présentée par
Hai Nam TRAN

Préparée à *Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance*

Cache Memory Aware Priority Assignment and Scheduling Simulation of Real-Time Embedded Systems

Hai Nam TRAN: *Cache memory aware priority assignment and scheduling simulation of real-time embedded systems*

Abstract

Real-time embedded systems (RTES) are subject to timing constraints. In these systems, the total correctness depends not only on the logical correctness of the computation but also on the time in which the result is produced (Stankovic, 1988). The systems must be highly predictable in the sense that the worst case execution time of each task must be determined. Then, scheduling analysis is performed on the system to ensure that there are enough resources to schedule all of the tasks.

Cache memory is a crucial hardware component used to reduce the performance gap between processor and main memory. Integrating cache memory in a RTES generally enhances the whole performance in term of execution time, but unfortunately, it can lead to an increase in preemption cost and execution time variability. In systems with cache memory, multiple tasks can share this hardware resource which can lead to cache related preemption delay (CRPD) being introduced. By definition, CRPD is the delay added to the execution time of the preempted task because it has to reload cache blocks evicted by the preemption. It is important to be able to account for CRPD when performing schedulability analysis.

This thesis focuses on studying the effects of CRPD on uniprocessor systems and employs the understanding to extend classical scheduling analysis methods. We propose several priority assignment algorithms that take into account CRPD while assigning priorities to tasks. We investigate problems related to scheduling simulation with CRPD and establish two results that allows the use of scheduling simulation as a verification method. The work in this thesis is made available in Cheddar - an open-source scheduling analyzer. Several CRPD analysis features are also implemented in Cheddar besides the work presented in this thesis.

Résumé

Les systèmes embarqués en temps réel (RTES) sont soumis à des contraintes temporelles. Dans ces systèmes, l'exactitude du résultat ne dépend pas seulement de l'exactitude logique du calcul, mais aussi de l'instant où ce résultat est produit (Stankovic, 1988). Les systèmes doivent être hautement prévisibles dans le sens où le temps d'exécution pire-cas de chaque tâche doit être déterminé. Une analyse d'ordonnancement est effectuée sur le système pour s'assurer qu'il y a suffisamment de ressources pour ordonnancer toutes les tâches.

La mémoire cache est un composant matériel utilisé pour réduire l'écart de performances entre le processeur et la mémoire principale. L'intégration de la mémoire cache dans un RTES améliore généralement la performance en terme de temps d'exécution, mais malheureusement, elle peut entraîner une augmentation du coût de préemption et de la variabilité du temps d'exécution. Dans les systèmes avec mémoire cache, plusieurs tâches partagent cette ressource matérielle, ce qui conduit à l'introduction d'un délai de préemption lié au cache (CRPD). Par définition, le CRPD est le délai ajouté au temps d'exécution de la tâche préempté car il doit recharger les blocs de cache évincés par la préemption. Il est donc important de pouvoir prendre en compte le CRPD lors de l'analyse d'ordonnancement.

Cette thèse se concentre sur l'étude des effets du CRPD dans les systèmes uni-processeurs, et étend en conséquence des méthodes classiques d'analyse d'ordonnancement. Nous proposons plusieurs algorithmes d'affectation de priorités qui tiennent compte du CRPD. De plus, nous étudions les problèmes liés à la simulation d'ordonnancement intégrant le CRPD et nous établissons deux résultats théoriques qui permettent son utilisation en tant que méthode de vérification. Le travail de cette thèse a permis l'extension de l'outil Cheddar - un analyseur d'ordonnancement open-source. Plusieurs méthodes d'analyse de CRPD ont été également mises en œuvre dans Cheddar en complément des travaux présentés dans cette thèse.

ACKNOWLEDGMENT

PUBLICATIONS

Journals

1. *Hai Nam Tran*, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. "Cache-Aware Real-Time Scheduling Simulator: Implementation and Return of Experience." ACM SIGBED Review 13, no. 1 (2016): 15-21. Special issue on 5th Embedded Operating Systems Workshop (EWiLi 2015) in conjunction with ESWEEK 2015.

2. Stéphane Rubini, Christian Fotsing, Frank Singhoff, *Hai Nam Tran*, and Pierre Dissaux. "Scheduling analysis from architectural models of embedded multi-processor systems." ACM SIGBED Review 11, no. 1 (2014): 68-73. Special issue on 3rd Embedded Operating Systems Workshop (EWiLi 2013).

International Conferences

3. *Hai Nam Tran*, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. "Addressing cache related preemption delay in fixed priority assignment." 20th IEEE Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, September 2015.

4. *Hai Nam Tran*, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. "Integration of Cache Related Preemption Delay Analysis in Priority Assignment Algorithm." 4th Embedded Operating Systems Workshop (EWiLi 2014), Lisbon, Portugal, November 2014. (Poster, two-page paper).

5. *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." 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC), Milan, Italy, August 2014, pp 104-111.

6. Pierre Dissaux, Olivier Marc, Stéphane Rubini, Christian Fotsing, Vincent Gaudel, Frank Singhoff, Alain Plantec, Vuong Nguyen-Hong, and *Hai Nam Tran*. "The SMART project: Multi-agent scheduling simulation of real-time architectures." 7th European Congress ERTSS Embedded Real Time Software and System, Toulouse, France, February 2014.

National Conferences

7. *Hai Nam Tran*, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. "Adapting a Fixed Priority Assignment Algorithm to Real-time Embedded Systems with Cache Memory." Colloque National du GDR SoC-SiP, Nantes, France, June 2016 (poster, two-page paper).

8. Frank Singhoff, Alain Plantec, Stéphane Rubini, *Hai Nam Tran*, Vincent Gaudel, Jalil Boukhobza, Laurent Lemarchand et al. "Teaching Real-Time Scheduling Analysis with Cheddar." 9ème édition de l'Ecole d'Eté Temps Réel (ETR), Rennes, France, August 2015.

Communications

9. *Hai Nam Tran*, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. "Addressing cache related preemption delay in fixed priority assignment." LABEX OVSTR Group Meeting, Paris, France, October 2015. (presentation)

10. *Hai Nam Tran*, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. "Cache Modelling and Scheduling Analysis with AADL." AS-2C SAE AADL Committee, Toulouse, France, March 2014. (presentation)

11. *Hai Nam Tran*, Frank Singhoff and Stéphane Rubini. "Handling cache in real-time scheduling simulator." SAPIENT Project meeting, Brest, France, July 2013. (presentation)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADL Architecture Description Language

BCET Best Case Execution Time

BCRT Best Case Response Time

BRT Block Reload Time

CAN Controlled Area Network

CFG Control Flow Graph

CPA CRPD-aware priority assignment

CRPD Cache Related Preemption Delay

CRMD Cache Related Migration Delay

CSH Context Switch Overhead

DM   Deadline Monotonic

DMA Direct Memory Access

DPP   Dynamic Priority Preemptive

DRAM Dynamic Random Access Memory

ECB   Evicting Cache Block

EDF   Earliest Deadline First

EPP   Effective Preemption Point

FIFO First In, First Out

FPP   Fixed Priority Preemptive

GUI   Graphic User Interface

LLF   Least Laxity First

LMB   Live Memory Block

LoC   Line of Code

LRU   Least Recently Used

MDE   Model Driven Engineering

MIT   Minimum Interarrival time

OS   Operating System

QoS   Quality of Service

OPA   Audsley's Optimal Priority Assignment

PPP   Potential Preemption Point

RF   Reuse Factor

RM   Rate Monotonic

RMB   Reaching Memory Block

RTES Real-Time Embedded Systems

RTOS Real-Time Operating Systems

SA   Simulated Annealing

SRAM Static Random Access Memory

UCB Useful Cache Block

WCET Worst Case Execution Time

WCRT Worst Case Response Time

INTRODUCTION

Embedded systems, which are contained within larger devices, are present in many aspects of our daily life. Their usage ranges from general civilian devices, such as cellphones, set-top boxes, car navigation to specific industrial systems, such as factory robots, aircraft control and air traffic management. These systems are designed for a specific function and use limited resources [46, 53]. Embedded systems are typically subject to meet timing constraints, for reasons such as safety and usability. Thus, many of these embedded systems are also *real-time systems*.

Real-time systems are computing systems that must process information and produce responses subject to timing constraints [53, 76, 74]. In these systems, the usefulness of correct outputs and responses either degrades or becomes meaningless if they are produced after a certain deadline. In many cases, missing a deadline can lead to catastrophic system failure such as in a flight control system. In this thesis, we investigate systems that are both embedded and real-time, called *real-time embedded system* (RTES).

Nowadays, most RTES are multi-tasking systems made up of several units of execution called tasks. Each task can have a computational requirement and one or several timing constraints. For a given RTES, information about tasks and available hardware resources are analyzed to ensure that all timing constraints are met. This is achieved by performing scheduling analysis on a model of the RTES.

Scheduling analysis [73] is a method used to verify that a given RTES will meet its timing constraints. It includes the analysis of the scheduling policies along with information about the tasks and available hardware resources to determine whether a system is schedulable or not.

Interactions between tasks and shared resources can potentially make scheduling analysis become complex. For instance, scheduling analysis must also take into account access to any shared hardware resources such as cache memory that can introduce additional delays in term of resource contention.

CONTEXT

The context of this thesis is *priority assignment* and scheduling simulation of RTES with *cache memory*.

Cache Memory and Cache Related Preemption Delay (CRPD)

Cache memory is a crucial hardware component used to reduce the performance gap between processor and main memory. In the context of RTES, the popularization of processors with large size and multi-level cache motivates the proposition of verification methods [52, 23, 2] to handle this hardware component.

Integrating cache memory in RTES generally improves the overall system performance, but unfortunately it can lead to execution time variability due to the variation of preemption cost [65]. When a task is preempted, memory blocks belonging to the task could be removed from the cache. Once this task resumes, previously removed memory blocks have to be reloaded. Thus, a new preemption cost named *Cache Related Preemption Delay* (CRPD) is introduced.

By definition, CRPD is the additional time to refill the cache with memory blocks evicted by preemption [23]. In [65], Pellizzoni and Caccamo showed that CRPD could represent up to 44% of the Worst Case Execution Time (WCET) of a task. In [56], Li et al. showed that the preemption cost could raises from of 4.2µs to 203.2µs when the data set size of programs increases. Thus, taking CRPD into account is crucial when performing scheduling analysis of RTES.

One can consider using cache partitioning technique in which each task has its own space of cache in order to reduce or completely eliminate the effect of CRPD. By doing so, we increase the predictability of a system but decrease the performance in terms of WCET of tasks due to smaller cache space. However, in [5], Altmeyer et al. pointed out that the increased predictability does not compensate for the performance decrease.

There are many research on different domains of scheduling analysis for RTES with cache memory that are presented in Chapter 2. In this thesis, we cover the two domains of scheduling analysis: *priority assignment* and *scheduling simulation*.

Priority Assignment

In most RTES, each task is assigned a priority level that indicates its order of importance. How should priorities be assigned to tasks is one of the most important question regarding the scheduling of a RTES. A poor priority assignment can schedule tasks in an order that is far from optimal [34]. The existence of CRPD raised a question about the applicability and optimality of classical priority assignment algorithms when CRPD is taken into account.

Scheduling Simulation

Scheduling simulation is a popular scheduling analysis method which provides a mean to evaluate the schedulability and detect the unschedulability of a RTES.

It allows RTES designers to perform fast prototyping with a certain level of accuracy. In order to perform scheduling simulation, first, one needs to provide an abstract model of a RTES. Second, the scheduling of the system over a given interval of time is computed and timing properties such as timing constraint violations are evaluated [75]. Cache memory adds a new hardware component that needs to be considered in the system model. In addition, CRPD needs to be taken into account when computing the scheduling of the system.

PROBLEM STATEMENT

There are three problems that are addressed in this thesis.

1. The first problem is regarding the applicability and optimality of classical priority assignments when CRPD is taken into account. One of the most popular assumptions taken in previous literature is that the preemption cost is equal to zero and completely negligible. Of course, this property is not true in the case of RTES with cache memory. Classical priority assignments are either not optimal or not applicable to RTES with cache memory [83]. Indeed, a solution to take CRPD into account while assigning priorities to task is needed.

2. The second problem is that scheduling simulation with regard to the effect of CRPD is still an open issue. There are two unanswered questions concerning (1) a method of modeling and computing CRPD in scheduling simulation and (2) a minimum interval of time needed to perform the simulation that can guarantee the schedulability of a RTES.

3. The third problem is the lack of scheduling simulation facilities that support RTES with cache memory even though there are existing research work in this domain [52, 23, 82, 58, 4, 67]. However, system models that are used in existing scheduling simulation tools do not support evaluating the effect of CRPD or are not compatible with existing research work.

SOLUTION OVERVIEW

In this thesis, we study the methodology of CRPD analysis and propose an application to scheduling analysis. Extensions and improvements are made to classical results in scheduling analysis of RTES in the subject of priority assignment and scheduling simulation. In addition, a scheduling simulator is implemented in order to provide a mean to perform experiments, analyze and observe the effect of CRPD from the perspective of the simulator.

CONTRIBUTION SUMMARY

In this thesis, we study the effect of CRPD on uniprocessor systems in fixed priority preemptive scheduling context where task priorities are statically assigned offline and higher priority tasks can preempt lower priority tasks. Furthermore, we employ our understanding to address the three presented problems. The solution proposed in this thesis is the result of work that leads to the following contributions.

1. **CRPD-aware priority assignment**: To address problem 1, we propose an approach to perform priority assignment and verify the schedulability of RTES while taking into consideration CRPD. To achieve this, we extend the feasibility test proposed by Audsley [7]. The approach consists in computing the interference from computational requirements and CRPD when assigning a priority level to a task and verifying this task's schedulability. There are five solutions proposed. According to the chosen solution, the CRPD computation can be more or less pessimistic and the results in terms of schedulable task sets can be higher or lower. The performance and efficiency of the proposed solutions are evaluated with randomly generated task sets.

2. **CRPD-aware scheduling simulation**: To address problem 2, we propose a CRPD computation model to be used in scheduling simulation. The model is designed to be compliant with the existing work in [52, 23, 2, 58]. We study two properties that make scheduling simulation with our model applicable namely sustainability analysis and feasibility interval.

3. **Available tools**: We address problem 3 by providing an implementation of our contributions and several existing scheduling analysis methods for RTES with cache memory in Cheddar - an Open-Source scheduling analyzer [75]. Cheddar is freely available to researchers and practitioners who want to investigate scheduling analysis of RTES with cache memory. Implementation, examples of use, performance and scalability analysis of our work in Cheddar are provided.

THESIS ORGANIZATION

This thesis is organized as follows. Chapter 1 covers key background knowledge on RTES and scheduling analysis. Chapter 2 discusses about cache memory and CRPD, reviews existing analysis techniques for computing an upper-bound CRPD when performing schedulability analysis and techniques to limit CRPD. The main contributions of this thesis are presented in Chapter 3, 4 and 5. Chap-

ter 3 introduces several priority assignment algorithms that take into account CRPD. Chapter 4 details how scheduling simulation can be used as a verification method for system model with CRPD. Chapter 5 presents the implementation of several CRPD analysis methods in a scheduling analysis tool that allows better study on the effect of CRPD on various scheduling parameters. Chapter 6 concludes the thesis and outlines future work.

Part I

BACKGROUND

Chapter 1

REAL-TIME EMBEDDED SYSTEM

Contents

In this chapter, we discuss about real-time embedded system (RTES) and knowledge that form the basis of the work presented in this thesis. Section 1.1 introduces basic concepts, general properties and classification of a RTES. The most significant property of a RTES is that there exists timing constraints that must be met in system life-cycle. Then, we proceed by introducing the organization of a system, system model and analysis methods that are applied to the model in order to verify that all timing constraints are met. A RTES is divided into three parts: software, real-time operating system and hardware. Each part is discussed in the three sections 1.2, 1.3, 1.4, respectively. Section 1.5 presents scheduling analysis methods that are used to verify whether timing constraints are satisfied or not. We discuss in detail about scheduling simulation of RTES in section 1.6. Finally, section 1.7 concludes the chapter.

1.1 PROPERTIES OF REAL-TIME EMBEDDED SYSTEM

The most simple definition is that a RTES is both a real-time system and an embedded system. First, we present the definition of a real-time system.

Definition 1 (Real-Time System [53, 76, 74])**.** *A real-time system is a computing system in which the total correctness of a program depends not only on the logical correctness of the computation but also on the time in which the result is produced.*

In the context of this thesis, the term real-time means the ability to receive and process a request subject to one or several timing constraints of a computing system [13]. A real-time system has timing constraints called *deadlines*. Timing constraints are the most significant characteristic that classifies a system as a real-time one. Programs in the system must produce results that are subjected to one or several timing constraints. A result that is produced after a program's deadlines may be considered as bad as an incorrect one.

A real-time system often interact with the environment. As a result, it is also described as a system that "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" [60].

Second, we now present the definition of an embedded system.

Definition 2 (Embedded System [46, 53]). *An Embedded System is a microprocessor-based system that is built to control a specific range of functions and not designed to be programmed by the end-user. This kind of system is embedded into a larger device.*

Embedded systems are designed to do specific tasks with limited resources and processing power. The term embedded means that the system is not visible to the end-user as it is part of a larger device.

1.1.1 *RTES Classification*

A RTES can be classified by its level of criticality. A level of criticality [76] is considered as the consequences that happen when a deadline or a timing constraint is missed as well as the ability of a system to recover.

- Hard Real-Time System [6, 24, 57]: the violation of timing constraints is not tolerable and leads to system failure that results in significant damage and casualties. In hard real-time systems, the usefulness of a computational result is zero after its deadline. In addition, interactions at a low level with physical hardware are typically included in these systems. Hard real-time systems are often built under pessimistic assumptions to handle the worst-case scenarios [24]. Examples of hard real-time systems are flight control systems, car engine control systems and medical systems such as heart pacemakers.

- Soft Real-Time System [6, 24, 57]: the violation of timing constraints is tolerable and does not cause system failure. However, they can lead to a degradation in the Quality of Service (QoS) [63]. Soft real-time systems can often tolerate a latency of few seconds. However, the usefulness of a computational result degrades after its deadline. These systems are not built under assumptions regarding the worst-case scenario but are built

to reduce resource consumption and tolerate overhead [24]. Comparing to hard real-time systems, soft real-time ones typically interact at a higher level with physical hardware. Examples of soft real-time systems are video conference and camera control systems.
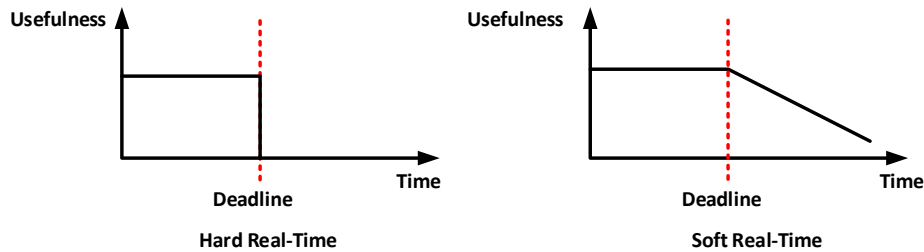
Figure 1: The usefulness of results produced after deadline between hard and soft real-time system. This figure is adapted from [6]

.

- Mixed Critical Real-Time System [85]: a mixed critical real-time system has two or more distinct criticality levels. Each task is assigned a criticality level and the consequences of missing a deadline vary from task to task. Both critical and non critical tasks share the same resources. Timing constraint violation at high criticality level is not tolerable and might cause system failure. Timing constraint violation at low criticality level is tolerable and might cause inconvenient or suboptimal behavior. For example, in an aircraft, we have a flight control system that coexists with a flight information system. The flight control system cannot tolerate any violation of timing constraint while it is possible for the flight information system.

1.1.2 *RTES Architecture*

RTES design is increasingly taking a processor-centric focus [8]. A system is a combination of software running on embedded processor cores, supporting hardware such as memories and processor buses with the help of a real-time operating system. In order to perform analysis, we can separate a RTES into three parts:

- Software

- Real-Time Operating System

- Hardware Platform

We discuss in detail about each part in Section 1.2, 1.3 and 1.4.

1.2   SOFTWARE

Let us consider the architectural decision that one has to take into account when designing the software of RTES. As presented in the previous section, for a given request, a system must produce a response within a specified time. In addition, the system needs to respond to different requests. Timing demands of different requests are different so a simple sequential loop is usually not adequate. The system architecture must allow for fast switching between request handlers. Thus, software in RTES are usually designed as cooperating tasks with a real-time executive controlling them. This design approach is known as the *multi-tasking* approach, which is the focus of this thesis. Multi-tasking approach has entered the mainstream of embedded system design because of the increase in processor speed and advanced operating systems. With multi-tasking, processing resources can be allocated among several tasks. The definition of the term multi-tasking is presented below.

Definition 3 (Multi-tasking [44]). *Multi-tasking is the process of scheduling and switching tasks, making use of the hardware capabilities or emulating concurrent processing using the mechanism of task context switching.*

The terms context switch is defined as follows:

Definition 4 (Context switch [44, 56]). *Context switch refers to the switching of the processor from one task to another.*

Task is the key component in software design of RTES using multi-tasking approach. In this section, we give the definition of a task, describe its life-cycle and introduce its properties.

1.2.1   *Task - Unit of Execution*

Definition 5 (Task [6, 73]). *A task, sometimes also called a process or a thread, is a unit of execution in an application program.*

A single executing program will typically consist of many tasks. Once released, a task has a number of instructions to execute sequentially. A release of a task is called a job. The life-cycle of a task consists of four states provided in Figure 2.

- Inactive: when a task is created, it is in the *inactive* state. The task does not execute nor perform any computation. When a message or an event of activation, which indicates the activation of the task, arrives, the task is released and becomes *ready*.

- Ready: when a task is released and all shared resources are available except the processor, the task is in the *ready* state. In this state, the task waits for
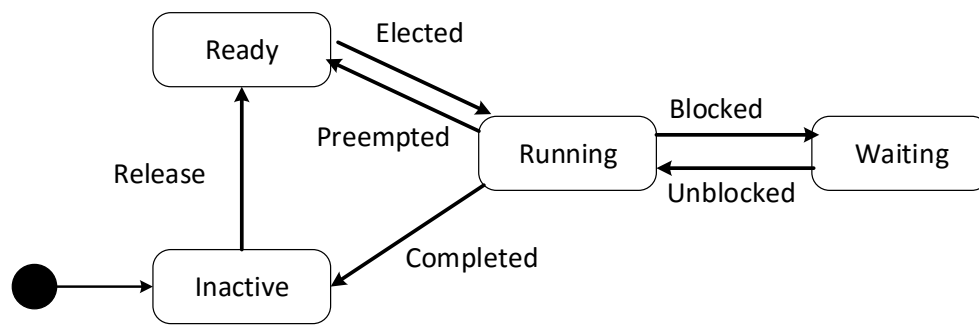
Figure 2: Task life cycle. Adapted from [53]

being elected in order to be executed by the processor amongst other tasks in the system. When a task is elected, it becomes *running*.

- Running: when a task is executed by the processor, it is in the *running* state. In this state, all shared resources and the processor are available to the task. If there is another task elected to be executed by the processor, the current running task is suspended and it returns to the *ready* state. This is a *preemption*. When a task completes its execution, it becomes *inactive*.

- Waiting: when a task is waiting for the availability of shared resources except the processor, it is in the *waiting* state. When the shared resources are available, the task becomes *running*.

1.2.2 *Task Properties*

A task has the following properties that helps determining its order of importance, computational requirement and timing constraints. These properties are used in scheduling analysis, which is introduced later in this chapter, in order to determine the ability of a task to meet its timing constraints.

Definition 6 (Priority [6, 73]). *The priority of a task indicates its order of importance for scheduling.*

A task τ_i has a priority level Π_i, which can be fixed or dynamically assigned. The higher the value of Π_i of a task, the higher this task's priority level. Most of the time, the highest priority task in the ready queue is elected to be executed by the processor.

Definition 7 (Execution Time). *The execution time of a task is the processor time spent executing this task.*

The execution time of a task is not always constant. For example, a task can have different execution paths and different number of loop iterations each time the task executes. The execution paths and the number of loop iterations vary because of the changes of input data. The upper-bound and lower-bound of a task's execution time is defined as follows.

Definition 8 (Worst Case Execution Time (WCET) [89]). *The WCET of a task is the longest execution time of this task.*

Definition 9 (Best Case Execution Time (BCET) [89]). *The BCET of a task is the shortest execution time of this task.*

When designing a system following the multi-tasking approach, in addition to execution time, one has to consider the response time of a task.

Definition 10 (Response Time [64]). *The response time of a job of a task is the interval from its release to its completion.*

In a multi-task scheduling context, a task is not executed immediately when it is released because the required shared resources or the processor may be used by higher priority tasks and are not available. In addition, in a preemptive scheduling context, a lower priority task can be suspended so that the processor can execute a higher priority task which is ready. As a result, a task's response time could be larger than its execution time.

Response time analysis techniques, which are presented in Section 1.5, are used in order to derive the worst case and best case response time of a task.

Definition 11 (Worst Case Response Time (WCRT) [64]). *The WCRT of a task is the longest response time of any of its jobs.*

Definition 12 (Best Case Response Time (BCRT) [64]). *The BCRT of a task is the shortest response time of any of its jobs.*

The WCRT and BCRT of a task are supposed to be smaller than its deadline.

Definition 13 (Deadline [6]). *The deadline of a task is the maximal allowed response time.*

The deadlines that we are using in this thesis are *relative deadlines*. A relative deadline is the relative to the release time of a job [21]. In contrary, an *absolute deadline* is a specific point in time. For example, a job of task τ_i has a relative deadline D_i and is released at time t. It must be completed at time $t + D_i$. In this example, $t + D_i$ is the absolute deadline.

Definition 14 (Offset [7]). *The offset of a task is the time of its initial release.*

Offset attribute is used to model systems in which all tasks are not released at the same point in time. With offsets, some tasks may have initial releases that are later than the other tasks.

Definition 15 (Release jitter [81]). *The release jitter of a task is the worst-case delay between a task arriving (i.e. logically being able to run, yet not having detected as runnable), and being released.*

We would expect a task to start its execution at the time it is released and elected to be run. In practice, this is delayed due to factors such as scheduler overhead and variable interrupt response times. The actual start time of a task is always deviated from its arrival and we can say that tasks suffer from release jitter.

1.2.3 Task Dependencies

Tasks in a RTES may need to cooperate in order to complete a mission so there could be dependencies between them. For example, they need to communicate with each other or sharing a limited number of resources such as I/O devices.

Definition 16 (Dependent Task [73, 6]). *A task whose progress is dependent upon the progress of other tasks.*

It is important to note that in this definition, the competition for processor time between tasks is not accounted as a dependency. Dependent tasks can interact in many ways including precedence dependency and shared resources [6].

Definition 17 (Precedence Dependency [31, 6]). *A task τ_i has a precedence dependency with task τ_j if either τ_i precedes τ_j or τ_j precedes τ_i.*

τ_i precedes τ_j means the nth job of τ_j only be executed after the nth jobs of τ_i is completed. An example of precedence dependency is two tasks that exchange messages. The receiver task needs to wait for a message from the sender task.

Definition 18 (Shared Resource [6]). *A shared resource is a resource accessed by several tasks, in a mutual exclusive manner to enforce data consistency.*

Examples of shared resources are data structures, variables, main memory areas or I/O units. Access to shared resources are often protected by some primitives. When a shared resource is accessed by a task, it becomes unavailable for the others. Other tasks that request access to an unavailable shared resource are blocked.

Tasks that do not have dependencies are called independent tasks.

Definition 19 (Independent Task [73, 6]). *A task whose progress is not dependent upon the progress of other tasks.*

1.2.4   *Task Types*

A task can be classified as either periodic, sporadic or aperiodic, which are defined as follows:

Definition 20 (Periodic Task [57]). *A periodic task is released regularly in a fixed interval.*

For a periodic task, the interval between two releases is called the task's *period*. An example of periodic task is a program that read the information received from a sensor every 1 second.

Definition 21 (Aperiodic Task [74]). *An aperiodic task is not released regularly and there is no minimum separation interval between two releases of this task.*

There are aperiodic events that need to be handled during the life time of a system. For example, emergency events, user interactions are non-periodic. There is a need of aperiodic tasks to handle such events. However, aperiodic tasks make formal verification of RTES much less useful because we cannot bound its resource utilization. From the simple analysis point of view, no system with an aperiodic task can be guaranteed to be feasible.

Definition 22 (Sporadic Task [74]). *A sporadic task is a task which is released regularly but not in a fixed interval. However, there is a minimum separation interval between two releases of this task.*

Sporadic task model is introduced to address scheduling analysis when aperiodic events occur. The context is that we do not know exactly how often a sporadic task will be released; however, there is a minimum interarrival time (MIT) between two releases. This interval provides a safe upper-bound which is used to determine resource utilization of a task. In practice, sporadic tasks are used to handle aperiodic events such as emergency events or user interactions For example, there is not a fixed period of how often a button is pushed by the users in a system; however, there must be a limit because of the hardware's use capability or the speed of interaction.

1.2.5   *Task Set Types*

A set of tasks can be classified into either synchronous or asynchronous, which are defined as follows:

Definition 23 (Synchronous Tasks [7]). *Tasks are called synchronous if the first jobs of all tasks are released at the same time.*

As introduced earlier, a task τ_i has an offset O_i. For synchronous tasks, we have $O_i = \text{Constant}, \forall \tau_i$. If we consider that a system starts when the first task is released, for synchronous tasks, we have $O_i = 0, \forall \tau_i$.

The term synchronous system is used to mention systems that consist of synchronous tasks.

Definition 24 (Synchronous Systems). *Systems in which the first jobs of all tasks are released at the same time.*

In the case of synchronous task, all tasks are released and ready to execute simultaneously at one point in time. This point in time is referred to as a *critical instant*.

Definition 25 (Critical Instant [57]). *A critical instant is a point in time at which all tasks become ready to execute simultaneously.*

Definition 26 (Asynchronous Tasks [7]). *Tasks are called asynchronous if there is at least one first job of a task that is not released at the same time as the first jobs of the other tasks.*

For asynchronous tasks, we have at least two task τ_i and τ_j that have different offsets ($O_i \neq O_j$). We can also classify asynchronous tasks into two subtypes [7] which are:

- Asynchronous tasks with a synchronous release: there exists an instant where all tasks are released and ready to execute simultaneously. In other words, there is a critical instant.

- Asynchronous tasks without a synchronous release: there does not exist an instant where all tasks are released and ready to execute simultaneously. In other words, there is not any critical instant.

The term asynchronous system is also used to refer to system that consists of asynchronous tasks.

Definition 27 (Asynchronous Systems). *Systems in which there is at least one first job of a task that is not released at the same time as the first jobs of the other tasks.*

In the sequel, we use the term *general tasks* to mention tasks that can be either asynchronous or synchronous.

1.3 REAL-TIME OPERATING SYSTEMS

An operating system (OS) is a software that is responsible for managing the hardware resources of a system and software applications running on this system. A

Real-Time Operating System (RTOS) is an OS designed to support the scheduling of real-time tasks with a very precise timing and a high degree of reliability and timing predictability.

A RTOS is different from a general purpose OS such as Microsoft Windows or GNU Linux. A general purpose OS is designed to run many programs and services at the same time and the goal is to maintain user responsiveness, enforce fairness and limit the case of resource starvation. By contrast, a RTOS is designed to run specific applications and the goal is to meet the requirement of timing constraints and reliability.

A RTOS can be defined as an OS with the additional following properties [37]:

- Maximum response time of critical operations such as OS calls and interrupt handling are known. A RTOS can guarantee that a program will run with very consistent timing.

- Interrupt latency and thread switching latency are bounded. It allows fast task preemption. The highest priority task is executed instantly by the processor when it arrives.

- Real-time priority levels are supported. Programmer can assign a priority level to a task. In addition, there are mechanisms to prevent priority inversion.

- A RTOS supports timers and clocks with adequate resolution.

- Advanced algorithms or scheduling policies are provided in order to schedule tasks on the processor. These are implemented as schedulers.

There are several RTOSes that are designed for RTES such as FreeRTOS [49] and RTEMS [71].

This section focuses on two features of a RTOS: scheduler and memory allocation.

1.3.1 *Scheduler*

A scheduler is the part of the RTOS kernel. It decides which task should be executed at a point in time. A formal definition of a scheduler is given by:

Definition 28 (Scheduler [6]). *A scheduler provides an algorithm or a policy for ordering the execution of the tasks on the processor according to some pre-defined criteria.*

A scheduler provides one or several *scheduling policies* that decide the *scheduling* of tasks on the processor.

Definition 29 (Scheduling). *Scheduling is a method by which tasks are given access to resources, noticeably the processor. Scheduling is done according to a scheduling policy.*

Definition 30 (Scheduling Policy). *A scheduling policy (or scheduling algorithm) is the algorithm which describes how tasks are given access to the processor and other shared resources.*

To sum up, tasks are scheduled on a processor by a scheduler following a given scheduling policy. The scheduling policy elects task according to several criteria, rules or algorithms. Those can be considered as characteristic of a scheduling policy and can be used to classify different policies. We present the most general characteristics below:

- Preemptive and non-preemptive

- Online and offline

- Fixed priority and dynamic priority

These characteristics are grouped in mutual exclusive pairs. For example, a scheduling policy cannot be both preemptive and non-preemptive at the same time.

A *Preemptive and non-preemptive scheduling*

Definition 31 (Non-preemptive scheduling [6]). *A non-preemptive scheduler does not suspend a task's execution once this task is executed.*

In non-preemptive scheduling, the RTOS never initiates a preemption. When a task is executed, it occupies the processor until it is completed.

Definition 32 (Preemptive scheduling [6]). *A preemptive scheduler can arbitrarily suspend a task's execution and restart it later without affecting the logical behavior of that task.*

Preemptive scheduling involves the use of an interrupt mechanism that suspends the currently executing task, invokes a scheduler to determine which task should execute next. Preemptive multitasking allows the system to more reliably guarantee each task a regular "slice" of operating time. It also allows the system to rapidly deal with important external events like incoming data, which might require the immediate attention of one task.

B *Online and offline scheduling*

Definition 33 (Offline Scheduler [6]). *A scheduler is offline if all scheduling decisions are made prior to the running of the system.*

Offline scheduling is usually carried out via a scheduling table that lists tasks and their activation times. It means that all tasks are clearly defined before the

system is deployed and will be released at predefined points in time. An offline algorithm takes complete information about the system activities, which reflects the knowledge about anticipated environmental situations and requirements, and creates a single table, representing a feasible solution to the given requirements [39].

The advantages of offline scheduling is that it is highly deterministic because everything is known before runtime. Testing can show that every timing constraint is met. In addition, scheduling is done via table lookup so runtime overhead is low. However, the limitations is that the cost of requirement analysis, system design and testing is very high because everything must be known before runtime. For instance, all information about environmental situations, systems and task parameters and their arrival times must be known for the entire lifetime of the system. Furthermore, it is also difficult to handle aperiodic events with offline scheduler.

Definition 34 (Online Scheduler [6]). *A scheduler is online if all scheduling decisions are made during the run-time of the system.*

An online scheduler makes scheduling decisions during the run-time of the system [6]. The decisions are based on a set of predefined rules or the current state of the system. An offline schedulability test can be used to show that, if a set of rules is applied to a given task set at runtime, all tasks will meet their deadlines [39].

Online scheduling is used because of its flexibility. A new task can be easily added in the system design. However, the limitation is that online scheduling could introduce higher runtime overhead because the need of electing task and handling shared resource at runtime. In addition, online scheduling is less predictable comparing to offline scheduling.

c *Fixed priority and dynamic priority scheduling*

Definition 35 (Fixed priority scheduling [6]). *In fixed priority scheduling, task priorities are fixed and assigned offline (before system starts).*

Task priorities are assigned based on several properties such as relative deadline or period [6]. In classical priority assignment algorithm, tasks only have one priority level. In [86], Wang and Saksena proposed preemption threshold which is a dual-priority system. A task is assigned one nominal priority level and one preemption threshold. Once a task is executed, its priority level raises to preemption threshold level. Thus, it cannot be preempted by higher priority tasks up to a certain priority. We still classify the work in [86] as fixed priority scheduling because task priorities and preemption threshold are fixed and assigned offline.

Definition 36 (Dynamic priority scheduling [6]). *In dynamic priority scheduling, task priorities can be updated during execution.*

The advantage of dynamic priority scheduling over fixed priority one is that it allows systems to be schedulable at a higher processor utilization. However, because of the need to compute and update task priorities online, dynamic priority schedulers are more complex to implement in general and introduce more scheduling overhead [25].

1.3.2 *Memory Allocation*

RTOS may support dynamic and static memory allocation.

Definition 37 (Static Memory Allocation [37]). *Static memory allocation is the allocation of memory at compile or design time. No memory allocation or deallocation actions are performed during execution.*

When using static memory allocation, sizes of the tasks must be known at compile or design time. As a result, the disadvantages are that sizes of data structures cannot be dynamically varied, and programs cannot be recursive. However, it is also fast and eliminates the possibility of running out of memory [50].

Definition 38 (Dynamic Memory Allocation [37]). *Dynamic memory allocation is the allocation of memory at run-time.*

Dynamic memory allocation is sometimes considered a poor design choice because spatial and temporal worst case for allocation and deallocation operations were insufficiently bounded [61]. They lead to unpredictable timing behaviors, which are a problem when designing a RTES. Fully static designs do not have those limitations.

1.4 HARDWARE

This section provides a brief summary about three hardware components of a RTES: processor, memory system and network. Cache memory, which is the focus of this thesis, is detailed in Chapter 2.

1.4.1 *Processor*

In RTES, processors are usually small and have low power consumption. In essence, they are different from processors used in a workstation, laptop or desktop computer. They can be a general purpose processor or application specific instruction-set processor.

We can classify a RTES based on the number of processors.

- Uniprocessor: RTES has only one processor.

- Multiprocessor: RTES has more than one processor. We can distinguish at RTES three kinds of multiprocessor RTES from a theoretical point of view [32].

    – Identical parallel machines: all the processors are identical in the sense that they have the same speed.

    – Uniform parallel machines: each processor is characterized by its own speed.

    – Unrelated parallel machines: there is an execution rate associated with each job-processor pair.

1.4.2 Memory System

On embedded systems, memory is often not expandable or very costly to expand. When programming embedded systems, one needs to be aware of the memory needed to complete a task.

A memory device can be classified based on several characteristics:

- Accessibility: random access, serial access or block access.

- Persistence of storage: volatile storage or non-volatile storage.

- Read/write speed.

- Size.

- Cost.

- Power consumption.

A memory system needs to meet the following requirements. First, processors are built to expect a *random-access* memory. Second, this memory must be *fast* compared to the speed of the processor. If memory speed is too slow compared to processor speed, a high proportion of the execution time of a program is waiting for data to arrive. It will be a significant waste of processing power and energy. Third, this memory needs to be *large*. Nowadays, software are using megabytes of code and expecting up to gigabytes of storage. Finally, from the consumer point of view, this memory must also be *cheap*.

It is possible to provide all technical requirements in a single memory technology; however, the cost will be very high [50]. Thus, in practice, people exploit the *locality of reference* in order to create a memory hierarchy which is able to answer all the requirements above. The idea is to have multiple levels of storage. Each level is optimized for a specific requirement. This point will be discussed in detail in Chapter 2.

