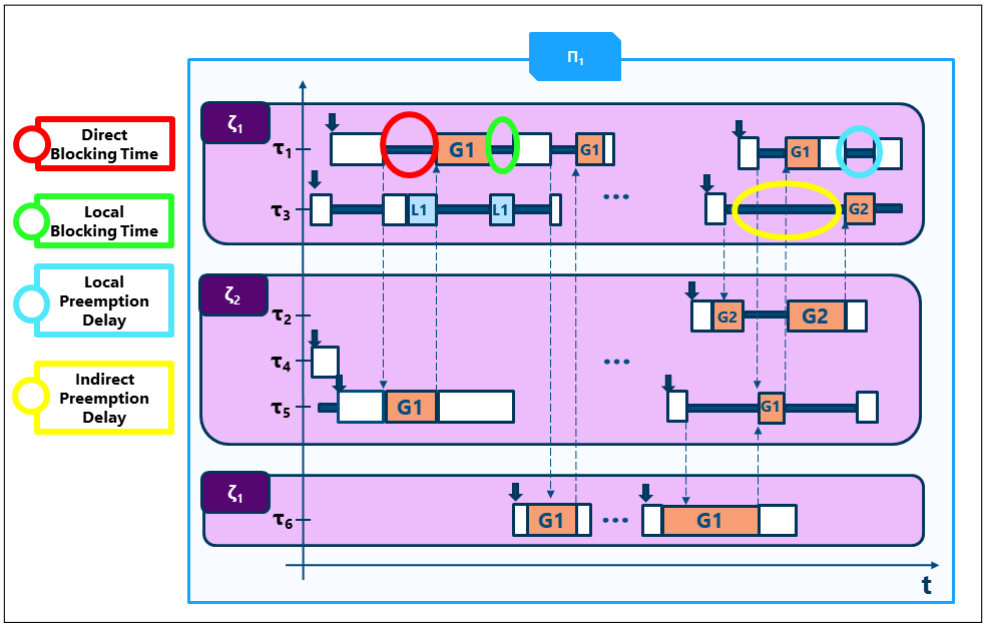

Fig. 4.5 Scheduling diagram.

shared by τ_1, τ_5, and τ_6, and iii) G_2 is a global resource shared by τ_3 and τ_2. As mentioned in Figure 4.5, τ_1 is locally blocked by τ_3 and directly blocked by τ_5 in ζ_2. In addition τ_3 is indirectly preempted by τ_5 which leads to leads to the local preemption of τ_1.

4.2.2.2 CPU Utilization Factor

The processor utilization factor of core ζ_s in implementation Π_i is denoted by U_{is}. It is given by

$$U_{is} = \sum_{j \in \{1..N_{is}\}} \frac{(C_{ij} + B_{ij})}{T_{ij}} \tag{4.2}$$

Where N_{is} is the number of tasks mapped to ζ_s in Π_i. In order to ensure that the design model meets the timing constraints the equation 4.3 must be verified

$$\forall i \in \{1..m\}, s \in \{1..M\}, U_{is} \leq N_{is}(2^{\frac{1}{N_{is}}} - 1) \tag{4.3}$$

4.2.2.3 Response Time

As we mentioned previously, the response time is one of the most important metric to be optimized in this thesis. We extend its expression by considering the multi-core notion. Let's $\mathscr{R}_{ijs}$ be the response time of the task τ_j in core ζ_s in implementation Π_i, it is given by
$\forall i \in \{1..m\}, s \in \{1..M\}, j \in \{1..N_{is}\}$

$$\mathscr{R}^0_{ijs} = 0, \quad \mathscr{R}^h_{ijs} = C_{ij} + B_{ij} + \sum_{l \in Hp(j)} \lceil \frac{\mathscr{R}^{h-1}_{ijs}}{T_{il}} \rceil C_{il} \tag{4.4}$$

4.2.3 Reconfiguration Time

We extend the reconfiguration time $\Re_i$ formula in the previous chapter to consider an additional time corresponding to the required time for a task's moving between cores. Thus, the reconfiguration time $\Re_i$ related to Π_i becomes [68]

$$\Re_i = (A + B) * T_{cost} + C * T_{move} \tag{4.5}$$

where *A* (resp, *B*) denotes the number of suspended tasks (resp, activated tasks), T_{cost} is the time spent to suspend/activate a task, *C* is the number of migrated tasks and T_{move} is the time spent to migrate from a core to another (See Figure 4.6). We aim to reduce the total reconfiguration time $\Re$

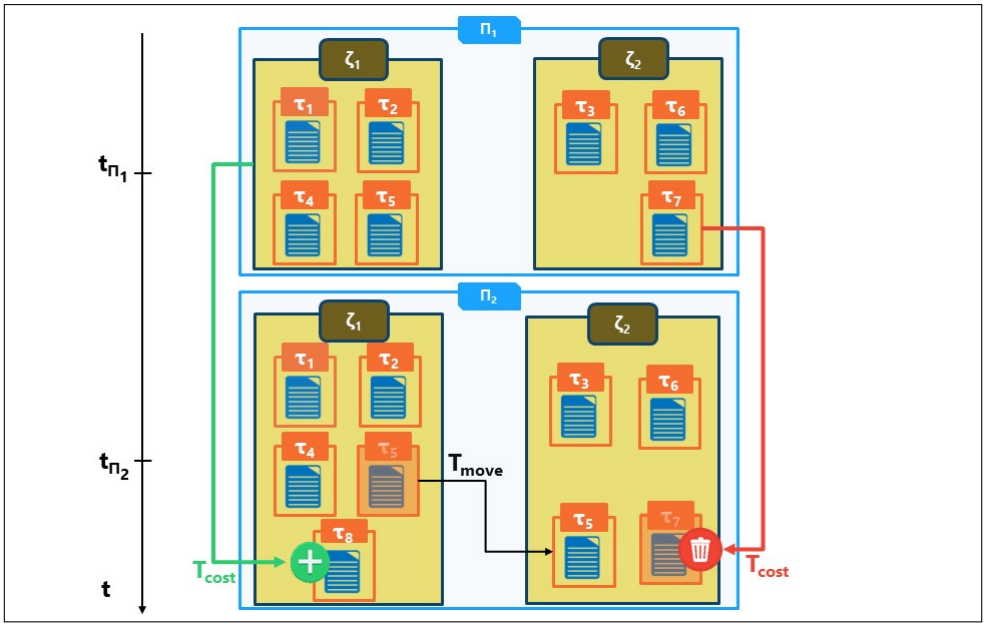

Fig. 4.6 Reconfiguration Time Scenario.

$$\Re = \sum_{1...m} \Re_i \tag{4.6}$$

Figure 4.6 shows an example of reconfiguration scenario which aims to compute $\Re$. The system is composed of two implementation Π_1 and Π_2, the first implementation Π_1 is defined by $\Pi_1 = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7\}$ to move to the second implementation Π_2 we need to suspend τ_7 and activate τ_8. So the reconfiguration time $\Re_2$ related to Π_2 is given by: $1 * T_{cost} + 1 * T_{cost} + 1 * T_{move}$, so $\Re_2 = 2 * T_{cost} + 1 * T_{move}$. If we assume that $Tcost = 5ms$ and $T_{move} = 10ms$, then $\Re_2 = 2 * 5 + 10 = 20\ ms$

4.2.4 Energy Consumption Model

In this chapter, the energy consumption model is adapted to the multi-core architecture by adding a definition of the energy consumption of each core ζ_s. Thus, we define E_{is} in core ζ_s by

$$E_{is} = K \sum_{j \in \{0, N_{is}\}} \frac{C_{n_{ij}}}{\eta_j^2} \tag{4.7}$$

where $K = V_n^2 f_n$, and N_{is} the number of task in implementation Π_i in core ζ_s.

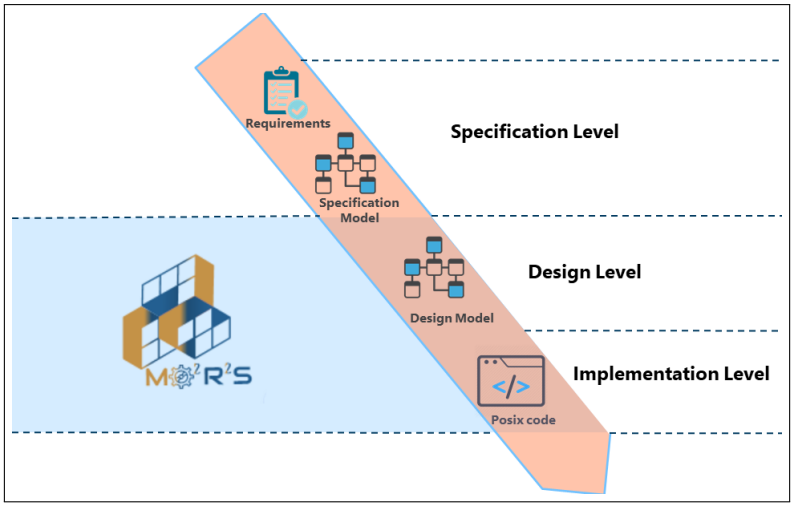

Fig. 4.7 The MO^{2}R^{2}S Process in the Synthesis of Reconfigurable Real-time Systems Flow.

4.3 Contribution Description

We first present an overview of the extended version of MO^{2}R^{2}S approach. We give then a detailed description of the different modules involved in this methodology.

4.3.1 MO^{2}R^{2}S Global Overview

In this section, we present an overview of our MO^{2}R^{2}S approach in multi-core architecture. The basic flow for the synthesis of real-time systems under reconfiguration constraints is illustrated in Figure 4.7. As shown in Figure 4.7, the implementation of multi-core real-time systems under reconfiguration constraints consists of three level i) specification level in the designer specifies the functional and non-functional proprieties of a system from the requirement set, ii) design level in which, the designer models the functional requirements, and iii) implementation level in which, the design model is transformed into code (POSIX in our case). MO^{2}R^{2}S approach revolve around design and implementation levels. As illustrated in Figure 4.8, the proposed framework provides three possible modes (.i.e., scenarios):

- 1- Normal mode: it is executed, when no real-time feasibility problems appear. It proposes two solution bases (i.e., software architectures) by optimizing in the first place the partitioning step then the local placement.
- 2- Resizing mode: the framework proceeds this mode when real-time feasibility problems occur and the designer chooses to change the hardware architecture by rising the number of cores to solve feasibility problems,

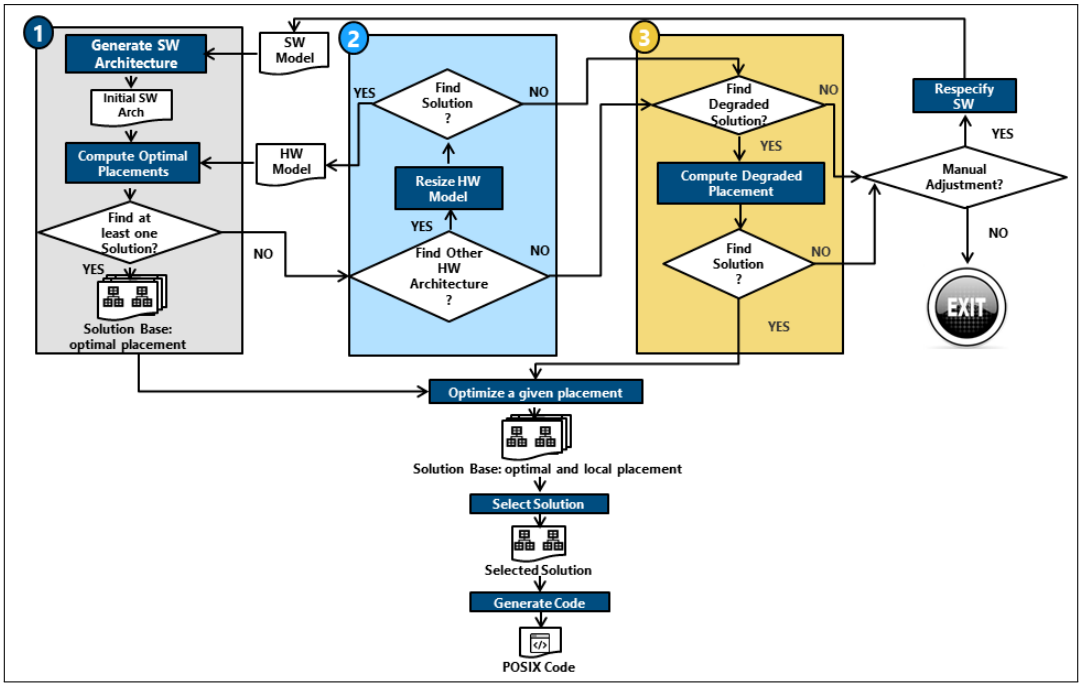

Fig. 4.8 MO^{2}R^2S methodology.

– 3- Degrading mode: same as the resizing mode it is executed when no feasible solution is found by the framework and the user chooses to degrade the quality of soft tasks (i.e., by increasing the percentage of instances of task τ_j that do not meet their deadlines). If no solution exists, the designer has to manually adjust software model parameters.

We explain the three modes in more detail in the following sections.

4.3.2 Normal Mode

As shown in Figure 4.9, this scenario consists of four basic steps:

– generation of software architecture (SW architecture),

– computation of optimal placements in term of blocking time and system stability,

– local optimization is performed locally in each processor and aims to optimize task number as well as the response time and energy consumption,

– generation of code from the selected solution.

4.3.2.1 SW Architecture Generation

The first step aims to generate an initial software architecture from the SW model which consists of functions and reconfiguration conditions (See Figure 4.10).

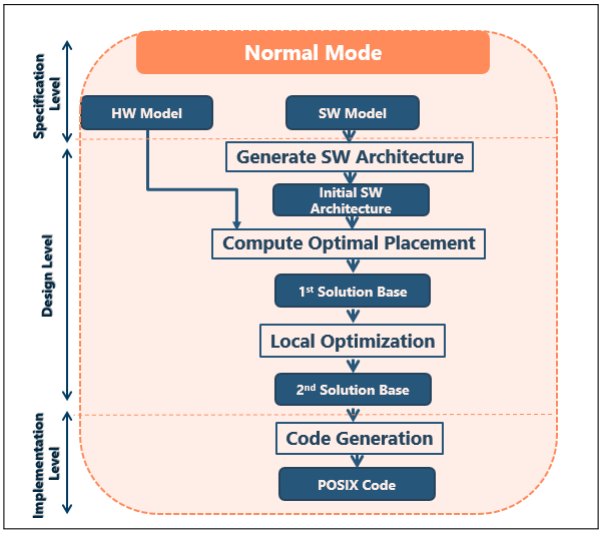

Fig. 4.9 Normal Mode Process.

This step is performed by the algorithm defined in the previous Chapter. As shown in Figure 4.10, it enables to generate:

- the implementation set for each reconfiguration condition,
- the task set from each normal function,
- the shared resource from each critical function.

As the next step (i.e., computation of optimal placement) needs an initial solution to start the search procedure, the SW architecture presents the input of this step. Note that in this step the feasibility concerns are not inspected. In addition, the number of tasks is equal to number of functions, no clustering technique is applied in this step.

4.3.2.2 Computation of Optimal Placement

Once the the initial software architecture is generated, we come to the partitioning step which has as input the software architecture and the hardware model specified by the designer (see Figure 4.8n and Figure 4.9). This step aims to assign tasks into cores while optimizing the blocking time and the system stability by reducing the moving time in order to generate the first solution base (See Figure 4.11). Note that the proposed approach is extensible to add other metrics. This step is implemented by two mixed integer linear programming (MILP) formulations and it is performed by CPLEX solver [79]. Thus, it is necessary to describe the problem using i) decision variables which present the set of design choices under the designer control, ii) constants or parameters which are the input of the MILP model, iii) constraints which define the feasibility region, and the domain

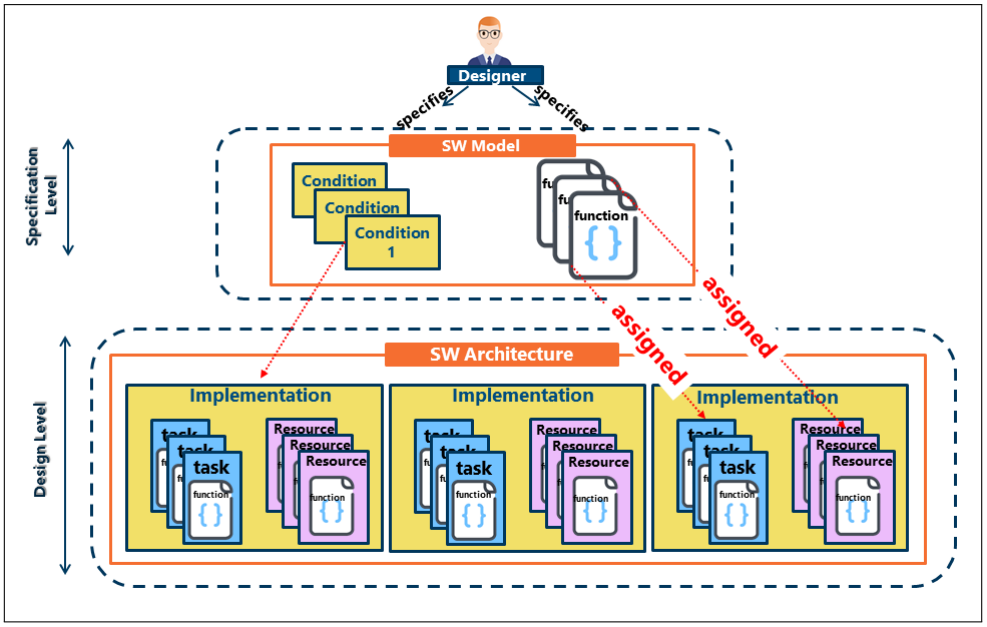

Fig. 4.10 SW Architecture Generation.

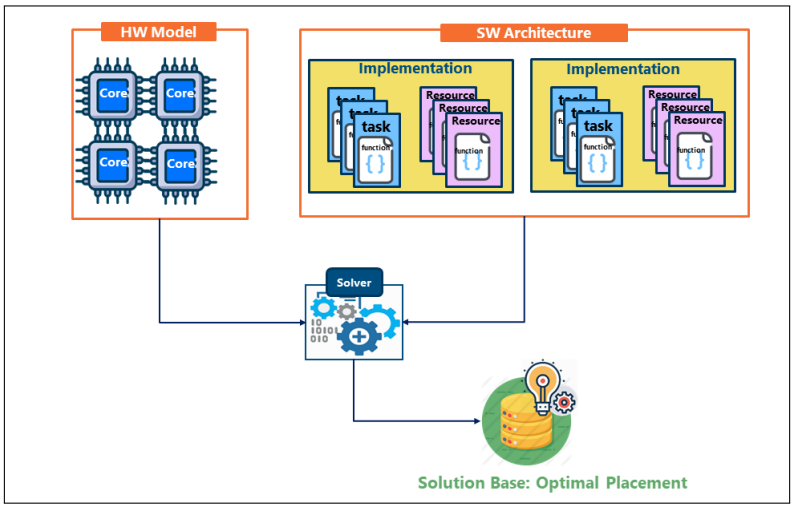

Fig. 4.11 Computation of Optimal Placement.

of the allowed values for the decision variables, iv) and an objective function which represents the optimization goal.

MILP Formulation for Blocking Time Optimization

We aim to map task to cores while minimizing blocking time, to do so we try to place tasks with shared resources in the same core. In other words, we aim to reduce the number of global resource which leads to minimize to total blocking time as well.

All model parameters and variables are depicted in the following table 4.1: Minimizing the number of global resource means maximizing the number of local resources. Thus, we define a binary variable ϱ_{il} such as

Table 4.1 First Model Parameters and Variables.

