Versatile and efficient mixed–criticality scheduling for multi-core processors

Romain Gratia

▶ **To cite this version:**

Romain Gratia. Versatile and efficient mixed–criticality scheduling for multi-core processors. Other [cs.OH]. Télécom ParisTech, 2017. English. NNT : 2017ENST0001 . tel-01844389

HAL Id: tel-01844389
https://pastel.hal.science/tel-01844389

Submitted on 19 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0001

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

présentée et soutenue publiquement par

Romain GRATIA

le 06/01/2017

Une Approche Efficace et Polyvalente pour l'Ordonnancement de Systèmes à Criticité Mixte sur Processeur Multi-Coeurs

Versatile and Efficient Mixed–Criticality Scheduling for Multi–Core Processors

Directeur de thèse : **M. Laurent PAUTET**
Co-encadrement de la thèse :**M. Thomas ROBERT**

Jury

Mme Alix MUNIER,Professeur, Université Pierre et Marie Curie, France	Présidente du jury
M. Laurent GEORGE,Professeur, ESIEE Paris, France	Rapporteur
M. Joël GOOSSENS, Professeur, Université Libre de Bruxelles, Belgique	Rapporteur
Mme Liliana CUCU-GROSJEAN, Chargé de Recherche, INRIA de Paris, France	Examinatrice
Mme Claire PAGETTI, Ingénieur de Recherche, ONERA, France	Examinatrice
M. Frank SINGHOFF, Professeur, Université de Bretagne Occidentale, France	Examinateur
M. Laurent PAUTET, Professeur, TELECOM ParisTech, France	Directeur de thèse
M. Thomas ROBERT, Maître de conférences, TELECOM ParisTech, France	Co-encadrant de thèse

T H È S E

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

© 2017 Romain GRATIA

Acknowledgments

Je voudrais tout d'abord remercier Thomas Robert et Laurent Pautet pour leur encadrement durant ces trois années. Merci pour leurs critiques pertinentes, leur implication dans la thèse, leur patience ainsi que pour leurs nombreux conseils et relectures, notamment durant l'écriture de ce manuscrit.

Je voudrais ensuite remercier Laurent George et Joël Goossens d'avoir accepté de rapporter cette thèse. Merci également aux examinateurs Alix Munier, Liliana Cucu–Grosjean, Claire Pagetti et Frank Singhoff.

Merci à l'Institut de Recherche SystemX de m'avoir offert de très bonnes conditions de travail et de m'avoir permis d'assister à des conférences en France et à l'étranger durant ces trois ans.

Merci à toutes les personnes que j'ai cotoyées tout au long de cette thèse, que ce soit à l'IRT ou au département INFRES à Télécom ParisTech: Smail R., Paolo C., Aymen B., Xavier J., Thomas W., Cédric K., Djamila D., Roberto M., Félix K., Frédéric T., Youssef L., Joris A., Jimmy D.W., Witold K., Matthieu D., Etienne B., Florian B., Paul L., Paul de N., Xavier R., Antoine B., Antoine J., Daniel J., Nicolas M., Antoine B., Julien V. de G., Pierre le M., Basile S., Mohamed B., Sylvain C., Nicolas T. .

Sur un plan plus personnel, merci à mes parents, Brigitte et Michel, ma soeur Coralie et mon frère Matthieu pour leur soutien.

© 2017 Romain GRATIA

© 2017 Romain GRATIA

Résumé

Ce document présente nos contributions aux algorithmes d'ordonnancement à criticité mixte pour multi-processeurs. La correction de l'exécution des applications temps réel critiques est assurée par l'utilisation d'un ordonnancement vérifié à la conception. Dans ce contexte, le dimensionnement des plate-formes d'exécution vise à minimiser le nombre de processeurs nécessaires pour assurer un ordonnancement correct. Ce dimensionnement est affecté par les exigences de sûreté de fonctionnement. Ces exigences poussent à surestimer le temps nécessaire garantissant l'exécution correcte des applications. Il en découle un dimensionnement assez coûteux. Les méthodes d'ordonnancement des systèmes à criticité mixte proposent des compromis sur les garanties d'exécution des applications améliorant le dimensionnement.

Différents compromis ont été proposés mais tous reposent sur la notion de mode d'exécution. Les modes sont ordonnés, et les tâches voient leur temps d'exécution requis croître avec les modes. Cependant, afin de diminuer le dimensionnement du système, seul l'ordonnancement des tâches les plus critiques est garanti. Ce modèle est appelé "discarding". La majorité des algorithmes proposés se limitent à deux modes d'exécutions par simplicité. De plus, les algorithmes les plus efficaces pour multi-processeurs exhibent un nombre élevé de préemptions, ce qui constitue un frein à leur adoption. Finalement, ces algorithmes sont rarement généralisables. Pourtant, la prise en compte de plus de deux modes, ou de tâches aux périodes élastiques permettrait une adoption plus large par le milieu industriel.

L'approche proposée repose sur la séparation des préoccupations entre la prise en compte des modes de fonctionnement, et l'ordonnancement des tâches sur multi-processeurs. Cette méthode permet de concevoir une politique d'ordonnancement efficace et adaptable à différents modèles de systèmes à criticité mixte. Notre approche consiste à transformer un lot de tâches à criticité mixte en un lot de tâches qui n'est plus à criticité mixte. Ceci nous permet d'utiliser un algorithme d'ordonnancement temps réel optimal engendrant peu de préemptions et de migrations, à savoir RUN. Cette approche, appliquée en premier pour le modèle discarding avec deux modes d'exécution, rempli son objectif d'efficacité. Nous illustrons sa généricité en utilisant le même principe pour ordonnancer des systèmes discarding avec plus de deux modes d'exécution. Enfin, une démarche reposant sur la décomposition de tâche permet de généraliser l'approche au cas des tâches élastiques.

Mots-clés: Système temps réel, criticité mixte, multi–processeurs

Abstract

This thesis focuses on the scheduling of mixed-criticality scheduling algorithms for multi-processors. The correctness of the execution of the real-time applications is ensured by a scheduler and is checked during the design phase. The execution platform sizing aims at minimising the number of processors required to ensure this correct scheduling. This sizing is impacted by the safety requirements. Indeed, these requirements tend to overestimate the execution times of the applications to ensure their correct executions. Consequently, the resulting sizing is costly. The mixed-criticality scheduling theory aims at proposing compromises on the guarantees of the execution of the applications to reduce this over-sizing.

Several models of mixed-criticality systems offering different compromises have been proposed but all are based on the use of execution modes. Modes are ordered and tasks have non decreasing execution times in each mode. Yet, to reduce the sizing of the execution platform, only the execution of the most critical tasks is ensured. This model is called the discarding model. For simplicity reasons, most of the mixed-criticality scheduling algorithms are limited to this model. Besides, the most efficient scheduling policies for multi-processors entail too many preemptions and migrations to be actually used. Finally, they are rarely generalised to handle different models of mixed-criticality systems. However, the handling of more than two execution modes or of tasks with elastic periods would make such solutions more attractive for the industry.

The approach proposed in this thesis is based on the separation of concerns between handling the execution modes and the scheduling of the tasks on the multi-processors. With this approach, we achieve to design an efficient scheduling policy that schedules different models of mixed-criticality systems. It consists in performing the transformation of a mixed-criticality task set into a non mixed-criticality one. We then schedule this task set by using an optimal hard real-time scheduling algorithm that entails few preemptions and migrations: RUN. We first apply our approach on the discarding model with two execution modes. The results show the efficiency of our approach for such model. Then, we demonstrate the versatility of our approach by scheduling systems of the discarding model with more than two execution modes. Finally, by using a method based on the decomposition of task execution, our approach can schedule systems based on elastic tasks.

Keywords: Real–time systems, mixed–criticality, multi–processors

Table of Contents

© 2017 Romain GRATIA

© 2017 Romain GRATIA

© 2017 Romain GRATIA

List of figures

© 2017 Romain GRATIA

© 2017 Romain GRATIA

© 2017 Romain GRATIA

1 Introduction

Table of Contents

This document presents findings and contributions of the PhD study carried out at IRT SystemX and Telecom ParisTech (full publication list can be found p 189). This chapter briefly summarises the motivations and contributions of our study in the field of real-time scheduling theory.

1.1 Context and motivations overview

This thesis has been carried out within the project "Électronique et Logiciel pour l'Automobile" (ELA) at the Institute for Technological Research SystemX. This project aimed at proposing new approaches and tools to address the challenges faced by the automotive industry. Since a decade, the number of in-car embedded applications increased drastically. Given this trend, cost reduction concerns at design and production levels prompt embedded system engineers to rethink their practices.

Our thesis focuses on scheduling methods for safety critical real–time applications that leverage sizing issues encountered on multi–processor platforms.

When designing safety critical and time constrained software, the scheduling approach (e.g. the allocation of processor execution time to applications) aims at guaranteeing integrity and timeliness. Hence, engineers ensure at design time that application provided

execution time is sufficient, and not arbitrarily shortened because of the interfering execution of other applications. Therefore, the isolation concerns have been predominant to the definition of the federated architecture: a network of processors, each executing a single application. Its design and production cost turns out to be an unbearable constraint for the future generations of cars.

Hence, researchers and engineers considered compromises on the isolation guarantees to optimise the hardware sizing. A first step has been to gather as many applications on a single node and to ensure isolation through partitioned access time. Despite its potential gain on sizing, this approach based on partitioned architectures offers limited benefits.

The low impact of this approach not only comes from the poor performances of scheduling approaches based on a partitioned allocation of processor execution time. It also comes from the way estimates of task's Worst Case Execution Time (WCET) are determined. In this case, WCETs are estimated only assuming the highest level of guarantee. But these estimations can be performed based on assumptions that provide different levels of guarantee and hence different estimates. In such critical systems, the more important (e.g. critical) the tasks, the more pessimistic the WCETs. Considering WCET estimates adapted to criticality is a first step to achieve real gains on sizing side. Yet, the true gain lies in accepting that some applications may not be scheduled at all or with the expected frequency. Such scheduling approaches belong to the Mixed–Criticality Scheduling theory.

The contributions of this thesis lie more specifically in the field of Mixed–Criticality Scheduling. Different estimates of WCET are provided for each application. At run-time, a Timing Failure Event (TFE) occurs when a task does not respect an estimate of its WCET (e.g. exhibits a higher execution time than expected). On such an event, the execution timing parameters of the whole task set are either decreased or increased depending on their criticality. It offers the opportunity to transfer the computing power from the low critical tasks to the high critical tasks. Yet, existing solutions lack of maturity concerning multi-processor scheduling, either providing poor schedulability performance, or relying on approach entailing many preemptions and migrations. Such phenomena are known to drastically impair the applicability of such schedulers. In this thesis, we define and evaluate GMC–RUN, a scheduling algorithm that handles various task models of mixed–criticality systems.

1.2 Contributions

Our first contribution consists of the definition of a new kind of execution server, a modal server. It allows us to transform the scheduling problem in two sub problems: i) find effi-

© 2017 Romain GRATIA

cient uniprocessor mixed–criticality schedules within execution servers, ii) schedule these execution servers on multi-processor platforms. It allows us reusing optimal schedulers for multi-processor platforms as parts of our solution.

In our second contribution, we instantiated the approach for two levels of criticality, HI and LO, assuming LO tasks are discarded upon TFE (discarding model). The combination of Reduction to UNiprocessor (RUN) optimal scheduler with our *modal execution servers* appeared highly efficient. Its worst case and average case performances rank our approach second in terms of schedulability efficiency, and first with respect to the number of preemptions and migrations (even compared with contributions issued simultaneously to the thesis).

The third contribution copes with mixed–criticality scheduling problems that do not restrict the number of criticality levels. This extension relies on a recursive approach to reduce successively the number of criticality levels (taking advantage of the two levels solution). Hence, GMC–RUN is a generic approach that allows us to schedule mixed-criticality systems with any number of criticality levels. We assessed the performances of this extension for three levels compared to the two levels approach. Results highlight conditions for significant sizing gains compared to the two levels approach. Identifying such conditions is the key to an efficient use of this generalised setting, and a pre–requisite to its adoption.

The last contribution generalises the concept of modal server to cope with the *elastic task model* variant of mixed–criticality task model. It illustrates the versatility of our approach, that is its capacity to handle different models. The elastic model is certainly more complex than the discarding model we initially considered. It a priori offers lesser sizing gain compared to the discarding model. However, it corresponds more accurately to the industrial needs.

Hence, this thesis core contribution is a mixed–criticality scheduler that is versatile, and efficient with respect to usual multiprocessors scheduling criteria.

1.3 Chapter content summaries

This thesis is structured as follows.

Chapter 2 settles the context of this study. It aims at providing sufficient background knowledge to define the problematic of the thesis. It contains a brief description of the role of real-time scheduling within safety critical system design. It highlights the purpose of mixed criticality scheduling: reducing sizing cost despite constraints derived from strict

© 2017 Romain GRATIA

timing and safety concerns. Finally, it points out the features and limitations of existing works, anterior and simultaneous to this study.

Chapters 3 and 4 define respectively the problematic tackled, and the proposed approach.

Chapter 3 states explicitly our objectives from observations made in chapter 2. In particular, it highlights that mixed–criticality schedulers appear to be extensions of simpler schedulers and inherit from their efficiency. Based on this observation, we claim that an extension of a scheduler such as Reduction to UNiprocessor (RUN) would produce a mixed–criticality scheduler that efficiently copes with preemptions and migrations on multi-processors.

Chapter 4 details how the approach relies on a top level scheduler and execution servers with their low level scheduler. These execution servers manage tasks and their possibly varying needs in terms of processing power. The purpose of these servers is to conceal variability of WCETs. The top level scheduler schedules these servers as efficiently as possible on the different processors. This chapter motivates the application of this principle to three instances of mixed–criticality scheduling problems of increasing complexity.

Chapter 5 details the execution server behaviour used to reduce mixed criticality task scheduling problems to a non mixed–criticiality scheduling problem. These servers are called modal servers, and their behavior is first described for two criticality levels.

Chapter 6 explains how modal servers can be used to schedule a task set with two criticality levels applying the so-called discarding degradation model on multi-processors. The chapter first justifies the choice of RUN as top level scheduler. Then, it explains how tasks are allocated to modal servers. A method based on genetic algorithms minimises the required number of processors by exploring possible allocations. Finally, we provide the algorithm speed–up factor to assess its worst case performances. Then, we compute the schedulability ratio and the number of preemptions entailed on randomly generated task sets.