1.4.3 *Network*

In a multiprocessor RTES, processors are connected by a network. Messages are sent over the network and could be scheduled by a scheduling policy.

For example, in a Controller Area Network (CAN) bus, each message has a fixed priority level. In addition, a message has several parameters and timing constraints that are similar to a task [90]. We can consider that messages are scheduled on the network while tasks are scheduled on the processors. Thus, scheduling theory can sometimes be applied to the network.

1.5 SCHEDULING ANALYSIS

Scheduling analysis provides a mean to assess the ability of a given RTES to meet its timing constraints. In other words, all the tasks will meet their deadlines during the life-time of the system. It includes the analysis and testing the *feasibility* and *schedulability* of several *scheduling policies* on a specific *system model*.

In this section, the system model used in this thesis is presented. Then, we explain what is feasibility and schedulability in the context of RTES. We present several scheduling policies and tests applied to them.

1.5.1 *System Model*

A system model is an abstraction of a system. A system can be described by different models with different levels of abstraction. In this thesis, we assume the following system model:

- A uniprocessor RTES

- There are n independent periodic tasks: $\tau_1, \tau_2, ..., \tau_i, ..., \tau_n$.

- A task is defined by a quintuple: $(C_i, T_i, D_i, O_i, \Pi_i)$. The five elements are respectively the capacity (or the worst case execution time), the period, the deadline, the offset and the priority of the task τ_i. Task τ_i makes its initial request after O_i units of time, and then releases periodically every T_i units of time. Each release of a task is called a job. Each job requires C_i units of computation time and must complete before D_i units of time. A unique priority level Π_i is assigned to each task. The higher the priority value of a task, the higher its priority level.

- The capacity of a task is smaller than its deadline: $C_i \leqslant D_i$.

- The deadline of a task is smaller than or equal to its period: $D_i \leqslant T_i$.

- There is no dependency and shared software resources between tasks.

- The task set can be either synchronous or asynchronous.

In addition, we introduce the following notations used to discuss about the system model.

- D_{max} is the largest relative deadline in the task set.

- O_{max} is the largest offset in the task set.

- A job of τ_i released at time $t = O_i + k \cdot T_i, k \in \mathbb{N}$ is denoted as $\tau_i[t]$.

- $hp(i)$ (respectively $lp(i)$) is the set of tasks with higher (respectively lower) priority than task τ_i.

- $hep(i)$ (respectively $lep(i)$) is the set of tasks with higher (respectively lower) or equal priority to task τ_i.

The hyper-period of a task set is defined as follows:

Definition 39 (Hyper-period [55]). *The hyper-period* P *is equals to the least common multiplier of all the periods of the tasks.* $P = lcm(T_1, T_2, ..., T_n)$.

The level-i hyper-period of a task is defined as follows

Definition 40 (Level-i hyper-period [7]). *The level-i hyper-period* P_i *of task* τ_i *is equal to the least common multiplier of the periods of* τ_i *and its higher priority tasks* $\tau_j \in hp(i)$. $P_i = lcm(T_i, (T_j | \forall \tau_j, \tau_j \in hp(i)))$.

1.5.2 *Feasibility and Schedulability*

The ability to meet timing constraints of a task set is accessed by its *feasibility* and *schedulability*. The two terms are defined as follows:

Definition 41 (Feasibility [6]). *Feasibility is the assessment of the ability to satisfy all timing constraints of a task set.*

Definition 42 (Feasible [6]). *A task set is feasible if there exists a scheduling policy guaranteeing that all timing constraints are met.*

During the life time of a system, a task set generates sequences of jobs. If all sequences of jobs can be scheduled without any deadline misses, the task set is feasible.

Definition 43 (Schedulability [6]). *Schedulability is the assessment of the feasibility of a task set under a given scheduling policy.*

Definition 44 (Schedulable [6]). *A task set is schedulable under a scheduling policy if none of its tasks, during execution, will ever miss their deadlines.*

Given a space of task sets, the set of schedulable task sets under a given scheduling policy will be a subset of feasible task sets, as illustrated in Figure 3.
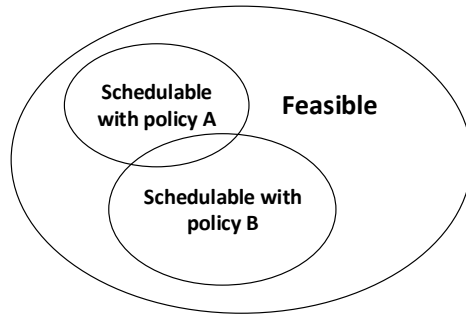
Figure 3: Feasible and schedulable task sets

Algorithms used to verify a system feasibility and schedulability are called feasibility test and schedulability tests.

Definition 45 (Feasibility Test [10]). *A feasibility test assesses whether a task set is feasible or not.*

Definition 46 (Schedulability Test [10]). *A schedulability test assesses whether a task set is schedulable with a given scheduling policy or not.*

Feasibility and schedulability tests use several conditions in order to assess whether a task set is feasible/schedulable or not. We can have three types of feasibility/schedulability condition.

- Sufficient: if these conditions are satisfied, a task set is guaranteed to be feasible/schedulable. If these conditions are not satisfied, a task set can still be feasible/schedulable.

- Necessary: if these conditions are satisfied, a task set is not guaranteed to be feasible/schedulable. If these conditions are not satisfied, a task set is not feasible/schedulable.

- Exact: sufficient and necessary conditions which guarantee that a task set is feasible/schedulable.

We can compare two schedulability tests by the set of schedulable task set found.

- If the set of schedulable task sets found by a test A is a subset of schedulable task set found by a test B, we say that test B dominates test A .

- If the set of schedulable task sets found by test A is identical to test B, we say that two tests are equal.

- In other cases, the two tests are incomparable.

Feasibility and schedulability tests use parameters that are specified in the system design. However, a part of these parameters can never be estimated exactly and there are always deviations in practice. Changes for better scenarios are covered by sustainability analysis and changes for worst scenario are covered by robustness analysis. These are introduced in the next sections.

1.5.3   *Sustainability and Robustness*

Definition 47 (Sustainability [20]). *A given scheduling policy and/or a schedulability test is sustainable if any system that is schedulable under its worst-case specification remains so when its behavior is better than worst-case. The term better means that the parameters of one or more individual task(s) are changed in any, some, or all of the following ways:* .

1. *Decreased capacities*

2. *Larger periods*

3. *Larger relative deadlines*

4. *Smaller release jitter*

Modeling and scheduling analysis by WCET is reasonable only when the analysis is sustainable regarding execution time parameter. Decreased execution time comes from the deviation in theoretical analysis and practical execution. A task can execute shorter than its WCET. This change is not predictable. If scheduling analysis with the WCETs of tasks is not sustainable regarding this change, we need to perform scheduling analysis with all possible values which are smaller than the WCETs of tasks, leading to an exponential complexity.

Sporadic task model is analyzable if the analysis is sustainable regarding period or MIT parameter. If not, we can only analyze system model with periodic tasks.

Definition 48 (Robustness [33]). *The capability of a system to meet its timing constraints despite the occurrence of additional interference.*

Robustness is a concept used in general system development. The term additional interference consists of unpredictable internal or external perturbation that can affect the system. Tasks in real-time system may experience various additional interferences as listed in [33]:

- Effects of interrupts; interrupts occurring in bursts/ at ill-defined rates, using more execution time than expected.

- Ill-defined RTOS overheads.

- Tasks exceeding their expected execution times.

- Processor cycle stealing by peripheral control units such as Direct Memory Access (DMA) devices.

- Ill-defined critical sections where interrupts and hence task switches are disabled, possibly due to the behavior of the RTOS.

- Errors occurring at an unpredictable rate, causing check-pointing mechanisms to re-run part or all of a task

1.5.4 *Fixed Priority Preemptive Scheduling*

Under fixed priority preemptive (FPP) scheduling, each task is assigned a priority level. Task can preempt each other based on the statically assigned priorities. In this section, first we introduce priority assignment algorithms. Second, we present schedulability tests applied to FPP scheduling.

A *Priority Assignment*

One of the most known priority assignment algorithms are the Rate Monotonic (RM) and Deadline Monotonic (DM).

RM assigns priority levels to periodic tasks based on their periods. The shorter the period of a task, the higher its priority level. It was shown that for synchronous periodic tasks with deadlines on requests ($\forall \tau_i : T_i = D_i, O_i = 0$), RM is the optimal priority assignment algorithm.

DM assigns priority to tasks based on their relative deadlines. The shorter the relative deadline of a task, the higher its priority level. Leung and Whitehead [55] showed that for synchronous tasks with deadlines less than or equal to their periods ($\forall \tau_i : D_i \leqslant T_i, O_i = 0$), (DM) is optimal.

Audsley [7] addressed asynchronous periodic tasks with arbitrary deadlines (T_i and D_i are not related). Audsley's priority assignment algorithm is optimal in the sense that for a given RTES model, it provides a feasible priority ordering resulting in a schedulable RTES whenever such an ordering exists. For n tasks, the algorithm performs at most $n \cdot (n + 1)/2$ schedulability tests and guarantees to find a schedulable priority assignment if one exists.

Audsley's algorithm starts by assigning the lowest priority level 1 to a given task τ_i. Then, a feasibility test is used to verify whether τ_i is schedulable or not. If τ_i is not schedulable at priority level n, the algorithm tries to assign the priority level n to a different task. If τ_i is schedulable, the algorithm assigns priority level 1 to τ_i and then, moves to the next priority level. The algorithm continues until all tasks are assigned a priority level. If there is not any schedulable task at a

given priority level, the RTES is not schedulable and the algorithm terminates. The pseudo code of Audsley's algorithm is given below.

```
1 for each unassigned priority level i, lowest first loop
2   for each unassigned task  τ loop
3     if τ is schedulable is priority i then
4       assign τ to priority i
5       break (continue outer loop)
6     end if
7   end loop
8   return unschedulable
9 end loop
10 return schedulable
```

Davis and Burns [33] improved Audsley's algorithm by introducing a robust priority assignment algorithm. This work deals with the problem of robustness. As defined earlier, a robust system retains schedulable even when it operates beyond the worst-case assumptions as permitted by the interpretation of its specification [20]. The problem is that tasks in RTES may be subject to additional interferences of various types such as: interrupt handling, scheduling overhead and tasks exceeding their WCET. The previous priority assignment algorithms did not take into account this factor. The proposed algorithm in [33] assigns priority to a task, which is not only feasible but also can tolerate highest number of additional interference. The pseudo code of this priority assignment is given below. The additional tolerable interference is denoted as α.

```
1 for each priority level i, lowest first loop
2   for each unassigned task τ loop
3     binary search for the largest value of α
4     for which task τ is schedulable at priority level i.
5   end loop
6   if no task are schedulable
7     return unschedulable
8   else
9     assign the schedulable task that
10    tolerates the max α at priority level i to
11    priority level i
12 end loop
13 return schedulable
```

In [86], Wang and Saksena proposed preemption threshold which is a dual-priority system. A task is assigned one nominal priority level and one preemption threshold. Once a task is executing, its priority level raises to preemption threshold level. Thus, it cannot be preempted by higher priority tasks up to a

certain priority. The author have shown that the proposed preemption threshold can improve schedulability and reduce preemption overhead.

B *Feasibility and Schedulability Test*

There are several feasibility/schedulability tests applied to FPP scheduling. In this section, we address the three popular tests.

The *first test* is based on processor utilization factor. This test is sufficient but not necessary. It can be applied to RM and DM in preemptive scheduling context. The utilization of the processor by a task τ_i is computed as follows:

$$U_i = \frac{C_i}{T_i} \tag{1}$$

The total processor utilization of a task set that consists of n tasks is computed as follows:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2}$$

In [57], Liu and Layland have presented two results regarding the schedulability analysis of synchronous independent periodic tasks:

Theorem 1 ([57])**.** *In FPP scheduling context, a task set of n synchronous independent periodic tasks with $D_i = T_i$, executing on a uniprocessor, is schedulable by RM if:*

$$U \leqslant n(2^{1/n} - 1) \tag{3}$$

Theorem 2 ([57])**.** *In FPP scheduling context, a task set of n synchronous independent periodic tasks with $D_i \leqslant T_i$ executing on a uniprocessor, is schedulable by DM if:*

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leqslant n(2^{1/n} - 1) \tag{4}$$

The *second test* is based on verifying that all the deadlines over the *feasibility interval* are met. We detail the definition and computation of feasibility interval of general tasks in Section 1.6.1. This test is sufficient and necessary for general tasks. It can be applied to any priority assignment algorithms in FPP scheduling context.

The *third test* is based on the computation of task WCRT. This test is sufficient and necessary for task set that consists of synchronous independent periodic tasks. It can be applied to any priority assignment in FPP scheduling context. For a given task set, the WCRT R_i of a task τ_i can be computed and compared against the deadline using the following equation [51]:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{5}$$

The task set is schedulable if all tasks meet their deadlines (i.e. $\forall i : R_i \leqslant D_i$).

1.5.5 *Dynamic Priority Preemptive Scheduling*

Under dynamic priority preemptive (DPP) scheduling, a task is not assigned a priority level. The scheduler decides which task has the highest priority level during run time.

A *Priority Assignment*

- Earliest Deadline First [57]: Earliest Deadline First (EDF) assigns priority levels to tasks based on their absolute deadline at a given instant during execution. The nearer the absolute deadline of a job of a task, the higher its priority level.

- Least Laxity First [35]: Least Laxity First (LLF) assigns priority levels to tasks based on the laxity attributes. For a job of a task, its laxity is defined as the difference between the task's relative deadline and its remaining execution time. The smaller the laxity value of a job of task, the higher its priority level.

B *Feasibility and Schedulability Test*

There are several feasibility/schedulability tests applied to DPP scheduling. In this section, we address the three popular tests that are applicable to EDF. The *first test* is based on processor utilization factor:

Theorem 3 ([57]). *A task set of* n *synchronous independent periodic tasks, executing on a uniprocessor, and with* $T_i \geqslant D_i$*, is schedulable by EDF scheduling if, and only if:*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1 \tag{6}$$

The *second test* is based on on verifying that all the deadlines over the feasibility interval are met. In [54], Leung and Merrill noted that a set of periodic tasks is schedulable if and only if all absolute deadlines in the interval $[0, O_{max} + 2H)$ are met. This is an exact test. In [9], Baruah and Rosier extended this condition for sporadic task systems. They showed that a task set is schedulable if and only if $\forall t > 0$, $h(t) < t$, where $h(t)$ is the processor demand function which calculates the maximum execution time requirement of all jobs which have both their arrival times and their deadlines in a contiguous interval of length t, $h(t)$ is given by:

$$h(t) = \sum_{i=1}^{n} \max \left\{ 0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right\} C_i \tag{7}$$

In addition, the value of t can be bound by an easily computed value. The feasibility condition is given by:

Theorem 4 ([9]). *A general task set is schedulable if and only if* $U \leqslant 1$ *and* $\forall t < L_a, h(t) < t$ *where* L_a *is defined as follows:*

$$L_a = \max\left\{ D_1, ..., D_n, \max_{1 \leqslant i \leqslant n}\{T_i - D_i\}\frac{U}{1-U} \right\} \tag{8}$$

The *third test* is based on worst-case response time computation. This is more complex to compute than in the case of FPP scheduling. This thesis does not focus on DPP scheduling so the presentation of these tests are not included.

1.6 SCHEDULING SIMULATION

Scheduling simulation is used to analyze the feasibility and schedulability of RTES. It focuses on evaluating scheduling events and the ability to satisfy timing constraints of a system.

The concept of simulation is well-known in computer sciences. It is the discipline of (1) designing a system model of an actual or theoretical physical system, (2) executing the model on a digital computer, and (3) analyzing the execution output [68].

A system model is an abstraction of a system architecture. A system can be described by different models with different levels of abstraction. The architecture of a system can be described by an Architecture Description Language (ADL). By definition, an ADL is a language that supports the modeling of high-level structure of the system. An ADL does not focus on modeling the implementation details of the system. Examples of ADLs are AADL [38] and MARTE-UML [16].

The system model is then executed by a scheduling simulator. The execution of a model must follow a scheduling policy. In general, scheduling policies are implemented in or handled by the simulator.

Execution output consists of information regarding system's feasibility and schedulability.

In the next sections, we discuss about the concept of feasibility interval and related work on this subject. In addition, we also present existing scheduling simulators.

1.6.1 *Feasibility Interval*

One of the most important question when performing scheduling simulation is how long should we simulate a system. Ideally, we need to be able to capture all the possible behaviors of our system model or at least the worst case in the simulation interval. The minimum interval in which we should perform the simulation is known as feasibility interval.

Definition 49 (Feasibility Interval [43]). *A feasibility interval* I_F *is a finite interval such that it is sure that no deadline will ever be missed if and only if, when we only keep the requests made in this interval, all deadlines for them in this interval are met [43].*

We present existing work on feasibility interval of RTES.

A *Synchronous systems*

In [57], Liu and Layland proved that for a synchronous system, if deadlines of tasks are guaranteed for releases starting at a critical instant, they can be guaranteed for the lifetime of the system. Later, in [43], Goossens and Devillers deduced the feasibility interval which is $[0, D_{max})$.

B *Asynchronous systems*

In [43], Goossens and Devillers proved that any feasible schedule of an asynchronous system is finally periodic, i.e. periodic from some point.

One of the first results about feasibility interval of an asynchronous system is presented in [55]. For an asynchronous system with a FPP scheduler, the feasibility interval is $[O_{max}, O_{max} + 2 \cdot P)$. This is not optimal since it does not reduce to $[0, D_{max})$ in the case of synchronous systems.

Later, the result is improved. For asynchronous systems, the concept of stabilization time was introduced in [7] and [43]. In these systems, there could be an interval of time, in which lower priority tasks are released and executed while higher priority tasks are not released. In this interval, a system is considered to be not stabilized. Stabilization time is defined as follows:

Definition 50 (Stabilization time [7, 43]). *Stabilization time* S_i *of a task* τ_i *is an instant at a release time of* τ_i *when all tasks* $\tau_j \in hp(i)$ *are released and stabilized.*

The computation of S_i is inductively defined by [43]:
$S_1 = O_1$,
$S_i = max(O_i, O_i + \lceil \frac{S_{i-1} - O_i}{T_i} \rceil \cdot T_i)$ $(i = 2, 3, ..., n)$.
For a task τ_i, the feasibility interval is $[0, S_i + P_i)$. It gives the feasibility interval of $[0, S_n + P]$ for all the system in which S_n is the stabilization time of the lowest priority task τ_n. Later, in [43], Goossens and Devillers pointed out that the interference a task experiences before the stabilization time is less than or equal to the one after. As a result, for each τ_i one only has to check the deadlines in the interval $[S_i, S_i + P_i)$.

1.6.2 *Scheduling simulator*

The general properties of a RTES scheduling simulator are given below.

- Supporting an abstract model of a system. A model could include software components or hardware components. Depending on the purpose of the simulation, a simulator can support all system components or a part of them.

- Supporting one or several scheduling policie(s).

- Supporting a simulation of tasks over a period of time.

There were many scheduling simulators developed. MAST [42] is a modeling and analysis suite for real-time applications. The hardware component abstraction of MAST model is generic and it includes processing resources and shared resources.

STORM [84], YARTISS [27] and RTSim [1] are scheduling simulation tools mainly designed for evaluating and comparing scheduling algorithms for multiprocessor architectures. YARTISS also supports energy-aware scheduling simulation.

SymTA/S [47] and RealTime-at-Work [2] are model-based scheduling analysis tools targeting automotive industry. The hardware components supported in those tools are specific to their domains (ECU, CAN and AFDX Networks).

SimSo [28] is a scheduling simulation tool that supports cache sharing on multi-processor systems. It takes into account impact of the caches through statistical models and also the direct overheads such as context switches and scheduling decisions. The memory behavior of a program is modeled based on Stack Distance Profile - the distribution of the stack distances for all the memory accesses of a task, where a stack distance is by definition the number of unique cache lines accessed between two consecutive accesses to a same line [62].

1.7 CONCLUSION

The main objective of this chapter was to provide a brief summary about subjects that form the background of the thesis. An introduction about the properties of a RTES and its main components including software, RTOS and hardware are provided. We have presented the system model, the notation used in this thesis and how scheduling analysis is done on a given system model in order to verify that all timing constraints are met. To sum up, scheduling analysis methods evaluate a system's schedulability based on a model of its software and hardware together with a scheduling policy provided by its RTOS.

The next chapter presents the problem that appears when cache memory is included in the hardware of a RTES. Indeed, software and hardware models must be updated in order to take into account this new hardware component

---

1 RTSim, http://rtsim.sssup.it/
2 RealTime-at-Work, http://www.realtimeatwork.com/

and its effect on task execution. Furthermore, scheduling analysis methods are also extended with regard to the effect created by this hardware component.

Chapter 2

CACHE MEMORY AND CACHE RELATED PREEMPTION DELAY

Contents

This chapter provides a brief summary of the basic concepts about cache memory and the problems created by the presence of cache memory in RTES. Cache memory is important because it provides data to a processor much faster than main memory. It helps reducing the memory latency and thus decreasing system response time. However, in RTES, because cache memory is shared between tasks and memory accesses are not always predictable, it creates several problems when applying scheduling analysis methods, which are based on pessimistic but highly predictable assumptions, to verify system schedulability.

In section 2.1, we present the need of cache memory and memory hierarchy. Basic concepts about cache memory are introduced in section 2.2. Section 2.3 details the problem of cache memory in RTES and introduces a new preemption cost named *Cache Related Preemption Delay* (CRPD). The computation of CRPD is presented in section 2.4. In section 2.5, we present how scheduling analysis methods are extended to take into account CRPD. Finally, section 2.6 concludes the chapter and presents the position of our work.

2.1 THE NEED OF CACHE MEMORY AND MEMORY HIER-ARCHY

In this section, we explain why a single level memory is not practical in modern RTES and the need of cache memory and memory hierarchy.

Memory accesses are very common in programs. The time it takes to load the data from memory to the processor is called the latency of the memory operation. It is usually measured in processor clock cycles or ns.

Modern processors are fast in the sense that they can run normally at clock speeds of several GHz and can execute more than one instruction per clock cycle. For example, a 3 GHz processor capable of executing 3 instructions per cycle has a peak execution speed of 9 instructions per ns. Thus, a memory must be fast in order to match the processor's speed. In addition, all microprocessors expect a *random access* memory [50]. In other words, any particular datum is needed at any given moment and there is no constraint about the placement order of instruction or data in the memory.

Modern software application is written to expect hundred megabytes or gigabytes of storage for data. For example, a camera control system needs memory to store the recorded images. Therefore, a memory must be *large* in order to match the storage requirement. In addition, it must also support *permanent storage*.

All the requirements above can be achieved with a single memory technology but the cost is tremendous and considered not practical. Beside the technical requirements, a memory must be *affordable* by the consumers. This last requirement is considered to be mutual exclusive with the others. Consequently, a solution that consists of a single level memory is not practical and memory hierarchy is introduced in order to address this problem.

2.1.1 *Memory Hierarchy*

One fundamental principle that found the interest of memory hierarchy and cache memory is locality. There are two types of locality:

- *Temporal locality [50]*: If a program uses a memory block, this memory block is likely to be used again. Temporal locality is also called locality in time.

- *Spatial locality* [50]: if a program uses a memory block, memory blocks that are close to this memory are likely to be used. Spatial locality is also called locality in space.

Based on temporal locality, memory blocks in higher level memory (e.g main memory) should be loaded into the cache memory to take advantage of latency. Based on spatial locality, memory blocks that are closer to an accessed one should be prefetched into the cache too.
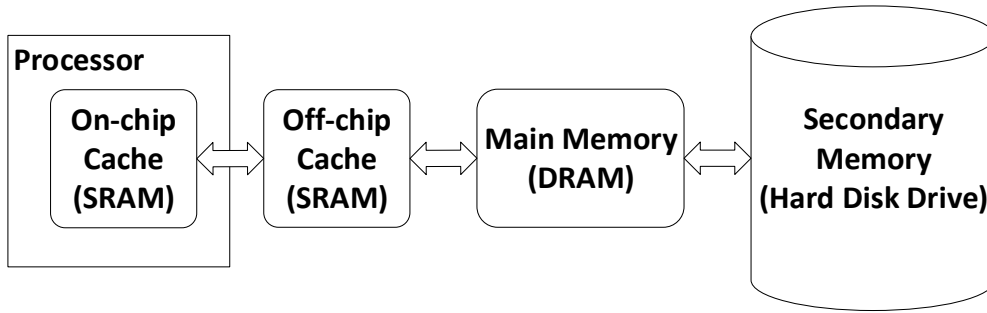
Figure 4: Memory hierarchy. Adapted from [50]

Technology	Access Latency	Cost per Megabyte
On-chip Cache (SRAM)	100 of picoseconds	$1-100
Off-chip Cache (SRAM)	Nanoseconds	$1-10
Main Memory (DRAM)	10-100 nanoseconds	$0.1
Secondary Memory (Hard Disk Drive)	Milliseconds	$0.001

Table 1: Performance-cost of memory technologies. Adapted from [50]

Because of the locality principle, a fast, large and expensive single-level memory system is unnecessary. In a small interval of time, a program does not need all of its data accessible immediately. Therefore, we can have a multi-level of storage. The first level of storage, which is fast, small and expensive, provides immediate access to a subset of the program's data. The remainder of the data is stored in higher levels of storage, which are slower but larger and cheaper than the first level memory.

A memory hierarchy that consists of multiple levels of storage is implemented. Each level of storage is optimized for a purpose. Figure 4 and Table 1 provide an illustration of the modern memory hierarchy and information regarding the cost and the performance of its main components.

- *Disk*: disk provides permanent storage at an ultra-low cost per bit [50].

- *Main memory*: main memory is usually made of DRAM (Dynamic Random Access Memory). It provides a random-access storage that is relatively large, relatively fast, and relatively cheap. The speed of main memory is quite slow comparing to processor's speed. As we can see in the Table 1, an access to main memory takes between 10 and 100 ns. Then, the processor may have to wait for the data to arrive. If a processor can execute 9 instructions per ns, it can execute more than 90 instructions in the time waiting to perform a single data access on main memory or hard disk. Memory

access latency on main memory is high comparing to an instruction execution time.

- *Cache memory*: cache memory is usually made of SRAM (Static Random Access Memory). It is a small, but extremely fast memory, lies between the processor and the main memory. Cache is introduced in order to reduce the memory access latency. Frequently used data are automatically loaded into the cache.

  The capacity of cache memory is often limited and much smaller than main memory because of the following reason: cost and chip size. It is considered expensive to have a large cache memory. In addition, the first level cache memory is typically embedded in the processor chip and the chip size is limited.

2.2 BASICS CONCEPTS ABOUT CACHE MEMORY

In this section, first, we present the classification of cache memory. Second, we detail cache memory organization and explain how a memory block in main memory is mapped into cache memory. In addition, we present the operations related to this hardware component.

2.2.1 *Cache classification*

Cache memory is classified based on size, memory access latency and also the closeness to the processor. Most of the time, there are three layers of cache on modern processors.

- *L1 cache*: L1 cache is a extremely fast but relatively small cache memory. The size of L1 cache is around 4-32 KiB. L1 cache is typically embedded in the processor chip. It is very close to the processor and is accessed on every memory access. As a result, from the architectural consideration, this cache needs to have a lot of read/write ports and very high access bandwidth. It is considered impossible or extremely costly to built a large L1 cache with these properties.

- *L2 cache*: L2 cache is a bit slower but larger than L1 cache. The size of L2 cache is around 128-512 KiB. L2 cache may be embedded in the processor chip or located on a separate chip with a high-speed alternative bus (separated from main system bus) interconnecting the cache to the processor. L2 cache is only accessed when a miss on L1 cache occurs. Thus, it can have a higher memory access latency, less ports and lower access bandwidth. These properties allow us to make L2 cache bigger.

- *L3 cache*: L3 cache is significantly slower and larger than L1 and L2 caches. The size of L3 cache is around 4-8 MiB. L3 cache is only accessed when a miss on L2 cache occurs.

Cache memory can also be classified by the data stored in the cache.

- *Instruction cache*: instruction cache only holds program instruction. Processor only reads from the instruction cache and performs no write operation.

- *Data cache*: data cache only holds program data. Processor reads from and writes to the data cache.

- *Unified cache*: unified cache stores both program instructions and data.

An access to a memory block in the main memory can be classified as a *cache hit* or *cache miss*, which are defined as follows:

Definition 51 (Cache hit). *A cache hit is an access to a memory block that is in the cache.*

Definition 52 (Cache miss). *A cache miss is an access to a memory block that is not in the cache.*

We proceed by presenting the three characteristics including cache architecture, associativity and replacement protocol.

2.2.2 *Cache organization*

To understand the organization of cache memory, we present the definition of the term cache line.

Definition 53 (Cache line [50]). *Cache line is the smallest unit of data that a cache can handle.*

A cache is subdivided into cache lines. The size of a cache line is determined by both the processor and the cache design. The physical location in the cache memory where a line is stored is called a *cache block*. In fact, for the reason of simplicity we consider that two terms are equivalent.

Now, we detail how a memory block in the main memory is mapped into cache memory.The term memory-to-cache mapping scheme is defined as follows:

Definition 54 (Memory-to-cache mapping scheme). *A memory-to-cache mapping scheme is a set of rules that specify how a memory block in the main memory is mapped into the cache memory.*

The hardware implementation of the cache memory can be seen as a hash table [50]. The key column is then the address of a memory block in the main memory. There are three types of memory-to-cache mapping scheme:

1. *Direct mapped*: a memory block in the main memory can only be mapped to one distinct cache block in the cache. The mapping is usually computed as follows:

$$\texttt{(Block address) MOD (Number of blocks in cache)}$$

Direct mapped is the most simple memory-to-cache mapping scheme. It only requires us to compare the address of a memory block in the main memory with the address of a cache block. The advantage of this memory-to-cache mapping scheme is that it is simple and not expensive to implement. However, the disadvantage is that a direct mapped cache is not flexible and often provides low performance due to high number of cache misses.

2. *Fully associative*: a memory block in the main memory can be placed anywhere in the cache.

   A fully associative memory-to-cache mapping scheme provides a better performance. Because any memory block in the main memory can be stored at any cache block, the number of cache miss is lower. The disadvantage of the memory-to-cache mapping scheme is its complexity. If we want to determine a memory block in the main memory is in the cache or not, we need to check all present memory blocks in the cache. In practice, it requires a large number of comparators that increase the complexity and cost of implementing large caches. Therefore, this type of cache is usually only used for small caches, typically less than 4KiB

3. *Set associative*: a memory block in the main memory can be placed in a set of cache blocks. The cache is called *n-way set-associative* cache. The cache is organized in *ways*; the most common is 2, 4 and 8. In fact, we can consider the *direct mapped* cache as a 1-way set associative cache.

   A set associative cache is a combination of direct mapped cache and fully associative cache. It has the middle levels of advantages and disadvantage of the two memory-to-cache mapping schemes.

When the set for a block is full and a cache miss occurs, one of the blocks must be chosen to be replaced. It should be the block that will not be used in the near future. There are various algorithms for the replacement policy:

- *Random*: a block is randomly chosen.

- *Least Recently Used - LRU*: the least used block is replaced, the cache access in this case is logged.

- *Other*: FIFO, LFU, etc.

2.2.3 *Cache operations*

We present two operations regarding memory block in the cache memory: reading and writing.

Reading

A block can be identified in the cache or not based on two information: a *valid-* or *invalid-flag* and a *tag* for each block. When the computer system starts, the cache memory is flushed and all blocks are marked as invalid. The flag becomes valid when the data is written into the cache set. The data access to the cache is done in the following order:

- The set field is used to find the set.

- All valid tag fields in the set are compared to the tag field of the address. If the comparison is equal for one tag field, we get a hit. If it is not, we get a miss and the correct block must be loaded from the lower level memory.

- The word field is used to find the position of the word in the block.

Cache misses can occur for three reasons [48]:

- Compulsory: the line is not in the cache since the associated blocks are empty.

- Conflict: the line is not in the cache and all blocks associated to the set are being used.

- Capacity: the cache memory is full.

Writing

There are two policies for writing on the hit and two policies for writing on the miss. When a cache hit occurs, writing can be done in two different manners:

- *Write-through*: the writing on the cache is also made to the lower memory level.

- *Write-back*: the writing is only done on the cache, writing on the lower memory is done when the block is replaced.

There are also two strategies for writing on the miss in write-through policy:

- *Write allocate*: the block is written in the lower memory and then loaded into the cache.

- *No-write allocate*: the block is only modified in the lower memory.

2.3 CACHE PROBLEMS IN RTES

In this section, we detail the problems with cache memory that are related to WCET and scheduling analysis of RTES in preemptive scheduling context. They come from two behaviors defined as follows.

Definition 55 (Intrinsic (inter-task) cache behavior [72, 12]). *Intrinsic behavior depends on task internal design and execution path and is independent of the execution environment.*

Two functions or data areas in the task may compete for the same cache space and increasing the cache size and/or associativity can reduce these effects.

The interference created by intrinsic behavior is named *intrinsic interference*. Intrinsic interference is related to WCET computation in non-preemptive scheduling context. Static analysis of program code can reliably predict the guaranteed minimal hit count and maximal miss count in order to compute the WCET. Prediction of single task execution time is subject for timing analysis. This thesis focuses on the second behavior that creates problems related to scheduling analysis:

Definition 56 (Extrinsic (intra-task) cache behavior [72, 12]). *Extrinsic cache behavior depends on the environment and the others task intrinsic cache behavior. In case of preemption, the cache contents of a (preempted) task could be displaced by the new running (preempting) task.*

When a task is preempted, memory blocks belonging to this task could possibly be removed from the cache. Once this task resumes, previously removed memory blocks have to be reloaded. Thus, a new preemption cost named *Cache Related Preemption Delay* (CRPD) is introduced:

Definition 57 (Cache related preemption delay (CRPD) [12]). CRPD *is the delay added to the execution time of the preempted task because it has to reload cache blocks evicted by the preemption.*

To clearly present the problem of CRPD, we compare it with context switch overhead (CSH). As we introduced in section 1.2, context switch makes multitasking possible by allowing the processor to switch from one task to another. However, it comes with an unavoidable overhead.

Definition 58 (Context switch overhead (CSH) [56]). *Context switch overhead is the cost of performing the following activities: (1) suspending a task, (2) storing the progress of this task, (3) electing a new running task and later (4) restoring the state of the preempted task.*

Experiment result in [56] has shown that CSH is small comparing to task WCET and fairly constant, ranging from 4.2 μs to 8.7 μs. Because of this reason,

in classical scheduling analysis, CSH is usually upper-bounded and included in task WCET.

There are two problems with CRPD:

- The first problem is that CRPD can be significantly larger than CSH. Result of the same experiment in [56] has shown that the addition preemption cost introduced by CRPD can be up to to 195 μs, which is 22 times larger than CSH. Another experiment result in [11] has shown that CSH varies between 5-10 μs while CRPD varies between 1-10000 μs depending on the cache usage and system load. In addition, an analysis in [65] has shown that CRPD can present up to 44% of task WCET.

- The second problem is that CRPD depends on the preempting task, the preempted task and also the point of preemption. Thus, it is not a constant and cannot be upper-bounded and included in task WCET without introducing a heavy pessimistic assumption.

We provide several simple examples in order to clearly illustrate the effect of CRPD and preemption.

A task τ_i experiences the effect of CRPD if there is an increase in the response time of τ_i due to CRPD. The CRPD does not only come from higher priority tasks preempting τ_i but it also comes from higher priority tasks preempting each others.

A task may experience the effect of CRPD in two cases presented below. For each case, a scheduling of a task set is given as an example.

Direct Preemption

A task could experience CRPD when it is directly preempted by a higher priority task. As shown in Fig. 5, τ_2 experiences 2 unit of time of CRPD when it is preempted by τ_1.

Figure 5: Direct Preemption

In Figure 5, the CRPD is represented as a delay added to the remaining capacity of task after the preemption. It is a simplified and pessimistic representation of CRPD because in practice, not all memory blocks are required to be reloaded into the cache at once. However, to the best of our knowledge, information about which memory blocks are required at an execution point in time of a task is difficult to obtain. Thus, we must make this pessimistic assumption that the preemp-

tion will result in CRPD added directly to the remaining capacity of task after the preemption..

Nested Preemption

A task experiences the effect of CRPD when an intermediate higher priority task is preempted. In Fig. 6, we have τ_2 experiences 2 unit of CRPD when preempted by τ_1. Because τ_2 preempted τ_3 previously, an increase in the response time of τ_2 leads to an increase in the response time of τ_3. We can say that τ_3 indirectly experiences the effect of CRPD when τ_1 preempts τ_2. In addition, the CRPD experienced by τ_3 must be computed by taking into account both τ_1 and τ_2.
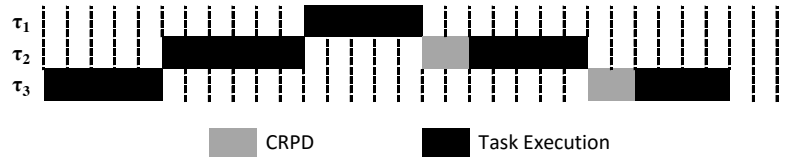
Figure 6: Nested Preemption

Methods of computing an upper-bound CRPD is detailed in the next section.

2.4 CRPD COMPUTATION APPROACHES

In this section, we explain how preemption cost and CRPD are computed. As presented in the previous section, the additional context switch overhead can be upper-bounded by a constant and included in the WCET [4]. The additional execution time due to preemption is mainly caused by cache eviction. Thus, CRPD can be used to refer to the preemption cost. CRPD is bounded by:

$$\gamma = g \cdot \text{BRT} \tag{9}$$

where g is an upper bound on the number of cache block reloads due to preemption and BRT is an upper-bound on the time necessary to reload a memory block in the cache (block reload time).

In [23, 52, 72], the authors presented five different approaches to compute g after a preemption.

1. g is equal to the number of cache blocks. In this case, CRPD is equal to the time to refill the entire cache.

2. g is equal to the number of cache blocks used by the preempting task.

3. g is equal to the number of cache blocks used by the preempted task.

4. g is equal to the number of intersection cache blocks between the pre-empted task and the preempting task.

5. g is equal to the number of cache blocks that are useful to the preempted task, which are named useful cache blocks and introduced later in the section.

All of the approaches above are based on the assumption that all or a set of cache blocks that have been replaced by the preempting task has to be loaded when the preempted task resumes execution.

2.4.1 *Evicting Cache Block*

The worst-case impact of a preempting task is given by the number of cache blocks that the task may evict during its execution. Busquet et al.[23] introduce the concept of evicting cache block (*ECB*):

Definition 59 (Evicting Cache Block). *A memory block of the preempting task is called an evicting cache block, if it is accessed during the execution of the preempting task.*

The notation ECB_j is used to present the set of ECBs of a task τ_j. In this case, the upper-bound CRPD can be computed by:

$$\gamma = \ BRT \cdot |ECB_j| \tag{10}$$

This preemption cost presents the worst-case effect of task τ_j on any arbitrary lower priority tasks, independent of such a task's actual cache behavior.