Constants	
Concepts	Definition
N	Number of task
m	Number of implementation
M	Number of core
R	Number of shared resource
X_{ijl}	A boolean variable used to mention whether τ_j in Π_i uses resource φ_l
C_{ij}	Task's WCET
D_{ij}	Task's Deadline
Variables	
Concepts	Definition
ν_{il}	Number of tasks that use the resource φ_l
Ur_{il}	Processor utilization factor of tasks that share resource φ_l
ϱ_{il}	Boolean variable used to mention if the set of tasks that share resource φ_l can be executed on the same core
N_s	Number of tasks in core ζ_s
Y_{ijs}	Boolean variable used to mention whether τ_j is executed in Π_i in core ζ_s

ϱ_{il} = $\begin{cases} 1\ if\ tasks\ that\ shared\ \varphi_l\ could\ be\ in\ the\ same\ core, \\ 0\ otherwise. \end{cases}$ The objective function 4.8 aims to maximize the sum of ϱ_{il}.

$$maximize \sum_{i \in \{0..m\}} \sum_{l \in \{0..R_i\}} \varrho_{il} \tag{4.8}$$

Constraints 4.9 and 4.10 compute $\varrho_{il}, \forall l \in \{1..R_i\}$ in implementation Π_i

$$if(\sum_{j \in 1..N_i} (X_{ijl} * C_{ij})/D_{ij}) - Ur_{il} <= 0\ then\ \varrho_{il} = 1; \tag{4.9}$$

$$if(\sum_{j \in 1..N_i} (X_{ijl} * C_{ij})/D_{ij}) - Ur_{il} > 0\ then\ \varrho_{il} = 0; \tag{4.10}$$

The both equations above ensure that tasks with shared resource could be in the same core only if the feasibility constraint is respected. The expression of processor utilization factor Ur_{il} of tasks that share resource φ_l in implementation Π_i is given by
$\forall l \in \{1..R_i\}$ in implementation Π_i

$$Ur_{il} = \nu_{il} * (2^{(1/\nu_{il})} - 1); \tag{4.11}$$

Where ν_{il} presents the number of tasks that use the resource φ_l in implementation Π_i. The expression of ν_{il} is given by

$$\nu_{il} = \sum_{j \in 1..N_i} X_{ijl}; \tag{4.12}$$

Where X_{ijl} the boolean variable is defined by
X_{ijl} = $\begin{cases} 1\ if\ tasks\ \tau_j\ in \Pi_i\ used\ \varphi_l, \\ 0\ otherwise. \end{cases}$ Constraint 4.13 ensures to assign tasks with shared resource φ_l to the same cores $\zeta_s\ \forall i \in \{1..m\},\ s \in \{1..M\}\ \forall l \in \{1..R_i\}\ \forall j \in \{1..N_i\}$,

$$if(X_{ijl} * \varrho_{il} == 1),\ then\ Y_{ijs} = 1; \tag{4.13}$$

As we mentioned previously, due to global resource, tasks would encounter distance other than local blocking. Thus, mapping tasks with shared resource to same core allows to minimize (simultaneously maximize) the global shared resource (simultaneously maximize local shared resource) thus minimize the blocking time as well.

In order to ensure that every task must be assigned to a single core, we define the

Table 4.2 Second Model Parameters and Variables.

Constants	
N	Number of Tasks
m	Number of implementation
M	Number of core
C_{ij}	Task's WCET
D_{ij}	Task's Deadline
Variables	
Concepts	Definition
N_s	Number of tasks in core ζ_s
Y_{ijs}	Boolean variable used to mention whether τ_j is executed in Π_i in core ζ_s

constraint 4.14

$$\sum_{s \in 1..M} Y_{ijs} \ <= \ 1; \tag{4.14}$$

Ensuring the feasibility of the system is an important constraint thus, we define constraint 4.15 to respect the feasibility constraint, $\forall s \in \{1..M\}\ \forall i \ \in \{1..m\}$,

$$\sum_{j \in 1..N} Y_{ijs} * C_{ij}/D_{ij} <= N_s * (2^{(1/N_s)} - 1); \tag{4.15}$$

Where N_s the number of tasks in core ζ_s is given by constraint 4.16, $\forall s \ \in \{1..M\}\ \forall i \ \in \{1..m\}$,

$$N_s = \sum_{j \in 1..N_i} Y_{ijs}; \tag{4.16}$$

MILP Formulation for System Stability Optimization

In order to optimize the stability of the system, we aim to minimize the moving time (i.e., the time spent to migrate from a core to another when switching from one implementation to another) which leads also to reduce the reconfiguration time (See Subsection 4.2.3). Table 4.2 depicts the model parameters and variables: In order to minimize the moving time of task from a core to another when switching from one implementation to another, we try by the following formulation to assign tasks to the same core for all implementation. Equation 4.17 defines the objective function: $\forall\ i \ \in \{1..m\},\ j \ \in \{1..N_i\}$,

$$minimize \sum_{j \in \{0..N_i\}} \sum_{s \in \{0..M\}} Y_{ijs} \tag{4.17}$$

In order to ensure that the task τ_j is assigned to just one core ζ_s when moving from one implementation to another, we define the following constraint 4.18:

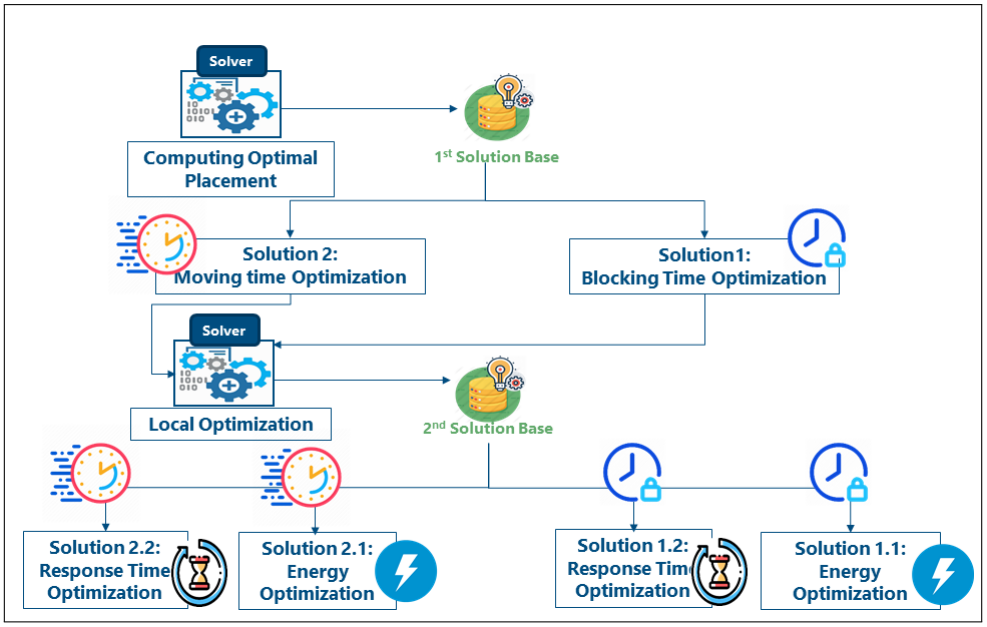

Fig. 4.12 MO^2R^2S Solution Bases.

$\forall i \in \{1..m-1\}, \forall j \in \{1..N_i\},\ \forall s \in \{1..M\},$

$$Y_{ijs} - Y_{i+1,j,s} = 0; \tag{4.18}$$

Constraint 4.19 assures that every task τ_j in implementation Π_i should be executed in the same core ζ_s

$$\forall i \in \{1..m\}, j \in \{1..N_i\}, \sum_{s \in \{1..M\}} Y_{ijs} <= 1; \tag{4.19}$$

Constraint 4.20 is defined to ensure the feasibility of the system
$\forall i \in \{1..m\},\ s \in \{1..M\},$

$$\sum_{j \in \{1..N_i\}} (C_{ij}/D_{ij}) Y_{ijs} <= N_s(2^{(1/N_s)} - 1); \tag{4.20}$$

4.3.2.3 Local Optimization

Once we have at least one solution from the previous step, we apply a second optimization technique in each local placement (i.e., core) to generate a second solution base. As shown in Figure 4.12, the second solution base is composed at most of four solutions such as for each placement obtained from the previous step we try to generate two solutions. This local optimization step aims to minimize the number of task while optimizing either i) response time, or ii) energy consumption. As we mention in the Subsection 4.3.1, this step is performed by

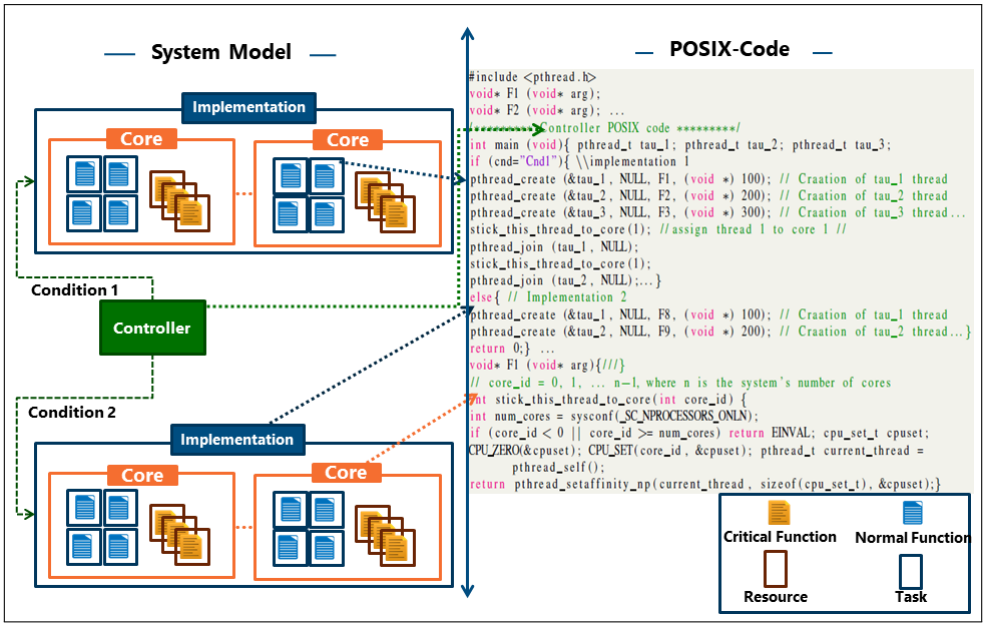

Fig. 4.13 Correspondence between task model and POSIX code.

the mono-core based approach defined in the previous chapter. The generalized version of MO^2R^2S for multi-core architecture offers the designers the ability to choose one solution from the solution base by affording them the characteristics of each one. The MILP formulation of the local optimization is presented in Appendix A.

4.3.2.4 Code Generation

Once the optimized model is generated, we generate the corresponding skeleton of POSIX code. The code generation process is provided by Algorithm 3. First, it generates the includes of the POSIX code. Then it generates for each i) function a POSIX function, ii) task a thread (pthread), iii) resource a critical function, and we use a semaphore for the synchronization process. The controller is presented by a main function that manages the switching of implementations following well defined conditions (i.e., user requirements). We extend the initial version of this step (i.e., in mono-core architecture) by adding a function *stick_this_thread_to_core()* that ensures the mapping of tasks to cores. The transformation rules from the task model to POSIX code are depicted in Table 4.3 and Figure 4.13. Figure 4.14 describes the POSIX code structure in a UML class diagram where the system is represented by a UML class called System. This class is composed of implementations' set (class Implementation) and a controller (class Controller) which executes the Implementation class. The System class defines an *IsScheduled* attribute to specify the used scheduling policy which is RM in this thesis. The Implementation class is composed of cores' set (class core). The

Algorithm 3: POSIX-CODE GENERATION

Input:
- Y: Mapping Matrix of tasks to implementations and cores
- X: Mapping Matrix of function to task
- F: List of function
- Cf: Function WCET vector
- m : Implementation number
- M: Core number
- N: Task number

Output:
- PC: $POSIX\_Code$

```
1  /*** Generation Of Includes ***/
2  Write("#include <stdio.h>")
3  Write("#include <stdlib.h>")
4  Write("#include <pthread.h>")
5  Write("#include <semaphore.h>")
6  /*** Creation of Semaphore Declaration ***/
7  for each Function F[k] do
8      if F[k].type = critical then
9          Write("Sem_t evt[k]") /*** Declaration of Function ***/
10         else
11             Write("void* F[k] (void* arg);")
12 /*** Generation of the Controller ***/
13 Write("int main (void);")
14 /*** Generation of the Threads Declaration ***/
15 for j = 0 to N do
16     Write("pthread_t tau[j];")
17 for i = 0 to m do
18     Write("if imp[i]") for s = 0 to M do
19         for j = 0 to N do
20             if Y[ijs] = 1 then
21                 /*** if task j in implemntation i and core s ***/ for each Function F[k] do
22                     if X[kj] = 1 then
23                         /
24                     *** if function k is assigned ti task j ***/ Write ("pthread_Create(and tau[j], NULL, F[k],
                         (void* ) Cf[k] ;)")
25                     Write ("pthread_join(tau[j], NULL);")
26                     Write ("stick_this_thread_to_core(s) ; // assign thread j to core s")
27 if (i== m) then
28     Write("else")
29     for s = 0 to M do
30         for j = 0 to N do
31             if Y[ijs] = 1 then
32                 for each Function F[k] do
33                     if X[kj] = 1 then
34                         Write ("pthread_Create(and tau[j], NULL, F[k], (void* ) Cf[k] ;)")
35                         Write ("pthread_join(tau[j], NULL);")
36                         Write ("stick_this_thread_to_core(s) ; // assign thread j to core s")
37 /*** Function ***/
38 for each Function F[k] do
39     if F[k].type = normal then
40         Write("void* F[i] (void* arg)") for each Function F[z] do
41             if F[k] depends on F[z] then
42                 Write("F[z];")
43                 Write("sem_post (_evt[k] );")
44 /*** Mapping task core function ***/
45 Write("int stick_this_thread_to_core(intcore_id){")
46 Write("int num_cores = sysconf(_SC_NPROCESSORS ONLN);")
47 Write("if (core_id < 0 ||| core_id >= num_cores )")
48 Write("return EINVAL;")
49 Write("cpu_set_t cpuset;")
50 Write("CPU_ZERO(andcpuset);")
51 Write("CPU_SET( core_id , andcpuset );")
52 Write("pthread_t current_thread = pthread_self();")
53 Write("return pthread_setaffinity_np (current_thread, sizeof(cpu_set_t ) , andcpuset );}")
54 Write("pthread_exit (NULL) ;") return PC
```

Table 4.3 Correspondence between the task model and POSIX specific language.

Task Model	POSIX_Code
Task	pthreadt~ $task\_name$; pthread_create(); pthread_ join ();
Resource	Facility is provided by mutexes and condition variables. sem_t mutex; /* used for mutual exclusive access to waiting and busy*/ sem_t cond[]; /* used for condition synchronization*/ SEM_WAIT(andmutex); /* lockmutex */ SEM_POST(andmutex); /* release mutex */
Function	void* Function_name (void* arg)
Implementation	Functions set to be executed in each implementation
Controller	int main (void) {pthread_t thread_name if (Cnd$_i$){ pthread_create() . . . pthread_join() . . . return 0 }

Scheduling Policy	$\#defineSCHED\_OTHER$
	/* implementation_defined scheduler
	(RM)*/
	int pthread_attr_setschedpolicy();
	int pthread_attr_getschedpolicy();
	/* set/get the contention scope attribute
	for a thread attribute object */

Core	stick_his_hread_o_ore (core_d) ;

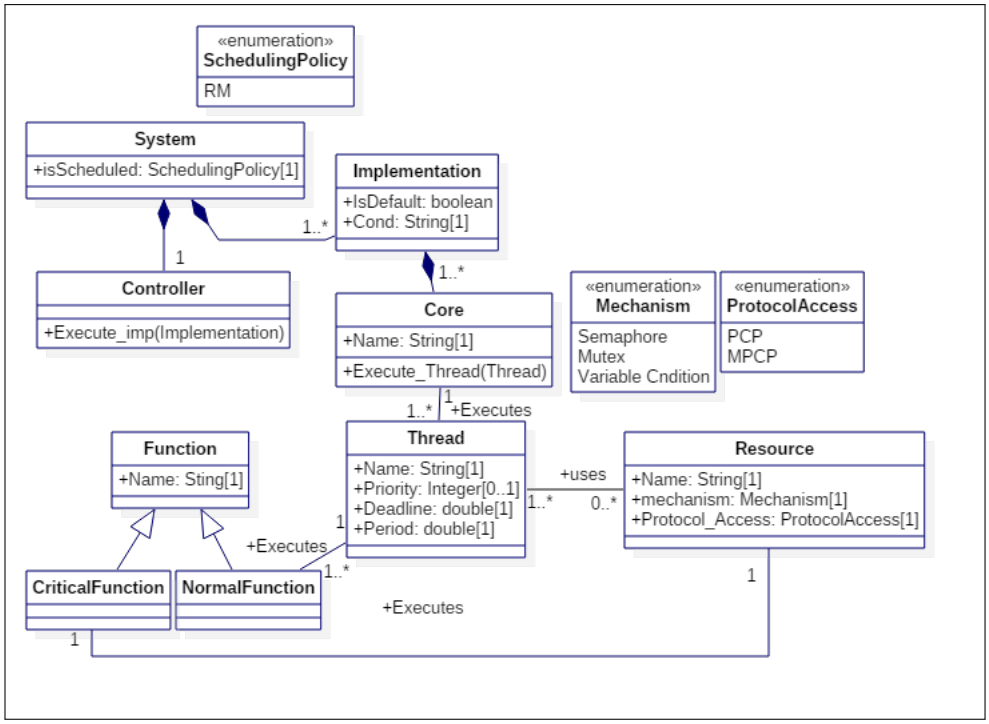

Fig. 4.14 UML class diagram describing the Skeleton of POSIX Code.

core class defines an *Execute Thread* method to execute the Thread class. The Thread executes one or more Function classes. It could share the Resource class which implements the CriticalFunction class. The resource class is characterized by *Mechanism* and *Protocol Access* attributes.

4.3.3 Resizing Mode

In case where the framework could not find any feasible solution for the initial architecture and the designer chooses to resize the HW architecture (See Figure 4.8), the tool increases the number of cores until it finds the minimal number that ensures the feasibility of the system by implementing Algorithm 4. In this algo-

Algorithm 4: Resizing HW model

Input:
- SW architecture
- Core_Number

Output:
New HW Model

```
1 while First_Fit (SW architecture) == false do
2     Core_Number++;
3 return New HW Model
```

rithm we apply the first fit algorithm to find a feasible partitioning with minimal number of cores. Note that in this step we just look for a feasible placement not optimal one. As shown in Figure 4.15 the tool executes again the normal mode steps.

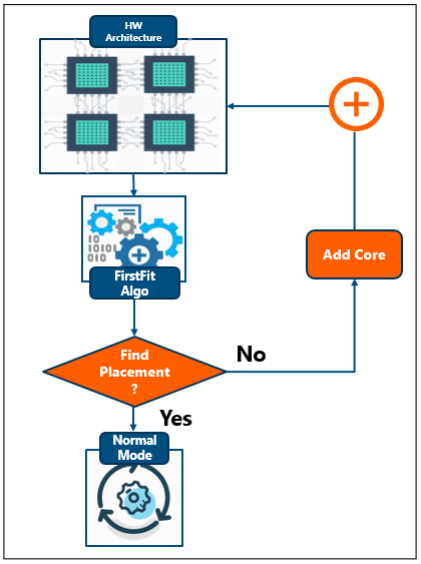

Fig. 4.15 Resizing Mode.