Chapter 7 presents an extension of the approach to schedule mixed–criticality systems with more than two criticality levels. We first extend the definition of modal servers to handle more than two criticality levels. Then, we present an iterative process to resolve the optimisation problem to minimise the required processing power. Finally, we assess our approach by comparing the performances of the scheduler for three levels to its performances on the same task set reduced to a two levels model.

Chapter 8 explains how elastic tasks can also be scheduled with simple adaption of the proposed modal server. We first transform the task set from the elastic task model into an equivalent one compliant with the discarding task model. Thus, we allocate an elastic

© 2017 Romain GRATIA

task (with its different periods and its single budget per mode) to two execution servers (with their own single periods and different budgets per mode). Then, we apply GMC–RUN, our generic approach, to deal with this new task set. As a result, this transformation increases the number of tasks and therefore the number of preemptions. But it enforces the scheduling of the elastic task model in its most general definition.

Chapter 9 is the conclusion of this document. It summarises contributions and presents possible future works.

© 2017 Romain GRATIA

© 2017 Romain GRATIA

2 Industrial concerns and related works

TABLE OF CONTENTS

This chapter refines the description of the context and motivations presented in the introduction. In particular, it provides details on the interest of real-time scheduling theory within the design processes of embedded safety critical systems. First, engineering objectives and technological trends are discussed. We also recall some real-time scheduling principles. Up-coming challenges are then detailed, and mixed–criticality approaches are presented as promising solutions. Finally, the chapter discusses limitations of existing approaches in the real-time mixed–criticality scheduling domain. At last, we present the foundations of the problem statement chapter.

2.1 Industrial context and motivations

This thesis has been carried out within the project "Électronique et Logiciel pour l'Automobile" (ELA) at the Institute for Technological Research SystemX. Major companies of automotive industry, such as PSA, Renault and Valéo, Continental, SMILE (as ex Openwide), collaborated with academic institutions Télécom ParisTech, and Université Paris 6 in this project.

The project aimed at defining and developing new methods and tools to address the challenges faced by the automotive industry. This section recalls the role of scheduling into the design process. It allows understanding limitations of current approaches.

2.1.1 Embedded System design: the sizing problem

Recent changes encountered by the automotive industry are so important that they may require major changes in their engineering practices.

A car is said to be a safety critical system because its use may put its passengers or its environment into hazardous situations. With respect to cars, passengers are clearly the main concern of designers. Such a high level requirement, once refined, impacts many aspects of the system design. Car subsystems can be classified into categories representing their impact on car safety. These categories are usually ordered into levels of increasing importance, called Automotive Safety Criticality Levels (ASIL). It helps defining safety requirements once for all subsystem belonging to a criticality level. It is a usual practice in safety engineering, as described in [1]. The higher the criticality level, the stronger the impact of safety requirement. Thus, for highest criticality levels no trade-off on safety is allowed, whereas on lower one, some compromises may be possible.

In this context, car manufacturers are faced to contradictory requirements when designing vehicles. On the one hand, car should integrate more and more functionalities such as navigation, entertainment, logs of maintenance, and still be safe. This trend is amplified with the growing demand for entertainment and advanced driver assistance systems (so called ADAS). Obviously, such functionalities lead to non trivial additional costs in design and production phases. On the other hand, the competition among manufacturers urges them to cut costs at each step of the development or production process. Yet, actions taken to reduce cost should not impair system safety or its attractiveness. Hence, manufacturers explore approaches that would help them to take advantage of new technologies to reduce cost, or improve car functionalities without impairing system safety.

Some elementary aspects of embedded software design need to be recalled to understand what are these changes. The design process of such system usually relies on three steps: i) define use cases and functionalities; ii) define system architecture, interfaces, and dependencies between the components of the architecture; iii) implement components and integrate them. All these steps contain validation or verification activities to cope with safety or quality requirements. One of the most complex issues with such embedded system is the sizing problem. Solving sizing problems is to determine vehicle attributes such as its size, weight, available electrical power, or amount of embedded processors. Some of these attributes represent resources on which other components depend. Hence, wrong sizing would affect in this case the ability of these components to deliver their functionalities.

© 2017 Romain GRATIA

First, entertainment and ADAS implementations depend on complex software applications. Second, past decades have seen the transition from analogical or hydraulic implementation of several functionalities to digital implementations like in engine control, steering, braking and diagnostic functions, described in [2] as the x-by-wire approach. Hence, the consequence is that software applications are now an important part of the vehicle. All these software components represent since past decade a significant share, 30%, of the value creation in car industry as reported by experts [3]. It means that software is a part as important as the engine in current vehicles.

In order to ensure that software implemented functionalities are correctly executed, one has to ensure that sufficient storage, and processing power is embedded. The more storage and processor embedded, the more expensive the car production. First, processing power mostly affects the timing of applications, their availability and responsiveness. Second, even if applications are not safety critical, their impact on car vehicle added value requires that reasonable guarantees on their responsiveness are provided. However, even a slight decrease on embedded processing power may have a significant economical impact as thousands of pieces are produced. Hence, designers may weight the pros and cons to lower processing power at the expense of these features. Taking such a decision is particularly difficult as an incorrect sizing may only be detected at run-time if no particular verification effort is done. Obtaining guarantees that the processing power sizing does not impair software worthiness or dependability requires determining the sufficient processing power ensuring timely execution of applications. Yet, the meaning of *sufficient* varies drastically depending on the functionalities, but also on the selected design approach.

Design patterns and architectures have been proposed to ease embedded software design. These architectures define the unit of activity and its dependency on the hardware. It thus eases the sizing process at the scale of each computing node, and more globally at the scale of the whole car. Standards have also been proposed to guide engineers when using some of these architectures.

2.1.2 Patterns and Architectures to ease integration and sizing

The design patterns and architectures enforced during the design process often represent a compromise between different requirements. Such architectures may become irrelevant as soon as the core assumptions or requirements they depend on are no longer valid. This subsection presents design patterns and architecture that impact strongly processing power sizing.

© 2017 Romain GRATIA

First, there is a design pattern that organises software applications in units of sequential activity: tasks. Such models ease analyses about correct sizing of processing resources.

The concept of task is used to define behavioral and timing requirements for a piece of sequential code, called task body. An execution of the task body is usually called a job and a task is an infinite sequence of jobs. These models mainly differ on three aspects: triggering conditions (called activations), timing constraints, and their processing power requirements, mostly represented with their Worst Case Execution Times. Triggering conditions are basically either driven by a sampling logic, or reactive logic. The first case leads to trigger jobs at time instants that can be computed prior the execution. The second case leads to trigger job execution only as a reaction to some external event (button pressed, detection of an obstacle, a change in tire pressure...). Timing constraints are usually defined as deadlines on job execution completion time. Tasks with such timing constraints are usually called real–time constraints. The last aspect characterising a task is an estimate of the processing power necessary for a job to complete its execution. In [3], authors describe the different task models considered to design a vehicle.

Once these three aspects are known, it is possible to study whether a set of tasks can be correctly deployed on a set of processors. Scheduling theory and more precisely in real–time scheduling theory provides answers depending on the considered task model. In this context, verification procedures have been designed to ensure if tasks access processors as expected in their task model, and respect all their timing constraints. Such procedures are called schedulability tests. In order to limit design decisions that need to be checked, architectures have been defined. Such patterns select a task model, a scheduling approach and its associated test. Most architectures also rely on dedicated services on the execution platform, processor and operating system.

The first architecture is the federated architecture. Its motivation is to avoid issues related to sharing processors between tasks of different applications. Such an architecture relies on many weak processing units, essentially micro-controllers that interact passing messages through wired networks. In this case, the software architecture within a processing unit called Electronic Control Unit (ECU) is often trivial and not detailed (most often applications are single tasked). Yet, the growth of the number of embedded applications leads to consider more than hundred computing units for most advanced vehicles. Increased interactions between these applications entail massive communications between processing units. Scheduling disciplines have been proposed to organize communication. Yet, the larger the number of ECUs, the more complex ensuring that the application dependability is not affected by the network insufficient throughput, or message transfer latencies, as explained in [4].

© 2017 Romain GRATIA

An alternative architecture is to use processing units with far higher processing power. The principle is to concentrate applications as much as possible on a same execution platform, processor and operating system, that supports multi–tasking execution. For years, only tasks with similar criticality, triggering conditions and timing constraints have been deployed together on a same processor. Respecting this constraint eases verifying that tasks have enough processing power, but also avoids complex analysis with respect to safety. Dedicated operating systems are used to deploy such architectures. For instance, OSEK VDX is the specification of the most used operating system for real–time tasks for automotive software, [5]. Despite its potential, the amount of tasks that can be gathered with identical criticality level remains limited. The so–called integrated architecture has been proposed as a refinement of the multi–tasking approach. With respect to scheduling issues, it proposes an approach that ensures that the time intervals during which a task accesses the processor are predefined and cannot be modified at run time. Such a strategy limits the risk that a misbehaving task prevents another task to deliver its service. This architecture is now widely used to design and deploy software in civil aeronautic industry. Operating systems implementing this principle also ensure memory isolation, preventing task data alterations. Finally, it is important to notice that such architectures have been proposed to simplify the deployment problem with respect to all the conflicting objectives: safety and scheduling. In practice, they prioritised, without compromises, safety requirements handling. With the increasing number of ECUs, such situation may no longer be affordable.

Another observation can be made for all these architectures. The key parameter to determine whether a deployment is relevant or not is the estimate of task Worst Case Execution Time (WCET). This value is the upper bound of the execution time of a task body.

2.1.3 WCET, a key parameter for sizing

WCET values for a set of tasks directly affect the number of processors required to schedule it correctly. Not surprisingly, WCET estimations have always been frustrating for engineers. A huge gap exists between average execution time and worst case execution time estimates. The execution time variability can be due mainly to the different possible paths of execution in the application code itself. Each path is likely to correspond to different numbers of instructions. The second cause lies in the fact that processors do not handle instructions in constant time. Moreover, a single instruction could see its execution

© 2017 Romain GRATIA

time multiplied by 100 depending on previously executed ones. Causes are mainly the behaviours of the processor pipeline, caches, and memory access mechanisms.

A source of difficulties in the estimation of the WCETs are the timing anomalies. These are situations that represent local worst cases but ultimately do not result in the global worst cases. These situations have been surveyed for different task models and processor technologies. A survey focused on single core processors, [6]. The usual anomalies that could impact execution time has been extended to account those specific to multi-cores processors, [7]. Different approaches exist ranging from empiric settings to detailed models combining information on processor state and application binary code, [8].

In any case, determining if anomalies will occur is non trivial and leads to approximate the values of WCETs. It is particularly true when timing anomalies are caused by other task executed simultaneously on the multi-processors. Methods mainly differ on the kind of anomalies they can handle, and the way they approximate the WCET. For these reasons, two methods that do not perform the same approximation would produce different values of WCET. It appears that multi–cores processors rely on design principles that significantly increase execution time variability. When anomaly timing effects are hardly measurable, the WCET estimates can simply be increased by a fixed percentage depending on the safety requirements associated to tasks. This percentage is called the safety margin. Thus, schedulers can tolerate unforeseen timing anomalies. This practice tends to produce large over approximations.

Hence, approximation on WCET suffers from this strong variability and exhibits highly different values. The more critical a task, the more pessimistic the approximation is expected to be. Conversely, strongly pessimistic approximations of WCETs may not be relevant for non critical tasks. Indeed, such tasks are not essential for the safe execution of the system. Hence, if most important timing anomalies occur only in exceptional situations, then ignoring these cases may yield far lower WCET approximations for these tasks. Considering such lower values is clearly a compromise between the task timeliness and its cost in terms of processing power.

The conclusion of these observations is that WCET estimates are parameters pretty difficult to define as they impact both sizing and dependability. Their definition represents a difficult compromise.

Next section presents the challenges derived from this context.

2.2 Challenges

In this section, we detail the challenges motivating this thesis.

© 2017 Romain GRATIA

2.2.1 Can we avoid or tolerate timing anomalies ?

Collocating tasks with different criticality levels on a processor should not impair the dependability of critical tasks. When timing anomalies prevent critical tasks to comply with their timing requirements, one can consider such phenomena as faults. Two approaches are usually possible concerning fault treatment: elimination, or tolerance. A part of the challenge is to determine which approach would be the most relevant.

Some preliminary results exist with respect to this question. They mainly rely on the analysis of the causes of anomalies. Anomalies on multi-core platforms are due to the use of shared hardware resources to link execution cores and main memory. Main shared resources are the shared caches and memory access bus. It means that the memory access latency could increase when there are contentions on shared resources. Authors in [9] have shown that memory bus contention can lead up to 183% higher execution times for applications on a platform with a shared L2 cache. In [10; 11], authors assessed the impact of contention on memory access bus and cache consistency mechanisms. They reached similar conclusions: impact of contentions can be significantly high on execution times. Hence, such impacts on access latency need to be either eliminated, or tolerated.

Eliminating such interactions relies on resource partitioning at design time and avoiding changes on this allocation at runtime [12; 13]. The elimination of interactions on shared resource relies on variants of Time Division Multiple Access principle, as explained in [14]. Such an approach exhibits satisfactory performances on single core processors. Yet, this principle appears not adapted to multi-core platforms. This principle has been refined for managing bus contention on avionic processor as presented in [15]. The main lesson learned is that the latency variability is greatly decreased but it increases significantly the minimal latency access time. Moreover, deploying such architecture requires advanced understanding on cache behavior to configure the mechanism eliminating interactions. For these reasons, such approach is considered unfit for the automotive context.

Hence, the alternative is to propose approaches to tolerate these anomalies. As explained in the previous section, up to now the simpler solution is to integrate their impact in WCET estimate. Yet, such an approach let even the smallest anomalies propagate up to the scheduler logic without any attempt to limit their effect. AUTOSAR standard proposes to systematically use detection mechanisms to specific timing anomalies to avoid their propagation. Such mechanisms aim at detecting that a task cannot use more than a fixed time budget. If task job are not complete within this time budget various recovery actions can be implemented. Hence, using WCET estimates with limited scope of timing anomalies is no longer a risk for other tasks as this architecture would confine timing

© 2017 Romain GRATIA

anomalies to the faulty task as much as possible. This approach is promising but does not provide any clue on how to exploit best such detection and recovery procedures. It leads to the second challenge.

2.2.2 Can real time scheduler be smarter and safe ?

Last section presented possible approaches to ensure that tasks of distinct criticalities can be deployed on a same processor with WCETs estimated according to their criticality.