2.4.2 *Useful Cache Block*

To analyze the effect of preemption on a preempted task, Lee et al. [52] introduced the concept of useful memory block and useful cache block (*UCB*):

Definition 60 (Useful Memory Block [52]). *A memory block m is called a useful memory block at program point P, if m may be cached at P and m may be reused at program point P' after P that may be reached from P without eviction of m on this path when tasks execute non-preemptively.*

Definition 61 (Useful Cache Block (UCB) [52]). *A cache block c that holds a useful memory block m is called useful cache block.*

Let us take an example with a direct mapped cache with 4 cache blocks: 0,1,2,3. At time t, the mapping is:

Cache block:	c0	c1	c2	c3
Memory block	m0	m5	m6	m3

The next memory access sequence is m4 -> **m5** -> **m6** -> m7. then, the cache mapping is:

Cache block:	c0	c1	c2	c3
Memory block	m4	**m5**	**m6**	m7

We can see that m5 and m6 are reused while they are still in the cache, as a result, c1 and c2 are useful cache blocks.

For a given task, the number of UCBs at each execution point can be statically analyzed by applying a data flow analysis technique over the control flow graph of this task (CFG). For each execution point, we use one array to store the memory blocks that are reachable (reaching memory blocks – RMB) and another stores live memory blocks (LMB). The amount of useful cache blocks at each execution point can be determined with an iterative method. The number of UCB at program point P gives an upper bound on the number of additional reloads due to a preemption at P. The maximum possible preemption cost for a task is determined by the program point with the highest number of UCBs. The notation UCB_i is used to present the set of UCBs of a task τ_i. The CRPD when a task τ_i is preempted can be computed by:

$$\gamma = \text{BRT} \cdot |UCB_i| \tag{11}$$

In [78], the authors exploits the fact that for the m-th preemption, only the m-th highest number of UCBs has to be considered. However, as shown in [4] and [14], a significant reduction typically only occurs at a high number of preemptions. Thus, we only consider the program point with highest number of UCBs.

The work in [23] and [52] concerning UCB and ECB have established a simple *cache access profile computation* method to be used in CRPD analysis. The term cache access profile is defined as follows.

Definition 62 (Cache Access Profile). *A cache access profile contains information that gives details about the cache usage of a task.*

We now established the computation of an exact CRPD based on the notion of UCB and ECB. First, we consider the most simple case in which there is a preemption between only two jobs of the higher priority task τ_j and the lower piority task τ_i. Let $\gamma_{i,j}$ denotes the CRPD between those tasks. Then, $\gamma_{i,j}$ is computed by:

$$\gamma_{i,j} = \text{BRT} \cdot |UCB_i \cap ECB_j| \tag{12}$$

Second, we consider the case of a nested preemption in which τ_j preempts several tasks. Let Θ_j denotes the set of tasks that are preempted by τ_j. Let $\gamma_{\Theta_j,j}$

denotes the CRPD when τ_j preempts lower priority tasks in Θ_j. Then, the CRPD can be computed by:

$$\gamma_{\Theta_j,j} = \text{BRT} \cdot \left| \left(\bigcup_{\forall \tau_i \in \Theta_j} \text{UCB}_i \right) \cap \text{ECB}_j \right| \tag{13}$$

Furthermore, we can have an observation that previous preemption between tasks in Θ_j can lead to UCB eviction before τ_j preempts. As a result, the set of UCB in the cache of a task τ_i may be a subset of UCB_i. Let UCB'_i denotes the set of UCBs in the cache of task $\tau_i \in \Theta_j$. Then, a more precise computation of $\gamma_{\Theta_j,j}$ can be given by:

$$\gamma_{\Theta_j,j} = \text{BRT} \cdot \left| \left(\bigcup_{\forall \tau_i \in \Theta_j} \text{UCB}'_i \right) \cap \text{ECB}_j \right| \tag{14}$$

2.5 CRPD ANALYSIS FOR FPP SCHEDULING

In this section, we present existing research which has been made to account for CRPD in scheduling analysis. It is divided into three subjects:

- CRPD analysis for WCRT: extensions that have been made to the WCRT computation equation proposed by Joseph and Pandya [51] to take into account CRPD.

- Limiting CRPD: approaches that can be used to limit CRPD by either eliminating CRPD, reducing CRPD of each preemption or lowering the number of preemptions.

- CRPD analysis for scheduling simulation: approaches used in order to take into account CRPD in scheduling simulation.

2.5.1 *CRPD analysis for WCRT*

This section presents the extensions that have been made in order to take into account the effect of CRPD in WCRT computation for FPP scheduling.

In FPP scheduling context, the WCRT R_i of a task τ_i can be computed and compared against the deadline using the following equation [51]:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{15}$$

To take into account the CRPD, the term $\gamma_{i,j}$ was introduced by [23]. In this case, $\gamma_{i,j}$ refers to the total cost of preemption due to each job of higher priority

task τ_j ($\tau_j \in hp(i)$) executing within the response time of task τ_i. Then, the worst case response time of task τ_i can be computed by:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot (C_j + \gamma_{i,j}) \tag{16}$$

The precise computation of $\gamma_{i,j}$ depends on the approach used. Next, a summary about $\gamma_{i,j}$ computation approaches is provided. A list of approaches is given below. The names of the first three approaches are not given by their authors but are based on the use of UCB and ECB in $\gamma_{i,j}$ computation. They are:

1. ECB-Only by Busquets et al. [23]

2. UCB-Only by Lee et al. [52]

3. UCB-Union by Tan and Mooney [79]

4. ECB-Union by Altmeyer et al. [4]

ECB-Only

This approach focuses on computing the worst-case effect of task τ_j preempting task τ_i. Busquets et al. [23] presented ECB-Only approach which takes into account the effect from the preempting task τ_j. It assumes that all cache blocks evicted by task τ_j will have to be reloaded without taking into account the UCBs of the preempted task τ_i:

$$\gamma_{i,j}^{ecb} = BRT \cdot |ECB_j| \tag{17}$$

UCB-Only

Lee et al. [52] presented UCB-Only approach which takes into account the effect from the preempting task τ_i It assumes that all cache blocks that are useful to τ_i (UCB_i) will have to be reloaded regardless of the preempting task τ_j 's UCBs:

$$\gamma_{i,j}^{ucb} = BRT \cdot |UCB_i| \tag{18}$$

However, we have to consider the case of nested preemptions. The CRPD of τ_j preempting an intermediate priority task τ_k could be larger than $BRT \cdot |UCB_i|$. Nested preemptions are taken into account by computing the maximum set of UCBs of any intermediate priority task that can be preempted by τ_j. $\gamma_{i,j}^{ucb}$ is computed by:

$$\gamma_{i,j}^{ucb} = BRT \cdot \max_{\forall k \in aff(i,j)} \{|UCB_k|\} \tag{19}$$

Neither UCB-Only nor ECB-Only dominates each other. Theoretically, the set of UCBs of the preempted task can be larger than the set of ECBs of the preempting task and vice versa.

The disadvantage of the ECB-Only and UCB-Only approaches is that they only consider either the preempting tasks or the preempted tasks. However, simply using the intersection between UCB_i and ECB_j is optimistic in case of nested preemptions.

UCB-Union

Tan and Mooney [79] presented UCB-Union approach which takes into account both the preempted task and the preempting task. It assumes that the UCBs of intermediate priority tasks and UCBs of τ_i are evicted by the ECBs of τ_j. We define the set $aff(i,j) = hep(i) \cap lp(j)$ that represents the set of intermediate tasks that have lower priority than τ_j but higher priority than or equal to τ_i. CRPD is then computed by:

$$\gamma_{i,j}^{ucb-u} = BRT \cdot \left| \left(\bigcup_{\forall k \in aff(i,j)} UCB_k \right) \cap ECB_j \right| \tag{20}$$

This approach complements ECB-Only approach. As shown in [4], it is clear that UCB-Union dominates ECB-Only.

ECB-Union

Altmeyer et al. [3] presented ECB-Union approach which also takes into account both the preempted task and the preempting task. It assumes that the preempting task τ_j can have itself preempted by all of the tasks with a higher priority. A preemption by task τ_j may result in the eviction of $\bigcup_{h \in hp(j) \cup j} ECB_h$. The maximum number of evicted cache blocks is computed by the maximum set of UCBs of any intermediate priority task that can be preempted by τ_j and the set above.

$$\gamma_{i,j}^{ecb-u} = BRT \cdot \max_{\forall k \in aff(i,j)} \left\{ \left| UCB_k \cap \left(\bigcup_{h \in hp(j) \cup j} ECB_h \right) \right| \right\} \tag{21}$$

This approach complements UCB-Only approach. As shown in [4], it is clear that the ECB-Union approach dominates the UCB-Only approach. The ECB-Union and the UCB-Union approach are incomparable.

In the four approaches presented above, $\gamma_{i,j}$ is the CRPD due to a single preemption between the preempting task τ_j and the preempted task τ_i. This method of computing $\gamma_{i,j}$ has to take into account nested preemption by making pessimistic assumptions. All approaches assumed that if τ_j preempts τ_i, it also preempts each intermediate task $\tau_k \in aff(i,j)$. Thus, the number of times that $\tau_k \in aff(i,j)$ is preempted by τ_j is equal to the number of times that τ_i is preempted by τ_j. Theoretically, this can potentially be true if $T_k = T_i$ and $O_k = O_i$. In other cases, it is a pessimistic assumption.

Staschulat [77] introduced a different computation method and concept of $\gamma_{i,j}$. It does not refer to the cost of a single preemption, but instead to the total cost of all preemptions due to jobs of task τ_j executing within the response time of task τ_i.

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + \gamma_{i,j}^{sta} \right) \tag{22}$$

There are four approaches based on this equation.

1. Multiset Approach by Staschulat et al. [77]

2. UCB-Union Multiset by Altmeyer et al. [4]

3. ECB-Union Multiset by Altmeyer et al. [4]

4. Combined Multiset by Altmeyer et al. [4]

The detailed CRPD computation of these approaches is not presented in this thesis. In term of schedulability task set coverage, as shown in [4], we have the following results:

- ECB-Union Multiset approach dominates the ECB-Union approach.

- UCB-Union Multiset approach dominates the UCB-Union approach.

- The ECB-Union Multiset and the UCB-Union Multiset approaches are incomparable.

- Combined Multiset approach dominates both ECB-Union and UCB-Union approaches.

2.5.2 *Limiting CRPD*

There are certain techniques that can be used to limit CRPD by (1) eliminating CRPD, (2) reducing CRPD of each preemption or (3) lowering the number of preemptions. In term of eliminating CRPD, we have *cache partitioning*. In term of reducing CRPD of preemption, we have *effective preemption points* and *memory layout optimization*. In term of lowering the number of preemptions, we have *preemption threshold* approach and *deferred preemptions scheduling* approach.

Cache Partitioning

In this approach, the cache is split into several partitions. Tasks are allocated into partitions. Each task has its own cache space so that there is no cache extrinsic interference between tasks. Cache partitioning can be achieved by hardware by

using a cache that can be locked on a way-by-way basic or by software by using a compiler with specific support.

The advantage of this approach is that it increases the predictability and eliminates CRPD. Classical scheduling analysis methods can be applied to systems with cache as ones without cache. However, the disadvantage of this approach is that the cache space per task is reduced. As a result, cache intrinsic interference is increased and task's WCET is increased. In [1], Altmeyer et al. have showed that the decrease in CRPD and the increase predictability does not compensate for the increase in WCET.

We note that the number of partitions can be smaller than the number of tasks and more than two tasks can share one partition. In this case, CRPD is not totally eliminated but the extrinsic cache interference between tasks are limited.

Memory Layout Optimization

Memory layout optimization is achieved by static code positioning technique. This technique ensures that the program codes are laid in predefined locations. Unlike cache partitioning, static code positioning does not reduce the cache space per task.

In this thesis, we focus on memory layout optimization techniques that focus on reducing cache extrinsic interference. In [41], Gebhard and Altmeyer exploit the fact that different memory arrangements lead to different cache interferences. First, the authors proposed a cost function that computes the number of cache conflicts for a given task placement. The cost is proportional to the number of memory blocks belonging to the preempted task that reside in the same location in the cache memory as the memory blocks of the preempting task. It also takes into account the lifespan of blocks due to the replacement policy. Second, they proposed a method to adjust the starting position of tasks in a given task set such that the cost is globally minimized with regard to a given cache configuration.

In [58], Lunniss et al. proposed an approach to reduce the impact of CRPD by performing a memory layout optimization based on simulated annealing (SA). This approach compliments the work in [41] by taking into account the location of task UCBs. During each iteration of SA algorithm, changes are made to the layout of tasks in memory, and then mapped to their cache layout for evaluation. The authors have shown that a near optimal solution could be achieved with the algorithm.

Deferred Preemptions Scheduling

In [19], Burns presented the deferred preemption model. In this model, each job of task τ_i is modeled by a sequence of non-preemptive regions separated by a fixed preemption point. It allows a task to run for a period of time without being preempted up to a certain limit. An exact schedulability analysis for fixed

priority scheduling with deferred preemptions has been presented by Bril et al. [18].

Effective Preemption Points

In [14], Bertogna et al. extended the work in [22] and introduced the concept of potential preemption point (PPP) and effective preemption point (EPP). Each job of task τ_i is modeled by a sequence of N_i non-preemptive basic blocks. Preemption is allowed only at basic block boundaries, so each task has $N_i - 1$ PPP. Critical sections and conditional branches are assumed to be executed entirely within a basic block.

An algorithm is designed to identify a subset of PPPs that minimizes the overall CRPD but still preserve the schedulability. The PPPs in the subset is then referred to as EPPs. Then, other PPPs are disabled and preemption is allowed only at the EPPs.

The advantage of this approach is that because tasks can only be preempted at selected program points, we do not have to always consider the worst case. In addition, preemption cost at these points can be precisely computed. The limitation of this approach is that it can only be applied to programs which can be modeled as a sequential flow of basic blocks. In practice, typical applications are composed of many conditional branches and loops. It requires all loops and branches to be contained within one basic block thus limiting the applicability of the proposed approach.

In [66], Peng et al. have explored utilizing a combination of graph grammars and dynamic programming to handle the EPP selection problem for control flow graphs with conditional structures. The authors have showed that their approach has pseudo polynomial-time complexity and also proposed a near-optimal heuristic with lower complexity, memory requirement and computation time.

Preemption Threshold

Wang and Saksena [86] proposed preemption threshold to improve the schedulability and to reduce preemption overhead. This is a dual-priority system. A task is assigned one nominal priority level and one preemption threshold. Once a task is executed, its priority level raises to preemption threshold level. Thus, it cannot be preempted by higher priority tasks up to a certain priority. An exact schedulability analysis for FPP with preemption thresholds is also presented in [86].

Task	C_i	T_i	D_i	O_i	UCB_i	ECB_i	Π_i
τ_1	2	8	8	0	$\emptyset$	{1,2}	2
τ_2	5	12	8	0	{1,2}	{1,2}	1

Table 2: Synchronous task set with critical instant of task τ_2 is not at the synchronous release.

2.5.3 *CRPD analysis for scheduling simulation*

CRPD analysis for scheduling simulation is still an open subject of discussion and the are several problems to be addressed. The first problem is what computation model we should use that in order to take into account the effect of CRPD. There are several design choices were made in order to study the effect of CRPD. In [67], the authors consider a computation model with constant value of CRPD for each task when it is preempted. In [4], the authors consider a model in which CRPD is computed by the set of UCBs and ECBs of tasks. However, in these works, scheduling simulation with CRPD was not the focus and were used as an example to illustrate the effect of CRPD.

SimSo[28] is a scheduling simulation tool that supports cache sharing on multiprocessor systems. It takes into account the impact of caches through statistical models and also the direct overheads such as context switches and scheduling decisions. As stated in [30], SimSo used a fixed value for CRPD.

The second problem is regarding the feasibility interval when CRPD is taken into account. As far as we know, there is no existing study that takes into account this problem. Critical instant for a task is not identified when CRPD is taken into account. Regarding synchronous tasks set, as far as we know, the classical critical instant defined in [57] is not applicable when we consider the effect of CRPD. A simple example is provided in Table 2.

The scheduling of the task set in Table 2 in the interval $[0, 24)$ is provided in Figure 7. In this example, τ_2 does not experience the highest interference at the synchronous release. The job of τ_2 released at time 0, denoted $\tau_2[0]$, can meet its deadline but $\tau_2[12]$ cannot.

Regarding asynchronous task set, in [7], Audsley stated that there is not any critical instant.

Furthermore, comparing the CRPD obtained by scheduling simulation to the real execution on a hardware platform, is also an open problem. There is a lack of facility that supports observing and analyzing cache memory access on a hardware platform. A potential solution to this problem is using a non-intrusive hardware observer that supports run-time verification of RTES by monitoring the bus such as the on presented in [69].
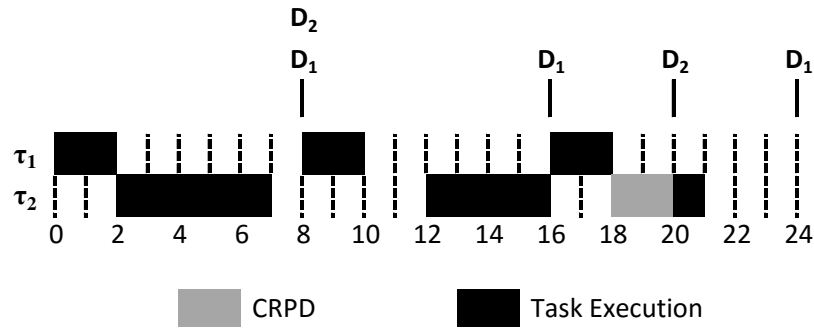
Figure 7: τ_2 does not experience the highest interference at the synchronous release,

2.6 CONCLUSION AND THESIS SUMMARY

We have presented a summary about cache memory, the definition of CRPD and the state of the art research in CRPD analysis for FPP scheduling. CRPD analysis for WCRT based on the notion of UCBs and ECBs are well developed. Later tests provide tighter bounds and results in finding more schedulable task sets. Several techniques are proposed in order to limit CRPD by eliminating CRPD, reducing CRPD of each preemption or lowering the number of preemptions.

In the existing work, the focus has been on verifying the system schedulability after task priorities are assigned. However, because CRPD computation depends on the preempting task and preempted tasks, priority ordering or priority assignment highly affects the result of CRPD computation and system schedulability. In Chapter 3, we present an approach to take into account CRPD when assigning priorities to tasks.

Scheduling simulation based on the concept of UCBs and ECBs still remains an open subject. Most of the simulation were done only at experimentation level. We lack a concrete result about how scheduling simulation with CRPD should be performed. In addition, the interval of time needed to perform the simulation of RTES with cache memory is still an open question. These issues greatly limit the use of scheduling simulation as a verification method. In Chapter 4, we propose a formalization of scheduling simulation with CRPD, investigate the problem of feasibility interval and evaluate the use of scheduling simulation as a verification method for RTES with cache memory.

Even though there are existing researches in this domain, there is a lack of scheduling simulation facilities that support RTES with cache memory. In Chapter 5, we address this problem by providing an implementation of our work and several CRPD analysis methods for FPP scheduling in Cheddar - an Open-Source scheduling analyzer [75].

Part II

CONTRIBUTION

Chapter 3

CRPD-AWARE PRIORITY ASSIGNMENT

Contents

CRPD is created by higher priority tasks preempting lower priority tasks and evicting their data in the cache. In FPP scheduling context, preemption is decided by the priority ordering. Therefore, CRPD that affects the WCRT of a task depends on the chosen priority assignment algorithm. However, as far as we know, there is no priority assignment algorithm that takes into account CRPD in the state of the art work. As a result, classical priority assignment algorithms are either not optimal or not applicable to RTES with cache memory. These problems are detailed in Section 3.2.

In this chapter, we present a *CRPD-aware priority assignment* (CPA) algorithm that assigns priority and evaluates the schedulability of a task set. For such a purpose, we propose five extensions to the original *Audsley's Optimal Priority Assignment* (OPA) algorithm [7] that have different degrees of pessimism, different complexities, and give different results in terms of schedulable task sets coverage. Exhaustive experimentations are achieved to evaluate the proposed approaches in terms of complexity and efficiency. The result shows that our approach provides a mean to guarantee the schedulability of the RTES while taking into account CRPD. This approach also discovers priority orderings that make a task set schedulable while it is not schedulable with classical priority assignment algorithm when CRPD is taken into account.

The rest of the chapter is organized as follows. Section 3.1 presents system model and assumptions taken for this work. Section 3.2 discusses about the limitation of classical fixed priority assignment algorithms and details the problem with OPA. Section 3.3 provides an overview of our approach. In Section 3.4, detailed approach and algorithms are presented. Section 3.5 discusses the complexity of the proposed solutions. In Section 3.6, an evaluation of our approach in terms of efficiency and complexity is given. Section 3.7 concludes the chapter.

3.1 SYSTEM MODEL AND ASSUMPTIONS

In this section, we present our system model and assumptions taken.

- We assume an uniprocessor system with one level of direct-mapped instruction cache that consists of n independent stricly periodic tasks $(\tau_1, \tau_2, ..., \tau_n)$ scheduled by a FPP scheduler.

- A task is defined by a quintuple: $(C_i, T_i, D_i, O_i, \Pi_i)$. The five elements are respectively the capacity, the period, the deadline, the offset and the priority of the task τ_i. The capacity of a task is smaller than its deadline $(C_i \leqslant D_i)$ and the deadline of a task is smaller than or equal to its period $(D_i \leqslant T_i)$.

- $hp(i)$ (respectively $lp(i)$) is the set of tasks with higher (respectively lower) priority than task τ_i.

- $hep(i)$ (respectively $lep(i)$) is the set of tasks with higher (respectively lower) or equal priority to task τ_i.

- Tasks can be either synchronous or asynchronous.

- UCB_i and ECB_i are respectively the set of UCBs and the set of ECBs of task τ_i.

We use the term *complete priority assignment* to mention a system in which each task is assigned a priority level.

3.2 LIMITATION OF CLASSICAL FIXED PRIORITY ASSIGNMENT ALGORITHMS

In this section, we discuss about the limitation of classical priority assignments including RM, DM and OPA.

Task	C_i	T_i	D_i	O_i	UCB_i	ECB_i	Π_i
τ_1	3	12	12	0	$\emptyset$	$\{1,2\}$	3
τ_2	8	24	24	8	$\{1,2\}$	$\{1,2,3,4\}$	2
τ_3	9	24	24	0	$\emptyset$	$\{3,4\}$	1

Table 3: Task set example
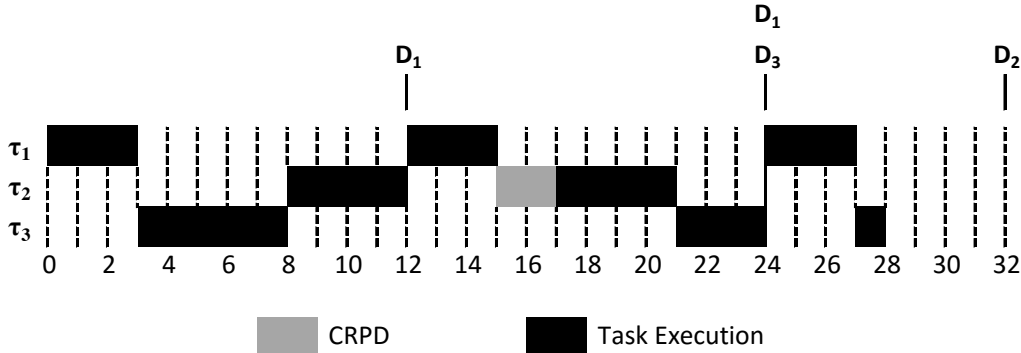
Figure 8: Priority ordering by RM: $\Pi_1 = 3, \Pi_2 = 2, \Pi_3 = 1$. Task τ_3 missed its deadline at time $t = 24$.

3.2.1 *Limitation of RM and DM*

The limitation of RM and DM priority assignment algorithms is that they are not optimal when CRPD is taken into account. The two priority assignment algorithms take into account the period and the deadline parameter respectively. These algorithms assign priorities to tasks in the sense that a task with a tighter timing constraint is assigned a higher priority level. However, in RTES with cache memory, a task that has a tight timing constraint but experiences potentially low CRPD because of low cache usage could be easier to be schedulable at a low priority level. By contrast, a task that has loose timing constraint but experience potentially high CRPD could be more difficult to be schedulable at a low priority level.

We give an example on how taking into account CRPD can change schedulability conditions and improve the schedulability with a task set in Table 3. We assume that BRT = 1 unit of time.

Let us analyze the example of RM where priorities are assigned according to the periods of tasks. In case of equal periods between two tasks, the task with lower index is assigned higher priority level. It results in a non-schedulable task set. The scheduling is displayed in Figure 8. As we can see, task τ_3 missed its deadline at time $t = 24$.
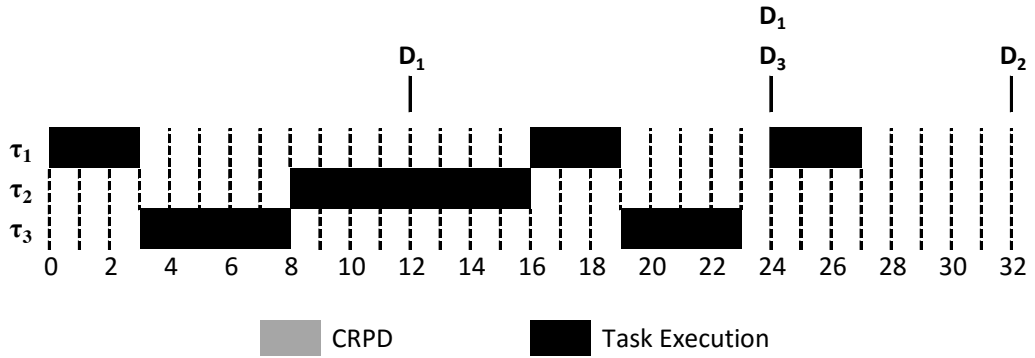
Figure 9: Priority ordering 1: $\Pi_1 = 2, \Pi_2 = 3, \Pi_3 = 1$. All tasks are schedulable
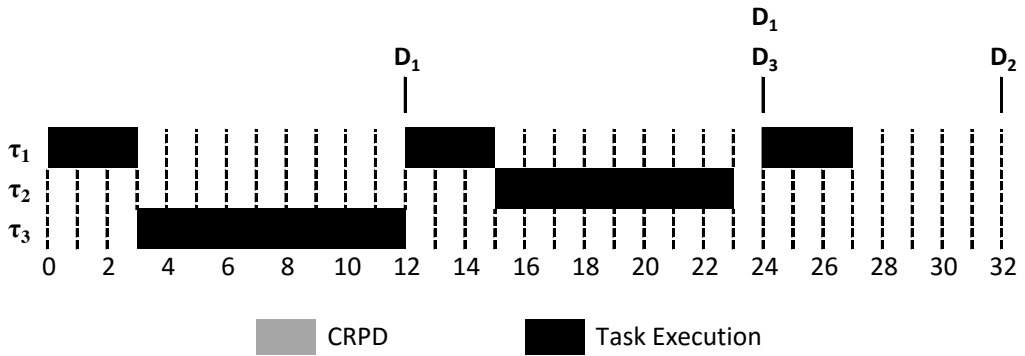
Figure 10: Priority ordering 2: $\Pi_1 = 3, \Pi_2 = 1, \Pi_3 = 2$. All tasks are schedulable

In this example, contrary to a RM priority assignment, there are two priority orderings that can make the task set schedulable. The first priority ordering is $\Pi_1 = 2, \Pi_2 = 3, \Pi_3 = 1$. The scheduling with this priority order is shown in Figure 9. In this priority ordering, CRPD is eliminated because τ_1 cannot preempt τ_2.

The second priority ordering is $\Pi_1 = 3, \Pi_2 = 1, \Pi_3 = 2$. The scheduling with this priority order is shown in Figure 10. In this priority ordering, the CRPD is eliminated because τ_2 does not start execute at time t = 8 and then it is not preempted by τ_1.

To conclude, one needs to take into account the CRPD early in the system model in order to verify the feasibility of tasks and adapt the priority assignment if it is necessary. Our observation is that the effect of CRPD cannot be evaluated by taking into account task attributes such as period and deadline. It needs to be evaluated based on relationship between tasks. As a result, priority assignment algorithms, in which priority ordering is only based on task static attributes, cannot be optimal with CRPD. In [67], the problem of finding the optimal priority assignment with CRPD has been proved to be NP-Hard. In addition, the optimal scheduling can only be achieved by offline scheduling [67].

3.2.2 *Limitation of* OPA

The limitation of OPA is that the original priority assignment algorithm is not applicable to RTES with cache memory. OPA assigns a priority level to a task and verifies its schedulability at the same time by using a feasibility test; however, the feasibility test used in the original algorithm cannot guarantee that a system is schedulable when CRPD is taken into account. For example, we performed an experiment in Section 3.6 to show that at a high processor utilization, there is a significant gap between the number of task sets assumed to be schedulable by OPA and the number of schedulable task sets when considering CRPD. Indeed, without taking CRPD into account, OPA failed to identify a high number of unschedulable task sets. For instance, OPA identified 600 schedulable task sets while only 100 are schedulable for a 90% processor utilization.

In the original work of Audsley [7] presented in Section A, the algorithm consist of four steps. At the start, all priority levels are not assigned:

- Step 1: The algorithm assigns the unassigned lowest priority level to an unassigned priority task τ_i.

- Step 2: A feasibility test is used to verify if τ_i is schedulable at the priority level or not.

- Step 3: If τ_i is not schedulable at the priority level, the algorithm chooses a different task in the set of un assigned priority tasks and comes back to Step 1.

- Step 4: If τ_i is schedulable at the priority level, τ_i is removed from the set of unassigned priority tasks. The algorithm moves to the next higher priority level and comes back to Step 1.

The feasibility test in Step 2 was designed with two assumptions. First, the response time of a task is not affected by the priority ordering of higher priority tasks. Second, preemption cost is assumed to be zero. The two properties are not true when CRPD is taken into account. As a result, we need to design an appropriate feasibility test. This test must be able to verify the feasibility of a task under a given priority level while the complete priority assignment of higher priority tasks is not achieved.

The problem lies in the fact that the CRPD, which affects a task's WCRT, can only be exactly computed when task priorities are completely assigned. It is not possible to apply the WCRT analysis with CRPD presented in Section 2.5.1 to OPA because of the computation of CRPD.

We remind that the computed upper-bound CRPD when a higher priority task τ_j preempts a lower priority τ_i, denoted $\gamma_{i,j}$, consist of two parts. First, $\gamma_{i,j}$ includes the CRPD of τ_j evicting UCBs of τ_i. Second, $\gamma_{i,j}$ includes the CRPD

of τ_j evicting UCBs of intermediate tasks $\tau_k \in aff(i,j) = hp(i) \cap lp(j)$. In order to compute $aff(i,j)$ for each task τ_j, the priorities of tasks in the set $hp(i)$ must be completely assigned. This is the main challenge of applying OPA to a system model with CRPD. The problem is that a complete priority assignment is not achieved in the feasibility testing phase. In step 1 of OPA, a task τ_i is assumed to have the lowest priority so the set $hp(i)$ can be computed. Other tasks have higher priorities than τ_i, however, specific priority assignments of those tasks are not set. As a result, the set $lp(j)$ cannot be computed.

3.3 PROBLEM FORMULATION AND OVERVIEW OF THE APPROACH

In this section, we present our approach and discuss the raised issues. We extend the feasibility test in step 2 of OPA in order to take into account the CRPD. We proceed by explaining the feasibility condition in [7] and how it is extended to take into account CRPD. Then, we formulate the problem regarding the extension.

3.3.1 *Feasibility condition of OPA*

Regarding the feasibility condition in [7], a task is schedulable if all its jobs released during the feasibility interval can meet their deadlines. Assume a job $\tau_i[t]$, $t = O_i + k \cdot T_i, k \in \mathbb{N}$, requires C_i units of computation time and must complete before D_i. $\tau_i[t]$ experiences interferences from higher priority tasks during the interval $[t, t + D_i)$. These interferences are denoted as I_i^t and defined as follows:

Definition 63 (Interference [7]). *The interference that is suffered by $\tau_i[t]$ due to jobs of higher priority tasks wishing to execute during the release of $\tau_i[t]$ is defined as I_i^t.*

Then, $\tau_i[t]$ is feasible if and only if the following condition is satisfied [7]:

$$C_i + I_i^t \leqslant D_i \tag{23}$$

A task τ_i is schedulable at a given priority level if and only if all jobs of τ_i released in the feasibility interval can meet their deadlines. In other words, Equation 23 is satisfied for all jobs of τ_i released in the feasibility interval. Algorithm verifying the schedulability of a task τ_i at a given priority level [7] is presented in Listing 1.

In this chapter, we assume that the feasibility interval is known. During all our experiments, we have observed a cyclic behavior of scheduling simulation with CRPD after the feasibility interval proposed in [7, 43]. Feasibility interval is discussed in Chapter 4.

```
1   for each τᵢ[t], t ∈ feasiblity interval of τᵢ
2     Iᵢᵗ = Rᵢᵗ + Kᵢᵗ.
3     if Cᵢ + Iᵢᵗ > Dᵢ then
4       schedulable ← FALSE
5     end if
6   end loop
```

Listing 1: Algorithm veryfing the schedulability of τ_i at a given priority level [7].

In this algorithm, I_i^t is made up of two parts.

1. The first part is the interference from jobs of higher priority tasks that have been released before t, did not complete at t and have deadlines after t. It is called remaining interference [7] and denoted as R_i^t. A naive approach to compute remaining interference is assessing jobs released in the interval $[0, t)$. In [7], the author provided a better approach by taking into account jobs released in the period $[t - D_i, t)$ plus the outstanding computation of the created interference [7] of the previous period $[t - T_i, t - T_i + D_i)$.

2. The second part is the interference from jobs of higher priority tasks released in $[t, t + D_i)$. This interference is called created interference [7] and denoted as K_i^t.

The computation of K_i^t and R_i^t was defined in the work of Audsley [7]. First, we explain the computation of K_i^t. Then, the computation of R_i^t is presented. In addition, at the end of each sections, we detail in which step CRPD interference is taken into account.

A *Computation of K_i^t*

The created interference K_i^t is due to jobs of higher priority tasks released in the interval $[t, t + D_i)$ to $\tau_i[t]$. To compute K_i^t, a set η is defined, with one element $(\tau_j[t_j])$ representing a release of $\tau_j \in hp(i)$ at time t_j in the interval $[t, t + D_i)$ (In other words, we have: $t_j \in [t, t + D_i)$).

The set is ordered by the release time t_j. Each element is used to step along the interval $[t, t + D_i)$ to calculate the demand of higher priority tasks. The algorithm that illustrates the approach is presented below:

```
1   CreatedInterference(η, Rᵢᵗ)
2   begin
3     next_free = Rᵢᵗ + t
4     Kᵢᵗ = 0
5     total_demand = Rᵢᵗ
6     for (τⱼ[tⱼ]) in α
```

```
7        total_demand = total_demand + Cj
8        if (next_free < tj) then
9            next_free = tj
10       end if
11       Kᵢᵗ = Kᵢᵗ + min (t+Dᵢ - next_free, Cj)
12       next_free = min (t+Dᵢ, next_free + Cj)
13    end for
14    return Kᵢᵗ
15 end
```

In the algorithms, there are three variables that are computed:

- next_free: the time instant at which all jobs higher priority tasks released before are completed and the processor is free (not occupied).

- total_demand: total execution demand of higher priority tasks released in the interval $[t, t + D_i)$

- K_i^t: total created interference of higher priority tasks released in the interval $[t, t + D_i)$

We use an example to illustrate the different between total_demand and K_i^t. We consider the task set presented in Table 3. We consider the task τ_2 at the lowest priority level. Figure 11 illustrate the interference from higher priority task to $\tau_2[8]$. We have $\alpha = \{\tau_1[12], \tau_1[24], \tau_3[24]\}$.

The total execution demand can be computed simply by taking into account the capacity of jobs in α. In this example $\texttt{total\_demand} = C_1 + C_1 + C_3 = 4 + 4 + 9 = 17$. However, we have $K_2^8 = 11$ because we only keep the execution demand that effects $\tau_2[8]$.
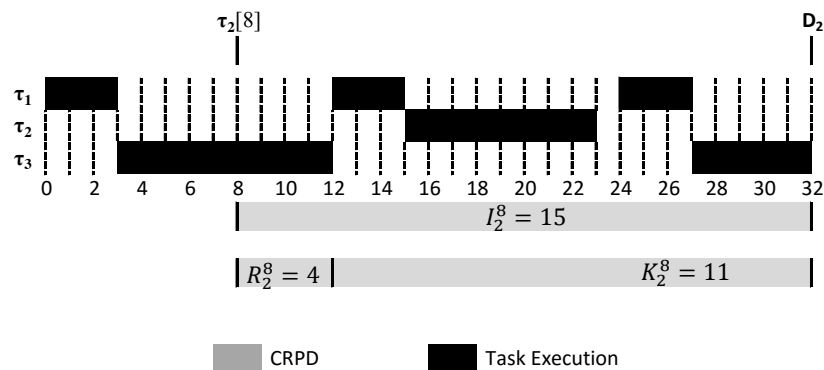
Figure 11: Interference from higher priority tasks to $\tau_2[8]$.

The computation of the CRPD created by the job of τ_j, denoted γ, is added between line 6 and line 7. Then, C_j in the algorithm is replaced by $C_j + \gamma$.

B   *Computation of R_i^t*

The remaining interference R_i^t to $\tau_i[t]$ is due to jobs of higher priority tasks that have not completed their executions at t. The easiest method to compute R_i^t is to construct and examine a schedule for the interval $[0, t)$. As presented in [7], this method is inefficient.

Another approach can be derived by nothing that when computing K_i^t, we can also compute the outstanding execution demand of higher priority tasks released in the interval $[t, t + D_i)$, denoted L_i^t. For the next release of τ_i at time $t + T_i$, we only need to take into account the set of higher priority tasks released in $[t + D_i, t + T_i)$, denoted β set, and L_i^t.

The algorithm that illustrates the approach is presented below:

```
1  RemainingInterference(β, Lᵢᵗ)
2  begin
3    time = t - Tᵢ + Dᵢ
4    Rᵢᵗ = Lᵢᵗ
5    for (τⱼ[tⱼ]) in β
6      if(tⱼ > time + Rᵢᵗ) then
7        Rᵢᵗ = 0
8      end if
9      time = tⱼ
10     Rᵢᵗ = Rᵢᵗ + Cⱼ
11   end for
12   Rᵢᵗ = Rᵢᵗ - (β'Last.tⱼ) --Release time of the last job in β
13   if(Rᵢᵗ < 0) then
14     Rᵢᵗ = 0
15   end if
16   return Rᵢᵗ
17 end
```

The computation of the CRPD created by the job of τ_j is added between line 9 and line 10. Then, C_j in the algorithm is replaced by $C_j + \gamma$.

To sum up, I_i^t is computed by taking into account the interference from jobs of higher priority tasks. In many cases, I_i^t can be only made up of either remaining interference, R_i^t, or created interference, K_i^t.