4.3.4 Degrading Mode

Whenever the partitioning step could not find any feasible solution and the designer chooses to degrade the performance of the system, the framework executes the degrading mode. In this mode, some soft tasks may miss their deadlines. In order to quantify the degradation, we define the metric Q_{ij} which is the degradation rate of scheduling performance (i.e., the percentage of instances of task τ_j that do not meet their deadlines). This factor for any hard task should be equal to zero, i.e., Q_{ij} =0. As shown in Figure 4.16, degrading mode is defined by three steps:

- Computation of degraded placement,
- Optimization of the given placement and generation of the solution base,
- Generation of POSIX code,

The first step which is computation of degrades placement is performed by MILP formulation. Note that if the solver could not find any feasible solution the designer has to adjust manually the SW model parameters The objective function is given by equation 4.21 which aims to minimize the degradation rate Q_{ij} while partitioning and scheduling task sets.

$$minimize \sum_{i \in \{1..m\}} \sum_{j \in \{1..N_i\}} Q_{ij} \tag{4.21}$$

The following Table 4.4 defines all model parameters and variables: Constraint 4.22 assures the mapping of tasks to the cores by respecting feasibility constraint.

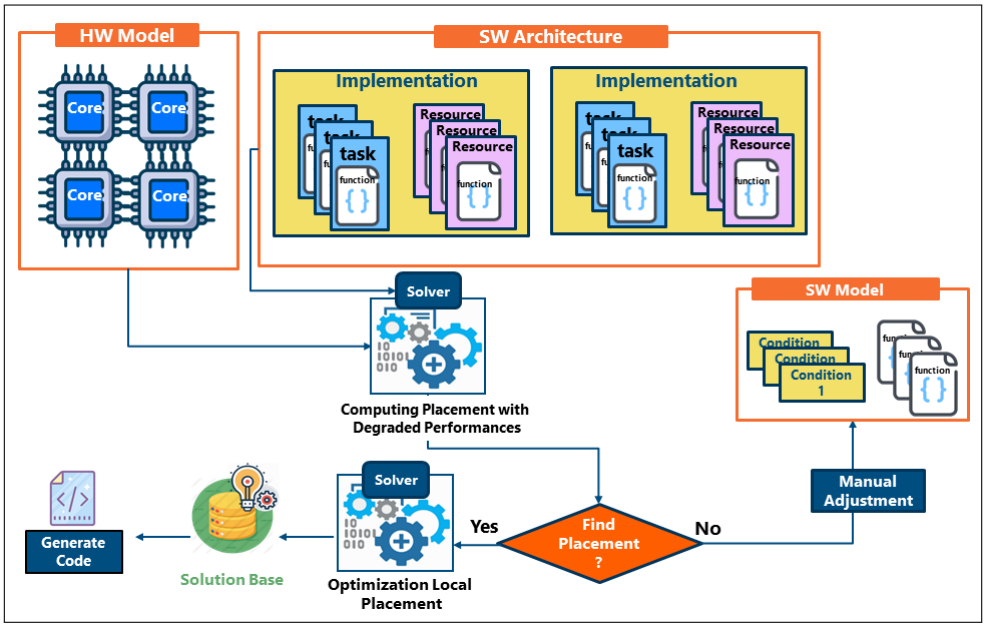

Fig. 4.16 Degraded Mode.

Table 4.4 Degrading Model Variables and parameters

Constants	
Concepts	Definition
N	Number of tasks
m	Number of implementations
M	Number of cores
C_{ij}	Task's WCET
D_{ij}	Task's Deadline
$Nature_{ij}$	A boolean variable used to mention if task τ_j is hard or soft
Variables	
Concepts	Definition
N_s	Number of tasks in core ζ_s
Y_{ijs}	Boolean variable used to mention whether τ_j is executed in Π_i in core ζ_s
Q_{ij}	Quality factor

$\forall\ s \in \{1..M\},\ \forall\ i \in \{1..m\},$

$$\sum_{j \in \{1..N_i\}} (\frac{C_{ij}}{D_{ij}}) * Q_{ij} * Y_{ijs} <= N_s(2^{\frac{1}{N_s}} - 1); \tag{4.22}$$

We define constraint 4.23 to make sure that every task is assigned to just one core in an implementation Π_i

$$\sum_{i \in \{1..m\}} \sum_{j \in \{1..N_i\},\ s \in \{1..M\}} Y_{ijs} == 1 \tag{4.23}$$

$$\forall\ i \in \{1..m\},\ j \in \{1..N_i\},\ 0 \leq Q_{ij} \leq 1; \tag{4.24}$$

Constraint 4.24 ensures that the quality factor Q_{ij} must be between 0 and 1. In order not to affect the performance of hard tasks and to guarantee that all its instances have to meet their deadlines, constraints 4.25 and 4.26 are defined to do so.

$$\forall\ i \in \{1..m\},\ j \in \{1..N_i\},\ if Nature_{ij} = 1 then Q_{ij} = 0; \tag{4.25}$$

$$\forall\ i \in \{1..m\},\ j \in \{1..N\},\ if Nature_{ij} = 0 then Q_{ij} \leq 1; \tag{4.26}$$

Where $Nature_{ij}$ as we defined previously in Section 3.1 defines the task type whether it is hard ($Nature_{ij}$ = 0) or soft ($Nature_{ij}$ = 1). After executing this step, the framework move to the third and fourth steps of normal mode defined in subsections 4.3.2.3 and 4.3.2.4.

4.4 Formal Case Study

Given an example of multi-core reconfigurable real-time system denoted by *Sys*. Its hardware model consists of 2 identical cores $\alpha\zeta = \{\zeta_1, \zeta_2\}$. The software model of *Sys* consists of 10 functions $\alpha F = \{F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8, F_9, F_{10}\}$. Let us suppose that the dependency between function is as following: $\mathscr{F}_1 =$, $\mathscr{F}_2 = \{F_1, F_3\}$, (i.e., F_2 is a software resource shared between F_1 and F_3) $\mathscr{F}_3 =$, $\mathscr{F}_4 =$, $\mathscr{F}_5 = \{F_4, F_6\}$, $\mathscr{F}_6 =$, $\mathscr{F}_7 =$, $\mathscr{F}_8 = \{F_7, F_9, F_{10}\}$, $\mathscr{F}_9 =$, and $\mathscr{F}_{10} =$. So $Nf = \{F_1, F_3, F_4, F_6, F_7, F_9, F_{10}\}$ is the set of normal functions and $Cf = \{F_2, F_5, F_8\}$ is the set of critical functions in *Sys*. Let us assume that the number of condition is 2. In condition 1 the system executes $\{F_1, F_3, F_4, F_7, F_9\}$, and in condition 2 *Sys* runs $\{F_3, F_4, F_6, F_9, F_{10}\}$ as shown in Figure 4.17. We apply the proposed approach to the considered case study.

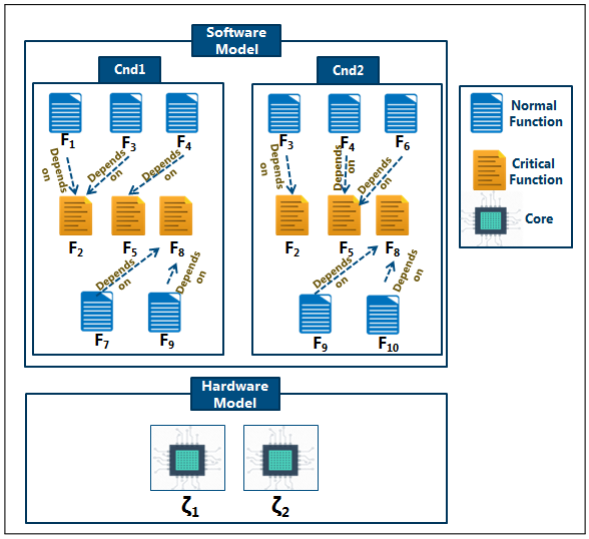

Fig. 4.17 Software and Hardware Models [68].

4.4.1 Normal Mode

4.4.1.1 Initial SW Architecture Generation

By applying the first step of MO^{2}R^{2}S which aims to generate an initial SW architecture from the software model. The software model is given in Table 4.5 [68] Each normal function of *Sys* is affected to a task, and each critical function

Table 4.5 Software Model [68].

Function	T_{F_k}	C_{F_k}	Type	$\mathscr{F}_k$
F_1	30	14	Soft	-
F_2	35	4	-	$\{F_1, F_3\}$
F_3	50	15	Hard	-
F_4	40	15	Hard	-
F_5	55	2	-	$\{F_4, F_6\}$
F_6	60	20	Soft	-
F_7	60	20	Hard	-
F_8	105	5	-	$\{F_7, F_9, F_{10}\}$
F_9	120	20	Hard	-
F_{10}	150	20	Soft	-

Table 4.6 Architecture Model [68].

Π_i	τ_i	T_{ij}	C_{ij}	φ_i	F_i
Π_1	τ_1	30	14	$\varphi_1 = F_2$	F_1
	τ_2	50	15	$\varphi_1 = F_2$	F_3
	τ_3	50	15	$\varphi_2 = F_5$	F_4
	τ_5	60	20	$\varphi_3 = F_8$	F_7
	τ_6	120	20	$\varphi_3 = F_8$	F_9
Π_2	τ_2	50	15	$\varphi_1 = F_2$	F_3
	τ_3	50	15	$\varphi_2 = F_5$	F_4
	τ_4	60	20	$\varphi_2 = F_5$	F_6
	τ_6	120	20	$\varphi_3 = F_8$	F_9
	τ_7	150	20	$\varphi_3 = F_8$	F_{10}

is executed by a resource. As we have two conditions so we obtain two implementations. The resulting architecture model is presented in Table 4.6.

4.4.1.2 Optimal Placement Computation

The placement computation step aims to partition and schedule tasks into the considered hardware model while minimizing the global blocking and the system

Table 4.7 Partitioning task model [68].

Π_i	ζ_s	τ_j
Π_1	ζ_1	τ_1
		τ_2
	ζ_2	τ_3
		τ_5
		τ_6
Π_2	ζ_1	τ_2
		τ_6
		τ_7
	ζ_2	τ_3
		τ_4

Table 4.8 Partitioning task model [68].

Π_i	ζ_s	τ_j
Π_1	ζ_1	τ_1
		τ_2
	ζ_2	τ_3
		τ_5
		τ_6
Π_2	ζ_1	τ_2
		τ_4
	ζ_2	τ_3
		τ_6
		τ_7

stability (i.e., moving time). The main input matrix of the MILP formulation is the "task to resource mapping" matrix X_{jl}. This matrix is defined as follow:

$$X_{jl} = \begin{pmatrix} & \varphi_1 & \varphi_2 & \varphi_3 \\ \tau_1 & 1 & 0 & 0 \\ \tau_2 & 1 & 0 & 0 \\ \tau_3 & 0 & 1 & 0 \\ \tau_4 & 0 & 1 & 0 \\ \tau_5 & 0 & 0 & 1 \\ \tau_6 & 0 & 0 & 1 \\ \tau_7 & 0 & 0 & 1 \end{pmatrix}$$

By applying the linear program for blocking time optimization, we obtain the following partitioning presented in Table 4.7. Let us compute task partitioning while optimizing *Sys* stability this time. The solver tries to find a feasible solution while optimizing the moving time of tasks from one core to another when *Sys* is switched from one implementation to another. The generated output Y_{ijs} is presented by the Table 4.10. We notice that the two linear programs give a

different placement result.

4.4.1.3 Local Optimization

By applying the local optimization proposed in our previous work to the obtained solution base which is composed of two solutions (i.e., Solution 1 with blocking time optimization, and Solution 2 with moving time optimization), we may obtain up to four solutions depending on the feasibility issue. For reasons of space availability, we do not report all the four solutions but only one solution. Let us apply the response time optimization to Solution 1 (presented in Section 4.4.1.2). For implementation Π_1 the solver merges task τ_5 and τ_6 in core ζ_2. For the second implementation (i.e., Π_2) tasks τ_2 and τ_7 are merged in core ζ_1. By comparing the optimized response time (231ms) to the one obtained before applying this step (277ms), it is clear that we have obtained better results [68].

4.4.1.4 Code Generation

From the obtained task model in the Section 4.4.1.3, we generate a POSIX code which is described in the Appendix B.

4.4.2 Resizing Mode

Let us take the same running example presented in the normal scenario by changing values of the system model parameters as presented in Table 4.9 . The solver couldn't find any feasible solution, so that we assume that the designer chooses to resize the hardware model. By executing Algorithm 4 the solver computes the number of cores that ensures the feasibility of the system. For this running example the minimum number of cores is three for the two implementations. The

Table 4.9 Architecture Model [68].

Π_i	τ_i	T_{ij}	C_{ij}	φ_i	F_i
Π_1	τ_1	30	25	$\varphi_1 = F_2$	F_1
	τ_2	50	40	$\varphi_1 = F_2$	F_3
	τ_3	50	40	$\varphi_2 = F_5$	F_4
	τ_5	60	55	$\varphi_3 = F_8$	F_7
	τ_6	120	105	$\varphi_3 = F_8$	F_9
Π_2	τ_2	50	40	$\varphi_1 = F_2$	F_3
	τ_3	50	40	$\varphi_2 = F_5$	F_4
	τ_4	60	50	$\varphi_2 = F_5$	F_6
	τ_6	120	105	$\varphi_3 = F_8$	F_9
	τ_7	150	140	$\varphi_3 = F_8$	F_{10}

Table 4.10 Partitioning task model [68].

Π_i	ζ_s	τ_j
Π_1	ζ_1	τ_1
		τ_2
	ζ_2	τ_3
		τ_5
	ζ_3	τ_6
Π_2	ζ_1	τ_2
		τ_3
	ζ_2	τ_4
		τ_6
	ζ_3	τ_7

possible partitioning given by the solver is described by Table 4.10. Then the tool executes again the normal mode steps with the new hardware model [68].

4.4.3 Degrade Mode

By considering the same running example presented in the previous subsection 4.4.2, and considering that the second step couldn't find any feasible solution but here the designer chooses to degrade *Sys* quality [68]. The solver is trying to find a feasible partitioning with a minimum value of the degradation factor Q. The obtained solution is [68]:
$\Pi_1 = \{\zeta_1 = \{\tau_1, \tau_2\},; \ \zeta_2 = \{\tau_3, \tau 5, \tau_6\}\}$ and $\Pi_2 = \{\zeta_1 = \{\tau_2, \tau_3\},; \ \zeta_2 = \{\tau_4, \tau_6, \tau_7\}\}$. Concerning the quality factor matrix is as follows:

$$Q_{ij} = \begin{pmatrix} & \Pi_1 & \Pi_2 \\ \tau_1 & 0.086 & 0 \\ \tau_2 & 0 & 0 \\ \tau_3 & 0 & 0 \\ \tau_4 & 0 & 0.454 \\ \tau_5 & 0.852 & 0 \\ \tau_6 & 0 & 0 \\ \tau_7 & 0 & 0.454 \end{pmatrix}$$

We note that in implementation Π_1, 8.6% of instances of task τ_1 and 85% of instances of task τ_5 will not meet their deadlines and in Π_2, 45.4% of instances of task τ_4 and task τ_7 will also miss their deadlines.

Conclusion

In this chapter, we extend the initial version of MO^{2}R^{2}S approach to provide a guidance framework for the generation of POSIX code for reconfigurable real-time system under multi-core architecture. The framework generate initially a software architecture from an input software model. Then, it ensure the assignment of tasks to cores while meeting timing properties and optimizing blocking and moving time to provide a solution base. In this step three scenarios are possible:

- Normal mode is executed when the framework finds at least one solution, then the obtained solution base would be an input for a second optimization step that minimizes the number of tasks in each given placement, response time, and energy consumption. Finally, the tool generates POSIX code.

- Resizing mode is executed whenever the tool could not find any solution and the user chooses to change the HW architecture, then the tool increases the number of cores until it finds the minimal number that ensures a feasible placement. Finally, it executes the normal mode.
- Degrading mode it is executed when the tool could not find any feasible placement and the designer chooses to downgrade the performance of the system by degrading the quality of soft tasks (i.e., by increasing the percentage of instances of soft task that do not meet their deadlines).

Note that this framework is extensible to add other metrics. In Future, we extend this approach to deal with scheduling/ partitioning sporadic and aperiodic tasks. It also will be tested with large size instances for more pertinence and reliability.

CHAPTER 5

Case Study *and* Evaluation of Performance

Introduction

This chapter presents a detailed description of the developed MO^{2}R^{2}S tool. Moreover, the proposed methodology is illustrated by two case studies in both architecture mono-core and multi-core to show the applicability of it. Simulations and different tests will also be presented at the end of this chapter in order to evaluate the performance of our proposed solutions.

5.1 MO^{2}R^{2}S Description

We exhibit in this subsection MO^{2}R^{2}S framework which is implemented in JAVA. Its main goal is to implement our proposed methodology described above which aims to a feasible synthesis of reconfigurable real-time systems under multi-core architecture from the specification model to POSIX code. It represents a passage from the theoretical studies to the real implementation of the proposed solutions. The developed tool is platform-independent and is an open source. As depicted in Figure 5.1, the designer can:

- Generate task sets from the SW model,
- Generate resource sets from the SW model,
- Generate implementation stes from the condition sets
- Perform partitioning tasks to cores
- Optimize the partitioning
- Generate controller code from the obtained HW model and the obtained optimized SW architecture.

Figure 5.2 presents tool class diagram. The classes are detailed in the following.

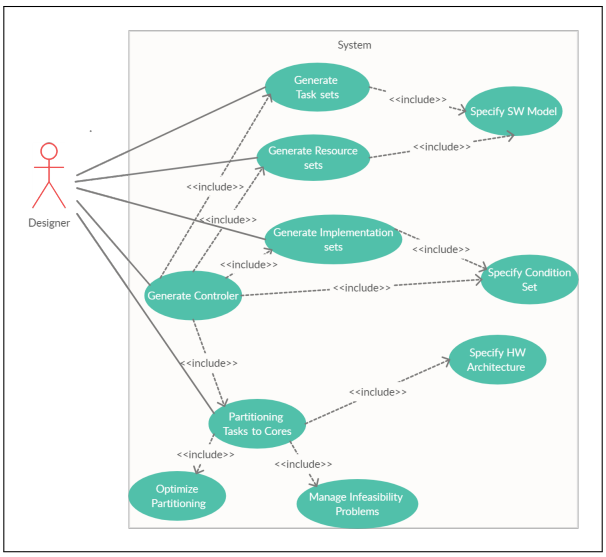

Fig. 5.1 Use Case of MO^{2}R^{2}S.

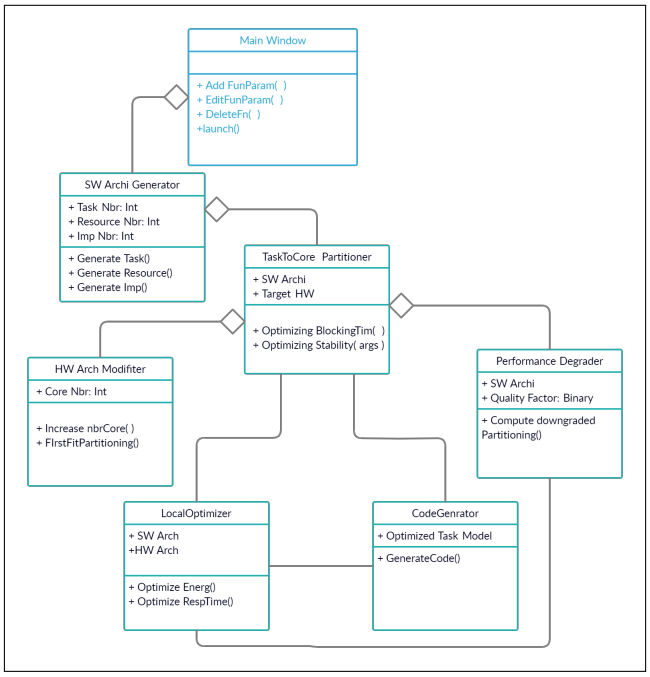

Fig. 5.2 MO^{2}R^{2}S Class diagram.