Since less criticality tasks are not essential for the safe execution of a system, these tasks could potentially see their timeliness not respected. It would then be safe to consider WCET estimates ignoring some anomalies for lesser criticality tasks. Such situation would be considered exceptional. Thus, it is accepted that these tasks timeliness is no longer a priority. It means that two possible behaviors need to be considered. The first case consists in considering exceptional situations never occur. The second case considers that timing anomalies have an exceptionally high impact on task execution time. In this case, timeliness requirements for tasks of lesser criticality can be reconsidered.

Real time scheduling community takes advantage of this situation by defining a scheduling approach called mixed–criticality scheduling.

The seminal paper of Vestal in 2007, [16] defines accurately this scheduling problem. It is actually a non trivial extension of the scheduling approach relying on a single WCET value for each task. This scheduling approach leads to consider either scheduling all tasks with reasonably low WCET approximation, or to only execute highest criticality tasks considering highest over approximations of their WCET.

Finally, the challenge is to thus determine whether using such scheduling approaches actually worth the effort. Can we find a scheduling algorithm that offers opportunities to significantly decrease the number of processors to be embedded and still provides strong guarantees on most critical tasks schedulability.

Next section briefly recalls main results in real time scheduling for multi-core processors, and provides a survey of most recent results in the field of mixed–criticality scheduling.

2.3 Scheduling of critical embedded systems

This section introduces tasks models that in fact define scheduling problems. Then, the section recalls the notion of schedulability, and usual assessment criteria used in the community. Finally, we discuss limitations of algorithms handling presented task models. In

© 2017 Romain GRATIA

real time scheduling theory, the difference between multi-cores and multi-processors is not made. Hence, we only use multi-processor to identify the task set execution platform. Similarly, instead of using execution core to identify a processing unit, we use the term processor.

2.3.1 Real–time task models

The definition of a task model is the usual approach to describe a scheduling problem. The model defines the activity represented by a task through parameters identifying its activations, information on its worst case execution time, and its deadlines.

Periodic and sporadic task models

The periodic task model and the sporadic task one are the most widespread task models in real-time scheduling theory. It is partly for its simplicity that this model is one of the most used. The periodic task model relies on very few parameters.

Definition 1 (Periodic Task)**.** *A periodic task is defined through two parameters, usually denoted T, its period, and its budget, usually denoted C.*

Yet different instances of this model exist. We can further assume that our periodic tasks are synchronous, independent and with implicit deadlines. The meaning of these assumptions being:

- Synchronous tasks: all tasks are started at the same time when the system starts executing. Hence, considering these first activations occur at time 0, next activations times of a task of period T would be in the set $T\mathbb{N}, T\mathbb{N} = \{k \cdot T | k \in \mathbb{N}\}$.
- Independent tasks: there are no precedence constraints between tasks, that is the execution of a task does not require that some specific tasks have completed their executions.
- Implicit deadline: the relative deadline of a task is equal to its period. Therefore, the absolute deadline of a job activated at time t, of a periodic task of period T, is $t + T$. The absolute deadline of a job is equal to the activation time of the next job.

We note Γ a set of such tasks. Hence, if the size of Γ, denoted $|\Gamma|$ is n, then it contains $\tau_1, ..., \tau_n$. The budget should be sufficient to ensure that if the task executes on the processor for a duration equal to its budget then it completes the execution of its body (of a job). Any value lower than the task WCET may not be sufficient to allow the task to complete

© 2017 Romain GRATIA

all its jobs. A budget value higher than the WCET can be considered as wasting resources, since it would never be completely used.

The second model popular among real-time system designers is the sporadic task model. The main difference of this model with respect to the periodic task model lies in the definition of activation times that are event triggered. There is an uncertainty on activation times. Yet, some assumptions on consecutive activation times allow defining a tractable scheduling problem. The sporadic model assumes that there is a non zero lower bound on the duration between any two consecutive activations. Other models add constraints of precedence or mutual exclusion between task execution. Such models elaborate the notion of task. In our case, such dependencies do not exist since our tasks are independent. Despite the attractiveness of covering many models, we decided to consider the periodic task model during this study. In particular, the synchronous independent task model with implicit deadlines. This model is one of the most widely used in the mixed–criticality approaches to simplify the mixed–criticality scheduling problem.

Selecting a task model defines most of the constraints corresponding to the scheduling problem. Usually, the number of processors available in the embedded hardware is known and remains constant, and is noted m. The scheduling problem generally consists in checking whether a task set can be scheduled on the m processors by a given scheduling policy. Yet, there exist scheduling policies that can schedule tasks on the smallest possible number of processors. In any case, the scheduler determines for each time instant at most m tasks to schedule. When all processors are identical, one can simply select for each time instant a subset of the task set that is no larger than the number of available processors. In this case, the scheduler does not assign tasks to processors. This decision needs to be taken in a second step.

The mixed–criticality task sets are mostly extensions of the periodic and sporadic models. We now describe the extension of the periodic task model.

Mixed–criticality task model

Mixed–criticality task model is extension an extension of the real–time task model. This model alters in deep the task behaviour. Indeed, with this model periods, deadlines and budgets are not unique for each task. These timing parameters now depend on the criticality level of the task. Hence, it is necessary to know each task criticality level. Criticality levels are considered fully ordered.

These criticality levels have to reflect the criticality of the timeliness and availability of the task (is it running and if so, is the result produced in time). In his paper [16], Vestal simplified the definition directly referring to Safety Integrity Levels. Yet, authors in [17]

© 2017 Romain GRATIA

pointed out that the task criticality level could differ from the task SIL. In practice, not all safety critical tasks have to comply with strict timeliness and high availability. In the remainder, we assume that the system designer assigns criticality levels to tasks expecting a gain of using mixed–criticality scheduling on multi-processors instead of simpler non mixed–criticality schemes. Such gains mainly concern the required processing power to schedule a system. Indeed, in mixed–criticality systems, it is acceptable to degrade the execution of tasks with low criticality levels to ensure those of higher criticality tasks. The probability of the degradation of a task execution being dependent on the confidence associated to the task timing parameters. A system designer can thus expect that adding a criticality level helps better tuning and respecting the execution requirements of the tasks compared to the initial system configuration. Indeed, adding a criticality level allows the engineer to use more appropriate timing parameter for a subset of tasks and possibly reduce the probability of the degradation of their execution.

Hence, the use of more or less criticality levels can be used to improve either the availability and the respect of timeliness of a subset of tasks or to reduce the required processing power.

The seminal paper provides a generic definition of criticality levels but illustrates it in the case of tasks exhibiting only two distinct levels, denoted LO and HI. HI level is the level representing critical tasks. Most contributions in mixed–criticality domain keep considering only two criticality levels. The periodic task model is extended as follows. Recall in section 2.2.2 we saw that in order to leverage the issue represented by WCET computation, it has been proposed to consider potentially under-approximated WCET values as long as the critical task schedulability is always ensured. Hence, mixed–criticality periodic tasks may have different values for their periods and budgets.

Task set definition

Definition 2 (Mixed–Criticality Task Set)**.** *A mixed–criticality periodic task set,* $\tau_1, ..., \tau_n$*, is characterized by a set of 5–tuples,* $(T_i(LO), T_i(HI), \chi_i, C_i(LO), C_i(HI))$ *with* $i \leq n$*, such that:*

- χ_i *the criticality level of the task taken in* $\{LO, HI\}$.
- $T_i(LO)$*,* $T_i(HI)$ *period values corresponding to levels LO and HI.*
- $C_i(LO), C_i(HI)$ *the execution time budget values of the task corresponding to levels LO and HI.*

© 2017 Romain GRATIA

We introduce additional notations that are used in the whole document.

- A LO task, and a HI task, are respectively tasks with criticality levels LO, and HI.
- If a task τ_i name is suffixed with $\{\chi_i = LO\}$, or $\{\chi_i = HI\}$, it means τ_i is respectively a LO task or a HI task.
- the utilisation of a periodic task of period T and budget C is the normalized processing power it requires, $U = \frac{C}{T}$. In the case of mixed–criticality scheduling, there are as many utilisation values as criticality levels. That is $U_i(LO) = \frac{C_i(LO)}{T_i}$ and $U_i(HI) = \frac{C_i(HI)}{T_i}$.
- utilisation definition can be extended to task sets Γ as follows: $U_\Gamma = \sum_{\tau_i \in \Gamma} U_i$.

The task set expected behaviour depends on the execution time effectively used at run-time to complete task executions. As long as the LO parameter values are sufficient, the parameters used to define task activations and deadlines are the LO parameter values. Whenever a job has not completed its execution, but has been executed for a duration equal to its task LO budget, then the task set expected behaviour changes. This change is called a *timing failure event.*

Definition 3 (Timing Failure Event)**.** *The time at which a job exceeds its task LO time budget without completing its execution is called a Timing Failure Event (TFE).*

It is assumed for any task set that a HI task job cannot execute longer than its $C_i(HI)$ value. This budget value is said safe. When a system encounters major changes in its expected behaviour, the system is said to have distinct operating modes. A change in task set behaviour is thus called a mode change, [18]. Hence, two modes can be defined. Modes are named according to criticality levels as their definition depends on it. In this context, a mode can be seen as configuration of the scheduler that enforces the expected behaviour.

Definition 4 (Execution mode)**.** *Let L be a criticality level, the execution mode of L, denoted L mode, requires that all tasks τ_i with criticality level χ_i such that $\chi_i \geq L$ are scheduled and respect their timing parameters of mode L.*

In practice, it means that activation times, deadlines and budgets are determined in a mode according to parameter values of the corresponding criticality level. During an execution of a task set, if a mode is said active at time t then it means that all tasks of criticality higher than L still have to respect their deadlines and activation times defined

© 2017 Romain GRATIA

by level L parameters. We assume criticality levels are ordered, that is HI mode is more important than LO mode, which is denoted $LO < HI$. A job execution is said to contradict a L mode, if the job has not completed its execution, while its execution time is larger or equal to the budget of level L. In this context, the active mode at time t corresponds to the mode of smallest criticality level in which no job executed for a time larger or equal to the budget of this criticality level at time t. This definition entails that the initial mode is always LO. Moreover, task scheduling on processor should be done so that whenever the Timing Failure Event occurs, all HI task deadlines are respected in the newly entered mode. This has to be achieved despite the possible increase of HI task budgets and the execution time used for LO tasks. We name this property the *schedule continuity property*.

Few more notations are introduced to easily describe the processing power required in each mode by the different type of tasks (LO and HI). As long as there is no ambiguity on the currently studied task set, we introduce the following notation: $\Gamma(LO)$ and $\Gamma(HI)$ denote respectively all LO tasks of the task set, and all HI tasks of the task set. Given this notation, we introduce a notation system to denote the processing power required by these task sets in the two possible execution modes LO and HI.

The utilisation $U_{\Gamma(X)}(Y)$ with X and Y criticality level denotes the utilisation of task set $\Gamma(X)$ with respect to Y mode parameter values. Hence, $U_{\Gamma(LO)}(LO)$ denotes LO tasks utilisation according to LO mode parameters, and $U_{\Gamma(HI)}(LO)$ denotes HI tasks utilisation according to LO mode parameters. Finally, $U_{\Gamma}(LO)$ and $U_{\Gamma}(HI)$ denote the utilisation of the whole task set Γ in LO and HI mode respectively.

While period parameters can be modified, the most widespread model currently used, HI tasks have usually identical periods in HI and LO modes. Only their budgets are changing. This can be justified by considering that HI task have to be executed with their timing parameters that ensures the maximum availability both in LO and HI modes. This requires to execute them with their periods that fulfill this objective. Budgets have an impact on the availability if they are insufficient to complete a job. In LO mode, an optimistic assumption is made on their execution times in order to reduce the required processing power to execute the system. If more execution time is required HI tasks can use their HI mode budget after a mode change. HI task availability objectives are thus always ensured.

- $T_i(LO) = T_i(HI)$
- $C_i(LO) \leq C_i(HI)$.

This definition leaves unspecified the expected behaviour of task of criticality lower than the mode level. Their timing requirements are usually degraded into weaker constraints.

© 2017 Romain GRATIA

The definition of the expected behaviour in the L mode of task of criticality lesser than L is called the execution degradation model.

Execution Degradation Model

Recall that mixed–criticality aimed at providing the mean to lower processing power required to schedule a set of tasks without impairing most critical task execution. This is achieved through the definition of two execution modes, one in which all tasks are executed using optimistic timing parameters, and another one in which only the most critical tasks see their executions ensured. Hence, it means that when the system runs in the highest mode (HI mode), it is necessary to accept compromises on LO task timeliness or even on their availability.

Two degradation models can be considered. The first degradation model, called the discarding task model, results in the complete stop of the execution of LO tasks. The second one, called the elastic task model, allows to gracefully degrade the execution of LO tasks.

The discarding task model: it assumes that, once a TFE occurs, all current and future jobs of tasks of criticality level LO can be stopped. Mixed–criticality task models enforcing this behaviour in HI mode are called discarding task models. This is the most spread degradation model. In this context, task periods are considered constant for all tasks, and tasks of criticality LO have the same LO and HI budgets: $T_i(LO) = T_i(HI)$, and $C_i(LO) = C_i(HI)$.

Hence, after a TFE, tasks of criticality LO are no longer available. Such degradation model can be found too extreme, that is why an alternative model has been proposed.

The elastic task model: this model assumes that tasks of criticality LO should always be executed. However, their parameters could represent a lower requirement in terms of processing power. Task utilisation as detailed before is a synthetic representation of processing needs. Hence, the elastic model still requires tasks of criticality LO to be executed but requires less processor utilisation for them. This goal could be achieved either by decreasing the budget in HI mode, or by increasing the task period (and thus deadline). The first case is also called imprecise computation and has received some attention prior to mixed–criticality scheduling definition, e.g. in [19]. Yet, the imprecise computation model is unfit as the latency to detect timing failure may not meet the desired objective to save processing power for task of higher criticality. Thus, we mainly focus on the case where the budget remains the same, and the period increases. It leads to the following assumptions timing parameters of each task τ_i: $T_i(LO) \geq T_i(HI)$ and $C_i(LO) = C_i(HI)$.

© 2017 Romain GRATIA

Note that there exist mechanisms to switch back to the LO mode [20; 21; 22] and hence to execute again LO task as before a TFE.

For completeness purpose, it is important to point out that a degradation model that does not degrade execution requirements of LO tasks corresponds to a multiple criticality model (or multi–criticality model for short).