3.3.2   *Extending the feasibility condition with CRPD*

Now, we analyze the interference from one job of higher priority task that made up either R_i^t or K_i^t. The interference from a job of higher priority task τ_j is made up of its capacity C_j. In systems with cache, we have to take into account the

CRPD created by this job. Then, the interference from a job of task τ_j now consists of two parts:

- The first part is the capacity of task τ_j, denoted as computational interference.

- The second part is the CRPD due to task τ_j preempting task lower priority tasks including τ_i and intermediate priority tasks τ_k, $\tau_k \in hp(i) \cap lp(j)$, denoted CRPD interference.

We analyze the interference created by the job of task τ_1 and τ_2 to the job of task τ_3, the scheduling is depicted in Figure 8. The first part the capacities of task τ_1 and τ_2 in the interval $[0, 24)$. The second part is the CRPD due to task τ_1 preempting task τ_2.

I_i^t is made up of computational requirement and CRPD interference from jobs of higher priority tasks. In [7], the algorithm that accounts for the computational requirement has been established. This algorithm evaluates each job individually. For a job, its interference to $\tau_i[t]$ is computed by taking into account its release time and capacity. In order to take into account CRPD, we need to extend this algorithm to compute also the CRPD interference created by a job. We proceed by explaining how CRPD interference is computed.

In order to compute CRPD interference, one needs to evaluate: (1) the number of preemptions and (2) the CRPD for each preemption.

Number of Preemptions

In OPA, when verifying the feasibility of a task at a given priority level, we only assumed other tasks have higher priority without a complete priority assignment. As a result, the occurrence of a preemption between jobs of those tasks is not identifiable. Thus, the exact computation of the number of preemptions in the interval $[t - T_i + D_i, t + D_i)$ poses a challenge and is an open issue.

In the task set example in Table 3, the priorities of task τ_2 and τ_3 affect the computation I_1^0. As we can see in Figure 12 and 13, there are two priority orderings that result in two different number of preemptions and CRPD. So, we need to find a solution to compute the number of preemptions with the previous constraint in mind.

CRPD

Assume that the sets of UCBs and ECBs of each task are preliminary computed, the problem now is that we can only compute the CRPD if the preempting task and preempted tasks are identified.

In the next section, we propose three different approaches to solve those two problems regarding number of preemptions and preemption cost.
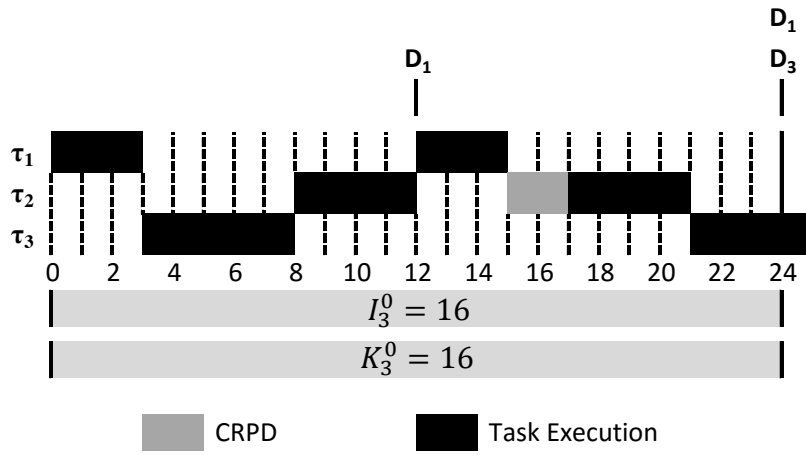
Figure 12: Complete priority assignment of task τ_1 and τ_2 affects the computation of I_3^0.
$\Pi_1 = 3, \Pi_2 = 2, \Pi_3 = 1$
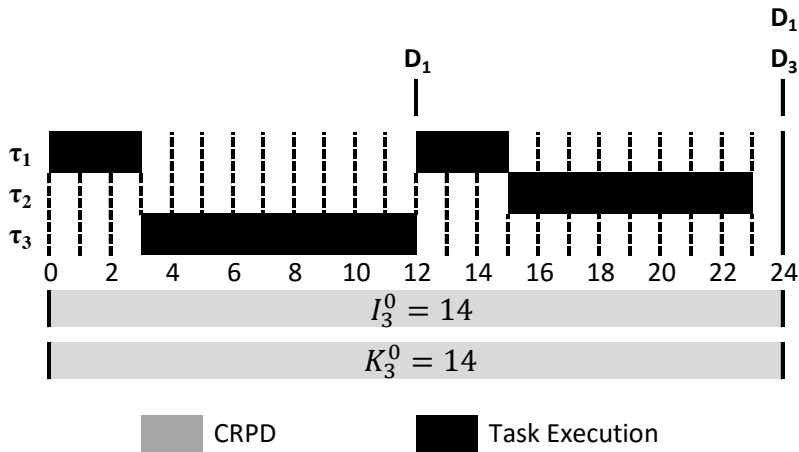
Figure 13: Complete priority assignment of task τ_1 and τ_2 affects the computation of I_3^0.
$\Pi_1 = 2, \Pi_2 = 3, \Pi_3 = 1$

3.4 CRPD INTERFERENCE COMPUTATION SOLUTIONS

In this section, we present four solutions to compute an upper-bound of CRPD interference. Each solution proposes a way to compute the number of preemption and CRPD. For each solution, we present the general idea and provide an example of interference computation in this section. The algorithms are provided in Appendix A.

Assuming a job of task τ_i is released at time t, the CRPD interference of I_i^t is now computed by evaluating the set of jobs composed of higher priority tasks τ_j released in the interval $[t - T_i + D_i, t + D_i)$. This set is called η, which is used in any CRPD computation we proposed later. In this set, the jobs are ordered by their release times. We use a set with ordered elements in order to be compliant with the presentation of the work in [7]. The presentation of η is as follows:

$$\eta = \{(\tau_j[t_j]) \mid \tau_j \in hp(i), t_j \in [t - T_i + D_i, t + D_i)\}$$

We define the following notation, which are used later to present our computation on η only in this section.

- $\eta[l]$: the lth element of the set η.

- C_l: the capacity of $\eta[l]$.

- t_l: the release time of $\eta[l]$.

- UCB_l: the set of UCBs of $\eta[l]$.

- ECB_l: the set of ECBs of $\eta[l]$.

Because jobs in η are ordered by their release time, we have $\forall \eta[l], t_l < t_{l+1}$

3.4.1 *CPA - ECB*

The first solution consists in adding the worst-case effect of CRPD to the capacity of all jobs in η. The CRPD analysis using only ECB method can be used for such purpose.

In this solution, the worst-case effect of a preemption is added directly to the capacity of jobs of higher priority tasks in η.

$$C_l' = C_l + BRT \cdot |ECB_l|, \forall \eta[l] \in \eta \tag{24}$$

In this solution, we take two pessimistic assumptions:

1. All activations of a task are considered to lead to preemptions, which results in CRPD. This answers the problem of number of preemption.

2. The CRPD is computed by the number of ECBs of the preempting task, which is an over-approximation as presented in Section 2.4. This answers the problem of CRPD.

By construction the CRPD Interference and number of preemptions computed by this solution is upper-bounded. The number of preemptions in practice is always lower than the number of jobs.

Example

We give an example of computing the interference and testing the feasibility of a task at a given priority level with the task set provided in Table 3.

Considering the job $\tau_3[0]$ at the lowest priority level, we need to check for jobs of higher priority tasks released in the interval $[0, 24)$ We have:

1 $\eta = \{\tau_1[0], \tau_2[8], \tau_1[12]\}$
2 $C_1' = C_1 + \text{BRT} \cdot |\text{ECB}_1| = 3 + 1 \cdot 2 = 5$
3 $C_2' = C_2 + \text{BRT} \cdot |\text{ECB}_2| = 8 + 1 \cdot 4 = 12$
4 $C_3' = C_3 + \text{BRT} \cdot |\text{ECB}_3| = 3 + 1 \cdot 2 = 5$

Applying the interference computation algorithm in [7], we have $I_3^0 = 22$. Given the capacity of τ_3 is 9 and the deadline of τ_3 is 24, we have $9 + 22 > 24$. We conclude that τ_3 is not schedulable at the lowest priority level.

The interference from higher priority tasks to $\tau_3[0]$ regarding CPA-ECB solution is depicted in Figure 14. In this figure, the execution of τ_1 and τ_2 are separated to improve the readability. It does not imply the priority levels of τ_1 and τ_2.
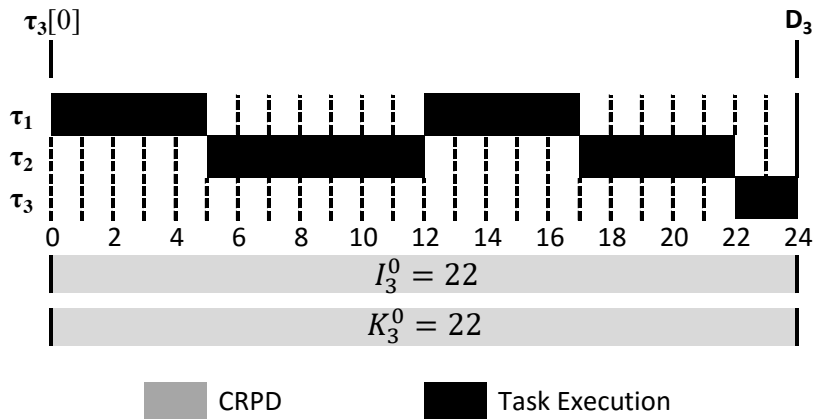
Figure 14: Interference from higher priority tasks to $\tau_3[0]$ regarding CPA-ECB.

Considering the job $\tau_2[8]$ at the lowest priority level, we need to check for jobs of higher priority tasks released in the interval $[0, 32)$, we have:

1 $\eta = \{\tau_1[0], \tau_3[0], \tau_1[12], \tau_1[24], \tau_3[24]\}$

2 $C_1' = C_1 + \text{BRT} \cdot |\text{ECB}_1| = 3 + 1 \cdot 2 = 5$

3 $C_2' = C_2 + \text{BRT} \cdot |\text{ECB}_2| = 9 + 1 \cdot 2 = 11$

4 $C_3' = C_3 + \text{BRT} \cdot |\text{ECB}_3| = 3 + 1 \cdot 2 = 5$

5 $C_4' = C_4 + \text{BRT} \cdot |\text{ECB}_4| = 3 + 1 \cdot 2 = 5$

6 $C_5' = C_5 + \text{BRT} \cdot |\text{ECB}_5| = 9 + 1 \cdot 2 = 11$

Applying the interference computation algorithm in [7], I_2^8 is computed by:

- The remaining capacity of $\tau_1[0]$ and $\tau_3[0]$ at time 8, which is 8. We can see that the total capacity of $\tau_1[0]$ and $\tau_3[0]$ is 16.

- Capacity and CRPD of $\tau_1[12], \tau_1[24]$ and $\tau_3[24]$ in the interval $[8,32)$, which is 13. We notice that I_2^8 does not include the capacity of $\tau_1[24]$ and $\tau_3[24]$ after time $t = 32$.

We have $I_2^8 = 21$. Given the capacity of τ_2 is 8 and $D_2 = 24$, we have $8 + 21 > 24$. We conclude that τ_2 is not schedulable at the lowest priority level.

The interference from higher priority tasks to $\tau_2[8]$ regarding CPA-ECB solution is depicted in Figure 15. In this figure, the execution of τ_1 and τ_3 are separated to improve the readability. It does not imply the priority levels of τ_1 and τ_3.

Figure 15: Interference from higher priority tasks to $\tau_3[0]$ regarding CPA-ECB.

Considering the job $\tau_1[0]$ at the lowest priority level. It is trivial to see that $\tau_1[0]$ is also not schedulable at the lowest priority level. Because there is no task feasible at the lowest priority level, the task set is concluded to be not schedulable.

From this example, we can see that CPA-ECB is pessimistic. The computed interference is significantly higher than the actual interference.

3.4.2 *CPA-PT and CPA-PT Simplified*

The second solution consists in finding all potential preemptions and in computing the upper-bound CRPD for each potential preemption. This upper-bound

CRPD is smaller than or equal to the number of ECB of the preempting task. This solution is less pessimistic than previous one on both parameters: number of preemptions and preemption cost.

When priority assignments of higher priority tasks are not set, there is no information to decide if a task may be preempted by another task or not. We describe this problem by using the defining *potential preemption*. We then assume that, a preemption may occur if the conditions of a potential preemption holds.

Definition 64 (Potential preemption). *A potential preemption amongst jobs of tasks with no complete priority assignment is a preemption that may occur when a job is released while other jobs did not complete their execution.*

In order to compute the CRPD interference upper-bound, we take two assumptions:

1. All potential preemptions occur.

2. A potential preemption occurs with the maximum number of preempted jobs and the maximum number of evicted UCBs.

Assume that a job $\eta[l]$ can potentially preempt several jobs represented by a set $\gamma_{\Theta_l, l}$. The CRPD can be computed by:

$$\gamma_{\Theta_l, l} = \mathrm{BRT} \cdot \left| \left(\bigcup_{\forall \eta[k] \in \Theta_l} \mathrm{UCB}_k \right) \cap \mathrm{ECB}_l \right| \tag{25}$$

In this equation, Θ_l is the set of jobs, which are potential preempted by $\eta[l]$. The problem is to compute the set Θ_l. Following the second assumption, Θ_l is constructed with two properties:

- The number of elements of the set, denoted $|\Theta_l|$, is the maximum number of incomplete jobs at the preemption point. The computation of $|\Theta_l|$ is based on the following observations. Given a job $\eta[l]$ released at t_l, there are $l-1$ jobs released previously, which are $\eta[1], ..., \eta[l-1]$, because jobs in η are ordered by their release times. We have $l-1$ jobs executing in the interval $[t_1, t_l)$.

  The problem statement can be presented as follows: given $l-1$ jobs released in the $[t_1, t_l)$, what is the maximum number of incomplete jobs at a given time instant ?

  We design an algorithm that evaluates $l-1$ jobs. The algorithm starts from job $\eta[1]$ released at time t_1. Without interference from other jobs, the time instant $t_1 + C_1$ guarantees that $\eta[1]$ is completed. Then, the following computations are performed for the next job $\eta[i], (i = 2, 3, ..., l-1)$.

    1. We compute the number of potential preempted jobs.

2. We compute the CRPD.

3. We compute the time instants, which can guarantee that there are $1, 2, ..., (i-1)$ jobs completed.

The detailed explanation and a simple example of this algorithm is provided in the Appendix A.

When $|\Theta_l|$ is computed, the next step is computing the CRPD by evaluating $|\Theta_l|$ combinations of $l-1$ previously released jobs. We find the combination resulting in the highest number of evicted UCBs by the preempting job. This is a classical problem of generate all combinations of $l-1$ elements, taken $|\Theta_l|$ at a time.

- The elements of the set $\Theta_{[l]}$ are jobs which produce the largest set of $\left(\bigcup\limits_{\forall \eta[k] \in \Theta_{[l]}} UCB_k \right) \cap ECB_l$. For example, if $\eta[l]$ can preempt m jobs out of p (with $m < p$), the CRPD is computed by the combination of m jobs producing the largest set above. The computation requires a binomial coefficient complexity of $\binom{p}{p/2}$ or $\binom{p}{(p/2)+1}$.

Instead of Equation 25, a simplified computation could be used. In case of nested preemption, the CRPD can be computed by:

$$\gamma_{\Theta_l, l} = \text{BRT} \cdot \sum_{\forall \eta[k] \in \Theta_l} |UCB_k \cap ECB_l| \qquad (26)$$

In this computation, we only need to compute the CRPD between $\eta[l]$ and a single job. This solution is simpler because if $\eta[l]$ can preempt m jobs out of p, we take m jobs that result in the highest CRPD instead of checking m combination of p.

In this solution, if the sets of UCBs of tasks in Θ_l are mutually disjoint, Equation 26 gives the same result as Equation 25. If not, the CRPD computed by Equation 26 is more pessimistic. The elements of the set Θ_l are computed by evaluating $|\Theta_l|$ jobs with the highest number of evicted UCBs per job.

We name the two solutions, which are introduced in Equation 25 and Equation 26, CPA-PT and CPA-PT-Simplified. The CRPD Interference and number of preemptions computed by these solutions are upper-bounded by potential preemption. The number of preemptions in practice is always lower than the number of potential preemptions because of the problem of implicit priority as presented in the next section. By construction, CPA-PT dominates CPA-PT-Simplified and CPA-PT-Simplified dominates CPA-ECB.

Example

We provide an example of computing the interference and testing the feasibility of a task at a given priority level with the task set provided in Table 3 regarding CPA-PT solution.

Considering the job $\tau_3[0]$ at the lowest priority level, we need to check for jobs of higher priority tasks released in the interval $[0, 24)$, we have:

1 $\eta = \{\tau_1[0], \tau_2[8], \tau_1[12]\}$
2 $\eta[1] : C_1 = 3, \Theta_1 = \emptyset, \gamma_{\Theta_1,1} = 0$
3 $\eta[2] : C_2 = 8, \Theta_2 = \emptyset, \gamma_{\Theta_2,2} = 0$
4 $\eta[3] : C_3 = 3, \Theta_3 = \{\eta[2]\}, \gamma_{\Theta_3,3} = 2$

Applying the interference computation algorithm in [7], we have $I_3^0 = 16$. Given the capacity of τ_3 is 9 and the deadline of τ_3 is 24, we have $9 + 16 > 24$. We conclude that τ_3 is not schedulable at the lowest priority level.

The interference from higher priority tasks to $\tau_3[0]$ regarding CPA-PT solution is depicted in Figure 16. In this figure, the execution of τ_1 and τ_2 are separated to improve the readability. It does not imply the priority levels of τ_1 and τ_2.
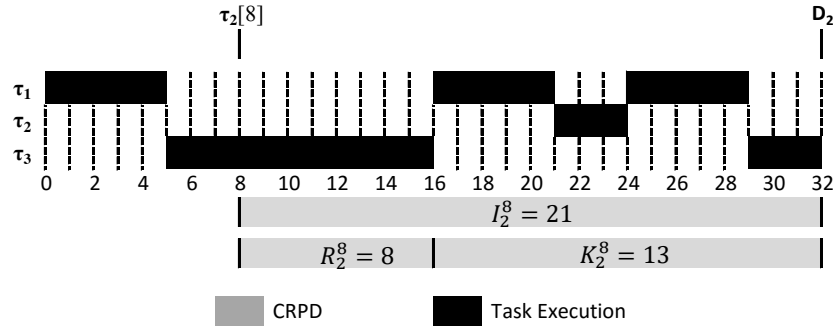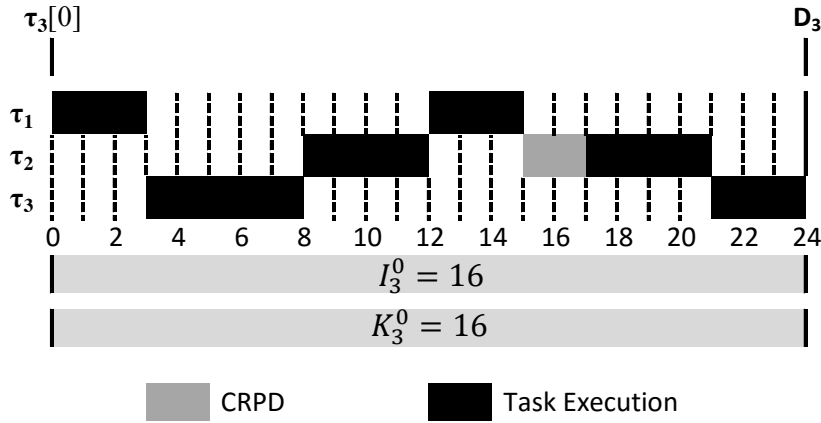

Figure 16: Interference from higher priority tasks to $\tau_3[0]$ regarding CPA-PT.

Considering the job $\tau_2[8]$ at the lowest priority level, we need to check for jobs of higher priority tasks released in the interval $[0, 32)$, we have:

1 $\eta = \{\tau_1[0], \tau_3[0], \tau_1[12], \tau_1[24], \tau_3[24]\}$
2 $\eta[1] : C_1 = 3, \Theta_1 = \emptyset, \gamma_{\Theta_1,1} = 0$
3 $\eta[2] : C_2 = 9, \Theta_2 = \{\eta[2]\}, \gamma_{\Theta_2,2} = 0$
4 $\eta[3] : C_3 = 3, \Theta_3 = \emptyset, \gamma_{\Theta_3,3} = 0$
5 $\eta[4] : C_4 = 3, \Theta_4 = \emptyset, \gamma_{\Theta_4,4} = 0$
6 $\eta[5] : C_5 = 9, \Theta_5 = \{\eta[4]\}, \gamma_{\Theta_5,5} = 0$

Applying the interference computation algorithm in [7], I_2^8 is computed by:

- The remaining capacity of $\tau_1[0]$ and $\tau_3[0]$ at time 8, denoted R_2^8, which is 4. We can see that the total capacity of $\tau_1[0]$ and $\tau_3[0]$ is 12.

- Capacity and CRPD of $\tau_1[12], \tau_1[24]$ and $\tau_3[24]$ in the interval $[8,32)$, denoted K_2^8, which is 11. We notice that I_2^8 does not include the capacity of $\tau_1[24]$ and $\tau_3[24]$ after time $t = 32$.
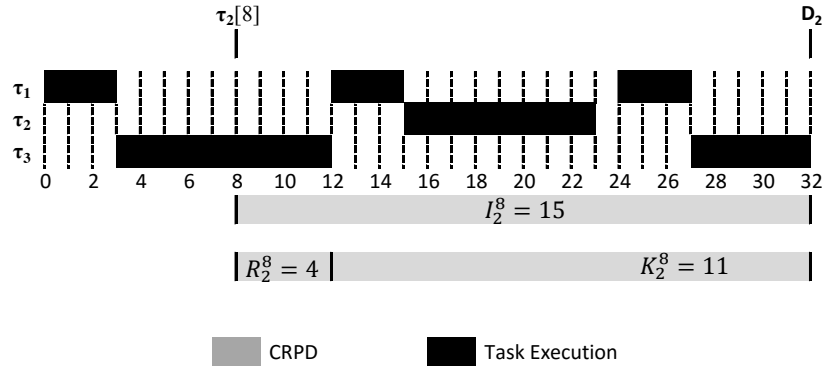
Figure 17: Interference from higher priority tasks to $\tau_2[8]$ regarding CPA-PT.

The interference from higher priority tasks to $\tau_2[8]$ regarding CPA-PT computation is depicted in Figure 17. In this figure, the execution of τ_1 and τ_3 are separated to improve the readability. It does not imply the priorities of τ_1 and τ_3.

We have $I_2^8 = 15$. Given the capacity of τ_2 is 8 and $D_2 = 24$, we have $8 + 15 < 24$. We conclude that τ_2 is schedulable at the lowest priority level.

From this example, we can see that CPA-PT is less pessimistic than CPA-ECB.

3.4.3   CPA -Tree

This solution consists in computing all possible preemption sequences of jobs in η set. This solution is called CPA-Tree. It reduces the pessimism regarding both the number of preemptions and the cost of preemptions. The number of preemptions is reduced by considering *implicit priorities* between tasks to reduce potential preemptions, while the cost of preemptions is lowered by identifying the exact preempting and preempted tasks at a given preemption point. We take into account the fact that relative priorities between two tasks could be implicitly set at a potential preemption instant. If the scheduler makes the decision allowing τ_j to preempt τ_k, it implicitly set the priority of τ_j higher than τ_k because we are assuming FPP scheduling context.

Definition 65 (Implicit priority). *An implicit priority is a priority assignment of tasks undergoing a potential preemption.*

This information is necessary to compute future events. For example, if the scheduler makes the decision of allowing a job of τ_j to preempt a job of τ_k, τ_k cannot preempt τ_j in the future.

To sum up, even if there is no complete priority assignment, priorities between two tasks can be set implicitly at the instant of a potential preemption. As a result, not all future potential preemptions will happen.

In this solution, we compute a tree structure to evaluate all possible preemption sequences. The tree $T = (N, E)$ is defined by N, the set of nodes and E, the set of edges:

- Each node n is defined by a 4-uplets (a, b, c, d) where a is a time stamp, b is the job executing at the instant a, c is the state of all jobs in the set at instant a, and d is the existing implicit priorities. The task-level priorities of jobs are set according to the scheduling decision.

- Each edge e from E models a scheduling decision. A scheduling decision must not violate existing implicit priorities.

- Branching is needed when the scheduler needs to make a decision. So each branch represents a set of scheduling decisions and preempting sequence. Interference including computational requirement and CRPD of jobs in η is computed for each branch. Concerning the preemption cost, CRPD is computed accurately at each preemption point according to the preempting and preempted tasks. If there exists a branch for which the job of task τ_i is not schedulable, then the task is not schedulable at this priority level.

A recursive algorithm is implemented to compute the tree. The algorithm assesses the η set. It starts from the first job and ends at the last job in the set. When the algorithm terminates, we can assess each branch in order to find if the job of task τ_i meets its deadline. The detailed explanation and a simple example of this algorithm is provided in the Appendix A.

Regarding the two problems in Section 3.3, in this solution, the number of preemptions of a branch is limited by the set of implicit priorities of this branch. The attribute d of each node provides information of implicit priorities. A scheduling decision must follow the set of existing implicit priorities. Each branch of the tree stores a set of consistent implicit priorities. CRPD is computed when a preemption scheduling decision happens.

This solution is close by computing the scheduling sequence of jobs in η set with all possible priority orderings. The actual complexity highly depends on the number of potential preemptions or the number of scheduling decisions.

All possible preemption sequences are addressed in this solution. Hence, this solution computes all possible total CRPD Interference and accounts for the worst case. By construction, CPA-Tree dominates the other solutions.

Example

We provide an example to illustrate the computation of CPA-Tree. Considering the job $\tau_3[0]$ at the lowest priority level, we need to check for jobs of higher priority tasks released in the interval $[0, 24)$ We have:

1 $\eta = \{\tau_1[0], \tau_2[8], \tau_1[12]\}$
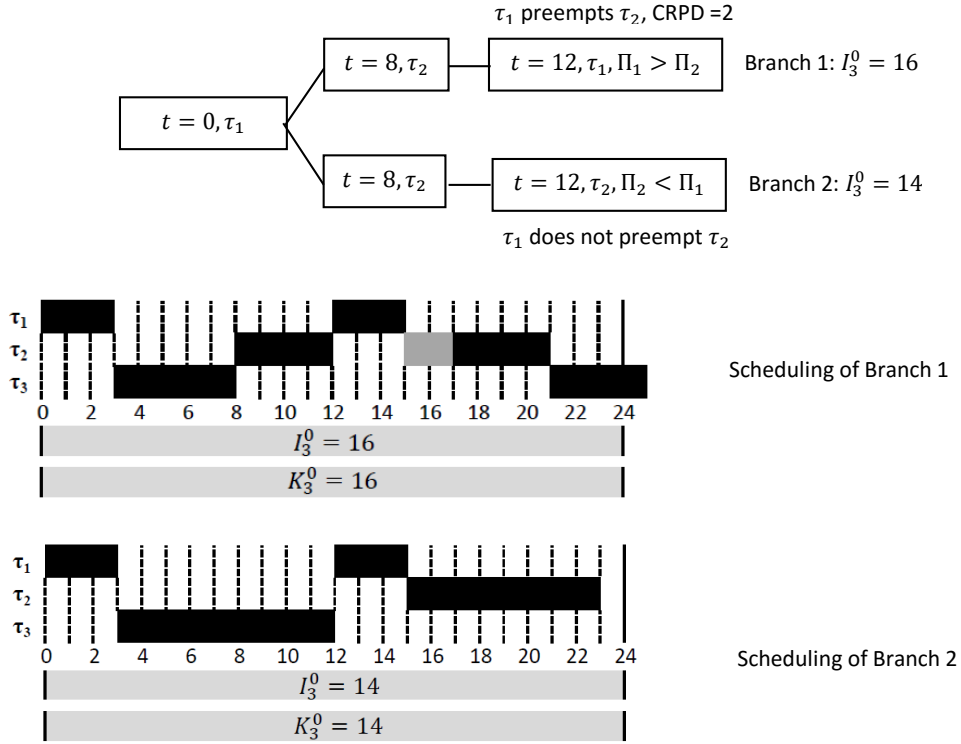
Figure 18: CPA-Tree for $\tau_3[0]$

The computation of the tree is depicted in Figure 18. We have two branches corresponding to two cases. The first case is τ_1 preempts τ_2 at time $t = 12$. The second case is τ_1 does not preempt τ_2 at time $t = 12$. The interference is computed by taking into account the capacity of jobs and the CRPD in each branch. In this example, τ_3 is concluded to be not schedulable at the lowest priority level because it is not schedulable in one branch.

3.5 COMPLEXITY OF THE ALGORITHMS

In this section, we present the complexity of our solutions. Considering a task τ_i, the complexity of each solutions lies in the computation of the interference from jobs of higher priority tasks for each release of τ_i during its feasibility interval. In [7], Audsley showed that the complexity of the original feasibility test is bounded by the complexity of testing task τ_n, with n is the number of tasks, at the lowest priority level. At this priority level, the level-i hyper-period of τ_n, denoted P_n, is equal to the hyper-period. The complexity of the feasibility test in [7] is given by:

$$O(X), \text{with } X = \left(\frac{P_n}{T_n} \sum_{j=1}^{n-1} \left(\left\lceil \frac{T_n - D_n}{T_j} \right\rceil + \left\lceil \frac{D_n}{T_j} \right\rceil \right) \right) \tag{27}$$

As stated in [7], the complexity of the priority assignment algorithm is given by n multiples with the feasibility test complexity. In the next section, we present how this *feasibility test complexity* is changed due to our propositions. We assume that cache access profiles of tasks are precomputed before assigning priorities to task. Thus, we do not take into account the complexity of UCB and ECB computation.

3.5.1 *CPA-ECB*

The complexity of the CPA-ECB is the same with the complexity in [7]. CPA-ECB only modifies the capacity of each task. No additional computation is needed.

3.5.2 *CPA-PT and CPA-PT-Simplified*

The complexity added by this solution lies in the computation of k combinations of m potential preempted jobs. The number of combination is bounded by the binomial coefficient of n tasks. The complexity of CPA-PT solution is then given by:

$$O\left(\binom{n}{n/2} \cdot X\right) \tag{28}$$

The complexity of CPA-PT-Simplified solution lies in the ordering the number of UCBs evicted by the preempting task of preempted tasks. It is bounded by $n\log(n)$:

$$O(n\log(n) \cdot X) \tag{29}$$

3.5.3 *CPA-Tree*

The tree represents all possible preemptions of a set of jobs in the interval $[t - T_i + D_i, t + D_i)$. In the worst case, computing the tree has a complexity similar to the complexity of computing the scheduling for all jobs with all possible priority assignments.

Besides priority level n, there are $(n-1)$ higher priority levels. The complexity is:

$$O((n-1)! \cdot X) \tag{30}$$

In conclusion, the less pessimistic the assumptions of the solution, the higher the complexity. In the next section, we evaluate the efficiency and the scalability of those solutions.

3.6 EVALUATION

To evaluate the proposed approaches, experiments investigating their performances and efficiency are made. The configuration of our experiments is based on the existing work in [4]. Task sets are generated with the following configuration:

- Task periods are uniformly generated from 5 ms to 500 ms, as found in most automotive and aerospace hard real-time applications [4].

- Generated task sets are harmonic in order to have a low feasibility interval and scheduling simulation period.

- Task deadlines are implicit, i.e. $\forall i : D_i = T_i$.

- Processor utilization values (PU) are generated using the UUniFast algorithm [15].

- Task execution times are set based on the processor utilizations and the generated periods: $\forall i : C_i = U_i \cdot T_i$, where U_i is the processor utilization of task i.

- Task offsets are uniformly distributed from 1 to 30 ms.

The cache and cache utilization of tasks are generated with the following configuration:

- The cache is direct mapped.

- The number of cache blocks is equal to 256.

- The block-reload time is 8 μs [4].

- The cache usage of each task is determined by the number of ECBs. They are generated using UUniFast algorithm for a total cache utilization (CU) of 5. UUniFast may produce values larger than 1 which means a task fills the whole cache. ECBs of each tasks are consecutively arranged from a cache block. For each task, the UCBs are generated according to a uniform distribution ranging from 0 to the number of ECBs times a reuse factor (RF). If set of ECBs generated exceeds the number of cache blocks, the set of ECBs is limited to the number of cache blocks. For the generation of the UCBs, the original set of ECBs is used.

3.6.1 *Evaluating the impact of CRPD on the original OPA*

The objective of this experiment is to evaluate the impact of CRPD to the original OPA algorithm.
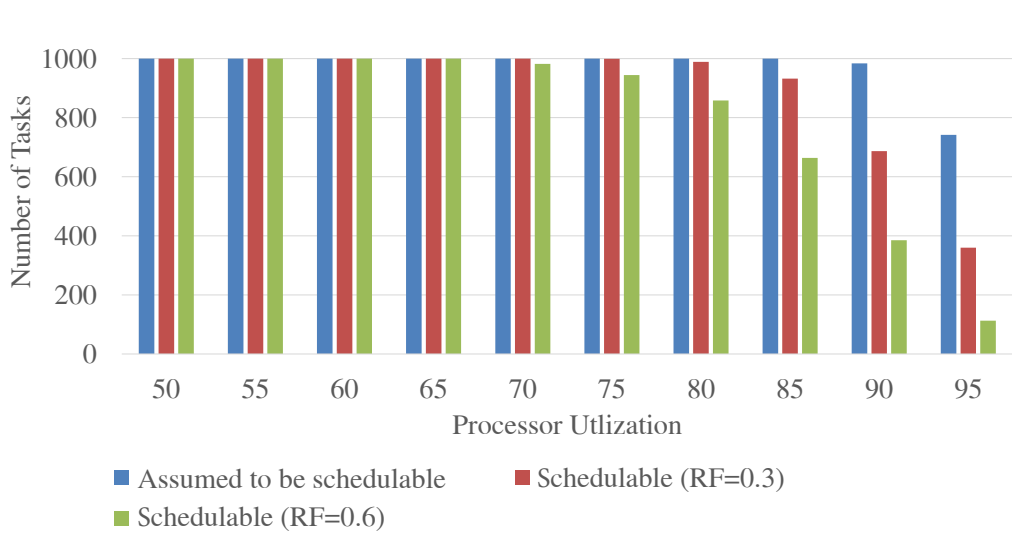
Figure 19: Number of task sets assumed to be schedulable by OPA and number of task sets actually schedulable when CRPD is taken into account.

In each experiment, the processor utilization, which does not include preemption cost, is varied from 0.50 to 0.90 with steps of 0.05. Experiments are performed with two RFs of 0.3 and 0.6. Task set size is fixed at 5 tasks per set. For each processor utilization value and reuse factor, 1000 task sets are generated.

Figure 19 shows the result of this experiment. For the chosen scenario, when the processor utilization is varied from 70 to 95, there is a significantly difference between the number of task sets analyzed as schedulable by OPA and the number of task sets which are actually schedulable. In addition, the number of schedulable task set decreases remarkably when the reuse factor increases from 0.3 to 0.6. Without taking CRPD into account, the OPA priority assignment failed to identify significant number of unschedulable task sets.

In conclusion, this experiment shows that without considering the effect of CRPD, unschedulable task set can be identified as schedulable ones.

3.6.2 *Efficiency evaluation of CPA solutions*

The objective of this experiment is to evaluate the efficiency of the proposed priority assignment algorithms. Each algorithm is evaluated by two metrics. First, we evaluate the number of task sets analyzed as being schedulable by our priority assignment algorithms. Second, we evaluate how close our algorithms are to the exhaustive search approach in terms of schedulable task sets. The configuration is the same with the previous experiment.

This experiment is composed of two steps. First, we perform priority assignments with different approaches to the generated task sets. A task set is assumed to be schedulable if the algorithm finishes assigning priorities to all tasks. Sec-
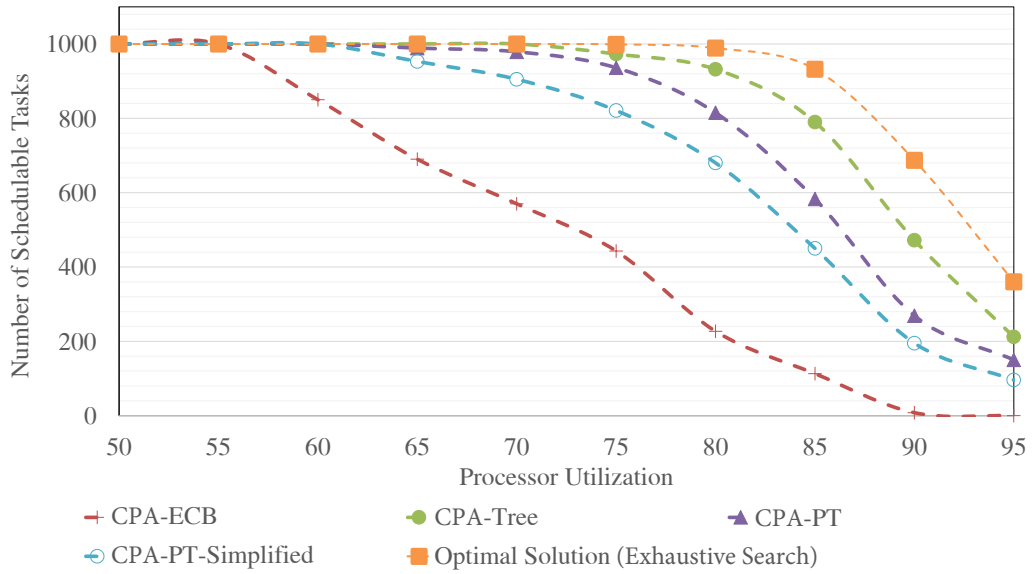
Figure 20: Number of task sets found schedulable by the priority assignment algorithms, RF = 0.3

ond, we perform scheduling simulations with the assigned priorities tasks to verify that the task set is practically schedulable or not while experimenting the effect of CRPD. In addition, we also perform an exhaustive search by testing all priority assignments for a task set and performing scheduling simulations to compare with.

Fig. 20 and Fig. 21 display the result of this experiment. Regarding the first metric, all task sets assumed to be schedulable by the proposed approaches are by construction schedulable. Indeed, the objective of this work was first to eliminate tasks sets that were found to be schedulable with OPA but that are not. In other words, our feasibility condition is only a sufficient condition. Of course, when comparing to the optimal solution, we can see that our solutions are using a sufficient but not necessary. However, the proposed priority assignment algorithms succeeded in identifying a large number of schedulable task sets. More importantly, depending on the chosen solution, one can get closer to the optimal (exhaustive) solution.

Amongst the four approaches, CPA-Tree found the highest number of schedulable task set.