- Main Window: is the main class. It is the first view of the tool. In which the user specify the SW model. It can call SW Archi generator class using the methods: launch,
- SW Archi Generator: it is responsible of generating the SW architecture.
- TaskToCore Partitioner: this class performs the optimized partitioning step by calling the solver to find a feasible solution,
- HW Arch Modifiter: This class is executed whenever the solver could not find any feasible solution and the user chooses to resize the HW architecture,
- Performance Degrader: It is called when the solver could not find any feasible solution and the user chooses to downgrade the performance of the system,
- LocalOptimizer: It is responsible to the second optimization step,
- CodeGenrator: It performs the code generation step.

The following Figures demonstrate the different parts of the user interface: The

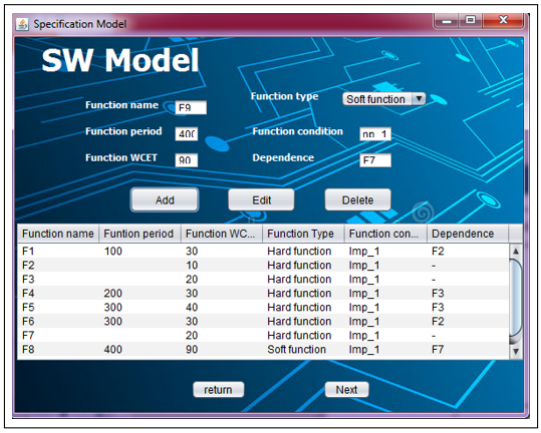

Fig. 5.3 MO^{2}R^{2}S SW Model Interface.

SW model parameters would be specified by the designer through the interface as presented in Figure 5.3. Then an initial SW architecture is generated as show in Figure 5.4. Through the interface presented in Figure 5.5, the design defines the HW model by specifying the number of cores. Then the solver is called to find at least one optimal placement through the interface in Figure 5.6. The user selects one of the obtained solution base via the interface shown in Figure 5.7. An example of selected solution is exposed in Figure 5.8. Then, the solver is called again for the second optimization step(See Figure 5.9). Finally, the code is generated through the interface shown in Figure 5.10. For more details concerning the framework please visit our website *https://project-lisi-lab.wixsite.com/mo2r2s*), we exposed all the scenarios.

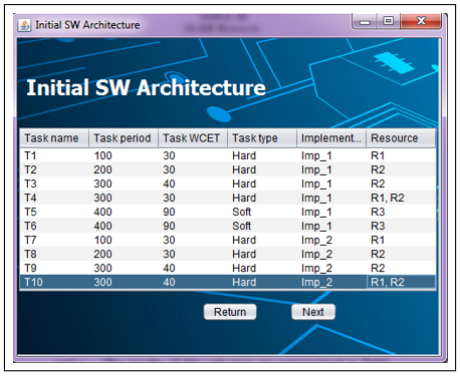

Fig. 5.4 MO^{2}R^{2}S SW Architecture Interface.

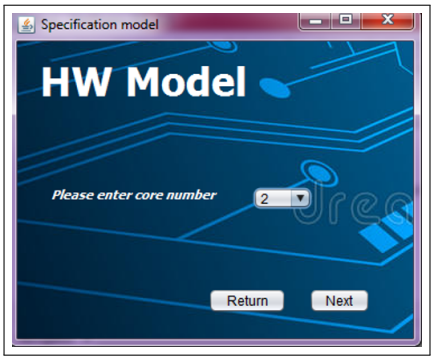

Fig. 5.5 HW Model Specification Interface.

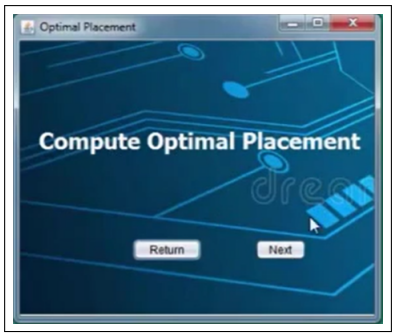

Fig. 5.6 Tool Computing optimal Placement Interface.

Fig. 5.7 Tool Selection Solution Interface.

5.2 Application

The proposed approach allows designers to design and implement multi-core as well as mono-core reconfigurable real-time systems through its graphical user

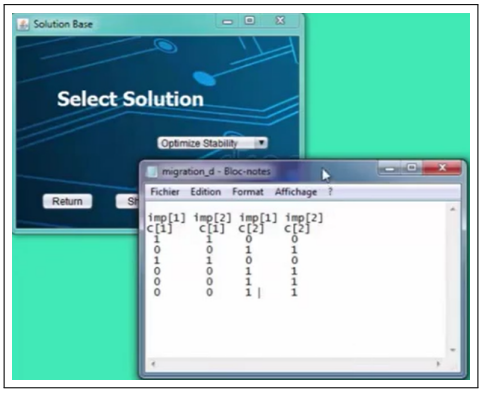

Fig. 5.8 MO^{2}R^{2}S Placement Result Interface.

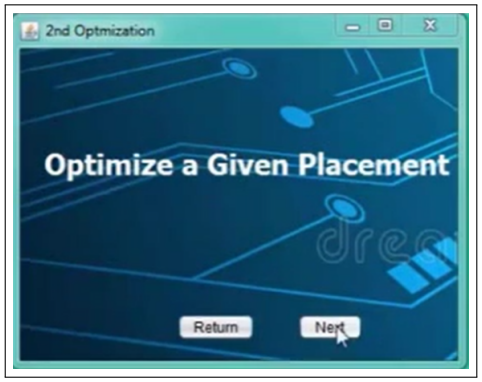

Fig. 5.9 Local Optimization Interface.

Fig. 5.10 MO^{2}R^{2}S Code Generation Interface.

interfaces. We illustrate the feasibility of our methodology as well as the gains offered from it by two case studies: “Car Collision Avoidance System” for mono-core architecture and “Autonomous Vehicles” for multi-core and architecture.

5.2.1 Car Collision Avoidance System Mono-core Case Study

The considered case study is Car Collision Avoidance System denoted (CCAS) [51]. It has as role to detect obstacles in front of the car to which it is mounted. In

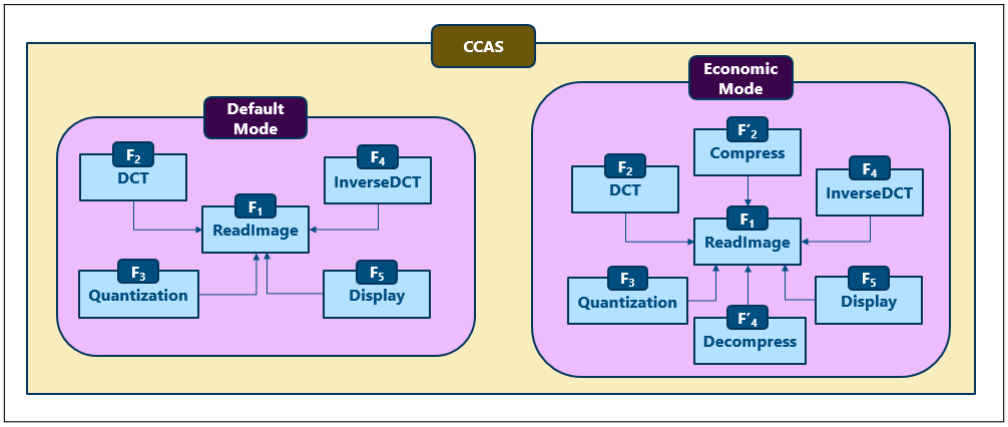

Fig. 5.11 CCAS Specification.

order to show the applicability of our framework, we consider a simplified version of CCAS by omitting several features of it. We consider just two operational modes:

- Default mode: represents a traditional use of CCAS,
- Economic mode: represents a restrictive use of CCAS with safety requirements.

Figure 5.11 depicts the specification model of CCAS (i.e., each mode with its functions) As shown in Figure 5.11 the default mode is composed of five functions: four normal functions and one critical function:

- F_1 (ReadImage): reads images from the input to the system from a radar,
- F_2 (Discrete Cosine Transformation : DCT): moves the representation of the image from the spatial domain into the frequency domain,
- F_3 (Quantization): selectively discards data in the frequency domain to compress the image,
- F_4 (InverseDCT): moves the image back into the spatial domain,
- F_5 (Display): displays the images for monitoring.

The economic mode is composed of six normal functions and one critical function. As shown in Figure 5.11, we added in this mode just two normal functions F_2' to compress the image and F_4' to decompress it. Table 5.1 depicts in details the specification model (i.e., modes, function sets with their parameters).

5.2.1.1 CCAS Initial Task Model

By applying the first step of MO^{2}R^{2}S which aims to generate an initial task model from the specification model (Table 5.1). Table 5.2 gives a tabular description of

Table 5.1 CCAS Specification.

Function	Cond	T_{F_k}	Cn_{F_k}	$Type$	$\mathscr{F}_k$
F_1	Default	20	1	Critical	-
F_2		10	2	Normal	F_1
F_3		15	1	Normal	F_1
F_4		20	1	Normal	F_1
F_5		20	2	Normal	F_1
F_1	Economic	20	1	Critical	-
F_2		10	2	Noraml	F_1
F_2'		15	2	Normal	F_1
F_3		15	1	Normal	F_1
F_4		20	1	Normal	F_1
F_4'		30	2	Normal	F_1
F_5		20	2	Normal	F_1

Table 5.2 Tabular description of the initial task model of the CCAS.

τ_j/φ_q	Π_i	T_{F_k}	Cn_{F_k}	Function
φ_1	Π_1	20	1	F_1
τ_1		10	2	F_2
τ_2		15	1	F_3
τ_3		20	1	F_4
τ_4		20	2	F_5
φ_2	Π_2	20	1	F_1
τ_5		10	2	F_2
τ_6		15	2	F_2'
τ_7		15	1	F_3
τ_8		20	1	F_4
τ_9		30	2	F_4'
τ_{10}		20	2	F_5

the initial task model describing the CCAS. As shown in Table 5.2, each normal function is affected to a task and each critical function is executed by a resource. Thus we obtain ten tasks and two shared resources. This model shows two possible implementations of the CCAS which refer respectively to the two execution modes already specified.

5.2.1.2 CCAS Optimized Models

As we consider a mono-core architecture so we dont need to do the partitioning, we apply directly the local optimization step which consists in generating two optimized task models by executing MILP formulations defined previously. Both proposed linear programs produce the same merge matrix which is given as follow:

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}
τ_1	1	0	1	1	1	0	0	1	0	1
τ_2	0	1	0	0	0	1	1	0	0	0
τ_3	0	0	0	0	0	0	0	0	0	0
τ_4	0	0	0	0	0	0	0	0	0	0
τ_5	0	0	0	0	0	0	0	0	0	0
τ_6	0	0	0	0	0	0	0	0	0	0
τ_7	0	0	0	0	0	0	0	0	0	0
τ_8	0	0	0	0	0	0	0	0	0	0
τ_9	0	0	0	0	0	0	0	0	1	0
τ_{10}	0	0	0	0	0	0	0	0	0	0

The MILP formulations allow to merge i) τ_1, τ_3, τ_4, τ_5, τ_8, and τ_{10}, and ii) τ_2, τ_6, and τ_7. We note that tasks τ_9 is not merged with τ_2, τ_6, and τ_7 even their period are harmonic because due to feasibility concerns (i.e., if the solver decide to merge them, the resulting task will not meet its deadline).

The result of the first MILP formulation which aims to reduce the total response time is shown in Table 5.4. For the second objective function which aims to

Table 5.4 CCAS Optimized Task Model in term of Total response time.

τ_j/φ_q	Π_i	T_{ij}	C_{ij}	B_{ij}	Total $\mathscr{R}_{Old}$	Total $\mathscr{R}_{New}$
φ_1	Π_1	20	1	-	-	-
τ_1		10	5	1	15	12
τ_2		15	1	0		
φ_2	Π_2	20	1	-	-	-
τ_1		10	5	1	33	25
τ_2		15	3	1		
τ_9		30	2	0		

minimize energy consumption, the linear program generates the following task model depicted in Table 5.5. We can see from Table 5.5 that this model allows to reduce the energy consumption of CCAS.

Table 5.5 CCAS Optimized Task Model in term of Energy Consumption.

τ_j/φ_q	Π_i	T_{ij}	Cn_{ij}	η_{ij}	$C_{new_{ij}}$	E_{old}	E_{new}
φ_1	Π_1	20	1	-	-	-	-
τ_1'		10	5	0.8	4	562.5	450
τ_2'		15	1	0.8	0.8	112.5	90
φ_2	Π_2	20	1	-	-	-	-
τ_1'		10	5	0.8	4	562.5	450
τ_2'		15	3	0.8	2.4	337.5	270
τ_9		30	2	1	2	225	225

5.2.1.3 CCAS POSIX Code

In order to implement the obtained task model in POSIX code, we apply the algorithm of POSIX-code generation presented in the previous chapter. The skeleton of the code implementation of CCAS is represented by the following listing 1.

Listing 5.1: CCAS POSIX code.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
sem_t evt; //Declaration of the semaphore representing the synchronization event
void* F1 (void* arg); // Critical function
void* F2 (void* arg); // Normal function
void* F3 (void* arg); // Normal function
void* F4 (void* arg); // Normal function
void* F5 (void* arg); // Normal function
void* F_prime_2 (void* arg); // Normal function
void* F_prime_4 (void* arg); // Normal function

int main (void){
pthread_t tau_1;
pthread_t tau_2;
pthread_t tau_9;
// Default Mode
pthread_create (&tau_1, NULL, F2, (void *) 2);
pthread_create (&tau_1, NULL, F4, (void *) 1);
pthread_create (&tau_1, NULL, F5, (void *) 2);
pthread_join (tau_1, NULL);
pthread_create (&tau_2, NULL, F3, (void *) 1);
pthread_join (tau_2, NULL);
// Secure Mode
if (cnd="Economic"){
pthread_create (&tau_1, NULL, F2, (void *) 2);
pthread_create (&tau_1, NULL, F4, (void *) 1);
pthread_create (&tau_1, NULL, F5, (void *) 2);
pthread_join (tau_1, NULL);
pthread_create (&tau_2, NULL, F_prime_2, (void *) 2);
pthread_create (&tau_2, NULL, F3, (void *) 1);
pthread_join (tau_2, NULL);
pthread_create (&tau_9, NULL, F_prime_4, (void *) 2);
pthread_join (tau_9, NULL);
}return 0; }