Multi–criticality systems

In this kind of systems, tasks are modelled as mixed–criticality tasks. But execution modes and execution degradation models are not used. Instead, each task is always executed following its own criticality level. Hence, a HI task is always executed with the execution time budget corresponding to its criticality level, that is $C(HI)$. A LO task is also always executed with the execution time budget of its criticality level, that is $C(LO)$. Since there is no mode change LO tasks are never stopped executing or changed of periods. A LO task is always executed with its period $T(LO)$. This forms what has been called multi–criticality systems [16].

The mixed–criticality task model describes how the tasks behave and their required execution resources. The scheduling algorithm provides a schedule, the next section presents correctness and quality criteria used to assess schedules, and more generally schedulers.

2.3.2 Scheduling algorithms, schedulability and performance criteria

Let first recall some basic definitions.

Background on scheduling algorithms for multi–processors

Recall that a scheduling algorithm or scheduler is an algorithm that aims at determining in which order the tasks are executed and on which processor.

To determine the order of execution of the tasks or jobs, a scheduling algorithm assigns priorities. These priorities can be fixed to the task, we then call such scheduling algorithm fixed task priority scheduling algorithm. They can be fixed to a job, the algorithm is then said to be a fixed job priority scheduling algorithm. Or they can be dynamic scheduling algorithms, that is algorithm that can change job priorities during their executions.

Concerning the allocation of tasks an algorithm can allow or not that a task or a job changes of processor during the execution. This is called a migration. This leads to three types of multi–processor scheduling algorithms: the partitioned and global scheduling algorithms [23] and semi–partitioned algorithms.

© 2017 Romain GRATIA

In partitioned algorithms, tasks are allocated to processors prior the execution and no migration is allowed.

In global real–time scheduling algorithms, tasks can be executed on any of the available processors, that is migration of tasks or jobs, is allowed between processors.

In semi–partitioned scheduling algorithms, tasks that can migrate is determined prior the execution. These migrations can be performed in two different ways. Either each job are executed on different processors. This approach can be referred to as *job portion migration* [24]. Or each job of a task are executed on a single processor but different jobs are not necessarily executed on the same processor. This can be referred to as the *restricted migration approach* [25]. In both case, the processors on which these tasks can migrate are also determined prior the execution.

Another criteria can be used to classify scheduling algorithms. A scheduling algorithm is said preemptive if a task of higher priority can preempt the execution another task. If not it is said to be non–preemptive. All the considered algorithms in this thesis are preemptive algorithms.

Background on schedulability

The first concept allows describing an execution of a task set without making assumption on how it is scheduled.

Definition 5 (Task set execution)**.** *The execution of a task set is the definition of the execution time of each job of each task in the task set.*

Each execution could be scheduled differently. Job execution time has to comply with the time budget parameter(s) of tasks. Hence, a task set execution is selected, one can define the schedule proposed by the scheduler.

Definition 6 (A schedule)**.** *A schedule is a function that returns for any instant t a set of tasks. All tasks from this set that are allowed to be executed on a processor at time t.*

The size of sets returned by this function is bounded by the number of available processors. Hence, the maximal size of returned set defines the minimal number on processors on which the schedule can be applied.

A refined definition could consider that a schedule is a tuple of functions, one for each available processor. Each function returns a pre–determined set of tasks to execute on a single processor at time t. In this context, a scheduler may propose to a task to use a processor while no job of the task can be executed. For instance, these jobs may have

© 2017 Romain GRATIA

already completed. Conversely, the scheduler can adapt its schedules to task set execution to avoid granting processors to jobs that cannot be executed.

We need basic validity definitions to identify schedules respecting deadlines, and schedulers that would prevent such situations to occur.

Definition 7 (Valid schedule)**.** *A valid schedule ensures for each job of each task that the duration during which the task is scheduled between its activation time and deadline is sufficient to complete job execution.*

This condition is synonymous to require that all jobs complete their execution before (or at) their deadline. If all schedules provided by a scheduler for a task set are valid, then the task set is said schedulable. A procedure to verify such a condition at design time is called a schedulabilty test. These tests are most often sufficient conditions. Not passing the tests does not necessarily mean that the scheduler provides non valid schedules. Yet, the test cannot ensure it at design time. Necessary and sufficient conditions provide more useful information. Failing the schedulability test means that there are executions of the task set for which schedules are not valid.

The notion of feasibility of a task is define independently of the scheduler used.

Definition 8 (Feasible task set)**.** *A task set is feasible on m processors if and only if there exists for each task set execution a valid schedule of this task set on m processor.*

Given this definition, we can discuss performance criteria usually considered to assess scheduler. Our purpose is to determine which one are relevant in the context of mixed–criticality task sets.

Performance criteria

Many aspects of the algorithms used to produce or enforce schedules at run–time can be assessed. First, the quality of schedules can be assessed, then the complexity of determining at run-time (said "on-line") the schedule. Each of these aspects could affect the worthiness of a scheduling approach.

Let us first focus on so–called schedulability related criteria. The objective is to determine the capacity of a scheduling algorithm to schedule feasible task sets. Usually such studies do not account for the processing time used by the scheduler to determine and enforce its schedules. Such overseen needs in terms of processing power are usually considered as overheads. Recall that margins were defined to ensure task scheduling could afford some overseen timing anomalies. It is expected that safety margins tolerate scheduling overheads. Yet, the larger these overheads, the larger the margins.

© 2017 Romain GRATIA

Different phenomena contribute to scheduling overheads: the on–line execution of the scheduling algorithm, execution context switch, and migration of execution context to a processor [26; 27; 28; 29]. Notice that usually the two last sources of overhead involve system calls, and affect cache contents (source of more timing anomalies). Preemptions are scheduling event corresponding to the time at which a job execution is stopped to grant its processor to another job. Migration and context switch are particularly more costly when preemptions occur. Hence, we are interested in assessing how many preemptions are produced by scheduling algorithms as it is a top level source of overheads on multiprocessors, [27]

Let us first present schedulability performance criteria ignoring overheads. These criteria are classified in theoretical one, and empirical ones. The first one does not take into account the processing power required by a scheduling algorithm to produce the schedule. It is hence useful to complete the performance assessments with empirical ones that can take into account these overheads. Empirical criteria rely on statistical assessment.

Theoretical criteria When designing a scheduling algorithm system designers seek to have the best possible scheduling performances. An algorithm with the best schedulability performances is said to be optimal towards that objective. It is defined as follows:

Definition 9. *A real–time scheduling algorithm is said optimal with respect to a task model, if any feasible task set on m processors is schedulable by the algorithm on at most the same number of processors.*

For non mixed–criticality task set, schedulability tests for optimal scheduler consist in verifying that the task set utilisation is lower or equal to the processor number.

Yet, finding optimal scheduler is not always granted.

In this case, it is interesting to know how far a scheduling algorithm is from the optimal performances. This can be achieved with the speed–up factor.

The speed-up factor [30] has been extensively used to compare the performances of scheduling algorithms. But before defining what the speed–up factor is, we need to define what is a *clairvoyant optimal scheduling algorithm*:

Definition 10 (Clairvoyant optimal algorithm [30])**.** *A clairvoyant optimal scheduling algorithm is a scheduling algorithm that knows prior execution for how long each job of each task will execute and is able to find a valid schedule for any feasible task set.*

Such clairvoyant scheduling algorithm cannot be implemented. Nonetheless, assessing quantitatively a scheduling algorithm performance with respect to the clairvoyant scheduling algorithm provides a normalized metrics.

© 2017 Romain GRATIA

This quantitative performance indicator is called speed–up factor:

Definition 11 (Speed–up factor [31])**.** *The speed–up factor s for a scheduling algorithm A corresponds to the minimal factor by which the speed of each processor of a set of unit–speed processors has to be increased such that any task set schedulable by a clairvoyant scheduling algorithm becomes schedulable by A.*

The lower speed s, the better the algorithm A. In this context, it is considered that an optimal mixed–criticality scheduling algorithm would have a speed–up factor of 1.

Such a characterization of the schedulability performances only gives an insight on the worst case performances of the scheduling algorithm.

Empirical criteria Empirical assessments are used to know the average performances of a scheduling algorithm. They are usually performed by randomly generating task sets that are thought to be representative of the systems that could effectively be scheduled with the studied algorithm.

Then statistics are build from these results to determine the likelihood that a task set is schedulable by the algorithm. Such an information is not proofs but it allows engineers to decide the risk they take when using such schedulers.

Once the task set samples have been generated, the count of successfully scheduled task sets is used to determine an average schedulability ratio. Different sampling methods have been proposed even for mixed–criticality task sets, [32; 33; 34].

Assessing overheads Task set sampling offers the opportunity to actually compute schedules, and thus assess the preemption count entailed by scheduling algorithms. Computing upper bounds is far too pessimistic, empirical analysis is the most spread way to assess this aspect for multi–processor scheduling algorithms.

An alternative approach is to implement the scheduler and embed it in operating system. Then, the prototype is assessed through monitoring task execution to check timeliness, and also measuring time spent in scheduler code. Several scheduling algorithms have been assessed this way recently [26; 35; 36]. These implementations enable to determine whether a scheduling algorithm can be efficiently implemented and to compare several scheduling algorithms. Negative results only mean that the proposed implementation is inefficient. Similarly positive results are not proofs, and can provide statistical information at best.

Next section presents respectively a state of the art for non mixed–criticality task model and mixed–criticality model. The idea is to show how a scheduler performs on the simpler multi–criticality system model (defined in 2.3.1) compared to the mixed–criticality one.

© 2017 Romain GRATIA

2.3.3 Hard–real scheduling algorithms

We first present scheduling algorithms that do not fully take advantage of the mixed–criticality task model and schedule them as multi–criticality task sets. Indeed, with these algorithms mode changes are not handled. Despite this major limitation, we present them as such algorithms have been used to design mixed–criticality scheduling algorithms handling the mode changes. We present algorithms of the three types previously described, partitioned, and semi–partitioned, and global. For each type of algorithms, we nonetheless present only the algorithms with remarkable performances.

Partitioned scheduling algorithms

The approach of partitioned algorithm can be decomposed in two steps. The first step consists in allocating the tasks to the processor, this is a NP–hard problem. To perform this allocation, heuristics are used to find the best possible allocation, that is the one using the fewest possible number of processors. The most common heuristics are the following:

- First Fit: allocate the task to the first processor with sufficient capacity.
- Best Fit: allocate the task to the processor with the largest utilisation of allocated tasks.
- Worst Fit: allocate the task to processor with the smallest utilisation of allocated tasks.

Tasks can be considered in the order of increasing or decreasing utilisations.

Then the scheduling on each processor is performed by using any uniprocessor scheduling algorithm, such as Earliest Deadline First that is optimal for uni–processor systems [37].

But partitioned scheduling algorithms suffer low theoretical scheduling performances. There is no optimal partitioned scheduling algorithm, and the highest provable utilisation bound for such algorithms is $\frac{m+1}{2}$ [38], where m denotes the number of processors. This limitation comes from the fact that each processors can have idle time and yet this idle is to small on each processor to execute any task but gathered it could be used execute tasks.

Semi–partitioned scheduling algorithms

Semi–partition algorithms aim at mitigating the disadvantage of partitioned algorithms that cannot use the idle time of several processors to execute a same task. This is achieved by allowing a subset of tasks to execute on several processors if they cannot be allocated to a single processor. These tasks are called migrating tasks. The determination of these

© 2017 Romain GRATIA

migrating tasks is performed as for partitioned algorithms during the allocation of tasks to processors prior the execution of the system. Two approaches exist to perform the migration of the tasks between processors. Either each job of a migrating task is executed on different processors, this is referred to as job portion migration [24]. Or each job of a migrating task is executed on a single processor out of a subset of processors. This is referred to as restricted migration approach [25].

Many semi–partitioned algorithms are based on the job portion migration approach. Among these algorithms, *EDF with task splitting and k processors in a group* (EKG) [39] is remarkable as it is optimal for determined configurations. It consists in first splitting the task set in heavy and light tasks. The determination of whether a task is heavy or not depends on a parameter k. During the allocation, heavy tasks are each assigned to a processor. Light tasks are allocated to processors without any heavy tasks. Since the scheduling on each processor is performed with EDF, a task can be assigned to processor if the added up utilisation is lower or equal to 1. If one light task does not fit on a single processors then it is split into two subtasks and allocated to a group of processors. The two subtasks are never executed at the same time thanks to a dispatcher algorithm that is called each time a task activates in a group of processors.

Another optimal approach is the "Notional Processor Scheduling – Fractional capacity" (NPS–F), [40] algorithm. This algorithm is based on the use of *periodic reserves* and *notional processors*. Periodic reserves are basically servers that are used to scheduled tasks following a uni–processors scheduling policy (such as EDF) and executed periodically in accordance with a *timeslot length*. Each reserve is executed periodically for a fixed length of time. These reserves have a resulting utilisation that amounts to the sum of the task utilisations they contain and is then inflated to ensure a correct schedule. They also use the notion of notional processor that describes the execution of a reserve over several processors without it executing at the same time on more than one processor. This notably describes the offset between the start of execution of each reserve so that this latter property is respected. The allocation of tasks is performed in two steps. First the tasks are partitioned in bins, as during a usual partitioned scheduling algorithm. Then each bin is associated to a notional processor. These notional processors are next allocated to the physical processor. Two approaches are possible. The first is called *flat* mapping. In this approach notional processors are allowed to use only two processors at most. Their allocations are performed by filling each physical processor at a time. If a notional processor cannot be completely allocated to a processor, the exceeding part is allocated to the next processor. The second approach is called the *semi–partitioned* mapping. If the number of processors is m, then the m first notional processors are each assigned to a processor.

© 2017 Romain GRATIA

The remaining notional processors are allocated in the remaining capacity of the processors potentially resulting in migrations between several processors. During the execution of the system, each reserve is executed periodically as specified by its timeslot length. Then tasks in servers are executed following the Earliest Deadline First (EDF) scheduling policy. It is optimal in particular configurations that also entails more preemptions and migrations [29].

One algorithm based on the restricted migration approach is EDF–RRJM [24]. A heuristic job partitioning is used to determine on which processors each task execute. Migrating tasks are assigned a subset of processors on which they can execute. Each job of a migrating task is executed on a single processor among this subset of processors following a *Round Robin job migration* policy. This policy consists in executing the jobs of a migrating task on each processor of their subsets in a cyclical way. Experiments show that this policy entails fewer preemptions than other algorithms based on the job partitioning approach, while being as competitive as these latter algorithms [24].

Global scheduling algorithms

Many global scheduling algorithms have been proposed.