The higher the processor utilization, the lower the percentage of schedulable task sets found by our approach as compared to the exhaustive search. For instance, at the processor utilization of 80% and RF=0.6, approximately 60% of the schedulable task sets were found by CPA-Tree while comparing to 80 % found by the optimal solution.
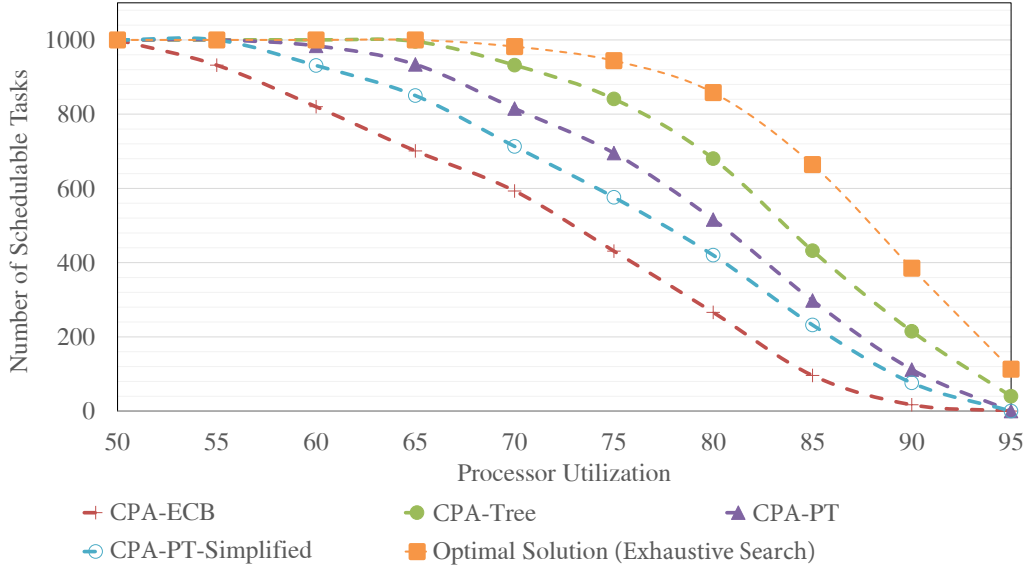
Figure 21: Number of task sets found schedulable by the priority assignment algorithms, RF = 0.6

The result is compliant with the level of pessimism of each approach, as discussed in Section 3.4. In addition, the higher the complexity of the proposed algorithms is, the closer our approach is to the optimal solution.

Our approaches do not only provide schedulable tasks taking into account CRPD, but they also provide several task sets that were not found to be schedulable with either OPA, RM or DM in our experiments. However, the number of those additional task sets are only 0.7 to 1% of the generated task sets when processor utilization is greater than 70%.

Furthermore, when RF increase, the gap between the optimal solution and our proposed solutions also increase. At the processor utilization of 80% and reuse factor of 0.3, the distance between the optimal solution and CPA-Tree is roughly 50 schedulable tasks. At the reuse factor of 0.6, that distance is roughly 200 tasks. We can conclude that the pessimism of our algorithms increase when RF increases.

We also use weighted schedulability [9] measure, which is shown in Table 4, in order to compare our approaches. We use the weighted schedulability measure $W_y(p)$ for schedulability test y as a function of parameter p. For each value of p, the measure combines data for all task sets τ generated for all the sets of equally spaced utilization levels. Let $S_y(\tau, p)$ be the binary result (1 if schedulable, 0 otherwise) of schedulability test y for a task set τ and parameter value p then:

$$W_y(p) = \left(\sum_{\forall \tau} u(\tau) \cdot S_y(\tau, p) \right) \Big/ \sum_{\forall \tau} u(\tau)$$

Approach	Weighted Schedulability	
	RF=0.3	RF=0.6
CPA-ECB	0.42	0.41
CPA-PT-Simplified	0.65	0.50
CPA-PT	0.72	0.56
CPA-Tree	0.80	0.65
Optimal_Solution	0.87	0.74

Table 4: Weighted Schedulability Measure

The result in Table 4 shows the distance between our solutions and the optimal solution for all generated task sets and processor utilizations. From the result of weighted schedulability analysis, the different in terms of schedulability task set coverage between CPA-Tree solution and the optimal solution is 8-9 %.

In conclusion, the feasibility conditions used in our feasibility tests are sufficient but not necessary; but our priority assignment approaches succeeded in identifying large schedulable task sets comparing to the optimal solution.

3.6.3 *Evaluating the performance of the proposed feasibility test*

The objective of this experiment is to evaluate the cost of computing the interference of a set of jobs in an interval. Algorithms of the approaches CPA-PT, CPA-PT-Simplified and CPA-Tree are evaluated. The CPA-ECB approach is not evaluated because it does not increase the complexity of the original solution [7] as presented in Section 3.5.

We evaluate the computation time of performing one feasibility test for a release of a task. The number of tasks is varied from 4 to 100. PU is 80% and RF is 0.3. For each number of tasks, 1000 task sets are generated. Then, the computation of interference is performed. Experiments are performed on a PC with Intel Core 2 Duo CPU E8400, having 4 GB of memory, running Ubuntu 12.04 32 bits version. Memory consumption measurement is achieved by using a script provided at https://gist.github.com/netj/526585.

The results of the experiment are shown in Table 5, 6 and 7. The first observation is that computation time of CPA-Tree increases exponentially when the number of tasks increases. It takes averagely 455 seconds and 3434253 KB of memory for 9 tasks. This is compliant with the exponential complexity of the feasibility test as shown in Equation 30.

CPA-PT solution has a better scalability. As an example, the computation of interference of 30 tasks takes averagely 400 seconds. Memory consumption in-

Tasks	Memory Consumption(KB)	Computation time (s)
4	89028	0.03278
5	94572	1.20466
6	170432	11.79959
7	394812	40.92083
8	1975860	222.95394
9	3434253	455.32684

Table 5: Space and time performances of the CPA-Tree

Tasks	Memory Consumption(KB)	Computation time (s)
5	12852	0.07290
10	18408	0.12694
15	86120	0.53614
20	153440	7.97529
25	655680	94.83195
30	3516704	399.27149

Table 6: Space and time performances of the CPA-PT

Tasks	Memory Consumption(KB)	Computation time (s)
5	12988	0.00008
10	13572	0.00019
20	13932	0.00036
30	14236	0.00053
40	14876	0.00062
50	15236	0.00096
60	15646	0.00118
70	16213	0.00218
80	16731	0.00328
90	17222	0.00398
100	17941	0.00518

Table 7: Space and time performances of the CPA-PT-Simplified
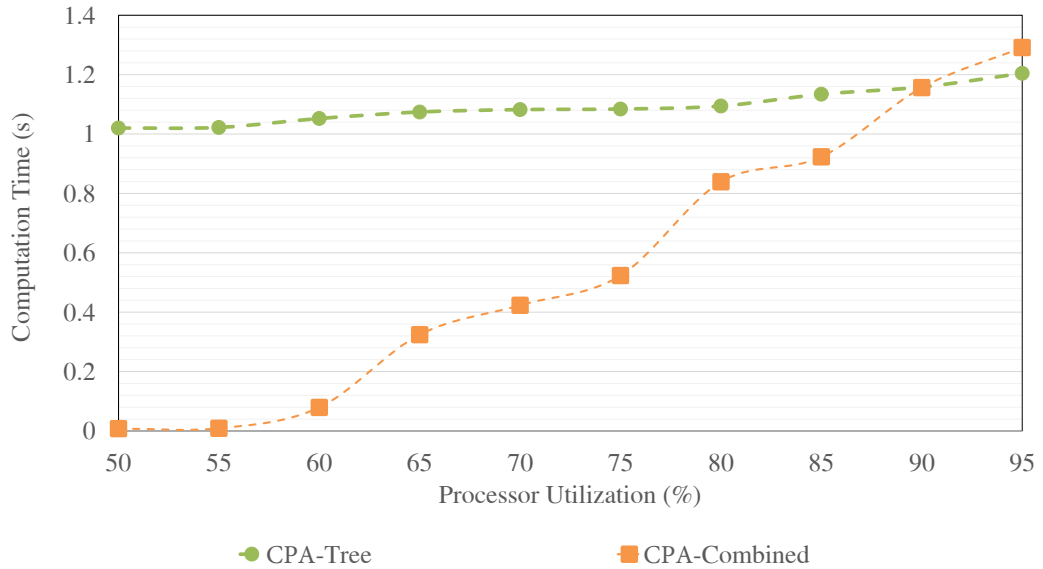
Figure 22: Comparison between CPA-Tree and combined approach in terms of computation time

crease significantly when the number of task increases. This is compliant with the binomial coefficient complexity of the feasibility test as shown in Equation 29. CPA-PT-Simplified has the best scalability. The computation time for 30 tasks is less than 1 second. In addition, memory consumption is less than 20000 KB (20 MB).

In conclusion, CPA-PT and CPA-Tree have the higher complexity and lower scalability comparing to CPA-PT-Simplified. However, CPA-PT-Simplified is the most pessimistic one. Again, the higher the complexity of the proposed algorithms, the closer to the optimal solution is our approach.

3.6.4 *Combined solution: CPA-Combined*

We perform experiment to measure the solution of combining all the four solutions in one priority assignment algorithm. The idea is to improve the performance on task sets that have low PU.

We implement a priority assignment algorithm with three level of feasibility tests. We start verifying the feasibility of a task set using the solution with the lowest level of complexity but highest level of pessimism: CPA-ECB. If the task set is not schedulable, a solution with higher level of complexity but lower level of pessimism is used until the task set is found schedulable. We perform experiment with task sets generated following the base configuration with task set size of 5 tasks.

The results of the experiment are shown in Fig 22. The average computation time of the combined solution are compared to the CPA-Tree solution. We can

have the following observation. First, the computation time of the combined solution is significantly lower than the CPA-Tree at low processor utilizations. Second, there are two significant increases of computation time when PU raises from 60% to 65% and from 75% to 80%. It can be explained as follows. They are two points where the less complex approach is not efficient and we need to use the more complex approach. The result gives an insight of which approach is appropriate for a specific PU. Finally, there is an overhead when using the combined solution for task sets at high PUs. When PU = 95, most of the task sets are not schedulable. As a result, the combined solution has to choose CPA-Tree most of the time. The overhead is due to the computation time of the less complex solutions.

To sum up, the combined solution helps reduce the computation time of the priority assignment on task sets that have PU < 90 %.

3.7 CONCLUSIONS

In this chapter, we investigate the problems with classical priority assignment algorithms and present an approach to perform priority assignment with CRPD taken into account. Our approach is based on the OPA and the original feasibility test proposed in [7]. We proposed and evaluated five solutions to extend the feasibility test in [7] to take into account the CRPD. They are named CPA-ECB, CPA-PT, CPA-PT-Simplified, CPA-Tree and CPA-Combined. Experiments have shown that task sets identified to be schedulable by our solutions are actually schedulable when performing scheduling simulation with CRPD. The different in terms of schedulability task set coverage between our best solution and the optimal solution is 8-9 %. In addition, there is a trade-off between the complexity and the pessimism of the proposed solutions. CPA-Tree has a high complexity but find more schedulable task sets than CPA-PT and CPA-PT-Simplified.

Chapter 4

CRPD-AWARE SCHEDULING SIMULATION

Contents

Scheduling simulation is a popular analysis method which provides a mean to evaluate the schedulability of RTES. It allows RTES designers to perform fast prototyping with a certain level of accuracy. There are various research work in this domain and several scheduling simulators [28, 75, 84, 47]. However, to the best of our knowledge, in the context of RTES with cache memory, applicability and validity of scheduling simulation are still open subjects.

One of the most important properties, which we need to identify before performing scheduling simulation, is the simulation interval. In other words, the question is how long we should run the simulation. Ideally, we need to be able to capture all the possible behaviors of our system or at least the worst case in the simulation interval. As introduced in section 1.5, the minimum interval of time over which we should perform the simulation is known as the feasibility interval [7, 43].

Established results and proofs about the feasibility interval did not take into account cache memory and the effect of CRPD. This issue comes from an uncertainty about the use of CRPD computation models in scheduling simulation and theirs sustainability analysis.

This chapter deals with the problems concerning CRPD-aware scheduling simulation for RTES with cache memory. Detailed assumptions regarding system model and cache accesses are provided in Section 4.2. This chapter addresses the following topics.

- First, we investigate CRPD computation models used in scheduling simulation. We present existing issues regarding the pessimism of these models. Then, we discuss about the sustainability of scheduling simulation with classical CRPD computation models. We explain the problem related to CRPD in sustainability analysis and the reason why CRPD-aware scheduling simulation is not sustainable in general cases.

- Second, we propose a new CRPD computation model named FSC-CRPD to address the previous issues. In this model, based on an observation from real system execution in [59], we take a new assumption that bounds the CRPD by the executed capacity of a task. When this assumption holds, scheduling simulation is less pessimistic and then becomes sustainable with regard to the capacity parameter. The conclusion about the sustainability of scheduling simulation with FSC-CRPD allows us to prove the feasibility interval of our system model.

The established results show that for some RTES with cache memory, scheduling simulation can be applied as a method to verify the feasibility and schedulability.

The rest of this chapter is organized as follows. Section 4.1 presents the definition and the characteristics of CRPD-aware scheduling simulation. In section 4.2, we investigate classical CRPD computation models used in scheduling simulation, analyze existing issues and propose our solution. In section 4.3 and 4.4, we present our analysis on sustainability of CRPD-aware scheduling simulation and feasibility interval of the system model with FSC-CRPD computation model. Finally, section 4.5 concludes the chapter.

4.1 DEFINITIONS

In this section, we present a definition of CRPD-aware scheduling simulation and its characteristics. The main objective of CRPD-aware scheduling simulation is to analyze the effect of CRPD on the schedulability of a RTES. Cache intrinsic behaviors [72, 12] are not taken into account and are assumed to be included in the capacity (WCET) of a task.

Definition 66. *A CRPD-aware scheduling simulation is a scheduling simulation that takes into account the effect of* CRPD *in preemptive scheduling context.*

We define one term that is important when discussing about CRPD-aware scheduling simulation: execution time. When CRPD is taken into account, there will be a difference between the capacity of a task and the execution time of a job of this task. Without considering the effect of CRPD, the time a job of a task is executed on the processor is equal to the task capacity. However, when CRPD

is taken into account, this job may occupy the processor longer than the task capacity because it has to spend time to reload memory blocks that are evicted by the preemption.

Definition 67 (Execution time). *The execution time of a job of task τ_i is the total time during which this job occupies the processor.*

We also define two scheduling events that need to be handled in CRPD-aware scheduling simulation:

1. Preemption event

2. Task execution event

The two scheduling events can be raised in either preemptive offline scheduling [6] or preemptive online scheduling [6] context. The definition of the two scheduling events and our proposed event handlers in scheduling simulation are as follows:

Definition 68 (Preemption event). *A preemption event is raised when a task is preempted by higher priority tasks.*

Cache state or data in the cache of task is updated at the event of preemption.

Definition 69 (Task execution event). *A task execution event is raised when a task is executing on the processor for each simulation time unit.*

CRPD is computed when a task resumes its execution. The CRPD added to the remaining capacity of task τ_i when it resumes execution at time t is represented by γ_i^t.[1]

Besides classical scheduling parameters such as scheduler and task model, CRPD-aware scheduling simulation requires the definition of a CRPD computation model.

Definition 70. *A CRPD computation model consists of a specification of cache access profiles of tasks, an algorithm to update cache state at preemption event and an algorithm to compute CRPD at task execution event based on the cache access profiles.*

In the next section, we present the classical CRPD computation models, point out several issues and propose a solution.

4.2 CRPD COMPUTATION MODELS

There are several assumptions that are made in order to study the effect of CRPD in scheduling simulation considering the analysis based on UCBs and ECBs. Each set of assumptions forms a specific CRPD computation model.

---

1 This notation is different from the notation of CRPD used in WCRT analysis ($\gamma_{i,j}$) that represents the worst-case CRPD when τ_i is preempted by a higher priority task τ_j

In this section, we discuss about classical CRPD computation models, analyze the assumptions which are made in each model. Then, we propose a new CRPD computation model that includes an assumption to bound the CRPD by the executed capacity of a task. This assumption helps reducing the pessimism in term of CRPD and improving the sustainability of scheduling simulation.

The following assumptions about the system model are applied to all CRPD computation model.

- We assume a RTES with cache memory that consists of n independent periodic tasks, $\tau_1, ..., \tau_n$ with constrained deadlines ($D_i \leqslant T_i$), scheduled by a FPP scheduler.

- The capacity (WCET) of a task is computed by assuming a non-preemptive scheduling starting from an empty clean cache. In other words, cache intrinsic behaviors [72, 12] are included in the capacity of a task.

- Cache access profiles of tasks are defined and computed before simulation time.

- When a task completes execution, its instructions in the cache are completely evicted. In other words, we do not take into account the problem of persistence cache block [80].

We have not yet investigated the problem of CRPD-aware scheduling simulation for tasks with arbitrary deadline. In this case, modeling cache accesses and evaluating the number of UCB loaded into the cache of a task could be complex because there are multiple jobs are released and executed.

4.2.1 *Classical CRPD computation models*

We present two CRPD computation models that are used as a part of experiments or examples regarding CRPD in [67, 29, 30, 4].

Constant CRPD for each task (CT-CRPD)

This CRPD computation model is described as follows:

- *Cache access profile*: cache access profile is taken into account by considering the worst-case effect of a preemption to a task. A task τ_i experiences a constant CRPD when it is preempted by a higher priority task τ_j.

  As cache access profile is not specified, this model is pessimistic because preempting tasks may not evict the data in the cache of the preempted task τ_i. In other words, τ_i does not always have to reload its data in the cache. In addition, the pessimism also depends on the method of computing the

constant CRPD for τ_i. For example, we can assume that either all the data in the cache of τ_i is evicted and needs to be reloaded or only the UCBs of τ_i is evicted and need to be reloaded.

- *Preemption event handler*: preemption event handler is not specified.

- *Task execution event handler*: when a task τ_i resumes execution after be preempted by a higher priority τ_j, a constant CRPD is added to the remaining capacity of τ_i.

This model was used in [67] to analyze scheduling abnormalities which occur when CRPD is taken into account. A fixed CRPD for each task is also used in SimSo scheduling simulator [29, 30].

Fixed Sets of UCBs and ECBs (FS-CRPD)

This CRPD computation model is described as follows:

- *Cache access profile*: the cache access profile of a task is modeled by its sets of UCBs and ECBs. It is assumed that any partial execution of a task needs to load all of its UCBs into the cache. In addition, a task uses all of its ECBs.

  This assumption is pessimistic considering the real execution of a task. However, to relax this assumption, information about which memory blocks are being used at a given instant must be provided. In other words, we need a more detailed task model in which each unit of task capacity is linked to one or several memory blocks or cache blocks. Only with this information, CRPD can be computed based on which UCBs are being used at a given instant. However, as far as we know, there is no timing analysis tool that can provide such information. Relaxing this assumption requires a timing analysis technique, which is beyond the scope of this thesis.

- *Preemption event handler*: when a preemption event is raised, we compute which UCBs of the preempted task are evicted by the preempting task.

  There are two types of preemption: direct preemption and nested preemption. As previously introduced, a direct preemption is a preemption between two tasks when the lower priority task is executing. As shown in Fig. 23, the preemption between τ_2 and τ_3 is a direct preemption. An indirect preemption is a preemption between two tasks when the lower priority task was previously preempted by another task and is not executing. In Fig. 23, the preemption between τ_1 and τ_3 is a nested preemption.
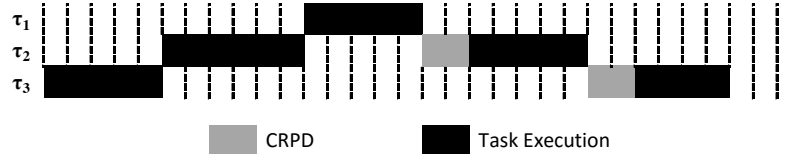
Figure 23: Example of direct preemption and nested preemption. We have three tasks τ_1, τ_2, τ_3 with $\Pi_1 > \Pi_2 > \Pi_3$.

In this example, the CRPD added to the remaining capacity of τ_3 must be computed based on the number of UCBs evicted by both τ_1 and τ_2.

- *Task execution event handler*: When the task execution event is raised, we compute the CRPD and add it to the remaining capacity of the preempted task.

The CRPD is computed as follows. We use the notation UCB_i^t that denotes the set of UCBs in the cache of τ_i at a time t. Assume a task τ_i is released at t_1, we have:

$$UCB_i^{t_1} = UCB_i \tag{31}$$

This assignment is done to take into account the assumption that cache intrinsic interference is included in the capacity of a task. The capacity of a task already includes the time to load memory blocks into the cache when it executes non-preemptively.

For each time unit after t, if τ_i is not preempted by any higher priority task, its UCBs are not evicted. As a result, the set is updated as follows:

$$UCB_i^t = UCB_i^{t-1} \tag{32}$$

Whenever τ_i is preempted by a higher priority task τ_j at time t_2, UCB_i^t is updated by taking into account the UCBs of τ_i evicted by the ECBs of τ_j. The set is updated as follows:

$$UCB_i^{t_2} = UCB_i^{t_2-1} - ECB_j \tag{33}$$

Assume that task τ_i resumes execution at time t_3, the CRPD added to the capacity of τ_i is computed as follows:

$$\gamma_i^{t_3} = |UCB_i - UCB_i^{t_3}| \cdot BRT \tag{34}$$

Then, the computed preemption cost is added to the remaining capacity of the task. In addition, the set of UCBs in the cache of a task is updated as follows:

$$UCB_i^{t_3} = UCB_i \tag{35}$$

FS-CRPD computation model provides a more precise preemption cost by taking into account the effect of both the preempting task and the preempted task. It was used in [4] to design a scheduling simulation experiment taking into account CRPD. Besides scheduling simulation, there are also several WCRT analysis methods that are based on this CRPD computation model [82, 4].

4.2.2 *Problems with classical models*

Let discuss about following observations from the real execution of a system. There must be a correlation between the executed capacity of a task and the number of UCBs loaded into the cache. If a task is preempted shortly after it is released and executed, it may not have yet loaded all of the UCBs and will not experience the worst-case CRPD. This observation is not taken into account by both CT-CRPD and FS-CRPD. It creates the two following problems:

- *The first problem* is that CT-CRPD and FS-CRPD lead to an over-estimation of preemption cost. In some cases, the CRPD computed can be larger than the executed capacity of a task before it is preempted.

  For example, we assume a task set of two tasks given in Table 8. In this example, we do not take into account the deadlines and periods of tasks.

Task	C_i	T_i	D_i	O_i	UCB_i	ECB_i	Π_i
τ_1	4	–	–	2	$\emptyset$	{1,2,3}	2
τ_2	7	–	–	0	{1,2,3}	{1,2,3,4}	1

Table 8: Task set example.

  We assume that $BRT = 1$ unit of time. The scheduling of this task set over 14 units of time is provided in Figure 24.
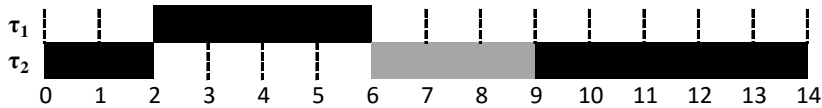
Figure 24: Over-estimation of CRPD

  At time $t = 2$, τ_2 is preempted by τ_1. With FS-CRPD, the computation of CRPD added to the capacity of task τ_2 in the scheduling is presented below. In order to keep the presentation short and clear, we do not use the notation UCB_i^t that denotes the set of UCBs in the cache of τ_i at a time t. It is replaced by the parameter UCB_i' for each time instant.

At time $t = 0$, τ_2 is released. At time $t = 2$, τ_2 is preempted by τ_1. Then, at time $t = 6$, τ_2 resumes and the CRPD added to the capacity of τ_2 is computed as follows:

---

1   `t = 1:` $UCB_2' = UCB_2 = \{1, 2, 3\}$

2   `t = 2:` $UCB_2' = UCB_2' - ECB_1 = \{1, 2, 3\} - \{1, 2, 3\} = \emptyset$

3   `t = 6:` $\gamma_2^6 = |UCB_2 - UCB_2'| \cdot BRT = |\{1, 2, 3\} - \emptyset| \cdot 1 = 3$

We see that there are 3 units of time of CRPD added to the remaining capacity of τ_2. In this case, with FS-CRPD, τ_2 experiences CRPD from cache blocks that may be not even loaded into the cache yet.

- *The second problem* is that scheduling simulation of this model is not sustainable with regard to capacity parameter. We discuss this problem in detail in section 4.3. This problem is more critical because it greatly discourages the use of scheduling simulation for RTES with cache. If we choose this model, we also have to assume that the operating system always executes a task up to its WCET: if the task completes before its WCET, it still holds the processor until the WCET is reached. In addition, a similar problem will raise with CRPD. As a result, whenever a task is preempted, the capacity must be added as the computed CRPD.

4.2.3 *Fixed sets of UCBs and ECBs with constraint (FSC-CRPD)*

We propose an extension of FS-CRPD computation model in order to address the two problems presented in the previous section. In our CRPD computation model, the following assumption is taken:

The interval of time that a task spends to load memory blocks into cache memory cannot be larger than the interval of time in which it is executed on the processor. In other words, if a task τ_i executes non-preemptively in an interval of time Δ, there cannot be more than $\lfloor \frac{\Delta}{BRT} \rfloor$ memory blocks loaded into the cache. From this assumption, we deduce the following theorem:

Theorem 5. *If task τ_i executed in an interval of time Δ and loaded ρ_i UCBs into the cache, we have $\rho_i \cdot BRT \leqslant \Delta$.*

In our CRPD computation model, based on the assumption, we assume that a task starts execution by loading its UCBs but there is a constraint about the number of UCBs loaded

Preemption cost is computed as follows. When a task τ_i is preempted, the number of loaded UCBs, denoted as ρ_i, is stored by the simulator. ρ_i is computed as follows:

$$\rho_i = \lfloor \frac{\Delta}{BRT} \rfloor \tag{36}$$

For the illustration of CRPD in the next examples, BRT = 1 units of time. The CRPD added to the capacity of τ_i when it resumes at time t_2 is now computed as follows:

$$\gamma_i^{t_2} = \min(|UCB_i - UCB_i^{t_2}|, \rho_i) \cdot BRT \tag{37}$$

This equation guarantees that the CRPD cannot be larger than the executed capacity of task τ_i by taking into account ρ_i parameter. We apply FSC-CRPD computation model to the example in Figure 24. At time $t = 0$, τ_2 is released. At time $t = 2$, τ_2 is preempted by τ_1. Because τ_2 has executed only 2 units of time, there is only 2 UCBs loaded into the cache. Then, at time $t = 6$, τ_2 resumes and the CRPD added to the capacity of τ_2 is computed as follows:

1   t = 1: $UCB_2' = UCB_2 = \{1, 2, 3\}$
2   t = 2: $UCB_2' = UCB_2' - ECB_1 = \{1, 2, 3\} - \{1, 2, 3\} = \emptyset$
3   t = 2: $\rho_2 = 2$
4   t = 6: $\gamma_2^6 = \min(|UCB_2 - UCB_2'|, \rho_2) \cdot BRT = \mathbf{2}$

The preemption cost added is only 2 units of time, which is not larger than the executed capacity of task τ_2.

In the next section, we discuss about the sustainability analysis of CRPD computation models.

4.3 SUSTAINABILITY ANALYSIS

In this section, we recall the definition of sustainability, discuss about sustainability analysis of classical CRPD computation models and analyze sustainability of scheduling simulation with FSC-CRPD computation model.

4.3.1 *Definitions*

The definition of sustainability was given in [20].

Definition 71. *A given scheduling policy and/or a schedulability test is sustainable if any system that is schedulable under its worst-case specification remains so when its behavior is better than worst-case. The term better means that the parameters of one or more individual task(s) are changed in any, some, or all of the following ways: (1) decreased capacity, (2) larger periods and (3) larger relative deadlines.*

We explain the reason why these changes are considered better behaviors. Assume a job of τ_i released at time t and has a deadline at $t + D_i$. In preemptive scheduling context, $\tau_i[t]$ experiences interferences [7] from higher priority tasks in the interval $[t, t + D_i)$, denoted I_i^t. The definition of I_i^t and its meaning were

presented in Section 3.3.1. The job of τ_i is feasible if the following condition is satisfied:

$$C_i + I_i^t \leqslant D_i \tag{38}$$

Decreased capacity decrease either C_i and could also decrease I_i^t as capacities of higher priority tasks are decreased. Larger periods could decrease I_i^t by reducing the number of higher priority tasks released in the interval $[t, t + D_i)$. Larger relative deadlines could increase D_i. All the changes should make the feasibility condition becomes easier to be satisfied.

Furthermore, we can have the following analysis about the predictability of these parameter changes:

- Decreased capacity comes from the deviation in theoretical analysis and practical execution. A task can execute shorter than its computed capacity (WCET). This change is not predictable. This is a practical problem that scheduling simulation tools have to take into account. If scheduling simulation with the WCETs of tasks is not sustainable regarding this change, we need to perform simulation with all possible values which are smaller than the WCETs of tasks, leading to an exponential complexity.

- Regarding periodic tasks, a larger period is a predictable change because the period can only be set by system designer. Regarding sporadic tasks, the period of a task is only the minimum interarrival time (MIT). In systems with sporadic tasks, we can consider that the change in period parameter always happens. If we take into account sporadic tasks, sustainability analysis with regard to the period parameter is more critical than when we only take into account periodic tasks.

- Larger relative deadline is a predictable change because the deadline is set by system designers.

As presented in Section 1.5, we assumed that the periods and the relative deadlines of tasks are statically assigned by system designer and thus cannot be dynamically computed. The case of dynamically computed periods and deadlines are beyond the scope of this thesis.

To sum up, a schedulability test, such as scheduling simulation or WCRT analysis, must be aware of unpredictable changes in task parameters even if these are considered better behaviors. A schedulability test must be sustainable regarding capacity parameter in order to be used to verify the schedulability of task sets with only periodic tasks. It must be sustainable regarding both capacity and period parameters in order to be used to verify the schedulability of task sets with sporadic tasks.

4.3.2 *CRPD problem in sustainability analysis*

The problem related to CRPD in sustainability analysis in FPP scheduling context can be defined as follows. As presented in the previous section, the two parameter changes (1) decreased capacities and (2) larger periods and could decrease I_i^t by Δ. However, as shown by examples in the next sections, the two changes can increase the number of preemptions. Thus, despite of the decrease in execution requirement, there is an increase in CRPD by γ. If $\gamma > \Delta$, parameter changes, which are considered a better scenario, increase the interference and could lead to unschedulable system.

In the next sections, we remind that with CT-CRPD and FS-CRPD, scheduling simulation with CRPD is not sustainable with regard to capacity and period parameters. However, we show that FSC-CRPD is sustainable with regard to capacity parameter but not sustainable with regard to period parameter.

4.3.3 *Sustainability analysis of scheduling simulation with classical CRPD computation models*

CT-CRPD

In [67], the authors have investigated the sustainability of scheduling simulation with CT-CRPD computation model.

Theorem 6 ([67])**.** *Scheduling simulation with CT-CRPD is not sustainable with regard to the capacity parameter.*

Theorem 7 ([67])**.** *Scheduling simulation with CT-CRPD is not sustainable with regard to the period parameter.*

Several counter examples have been shown to prove that a schedulable task set does not remain schedulable when a better change in capacity or period parameter occurs.

FS-CRPD

We prove two theorems concerning the sustainability of scheduling simulation with FS-CRPD computation model. The first theorem is related to the capacity parameter.

Theorem 8. *Scheduling simulation with FS-CRPD is not sustainable with regard to the capacity parameter.*

Proof. We prove this theorem by using a counter example. In this example, a task set is schedulable with CRPD taken into account. When the capacity of a task is decreased, this task set becomes not schedulable.

Task	C_i	T_i	D_i	O_i	Π_i	**UCB_i**	**ECB_i**
τ_1	4	12	12	0	3	$\emptyset$	$\{1,2\}$
τ_2	8	24	24	0	2	$\{3\}$	$\{3,4\}$
τ_3	8	24	24	0	1	$\{1,2\}$	$\{1,2\}$

Table 9: Task set example
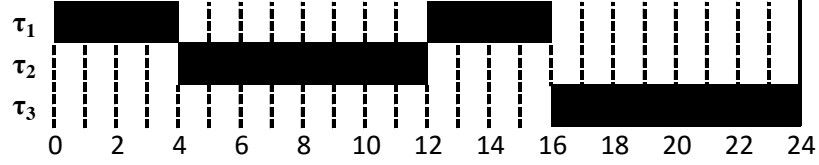
Figure 25: Scheduling simulation of task set in Table 9 in the first 24 units of time. All deadlines are met. There is no preemption.

A task set is provided in Table 9. In Fig. 25, we have the scheduling simulation of this task set in the first 24 units of time. All deadlines are met.

Regarding the job of task τ_3 released at $t = 0$, it experiences the interference from higher priority tasks τ_1 and τ_2. The feasibility condition is satisfied as we have:

1   $C_3 = 8, I_3^0 = 16$
2   $\rightarrow C_3 + I_3^0 = 8 + 16 \leqslant 24$
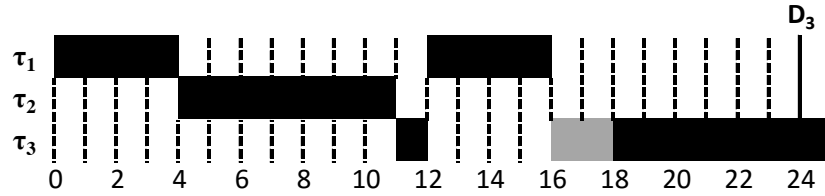
Figure 26: Non-sustainable scheduling simulation regarding capacity parameter with FS-CRPD computation model. The capacity of τ_2 is reduced to $7 < C_2 = 8$. τ_1 preempts τ_3 at time $t = 12$

In Fig. 26, we assume that the capacity of τ_2 is reduced to 7 instead of $C_2 = 8$. Because of this change, the job of τ_2 is completed at time $t = 11$. Then, τ_3 can start at time $t = 11$ and then be preempted by τ_1 at time $t = 12$. Later, τ_3 resumes at time $t = 16$. Regarding FS-CRPD computation model, the CRPD added to the capacity of τ_3 at time $t = 16$ is computed as follows:

1   t = 11: $UCB_3' = UCB_3 = \{1,2\}$
2   t = 12: $UCB_3' = UCB_3' - ECB_1 = \{1,2\} - \{1,2\} = \emptyset$
3   t = 16: $\gamma_3^{16} = |UCB_3 - UCB_3'| \cdot BRT = |\{1,2\} - \emptyset| \cdot 1 = \mathbf{2}$

The CRPD computed is 2 units of time and τ_3 missed its deadline. □

Theorem 9. *Scheduling simulation with FS-CRPD is not sustainable with regard to the period parameter.*

Proof. We prove this theorem by using a counter example. In this example, a task set is schedulable with CRPD taken into account. When the period of a task is larger, this task set is not schedulable.

We use the task set provided in Table 9. As shown in Fig. 25, this task set is schedulable.

Figure 27: Non-sustainable scheduling simulation regarding the period parameter with FS-CRPD computation model. The period of τ_1 is increased to $13 > T_1 = 12$. τ_1 preempts τ_3 at time $t = 13$. τ_3 missed its deadline at time $t = 24$.

In Fig. 27, we assume that the period of τ_1 is increased to 13 instead of $T_1 = 12$. Because of this change, τ_1 is not released at time $t = 12$. As a result, at time $t = 12$, τ_3 can execute. At time $t = 13$, τ_1 is released and preempts τ_3. Regarding FS-CRPD computation model, we have:

$$_1 \quad t = 13: \ UCB_3' = UCB_3' - ECB_1 = \{1,2\} - \{1,2\} = \emptyset$$
$$_2 \quad t = 17: \ \gamma_3^{17} = |UCB_3 - UCB_3'| \cdot BRT = |\{1,2\} - \emptyset| \cdot 1 = \mathbf{2}$$

We can see that later, τ_3 missed its deadline at time 24. □

4.3.4 *Sustainability analysis of FSC-CRPD*

In this section, we prove the sustainability of scheduling simulation with FSC-CRPD computation model regarding each task parameter change defined in Definition 71, section 4.3.

A *Decreased capacity*

We prove that scheduling simulation with FSC-CRPD computation model is sustainable regarding the capacity parameter.

Theorem 10. *Assuming FSC-CRPD computation model, a decrease of Δ in execution time of higher priority tasks can only lead to a maximum increase of γ execution time of the lower priority task where $\gamma \leqslant \Delta$.*

Proof. A decrease of Δ in execution time of higher priority tasks could cause a lower priority task τ_i executes Δ sooner and be preempted. Thus, there is an increase in the number of preemptions and the execution time of τ_i is increased by γ.

Suppose that $\gamma > \Delta$, this can only occur if:

$$\min(|UCB_i - UCB_i'|, \rho_i) \cdot BRT > \Delta$$

It means that two conditions must be satisfied:

$$\begin{cases} \rho_i \cdot BRT > \Delta \\ |UCB_i - UCB_i'| \cdot BRT > \Delta \end{cases}$$

The number of additional UCBs loaded into the cache thanks to a decrease of Δ in execution time is ρ_i. The condition $\rho_i \cdot BRT > \Delta$ above cannot hold following Theorem 5. $\qquad\square$

We now prove that a decrease in execution time of higher priority task does not create additional interference to lower priority task. The decrease in execution time is always larger than or equal to the CRPD introduced by the possible increase in the number of preemption.

Theorem 11. *Scheduling simulation with FSC-CRPD computation model is sustainable with regard to the capacity parameter.*

Proof. Suppose that a system is deemed schedulable; i.e., for all jobs of all tasks, the feasibility condition defined in Equation 38 is satisfied.

We evaluate a job of task τ_i following the feasibility condition. A decrease in capacity of τ_i means that it has a new capacity $C_i' \leqslant C_i$.

A decrease in capacity of higher priority task can introduce a new interference denoted as $I_i'^t$. We have $I_i'^t = I_i^t - \Delta + \gamma$, where Δ is the decrease in capacity and γ is the CRPD introduced by the change. We have $\gamma \leqslant \Delta$ according to Theorem 10. Thus, $I_i'^t \leqslant I_i^t$. To conclude, we have the following equation.

$$C_i' + I_i'^t \leqslant C_i + I_i^t \leqslant D_i \tag{39}$$

We conclude that a job of task τ_i still feasible when experiencing a decrease in execution time. $\qquad\square$

We apply FSC-CRPD computation model to the example presented in Figure 26 in which the capacity of τ_2 is reduced to $7 < C_2 = 8$. τ_1 preempts τ_3 at time $t = 12$. Regarding FSC-CRPD computation model, when taking into account ρ_3 parameter, we have:

---

1 $\texttt{t = 12:}$ $UCB_3' = UCB_3' - ECB_1 = \{1,2\} - \{1,2\} = \emptyset$
2 $\texttt{t = 12:}$ $\rho_3 = 1$
3 $\texttt{t = 16:}$ $\gamma_3^{16} = \min(|UCB_3 - UCB_3'|, \rho_3) \cdot BRT = \mathbf{1}$
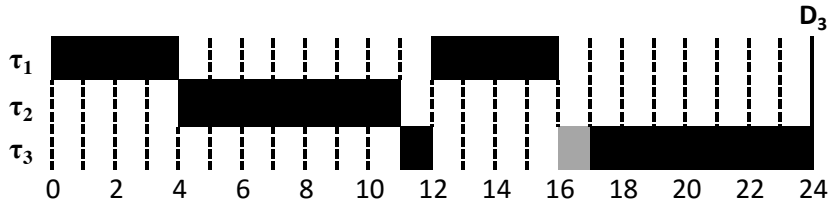
Figure 28: Sustainable scheduling simulation regarding capacity parameter with FSC-CRPD computation model. The capacity of τ_2 is $7 < C_2 = 8$. τ_1 preempts τ_3 at time $t = 12$

The CRPD computed is 1 unit of time and τ_3 can meet its deadline as illustrated in Figure 28.