void* F2 (void* arg){
F1(void);       // Critical setion
 sem_post(&evt);
pthread_exit(NULL); /*end of thread */}
```

The presented skeleton in listing 1 helps the developer to implement the full code and details the implementation of functions.

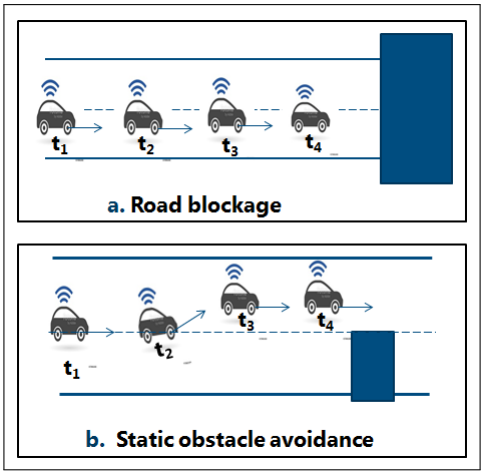

Fig. 5.12 Autonomous vehicles scenarios [68].

5.2.2 Autonomous Vehicles System Multi-core Case Study

We provide a case study of Autonomous Vehicles (AV) [62] based in multi-core architecture with reconfiguration requirements. AV system is composed of several scenarios. In order to show the applicability of the proposed framework, we consider a simplified version of this case study by omitting many scenarios and functionalities. We investigate just two scenarios [68]: (See Figure 5.12)

- Road blockage: in which a car can reach a full stop just in time to avoid collision with a blocking obstacle,
- Static obstacle avoidance: in which a car slightly nudges to the left, and decreases the speed slightly to avoid collision.

Each scenario is defined by nine functions depicted in Table 5.6:

We did not mention the function parameters in details in Table 5.6. In order to illustrate the implementation of AV system in the different modes(i.e., normal, resizing, and degrading mode), we have to consider different values of the SW model.

5.2.2.1 AV in Normal Mode

The considered SW model is depicted in Table 5.11. We assume that the AV system is mapped to a preemptive execution platform composed of one processor containing two cores ζ_1 and ζ_2.

Initial SW Architecture

The SW model presented in Table 5.11 is the input of SW architecture generation step. For each scenario, the framework generates an implementation, and we assign each normal function to a task and each critical function to a resource. Thus,

Autonomous Vehicle SW Model

Condition	Function Name	Nature	$\mathscr{F}_k$
Road Blockage	F_1: BehaviorFn	Hard	F_2
	F_2: MissionPlannerFn	-	-
	F_3: PositionSensorFn	-	-
	F_4: RoadMotionPlanFn	Hard	F_3
	F_5: PrePlannerFunction	Hard	F_3
	F_6: RoadBlockDetect	Hard	F_2
	F_7: MassiveFlowSensor	-	-
	F_8: BaseFuelMass	Soft	F_7
	F_9: ControllerFuel	Soft	F_7
Static Obstacle Avoidance	F_1:BehaviorFn	Hard	F_2
	F_2: MissionPlannerFn	-	-
	F_3: PositionSensorFn	-	-
	F_4: RoadMotionPlanFn	Hard	F_3
	F_5: PrePlannerFunction	Hard	F_3
	F_{10}: ObstacleDetector	Hard	F_7
	F_7: MassiveFlowSensor	-	-
	F_8: BaseFuelMass	Soft	F_7
	F_9: ControllerFuel	Soft	F_7

Normal Mode: SW Model.

Condition	Function Name	C_{F_k}	T_{F_k}	Nature	$\mathscr{F}_k$
Road Blockage	F_1	30	100	Hard	F_2
	F_2	10	100	-	-
	F_3	20	100	-	-
	F_4	30	200	Hard	F_3
	F_5	40	300	Hard	F_3
	F_6	30	300	Hard	F_2
	F_7	20	300	-	-
	F_8	100	400	Soft	F_7
	F_9	90	400	Soft	F_7
Static Obstacle Avoidance	F_1	30	100	Hard	F_2
	F_2	10	100	-	-
	F_3	20	100	-	-
	F_4	30	200	Hard	F_3
	F_5	40	300	Hard	F_3
	F_{10}	40	300	Hard	F_7
	F_7	20	300	-	-
	F_8	100	400	Soft	F_7
	F_9	90	400	Soft	F_7

the obtained SW architecture consists of two implementation Π_1 and Pi_2 which are composed of six tasks and three shared resources (See Table 5.8). Each task

Table 5.8 Normal mode: SW Architecture.

Pi_i	τ_j	C_{ij}	T_{ij}	Type	$C_{\varphi_{q_j}}$
Pi_1	τ_1	30	100	Hard	10
	τ_2	30	200	Hard	20
	τ_3	50	300	Hard	20
	τ_4	50	300	Hard	10
	τ_5	100	400	Soft	20
	τ_6	90	400	Soft	20
Pi_2	τ_7	30	100	Hard	10
	τ_8	30	200	Hard	20
	τ_9	50	300	Hard	20
	τ_{10}	50	300	Hard	20
	τ_{11}	100	400	Soft	20
	τ_{12}	90	400	Soft	20

inherits the real time parameters of the implemented function. It may also use a set of software resources.

AV Placement Computation Step

This step consist in partitioning/ scheduling tasks into multi-core architecture. It takes as inputs the SW architecture and the HW model which is defined by two core. The partitioning step aims to ensure the mapping of tasks into cores while minimizing either blocking time or moving time. The output of this step is depicted in Table 5.9. We note that for this case study the framework provides a solution base containing two solutions. Table 5.9 shows that the two obtained

Table 5.9 AV Solution Base: Optimal Placement [68].

Solution 1: Blocking Time Optimization							Solution 2: Moving Time Optimization						
Π_i	ζ_s	τ_j	B_{ij} (ms)	$\Re_i$ (ms)	$\mathscr{R}_{ij}$ (ms)	E_{ij} (Joules)	Π_i	ζ_s	τ_j	B_{ij} (ms)	$\Re_i$ (ms)	$\mathscr{R}_{ij}$ (ms)	E_{ij} (Joules)
Π_1	ζ_1	τ_1	9	3	39	9.47	Π_1	ζ_1	τ_1	9	2	39	9.47
		τ_2	19		69	11.69			τ_3	38		69	11.69
		τ_3	19		81	6.06		ζ_2	τ_2	19		81	6.06
		τ_4	0		65	6.5			τ_4	18		65	6.5
	ζ_2	τ_5	19		118	6.4			τ_5	19		118	6.4
		τ_6	0		99	6.35			τ_6	0		99	6.35
Π_2	ζ_1	τ_7	9	3	39	9.47	Π_2	ζ_1	τ_7	9	2	39	9.47
		τ_8	19		69	11.69			τ_9	38		69	11.69
		τ_{10}	0		84	6.06		ζ_2	τ_8	19		84	6.06
	ζ_2	τ_9	19		81	6.5			τ_{10}	19		81	6.5
		τ_{11}	19		118	6.4			τ_{11}	19		118	6.4
		τ_{12}	0		99	6.35			τ_{12}	0		99	6.35

solutions (i.e., *Solution 1* and *Solution 2*) give different placement depending on

optimizing metric (i.e., blocking time or moving time). It is clear that the total blocking time in *Solution 1* is less than *Solution 2* but concerning the reconfiguration time in *Solution 1* is greater than *Solution 2*. That proves the efficiency of the proposed framework and gives to designers the flexibility to explore multiple solutions while ensuring the feasibility of the system.

AV Local Optimization

The obtained solution from the previous step is considered as an input for this step (i.e., second level of optimization). For each obtained placement, we apply the optimization step proposed in the initial version of MO^{2}R^{2}S in each core which addresses the mono-core architecture (See Chapter 3) in order to get a second solution base. We obtained four solutions, we do not present all of them but only the result of applying this methodology to *Solution 1* to obtain two solutions where each solution aims to optimize either the response time or the energy consumption. Table 5.10 provides the resulting solution base. It is clear from Tables

Table 5.10 AV Solution Base: Optimal local Placement.

Solution 1.1 Opt Response Time				Solution 1.2 Opt Energy			
Π_i	ζ_s	τ_j	$\mathscr{R}_{new}$	Π_i	ζ_s	τ_j	E_{new}
Π_1	ζ_1	τ_1	90	Π_1	ζ_1	τ_1	19.76
		τ_3	125			τ_2	11.4
	ζ_2	τ_5	180		ζ_2	τ_5	12.03
Π_2	ζ_1	τ_1	90	Π_2	ζ_1	τ_1'	19.76
		τ_{10}	65			τ_{10}	5.01
	ζ_2	τ_3	62		ζ_2	τ_3	5.3
		τ_5	190			τ_5	12.03

5.10 and 5.9 that i) the energy consumption is optimized by an average of 8.7%, and the response time by 10.9%, and ii) the tasks count is reduced from 12 to 4. The obtained merge matrix is given as follow:

The MILP formulations allow to merge i) τ_1, τ_2, τ_7, and τ_8, ii) τ_3, τ_4, and τ_9, and

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}
τ_1	1	1	0	0	0	0	1	1	0	0	0	0
τ_2	0	0	0	0	0	0	0	0	0	0	0	0
τ_3	0	0	1	1	0	0	0	0	1	0	0	0
τ_4	0	0	0	0	0	0	0	0	0	0	0	0
τ_5	0	0	0	0	1	1	0	0	0	0	1	1
τ_6	0	0	0	0	0	0	0	0	0	0	0	0
τ_7	0	0	0	0	0	0	0	0	0	0	0	0
τ_8	0	0	0	0	0	0	0	0	0	0	0	0
τ_9	0	0	0	0	0	0	0	0	0	0	0	0
τ_{10}	0	0	0	0	0	0	0	0	0	1	0	0
τ_{11}	0	0	0	0	0	0	0	0	0	0	0	0
τ_{12}	0	0	0	0	1	1	0	0	0	0	0	0

iii) τ_5, τ_6, τ_{11}, and τ_{12}.

AV POSIX Code

Depending on requirements, users have to select one model from the solution base to generate the POSIX code. We suppose that the designer chooses to implement

the solution 1.2 (i.e., Opt Energy). The skeleton of the code implementation of AV is represented by the following listing 1:

Listing 5.2: POSIX code for AV.

```
#include <pthread.h>
void* F1 (void* arg); // Normal function
void* F2 (void* arg); // Critical function
void* F3 (void* arg); // Critical function
void* F4 (void* arg); // Normal function
void* F5 (void* arg); // Normal function
void* F6 (void* arg); // Normal function
void* F7 (void* arg); // Critical function
void* F8 (void* arg); // Normal function
void* F9 (void* arg); // Normal function
void* F10 (void* arg); // Normal function

/********* Controller POSIX code *********/
int main (void){
pthread_t tau_1;
pthread_t tau_3;
pthread_t tau_5;
pthread_t tau_10;

// Implementation 1
pthread_create (&tau_1, NULL, F1, (void *) 100);
pthread_create (&tau_1, NULL, F4, (void *) 200);

pthread_create (&tau_3, NULL, F5, (void *) 300);
pthread_create (&tau_3, NULL, F6, (void *) 300);

 pthread_create (&tau_5, NULL, F8, (void *) 400);
pthread_create (&tau_5, NULL, F9, (void *) 400);

pthread_join (tau_1, NULL);
stick_this_thread_to_core(1);
pthread_join (tau_3, NULL);
stick_this_thread_to_core(1);
pthread_join (tau_5, NULL);
stick_this_thread_to_core(1);

else if (cnd="cnd2"){ // Implementation 2
pthread_create (&tau_1, NULL, F1, (void *) 100);
pthread_create (&tau_1, NULL, F4, (void *) 200);

pthread_create (&tau_3, NULL, F5, (void *) 300);

pthread_create (&tau_10, NULL, F10, (void *) 300);

pthread_create (&tau_5, NULL, F8, (void *) 400);
pthread_create (&tau_5, NULL, F9, (void *) 400);

pthread_join (tau_1, NULL);
stick_this_thread_to_core(2);
pthread_join (tau_3, NULL);
stick_this_thread_to_core(2);
pthread_join (tau_10, NULL);
stick_this_thread_to_core(2);
pthread_join (tau_5, NULL);
stick_this_thread_to_core(2);}
}
return 0;}