One of the first optimal scheduling algorithms for multi–processors proposed was Pfair algorithm [41]. This algorithm schedules the tasks by dividing the time into quanta of same size. Inside each quanta, all tasks are executed proportionately to their utilisations. But it has been shown that this algorithm incurs high overheads, in particular through the generation of many preemptions. Besides it requires the synchronization of all processors at each quanta. A variant of Pfair called DP–Fair, for *Deadline–Partitioning Fair*, that partly remedies to these downsides has been proposed. Tasks are fairly executed between two consecutive deadlines of any two tasks to schedule.

But several optimal scheduling algorithms entail fewer preemptions.

Among them is RUN [42]. This algorithm works with servers. Authors define servers as *virtual tasks* that are composed of a rate, a set of client tasks or other servers and a set of activation times and deadlines. These servers are also provided with a scheduling policy that is used to schedule the set of clients for a duration proportional to the rate of the server. RUN works in two steps. The first step is performed offline and consists in the derivation of a hierarchy of servers called the *Reduction tree*. Its derivation is based on two principles. The first principle is to decompose the multi–processor scheduling problem into several uni–processor scheduling problems. This is achieved with the use of a first kind of servers, referred to as primal servers. These servers are used to group tasks and form uni–processor scheduling problems. The second principle is based on the observation

© 2017 Romain GRATIA

that a primal server may not fully use a processor to execute its clients. This happens when a server has a rate strictly lower than 1. In this case, the processor would have time intervals of inactivity called idle time. The idle time of a primal server is modeled through the use of a second kind of server called *dual server*. Such servers have each a single client: the primal server from whom they have been derived. By grouping these dual servers and scheduling them in other primal servers, the number of processors required to schedule all the uni–processor scheduling problems can be reduced. The Reduction tree is formed by alternatively creating primal and dual servers until a single primal server is formed. The second step of the algorithm RUN is performed online and consists in taking the scheduling decisions. These decisions are made based on the Reduction tree and on the uni–processor scheduling policy used in the different kind of servers. For primal servers, these decisions are taken in accordance with the Earliest Deadline First (EDF) scheduling policy. For the dual servers, when it is itself executed, then its corresponding primal server is not executed. Scheduling decisions are taken at each releasing of a job and at each depletion of the budget of a server, primal or dual. This scheduling algorithm is optimal for periodic tasks with implicit deadlines and has been demonstrated to entail the fewest number of preemptions and migrations among global scheduling algorithms.

Another optimal scheduling algorithm called U-EDF [43], for Unfair EDF, is also known to entail few preemptions and migrations. Note that this algorithm can also schedule sporadic tasks with implicit deadlines. This algorithm also works in two steps. The first step consists in reserving an execution time budget on each processor for each task through a global process. This allocation is based on the "horizontal" generalisation of the EDF scheduler. In this generalisation, tasks are still considered in the order of increasing deadlines but tasks with the highest priorities are not each assigned a processor as in Global–EDF, the usual generalisation of EDF to multi–processor. Instead, this generalisation consists in allocating execution time budget to as many tasks as possible on a same processor. When a processor cannot completely execute a task, then its remaining execution is performed on another processor that is in turn allocated budgets for as many tasks as possible. This allocation is also done for tasks that do not have active jobs, to ensure that their future jobs will have enough execution time budgets. This allocation of budgets takes place each time a new job is released. Once this allocation of tasks to processor is done, the scheduling is performed on each processor using a modified version of EDF called EDF with Delays (EDF–D). Between two jobs releasing, EDF–D executes the tasks on a processor in the order of increasing deadlines that are not already executed on other processors and that have available budget on the processor. Each time a task is executing on a processor its budget on the processor is decreased by as much as it has executed.

© 2017 Romain GRATIA

Among the fixed–priority global scheduling algorithm fpEDF [44] has been proved to have an optimal utilisation bound when considering only fixed–priority scheduling algorithm with limit equal to $\frac{m+1}{2}$. It basically assigns the highest priority to tasks with an utilisation strictly greater than 0.5 and schedules the other tasks using Earliest Deadline First.

2.3.4 Mixed–criticality scheduling algorithms for the discarding tasks

We now present multi–processor scheduling algorithms that fully take advantage of the mixed–criticality discarding task model. They are all standard real–time scheduling algorithms modified or composed to handle the mixed–criticality task model. These modifications aim at allowing the correct and efficient use of the mixed–criticality task model. This is what is called adaptation of a real–time scheduling algorithm. For interested readers, a thorough review of the research results on mixed–criticality systems is available in [45]. We classify these algorithms depending on whether they handle only two criticality levels, also called dual–criticality systems, or more than two criticality levels. And then as previously, we distinguish partitioned, semi–partitioned and global scheduling algorithms.

Algorithms for dual–criticality systems

Partitioned mixed–criticality scheduling algorithms. In [46], authors propose an extension of the zero-slack rate-monotonic scheduling approach (ZSRM) [47]. It consists in computing offline zero slack instants. These are instants after which a HI task cannot complete its execution with its HI budget before its deadline if LO tasks are not stopped. This approach can be used in support of a uniprocessor scheduling policy such as EDF or RM. In the extension to multi–processors, they notably perform the task allocation first using usual heuristics such as Best Fit and Worst Fit. Then they use ZSRM to reduce the number of processors required.

The uniprocessor mixed–criticality scheduling policy EDF–VD has also been used in partitioned way [48]. EDF–VD, *EDF with Virtual Deadlines*, being the best uniprocessor mixed–criticality scheduling algorithm [49] in terms of speed–up factor. In this algorithm, virtual deadlines are computed for HI tasks. They are computed so that the execution of HI tasks in LO mode is performed sufficiently soon enough before their real deadlines. The idea is that if a mode change occurs they are able to respect their real deadlines. It is achieved by computing a factor that is used to diminished the deadline in LO mode of HI tasks. This increases of the utilisation of the HI task in LO mode. But this increase is performed such that the utilisations of the task set after the increase stay lower or equal

© 2017 Romain GRATIA

to 1. The partitioned version of this algorithm has a speed–up factor of $\frac{8m-4}{3m}$, with m the number of processors. Several heuristics for the allocation of tasks to processors were tested in [33]. One conclusion of this paper is that allocating HI task in the order of decreasing utilisation using the Worst Fit heuristic, and LO tasks in the order of decreasing utilisation with the First Fit heuristic gives the best results.

Yet, partitioned algorithms do not present the best theoretical schedulability performances, notably compared to global scheduling algorithms.

Semi–partitioned mixed–criticality scheduling algorithms. In [50], authors propose a semi–partitioned approach. Mode changes are performed on a processors basis, that is the mode change is only performed on the processor on which a TFE occurred. LO tasks executed on the processor on which the TFE occurred are migrated to another processors. The schedulability analysis is based on the Adaptive Mixed Criticality (AMC) approach. AMC is based on a response time analysis that verifies the schedulability in LO mode, HI mode and during a mode change [51]. Mechanisms are also proposed to switch back to the LO mode. But being based on partitioned approach, allowing migrations only to enforce a mode change, the theoretical schedulability performances can also be assumed to be no better than those of a global scheduling algorithm.

Authors in [52] propose an extension of NPS–F [40] a semi-partitioned scheduling algorithms to mixed–criticality systems. Their approach consists in partitioning HI tasks over the available processors. The remaining capacity on all processors is then used to execute LO tasks. The complete description of this algorithm remains to be done.

Global mixed–criticality scheduling algorithms. The principle of Virtual Deadlines presented in the previous section about non mixed–criticality schedulers has been used in two global mixed–criticality scheduling algorithms.

First, with the scheduling algorithm fpEDF to form fpEDF–VD [53]. It uses the same principle than for EDF–VD and simply replaces the schedulability test of EDF by those of fpEDF. This scheduling algorithm has a speed–up factor of $\sqrt{5}+1$. Hence, this algorithm has not the best schedulability performances.

The second algorithm in which virtual deadlines are used is MC–DP–Fair[34], the current best scheduling algorithm in terms of schedulability performances. Authors first adapt the fluid scheduling concept to the mixed–criticality systems to yield MC–Fluid. It basically computes rates so that HI tasks execute sufficiently in LO mode. It takes into account that if a mode change occurs then they are able to complete theirs execution in HI mode. Another fluid algorithm, DP–Fair, is also adapted to mixed–criticality systems with

© 2017 Romain GRATIA

the additional requirements of the computation of Virtual Deadlines. These algorithms have been proved to have a speed–up factor of $\frac{4}{3}$ [54]. This speed–up value has been proved to be the optimal value of the speed–up factor for mixed–criticality scheduling algorithms [49]. Later methods to more easily compute the rates of each task have been proposed [55; 54]. But as all fluid algorithms, MC–fluid is hardly implementable and entail many preemptions and migrations. Although, MC–DP–Fair can be implemented but it is known to entail many preemptions and migrations.

Algorithms generalised to more than two criticality levels

In this section, we only consider scheduling algorithms for tasks using the discarding task model.

The first mixed–criticality scheduling framework was presented in [56], with the particularity of being able to handle five criticality levels. It is a hierarchical framework with five containers, one for each criticality level. Each container has a particular scheduling policy. For the most critical tasks, a scheduling table is used on each processor. For the second most critical ones, partitioned EDF is used. Tasks of the third and fourth levels are scheduled with global EDF. The last one using a best effort scheduling algorithm. On each processor, tasks are scheduled in decreasing order of criticality levels and then within each criticality level with one of the aforementioned scheduling policy. Although this algorithm is implementable [32], it requires that the periods of the tasks of the second criticality level are harmonic to those of the first level. Hence, it can only be used in specific cases.

2.3.5 Mixed–criticality scheduling algorithms for elastic task model

We present here multi–processor scheduling algorithm for elastic tasks. They are also adaptation of real–time scheduling algorithms.

In [57], authors propose a semi–partitioned scheduling algorithm for elastic tasks. It consists in performing the partitioning of the LO and HI tasks on the processors by considering the HI mode timing parameters of the tasks. Then the scheduling online is performed by using early–release Earliest Deadline First [58]. In this scheduling policy tasks are scheduled following the EDF policy. But LO tasks are scheduled with LO mode periods if HI tasks do not completely use their HI mode budgets and hence leave unused execution time budget. The determination of whether the unused execution time budget is sufficient to execute more often a LO task is performed online at so called "early released points". The unused execution time budget considered is either those available on the same processor than the LO task. Or the LO task current job can migrate to another processor if

© 2017 Romain GRATIA

the unused execution time budget is available on another processor. The issue with this approach is that no guarantees can be given before the execution of the system that HI tasks will leave enough processor capacity to execute LO tasks in LO mode.

In [59], authors propose a partitioned scheduling algorithm for elastic task. Two allocation table of tasks on processors are defined, one for each mode. These tables are defined so that each HI task keeps executing on its processor whatever the mode is. Each LO task may have a different processor assigned to it, for each mode. In case of mode change, a LO task may have to migrate from processors. They also propose an approach to minimise the number of migrations once a first draft of allocation tables has been computed. Online tasks are scheduled with fixed priorities assigned using the response time analysis when performing the partitioning.

We presented two scheduling algorithms for the elastic task model. Both are partitioned or semi-partitioned (fixed-priority) scheduling algorithms. To our knowledge, there is no global scheduling algorithm that can schedule elastic task. It would be of interest to have such algorithm since global scheduling algorithms are known to have better theoretical performances than partitioned ones.

2.4 Conclusion

This chapter presented the context, and related works to the problematic studied in this thesis.

A need for more elaborated task models raised from the automotive industry context. This chapter detailed this need. Then, the challenges represented by possibles approaches to treat this need have been explained. In particular, the link between WCET computation, safety and sizing has been highlighted. Such a link leads engineers to see safety as contradictory with cost reductions.

In this context, it appeared that finding compromises in design process between safety guarantees and sizing issues would be a key result. Such a problem is not new but technologies have changed and the tension between safety and sizing constraints increased. For this reason, studies on this compromise received lately a lot of attention, and lead mixed–criticality scheduling theory.

Background on real time scheduling has been recalled. Then, we detailed most relevant contributions with respect to mixed–criticality approaches. Indeed, such results aim at lowering the tension resulting from the contradictory objectives of safety engineers, and system designers (usually in charge of the sizing).

© 2017 Romain GRATIA

It appeared that many contributions are available in the field of mixed–criticality scheduling for multi-processors. Yet, such contributions seem to have either poor schedulability performances, or provide poor results with respect to schedulability overheads. Next chapter details our problematic and objectives with respect to this issue.

© 2017 Romain GRATIA

3 Problem Statement

Table of Contents

The focus on mixed–criticality systems has led to the design of many scheduling algorithms. For multi–processor platforms, all types of scheduling algorithms have been proposed, from the hierarchical scheduling framework MC^2[56], to the partitioned scheduling algorithm based on EDF–VD, or global scheduling algorithms, such as fp–EDF–VD and MC–fluid [34]. Yet, as pointed out in the section 2.3.4 p 30, each of these algorithms presents shortcomings when it comes to efficiency or applicability. Hence, we aim at designing a mixed–criticality scheduling algorithm which is efficient and practicable. As for all current mixed–criticality scheduling algorithms, its design is also achieved by adapting an existing real–time scheduling algorithm. But to make sure that the resulting scheduling algorithm presents the targeted characteristics, the design of our mixed–criticality scheduling algorithm follows a step-by-step approach. The first three steps aim at avoiding that the resulting mixed–criticality scheduling algorithm exhibits the same defects of current mixed–criticality scheduling algorithms. The following three next steps are the requirements that any mixed–criticality scheduling algorithm has to fulfil to correctly and efficiently schedule a mixed–criticality system. The last step considers the use of another degradation model of task execution than the discarding task model.

3.1 Conditions for a well set up adaptation

There is currently no mixed–criticality scheduling algorithm adapted from a real–time scheduling algorithm both efficient and practicable. Indeed, all current adaptations have led to mixed–criticality scheduling algorithms with defects, such as a high number of preemptions or the scheduling of systems with only two criticality levels, among others. Therefore, the adaptation has to be carried out by always bearing in mind all the characteristics we want the resulting mixed–criticality algorithm to have. It starts by carefully choosing the initial real–time scheduling algorithm to adapt. The second step consists in carefully assessing the cost of each strategy of adaptation for the chosen scheduling algorithm. Finally, the need to handle more than two criticality levels is not to be forgotten.

3.1.1 Inheritance of qualities and defects of the initial algorithm

The design of a mixed–criticality scheduling algorithm starts by the choice of a real–time scheduling algorithm. The choice among real–time multi–processor scheduling algorithms is sizeable but all are not worth to be adapted.