From this example, FSC-CRPD computation model is not only less pessimistic, but scheduling simulation of this model is also sustainable regarding execution time parameter.

In conclusion, we have investigated and proved the sustainability of scheduling simulation with FSC-CRPD computation model regarding the capacity parameter.

In the next section, we prove that scheduling simulation with FSC-CRPD computation model is not sustainable regarding the period parameter.

B *Larger Period*

Theorem 12. *Scheduling simulation with FSC-CRPD computation model is not sustainable with regard to the period parameter.*

Proof. We prove this theorem by using a counter example. Changing period of tasks can lead to unschedulable task sets when CRPD is considered. This problem is illustrated in Figure 29.
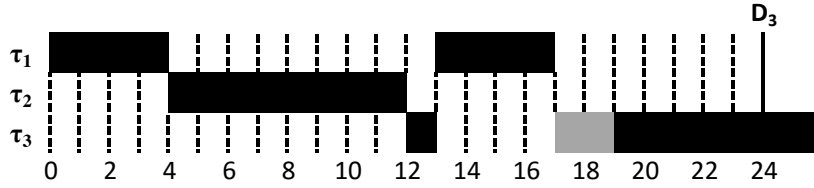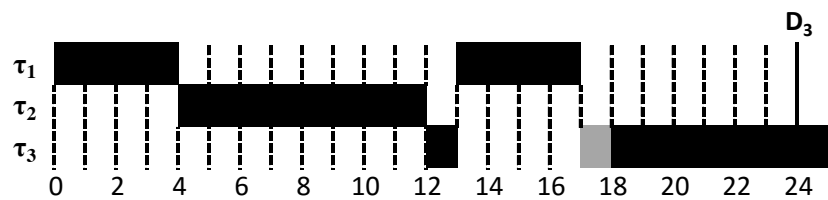

Figure 29: Non-sustainable scheduling simulation regarding period parameter with FSC-CRPD computation model. The period of τ_1 is increased to $13 > T_1 = 12$. τ_1 preempts τ_3 at time $t = 13$. τ_3 missed its deadline at time $t = 24$.

In this figure, the period of task τ_1 is changed to 13. As a result, at time $t = 12$, τ_3 can execute. At time $t = 13$, τ_3 is preempted by τ_1 and there is one unit of preempted cost added to the capacity of τ_3. Finally, τ_3 missed the deadline at time $t = 24$. $\square$

We can observe that the change in the period of τ_1 does not decrease the interference from higher priority tasks to the job of τ_3 released at $t = 0$, even if CRPD is not considered. Furthermore, it also creates one additional preemption.

C *Larger Relative Deadline*

In fixed priority preemptive scheduling, an increase in relative deadlines is simply a better timing constraint if we do not reassign task priorities. In this case, larger deadlines neither decrease execution time of tasks nor create additional preemptions.

Theorem 13. *Scheduling simulation with FSC-CRPD computation model is sustainable with regard to the deadline parameter.*

It is important to mention that we do not investigate the case where task priorities are reassigned according to new deadlines.

To sum up, in this section, we have investigated the sustainability analysis of scheduling simulation with FSC-CRPD computation model regarding the three task parameter changes: capacity, period and relative deadline. We have proved that scheduling simulation with FSC-CRPD is sustainable regarding capacity and relative deadline parameter and is not sustainable regarding period parameter. The result means that scheduling simulation with FSC-CRPD is an improvement comparing to FS-CRPD and CT-CRPD. It can be used to verify and guarantee the schedulability of periodic tasks where the changes in the period parameter are predictable and sustainability regarding this parameter is not an issue. However, it cannot be applied to task set with sporadic tasks.

In the next section, we discuss about the feasibility interval.

4.4 FEASIBILITY INTERVAL ANALYSIS

In this section, we present our analysis on the feasibility interval of the system model presented in section 4.2 regarding FSC-CRPD computation model. We analyze two properties that are used to establish the feasibility interval in previous literature [7, 43]: stabilization time and periodicity. In FPP scheduling context, a well established result on these properties regarding RTES without cache memory is that for a task τ_i, after an initial stabilization time S_i, the execution of τ_i is periodic in the interval P_i. Then, the feasibility interval of τ_i is $[0, S_i + P_i)$.

To determine the feasibility interval of our system model, we investigate the stabilization time when CRPD is taken into account. Second, we prove the periodic behavior and establish the feasibility interval.

4.4.1 Stabilization Time

For asynchronous systems, the concept of stabilization time was introduced in [7] and [43]. In these systems, there could be an interval of time, in which lower priority tasks are released and executed while higher priority tasks are not released. In this interval, a system is considered to be not stabilized. Stabilization time is defined as follows:

Definition 72 (Stabilization time [7, 43]). *Stabilization time S_i of a task τ_i is an instant at a release time of τ_i when all tasks $\tau_j \in hp(i)$ are released and stabilized.*

The computation of S_i is inductively defined by [43]:
$S_1 = O_1$,
$S_i = \max(O_i, O_i + \lceil \frac{S_{i-1} - O_i}{T_i} \rceil \cdot T_i)$ $(i = 2, 3, ..., n)$.
The main idea of the initial stabilization time is for a job $\tau_i[t], 0 < t \leqslant S_i$, not all higher priority tasks are released so the interference to $\tau_i[t]$ is lower than $\tau_i[t + k \cdot P_i], k \in \mathbf{N}^*$. In other words, the execution of $\tau_i[t], 0 < t \leqslant S_i$ could not be repeated in the future.

The stabilization time proposed in [7, 43] can be applied to systems with cache as the computation of stabilization time only needs to take into account the offsets and the periods of tasks. CRPD is a factor which affects the execution of tasks, not the release time of tasks and the stabilization time.

In [43], Goossens and Devillers has proved that for systems without cache, for a task τ_i, the interference from higher priority tasks to $\tau_i[t], t \leqslant S_i$ is less than or equal to the interference from higher priority tasks to $\tau_i[t'], t' \in [S_i, S_i + P_i)$. We recall that P_i is level i hyper period of task τ_i, which is defined in Definition 40. It was concluded that the execution before S_i is only needed to lead τ_i to its periodic behavior after S_i.

For synchronous systems, all tasks are released at the same time. It does not exist an interval of time, in which lower priority tasks are released and executed while higher priority tasks are not released. Thus, stabilization times of all tasks are equal to 0.

4.4.2 Periodic Behavior

In this section, we analyze the periodic behavior of systems with cache after the initial stabilization time. First, we revise the proof of periodic behavior of RTES without cache in the previous literature.

Feasibility interval proof by Audsley [7]

In [7], Audsley proved that after the initial stabilization time S_i, the execution of a task τ_i at time t, denoted as $E(\tau_j, t)$ where $t \geqslant S_j$, implies the execution of τ_i

at time $t + k \cdot P_i$. In other words, if τ_i is executed on the processor at time t, it is also executed on the processor at time $t + k \cdot P_i$.

Theorem 14 ([7]). *For all task τ_i, the execution of τ_i at time t, denoted $E(\tau_i, t)$ where $t \geqslant S_i$, implies $E(\tau_i, t + k \cdot P_i)$.*

Proof. Consider the highest priority task τ_1, it executes for the first C_i units of time in any interval $[O_1 + k \cdot T_1, O_1 + k \cdot T_1 + D_1), k \in \mathbb{N}$. Therefore, the behavior of τ_1 is static, in that for every time t_1 that the task executes, it will also execute at $t_1 + P_1$. Hence,

$E(\tau_1, t_1) \implies E(\tau_1, t_1 + k \cdot P_1), k \in \mathbb{N}, t_1 \geqslant S_1$

The behavior of τ_2 can be expressed in a similar manner. After the initial stabilization time, τ_2 executes in the first C_2 time units in the interval $[O_2 + k \cdot T_2, O_2 + k \cdot T_2 + D_2), k \in \mathbb{N}$ which will not be used by any higher priority task, namely τ_1. Therefore, since the times that τ_1 executes are already determined, and τ_1 has been released, we can assert:

$E(\tau_2, t_2) \implies E(\tau_2, t_2 + k \cdot P_2), k \in \mathbb{N}, t_2 \geqslant S_2$

The argument can be combined until τ_i is reached. This task will reserve the first C_i units of computation time that are not required by any higher priority task. Thus,

$E(\tau_i, t_i) \implies E(\tau_i, t_i + k \cdot P_i), k \in \mathbb{N}, t_i \geqslant S_i$

Therefore, we have built up the static requirements of all tasks, assuming all higher priority tasks have been released after the initial stabilization time. This assumption is indeed true and was also proved in [7]. $\qquad\square$

This inductive proof of Theorem 14 cannot be applied straight forward to systems with cache. We start with the cases of task τ_1 and τ_2. Because, τ_1 is the highest priority task and is not affected by CRPD, the following assertion still holds:

$E(\tau_1, t_1) \implies E(\tau_1, t_1 + k \cdot P_1), k \in \mathbb{N}, t_1 \geqslant S_1$

In the next step regarding τ_2, the following argument is not applicable: "*After the initial stabilization time, τ_2 executes in the first C_2 time units in the interval $[O_2 + k \cdot T_2, O_1 + k \cdot T_2 + D_2), k \in \mathbb{N}$ which will not be used by any higher priority task, namely τ_1*". The reason is that if τ_2 is preempted by τ_1, it will execute in C_2 time units plus the CRPD.

Feasibility interval proof for our system model with FSC-CRPD

We choose a different approach to prove the periodicity of task τ_i after the initial stabilization time S_i. We make an initial observation that the execution of an individual task in a FPP scheduling context depends only upon its own properties and higher priority tasks [7]. Thus, we define two conditions that make the execution of τ_i periodic:

- The first condition is that τ_i is released periodically in a fixed interval. This condition is satisfied in our system model because we only take into account periodic tasks.

- The second condition is that the interference from higher priority tasks to τ_i is periodic in a fixed interval. We proceed by proving that the second condition is also satisfied when CRPD is taken into account.

Based on the two conditions, if we can prove that the job $\tau_i[t_i], t_i = O_i + m \cdot T_i, m \in \mathbb{N}|t_i \geqslant S_i$ and the job $\tau_i[t_i + k \cdot P_i], k \in \mathbb{N}$ experience identical interferences from higher priority task, we can conclude that τ_i is periodic in interval P_i after the initial stabilization time S_i. We establish the following theorem:

Theorem 15. *For all task τ_i, the job $\tau_i[t_i], t_i = O_i + m \cdot T_i, m \in \mathbb{N}|t_i \geqslant S_i$ and the job $\tau_i[t_i + k \cdot P_i], k \in \mathbb{N}$ experience identical interferences from higher priority tasks $\tau_0, ..., \tau_{i-1}$.*

Proof. This theorem is proved by induction.

Trivial case

Consider the highest priority task τ_1, it always experiences 0 interference. Thus, the schedule of τ_1 is periodic from S_1 with the period $P_1 = T_1$.

Consider the second highest priority task τ_2, since the task is ordered by priority, the periodicity of τ_1 cannot be changed by τ_2. Because of the schedule of task τ_1 is periodic from S_1 with the period $P_1 = T_1$, τ_1 is also periodic from S_2 $(S_2 \geqslant S_1)$ with the period $P_2 = \text{lcm}\{P_1, T_2\}$.

The interference created by the capacity of τ_1 to the two jobs $\tau_2[t_2](t_2 = O_2 + m \cdot T_2, t_2 \geqslant S_2)$ and $\tau_2[t_2 + k \cdot P_2]$ $(k \in \mathbb{N})$ is periodic and identical as we assumed that task capacity is constant.

The two jobs $\tau_2[t_2]$ and $\tau_2[t_2 + k \cdot P_2]$ experience identical sequence of preempting tasks. If $\tau_2[t_2]$ is firstly preempted by τ_1 at time $t_2 + \Delta$, then $\tau_2[t_2 + k \cdot P_2]$ will be firstly preempted by τ_1 at time $(t_2 + k \cdot P_2) + \Delta, 0 \leqslant \Delta < D_2$. Because the sets of UCBs and ECBs of τ_1 and τ_2 are fixed and both jobs of τ_2 have executed Δ units of time, the CRPD of two preemptions are identical. The same argument can be applied to subsequent preemptions by τ_1 to $\tau_2[t_2]$ and $\tau_2[t_2 + k \cdot P_2]$ if they exist. We can conclude that the CRPD by τ_1 that $\tau_2[t_2]$ and $\tau_2[t_2 + k \cdot P_2]$ experience are identical.

From the two deductions, $\tau_2[t_2]$ and $\tau_2[t_2 + k \cdot P_2]$ experience identical interference.

Induction step

We assume that this theorem is true for $\tau_1, ..., \tau_i$. The objective now is to prove that it is also true for τ_{i+1}.

From the assumption, the schedule of the task subset $\{\tau_1, ..., \tau_i\}$ is periodic from S_i with the period of P_i. Since the task is ordered by priority, the periodicity of the task subset cannot be changed by τ_{i+1}. Hence, we can deduce that the

schedule of the task subset is also periodic from S_{i+1} ($S_{i+1} \geqslant S_i$) with the period $P_{i+1} = \text{lcm}\{P_i, T_{i+1}\}$ is identical. We can have the following deductions.

The interference created by the capacity of $\tau_1, ..., \tau_i$ to the two jobs $\tau_{i+1}[t_{i+1}]$ ($t_{i+1} = O_{i+1} + m \cdot T_{i+1}, t_{i+1} \geqslant S_{i+1}$) and $\tau_{i+1}[t_{i+1} + k \cdot P_{i+1}]$ ($k \in \mathbb{N}$) are identical.

The CRPD created by $\tau_0, ..., \tau_i$ preempting each other to the two jobs $\tau_{i+1}[t_{i+1}]$ and $\tau_{i+1}[t_{i+1} + k \cdot P_{i+1}]$ are identical.

The two jobs $\tau_{i+1}[t_{i+1}]$ and $\tau_{i+1}[t_{i+1} + k \cdot P_{i+1}]$ experience identical sequence of preempting tasks. Thus, the CRPD created by $\tau_0, ... \tau_i$ preempting $\tau_{i+1}[t_{i+1}]$ and $\tau_{i+1}[t_{i+1} + k \cdot P_{i+1}]$ is identical.

From these deductions, we can conclude that $\tau_{i+1}[t_{i+1}]$ and $\tau_{i+1}[t_{i+1} + k \cdot P_{i+1}]$ experience identical interference. $\qquad\square$

As the theorem regarding periodic behavior is proved, we can now prove the following theorem about the feasibility interval.

Theorem 16. *A task τ_i is feasible if and only if the deadlines corresponding to the releases of the task in $[0, S_i + P_i)$ are met.*

Proof. From Theorem 15, we deduce that the execution of $\tau_1, \tau_2, ..., \tau_i$ in the interval $[S_i, S_i + P_i)$ and $[S_i + k \cdot P_i, S_i + (k+1) \cdot P_i), k \in \mathbb{N}^*$ are identical. Thus, it is sufficient to check if τ_i can meet its deadlines in only one interval of time plus the interval $[0, S_i)$. $\qquad\square$

From Theorem 16, we can conclude that for a task set of n periodic tasks, the feasibility interval is $[0, S_n + P_n)$.

4.5 CONCLUSIONS

In this chapter, we investigate the problems related to scheduling simulation of RTES with cache memory by taking into account CRPD. Several assumptions are taken regarding system model, task execution and cache access profile.

We investigate classical CRPD computation models used in scheduling simulation and present existing issues regarding the pessimism of these models. Then, we discuss about the sustainability of scheduling simulation with classical CRPD computation models. We explain the problem related to CRPD in sustainability analysis and the reason why CRPD-aware scheduling simulation is not sustainable in general cases.

We propose a new CRPD computation model named FSC-CRPD to address the previous issues. In this model, based on an observation from real system execution presented in [59], we take a new assumption that bounds the CRPD by the executed capacity of a task. When this assumption holds, scheduling simulation

is less pessimistic and then becomes sustainable with regard to the capacity parameter. The conclusion about the sustainability of scheduling simulation with FSC-CRPD allows us to prove the feasibility interval of our system model.

The established results allow the use of CRPD-aware scheduling simulation as a verification method to evaluate the schedulability of periodic tasks. In addition, this work gives perspectives about in which cases CRPD-aware scheduling simulation is sustainable and is not.

We have not yet investigated the problem of CRPD-aware scheduling simulation for tasks with arbitrary deadline. In this case, modeling cache accesses and evaluating the number of UCB loaded into the cache of a task could be complex because there are multiple jobs of this task are released and executed.

Chapter 5

CACHE-AWARE SCHEDULING ANALYSIS TOOL IMPLEMENTATION

Contents

In this thesis, we have presented the following CRPD analysis: cache access profile computation [52, 23], CRPD analysis for WCRT [23, 52, 82, 4], CRPD-aware scheduling simulation [29] and limiting CRPD [86, 14, 58]. The work in this thesis has focused in CRPD-aware scheduling simulation and we proposed a CRPD-aware priority assignment.

The result obtained by cache access profile computation is required to perform the analysis of the other subjects. In addition, the analysis of each subject is done based on one or several parameters of a given RTES system and there are parameters that are shared amongst these subjects. A parameter can be either a system configuration or a scheduling parameter. There are seven parameters thats are involved in CRPD analysis for RTES with cache memory: (1) cache configuration, (2) memory layout, (3) task control flow graph, (4) capacity - WCET, (5) period, (6) deadline and (7) scheduling policy. An example of a shared parameter is that task period is used by both CRPD analysis for WCRT and CRPD-aware scheduling simulation. In Figure 30, a big picture of CRPD analysis subjects, parameters and their relationship is provided.
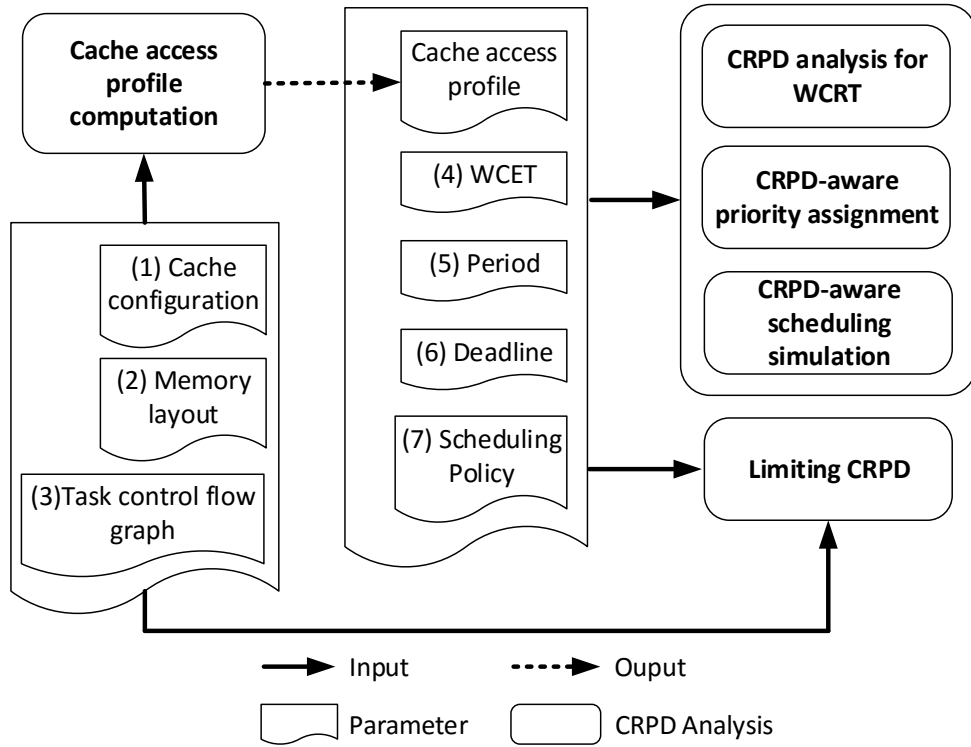
Figure 30: CRPD analysis subjects and parameters

The relationship between CRPD analysis subjects and shared parameters motivates the implementation of a scheduling analysis tool that take all of them into account.

The problem statement of this chapter can be summarized as follows. CRPD analysis for RTES with cache memory in FPP scheduling context consists in several subjects that are related to each other. Despite of the relationship, the proposed solution or analysis technique in each subject is evaluated individually. Thus, dependencies amongst those subjects are not investigated. As far as we know, there are no scheduling analysis tools that address the whole problem that can be used to study the dependencies amongst the subjects in the state-of-the-art work.

In this chapter, we present the implementation of several CRPD analysis methods for RTES with cache memory in a scheduling analysis tool. Implementation is made in Cheddar [75], an open-source scheduling analyzer, which is freely available to researchers and practitioners. Experiments are conducted in order to illustrate applicability and performance of our implementation. Furthermore, we discuss about implementation issues, problems raised and lessons learned from those experiments.

The rest of the chapter is organized as follows. Section 5.1 provides an overview of our approach and the implemented CRPD analysis methods. Those analysis

methods are implemented in Cheddar - an open source real-time scheduling analysis tool. Section 5.2 presents the Cheddar framework and the development process of a new analysis feature in Cheddar scheduling analyzer. In section 5.3, 5.4, 5.5 and 5.6, we present in detail the implementation of each CRPD analysis method following the presented development process. In section 5.7, we discuss several implementation issues that we identified during the implementation of CRPD analysis features in Cheddar. Finally, section 5.8 concludes the chapter.

5.1 CRPD ANALYSIS IMPLEMENTED IN CHEDDAR

We implemented the following CRPD analysis methods in Cheddar scheduling analyzer.

- Cache access profile computation based on the notion of UCB and ECB which is presented in Section 2.4.

- CRPD analysis for WCRT that is presented in Section 2.5.1. We implemented the following analysis methods: ECB-Only [23], UCB-Only [52], UCB-Union [79], ECB-Union [4], UCB-Union Multiset [4], ECB-Union Multiset [4] and Combined Multiset [4].

- Our proposed CRPD-aware priority assignment algorithm presented in Chapter 3.

- CRPD-aware scheduling simulation with our proposed CRPD computation model presented in Chapter 4.

We continue by presenting the Cheddar framework and explaining how this framework was extended.

5.2 CHEDDAR FRAMEWORK

Cheddar framework consists of three parts, which are depicted in Figure 31: (1) Cheddar architecture description language (ADL), (2) meta-workbench Platypus and (3) Cheddar scheduling analyzer.

1. Cheddar ADL is a simple architecture description language devoted to real-time scheduling theory. An ADL provides the abstraction of *components*, *connections* and *deployments*. A component is an entity modeling a part of the system. ADLs allow the specification of both hardware parts and software parts of the system with dedicated kinds of components. Connections usually model relationships between components and finally, deployments specify how software components are deployed on hardware components,
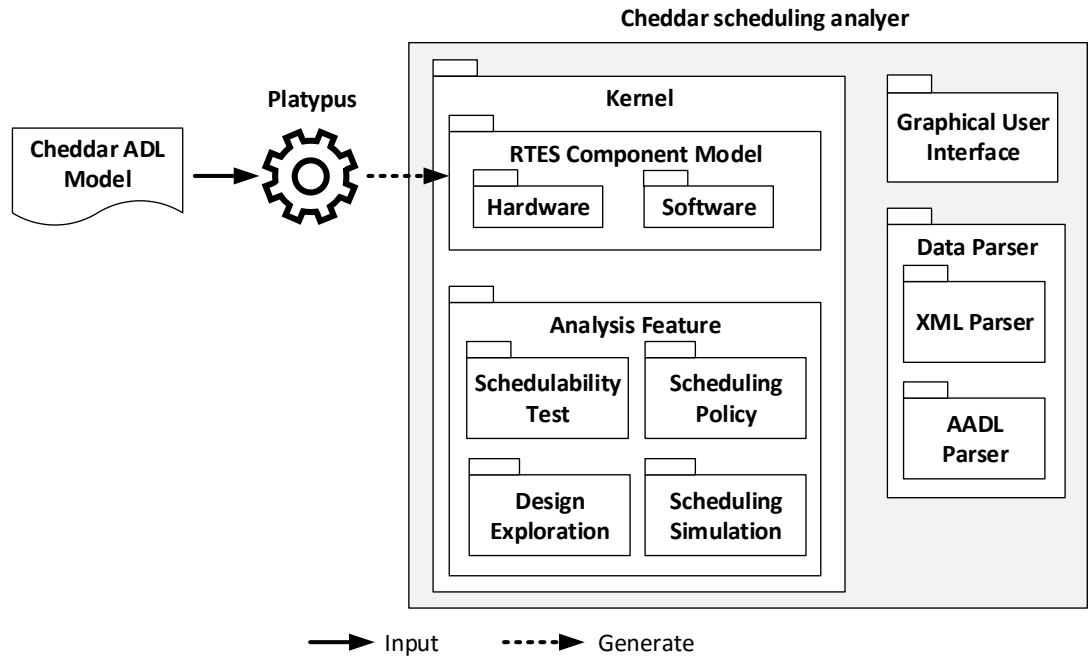
Figure 31: Cheddar Framework

i.e. how the resources of the system are shared. A Cheddar ADL meta-model is specified with the general purpose modeling language EXPRESS [88].

2. The meta-workbench Platypus [70] is used to implement the code generators. A part of Cheddar is automatically generated from its meta-models through a model driven engineering (MDE) process.

3. Cheddar scheduling analyzer includes three parts: kernel, graphical user interface (GUI) and data parsers.

  • The kernel consists in RTES component models and analysis features. RTES component models provide an abstraction of a system including its hardware and software components. It includes Ada class files that are automatically generated by the meta-workbench Platypus [70]. Several analysis features are implemented in the kernel. However, regarding the scope of this thesis, we only focus on schedulability test and scheduling simulation analysis features.

    Cheddar kernel can be called alone and embedded in a toolset. The framework is embedded in specific tool sets such as AADLInspector [36] and TASTE (ESA) (http://taste.tuxfamily.org). Cheddar was used to automate the computation of task WCRT in an architecture model refinement approach [17] implemented in RAMSES [26].

- The GUI can be used by the users to design a system model, apply analysis methods and receive results.

- The data parser supports importing and exporting a RTES architecture model in Cheddar ADL or AADL.

In the next sections, detailed information about parts in Cheddar framework is presented.

- Section 5.2.1 presents Cheddar ADL and the process of using the meta-workbench Platypus to generate RTES component model in the kernel.

- Section 5.2.2 introduces two analysis features of Cheddar scheduling analyzer: schedulability test and scheduling simulation.

- Section 5.2.3 presents the use process. It introduces how to use the Cheddar GUI to design a system model and how to import/export a model in Cheddar ADL.

- Section 5.2.4 shows the development process. It includes the process of extending Cheddar ADL to model a new RTES components model, generating Ada class files and implementing new analysis features.

5.2.1 *Cheddar ADL model of* RTES *components*

In this section, first, we provide a summary of RTES components that are supported by Cheddar ADL. These components are separated into hardware and software component. Second, we present the process of generating Ada class files of RTES component models in the kernel from Cheddar ADL.

Cheddar ADL model of hardware components
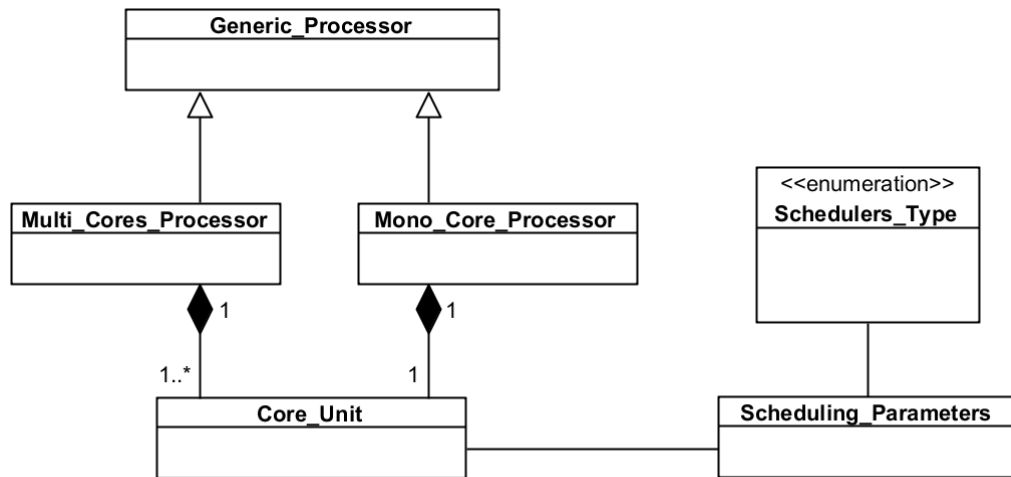
Figure 32: Cheddar ADL model of hardware component

Hardware components represent the resources provided by the environment. Cheddar ADL provides limited capabilities to model hardware components. Indeed, real-time scheduling theory usually assumes simple models of hardware. As shown in Figure 32, hardware components can be of two kinds:

- Core components model entities that provide a resource to sequentially run tasks. In Cheddar, scheduling parameters are attached to a core. An example of scheduling parameter is the scheduling policy used to schedule tasks on a core.

- Processor components are composed of sets of cores. A processor is either multi-cores or mono-core.

Cheddar ADL model of software components

Software components can be deployed on either core or processor components. Those deployments model two kinds of component connections that allow designers to express either global scheduling or partitioned scheduling. The design of the software part of a real-time system can be specified with five component types. These component types are depicted by Figure 33:

1. Address space components model a group of resources that can be accessed. They may be associated to an address protection mechanism.

2. Task components model flows of control. They are statically connected to address space components.
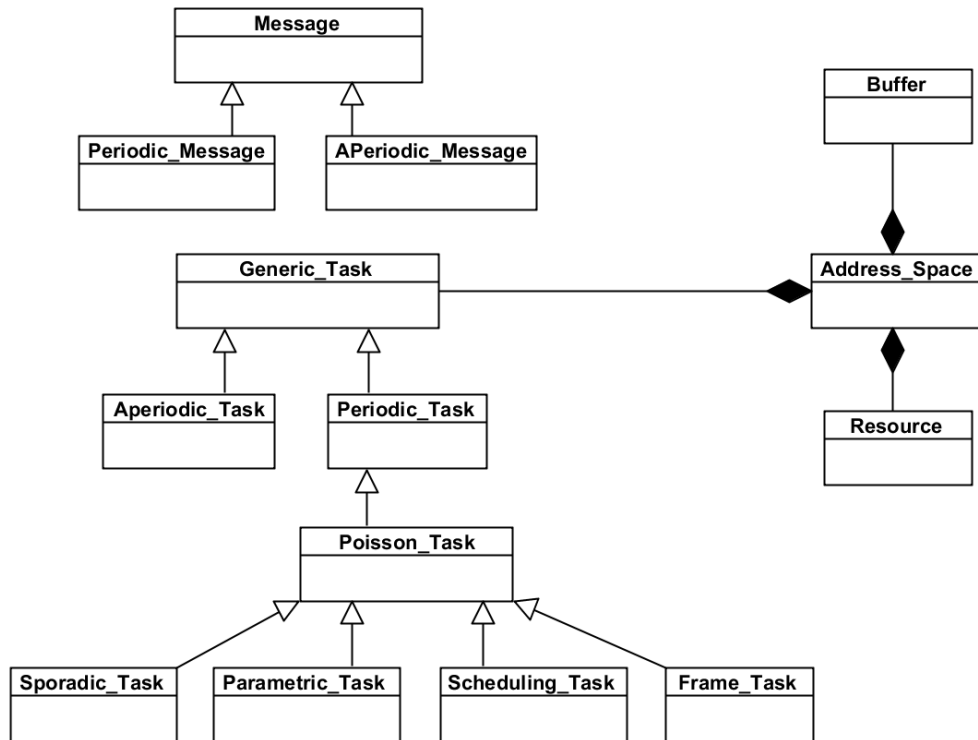
Figure 33: Cheddar ADL model of software component

3. Resource components may model any data structure, shared by tasks or not, synchronized or not. They may be accessed through classical priority inheritance protocols. They may model asynchronous communications between tasks located in the same address space.

4. Buffer components model queued asynchronous data exchanges between tasks located in the same address space.

5. Message components model queued asynchronous data exchanges between tasks located in different address spaces. Buffer, resource and message components specify types of connection between components, i.e types of dependencies between tasks.

We have presented Cheddar ADL model of software and hardware components. Regarding the implementation of CRPD analysis methods in Cheddar framework, Figure 34 sums up the requirements.

The four parameters: WCET, period, deadline and scheduling policy are already supported by Cheddar ADL. WCET, period and deadline are supported by the *Periodic_Task* model. Scheduling policy is supported by *Scheduling_Parameters* and *Scheduler_Type* model.

The three parameters: cache configuration, memory layout and task control flow graph are not supported. In addition, we also consider cache access profile

as an input parameter to be modeled in Cheddar ADL. The motivation of this decision is to facilitate the process of importing an existing cache access profile and applying CRPD analysis methods.
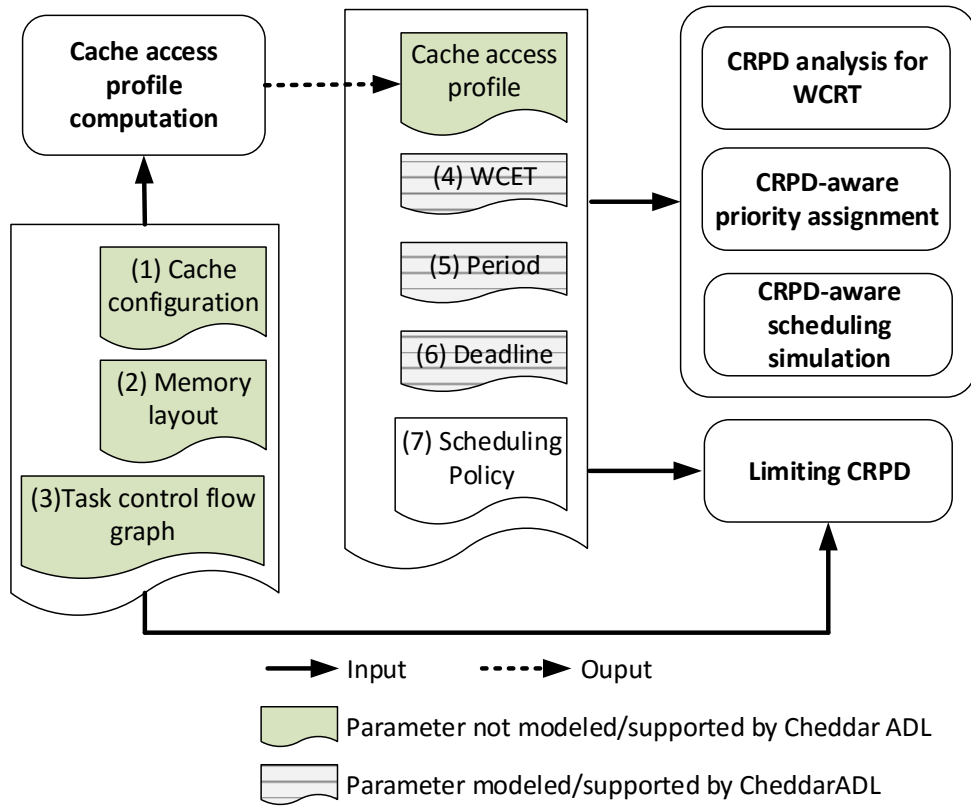
Figure 34: CRPD analysis and parameters

Next, we present how to generate Ada class files from Cheddar ADL.

Generating Ada class files from Cheddar ADL model

From a Cheddar ADL schema of a RTES component, Ada class files can be automatically generated by the meta-workbench Platypus [70] through a model-driven engineering process.

We provide an example to illustrate the process of generating Ada class files from a Cheddar ADL model of the processor component in Figure 35. From an EXPRESS schema, two Ada class files are generated. The ".ads" file is specification file and the ".adb" file is implementation file.
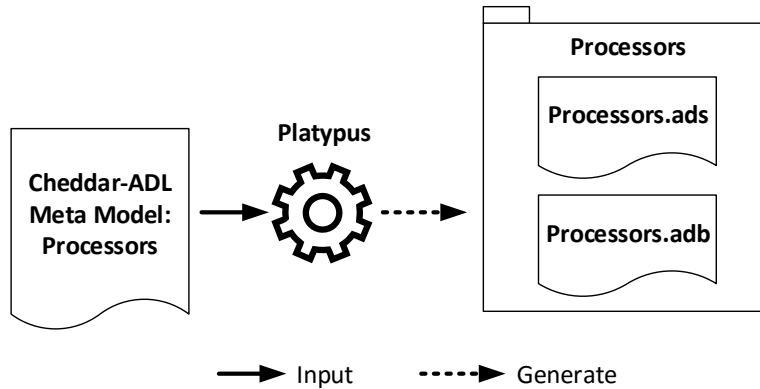
Figure 35: Generating Ada class files from Cheddar ADL of processor component.

In Listing 2, we provide an EXPRESS schema of the processor component. The schema is 21 Lines of Code (LoC). It is then used to generate two Ada class files: processors.ads (176 LoC) and processor.adb (429 LoC). These Ada class files include the specification of the entities and the functions and the procedures that can be used to access the entities. A part of the generated code in processor.ads file is provided in Listing 3.

```
1  SCHEMA Processors;
2    ENTITY Generic_Processor
3      SUBTYPE OF ( Named_Object );
4      network_name : STRING;
5      processor_type : Processors_type;
6      migration_type : migrations_type;
7    DERIVE
8      SELF\Generic_Object.object_type : Objects_Type := Processor_
         Object_Type;
9    END_ENTITY;
10
11   ENTITY Mono_Core_Processor
12     SUBTYPE OF ( Generic_Processor );
13     core : core_unit;
14   END_ENTITY;
15
16   ENTITY Multi_Cores_Processor
17     SUBTYPE OF ( Generic_Processor );
18     cores : Core_Units_Table;
19     l2_cache_system_name : STRING;
20   END_ENTITY;
```

```
21  END_SCHEMA;
```

Listing 2: EXPRESS schema of the processor component

```
1   type Generic_Processor is new Named_Object with
2   record
3  network_name : Unbounded_String;
4  processor_type : Processors_type;
5  migration_type : migrations_type;
6   end record;
7
8   procedure Initialize(obj : in out Generic_Processor);
9   procedure Put(obj : in Generic_Processor);
10  procedure Put(obj : in Generic_Processor_Ptr);
11  procedure Put_Name(obj : in Generic_Processor_Ptr);
12  procedure Build_Attributes_XML_String(obj : in Generic_Processor;
          result : in out Unbounded_String);
13  function XML_String(obj : in Generic_Processor) return Unbounded_
      String;
14  ...
```

Listing 3: Part of the generated code in processors.ads

In addition, the data parser in Cheddar scheduling analyzer is updated following the Cheddar ADL of processor component. Cheddar scheduling analyzer supports importing and exporting a Cheddar ADL model written in XML or AADL.