void* F1 (void* arg){//}

int stick_this_thread_to_core(int core_id) {
int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
if (core_id < 0 || core_id >= num_cores) return EINVAL; cpu_set_t cpuset;
CPU_ZERO(&cpuset); CPU_SET(core_id, &cpuset); pthread_t current_thread = pthread_self();
return pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset);}
```

5.2.2.2 AV in Resizing Mode

In order to execute this mode,the SW model values need to be adjusted in a way that the placement computations step could not find any feasible solution (See Table 5.12).

AV SW Architecture Generation

Table 5.12 Resizing Mode: SW Model.

Condition	Function Name	C_{F_k}	T_{F_k}	Nature	$mathcal F_k$
Road Blockage	F_1	14	30	Soft	F_2
	F_2	4	35	-	-
	F_3	15	50	Hard	F_2
	F_4	15	40	Hard	F_5
	F_5	2	55	-	-
	F_7	20	60	Soft	F_8
	F_8	5	105	-	-
	F_9	20	120	Hard	F_8
Static Obstacle Avoidance	F_2	4	35	-	-
	F_3	15	50	Hard	F_2
	F_4	15	40	Hard	F_5
	F_5	2	55	-	-
	F_6	20	60	Soft	F_5
	F_8	5	105	-	-
	F_9	20	120	Hard	F_8
	F_{10}	20	150	Soft	F_8

The SW architecture generation step produce the following model presented in Table 5.13

AV Partitioning Step

Table 5.13 Resizing mode: SW Architecture.

Pi_i	τ_j	C_{j}	T_{ij}	$C_{\varphi_{q_j}}$
Pi_1	τ_1	25	30	4
	τ_2	40	50	4
	τ_3	40	50	2
	τ_5	55	60	5
	τ_7	105	120	5
Pi_2	τ_2	40	50	4
	τ_3	40	50	2
	τ_4	50	60	2
	τ_7	105	120	5
	τ_8	140	150	5

The placement computation step could not find any feasible solution, so we assume that the user chooses to resize the HW architecture. By executing Algorithm

4 the solver tries to find the minimum number of core that ensures the AV system feasibility. In this case, the minimum number of cores is three. This partitioning is giving by Table 5.14. Note that the used partitioning algorithm is the firs fit, so

Table 5.14 Resizing Mode: Partitioning task model.

Π_i	ζ_s	τ_j
Π_1	ζ_1	τ_1
		τ_2
	ζ_2	τ_3
		τ_5
	ζ_3	τ_6
Π_2	ζ_1	τ_2
		τ_3
	ζ_2	τ_4
		τ_6
	ζ_3	τ_7

we only consider the AV system feasibility in the partitioning not the optimization. After this step, the framework executes again the normal mode steps with new HW model.

5.2.2.3 AV in Degrading Mode

In this mode we consider the same SW model presented in Table 5.12. Thus the SW architecture generation step produces the same SW architecture generated in resizing mode which is depicted in Table 5.13. The computation of optimal placement could not find any feasible solution, so we assume that the designer chooses to degrade the AV system performance. The framework tried to find a feasible partitioning with a minimum value of the degradation factor Q. Table 5.15 depicted the obtained solution. We note that in implementation Π_1, 8.6% of

Table 5.15 Degrading Mode: Partitioning task model.

Π_i	ζ_s	τ_j	Q_{ij}
Π_1	ζ_1	τ_1	0.086
		τ_2	0
	ζ_2	τ_3	0
		τ_5	0.852
		τ_6	0
Π_2	ζ_1	τ_2	0
		τ_3	0
	ζ_2	τ_4	0.454
		τ_6	0
		τ_7	0.454

instances of task τ_1 and 85% of instances of task τ_5 will not meet their deadlines and in Π_2, 45.4% of instances of task τ_4 and task τ_7 will also miss their deadlines. After this step, the framework executes the third and fourth steps of normal mode.

5.3 Evaluation of Performance

In order to expose the contribution of this work, we evaluate the performance of MO^{2}R^{2}S framework by generating a random multi-core reconfigurable real-time systems with a random function set, core set, and condition set.

5.3.1 Evaluation Of MO^{2}R^{2}S on mono-core architecture

We compare the proposed approach with mono-core based approach. The number of task has been varied between 5 and 100. The evaluation of the reconfiguration

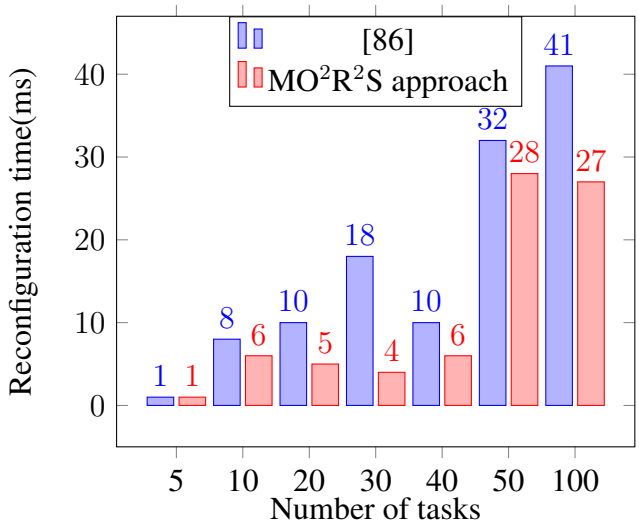

Fig. 5.13 Comparison in terms of reconfiguration time between the proposed approach and the approach proposed in [86].

time of the proposed approach compared with work reported in [86] is shown in Figure 5.13. In fact, it is remarkable through this comparison that we obtain better results in term of reconfiguration time. The gain is more important when increasing the number of tasks, it is about 14 ms for 100 tasks. This is due to the tasks merging technique. Figure 5.14 depicts the comparison of context switching of the proposed approach with the related works reported in [86] and [21]. It is clear that the work reported in [21] and MO^{2}R^{2}S approach give always a better context switching than that reported in [86]. This is due to the minimization of task count used in [21] and the MO^{2}R^{2}S approach. Also the comparison shows that our proposed approach is more efficient than the work reported in [21] since our the latter merges only tasks having the same periods unlike our approach

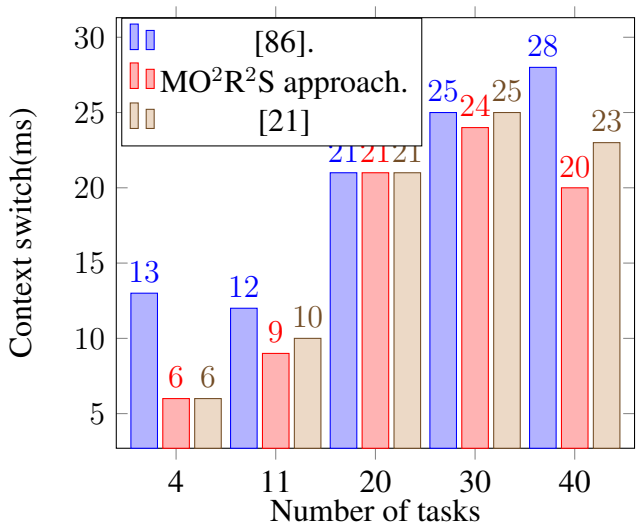

Fig. 5.14 Comparison in terms of context switching between the proposed approach and the approaches proposed in [86] and [21].

which merges also harmonic tasks. In a random generated system with a number of tasks that varies between 5 and 40, the gain in MO²R²S approach ranges from 0 ms to 8 ms depending on the reconfiguration scenario. In Figure 5.15, we compare the execution time of the POSIX code generated following the proposed approach with the work reported in [52] that deals with reconfigurable systems. This computation is based on the fact of implementing the task model obtained in [52] in POSIX code and computing its execution time. The evaluation shows that

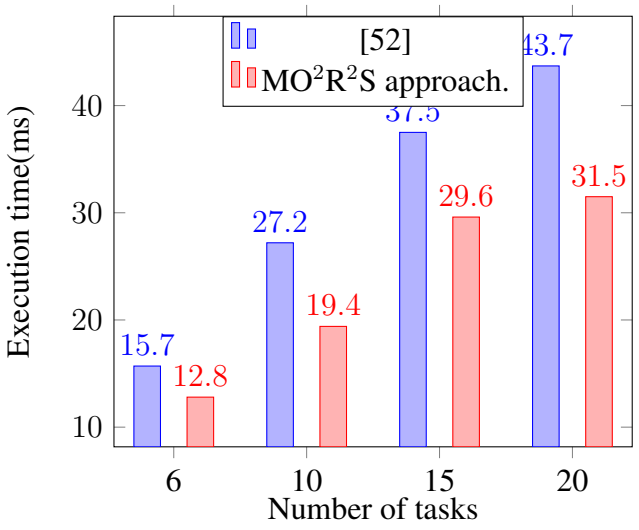

Fig. 5.15 Comparison in terms of code execution time between the proposed approach and the approach proposed in [52].

MO²R²S approach has always a better execution time than that reported in the related work. The gain is about 12 ms which is due to the merging technique that allowing the reduction of the tasks number thus the number of thread is reduced.

Figure 5.16 shows the variation of response time with and without merging tech-

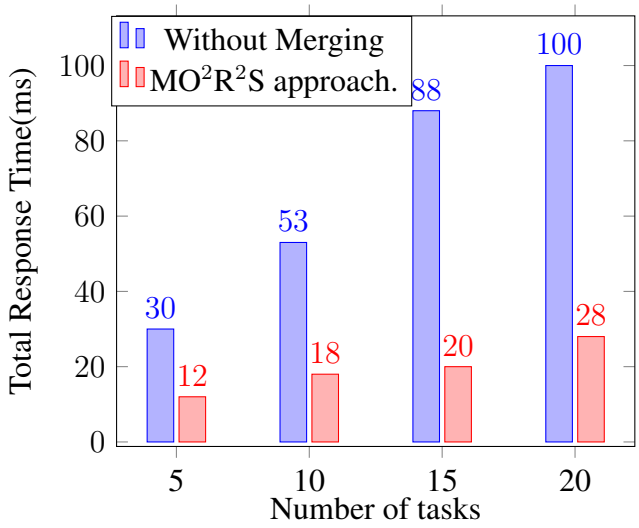

Fig. 5.16 Rate of Response Time With and Without Merging Technique.

nique. It is clear that this technique offer a gain goes to 72ms by increasing the number of tasks. Figure 5.17 shows a comparison of energy consumption of

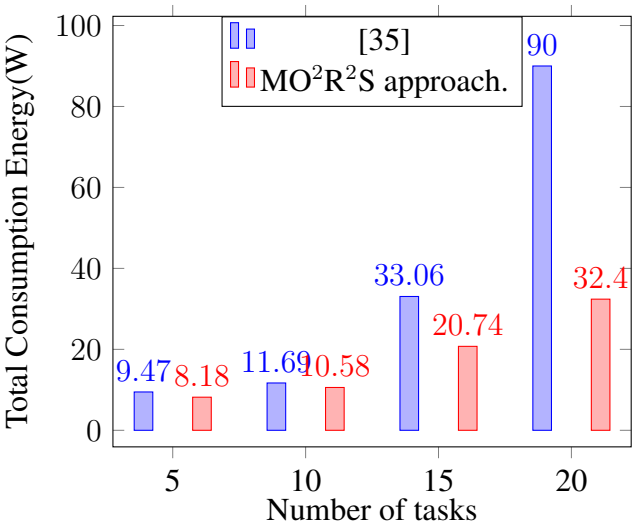

Fig. 5.17 Comparison in terms of energy consumption between our approach and the approach proposed in [35].

MO^2R^2S approach with the obtained one in the work reported in [35]. Despite, we use the same technique of optimization, we have obtained better results. This is due to the additional used technique i.e., merging technique. This approach permits to gain up in energy to 57.6W.

5.3.2 Evaluation Of MO^2R^2S on multi-core architecture

In the following evaluations, the number of function has been varied between 4 and 20 under duo-core architecture. In this approach, we aim to improve the sys-

tem stability so that the reconfiguration time is also would be optimized. Figure 5.18 shows an evaluation of the reconfiguration time in the proposed approach compared with the work [49]. It was clear that this work allows obtaining a lower reconfiguration time. This is due to minimization of the moving time. The main

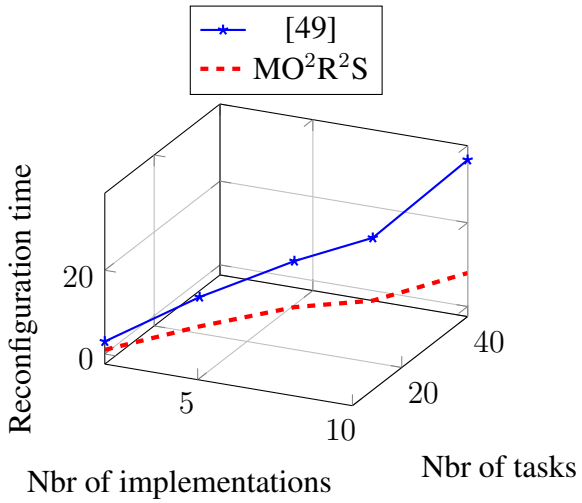

Fig. 5.18 Evaluation of reconfiguration time.

objective in this approach is ensuring a feasible partitioning while optimizing the blocking time. Figure 5.19 depicts the comparison of the obtained blocking time in this approach with the related work reported in [56]. We note that the performance of the proposed approach has always a smaller blocking time than that reported in the related work this is due to minimization technique of global resource count among cores. The optimization of the blocking time leads to the op-

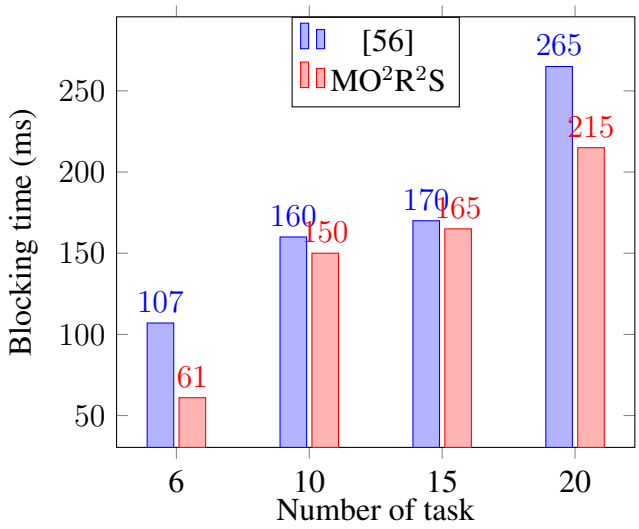

Fig. 5.19 Comparison in terms of blocking time between the proposed approach and that in [56].

timization of both process utilization (Figure 5.20) and the latency (Figure 5.21).

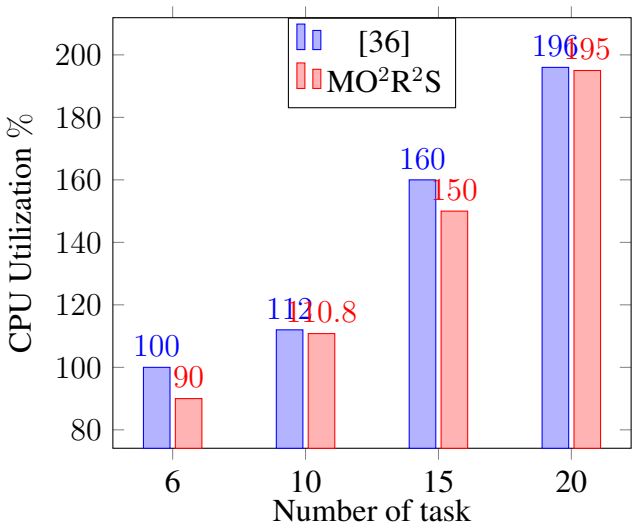

Fig. 5.20 Comparison in terms of processor utilization factor between the proposed approach and that in [36].

Figure 5.20 depicts the impact of our approach on processor utilization. The comparison shows the efficiency of the proposed approach comparing with the work reported in [36]. We evaluate latency in Fig. 5.21. By comparing this metric in

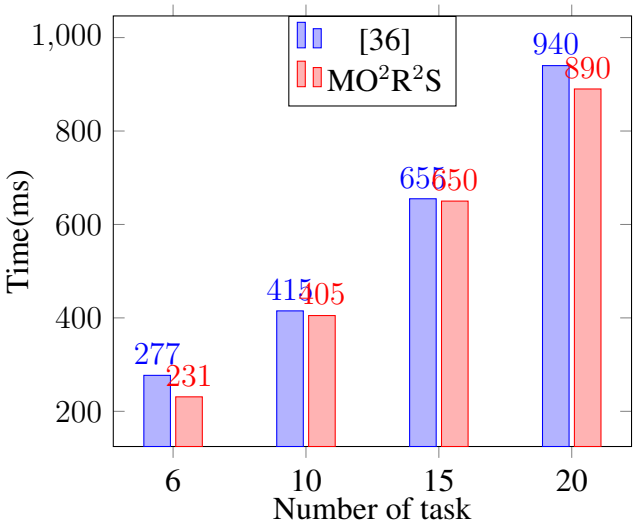

Fig. 5.21 Comparison in terms of latency between the proposed approach and that in [36].

this work with that in [36], the efficiency of the proposed approach is confirmed. We evaluate in the flowing figure 5.22 both context switching (Cs) and preemption in this work compared with the approach in [114] by generating a set of 4 to 30 tasks on 2 to 6 cores. It is clear that our approach is more efficient in term of context switching and preemption comparing with that obtained in the related work [114]. We evaluate in the flowing graphs both the number of line (Figure 5.23) and the code execution time (Figure 5.24). Figure 5.23 shows the evolution

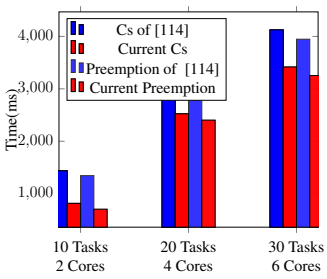

Fig. 5.22 Comparison in terms of context switching (Cs) and preemption between the proposed approach and that reported in [114].

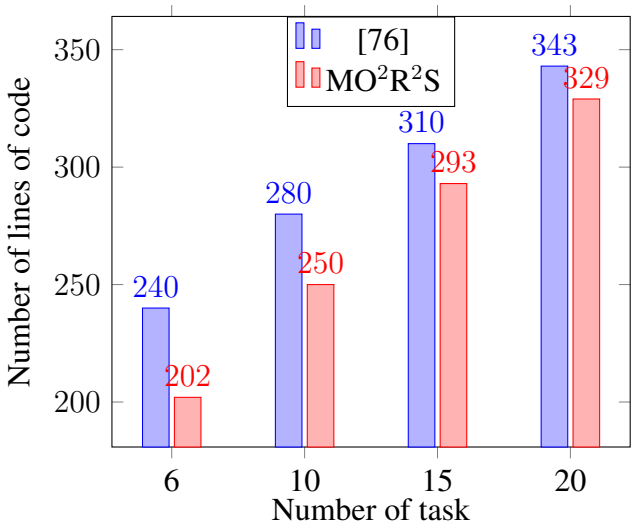

Fig. 5.23 Comparison in terms of number of lines of code between the proposed approach and the work reported in [76].

of code size, i.e., the number of lines in code, of the proposed approach and that of [76]. From it we can see that the proposed approach increases slowly in term of code size with the problem size compared with the work in [76]. We evaluate

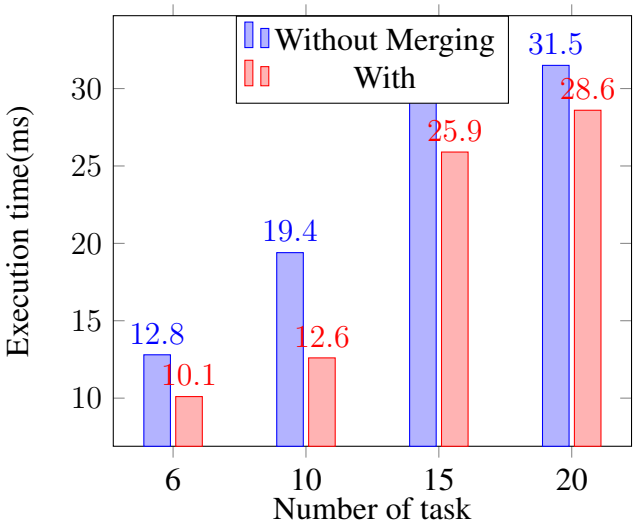

Fig. 5.24 Code execution Time before and after the application of the merge technique.

in Figure 5.24 the code execution time of the proposed approach by comparing it with the obtained one without using merging technique and in mono-core architecture. We can see a significant improvement in term of the code execution time which is due to the adopted multi-core architecture. The proposed solution covered a multitude axes that are assured through the framework MO^{2}R^{2}S. In the related works, each one does not deal with all of axes covered by our work. In Figure 5.25, it is obvious that the current work is the only one among the related ones to treat all of the seven modules.

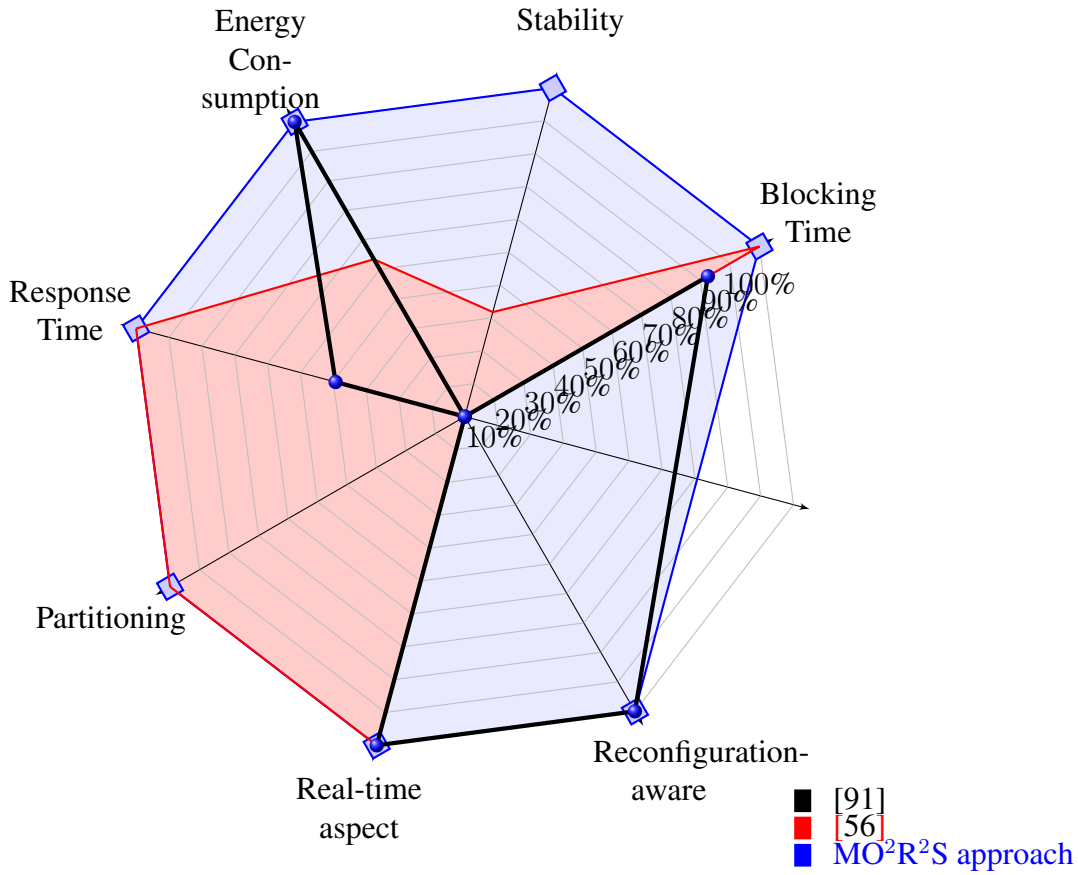

Fig. 5.25 Graph of comparison between the axes covered by the current work compared to the related ones.

Conclusion

In this chapter, we have experimented the proposed methodology where we have implemented the tool-chain environment. The developed tool perform the task generation, partitioning, scheduling, optimization, code generation under real-time and reconfiguration constraints. In the second part of this chapter, we present two real-time case studies, CCAS in mono-core architecture and AV system in

multi-core architecture having reconfiguration scenarios. The comparison studies confirm the performance of the proposed framework compared with related works. The performance of this solution is also seen through the gain in terms of:

- reconfiguration time by 68.2%,
- blocking time by 17.45%,
- processor utilization factor by 2.22%,
- latency by 6.25%,
- code execution time by 19.46%,
- context switching by 26.35%
- preemption by 19.21%.

Note that this framework is extensible to add other metrics. In Future, we extend this approach to deal with scheduling/ partitioning sporadic and aperiodic tasks. It also will be tested with large size instances for more pertinence and reliability.

CHAPTER 6

Conclusion

This final chapter presents a summary of our work and restates the research contributions. The issues for the future work are also identified.

6.1 Context and Problems

Reconfigurble real-time systems are frequently specified by a huge number of functions. Such functions are mostly interacting with each other. Thus, designers are expected to provide the appropriate association of these functions to the real-time tasks that will implement these functions. In addition, the communication of dependent tasks must be carefully managed in order to preserve the efficiency of the system. Designers of reconfigurable real-time systems in multi-core architecture have to tackle these issues:

- software architecture exploration (i.e., assigning functions to tasks),
- task partitioning (i.e., mapping task to core) , and scheduling of tasks,
- code generation.

Comparing to mono-core based approach , multi-core architecture induces an additional challenges in term of how to find a feasible partitioning when taking into consideration shared resources. The latter can introduce a significant blocking time. Moreover reconfiguration causes additional difficulties such as turning from an implementation to another produces a huge moving time overhead, that may affects the total stability of a system [68]. In addition, such system is generally specified by a large number of applicative functions which may cause

- an important time overhead,
- many reconfigurations between the different implementations which increases the reconfiguration time [64],
- increase response time, and energy consumption,

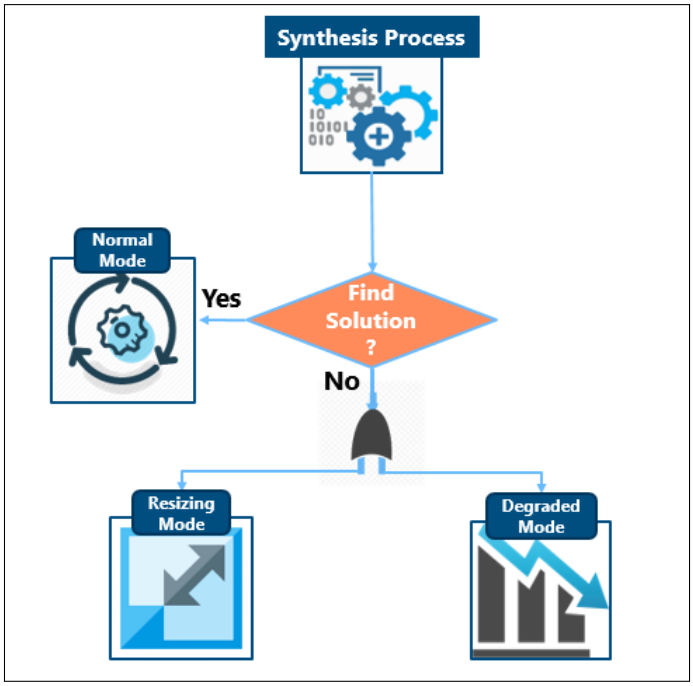

Fig. 6.1 MO^{2}R^2S Modes.

– produce complex system code.

Furthermore, when carrying out certain design steps designers have to make good decisions which is not an easy task. The considered problems in this work are widely explored in literature but none of the proposed solutions simultaneously considers the real-time constraints, reconfiguration property, software architecture exploration, task partitioning, task scheduling, optimization concept and user guidance [68].

6.2 Contributions

This thesis presents a guidance framework for the generation of POSIX code for Reconfigurable real-time systems, through a development process. In order to help designer to make good decisions, the proposed framework provides three possible modes (.i.e., scenarios) based on feasibility constraint (See Figure 6.1).

– 1- Normal mode: it is executed, when no real-time feasibility problems appear.

– 2- Resizing mode: the framework proceeds this mode when real-time feasibility problems occur and the designer chooses to change the hardware architecture to solve feasibility problems,

– 3- Degrading mode: same as the resizing mode it is executed when no feasible solution is found by the framework and the user chooses to degrade the

system performance.

In the first step of the synthesis process, a software architecture is generated from the software model. Then, tasks are assigned to cores while meeting timing properties and minimizing blocking time and moving time. The output of this step is a solution base. Here the proposed approach provides three scenarios either: i) the framework finds at least one solution, so it executes the normal mode in which the obtained solution base would be the input for the second optimization step that optimize the number of tasks in each placement, energy consumption and response time. Finally, the tool generates POSIX code, ii) the framework cannot find any solution and designers choose to resize an HW architecture, then the tool increases the number of cores until it finds the minimal number that ensures a feasible placement. Then it executes the normal mode, iii) The tool cannot find any feasible solution and the user accepts degradation, then the framework tries to find a feasible placement with degraded system performance. Finally, it executes the third and fourth steps of the normal mode to generate POSIX code. The proposed methodology is explained via an experimental study to verify its viability by comparing the obtained results with the related work.

6.3 Perspectives

The developed work is open on many perspectives which we quote below.

- The extension of the proposed models under distributed architectures seems to be the most immediate future work.
- In this dissertation, tasks are periodic. The extension of our proposed model for stochastic, aperiodic and sporadic tasks scheduling is a necessity. This makes the proposed solutions more complete and reliable.
- We aim to consider energy harvesting constraints to achieve energy autonomy and to improve a system's lifetime
- We intend to address security issues in the synthesis process.
- Application of formal verification techniques to the proposed approach,
- We aim to introduce the artificial intelligence AI for the reconfigurable real-time system synthesis.

REFERENCES

[1] Nadine Abdallah, Audrey Queudet, and Maryline Chetto. Task partitioning strategies for multicore real-time energy harvesting systems. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 125–132. IEEE, 2014.

[2] Hedi Abdelkrim, Slim Ben Othman, and Slim Ben Saoud. Fpga implementation of self-reconfigurable fuzzy logic controller. In *2018 International Conference on Advanced Systems and Electric Technologies (IC_ASET)*, pages 151–156. IEEE, 2018.

[3] Omid Abrishambaf, Pedro Faria, Luis Gomes, João Spínola, Zita Vale, and Juan M Corchado. Implementation of a real-time microgrid simulation platform based on centralized and distributed management. *Energies*, 10 (6):806, 2017.

[4] Adewale Akinlawon Adetomi. Dynamic reconfiguration frameworks for high-performance reliable real-time reconfigurable computing. 2019.

[5] Manzoor Ahmad, Nicolas Belloir, and Jean-Michel Bruel. Modeling and verification of functional and non-functional requirements of ambient self-adaptive systems. *Journal of Systems and Software*, 107:50–70, 2015.

[6] Asma Ben Ahmed, Olfa Mosbahi, Mohamed Khalgui, and Zhiwu Li. Toward a new methodology for an efficient test of reconfigurable hardware systems. *IEEE Transactions on Automation Science and Engineering*, 15 (4):1864–1882, 2018.

[7] Zaid Al-Bayati, Youcheng Sun, Haibo Zeng, Marco Di Natale, Qi Zhu, and Brett H Meyer. Partitioning and selection of data consistency mechanisms for multicore real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(4):1–28, 2019.

[8] James H Anderson, John M Calandrino, and UmaMaheswari C Devi. Real-time scheduling on multicore platforms. In *12th IEEE Real-Time and Em-*

bedded Technology and Applications Symposium (RTAS'06), pages 179–190. IEEE, 2006.

[9] Christo Angelov, Krzysztof Sierszecki, and Nicolae Marian. Design models for reusable and reconfigurable state machines. In *International Conference on Embedded and Ubiquitous Computing*, pages 152–163. Springer, 2005.

[10] Andreas Antoniou and Wu-Sheng Lu. Linear programming part ii: Interior-point methods. *Practical Optimization: Algorithms and Engineering Applications*, pages 373–406, 2007.

[11] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.

[12] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[13] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor scheduling for real-time systems*. Springer, 2015.

[14] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[15] Pablo Basanta-Val and Marisol García-Valls. A distributed real-time java-centric architecture for industrial systems. *IEEE Transactions on Industrial Informatics*, 10(1):27–34, 2013.

[16] Diana Bautista, Julio Sahuquillo, Houcine Hassan, Salvador Petit, and José Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7. IEEE, 2008.

[17] Ikbel Belaid, Fabrice Muller, and Maher Benjemaa. Optimal static scheduling of real-time dependent tasks on reconfigurable hardware devices. In *2011 International Conference on Communications, Computing and Control Applications (CCCA)*, pages 1–6. IEEE, 2011.

[18] Kirstie L Bellman, Christian Gruhl, Chris Landauer, and Sven Tomforde. Self-improving system integration-on a definition and characteristics of the

challenge. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 1–3. IEEE, 2019.

[19] Muhammad Herwindra Berlian, Tegar Esa Rindang Sahputra, Buyung Jofi Wahana Ardi, Luhung Wahya Dzatmika, Adnan Rachmat Anom Besari, Rahardhita Widyatra Sudibyo, and Sritrusta Sukaridhoto. Design and implementation of smart environment monitoring and analytics in real-time system framework based on internet of underwater things and big data. In *2016 International Electronics Symposium (IES)*, pages 403–408. IEEE, 2016.

[20] Guillem Bernat, Antoine Colin, and Stefan M Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 279–288. IEEE, 2002.

[21] Antoine Bertout, Julien Forget, and Richard Olejnik. Minimizing a real-time task set through task clustering. In *Proc. International Conference on Real-Time Networks and Systems 22nd*, pages 23–31. ACM, 2014.

[22] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable fpgas. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12. IEEE, 2016.

[23] Aaron Block, Hennadiy Leontyev, Bjorn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2007)*, pages 47–56. IEEE, 2007.

[24] Rahma Bouaziz, Laurent Lemarchand, Frank Singhoff, Bechir Zalila, and Mohamed Jmaiel. Multi-objective design exploration approach for ravenscar real-time systems. *Real-Time Systems*, 54(2):424–483, 2018.

[25] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4nd edition, 2009.

[26] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[27] Giorgio C Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Systems*, 29(1):5–26, 2005.

[28] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science and Business Media, 2011.

[29] Thomas Carle, Dumitru Potop-Butucaru, Yves Sorel, and David Lesens. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. *Leibniz Transactions on Embedded Systems*, 2015.

[30] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James H Anderson, and Sanjoy K Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms., 2004.

[31] Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Heiko Falk. Real-time task scheduling on island-based multi-core platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):538–550, 2015.

[32] T. Chapeaux, P. Rodriguez, L. George, and J. Goossens. The space of feasible execution times for asynchronous periodic task systems using definitive idle times. In *2013 III Brazilian Symposium on Computing Systems Engineering*, pages 95–100, Dec 2013. doi: 10.1109/SBESC.2013.11.

[33] Guangyi Chen and Wenfang Xie. On a laxity-based real-time scheduling policy for fixed-priority tasks and its non-utilization bound. In *International Conference on Information Science and Technology*, pages 7–10. IEEE, 2011.

[34] Jinchao Chen, Chenglie Du, Fei Xie, and Bin Lin. Scheduling non-preemptive tasks with strict periods in multi-core real-time systems. *Journal of Systems Architecture*, 90:72–84, 2018.

[35] Hamza Chniter, Fethi Jarray, and Mohamed Khalgui. Combinatorial Approaches for Low-power and Real-time Adaptive Reconfigurable Embedded Systems. In *Proc. Pervasive and Embedded Computing and Communication Systems 4th*, pages 151–157, 2014.

[36] Hamza Chniter, Yuting Li, Mohamed Khalgui, Anis Koubaa, Zhiwu Li, and Fethi Jarray. Multi-agent adaptive architecture for flexible distributed real-time systems. *IEEE Access*, 6:23152–23171, 2018.

[37] Hamza Chniter, Olfa Mosbahi, Mohamed Khalgui, Mengchu Zhou, and Zhiwu Li. Improved multi-core real-time task scheduling of reconfigurable systems with energy constraints. *IEEE Access*, 2020.

[38] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[39] Robert I Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *Journal of systems architecture*, 65:64–82, 2016.

[40] David Decotigny. *Une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps-réel*. PhD thesis, Université Rennes 1, 2003.

[41] Christian Dietrich, Peter Wägemann, Peter Ulbrich, and Daniel Lohmann. Syswcet: Whole-system response-time analysis for fixed-priority real-time systems (outstanding paper). In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–48. IEEE, 2017.

[42] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4(4):437–455, 2003.

[43] Andreas Ermedahl. *A modular tool architecture for worst-case execution time analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2003.

[44] Leslie R Foulds. *Optimization techniques: an introduction*. Springer Science and Business Media, 2012.

[45] Yong Fu, Nicholas Kottenstette, Yingming Chen, Chenyang Lu, Xenofon D Koutsoukos, and Hongan Wang. Feedback thermal control for real-time systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 111–120. IEEE, 2010.

[46] Kenji Funaoka, Akira Takeda, Shinpei Kato, and Nobuyuki Yamasaki. Dynamic voltage and frequency scaling for optimal real-time scheduling on multiprocessors. In *2008 International Symposium on Industrial Embedded Systems*, pages 27–33. IEEE, 2008.

[47] Aymen Gammoudi, Adel Benzina, Mohamed Khalgui, and Daniel Chillet. Energy-efficient scheduling of real-time tasks in reconfigurable homogeneous multicore platforms. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(12):5092–5105, 2018.

[48] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[49] M. Gasmi, O. Mosbahi, M. Khalgui, L. Gomes, and Z. Li. Performance optimization of reconfigurable real-time wireless sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–15, 2018. ISSN 2168-2216. doi: 10.1109/TSMC.2018.2824900.

[50] Maroua Gasmi, Olfa Mosbahi, Mohamed Khalgui, Luis Gomes, and Zhiwu Li. R-node: New pipelined approach for an effective reconfigurable wireless sensor node. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(6):892–905, 2018.

[51] Stefan K Gehrig and Fridtjof J Stein. Collision avoidance for vehicle-following systems. *IEEE transactions on intelligent transportation systems*, 8(2):233–244, 2007.

[52] Hamza Gharsellaoui, Mohamed Khalgui, Olfa Mosbahi, and Samir Ben Ahmed. New Optimal Solutions For Real-time Reconfigurable Periodic Asynchronous OS Tasks with Minimizations of Response Times. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design*, page 236, 2013.