To guide the choice of the original real–time scheduling algorithm, the targeted characteristics have to be taken into account. Indeed, if we examine current mixed–criticality scheduling algorithms, the best global mixed–criticality scheduling algorithm is MC–fluid [34], when considering only theoretical and experimental schedulability performances. MC–fluid has been shown to be far better than fp–EDF–VD the second best global mixed–criticality scheduling algorithm. When the comparison is also performed between the initial real–time scheduling algorithms, the same conclusion can be drawn. MC–fluid is based on the fluid real–time scheduling algorithm, which is known to be optimal, and is also far better than fp–EDF used by fp–EDF–VD. For uni–processors and partitioned scheduling algorithms, the best mixed–criticality scheduling algorithm is based on EDF, and is called EDF–VD. EDF is on the whole the best real–time scheduling algorithm for uniprocessor, both in terms of schedulability performances, as it is optimal, and in terms of number of entailed preemptions [60]. Hence, there seems to have a relation between the performances of the real–time scheduling algorithm and the performances of the resulting mixed–criticality scheduling algorithm.

Therefore, it can be assumed that not only schedulability performances are impacted by the initial real–time scheduling algorithm performances. In the two previous examples, fp–EDF–VD and MC–fluid, the computation of the timing parameters used to take the scheduling decision is changed but not the event used to trigger a call to the scheduler.

© 2017 Romain GRATIA

Hence, the number of calls to the scheduler remains unchanged. Accordingly, the resulting number of preemptions and migrations should remain unchanged between the initial real–time scheduling algorithm used as a basis and the corresponding mixed–criticality scheduling algorithms. As a result, fluid algorithm and its derivatives being known to produce many preemptions, it is highly likely that the adaptations of these algorithms to mixed–criticality systems also entail many preemptions. This is illustrated later in this thesis.

Finally, the feasibility of the implementation of the scheduling algorithm should also be taken into account. Indeed, some algorithms are based on concept that can hardly be put into practice. For example, fluid algorithm is not implementable on usual processors as it supposes to provide a share of execution to all tasks at each instant. This also affects MC–fluid making it therefore highly difficult to implement, that is why derivatives of fluid algorithms based on a degraded fluid hypothesis have been proposed. But again these derivatives can be affected by other defects such as Pfair [31] that requires the synchronisation of all the processors. Thus the re–scheduling on all processors is performed at the time and can result in memory contention due to the loading of all data of the tasks to schedule in caches [61] at the same time.

Consequently, the choice of the algorithm should be done by selecting the real–time scheduling algorithm that presents the targeted characteristics. Once the real–time scheduling algorithm is chosen, the next step consists in determining which strategy of adaptation to use.

3.1.2 Complexity to perform the adaptation

The adaptation aims at altering real–time scheduling algorithms so that they can handle the peculiarities of the mixed–criticality system model. Mixed–criticality scheduling algorithms have to handle different sets of timing parameters while real–time scheduling algorithms have been designed to handle a single one. In particular, real–time scheduling algorithms have not been designed to switch online from a given set of timing parameters to another one, and in particular to let tasks to potentially execute with a larger budget. To proceed to this adaptation two strategies can be distinguished from the current mixed–criticality scheduling algorithms. But depending on the initial real–time scheduling algorithm, a strategy can require more modifications than the other. Besides, the more modifications are carried out, the more the real–time scheduling algorithm is changed and the more its characteristics are likely to be altered. Hence, the choice of the strategy to per-

© 2017 Romain GRATIA

form this adaptation should be made with the aim of limiting the modifications performed on the real–time scheduling algorithm.

In real–time systems, tasks have only a single value for their timing parameters. As described in section 2.3.1 p 16, in mixed–criticality systems, a task can have several values for a given timing parameter and its execution can be stopped. Their proper scheduling requires the respect of new constraints, that are described later in section 3.2 p 40, that have to be respected by the adaptation of the real–time scheduling algorithm. Current adaptations of real–time scheduling algorithms have been done using, to our knowledge, one of the following strategies:

- Strategy based on timing parameters: with this strategy either the computation of the timing parameters used to take scheduling decisions is adapted to the peculiarities of mixed–criticality systems, or new timing parameters are used in the scheduling policy. This strategy generally means that when a mode change occurs, tasks change of timing parameters. It presents the advantage to leave unaltered the scheduling overheads of the scheduling policy since the scheduling decision process remain unaltered.

- Hierarchisation of the scheduling algorithm: one or several scheduling levels are added, with possibly a different scheduling algorithm at each level. This strategy enables the use as is of the real–time scheduling algorithms of the different levels of the hierarchy. Besides, it generally avoids changing timing parameters when a mode change occurs. But each added scheduling level increases the scheduling overheads.

Each strategy has cases where it requires fewer modifications to perform the adaptation than in other cases. For instance, the strategy based on timing parameters allows to adapt EDF without deeply modifying it. Indeed, it simply requires to compute proper so–called *virtual deadlines* for each task to be used in LO mode leaving other aspects of EDF untouched. But, it might not be the case for other scheduling algorithm, as for example RUN [42]. Indeed, RUN bases its online scheduling decisions on a complex structure computed offline from the timing parameters of the task set to schedule. Using the strategy based on timing parameters with RUN does not only require to correctly change timing parameters but also the structure. Thus, in the case of RUN, using this strategy, we might not have to consider only the need of each task but also how the structure is computed and how it is used. Indeed, the functioning of this structure might even require to be changed to fit with the mixed–criticality model. Consequently, adapting RUN with this strategy may require a lot of modifications, affecting potentially critical aspects and hence its characteristics.

© 2017 Romain GRATIA

Therefore, the adaptation has to address the following problem:

Problem of the complexity of the adaptation:

the adaptation of the real–time scheduling algorithm should result in as few modifications as possible.

Independently of the chosen algorithm and of the strategy used to adapt it, all mixed–criticality scheduling algorithms are first designed using a restricted model of the mixed–criticality systems. Indeed, most of them consider mixed–criticality systems with only two criticality levels, called dual–criticality systems. This enables us to simplify the problem of designing such algorithm as the resolution of the mixed–criticality schedulability problem has been shown to be NP–hard [62] even with only two criticality levels.

3.1.3 Dual–criticality model: a restrictive model

In mixed–criticality systems, criticality levels are often linked to Safety Integrity Levels (SILs) whose number depends on the industry sector. Although the relation between SILs and criticality level is not necessarily correct, as argued in [17], it means that mixed–criticality scheduling algorithms should handle up to five criticality levels. Yet, even if we consider this link as erroneous, criticality levels can be seen as a design parameter that engineers can use to better fulfil their requirements. Indeed, if a mixed–criticality system can always be transformed in a dual–criticality system, as proposed in [63], results in loss of information on task requirements without bringing any clear advantage [64]. Besides, handling more criticality levels leaves engineers more scope to finely classify their tasks in different criticality levels and thus add more executions modes. The use of more execution modes enables to potentially delay the discarding of tasks whose criticality level is not the lowest nor the highest. It could hence improve the availability of tasks with intermediate criticality levels. Finally, by designing each task at an adapted criticality level it limits design costs. Indeed, the higher the criticality level, the higher the confidence in the design is required, and hence the larger the costs are [65].

But a majority of mixed–criticality scheduling algorithms have been designed to schedule dual–criticality systems, that is mixed–criticality systems with two criticality levels. The resulting algorithm should then be extended to handle more criticality levels. But it is rarely done and it is not clear whether it can be done at all and, then, done without degrading the performances observed for only two criticality levels. The limitation to only

© 2017 Romain GRATIA

two criticality levels is aimed at simplifying the scheduling problem when designing the scheduling algorithm. Indeed, it has been proven that even with only two criticality levels, the resolution of the mixed–criticality schedulability problem is a NP–hard problem [62].

However, if a scheduling algorithm designed for dual–criticality systems, may not work for system with more criticality levels since it is a different problem. Finding a solution for dual–criticality systems does not imply that this solution also works for systems more criticality levels. Consequently, the following problem is to be addressed:

Problem of the generalization to N criticality levels:
a real–time scheduling algorithm adapted to dual–criticality systems may not be extended to schedule systems with more than two criticality levels.

The three following steps describe the requirements to fulfill so that a mixed–criticality scheduling algorithm is correct and efficient.

3.2 Requirements for a correct and efficient adaptation

The task model in mixed–criticality systems rests upon the multiplicity of timing parameters for each task and the definition of several execution modes. This enables to adapt the task timing parameters to the observed execution behaviour of the system. These online changes of timing parameters alter the scheduling sequences and can occur at the exhaustion of execution budget of any task. Hence, a mixed–criticality scheduling algorithm has to fulfil three requirements. The first is to ensure that each mode can be correctly scheduled. Secondly, we have to ensure that mode changes are correctly performed whenever they happen. Finally, the adaptation has to ensure the correctness of the scheduling the most efficiently as possible.

3.2.1 Schedulability per execution mode

In mixed–criticality systems, there are several execution modes defined and, in each of these modes, tasks have different timing parameters. But real–time scheduling algorithms are not designed to handle such a diversity in their task set model that results in having several different task sets to execute. For a mixed–criticality task set to be schedulable, it first requires that all execution modes are schedulable by the scheduling algorithm.

© 2017 Romain GRATIA

Task name	Period / Deadline	Criticality level	C(LO)	C(HI)
τ_1	5	HI	2	4.5
τ_2	3	LO	1	1
τ_3	10	HI	3	6.5

Table 3.1: Example of a simple MC task set

Mixed–criticality scheduling algorithms have to be able to allow tasks to use different WCETs. It means that it has to be able to handle the different processing power requirements of each execution mode. For example, the dual–criticality task set in table 3.1 if scheduled by fp–EDF requires 2 processors in mode LO, but in mode HI it requires 3 processors. Hence, fp–EDF would require at least 3 processors to schedule both modes, because of the need of the mode HI, otherwise the mixed–criticality system would not be schedulable. If only two processors are provided, the mixed–criticality system is not schedulable as the mode HI is not schedulable.

The scheduling of a mixed–criticality system requires first to solve the following problem:

Problem of the scheduling of a multi–mode task set:

all execution modes of mixed–criticality system have to be schedulable by the chosen real–time scheduling algorithm.

This condition is necessary but not sufficient. Indeed, when a Timing Failure Event occurs, a mode change is performed but for it to be correct safe guard mechanisms have to be provided in order to avoid a deadline miss.

3.2.2 Disruption of the mode change

Mode changes must be done without making the system unschedulable. In particular, higher criticality tasks must still meet their deadlines after such event.

However, checking that each execution mode passes the schedulability test associated with the scheduling algorithm does not ensure that a mode change will be properly handled. Indeed, consider the simple dual–criticality task set in table 3.1. If we schedule tasks τ_1 and τ_2 using the real–time scheduling algorithm Earliest Deadline First (EDF), we can easily check that both modes LO and HI pass the associated schedulability test:

© 2017 Romain GRATIA

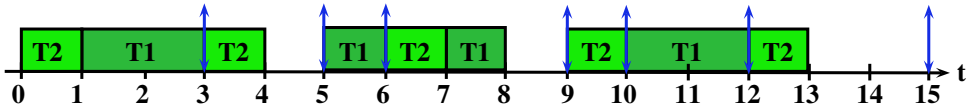


Figure 3.1: Task set from table 3.1 executing in mode LO

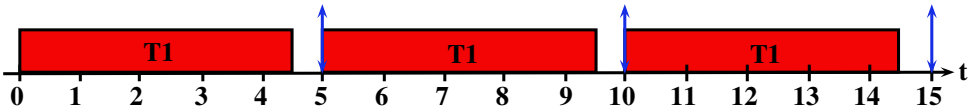


Figure 3.2: Task set from table 3.1 executing in mode HI

$$U_{\Gamma}(LO) = \frac{11}{15} \leq 1 \text{ and } U_{\Gamma(HI)}(HI) = 0.9 \leq 1$$

The resulting scheduling sequences are shown in figures 3.1 and 3.2 for modes LO and HI respectively.

Now, assume a Timing Failure Event (TFE) occurs at time t = 3. We can observe in the scheduling sequence of figure 3.3, that the HI task τ_1 misses its deadline. It does not have sufficient time between its deadline and the time of occurrence of the TFE to complete its execution using the budget of the HI mode.

A mode change results in the modification of the task set to schedule and of the task timing parameters. In particular, it enables higher criticality tasks to execute for longer. But this additional execution time must be consumed within the remaining time that separates a task from its deadline, otherwise, a deadline miss follows. The issues are that a TFE can happen after the exhaustion of the execution budget of any task and that passing the schedulability test associated to the real–time scheduling algorithm in each mode is not sufficient. Consequently, the correctness of mode changes has to be ensured by providing mechanisms. The aim of these mechanisms is to ensure that higher criticality tasks can execute safely with a larger execution time budget without missing their deadlines.

The scheduling of a mixed–criticality system requires to solve the following problem:

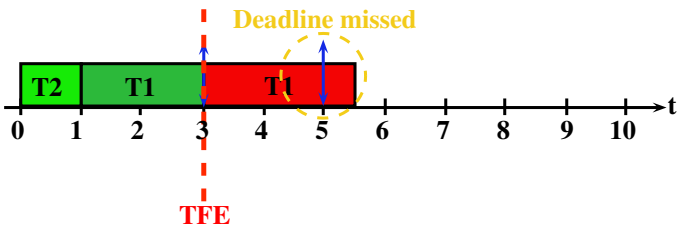


Figure 3.3: Example of a failed mode change: a deadline is missed by the HI task

© 2017 Romain GRATIA

Problem of the disruption created by a mode change:

The scheduling of a mode cannot be made independently of the other modes.

In any case, ensuring correct mode change constrains the adaptation of a real–time scheduling algorithm affecting the efficiency of the mixed–criticality scheduling algorithm.

3.2.3 Mitigation of the negative impact of the adaptation

Mixed–criticality scheduling theory emerged to allow a more efficient use of the execution platforms. But the adaptation of a real–time scheduling algorithm has to ensure the correct scheduling of each mode and the correct handling of mode changes to ensure a correct scheduling of the mixed–criticality system. However, fulfilling these requirements can have a negative impact on the efficiency of the resulting mixed–criticality scheduling algorithm.