In the next section, we introduce the analysis features that are supported by Cheddar scheduling analyzer.

5.2.2 *Analysis features in Cheddar scheduling analyzer*

From a Cheddar ADL model, Cheddar scheduling analyzer provides various scheduling analysis features [40]. Scheduling analysis can be performed either with feasibility tests or with scheduling simulations on the feasibility interval. Cheddar scheduling analyzer implements classical methods of both verification techniques. In this section, we first introduce schedulability tests implemented into Cheddar scheduling analyzer and then, we present its scheduling simulation features.

Schedulability Test

Cheddar scheduling analyzer implements various feasibility tests. Processor utilization feasibility tests can be applied on other scheduling policies. Furthermore, WCRT can be computed on periodic tasks. Those WCRTs can integrate delays related to shared resources (i.e. shared resource blocking time). Finally, few feasibility tests for hierarchical architectures have also been implemented.

It is not possible to analyze all systems by feasibility tests, and some theoretical results are often known as being too pessimistic. That is why additional techniques such as simulation are introduced.

Scheduling Simulation

Several classical scheduling algorithms are implemented in Cheddar scheduling analyzer. Users may experiment classical schedulers such as RM, DM, EDF, LLF or POSIX 1003 policies, both preemptive and non preemptive. Those algorithms have been implemented in the context of uniprocessor scheduling and also in the context of global multiprocessor scheduling

Scheduling simulations can be run for usual task models such as periodic, aperiodic and sporadic. Tasks can be constrained by dependencies related to shared resources, precedence or communication task relationships.

From an architecture model, various performance criteria can be extracted from scheduling simulation: worst/best/average response time, probability distribution of response time, worst/best/average shared resource blocking time, number of context switch or preemption, deadlock, priority inversion or specific properties defined with a domain specific language.

Furthermore, specific schedulers or task models can also be specified with the help of the Cheddar ADL. Those specific schedulers allow users to extend the scheduling analysis capability without a deep understanding of Cheddar design and implementation. This feature allows users to quickly adapt the scheduling verification tool to their needs (i.e. implementing a scheduling method which does not exist yet in Cheddar).

In the next section, we present the process of using an analysis feature in Cheddar scheduling analyzer.

5.2.3 *Use Process*

The basic process of using Cheddar scheduling analyzer consists of three steps illustrated in Figure 36.

| 1. Design System Model | | 2. Call Analysis Features | | 3. Export Data |

A system model can be represented in:
- Cheddar ADL: XML
- Cheddar ADL: AADL
or designed with:
- Ada
- Cheddar GUI

Analysis features can be called by:
- Command-line interface.
- Cheddar GUI

Data can be exported in:
- Cheddar ADL: XML
- Cheddar ADL: AADL
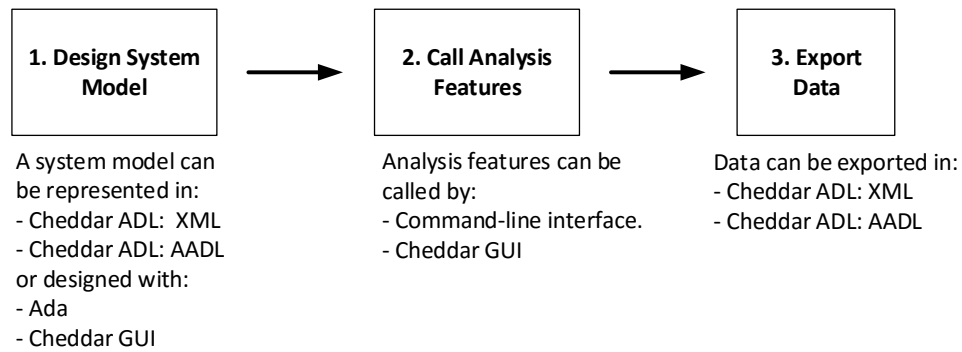
Figure 36: Cheddar scheduling analyzer use process

Step 1: Design system model

A system model consists of the initialization and deployments of RTES component models in Cheddar. As shown in Figure 36, Cheddar provides four methods to design a system model.

- *XML*: Cheddar supports importing and also exporting a RTES architecture model that is represented in XML.

- *AADL* : Cheddar supports importing and also exporting a RTES architecture model that is represented in AADL [38].

- *Ada*: Users can manually create a system model by writing Ada class files. However, the process could be tedious, error prone and not user-friendly.

- *Cheddar GUI*: Users can use the GUI to select, add, modify RTES components to a system model. In addition, a system model can be saved by exporting it in Cheddar ADL.

Step 2: Call analysis features

Analysis feature can be called by using a command line interface or Cheddar GUI.

Step 3: Export Data

Cheddar supports exporting a system model and scheduling simulation result in XML format. We provide an example of the exported system model that consists of a processor and a core unit in Listing 4.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <cheddar>
```

```
3   <core_units>
4     <core_unit id="id_66">
5      <object_type>CORE_OBJECT_TYPE</object_type>
6       ...
7     </core_unit>
8    </core_units>
9    <processors>
10    <mono_core_processor id="id_67">
11     <object_type>PROCESSOR_OBJECT_TYPE</object_type>
12     <name>CPU_01</name>
13     <processor_type>MONOCORE_TYPE</processor_type>
14     <migration_type>NO_MIGRATION_TYPE</migration_type>
15     <core ref="id_66">
16     </core>
17    </mono_core_processor>
18   </processors>
19 ...
20 </cheddar>
```

Listing 4: Cheddar ADL model of a processor in XML format

In the next section, we present the process of developing a new analysis feature in Cheddar framework.

5.2.4 *Development Process*

The process of developing an analysis feature for a new system model in Cheddar framework consists of three steps.

Step 1: Extending Cheddar ADL

Cheddar ADL needs to be extended when there is new hardware or software components that have to be taken into account during the analysis processes. From the updated Cheddar ADL, the meta-workbench Platypus is used to generate Ada class files.

Step 2: Implementing analysis feature

An analysis feature such as a schedulability test, a optimization algorithm or a priority assignment algorithm is added to Cheddar by programming Ada class files. In addition, Cheddar provides support for user-defined scheduler in Cheddar ADL. User can define a specific scheduler in Cheddar ADL, which is simpler

and less error prone than manually implementing in Ada, and the corresponding Ada code can be automatically produced and integrated in Cheddar.

Step 3: Updating Cheddar GUI and Data Parser

If a new analysis feature includes the use of a new RTES component model, the GUI should be updated so that users can create this component.

Furthermore, a new RTES component model requires an update to the data parser. Thanks to the model driven engineering process, functions and procedures that read and export a component with its attributes are automatically generated. However, the process of handling the data and attaching a component to a system must be manually implemented.

In the next sections, we present the process of implementing CRPD analysis features in Cheddar following the presented development process. For each analysis feature, first, we provide a specification that describes the following characteristics:

1. Purpose: the purpose of the analysis feature.

2. Input: the lists of input parameters required to perform the analysis.

3. Output: the result of the analysis.

4. Method: the theoretical method that the implementation of the analysis feature is based on.

Second, we show how the RTES component models in Cheddar are extended. Finally, we provide the implementation of the analysis feature in Ada. The following analysis features are implemented in Cheddar:

- Cache access profile computation.

- CRPD analysis for WCRT.

- CRPD-aware priority assignment algorithm

- CRPD-aware scheduling simulation.

5.3   CACHE ACCESS PROFILE COMPUTATION

In this section, we present the implementation of cache access profile computation in Cheddar framework. The specification of this analysis feature is described as follows:

Purpose	- Compute the cache access profile of a task. A cache access profiles is represented by a set of UCBs and a set of ECBs.
Input	- Cache configuration: cache size, line size and associativity. - Memory layout: the position of the data and instruction of a task in the main memory - CFG of a task. In the CFG, the size and position in main memory of each basic block and data used by each basic block are known
Output	- Computed cache access profile. A cache access profile is represented by a set of UCBs and a set of ECBs
Method	- The set of ECBs represents all the cache blocks used by a task [23]. This set is computed by taking into account the memory usage of a task, memory layout, and cache memory configuration including line size and associativity. - The set of UCBs represents the cache blocks that are reused by a program during its execution. This set is computed applying the UCB computation algorithm presented in [52]. The algorithm consists of two steps. First, it computes the cache blocks that are used by each basic block in the CFG of a program. This step requires information about the memory usage of each basic block and cache memory configuration. Second, a data flow analysis technique is applied in order to deduce the set of UCBs of each basic block.

Table 10: Summary of cache access profile computation

5.3.1 *Extending Cheddar ADL*

Implementation of cache access profile computation in Cheddar framework requires extending the Cheddar ADL. As shown in Figure 34 and the specification in Table 10, there are three input parameters required.

1. Cache configuration.

2. Memory layout.

3. Control flow graph.

Cache configuration is taken into account by extending Cheddar ADL model of hardware components. Memory layout and control flow graph are taken into account by extending Cheddar ADL model of software components. These extension are presented in two sections. For each section, we proceed by presenting how the new RTES component models are linked to existing RTES component

models in Cheddar ADL. Then, the specifications of new RTES component models in Cheddar ADL and the generated Ada classes are provided.

Extending Cheddar ADL model of hardware components
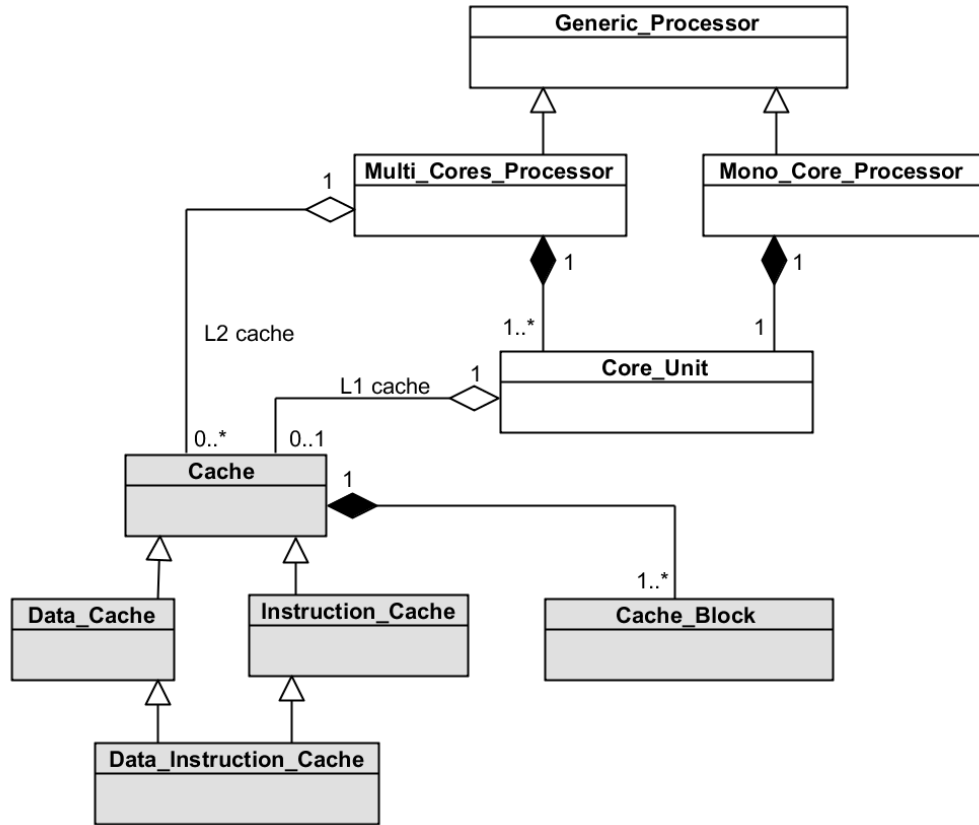
Figure 37: Extended Cheddar ADL model of hardware components

The extended Cheddar ADL for hardware component model is illustrated in Figure 37. Cache is linked to a core unit or a processor. A cache can belong to a single core unit (L1 cache) or shared between core units (L2, L3 cache).

There are five entities added into the Cheddar ADL to support the modeling of cache memory. The Cheddar ADL schema of these entities are given in Appendix B - Listing 7.

The descriptions of added entities and their attributes are given below.

- Generic_Cache entity: a model of cache memory contains the following attributes:

    - *cache_size*: the size of a cache

    - *line_size*: the size of a cache line.

    - *associativity*: the associativity of a cache. If associativity is 1, the cache is a direct-mapped cache. If associativity is higher than 1, the cache is a set-associative cache.

- *block_reload_time*: the time it takes to load a memory block from main memory to cache memory.

- *replacement_policy*: the replacement policy of cache lines. There are two replacement policies: FIFO and LRU. Replacement policy is only applicable to set-associative cache.

- *cache_category*: a cache can be either data cache, instruction cache or both.

- *cache_blocks*: a cache consists of a set of cache blocks.

- Cache_Block entity: a model of cache block.

  - *cache_block_number*: a cache block number is used as the id of the cache block.

- Instruction_Cache entity: a model of instruction cache.

- Data_Cache entity: a model of data cache.

  - *write_policy*: a write policy describes the technique of updating the data in the cache and in the main memory. The policies are presented in Section 2.2.3.

- Data_Instruction_Cache entity: a model of cache memory that stores both data and instruction of a program.

Extending Cheddar ADL model of software components

The extended Cheddar ADL model of software components is illustrated in Figure 38. Two new software components are linked to the existing task model in Cheddar ADL: CFG and cache access profile.

There are five entities added into the Cheddar ADL to support modeling CFG and cache access profile. The EXPRESS schema of these entities is given in Appendix B - Listing 8. The descriptions of added entities and their attributes are given below.

- CFG entity: a control flow graph. A graph is modeled as a set of nodes and directional edges.

  - *nodes*: a set of CFG_node.

  - *edges*: a set of CFG_edge.

- CFG_Node entity: a node of a control flow graph.

  - *graph_type*: the type of the graph that a node belongs to. In Cheddar framework, we support different graph types rather than just CFG; however, this is not presented in this thesis.

  - *node_type*: the type of a node that corresponds to its graph type.
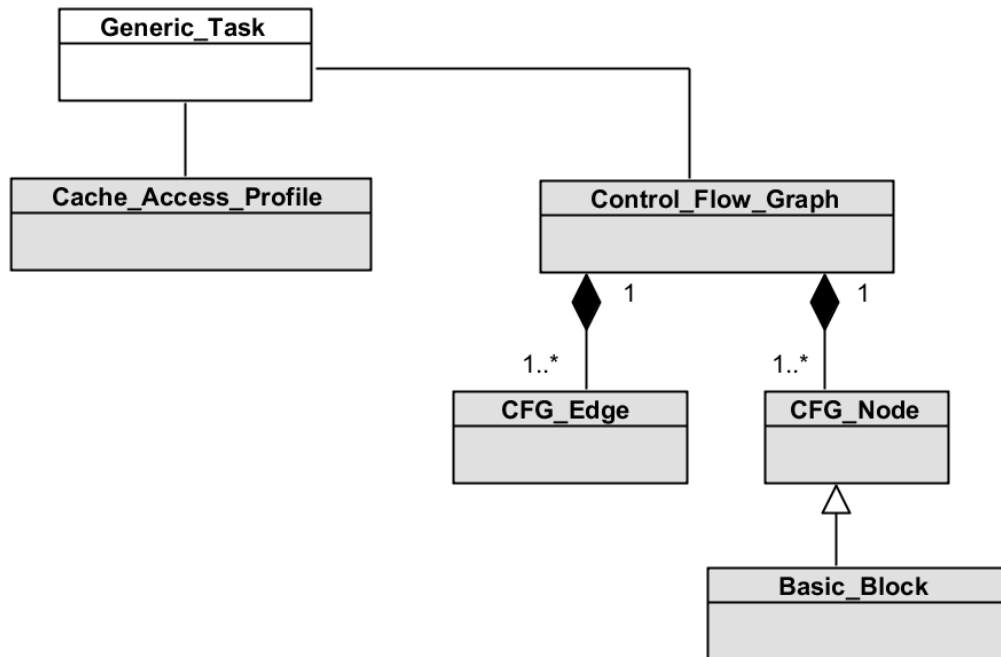
Figure 38: Extended Cheddar ADL model of software components

- CFG_Edge entity: a directional edge of a control flow graph.

    - *node*: head node.

    - *next_node*: tail node.

- Basic_Block entity: an extended model of CFG_node that provides information about the assembly instruction of a program. At the moment, our analysis method takes into account direct mapped instruction cache because instruction cache access pattern is simpler to be computed/extracted from the CFG of a program.

    - *instruction_offset*: the position of the first assembly instruction of a basic block in the main memory.

    - *instruction_capacity*: the size of the assembly instructions of a basic block.

    - *loop_bound*: the upperbound on the number of iterations of the loop that a basic block belongs to.

- Cache_Access_Profile entity: a cache access profile is computed by applying data flow analysis technique in [52].

    - *UCBs*: a set of *cache_block* that represents the UCBs of the *cfg_node* with the highest number of UCBs.

    - *ECBs*: a set of *cache_block* that represents the set of ECBs of a task.

5.3.2 *Implement analysis features: cache access profile computation*

Our tool analyzes and computes the set of UCBs and ECBs of a program. The set of ECBs is computed by taking into account memory blocks accessed in the execution of a program. In our models, position and size of assembly instructions of each basic block are included. Knowing these data and associativity of the cache, we can compute easily the set of ECBs.

The set of UCBs is computed applying the UCBs computation algorithm presented in [52]. The input of the algorithm is the CFG of a program. The set of UCBs of each basic block is computed. The set of UCBs of the basic blocks with highest number of UCBs is chosen to represent the set of UCBs of a program.

The signature of the procedure *Compute_Cache_Access_Profile* is given in Appendix C.

5.3.3 *Implementation summary*

A summary of the implementation of cache access profile computation in Cheddar framework is provided in Table 11.

Part	Description	Packages	LoC
EXPRESS schema	Extended Cheddar ADL with new RTES component models.	5	200
RTES component model	Ada class files of RTES component models generated by the meta-workbench Platypus	10	2280
RTES component handler	Handlers of new RTES component models added in Cheddar scheduling analyzer	14	3769
Analysis feature implementation	- ECB computation algorithm - UCB computation by data flow analysis algorithm in [52]	2	1205

Table 11: Implementation of cache access profile computation in Cheddar framework

5.3.4 *Experiments*

We perform an experiment in order to demonstrate the use of the analysis feature. In this experiment, first, we compute the cache access profiles of tasks. Second, we compute the CRPD upperbound by the number of UCBs multiples with BRT. This CRPD upperbound is then compared to the WCET of tasks. The programs

used in the experiment are taken from Malardalen benchmark suite [45]. They are popular WCET benchmark programs, used to evaluate and compare different types of WCET analysis tools and methods.

The analysis is performed for LEON V3 processor, clock speed 400 MHz, with 1 KB instruction cache and 16 bytes line size. BRT is 10 clock cycles. Data cache is disabled. Each instruction of LEON processor is encoded on 32 bits.

Program	WCET 1 w/o cache (μs)	WCET 2 w/ cache (μs)	CRPD (μs)	UCB
bs.c	6.1	4.5	0.35	14
fac.c	5.9	4.9	0.25	10
fdct.c	80.9	80.2	1.23	49
fibcall.c	8.1	4	0.18	7
insertsort.c	41.07	22.2	0.28	11
ns.c	545.3	273.3	0.50	20
prime.c	6.6	6.8	0.6	24

Table 12: Comparison of CRPD upperbound and WCET for tasks in Malardalen benchmark suite

The result of the analysis is shown in the Table 12. The first and second columns are the WCETs of the programs without and with cache, respectively. The data is obtained by using the WCET analysis feature of the aiT tool provided by AbsInt (http://absint.com/ait). After the CFG is generated, we apply the analysis method that computes the cache access profile of each program and deduce the upper-bound CRPD by taking into account the program point with the highest number of UCBs, which are displayed in the third and the fourth column.

We notice that tasks *bs.c* and *fibcall.c* in the Table 12 have WCETs with a difference of 0.5 μs but the CRPD of *bs.c* is 0.35 μs compared to 0.18 μs of *fibcall.c*. In some cases, it should be consider to reduce the overall response time of the system.

From this result, first, we can see the substantial reduce in WCET of tasks when the cache is enabled (except for *prime.c*). Second, we see that the impact of CRPD on task WCET should not be excluded. We can see that the upper-bound CRPD of one preemption varies from 1 % to 7 % of task WCET.

The computed cache access profile is used as an input parameters for the other CRPD analysis methods implemented in Cheddar as shown in Figure 34. In the next section, we present the implementation of CRPD analysis for WCRT.

5.4 CRPD ANALYSIS FOR WCRT

In this section, we present the implementation of CRPD analysis for WCRT in Cheddar framework. The specification of this analysis features is described as follows:

Purpose	- Compute the WCRT of each task in a task set while taking into account the effect of CRPD in RTES with cache memory
Input	- For each task in a task set, the following information is required: • Capacity - WCET. • Deadline • Period • Priority • Cache access profile
Output	- Computed WCRT of tasks
Method	- The following CRPD analysis for WCRT methods, which are presented in Section 2.5.1, are implemented: ECB-Only [23]. , UCB-Only [52]. , UCB-Union [79]. , ECB-Union [4]. , UCB-Union Multiset [4]. , ECB-Union Multiset [4]. , Combined Multiset [4].

Table 13: Summary of CRPD analysis for WCRT

5.4.1 *Extending Cheddar ADL*

CRPD analysis for WCRT methods presented in Section 2.5.1 are based on the notions of UCB and ECB. In other words, they are compliant with the cache access profile implemented in Cheddar framework. We do not need to extend Cheddar ADL. The prerequisite of these techniques is that the cache access profile of all task are computed.

5.4.2 *Implementing analysis features: CRPD analysis for* WCRT

CRPD analysis for WCRT techniques are based on extending Equation 5 that computes the WCRT of a task in FPP scheduling context with CRPD computation.

Equation 5 was implemented in Cheddar as the procedure *Compute_Response_Time*. The signature of this procedure is given in Appendix C.

5.4.3 *Implementation Summary*

A summary of the implementation of CRPD analysis for WCRT in Cheddar framework is provided in Table 14.

Part	Description	Packages	LoC
Analysis feature implementation	- Implementation of CRPD analysis methods for WCRT: ECB-Only [23]. , UCB-Only [52] , UCB-Union Multiset [4] , ECB-Union Multiset [4] , Combined Multiset [4].	2	1060

Table 14: Implementation of CRPD analysis for WCRT in Cheddar framework

5.5 CRPD-AWARE PRIORITY ASSIGNMENT ALGORITHM

In this section, we present the implementation of CRPD-aware priority assignment algorithm in Cheddar framework. The specification of this analysis feature is described as follows:

Purpose	- Assign priority to tasks of a task set and verify their feasibility while taking into account the effect of CRPD
Input	- For each task in a task set, the following information is required: • Capacity - WCET • Deadline • Period • Cache access profile
Output	- Conclusion about the feasibility of tasks in the task set. - If all tasks are feasible, each task is assigned a priority level.
Method	- The implementation is based on our proposed CRPD-aware priority assignment algorithm presented in Chapter 3. The following algorithms are implemented: CPA_ECB , CPA_PT , CPA_PT-Simplified , CPA_PT-Tree , CPA_PT-Combined

Table 15: Summary of CRPD-aware priority assignment.

5.5.1 *Extending Cheddar ADL*

The proposed CRPD-aware priority assignment algorithm uses the cache access profiles that are compliant with the cache access profile implemented in Cheddar. We do not need to extend Cheddar ADL.

5.5.2 *Implementing analysis feature:* CRPD-*aware priority assignment algorithm*

We implemented our CRPD-aware priority assignment algorithm presented in Chapter 3 in Cheddar. The CRPD-aware priority assignment algorithm is called by using the procedure *OPA_CRPD*. The input of the procedure are a task set and cache access profiles of tasks. A CRPD interference computation solution can be chosen by setting the input variable *"complexity"*. As presented in Section 3, there are five solutions with different levels of complexity and schedulable task set coverage. The signature of the procedure *CPA_CRPD* in Appendix C

5.5.3 *Implementation Summary*

A summary of the implementation of CRPD-aware priority assignment in Cheddar framework is provided in Table 16.

Part	Description	Packages	LoC
Analysis feature implementation	Implementation of CRPD-aware priority assignment algorithm: CPA_ECB , CPA_PT , CPA_PT-Simplified , CPA_PT-Tree , CPA_PT-Combined and OPA	8	2324

Table 16: Implementation of CRPD-aware priority assignment in Cheddar framework

5.6 CRPD-AWARE SCHEDULING SIMULATION

In this section, we present the implementation of CRPD-aware scheduling simulation in Cheddar framework. The specification of this analysis feature is described as follows:

Purpose	- Scheduling simulation taking into account the effect of CRPD for RTES with cache memory
Input	- Scheduling policy - For each task in a task set, the following information is required: • Capacity - WCET • Deadline • Period • Cache access profile
Output	- Scheduling simulation event table. From this table, the following information can be extracted. • Number of deadline misses • Number of preemptions • Total preemption cost • Per task preemption cost • Cache state at each time unit
Method	The implementation is based on our study on CRPD-aware scheduling simulation presented in Chapter 4.

Table 17: Specification of CRPD-aware scheduling simulation.

5.6.1 *Extending Cheddar ADL*

The cache access profile used in our scheduling simulator is compliant with the cache access profile implemented in Cheddar. We do not need to extend Cheddar ADL components model.

5.6.2 *Implementing analysis feature:* CRPD-*aware scheduling simulation*

The scheduling simulator in Cheddar works as follows. First, a system architecture model, including hardware/software components, is loaded. Then, the scheduling is computed by three successive steps: computing priority, inserting ready task into queues and electing task [75]. The elected task will receive the processor for the next unit of time.

The scheduling simulator records different events raised during the simulation, such as task releases, task completions and shared resources lockings or

unlockings. The result of the scheduling analysis is the set of events produced at simulation time.

The scheduling simulator of Cheddar is extended as follows. First, we extend the set of events Cheddar can produce. For example, an event PREEMP-TION, which is raised when a preemption occurs, is added. Second, event RUN-NING_TASK, which is raised when a task executes, is extended to take into account the CRPD. CRPD can be computed by either FS-CRPD computation model or FSC-CRPD computation model that are presented in Chapter 4.

5.6.3 *Implementation summary*

A summary of the implementation of CRPD-aware scheduling simulation in Cheddar framework is provided in Table 18.

Part	Description	Packages	LoC
Analysis feature implementation	Extend the set of scheduler events and events handler of Cheddar scheduling analyzer	3	682

Table 18: Implementation of CRPD-aware scheduling simulation in Cheddar framework

5.6.4 *Experiments and evaluation*

In this section, we perform experiments to demonstrate that the implemented scheduling simulator can handle parameters that are compliant with the existing work in [52], [23], [2]. In addition, we discuss about the dependency between CRPD and scheduling parameters. Furthermore, we point out that our scheduling simulator can run CRPD optimization techniques by taking an example of memory layout optimization by simulated annealing following the work of [58]. We also provide performance and scalability tests of the scheduling simulator.

Experiments are performed with randomly generated task sets. The configuration of our experiments is similar to the configuration presented in section 3.6, which is based on the existing work in [4]. The CRPD computation model, which is used for the experiments, is FS-CRPD.

Experiment 1: CRPD-aware scheduling simulation with priority assignment and processor utilization

In this experiment, we present CRPD-aware scheduling simulation with different priority assignments, scheduling algorithms and processor utilization (PU). In
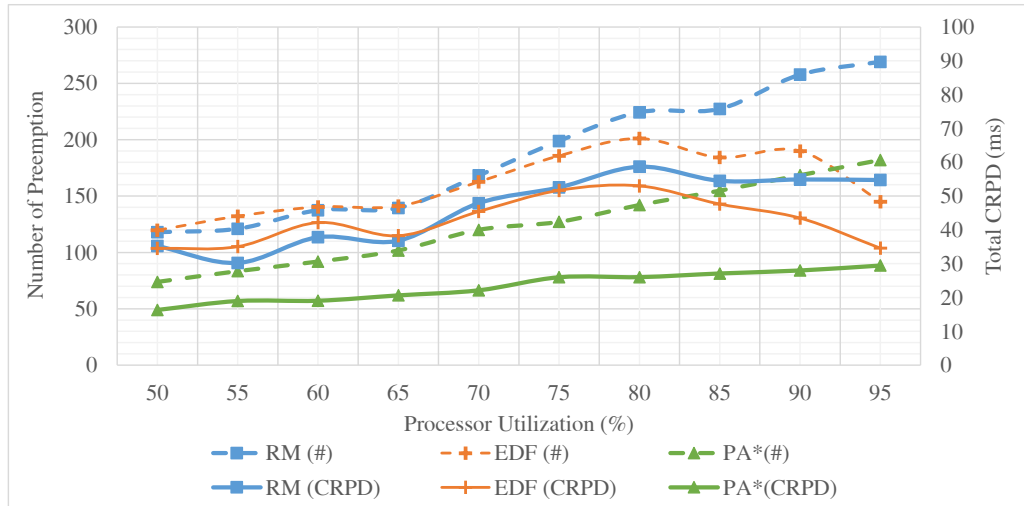
Figure 39: Varying PU, RF=0.3

addition, we discuss about the impact of changing priority assignment/scheduling algorithm and increasing PU to CRPD.

The configuration of this experiment is as follows. PU is varied from 50% to 95% in steps of 5%. RF is fixed at 0.3. For each value of PU, we perform scheduling simulations with 100 task set and compute the average number of preemptions and average total CRPD in a scheduling interval of 1000 ms. Experiments are conducted with two priority assignment algorithms: Rate Monotonic (RM) and another we called PA*, which assigns the highest priority level to the task with the largest set of UCB. In addition, we take into account EDF scheduling policy.

The result of this experiment is sketched in Fig. 39. As the graph illustrates, the number of preemptions and the preemption cost increases steadily from the processor utilization of 50% to 80%. After this point, there is a downward trend in the preemption cost and in the number of preemptions of EDF while there is an upward trend in those data for RM and PA*. Observed from the scheduler, when PU is larger than 80%, many task sets are not schedulable.

In conclusion, first, when PU increases, the total number of preemption and CRPD also increase. However, the change is not linear. Second, a priority assignment algorithm with less number of preemptions tends to give lower total CRPD. EDF and PA* generate less preemptions and CRPD than RM. In fact, to enforce the fixed priority order, the number of preemptions that typically occurs with RM is higher than that with EDF [25]. From this experiment, we see that CRPD depends on the chosen priority assignment or scheduler.

In addition, this experiment shows that both scheduling analysis and CRPD analysis should be performed jointly. PA*, a priority assignment taking CRPD into account has a significant lower total CRPD. The decrease in total CRPD of PA* with RM and EDF is roughly 30 ms on a scheduling interval of 1000
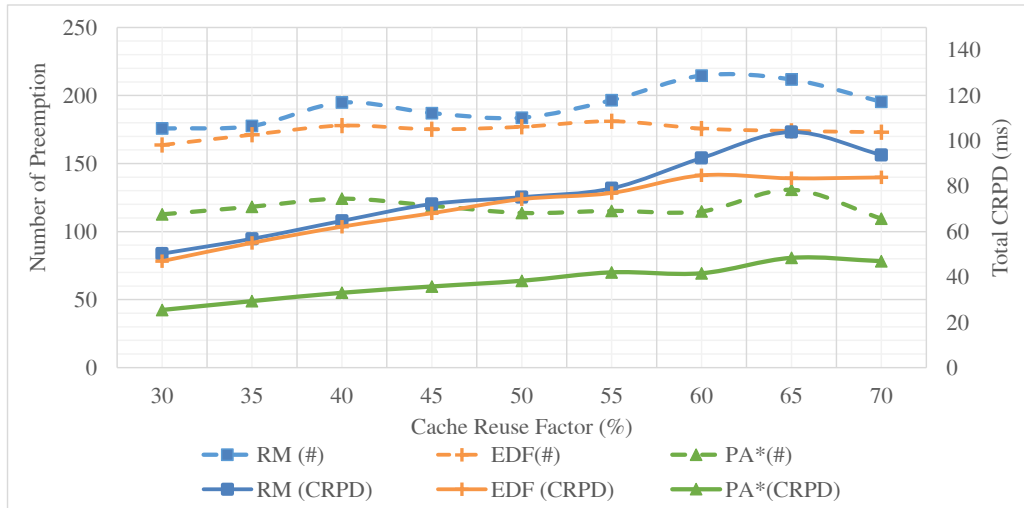
Figure 40: Varying RF, PU=0.7

ms. However, comparing to RM and EDF, feasibility constraints of tasks are not respected with PA*, only total CRPD is reduced. In other words, the number of tasks that missed deadline in PA* is higher than that of RM and EDF.

Experiment 2: CRPD-aware scheduling simulation with priority assignment and cache reuse factor

In this experiment, we observe the change in the result of CRPD-aware scheduling simulation when varying RF parameter instead of PU parameter. The configuration of this experiment is similar to the first experiment, except that PU is fixed at 0.7 and RF is varied from 0.3 to 0.7. For each value of RF, we perform scheduling simulations with 100 task sets and compute the average number of preemptions and average total CRPD in a scheduling interval of 1000 ms.

The result of this experiment is shown in Fig. 40. We get a similar observation with the first experiment in terms of number of preemption and total CRPD regarding those three priority assignment algorithms. However, when varying RF, the change in number of preemption is less significant, with a maximum difference of 50 preemptions; and the change in total CRPD is more significant, with a maximum difference of 50 ms, than when varying PU (with maximum difference of number of preemption and total CRPD are 150 preemptions and 35 ms, respectively).

To conclude, experiment 1 and 2 showed that our tool can perform scheduling simulation of RTES with cache with various scheduling parameters and can be used to study the dependencies amongst those parameters.
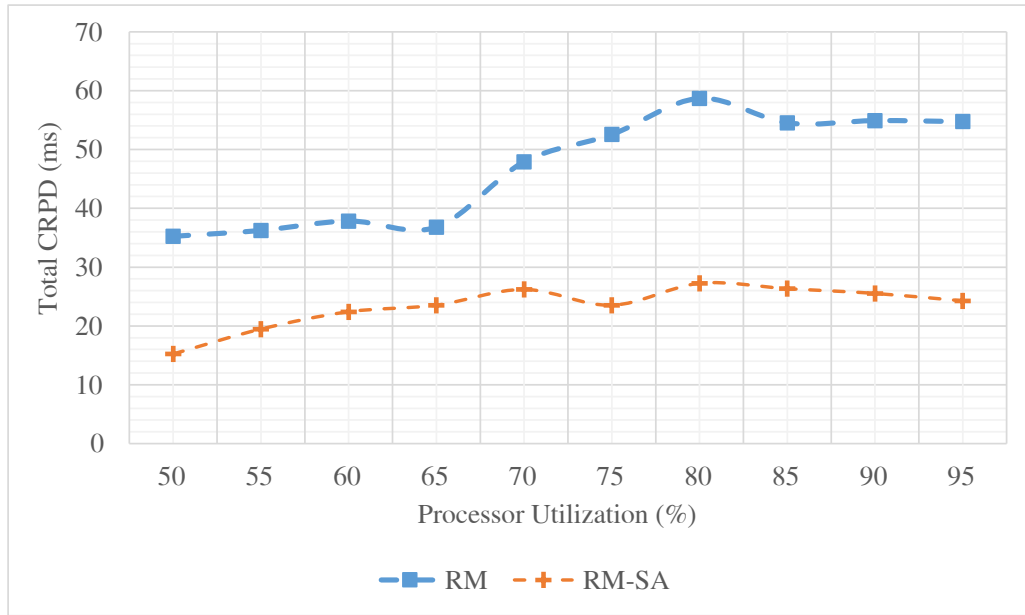
Figure 41: Total CRPD with and without memory layout optimization

Experiment 3: CRPD-aware scheduling simulation with memory layout optimization by simulated annealing

The objective of this experiment is to show that users can perform CRPD optimization approaches with our scheduling simulator. We apply memory layout optimization by simulated annealing (SA) based on the work of [58] with our generated task sets. In our experiment, the objective of SA is to lower the total CRPD after a scheduling simulation over a scheduling interval of 1000 ms.

For each iteration of SA, we perform a swap in memory layout between two random tasks. Changes are made to the layout of tasks in memory, and then mapped to their cache layout for evaluation. The total CRPD is computed by scheduling simulation. The optimum layout is the layout which has the lowest total CRPD. Initial temperature of SA is 1.0, and after every iteration, it is reduced by multiplying it by a cooling rate of 0.5 until it reaches the target temperature of 0.2. The number of iteration for each temperature is 10.

The result of this experiment is shown in Fig. 41. From the graph, we can see the impact of memory layout optimization to total CRPD. We can reduce roughly 30-50% of total CRPD. To sum up, this experiment shows that our tool allows users to perform a specific optimization of CRPD for a given scheduling algorithm.

Experiment 4: Performance/Scalability Analysis

The objective of this experiment is to evaluate the performance and the scalability of the scheduling simulator when scheduling simulation interval increases. In general, there are four factors affecting the performance of a scheduling sim-
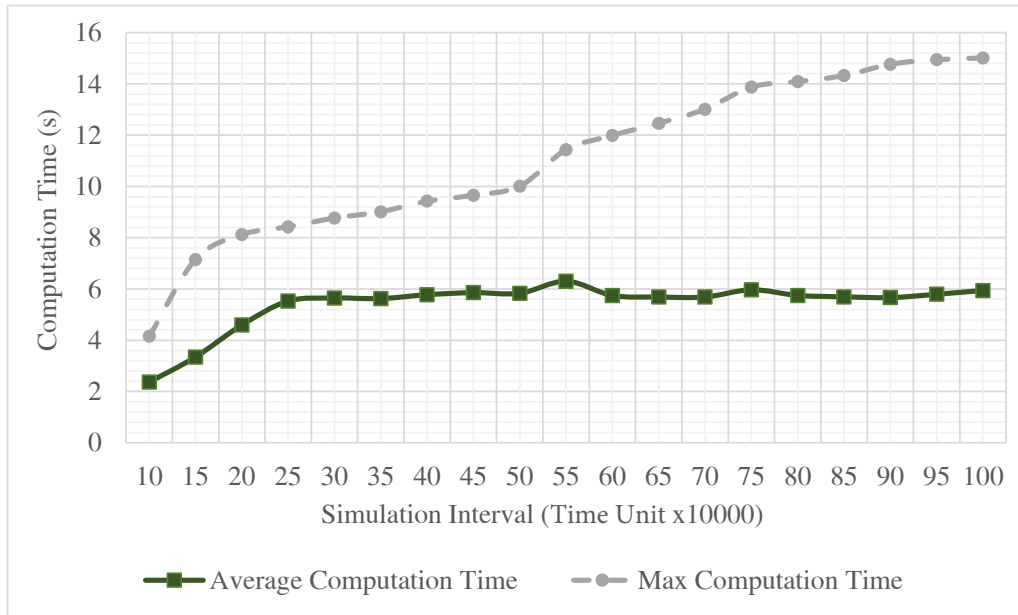
Figure 42: Computation time of the simulator

ulator: (1) the number of tasks, (2) the scheduling simulation interval, (3) the cache size and (4) the number of events. The first three factors depend on the chosen RTES model. The number of events depends on characteristics of the RTES model; for example, a higher processor utilization means a higher number of preemption events. In this experiment, we choose to test a RTES model of 10 tasks and 256 cache blocks. Processor utilization is set to 70 %. Scheduling simulation is ranging from 100,000 to 1,000,000 units of time where 1 unit = 8μs.

Fig. 42 displays results of our experiment on a PC with Intel Core i5-3360 CPU, 4 GBs of memory, running Ubuntu 12.04. For each simulation interval, 100 task sets are generated. We perform scheduling simulation and compute the maximum and average computation time.

As we can see, while maximum computation time increases slightly when simulation interval increases, average computation time only fluctuates around 6 seconds. This shows that the tool is scalable when simulation interval is high.

5.7 IMPLEMENTATION ISSUES

Several issues were raised when implementing CRPD analysis features in Cheddar. Most of them are related to mixing timing specifications of different orders of magnitude. Others are related to tools interoperability.

Mixing timing specifications of different orders of magnitude makes the computation of the feasibility interval complex. Feasibility interval is required to perform CRPD-aware scheduling simulation and CRPD-aware priority assignment. In practice, cache block reload time is significantly smaller than period or

capacity of a task. In Cheddar, we do not prescribe 1 unit of time is equivalent to 1 ms or 1 μs, which are the granularity of task period and block reload time. The scheduling simulation interval needed to verify the schedulability of a task set could be significantly large if, for example, one μs is chosen as a time unit. A solution in practice is to design systems with harmonic task sets in order to minimize the feasibility interval; however, it is clearly not always possible. In addition, instead of using 1 μs, we use the BRT as a base value for 1 unit of time and round up the WCETs.

A large scheduling simulation interval also raises issues regarding performance and scalability. Even with harmonic task sets, the tool must be able to perform scheduling simulations in a large interval to overcome the difference between cache block reload time and task period, which may be CPU and memory expensive. As Cheddar stores scheduling simulation results into XML files, it can also be I/O intensive. To reduce memory and I/O overhead, we selected a subset of events the simulator has to handle and store.

A second issue we were facing is about tool interoperability. The input data of the CRPD analysis in our tool is designed to be compatible with data provided by a WCET analysis tool. We also support data input in XML format, but, at the moment, we do not enforce tool interoperability and we expect to investigate WCET tools in order to overcome this issue.

5.8 CONCLUSIONS

In this chapter, we presented the implementation of several CRPD analysis methods for RTES with cache in Cheddar framework and point out several implementation issues. Our implementation addressed CRPD analysis methods in the four subjects:

- Cache access profile computation;

- CRPD analysis for WCRT;

- CRPD-aware scheduling simulation;

- CRPD-aware priority assignment.

Regarding analysis method of limiting CRPD, a complete implementation of an analysis method is not supported in Cheddar framework. However, as shown in Section 5.6.4, a memory layout optimization technique to limit CRPD was used in tandem with our scheduling simulator.

Information about Cheddar and its analysis features can be found at: http://beru.univ-brest.fr/~singhoff/cheddar/. The source code of the presented work is available under GNU GPL licence at http://beru.univ-brest.fr/svn/CHEDDAR/branches/caches/src/.

Part III

CONCLUSION

Chapter 6

CONCLUSION

Contents

The work presented in this thesis contributed to scheduling analysis of real-time embedded systems RTES with cache memory. It was done with the idea that the analysis of cache related preemption delays (CRPD) is essential for scheduling of RTES. This is not a new idea, which has been proved by many existing research work [23, 52, 82, 72, 4, 58, 67]. There are developed analysis methods for fixed priority preemptive (FPP) scheduling and some basic analysis for dynamic priority preemptive (DPP) scheduling. However, the focus has mainly been on the computation of the worst-case response time WCRT and limiting CRPD. Furthermore, up until now, it has not been possible to account for the effect of CRPD when assigning priorities to tasks. In addition, there has not been only few work to make use of scheduling simulation as a verification method for RTES with cache memory.

6.1 CONTRIBUTION SUMMARY

The work in this thesis contributed in two domains of scheduling analysis of RTES with cache memory by taking into account the effect of CRPD: priority assignment and scheduling simulation. In addition, we implemented several CRPD analysis methods in a scheduling analysis tool which is available to the community.

CRPD-aware priority assignment algorithm

Chapter 3 presented a CRPD-aware priority assignment algorithm based on Audsley's priority assignment [7] (OPA). The main advantage of our approach is that

it can detect early unschedulable task at a specific low priority level. This is especially useful when there are several tasks with similar periods but significant different in cache utilization. With this approach, the schedulability of a task set is verified in the process of assigning priorities to tasks. Schedulability verification is provided by five solutions that are different in terms of complexity and schedulable task set coverage.

There are several limitations of our approach that provide perspective for the future works. The first limitation is that our proposed priority assignment algorithm, which is based on OPA, is not optimal. OPA is known to be optimal if several conditions are satisfied. However, the requirement cannot be met with CRPD. At the moment, finding an optimal CRPD-aware priority assignment is still an open issue. The second limitation of our approach is that the scalability of the most efficient solution in terms of schedulable task set coverage is limited by its complexity. To overcome this limitation, we need to increase the efficiency of less complex solutions.

CRPD-aware scheduling simulation

Chapter 4 presented our study on CRPD-aware scheduling simulation. We investigate classical CRPD computation models used in scheduling simulation and present existing issues regarding the pessimism of these models. Then, we discuss about the sustainability of scheduling simulation with classical CRPD computation models. We explain the problem related to CRPD in sustainability analysis and the reason why CRPD-aware scheduling simulation is not sustainable in general cases.

We propose a new CRPD computation model named FSC-CRPD to address the previous issues. In this model, based on an observation from real system execution presented in [59], we take a new assumption that bounds the CRPD by the executed capacity of a task. When this assumption holds, scheduling simulation is less pessimistic and then becomes sustainable with regard to the capacity parameter. The conclusion about the sustainability of scheduling simulation with FSC-CRPD allows us to prove the feasibility interval of our system model.

The established results allow the use of CRPD-aware scheduling simulation as a verification method to evaluate the schedulability of periodic tasks. In addition, this work gives perspectives about in which cases CRPD-aware scheduling simulation is sustainable and is not.

Available tool

The work in this thesis is made available in Cheddar - an open-source scheduling analyzer. First, the architecture description language (ADL) of Cheddar is

extended to allow the modeling of RTES with cache memory. Second, the following CRPD analysis features are available in Cheddar:

- Cache access profile computation based on the notion of useful cache block [52] (UCB) and evicting cache block (ECB) [23].

- CRPD analysis for WCRT approaches including ECB-Only [23], UCB-Only [52], UCB-Union [79], ECB-Union [4], UCB-Union Multiset [4], ECB-Union Multiset [4] and Combined Multiset [4].

- Our proposed CRPD-aware priority assignment algorithm

- CRPD-aware scheduling simulation with two CRPD computation models: FS-CRPD and FSC-CRPD.

Cheddar is now a framework that allows modeling and scheduling analysis of RTES with cache memory.

6.2 FUTURE WORK

The work presented in this thesis has addressed issues regarding priority assignment and scheduling simulation of RTES with cache memory. Our plan is to utilize the knowledge and experience learned to address the identified limitations of our work in these two subjects.

Regarding the work presented in Chapter 3, our next objective is to employ a CRPD optimization technique such as memory layout optimization [58] when assigning priority to task. The idea is to integrate a CRPD optimization technique in the CRPD-aware priority assignment algorithm to improve the percentage of schedulability task set coverage.

In addition, in Chapter 4, we have not yet investigated the problem of CRPD-aware scheduling simulation for tasks with arbitrary deadline. In this case, modeling cache accesses and evaluating the number of UCB loaded into instruction cache of a task could be complex because there are multiple jobs of this task are released and executed. The general idea proposed in Chapter 5 was to compute an upper-bound CRPD based on the previous scheduling at a given point in time. This idea could be applied to study more complex system models and evaluate other scheduling properties besides sustainability.

One the limitation of the work in Chapter 4 is the lack of actual comparison between the proposed CRPD-aware scheduling simulation and real execution of a practical system. Given a case study, we plan to perform scheduling simulation and observe its execution on a hardware platform. However, designing facilities that are needed to observe and analyze accesses to cache memory on a hardware platform is a challenge.

At the moment, the implementation of CRPD analysis methods in Cheddar only supports simple cache architecture. An improvement to the tool would be to take into account advanced cache architecture with multi-level of cache. Furthermore, an improvement that could be made to the framework is to enforce tool interoperability so that Cheddar ADL compatible control flow graph can be generated by another timing analysis tool.

Finally, all the work in this thesis has focused on the effect of CRPD on a single core processor. The next major improvement is to extend the proposed approaches to multi-core processor. However, we have to take into account an additional effect named cache related migration delay (CRMD), which occurs when a task is migrated to a different processor thus its private cache is lose. In this context, our approaches must be extended to take into account both CRPD and CRMD.

Part IV

APPENDIX

Appendix A

ALGORITHM AND PSEUDO CODE

A.1 CPA-PT: CRPD POTENTIAL PREEMPTION COMPUTATION

In this section, we present the algorithm that computes the number of potential preempted task for each job in the set η. In chapter 3, the problem was presented as follows: "*given $l - 1$ jobs released in the $[t_1, t_l)$, what is the maximum number of incomplete jobs at a given time instant*".

Example

First, we illustrate our solution with a simple example. We assume three jobs of $\eta[1], \eta[2], \eta[3]$ released at $t_1, t_2, t_3, t_1 < t_2 < t_3$. To facilitate the computation, for job $\eta[i]$, we construct an array $A_i[]$. The size of the array is an upper-bound on the number of incomplete jobs. The elements of the array are time instants. At time $A_i[m]$, there is at least m jobs that are completed. Considering a job $\eta[i + 1]$ released at time t_{i+1}. By comparing t_{i+1} and $A_i[m]$, we get the number of jobs that are potentially preempted by $\eta[i + 1]$.

We consider $\eta[1]$ released at time t_1:

- At the beginning, the time $t_1 + C_1$ of τ_1.

  Array $A_1[]$ consists of 1 element.

  ---
  1      $A_1[1] = t_1 + C_1$

We consider $\eta[2]$ released at time t_2:

- If $t_2 > A_1[1]$, $\eta[2]$ potentially preempts 0 job.

- If $t_2 < A_1[1]$, $\eta[2]$ potentially preempts 1 job, which is τ_1, with the preemption cost $\gamma_{\theta_2,2}$.

  Assuming $t_2 < A_1[1]$, array $A_2[]$ now consists of 2 elements.

1      $A_2[1] = \max(t_2 + C_2, A_1[1])$.

2      $A_2[2] = A_1[1] + C_2 + \gamma_{\theta_2,2}$

$A_2[1]$ is the time instant at which there is at least one job completed. If $\eta[2]$ is not the lowest priority job, then $\eta[2]$ completes at time $t_2 + C_2$. If $\eta[2]$ is the lowest priority job, then $\eta[1]$ completes at time $A_1[1] = t_1 + C_1$.

$A_2[2]$ is the time instant at which two tasks are completed by assuming that $\eta[2]$ preempts $\eta[1]$.

We consider $\eta[3]$ released at time t_3:

- If $t_3 \geqslant A_2[2]$, $\eta[2]$ potentially preempts 0 job.

- If $A_2[1] \leqslant t_3 < A_2[2]$, $\eta[3]$ potentially preempts 1 job, with the preemption cost $\gamma_{\theta_3,3}$.

- If $t_3 < A_2[1]$, $\eta[3]$ potentially preempts 2 job, with the preemption cost $\gamma_{\theta_3,3}$.

The deduction process in this example can be generalized to a set of $l-1$ jobs. The algorithm for $l-1$ is provided in the Appendix A.

Algorithm

We implement an iterative construction algorithm to solve the problem. This algorithm starts from the first job of η set, $\eta[1]$. For each job after $\eta[i]$, the algorithm computes the number of potential preempted jobs, CRPD and the array $A_i[]$. The number of potential preempted jobs of $\eta[i]$ is computed by taking into account the release time t_i and the array $A_{i-1}[]$. Let $n_{i-1} = A_{i-1}[].size$ denotes the number of element of $A_{i-1}[]$.

- If $t_i \geqslant A_{i-1}[n_{i-1}]$, then all jobs released prior to $\eta[i]$ have completed and $\eta[i]$ does not preempt any jobs.

- If $A_{i-1}[n_{i-1} - 1] \leqslant t_i < A_{i-1}[n_{i-1}]$, then $\eta[i]$ potentially preempts 1 job out of n_{i-1} jobs.

- If $A_{i-1}[n_{i-1} - 2] \leqslant t_i < A_{i-1}[n_{i-1} - 1]$, then $\eta[i]$ potentially preempts 2 jobs out of n_{i-1} jobs.

- if $A_{i-1}[n_{i-1} - m] \leqslant t_i < A_{i-1}[n_{i-1} - (m-1)]$, then $\eta[i]$ potentially preempts m jobs of out n_{i-1} jobs.

The construction of array $A_i[]$ is described as follows.

- $A_i[1]$ is the time instant, which guarantees that there is at least 1 job completed by assuming that there is at least one job released and then executed non-preemptively.

- $A_i[n_i]$ is the time instant, which guarantees that all jobs released before $\eta[i]$ are completed by assuming that all potential preemption occurs.

- $A_i[k], 1 < k < n_i$ is the time instant, which guarantees that there are k jobs completed. There are two cases

  - The job of $\eta[i]$ is completed and $k-1$ jobs released previously are completed. It is given by the time to complete the remaining computational requirement of $k-1$ jobs at time t_i, the capacity of $\eta[i]$ and the potential CRPD created by $\eta[i]$.

  $$t_i + max(0, t_i - A_{i-1}[k-1]) + C_i + \gamma_{\Theta_i,i}$$

  - The job of $\eta[i]$ is not completed and k jobs released previously are completed. This case is provided by $A_{i-1}[k]$

  $A_i[k]$ is computed as follows:

  $$A_i[k] = max(t_i + max(0, t_i - A_{i-1}[k-1]) + C_i + \gamma_{\Theta_i,i}, A_{i-1}[k-1])$$