[53] Aicha Goubaa, Mohamed Khalgui, Zhiwu Li, Georg Frey, and MengChu Zhou. Scheduling periodic and aperiodic tasks with time, energy harvesting and precedence constraints on multi-core systems. *Information Sciences*, 2020.

[54] Monique Guignard-Spielberg and Kurt Spielberg. *Integer programming: State of the art and recent advances*, volume 139. Springer, 2005.

[55] Michael González Harbour. Real-time posix: an overview. In *VVConex 93 International Conference, Moscu*. Citeseer, 1993.

[56] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. In *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 225–230. IEEE, 2018.

[57] Jens Hildebrandt, Frank Golatowski, and Dirk Timmermann. Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 208–215. IEEE, 1999.

[58] Wiem Housseyni, Olfa Mosbahi, Mohamed Khalgui, Zhiwu Li, Li Yin, and Maryline Chetto. Multiagent architecture for distributed adaptive scheduling of reconfigurable real-time tasks with energy harvesting constraints. *IEEE Access*, 6:2068–2084, 2017.

[59] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86. IEEE, 2015.

[60] Xiaowen Jiang, Kai Huang, Xiaomeng Zhang, Rongjie Yan, Ke Wang, Dongliang Xiong, and Xiaolang Yan. Energy-efficient scheduling of periodic applications on safety-critical time-triggered multiprocessor systems. *Electronics*, 7(6):98, 2018.

[61] Yu Jiang and Zhong-Ping Jiang. *Robust adaptive dynamic programming*. John Wiley and Sons, 2017.

[62] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[63] Padmanaban Kesavan and Paul I Barton. Generalized branch-and-cut framework for mixed-integer nonlinear optimization problems. *Computers and Chemical Engineering*, 24(2-7):1361–1366, 2000.

[64] W. Lakhdhar, R. Mzid, M. Khalgui, Z. Li, G. Frey, and A. Al-Ahmari. Multiobjective optimization approach for a portable development of reconfigurable real-time systems: From specification to implementation. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–15, 2018. ISSN 2168-2216. doi: 10.1109/TSMC.2017.2781460.

[65] Wafa Lakhdhar, Rania Mzid, Mohamed Khalgui, and Nicolas Trèves. MILP-based Approach for Optimal Implementation of Reconfigurable Real-time systems. In *Proc. International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 11th*, pages 330–335, 2016.

[66] Wafa Lakhdhar, Rania Mzid, Mohamed Khalgui, and Georg Frey. A new approach for optimal implementation of multi-core reconfigurable real-time systems. In *ENASE*, pages 89–98, 2018.

[67] Wafa Lakhdhar, Rania Mzid, Mohamed Khalgui, Zhiwu Li, Georg Frey, and Abdulrahman Al-Ahmari. Multiobjective optimization approach for a portable development of reconfigurable real-time systems: From specification to implementation. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(3):623–637, 2018.

[68] Wafa Lakhdhar, Rania Mzid, Mohamed Khalgui, Georg Frey, Zhiwu Li, and MengChu Zhou. A guidance framework for synthesis of multi-core reconfigurable real-time systems. *Information Sciences*, 539:327–346, 2020.

[69] Hongtao Lei, Rui Wang, Tao Zhang, Yajie Liu, and Yabing Zha. A multi-objective co-evolutionary algorithm for energy-efficient scheduling on a green data center. *Computers and Operations Research*, 75:103–117, 2016.

[70] Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2 (4):237–250, 1982.

[71] Chang Liu, Yujie Wang, Li Wang, and Zonghai Chen. Load-adaptive real-time energy management strategy for battery/ultracapacitor hybrid energy storage system using dynamic programming optimization. *Journal of Power Sources*, 438:227024, 2019.

[72] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[73] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. A statistical response-time analysis of real-time embedded systems. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 351–362. IEEE, 2012.

[74] Rein Luus. *Iterative dynamic programming*. CRC Press, 2019.

[75] James C Lyke, Christos G Christodoulou, G Alonzo Vera, and Arthur H Edwards. An introduction to reconfigurable systems. *Proceedings of the IEEE*, 103(3):291–317, 2015.

[76] Y. Ma, T. Chantem, R. P. Dick, and X. S. Hu. Improving system-level lifetime reliability of multicore soft real-time systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1895–1905, June 2017. ISSN 1063-8210. doi: 10.1109/TVLSI.2017.2669144.

[77] Kim F Man, Kit Sang Tang, and Sam Kwong. *Genetic algorithms for control and signal processing*. Springer Science and Business Media, 2012.

[78] Syrine Ben Meskina, Narjes Doggaz, Mohamed Khalgui, and Zhiwu Li. Reconfiguration-based methodology for improving recovery performance of faults in smart grids. *Information Sciences*, 454:73–95, 2018.

[79] Microsoft. Samples for Solver Foundation. https://msdn.microsoft.com/en-us/library/ff524501(v=vs.93).aspx, 2017. [Online; accessed 27-July-2020].

[80] Arezou Mohammadi and Selim G Akl. Scheduling algorithms for real-time systems. *School of Computing Queens University, Tech. Rep*, 2005.

[81] Alessia Napoleone, Alessandro Pozzetti, and Marco Macchi. A framework to manage reconfigurability in manufacturing. *International Journal of Production Research*, 56(11):3815–3837, 2018.

[82] Christine Niyizamwiyitira and Lars Lundberg. A utilization-based schedulability test of real-time systems running on a multiprocessor virtual machine. *The Computer Journal*, 62(6):884–904, 2019.

[83] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338, 2011.

[84] CG Papanicolaou and IC Papantoniou. Optimum design of textile-reinforced concrete as integrated formwork in slabs. In *Textile Fibre Composites in Civil Engineering*, pages 245–274. Elsevier, 2016.

[85] Arturo Pérez, Leonardo Suriano, Andrés Otero, and Eduardo de la Torre. Dynamic reconfiguration under rtems for fault mitigation and functional adaptation in sram-based sopcs for space systems. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 40–47. IEEE, 2017.

[86] Franck Petitdemange, Isabelle Borne, and Jérémy Buisson. Modeling system of systems configurations. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, pages 392–399. IEEE, 2018.

[87] Anju S Pillai, Kaumudi Singh, Vijayalakshmi Saravanan, Alagan Anpalagan, Isaac Woungang, and Leonard Barolli. A genetic algorithm-based method for optimizing the energy consumption and performance of multiprocessor systems. *Soft Computing*, 22(10):3271–3285, 2018.

[88] Luis Miguel Pinho and Francisco Vasques. To ada or not to ada: Ada ing vs. java ing in real-time systems. *ACM SIGAda Ada Letters*, 19(4):37–43, 1999.

[89] Luís Miguel Pinho, Brad Moore, Stephen Michell, and S Tucker Taft. Real-time fine-grained parallelism in ada. *ACM SIGAda Ada Letters*, 35 (1):46–58, 2015.

[90] Audrey Queudet-Marchand and Maryline Chetto. Quality of service scheduling in the firm real-time systems. *Real-Time Systems, Architecture, Scheduling, and Application*, page 191, 2012.

[91] Marco Rabozzi. Caos: Cad as an adaptive open-platform service for high performance reconfigurable systems. In *Special Topics in Information Technology*, pages 103–115. Springer, Cham, 2020.

[92] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 116–123. IEEE, 1990.

[93] Ragunathan Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*, volume 151. Springer Science and Business Media, 2012.

[94] Ragunathan Rajkumar, Lui Sha, and John P Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings. Real-Time Systems Symposium*, pages 259–260, 1988.

[95] MV Ganeswara Rao, P Rajesh Kumar, and A Mallikarjuna Prasad. Implementation of real time image processing system with fpga and dsp. In *2016 International Conference on Microelectronics, Computing and Communications (MicroCom)*, pages 1–4. IEEE, 2016.

[96] James Edmund Reeb, Scott A Leavengood, et al. Using the graphical method to solve linear programs. 1998.

[97] Jan Reineke. Challenges for timing analysis of multi-core architectures. In *Workshop on Foundational and Practical Aspects of Resource Analysis*, 2017.

[98] Tobias Ritzau. *Real-time reference counting in RT-Java*. Citeseer, 1999.

[99] Sudipta Roy, Sanjay Nag, Indra Kanta Maitra, and Samir K Bandyopadhyay. International journal of advanced research in computer science and software engineering. *International Journal*, 3(6), 2013.

[100] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[101] Lui Sha, John P Lehoczky, and Ragunathan Rajkumar. Task scheduling in distributed real-time systems. In *IECON'87: Automated Design and Manufacturing*, volume 857, pages 909–917. International Society for Optics and Photonics, 1987.

[102] Lui Sha, Tarek Abdelzaher, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, Aloysius K Mok, et al. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[103] Tom Sheppard. Real-time embedded systems fundamentals. 2011.

[104] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[105] John A Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms*, volume 460. Springer Science and Business Media, 2012.

[106] Georgios L Stavrinides and Helen D Karatza. Scheduling real-time parallel applications in saas clouds in the presence of transient software failures. In *2016 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–8. IEEE, 2016.

[107] Neeraj Suri, Michelle M Hugue, and Chris J Walter. Synchronization issues in real-time systems. *Proceedings of the IEEE*, 82(1):41–54, 1994.

[108] Michael J Todd. An implementation of the simplex method for linear programming problems with variable upper bounds. *Mathematical Programming*, 23(1):34–49, 1982.

[109] Hai Nam Tran, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. Instruction cache in hard real-time systems: modeling and integration in scheduling analysis tools with aadl. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 104–111. IEEE, 2014.

[110] Ahmet Kursad Turker, Adem Golec, Adnan Aktepe, Suleyman Ersoz, Mumtaz Ipek, and Gultekin Cagil. A real-time system design using data mining for estimation of delayed orders an application. 2020.

[111] Osman S Unsal and Israel Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069, 2003.

[112] MMHP van den Heuvel, Reinder J Bril, Johan J Lukkien, Moris Behnam, and Thomas Nolte. Uniform interfaces for resource-sharing components in hierarchically scheduled real-time systems. In *Real-time systems*. InTech, 2016.

[113] Vanessa Vargas, Pablo Ramos, Jean-Francois Méhaut, and Raoul Velazco. Nmr-mpar: A fault-tolerance approach for multi-core and many-core processors. *Applied Sciences*, 8(3):465, 2018.

[114] A. Vulgarakis, R. Shooja, A. Monot, J. Carlson, and M. Behnam. Task synthesis for control applications on multicore platforms. In *Proc. 2014 11th International Conference on Information Technology: New Generations*, pages 229–234, April 2014. doi: 10.1109/ITNG.2014.61.

[115] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. *Dynamic Reconfiguration in Real-Time Systems:)*. 2013.

[116] Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. Real-time high-quality stereo vision system in fpga. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(10):1696–1708, 2015.

[117] X. Wang, I. Khemaissia, M. Khalgui, Z. Li, O. Mosbahi, and M. Zhou. Dynamic low-power reconfiguration of real-time systems with periodic and probabilistic tasks. *IEEE Transactions on Automation Science and Engineering*, 12(1):258–271, Jan 2015. ISSN 1545-5955. doi: 10.1109/TASE.2014.2309479.

[118] X. Wang, Z. Li, and W. M. Wonham. Dynamic multiple-period reconfiguration of real-time scheduling based on timed des supervisory control. *IEEE Transactions on Industrial Informatics*, 12(1):101–111, Feb 2016. ISSN 1551-3203. doi: 10.1109/TII.2015.2500161.

[119] Xi Wang, Imen Khemaissia, Mohamed Khalgui, ZhiWu Li, Olfa Mosbahi, and MengChu Zhou. Dynamic low-power reconfiguration of real-time systems with periodic and probabilistic tasks. *IEEE Transactions on Automation Science and Engineering*, 12(1):258–271, 2014.

[120] Xi Wang, ZhiWu Li, and WM Wonham. Dynamic multiple-period reconfiguration of real-time scheduling based on timed des supervisory control. *IEEE Transactions on Industrial Informatics*, 12(1):101–111, 2015.

[121] Xi Wang, Zhiwu Li, and Walter Murray Wonham. Optimal priority-free conditionally-preemptive real-time scheduling of periodic tasks based on des supervisory control. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(7):1082–1098, 2016.

[122] Jack Whitham, Neil C Audsley, and Robert I Davis. Explicit reservation of cache memory in a predictable, preemptive multitasking real-time system. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–25, 2014.

[123] Niklaus Wirth. Tasks versus threads: An alternative multiprocessing paradigm. *Software-Concepts and Tools*, 17(1):6–12, 1996.

[124] G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *IEEE Transactions on Computers*, 65(2):601–614, Feb 2016. ISSN 0018-9340. doi: 10.1109/TC.2015.2425874.

[125] Jiafeng Zhang, Mohamed Khalgui, Zhiwu Li, Georg Frey, Olfa Mosbahi, and Hela Ben Salah. Reconfigurable coordination of distributed discrete event control systems. *IEEE Transactions on Control Systems Technology*, 23(1):323–330, 2014.

[126] Jiafeng Zhang, Georg Frey, Abdulrahman Al-Ahmari, Ting Qu, Naiqi Wu, and Zhiwu Li. Analysis and control of dynamic reconfiguration processes of manufacturing systems. *IEEE Access*, 6:28028–28040, 2017.

[127] Yi-wen Zhang, Cheng Wang, and Jin Liu. Energy aware fixed priority scheduling for real time sporadic task with task synchronization. *Journal of Systems Architecture*, 83:12–22, 2018.

[128] Wei Zhao. Challenges in design and implementation of middlewares for real-time systems: Guest editors introduction. In *Challenges in Design and Implementation of Middlewares for Real-Time Systems*, pages 1–2. Springer, 2001.

[129] Yecheng Zhao and Haibo Zeng. An efficient schedulability analysis for optimizing systems with adaptive mixed-criticality scheduling. *Real-Time Systems*, 53(4):467–525, 2017.

Appendices

The two models parameters and variables are depicted in the following table 1

Table 1 Models Parameters and Variables.Constants

Constants	
Concepts	Defintion
Hm_{sjk}	A boolean variable used to mention if two tasks are harmonic. Thus if the value of Hm_{sjk} is equal to 1, then the corresponding tasks τ_j and τ_k have harmonic rates in core ζ_s
C_{ij}	Task's WCET
D_{ij}	Tasks' deadline
M	Big constant
N	Number of tasks
Variables	
Concepts	Defintion
Mg_{sjk}	A boolean variable used to mention whether two tasks τ_j and τ_k in core $\zeta_s$are merged such that Mg_{sjk} is equal to 1 if task $\tau_j \in imp_i$ and task $\tau_k \in imp_r$ are merged, the merge corresponds to the situation in which τ_j absorbs τ_k, to be deleted from the model
NewTask	The resulting task model after merging the different tasks (i.e., optimized task model)
$C_{new_{ij}}$	The new task's WCET
$D_{new_{ij}}$	The new task's deadline
$T_{new_{ij}}$	The new task's period
μ_{ijk}	A binary variable where μ_{ijk} = 1 when τ_j is executed before τ_k
U	CPU utilization factor
$B_{new_{ij}}$ \ B_{ij}	The new blocking time \ The old blocking time (i.e., before merging techinque)
$\mathscr{R}_{ij}$	The reponse time of τ_j
y_{ijk}	Number of possible interference of τ_k on τ_j.
x_{ijk}	Number of possible interference of τ_k on τ_j. if μ_{ikj} = 1
η_j	Scaling factor of τ_j
η_{min}	η_{min} = max (ηj), j = 1...N
$C_{new_{n_{ij}}}$	The new normalized WCET of τ_j

.1 General Objective Function

We define in expression (.1) the shared objective function. It aims to maximize the number of merges while minimizing the metric *Metric* which could be either the total response time or the energy consumption.

$$\forall\, s \in \{1..M\}\ Maximize \sum_{i \in \{1..m\}} \sum_{j,k \in \{0,N_i\}} Mg_{sjk} - Metric \qquad (.1)$$

In the following, we present first the common constraints (i.e., the constraints related to merging situations and real-time constraints), then we define the constraints specific to each metric.

.1.1 Common constraints

A we mentioned above, the two formulations share common constraints that we define in this section.

Merging situation constraints.

In order to avoid the merge of non-harmonic tasks we define constraint (.2) [64] $\forall\ s\ \in \{1..M\}$

$$\forall j,k \in \{1..N_i\} \quad Mg_{sjk} = 0 \quad if \quad (Hm_{sjk} = 0) \tag{.2}$$

If Hm_{sjk} = 0 this means that task τ_j and τ_k are not harmonic, therefore Mg_{sjk} will be equal to zero and the two tasks could not be merged. Constraint (.3) is defined to avoid merging task which is already merged i.e., $\forall\ s\ \in \{1..M\}$, $\forall\ k\ \in \{1..N_i\}$

$$\sum_{j \in \{1..N_i\}} Mg_{sjk}\ \leq\ 1,\ \forall\ j,\ k,\ z\ \in\ \{1..N_i\},\ k,z \neq\ j,\ Mg_{sjk}\ +\ Mg_{szj}\ \leq\ 1 \tag{.3}$$

This constraint ensure that the task τ_j can be absorbed by just one task τ_i. Constraint (.4) is defined to create the new obtained model i.e., $\forall\ s\ \in\ \{1 \ldots M\}$

$$\forall\ j\ \in\ \{1 \ldots N_i\},\ NewTask_k\ =\ 1 - \sum_{j \in \{1 \ldots N_i\}} Mg_{sjk} \tag{.4}$$

We define a new boolean variable $NewTask_j$ which presents the new task model after merging. The two constraints .3 and .4 ensure that if Mg_{sjk} = 1 than $NewTask_k$ = 0 and $NewTask_j$=1, it means that the task τ_k is absorbed by τ_j.

Real-time constraints.

The constraints defined in this section are related to real-time requirements. First, we define the model obtained after applying the merge technique. The new WCET $C_{new_{zl}}$ is given by
$\forall j \in \{1..N_i\},\ \forall k \in \{1..N_l\}, \forall i,l \in \{1 \ldots m\}$

$$C_{new_{zl}} = \begin{cases} NewTask_j * (C_{ij} + C_{rk})\ if\ (r = k) \\ (NewTask_j * C_{ij}\ \in\ \Pi_i, NewTask_k * C_{rk}\ \in\ \Pi_r) \text{ otherwise} \end{cases} \tag{.5}$$

As we mentioned previously, if two harmonic task are in the same implementation then the execution time of the $C_{new_{zl}}$ resulting task will be equal to the sum of the execution time of the merged task otherwise it will have different execution time depending on implementation in which it is executed. The constraint .6 computes the new period $T_{new_{zl}}$ which is equal to the minimum period between the merged tasks.

$$T_{new_{zl}} = \ min(T_{ij},\ T_{rk}) \tag{.6}$$

The new priority P_{zl} is defined by the maximum priority between merged tasks.

$$P_{new_{zl}} = \ max(P_{ij},\ P_{rk}) \tag{.7}$$

The CPU utilization factor is an important term in scheduling analysis. In order to ensure that the design model meets the timing constraints the constraint .9 must be verified

$$U \ = \leq \sum_{i \in \{1...m\}} \ \sum_{j \in \{1...N_i\}} N_i(2^{\frac{1}{N_i}} - 1) \tag{.8}$$

Where U is defied by

$$U \ = \ \sum_{i \in \{1...m\}} U_i = \sum_{i \ \in \{1...m\}} \ \sum_{j \ \in \{1...N_i\}} \frac{C_{new_{ij}} \ + \ B_{new_{ij}}}{T_{new_{ij}}} \tag{.9}$$

Where $B_{new_{ij}}$ presents the new blocking time which is defined by: $\forall\ s \in \{1...M\}$

$$B_{new_{ijs}} = \begin{cases} -B_{ijs}\ if\ \sum_{jk \in \{1...N_i\}}\ Mg_{sjk} = \ 0 \\ -max\{B_{ijs}, B_{iks}\} \text{ otherwise} \end{cases} \tag{.10}$$

Where B_{ijs} in (.11) represents the local blocking time of task τ_j in implementation Π_i in core ζ_s which is defined by $\forall\ \tau_k \in Hp_j,\ \forall \varphi_q \in \varphi$

$$B_{ijs} = max\{C_{\varphi_{q_k}}\} - 1 \tag{.11}$$

.1.2 Response Time Optimization Model

If the designer chooses to optimize the response time besides the minimization of the number of tasks, the objective function becomes

$$\forall\ s \in \{1...M\}\ maximize \sum_{i \in \{1..m\}}\ (\ \sum_{\ j,k \in \{0,N_i\}} Mg_{sjk} - \sum_{j \in \{0,N_i\}} \mathscr{R}_{ij}) \tag{.12}$$

Response time $\mathscr{R}_{ij}$ of τ_{ij} is given by

$$\mathscr{R}_{ij} = C_{new_{ij}} + B_{new_{ij}} + \sum_{k \in Hp(j)} I_{ikj} * C_{new_{ik}} \tag{.13}$$

where I_{ikj} the number of interference of τ_{ik} on τ_{ij} during its response time.

$$I_{ikj} = \lceil \frac{\mathscr{R}_{ij}}{T_{ik}} \rceil \tag{.14}$$

Constraint .14 is non linear so in order to compute this constraint we start by adding the following variables:

$y_{ijk} = \begin{cases} N_i \text{ number of possible interference of } \tau_k \text{ on } \tau_j \text{ in } \Pi_i \\ 0, \text{ otherwise} \end{cases}$

y_{ijk} is equal to 0 if there are no interference of τ_k on τ_j in Π_i otherwise it is equal to number of interference of τ_k on τ_j in Π_i. The possible number of interference is defined as function of the response time and period by the following constraint

$$0 \leq y_{ijk} - \frac{\mathscr{R}_{ij}}{T_{ik}} \leq 1 \tag{.15}$$

We define an additional variable x_{ijk} by

$x_{ijk} = \begin{cases} N_i \text{ number of possible interference of } \tau_k \text{ on } \tau_j \text{ in } \Pi_i \; if \; \mu_{ikj} = 1 \\ 0, \text{ otherwise} \end{cases}$

x_{ijk} is defined in constraints .16 and .17 in terms of y_{ijk} and μ_{ijk} by introducing the big M formulation (i.e., M is a big constant [129]).

$$y_{ijk} - M(1 - \mu_{ikj}) \leq x_{ijk} \leq y_{ijk} \tag{.16}$$

$$0 \leq x_{ijk} \leq M * \mu_{ikj} \tag{.17}$$

M is a constant larger than any other quantity involved in the constraint and it is typically used to encode alternative constraints that depend on a binary variable (the value of μ_{ikj} makes one of the constraints trivially true). μ_{ikj} is a binary variable, μ_{ijk} = 1 when τ_j is executed before τ_k it is equal to 0 otherwise. Finally, the response time of task τ_j in Π_i can be computed as

$$\mathscr{R}_{ij} = C_{new_{ij}} + B_{new_{ij}} + \sum_{k \in \{1..N_i\}} x_{ijk} * C_{new_{ik}} \tag{.18}$$

To sum up the full model of the response time optimization is given by:

$$
\begin{cases}
Maximize\ \sum_{i\in\{1..m\}}(\sum_{j,k\in\{0,N_i\}} Mg_{sjk}\ -\ \sum_{j\in\{0,N_i\}} \mathscr{R}_{ij})\ (.12) \\
\\
\forall j,k \in \{1..N_i\} \quad Mg_{sjk} = 0 \quad if \quad (Hm_{sjk} = 0)\ (.2) \\
Mg_{sjk}\ \leq\ 1,\ z\ \in\ \{1..N_i\},\ k,z \neq\ j,\ Mg_{sjk}\ +\ Mg_{szj}\ \leq\ 1\ (.3) \\
\forall\ j\ \in\ \{1\dots N_i\},\ NewTask_k\ =\ 1-\ \sum_{j\in\{1\dots N_i\}} Mg_{sjk}\ (.4) \\
\forall\ i,r,z \in \{1\dots m\},\ j,k,l \in \{1\dots N\} \\
if\ (r = k)\ then\ C_{new_{zl}} = NewTask_j * (C_{ij} + C_{rk})\ (.5) \\
if\ (r <> k)\ then\ C_{new_{zl}} =\ (NewTask_j * C_{ij}\ in\ \Pi_i, NewTask_k * C_{rk}\ in\ \Pi_r)\ (.5) \\
\forall\ i,r,z \in \{1\dots m\},\ j,k,l \in \{1\dots N\}\ T_{new_{zl}} =\ min(T_{ij},\ T_{rk})\ (.6) \\
\forall\ i,r,z \in \{1\dots m\},\ j,k,l \in \{1\dots N\}\ P_{new_{zl}} =\ max(P_{ij},\ P_{rk})\ (.7) \\
B_{new_{ij}} = B_{ij}\ if\ \sum_{jk\in\ \{1\dots N_i\}}\ Mg_{sjk} =\ 0(.10) \\
B_{new_{ij}} = max\{B_{ij}, B_{ik}\}\ \text{otherwise}(.10) \\
B_{ij} = max\{C_{\varphi_{q_k}}\} - 1\ (.11) \\
\forall\ j\ \in\ \{1\dots N_i\},\ NewTask_k\ =\ 1-\ \sum_{j\in\{1\dots N_i\}} Mg_{sjk}\ (.4) \\
0\ \leq y_{ijk}\ -\ \frac{\mathscr{R}_{ij}}{T_{ik}} \leq\ 1\ (.15) \\
y_{ijk}\ -\ M(1\ -\ \mu_{ikj})\ \leq\ x_{ijk}\ \leq\ y_{ijk}\ (.16) \\
0\ \leq\ x_{ijk}\ \leq\ M\mu_{ikj}\ (.17) \\
\mathscr{R}_{ij} =\ C_{new_{ij}} +\ B_{new_{ij}}\ + \sum_{k\in\{1..N_i\}} x_{ijk} * C_{new_{ik}}\ (.18) \\
U\ \leq\ \sum_{i\in\ \{1\dots m\}}\ \sum_{j\in\ \{1\dots N_i\}} N_i(2^{\frac{1}{N_i}}\ -\ 1)\ (.9)
\end{cases}
$$

.1.3 Energy consumption Optimization Model

If the designer chooses to optimize the energy consumption, the objective function is expressed by

$$\forall\ s \in \{1...M\}\ Maximize \sum_{i\in\{1..m\}} (\sum_{j,k\in\{0,N_i\}} Mg_{sjk}\ -\ \sum_{j\in\{0,N_i\}} E_{ij}) \qquad (.19)$$

As we mentioned previously in Section 3.2.4 in Eq. 4.7 the expression of E_{ij} is given by

$$E_{ij} = K \sum_{j\in\{0,N_i\}} \frac{C_{new_{n_{ij}}}}{\eta_j^2} \qquad (.20)$$

We notice that this equation is fractional due to the fact that the WCET of the task $C_{new_{n_{ij}}}$ is proportional to the reduction factor η_j. Thus, we simplify this program by maximizing the minimum of the reduction factor η_j. Hence, we introduce an additional variable η_{min} which is equal to the minimum of η_j. The constraint .21 establishes that

$$\eta_{min}\ \leq\ \eta_j \qquad (.21)$$

The new normalized WCET $C_{new_{n_{ij}}}$ is given by

$$C_{new_{n_{ij}}} = \begin{cases} NewTask_j * (C_{n_{ij}} + C_{n_{rk}}) \; if \; (r = k) \\ (NewTask_j * C_{n_{ij}}, NewTask_k * C_{n_{rk}}) \text{ otherwise} \end{cases} \tag{.22}$$

In order to confirm that the obtained model meets the timing constraints the following constraint must be verified:

$$U = \sum_{i \in \{1...m\}} \sum_{j \in \{1...N_i\}} \frac{C_{new_{n_{ij}}} \eta_j + B_{new_{ij}}}{T_{new_{ij}}} \leq \sum_{i \in \{1...m\}} \sum_{j \in \{1...N_i\}} N_i(2^{\frac{1}{N_i}} - 1) \tag{.23}$$

The full formulation of blocking time optimization is given by

$$\begin{cases} Maximize \; \sum_{i \in \{1..m\}} (\sum_{j,k \in \{0,N_i\}} Mg_{sjk}) + \eta_{min} \; (.12) \\ \\ \forall j,k \in \{1..N_i\} \quad Mg_{sjk} = 0 \quad if \quad (Hm_{sjk} = 0) \; (.2) \\ Mg_{sjk} \leq 1, \; z \in \{1..N_i\}, \; k, z \neq j, \; Mg_{sjk} + Mg_{szj} \leq 1 \; (.3) \\ \forall \; j \in \{1 \dots N_i\}, \; NewTask_k = 1 - \sum_{j \in \{1...N_i\}} Mg_{sjk} \; (.4) \\ \forall \; i,r,z \in \{1 \dots m\}, \; j,k,l \in \{1 \dots N\} \\ if \; (r = k) \; then \; C_{new_{n_{zl}}} = NewTask_j * (C_{n_{ij}} + C_{n_{rk}}) \; (.22) \\ if \; (r <> k) \; then \; C_{new_{n_{zl}}} = (NewTask_j * C_{n_{ij}} \in \Pi_i \\ NewTask_k * C_{n_{rk}} \; in \; \Pi_r) \; (.22) \\ \forall \; i,r,z \in \{1 \dots m\}, \; j,k,l \in \{1 \dots N\} \; T_{new_{zl}} = min(T_{ij}, \; T_{rk}) \; (.6) \\ \forall \; i,r,z \in \{1 \dots m\}, \; j,k,l \in \{1 \dots N\} \; P_{new_{zl}} = max(P_{ij}, \; P_{rk}) \; (.7) \\ B_{new_{ij}} = B_{ij} \; if \; \sum_{jk \in \{1...N_i\}} Mg_{sjk} = 0 (.10) \\ B_{new_{ij}} = max\{B_{ij}, B_{ik}\} \text{ otherwise} (.10) \\ B_{ij} = max\{C_{\varphi_{q_k}}\} - 1 \; (.11) \\ \eta_{min} \leq \eta_j \; (.21) \\ 0 \leq \eta_j \; (.22) \\ U \leq \sum_{i \in \{1...m\}} \sum_{j \in \{1...N_i\}} N_i(2^{\frac{1}{N_i}} - 1) \; (.23) \end{cases}$$

.2 POSIX CODE

Listing 1: POSIX code for the running example.