Mixed–criticality scheduling theory aims at reconciling safety and efficiency. Safety has to deal with the problems of the scheduling of several modes and with the handling of mode changes. These problems require safe guarding mechanisms that affect the performances of the mixed–criticality algorithms. Indeed, it has been proved that the minimum speed–up factor of any mixed–criticality scheduling algorithm is $\frac{4}{3}$ [49], while real–time scheduling algorithms are optimal, that is they have a speed–up factor of 1. This is because unlike a clairvoyant scheduling algorithm the mode in which the system will execute cannot be known before the execution.

The objective is to execute the system on the maximal number of processors required by one of the execution modes while ensuring a correct scheduling. It is at least expected that any mixed–criticality system can be scheduled on no more processors than the number required if the system was executed as a multi–criticality system. Current mixed–criticality scheduling algorithms fulfil this objective with one of the two following strategies:

1. The use of idle time. Each mode may not completely use the available processors. Hence, there are time intervals during which processors are idle, we refer to these time intervals as idle time. A strategy toward efficiency is to distribute the idle time of each mode to the different tasks. MC–fluid employs this strategy by inflating the rates of the HI tasks both in LO and HI modes using the available idle time of those modes.

© 2017 Romain GRATIA

2. The use of the slack time of tasks. HI task budgets are different in mode LO and HI. The difference of budget between mode is called slack time. This slack time can be used to execute LO tasks. It is employed in the scheduling policy presented in [47] and in MC^2.

The distribution of the idle time between tasks presents the advantage of being available to all tasks and under no conditions. On the contrary, the slack time of the higher criticality tasks is only available when the higher criticality task has completed its execution. But using the slack time of higher criticality tasks presents the advantage of enabling the sharing of executed time budget between tasks. It enables the scheduling of tasks without taking into account their execution requirements when sizing the processing power needed to execute the system. Either way the schedulability performances of mixed–criticality scheduling algorithm depends on the efficient use of available unused time.

Therefore, besides the correctness issue, we also have to address the following problem:

Problem of the efficiency of the mixed–criticality scheduling:

Ensuring the correctness of the scheduling of mixed–criticality system leads to the alteration of the performances.

Mixed–criticality system theory has emerged as a solution to industrial problems. As stated in the introduction, this thesis has been carried out within an industrial project. Our industrial partners provided feedback on the assumptions of mixed–criticality system theory, and in particular, on the discarding task model.

3.3 Discarding task model: a limited degradation model

Industrial partners have expressed the need for another model of degradation model of task execution enabling a better availability of lower criticality tasks after a mode change. Indeed, most mixed–criticality scheduling algorithms use as degradation model of task execution for their LO tasks what we call the discarding degradation model of task execution. In this degradation model of task execution, LO tasks are completely stopped after a mode change and are hence unable to provide any level of service. Another degradation model of task execution allowing lower criticality tasks to provide a minimal service

© 2017 Romain GRATIA

level is the *elastic task model*. But scheduling elastic tasks and discarding tasks may each require specific adaptation of a real–time scheduling algorithm.

The elastic task model consists in lengthening the periodicity and deadlines of the LO tasks. Thus, we still have to ensure that LO tasks are provided with a minimal execution time budget even after a mode change occurred. Therefore, the way the adaptation is done for discarding tasks may not handle this change in periods.

Indeed, here the challenges for the adaptation is not to deal only with a change in WCETs but also in periods. Therefore, the problem to solve is not to only provide more execution time to HI tasks and **nothing** to LO tasks after a mode change occurred. But it is to provide more execution time to HI tasks and execute LO tasks with **less stringent** timing requirements. This distinction in objectives influences how the adaptation is carried out.

This can be observed when comparing Early–Release Earliest Deadline First (ER–EDF [58]) and EDF–VD. ER–EDF has been designed to schedule elastic task while EDF–VD schedules discarding tasks. Both algorithms manage to schedule dual–criticality systems by modifying timing parameters. But ER–EDF computes particular release times for LO tasks while EDF–VD computes particular deadlines for HI tasks. Hence it seems that from a same initial real–time scheduling algorithm, the adaptation to schedule discarding tasks or elastic tasks can be not carried out in the same way.

One might consider the discarding degradation model of task execution as a particular case of the elastic task model, as it would correspond to a task with a periodicity approaching infinity. However, the extension from the discarding model to the elastic task model is less obvious because of the remaining execution in HI mode with the elastic task model. Therefore, designing a dual–criticality scheduling algorithms first for discarding tasks and then use this scheduling algorithm to also schedule elastic tasks seems highly difficult.

Problem of the limitation of the discarding task model:

Designing a mixed–criticality scheduling algorithm for tasks using the discarding degradation model of task execution, makes the use of another degradation model of task execution highly difficult.

© 2017 Romain GRATIA

3.4 Conclusion

In this chapter, we described the issues to overcome in order to design a mixed–criticality scheduling algorithm that does not present the same defects than current mixed–criticality scheduling algorithms.

We thus started by describing the precautions to take to avoid the defects of the current mixed–criticality scheduling algorithms. First, when we choose the real–time scheduling algorithm to adapt. Second, when we choose the strategy to perform the adaptation. And finally, it requires to anticipate the scheduling of systems with more than two criticality levels.

We have then presented the requirements to respect when designing a mixed–criticality scheduling algorithms. In particular, we described the issues raised by mode changes. We also exposed the negative effects on the efficiency of scheduling algorithms of the mechanisms required to handle mode changes.

We finally pointed out the need expressed by industrial partners for another degradation model of task execution than the discarding task model. The elastic task model offers a degradation model that can suit their needs. Yet, scheduling elastic tasks with a mixed–criticality scheduling algorithm designed for the discarding task model is not straightforward.

In the next, chapter we briefly present our solutions to tackle these issues.

© 2017 Romain GRATIA

4 Approach overview

Table of Contents

The previous chapter has detailed the issues to cope with when designing a mixed–criticality scheduling algorithm. In this chapter, we give an overview of our solutions to address these problems and to design our mixed–criticality scheduling algorithm.

Our approach to design a mixed–criticality scheduling algorithm for multi–processor platforms consists in splitting the multi–processor mixed–criticality scheduling problem into two distinct and simpler scheduling problems. This decomposition is illustrated in figure 4.1. To achieve this decomposition, we partition the task set and execute these partitions into servers. These servers, called **modal servers**, enforce a mixed–criticality scheduling policy and deal with the peculiarities of the mixed–criticality systems. This makes them schedulable by a multi–processor real–time scheduling algorithm. The use of these servers results in a hierarchical scheduling framework in which a *top level* scheduler, the multi–processor real–time scheduling algorithm, determines the modal servers to execute. Then schedulers in modal servers, designated as *low level* schedulers, elect tasks to execute. Thanks to the use of this hierarchical framework, we only have to design a mixed–criticality scheduling policy for the low level scheduler.

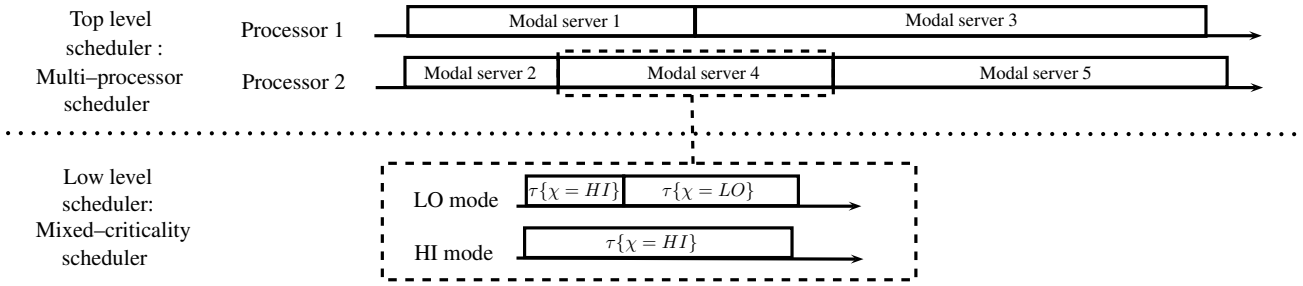


Figure 4.1: Representation of the hierarchical scheduling framework

Our mixed–criticality scheduling policy takes advantage of the slack time provided by HI tasks in LO mode. This slack time is used to either complete the execution of HI tasks, if a mode change is performed, or to schedule LO tasks in LO mode. We prove prior to the execution that the slack time provided by HI tasks is sufficient to ensure the correct execution of LO tasks as long as the LO mode remains active. Hence, a same piece of execution budget is either used to schedule LO or HI tasks, instead of using two distinct budgets to schedule LO and HI tasks. This sharing of budgets reduces the number of processors required to schedule the system. The potential LO tasks not executed in slack time of HI tasks are scheduled in modal servers with dedicated execution resources.

Performances of the resulting hierarchical scheduling framework, such as the ratio of schedulable task sets and the number of preemptions entailed, depend on the performances of the scheduling policies used in the hierarchy. First, the offline determination of which

© 2017 Romain GRATIA

LO tasks can be executed in HI task slack time has a key impact on the schedulability performances of our algorithm. Indeed, the larger the utilisation of LO tasks scheduled in slack time is, the fewer processors are required. Secondly, we choose the multi–processor scheduling algorithm based on performance criteria. These criteria aim at ensuring that our mixed–criticality scheduling algorithm has the targeted characteristics. Indeed, in section 3.1.1 p 36, we observed that the current mixed–criticality scheduling algorithms inherited the same defects than their initial real–time scheduling algorithms.

This approach only schedules dual–criticality systems, we then describe our approach to schedule system with more criticality levels. Mixed–criticality peculiarities are handled in modal servers the system appears as a non mixed–criticality one to the multi–processor scheduling algorithm. Indeed, HI tasks are scheduled in modal servers and LO tasks are either scheduled in HI task slack time or in other modal servers. Hence, the use of the modal servers somehow reduces by one the number of criticality levels. Therefore, to handle more criticality levels, we design an inductive process that performs the partitioning of the tasks by considering that the system comprises only two criticality levels during each step. This process iterates from the lowest criticality levels to the highest ones and works for any number of criticality level.

Finally, we generalise our approach to also schedule LO elastic tasks. It requires to adapt the elastic task model before the partitioning of tasks in modal servers. This adaptation consists in determining for each LO elastic task the minimal execution time to provide in HI mode and the additional execution time required in LO mode. It is achieved by decomposing the elastic task execution into two subtasks. The first subtask is always scheduled and ensures the execution of the LO elastic task in HI mode. The second subtask is executed only in LO mode and provides the additional execution time required in LO mode.

We start by giving an overview on how the composition of the two scheduling policies is carried out.

4.1 Decomposition of the mixed–criticality multi–processor scheduling problem

To design our mixed–criticality scheduling algorithm, we decompose the scheduling of the mixed–criticality system on multi–processor platform into two scheduling problems. We use servers to perform this decomposition of the scheduling problem as shown in figure 4.1. In the lower part of this figure, servers are used to enforce a mixed–criticality

© 2017 Romain GRATIA

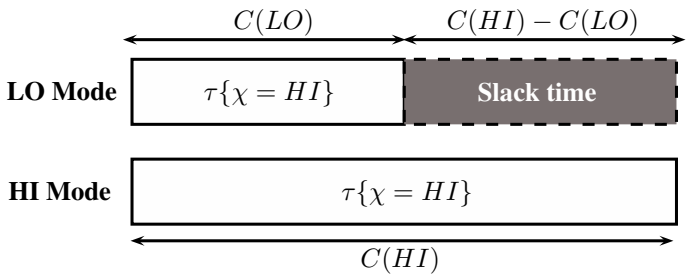


Figure 4.2: Origin of the slack time

scheduling policy on a subset of mixed–criticality tasks schedulable on a uniprocessor. This mixed–criticality scheduling policy takes advantage of the slack time of HI tasks to either schedule LO tasks or complete HI tasks depending on the active mode. That is why, these servers are called **modal servers**. These servers are then scheduled by a multi–processor real–time scheduling algorithm as pictured in the upper part of the figure 4.1. These scheduling algorithms form a hierarchical scheduling framework with a top level and a low level schedulers. Where the multi–processor scheduling algorithm is the top level scheduler and our mixed–criticality scheduling policy is the low level one.

Mixed–criticality systems are defined with two execution modes, LO and HI modes, each with their own timing parameters and task sets to schedule. If HI tasks were executed using their budgets of HI mode even in LO mode, then a part of this budget would be unused as long as the active mode is the LO mode. This unused part of the budget is referred to as *slack time*. This slack time can be estimated prior execution, as pictured in figure 4.2, and is equal to the difference of budgets between HI and LO modes for each HI task. It can be used to either schedule LO tasks or to complete HI tasks.

We need to design a mixed–criticality scheduling policy, that ensures the correct scheduling of LO and HI tasks in the slack time. It notably has to ensure that HI tasks can complete their executions after a Timing Failure Event (TFE) occurred. Our scheduling policy first completes HI task executions, so that we have the assurance that slack time is not needed by HI tasks. Then LO tasks are then safely executed in the slack time left by the HI tasks. Since, the scheduling performed in the slack time of a HI task is akin to the scheduling on a uniprocessor, only one LO task at a time can be scheduled. Thus, a uniprocessor scheduling policy is used to select the LO task to execute.

A way to enforce a particular scheduling policy on a limited set of tasks is to use servers. We call them modal servers because of their particular mixed–criticality scheduling policies. As modal servers are used to schedule tasks in the slack time of HI task, each modal server schedules a HI task. The set of LO tasks is determined offline by performing a partitioning of all LO tasks in modal servers.

© 2017 Romain GRATIA

Servers provide limited and constant amount of execution time available during only limited time interval corresponding to the slack time of HI tasks. Furthermore, available slack time can be large but split among many HI tasks. Therefore, each HI task can only provide a very limited amount of slack time potentially preventing the scheduling of any LO task in their slack time. To limit this scattering of the slack time, we can use the slack time of several HI tasks to execute a same subset of LO tasks. The modal servers scheduling these HI tasks form then what we call **aggregated modal servers**. Aggregated or not, modal servers cannot correctly schedule all tasks with their resources due to the limitation of the available slack time. Hence, we need schedulability tests to ensure that tasks executed within this limited resource are correctly scheduled. This schedulability test is performed during the partitioning of the LO tasks in the modal servers. If a LO task passes the schedulability test, it is said to be allocated to the modal server.