```
1   CPA-PT(η)

2   begin

3     A₁[1] = t₁ + C₁

4     for i in 2..η.size loop

5        n ← A_{i-1}.size

6        if tᵢ ⩾ A_{i-1}[n] then

7           Aᵢ[1] ← tᵢ + Cᵢ

8           γ_{Θᵢ^PT,i} ← 0

9        else if tᵢ < A_{i-1}[m] then

10          |Θⱼ^PT| ← n − m

11          γ_{Θᵢ^PT,i} ← Get_CRPD(|Θⱼ^PT|)

12          Aᵢ[1] ← max(tᵢ + Cᵢ, A_{i-1}[1])

13          for k in 2..n loop

14             Aᵢ[k] ← max(

15                tᵢ + max(0, tᵢ − A_{i-1}[k−1]) + Cᵢ + γ_{Θᵢ,i},

16                A_{i-1}[k])

17          end loop

18          Aᵢ[n + 1] = A_{i-1}[n] + Cᵢ + γ_{Θᵢ^PT,i}

19       end if

20    end

21 end
```

A.2   CPA-TREE: TREE COMPUTATION

The algorithm of computing the tree composes of two parts. The first part handles the decision of preemption at the release time of a job. The second part handles the finding of the executing job at the release time of a job.

Compute Tree function

The first part takes into account the two decisions of the scheduler: *allow preemption* and *deny preemption*. Assuming a job $\eta[l]$, at t_l the algorithm checks if there is a job $eta[k], t_k \leqslant t_l$ executing.

- If there is $\eta[k]$, which is *released*, *executing* and not *completed* at t_l, two child nodes are added. This is the case of a *potential preemption*. One node presents the case where the $eta[l]$ preempted $\eta[k]$ and one node presents the case where $\eta[l]$ does not preempt. Two decisions of the scheduler are addressed: *allow preemption* and *deny preempting*.

- If there is a job $\eta[k]$, which is *released* at the release time of $\eta[l]$, i.e. two jobs share a same release time, we add one sibling node..

- If there are no job executing, we add one child node to mark the release of $\eta[l]$.

Preemption cost is taken into account when the scheduler makes a decision of allowing preemption. The set Θ_l is consists of all job $\eta[k]$ previously released and executed. This preemption cost is added to the remaining capacity of $\eta[k]$. Nested preemption is considered by updating the remaining capacity of every preempted jobs.

The task-level priorities of jobs are set according to the decisions of the scheduler. Before making any decisions, the scheduler checks if there are any policies violated. The algorithm finishes when all jobs are assessed. In the end, each end node of a branch presents total interference caused by η in an interval. The interference consists of the computational requirement of the jobs and preemption cost corresponding to the decisions of the scheduler.

If the task-level priority of a job is not assigned, all possible decisions of the scheduler are made. Then, the implicit priority corresponding to each decision is stored in each branch. For example, in the branch where a job of τ_j preempted a job of τ_k, $\Pi_j > \Pi_k$. In case there are several policies, *transitivity* is needed to be taken into account. Considering this problem as find the existing path between two nodes in a graph, transitive closure [87] is used to solve this issue.

The algorithm of computing the tree is as follows.

```
1  Compute_Tree(η, j)
2  begin
3     if exist η[k] executing then
```

```
4    if η[j] can preempt η[k] then
5      add child node η[j]
6      set Π_j > Pi_k
7      Compute_Tree(η, j+1)
8    end if
9    if η[k] can continue execution then
10     add child node η[k]
11     set Π_k > Π_j
12     Compute_Tree(η, j+1)
13   end if
14 end if
15 if no job executing then
16   add child node η[j]
17   Compute_Tree(η, j+1)
18 end if
19 end
```

A.3   EVENT HANDLERS FOR SCHEDULING SIMULATION WITH FS-CRPD

The pseudo code of the event handler regarding FS-CRPD computation model is written below. The notation $\tau_i.cUCB$ represents the set of UCBs of task τ_i in the cache. It is computed from a system model at scheduling simulation time. The function Remove() at line 8 is used to remove elements from a set.

```
1  event SCHED_START
2    for each task τ_i loop
3      τ_i.cUCB ← τ_i.UCB
4    end loop
5  event PREEMPTION
6    τ_j ← preempting_task
7    for each task τ_i preempted loop
8      τ_i.cUCB ← Remove(τ_i.cUCB, τ_j.ECB)
9    end loop
10 event RUNNING_TASK
11   τ_i ← executing_task
12   CRPD ← (τ_i.UCB - τ_i.cUCB) * Miss_Time
13   τ_i.cUCB ← τ_i.UCB
```

Listing 5: Extended event handlers regarding FS-CRPD computation model

The event handler is described as follows. At the start of the scheduling simulation, a SCHED_START event is raised. WCET of a task is assumed to include the cache intrinsic interference when the task is executed non-preemptively. So, on event SCHED_START, the set of UCBs of a task is assumed to be filled. In other words, the set of UCBs of task τ_i in the cache is equal to its set of UCBs. We can see that at line 3, τ_i.cUCB is set to be equal to τ_i.UCB.

When a preemption occurs, a PREEMPTION event is raised and the simulator computes the evicted UCBs of preempted tasks by taking into account the ECBs of the preempting task. The scheduler keeps track of the number of UCBs in the cache of each task. We can see that at line 8, elements in the set τ_i.cUCB is removed if they are also in the set τ_j.ECB. At this event, the CRPD is not computed yet.

When a task executes, a RUNNING_TASK event is raised. The scheduler first checks if all the UCBs of this task are loaded into the cache. If so, the task continues its execution. If not, the task reloads the evicted UCBs. The CRPD is added to the remaining capacity of the task itself. In our implementation, CRPD is not added to the capacity of preempted tasks at the preemption point but at the instant, of which those tasks resume execution.

In FS-CRPD computation model, it is assumed that any partial execution of a task uses all its UCBs and ECBs. As a result, the CRPD is computed by taking into account the number of evicted UCBs multiplies with BRT, as we can see at line 12.

A.4   EVENT HANDLERS FOR SCHEDULING SIMULATION WITH FSC-CRPD

The pseudo code of the event handler regarding FSC-CRPD computation model is written below. It is an extension of the event handler for FS-CRPD computation model. The parameter ρ_i, which represents the actual number of UCBs loaded into the cache, is taken into account.

In the event SCHED_START, ρ_i is set to 0. In the event PREEMPTION, there is no update made to ρ_i. In the event RUNNING_TASK, the CRPD is computed by taking into account both the number of evicted UCBs and ρ_i. In addition, ρ_i is increased by U_N, that represent the number of UCBs loaded into the cache per unit of execution.

```
14    event SCHED_START
15      for each task τᵢ loop
16        τᵢ.cUCB ← τᵢ.UCB
17        ρᵢ ← 0
18    event PREEMPTION
```

```
19      τⱼ ← preempting_task
20      for each task τᵢ preempted loop
21        τᵢ.cUCB ← Remove(τᵢ.cUCB, τⱼ.ECB)
22    event RUNNING_TASK
23      τᵢ ← executing_task
24      CRPD ← min((τᵢ.UCB - τᵢ.cUCB),ρᵢ) * BRT
25      ρᵢ ← max((ρᵢ - (τᵢ.UCB - τᵢ.cUCB)),0))
26      τᵢ.cUCB ← τᵢ.UCB
27      ρᵢ ← ρᵢ + Uₙ
```

Listing 6: Extended event handlers regarding FSC-CRPD computation model

Appendix B

EXPRESS SCHEMA

B.1 EXPRESS SCHEMA OF CACHE MEMORY

```
1  SCHEMA Caches;
2    ENTITY Generic_Cache
3       SUBTYPE OF ( Named_Object );
4       cache_size : Natural;
5       line_size : Natural;
6       associativity : Natural;
7       block_reload_time : Natural;
8       replacement_policy : Cache_Replacement_Policy_Type;
9       cache_category : Cache_Type;
10      cache_blocks : Cache_Blocks_Table;
11   END_ENTITY;
12
13   ENTITY Data_Cache
14     SUBTYPE OF ( Generic_Cache );
15     write_policy : Write_Policy_Type;
16   END_ENTITY;
17
18   ENTITY Instruction_Cache
19     SUBTYPE OF ( Generic_Cache );
20   END_ENTITY;
21
22   ENTITY Data_Instruction_Cache
23     SUBTYPE OF ( Generic_Cache );
24     write_policy : Write_Policy_Type;
25   END_ENTITY;
26
27   ENTITY Cache_Block
28     SUBTYPE OF ( Named_Object );
```

```
29    cache_block_number : Natural;
30  END_ENTITY;
31 END_SCHEMA;
```

Listing 7: EXPRESS schema of cache memory

B.2 EXPRESS SCHEMA OF CFG AND CACHE ACCESS PRO-FILE

```
1 ENTITY CFG
2   nodes : CFG_Nodes_Table;
3   edges : CFG_Edges_Table;
4 END ENTITY;
5 ENTITY CFG_Node
6     graph_type : CFG_Graph_Type;
7     node_type : CFG_Node_Type;
8 END_ENTITY;
9
10 ENTITY CFG_Edge
11   node : STRING;
12   next_node : STRING;
13 END_ENTITY;
14
15 ENTITY Basic_Block
16     SUBTYPE OF ( CFG_Node );
17     instruction_offset : INTEGER;
18     instruction_capacity : INTEGER;
19     data_offset : INTEGER;
20     data_capacity : INTEGER;
21     loop_bound : INTEGER;
22 DERIVE
23     SELF\CFG_Node.graph_type : CFG_Graph_Type := CFG_Basic_Block;
24 END_ENTITY;
25
26 ENTITY Cache_Access_Profile
27   UCBs : Cache_Blocks_Table;
28   ECBs : Cache_Blocks_Table;
29 END ENTITY;
```

Listing 8: EXPRESS schema of CFG and cache access profile

Appendix C

METHOD SIGNATURE

C.1 PROCEDURE COMPUTE_CACHE_ACCESS_PROFILE

```
1  procedure Compute_Cache_Access_Profile
2    (Sys                    : in out System;
3     Task_Name              : in Unbounded_String;
4     A_Cache_Access_Profile : out Cache_Access_Profile_Ptr);
```

Listing 9: Procedure Compute_Cache_Access_Profile

There are two input parameters:

- Sys: Sys is a container object that contains all created software and hardware components. Cache configuration, memory layout, tasks and control flow graphs are included.

- Task_Name: The name of the task that the cache access profile is computed. In Cheddar, *Task_Name* is unique and acts as an identifier of a task.

The output parameter is the computed cache access profile.

C.2 PROCEDURE COMPUTE_RESPONSE_TIME

This procedure is extended to take into account CRPD computation. The method signature of the procedure *Compute_Response_Time* is given below:

```
1  procedure Compute_Response_Time
2    (My_Scheduler           : in Fixed_Priority_Scheduler;
3     My_Tasks               : in out Tasks_Set;
4     Processor_Name         : in Unbounded_String;
5     Msg                    : in out Unbounded_String;
6     Response_Time          : out Response_Time_Table;
7     With_CRPD              : in Boolean := false;
```

```
8       CRPD_Computation_Approach : in CRPD_Computation_Approach_Type;
9       Block_Reload_Time         : in Natural := 0;
10      My_Cache_Access_Profiles  : in Cache_Access_Profiles_Set);
```

Listing 10: Procedure Compute_Response_Time

This procedure computes the WCRT of all tasks in a given processor. There are eights input parameters.

- *My_Scheduler*: The scheduler that is used to schedule tasks. It must be a fixed priority scheduler.

- *My_Tasks*: The set of tasks that the WCRTs are computed.

- *Processor_Name*: The name of the processor that tasks are assigned to.

- *With_CRPD*: A boolean parameter indicates that the WCRT computation takes into account CRPD or not.

- *CRPD_Computation_Approach*: A parameter indicate which CRPD analysis for WCRT technique is used.

- *Block_Reload_Time*: The block reload time of the cache.

- *My_Cache_Access_Profiles*: The cache access profiles of tasks that the WCRT are computed.

There are two output parameters:

- *Msg*: A message is returned to higher function call.

- *Response_Time*: A table contains the computed WCRTs of tasks.

C.3 PROCEDURE CPA_CRPD

```
1    type CRPD_Inteference_Computation_Complexity is
2      (ECB_Only,
3       PT_Simplified,
4       PT_Binomial_Coefficient,
5       Tree,
6       Combined);
7
8    procedure CPA_CRPD
9      (my_tasks                 : in out Tasks_Set;
10      my_cache_access_profiles : in Cache_Access_Profiles_Set;
11      complexity               : in CRPD_Inteference_Computation_
            Complexity);
```

There are three input parameters:

- *my_tasks*: a set of tasks with unassigned priorities.

- *my_cache_access_profiles*: cache access profiles of the tasks in the set.

- *complexity*: a parameter indicates which CRPD interference computation approach is used. It is an enumeration types with five options.

There is one output parameter:

- *my_tasks*: a set of tasks with assigned priorities.

An exception NO_FEASIBLE_PRIORITY_ASSIGNMENT is raised if the algorithm cannot find a priority ordering that makes the task set schedulable.

BIBLIOGRAPHY

[1] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis. Outstanding paper: Evaluation of cache partitioning for hard real-time systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 15–26, 2014.

[2] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, 2011.

[3] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, pages 261–271, 2011.

[4] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[5] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I Davis. On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Systems*, pages 1–46, 2015.

[6] Neil Audsley, Alan Burns, Rob Davis, Ken Tindell, and Andy Wellings. *Real-time system scheduling*. Springer Berlin Heidelberg, 1995.

[7] Neil C Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. In *Technical Report YCS 164, Dept. Computer Science*. University of York, UK, 1991.

[8] Brian Bailey and Grant Martin. *Processor-Centric Design: Processors, Multi-Processors, and Software*, pages 225–272. Springer US, Boston, MA, 2010.

[9] Aloysius K. Mok Baruah, Sanjoy K. and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings 11th Real-Time Systems Symposium*, pages 182–190, 1990.

[10] Sanjoy K Baruah. Dynamic and static priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[11] Andrea Bastoni, Bjorn B Brandenburg, and James H Anderson. Is semi-partitioned scheduling practical? In *23rd Euromicro Conference on Real-Time Systems*, pages 125–135. IEEE, 2011.

[12] Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, volume 5, 1994.

[13] Mordechai Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[14] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *23rd Euromicro Conference on Real-Time Systems*, pages 217–227. IEEE, 2011.

[15] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., 1999.

[17] Etienne Borde, Smail Rahmoun, Fabien Cadoret, Laurent Pautet, Frank Singhoff, and Pierre Dissaux. Architecture models refinement for fine grain timing analysis of embedded systems. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*, pages 44–50. IEEE, 2014.

[18] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 269–279, July 2007.

[19] Alan Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall, 1994.

[20] Alan Burns and Sanjoy Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

[21] Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[22] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

[23] José V Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the* 2nd *IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 204–212, 1996.

[24] Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science)*. Plenum Publishing Co., 2005.

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

[26] Fabien Cadoret, Etienne Borde, Sebastien Gardoll, and Laurent Pautet. Design patterns for rule based refinement of safety critical embedded systems models, 2012.

[27] Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, Manar Qamhieh, et al. Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms. In *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 21–26, 2012.

[28] Maxime Chéramy, Anne-Marie Déplanche, Pierre-Emmanuel Hladik, et al. Simulation of real-time multiprocessor scheduling with overheads. In *International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 2013.

[29] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014.

[30] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, and Silvano Dal Zilio. Simulation of real-time scheduling algorithms with cache effects. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

[31] Houssine Chetto, Maryline Silly, and T Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[32] Liliana Cucu-Grosjean and Joël Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of systems architecture*, 57(5):561–569, 2011.

[33] Robert I Davis and Alan Burns. Robust priority assignment for fixed priority real-time systems. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 3–14, 2007.

[34] 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.

[35] Michael L Dertouzos and Aloysius Ka-Lau Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.

[36] Pierre Dissaux, Olivier Marc, Stéphane Rubini, Christian Fotsing, Vincent Gaudel, Frank Singhoff, Alain Plantec, Vuong Nguyen-Hong, and Hai Nam Tran. The SMART project: Multi-agent scheduling simulation of real-time architectures. *Embedded Real Time Software and Systems*, 2014.

[37] Dipti Diwase, Shraddha Shah, Tushar Diwase, and Priya Rathod. Survey report on memory allocation strategies for real time operating system in context with embedded devices. *IJERA Internet Computing [Online]*, 2(3): 1151–1156, 2012.

[38] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): an introduction. Technical report, Software Engineering Institute, Pittsburgh, 2006.

[39] Gerhard Fohler. How different are offline and online scheduling? *Real-Time Scheduling Open Problems Seminar*, 2011.

[40] Christian Fotsing, Frank Singhoff, Alain Plantec, Vincent Gaudel, Stéphane Rubini, Shuai Li, Hai Nam Tran, Laurent Lemarchand, Pierre Dissaux, and Jérôme Legrand. Cheddar architecture description language. *Lab-STICC technical report*, 2014.

[41] Gernot Gebhard and Sebastian Altmeyer. Optimal task placement to improve cache performance. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 259–268. ACM, 2007.

[42] M González Harbour, JJ Gutiérrez García, JC Palencia Gutiérrez, and JM Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE, 2001.

[43] Joël Goossens and Raymond Devillers. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. *Real-Time Systems*, 13(2):107–126, 1997.

[44] Rick Grehan, Ingo Cyliax, and Robert Moote. *Real-Time Programming: A Guide to 32-Bit Embedded Development with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1998. ISBN 0201485400.

[45] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *WCET*, pages 136–146, 2010.

[46] Steve Heath. *Embedded systems design*. Newnes, 2002.

[47] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis the symta/s approach. *IEEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[48] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[49] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Seyed MH Ashjaei, and Sara Afshar. Support for hierarchical scheduling in freertos. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. IEEE, 2011.

[50] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[51] Mathai Joseph and Paritosh Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[52] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[53] Insup Lee, Joseph YT Leung, and Sang H Son. *Handbook of real-time and embedded systems*. CRC Press, 2007.

[54] Joseph Y-T Leung and ML Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[55] 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.

[56] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007.

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

[58] Will Lunniss, Sebastian Altmeyer, and Robert I Davis. Optimising task layout to increase schedulability via reduced cache related pre-emption delays. In *Proceedings of the 20th ACM International Conference on Real-Time and Network Systems*, pages 161–170, 2012.

[59] William Richard Elgon Lunniss. *Cache Related Pre-emption Delays in Embedded Real-Time Systems*. PhD thesis, University of York, 2014.

[60] James Thomas Martin. *Programming real-time computer systems*. Englewood Cliffs, N.J. : Prentice-Hall, 1965.

[61] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. Dynamic storage allocation for real-time embedded systems. *Proc. of Real-Time System Simposium WIP*, 2003.

[62] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[63] Andreas Menychtas, Dimosthenis Kyriazis, and Konstantinos Tserpes. Real-time reconfiguration for guaranteeing qos provisioning levels in grid environments. *Future Generation Computer Systems*, 25(7):779 – 784, 2009. ISSN 0167-739X.

[64] José Carlos Palencia and M González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 26–37. IEEE, 1998.

[65] Rodolfo Pellizzoni and Marco Caccamo. Toward the predictable integration of real-time cots based systems. In *28th International Real-Time Systems Symposium (RTSS)*, pages 73–82. IEEE, 2007.

[66] Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *Proceedings of the 2014 Agile Conference*, AGILE '14, pages 177–188, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5798-9.

[67] Guillaume Phavorin, Pascal Richard, Joël Goossens, Thomas Chapeaux, and Claire Maiza. Scheduling with preemption delays: anomalies and issues. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 109–118. ACM, 2015.

[68] Michael Pidd. Computer simulation in management science. 1998.

[69] Ricardo C Pinto and José Rufino. Towards non-invasive run-time verification of real-time systems. In *26th Euromicro Conf. on Real-Time Systems-WIP Session*, pages 25–28, 2014.

[70] Alain Plantec and Frank Singhoff. Refactoring of an ada 95 library with a meta case tool. In *ACM SIGAda Ada Letters*, volume 26, pages 61–70. ACM, 2006.

[71] José Rufino, Sergio Filipe, Manuel Coutinho, Sérgio Santos, and James Windsor. Arinc 653 interface in rtems. In *Proc. DASIA*, 2007.

[72] Filip Sebek. Cache memories and real-time systems. *MRTC Technincal Report*, 1:37, 2001.

[73] Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, Nov-Dec, 2004.

[74] Kang G Shin and Parameswaran Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82 (1):6–24, 1994.

[75] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8, 2004.

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

[77] Jan Staschulat and Rolf Ernst. Scalable precision cache analysis for preemptive scheduling. In *ACM SIGPLAN Notices*, volume 40, pages 157–165. ACM, 2005.

[78] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 41–48, 2005.

[79] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):7, 2007.

[80] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000.

[81] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2-3):117–134, 1994.

[82] Hiroyuki Tomiyama and Nikil D Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the eighth international workshop on Hardware/software codesign*, pages 67–71. ACM, 2000.

[83] Hai-Nam Tran, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. Addressing cache related preemption delay in fixed priority assignment. In *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2015.

[84] Richard Urunuela, A Deplanche, and Yvon Trinquet. Storm, a simulation tool for real-time multiprocessor scheduling evaluation. In *Proceeding of the 15th Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2010.

[85] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243. IEEE, 2007.

[86] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Sixth IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 328–335, 1999.

[87] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[88] ISO TC184/SC4/WG11 No41 WD. *EXPRESS Language Reference Manual*, 1997.

[89] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[90] Patrick Meumeu Yomsi, Dominique Bertrand, Nicolas Navet, and Robert I Davis. Controller area network (can): Response time analysis with offsets. In *Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop on*, pages 43–52. IEEE, 2012.

Affectation de priorité et simulation d'ordonnancement de systèmes temps réel embarqués avec prise en compte de l'effet des mémoires cache.

Résumé : Les systèmes embarqués en temps réel (RTES) sont soumis à des contraintes temporelles. Dans ces systèmes, l'exactitude du résultat ne dépend pas seulement de l'exactitude logique du calcul, mais aussi de l'instant où ce résultat est produit (Stankovic, 1988). Les systèmes doivent être hautement prévisibles dans le sens où le temps d'exécution pire-cas de chaque tâche doit être déterminé. Ensuite, une analyse d'ordonnancement est effectuée sur le système pour s'assurer qu'il y a suffisamment de ressources pour ordonnancer toutes les tâches. La mémoire cache est un composant matériel utilisé pour réduire l'écart de performances entre le processeur et la mémoire principale. L'intégration de la mémoire cache dans un RTES améliore généralement la performance en terme de temps d'exécution, mais malheureusement, elle peut entraîner une augmentation du coût de préemption et de la variabilité du temps d'exécution. Dans les systèmes avec mémoire cache, plusieurs tâches partagent cette ressource matérielle, ce qui conduit à l'introduction d'un délai de préemption lié au cache (CRPD). Par définition, le CRPD est le délai ajouté au temps d'exécution de la tâche préempté car il doit recharger les blocs de cache évincés par la préemption. Il est donc important de pouvoir prendre en compte le CRPD lors de l'analyse d'ordonnancement. Cette thèse se concentre sur l'étude des effets du CRPD dans les systèmes uni-processeurs, et étend en conséquence des méthodes classiques d'analyse d'ordonnancement. Nous proposons plusieurs algorithmes d'affectation de priorités qui tiennent compte du CRPD. De plus, nous étudions les problèmes liés à la simulation d'ordonnancement intégrant le CRPD et nous établissons deux résultats théoriques qui permettent son utilisation en tant que méthode de vérification. Le travail de cette thèse a permis l'extension de l'outil Cheddar - un analyseur d'ordonnancement open-source. Plusieurs méthodes d'analyse de CRPD ont été également mises en oeuvre dans Cheddar en complément des travaux présentés dans cette thèse.

Mots-clés : Mémoire cache, CRPD, affectation de priorité, simulation d'ordonnancement, systèmes temps réel embarqués.

Cache Memory Aware Priority Assignment and Scheduling Simulation of Real-Time Embedded Systems

Abstract : Real-time embedded systems are subject to timing constraints. In these systems, the total correctness depends not only on the logical correctness of the computation but also on the time in which the result is produced (Stankovic, 1988). The systems must be highly predictable in the sense that the worst case execution time of each task must be determined. Then, scheduling analysis is performed on the system to ensure that there are enough resources to schedule all of the tasks.
Cache memory is a crucial hardware component used to reduce the performance gap between processor and main memory. Integrating cache memory in a RTES generally enhances the whole performance in term of execution time, but unfortunately it can lead to an increase in preemption cost and execution time variability. In systems with cache memory, multiple tasks can share this hardware resource which can lead to cache related preemption delay (CRPD) being introduced. By definition, CRPD is the delay added to the execution time of the preempted task because it has to reload cache blocks evicted by the preemption. It is important to be able to account for CRPD when performing schedulability analysis.
This thesis focuses on studying the effects of CRPD on uniprocessor systems and employs the understanding to extend classical scheduling analysis methods. We propose several priority assignment algorithms that take into account CRPD while assigning priorities to tasks.We investigate problems related to scheduling simulation with CRPD and establish two results that allows the use of scheduling simulation as a verification method. The work in this thesis is made available in Cheddar - an open-source scheduling analyzer. Several CRPD analysis features are also implemented in Cheddar besides the work presented in this thesis.

Keywords : Cache memory, priority assignment, scheduling simulation, CRPD, real-time embedded systems.