```
#include <pthread.h>
void* F1 (void* arg);
void* F3 (void* arg);
void* F4 (void* arg);
void* F5 (void* arg);
void* F6 (void* arg);
void* F7 (void* arg);
void* F8 (void* arg);
void* F9 (void* arg);
void* F10 (void* arg);

```

```
12 /********* Controller POSIX code *********/
13 int main (void){ pthread_t tau_1; pthread_t tau_2; pthread_t tau_3;  pthread_t tau_4; pthread_t tau_5;
       pthread_t tau_6;  pthread_t tau_7;
14 if (cnd="Cnd1"){ \\implementation 1
15 pthread_create (&tau_1, NULL, F1, (void *) 30); // Craation of tau_1 thread
16 pthread_create (&tau_2, NULL, F3, (void *) 35); // Craation of tau_2 thread
17 pthread_create (&tau_5, NULL, F7, (void *) 60); // Craation of tau_5 thread...
18 stick_this_thread_to_core(1); //assign thread 1 to core 1 //
19 pthread_join (tau_1, NULL);
20 stick_this_thread_to_core(1);
21 pthread_join (tau_2, NULL);
22 stick_this_thread_to_core(2);
23 pthread_join (tau_3, NULL);...}
24 else{ // Implementation 2
25 pthread_create (&tau_2, NULL, F3, (void *) 35); // Craation of tau_2 thread
26 pthread_create (&tau_6, NULL, F9, (void *) 120); // Craation of tau_6 thread...}
27 return 0;} ...
28 void* F1 (void* arg){///}
29 // core_id = 0, 1, ... n-1, where n is the system's number of cores
30 int stick_this_thread_to_core(int core_id) {
31 int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
32 if (core_id < 0 || core_id >= num_cores) return EINVAL; cpu_set_t cpuset;
33 CPU_ZERO(&cpuset); CPU_SET(core_id,&cpuset); pthread_t current_thread = pthread_self();
34 return pthread_setaffinity_np(current_thread, sizeof(cpu_set_t),&cpuset);}
```