This requires to have a resource model for these servers, a model that fits with the slack time availability. Modal servers handle the mixed–criticality peculiarities through the use of a mixed–criticality scheduling policy. This policy does not require changing timing parameters of the modal servers between modes. Hence, this resource model can be a model used for hard real–time tasks. By using such model it makes modal servers compatible with the models used by multi–processor real–time scheduling algorithms. Hence, an adapted resource model is one that mimics the periodic task model behaviour: the periodic resource model [66]. In this model, the resource can be accessed for an amount of time fixed by a budget that is replenished periodically. Besides, it is a simple and widespread resource model compatible with most real–time scheduling algorithms giving us a large choice for the multi–processor scheduling algorithm.

4.2 Performances of the composition

The hierarchical scheduling framework is composed of two different scheduling policies. Its performances are dependent on the performances of each scheduling algorithm. As a consequence, performances of each scheduling algorithm has to be in line with our objectives in terms of number of preemptions and schedulability performances, in particular concerning the ratio of task sets successfully scheduled.

First, each scheduling algorithm of the hierarchy has an impact on the number of preemptions entailed by the whole scheduling framework. The number of preemptions entailed by the hierarchical scheduling algorithm corresponds to at most the sum of the preemptions entailed by its two scheduling algorithms. Indeed, if a preemption occurs at the top level, then it results at the low level in a preemption that was not going to happen

© 2017 Romain GRATIA

if the scheduling algorithm of the low level was alone. Inversely, low level scheduling algorithm can entail preemptions that top level scheduling algorithm would not have.

Besides, the overall schedulability test consists in performing the schedulability test of each level. A mixed–criticality system is schedulable by the hierarchical scheduling framework if two conditions are fulfilled. First, if subsets scheduled in modal servers successfully pass the schedulability test associated to modal server scheduling policy. The second condition for the system to be schedulable is that the schedulability test associated to the multi–processor scheduling algorithm is successfully passed by the modal servers. Hence, if the two scheduling algorithms have pessimistic schedulability tests or a low utilisation bound then the resulting scheduling algorithm will present poor schedulability performances.

Therefore, the choice of the scheduling algorithm, for the multi–processor scheduling or uniprocessor one in modal servers, must take into account these aspects, as we anticipated in section 3.1.1 p 36 of the problem statement. Indeed, we then observed that the current mixed–criticality scheduling algorithms inherited the defects of the real–time scheduling algorithms used for their designs. These considerations lead us to choose RUN [42] as multi–processor scheduling algorithm and Earliest Deadline First (EDF) to schedule LO tasks in HI task slack time. These two scheduling algorithms entail few preemptions [60; 42] and are both optimal scheduling algorithms for multi–processor and uniprocessor platforms respectively.

The scheduling performances are also affected by the partitioning of the LO tasks in modal servers. During this partitioning, we determine which LO tasks is scheduled in HI task slack time by allocating LO tasks to modal servers. As LO tasks executed in HI task slack time do not need their own execution time budgets, the more LO tasks are executed in HI task slack time, the fewer processors are required to schedule the whole system. We have to solve an optimisation problem, with the aim of maximising the utilisation of LO tasks scheduled in HI task slack time.

4.3 Mixed–criticality systems with any number of criticality levels

As exposed in section 3.1.3 p 39 in the problem statement, most existing mixed–criticality scheduling algorithms are actually dual–criticality scheduling algorithms, that is scheduling algorithms for systems with only two criticality levels. The demonstration that these dual–criticality scheduling algorithms can also schedule mixed–criticality systems with

© 2017 Romain GRATIA

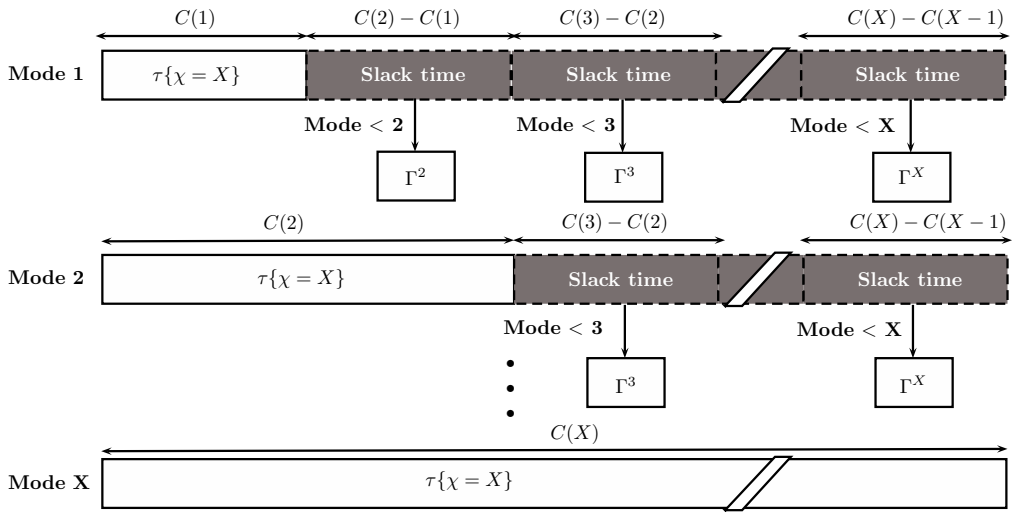


Figure 4.3: Available slack time in each mode of a CL–X task

more than two criticality levels is, in most cases, not given nor straightforward. In our case, it is achieved by performing the allocation of lower criticality task in modal servers with an inductive process.

First, notations have to be extended to the case of systems with more than two criticality levels. When speaking of systems with more than two criticality levels, we use a notation based on numbers. A task of criticality level X is noted a CL–X task. For example, a task of criticality level 2 is noted a CL–2 task. The greater X the more critical the task is. Identically, we note mode X the mode in which all tasks of criticality levels greater or equal to X are executed with their budgets corresponding to criticality level X.

If executed with its budget of criticality level X, a CL–X task would not completely use its budget in modes below mode X as depicted in figure 4.3. As its budget is not completely used, slack time is available. And the lower the active mode is, the more slack time is available. Hence, in a similar way to dual–criticality systems, the slack time corresponding to each mode for tasks of criticality level greater than 2 can be used to schedule tasks with the help of modal servers. The difference here is that a task of criticality level greater than 2 can still have slack time after the several Timing Failure Events (TFE). After one TFE, only the difference of budgets between the old active mode and the new one is required to complete the execution of the higher criticality task. For instance in the figure 4.3, when changing from mode 2 to mode 3 the slack time use to complete the CL–X task amounts to $C(3) - C(2)$. We then say that this slack time is of mode 3, as it is used to complete the higher criticality task when mode 3 becomes active. Consequently, for a CL–X task its slack time can be used to schedule X-1 sets of lower criticality tasks Γ^l, with $l \in [2, X]$, one for each mode from mode 2 to mode X.

© 2017 Romain GRATIA

Each set Γ^l is scheduled in the slack time that is equal to the difference of budgets between two modes. In figure 4.3, the task set Γ^3 is scheduled in the slack time of mode 3. This slack time of mode 3 can be used in modes 1 and 2 to execute lower criticality tasks but is needed in mode 3 to complete the CL–X task. A modal server has now to handle several sets of tasks to take advantage of the slack time of a task.

To determine the criticality level of the tasks that can be scheduled in each of these of sets of tasks, we examine when the slack time is required to complete the higher criticality task. A set of lower criticality tasks must not be scheduled if the CL–X needs to complete its execution. This happens when a TFE occurs and a higher mode becomes active. Hence, each of these sets have to contain tasks that no longer need to be executed when the CL–X task needs to complete its execution. In the case of the slack time needed by the CL–X task when mode 3 becomes active, only tasks with a criticality level strictly lower than 3 can be scheduled in it. This ensures that when the CL–X task needs its slack time to complete its execution, it can use it without preventing another task from being executed while it should be. To always respect this condition when performing the allocation of lower criticality tasks in modal servers, we develop an inductive process.

In order to ensure that each slack time of mode M+1 has only tasks with a criticality level strictly lower than M+1, we form two groups of tasks based on criticality levels, as pictured in figure 4.4. First group contains tasks with criticality level strictly lower to M+1 and are represented in the lower part of the figure. The second one contains tasks of all other criticality levels from M+1 to X, the highest one, represented in the upper part of the figure. Once these groups are formed, we partition tasks of criticality level lower than M+1 into the sets of modal servers. This partitioning aims at executing tasks of lower criticality levels in slack time of mode M+1 of tasks of criticality level at least M+1. In figure 4.4, lower criticality tasks are allocated in slack time of mode M+1 equal to $C(M+1) - C(M)$. After this partitioning, lower criticality tasks are in task sets of modal servers corresponding to slack time of mode M+1.

All criticality levels are then processed by repeating this grouping and partitioning with the remaining criticality levels. Each step results in the reduction of the number of criticality levels by one thanks to the allocation, until reaching the highest criticality level. The resulting task set exhibits no mixed–criticality characteristics as for dual–criticality systems.

The scheduling in modal servers follows the same principle than for dual–criticality system. A modal server executes first its task of highest criticality level and then schedules tasks of lower criticality level in the slack time left.

© 2017 Romain GRATIA

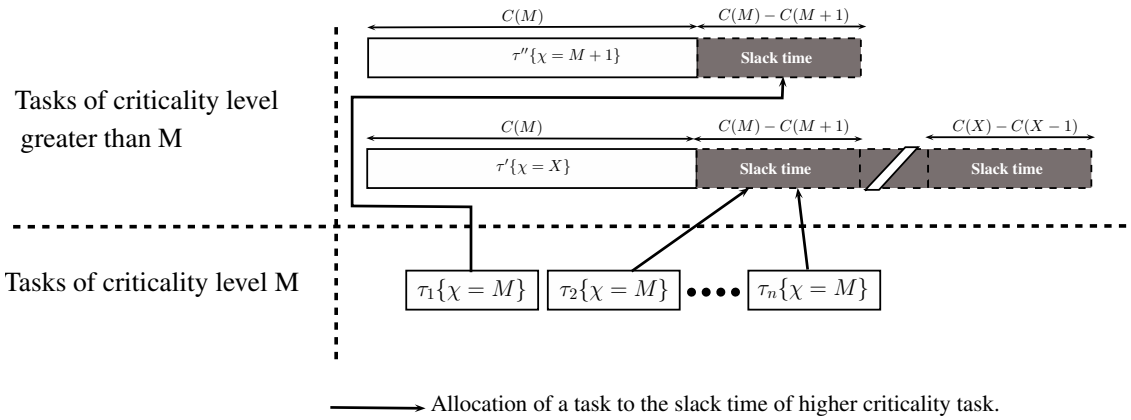


Figure 4.4: Allocation of CL–L tasks into slack time of tasks of higher criticality level

4.4 Elastic task model

When a TFE occurs, LO task execution is impacted. The impact on their execution depends on the degradation model of task execution. The degradation model of task execution used in most mixed–criticality scheduling algorithm is what we call the **discarding model** presented in section 2.3.1 p 20 of the related work. In this model, LO tasks are completely stopped after a mode change. But, as explained in section 3.3 p 44 of the problem statement, such a model may be unfit for industrial needs. Indeed, industrials want that some tasks that are not HI tasks remain executed in HI mode but with a lower rate of execution. Hence, we consider the use of another task model with a different degradation model of task execution, called the elastic task model. With this model, LO tasks are not stopped altogether but their execution is performed with less stringent timing requirements after a mode change. LO tasks are executed with larger period and deadline but with a same execution time budget as pictured in figure 4.5. In this figure, a LO task is executed C time units with a period T(LO) in LO mode. When HI mode becomes active this same task is still executed C time units but only with a period T(HI) = 2·T(LO). LO elastic tasks execution is slowed down in HI mode but not completely stopped. Hence, the use of modal servers based on our previous approach requires some adaptation as model server budget is only available in LO mode.

In HI mode, a LO elastic task receives its minimal execution requirements. In LO mode, it receives its nominal execution requirements. A mode change results in lower execution requirements for LO elastic tasks. To match with our previous proposal, we compute the minimal execution requirements in HI mode and the additional execution requirements compared to HI mode needed in LO mode. This additional execution requirements, only required in LO mode, can then be provided by modal servers.

© 2017 Romain GRATIA

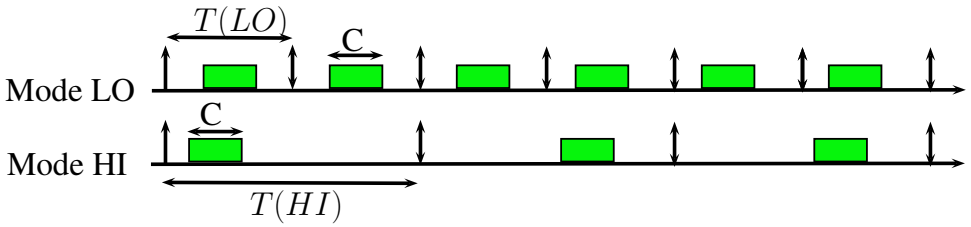


Figure 4.5: Execution of an elastic task in different modes

This requires to alter the elastic task model. We do not execute a LO elastic task in HI mode with a lower period and with a same budget than in LO mode. We execute it with a same period in both modes but with a lower budget in HI mode. We transform a change in periods into a change in budgets.

By doing this, a LO elastic task τ can be seen as the combination of two subtasks, as pictured in figure 4.6. These two subtasks are servers that are used to execute the LO elastic task and each has the following scheduling objective:

1. Subtask τ^{NMC} provides the execution time required in HI mode as pictured in figure 4.6. It ensures that the LO elastic task receives its budget C every period of the HI mode T(HI). It is executed in both modes and hence cannot be executed in a modal server. It is noted τ^{NMC}, with NMC for non mixed–criticality since it is always executed.

2. Subtask τ^{DIS} provides the additional execution time to subtask τ^{NMC} so that LO elastic task execution requirements are met in LO mode. As it is executed only in LO mode, this subtask can be executed in a modal server. It is noted τ^{DIS}, with DIS for discarding, as it is discarded after a mode change.

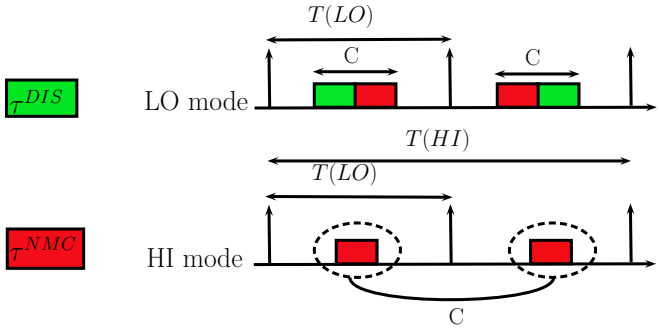


Figure 4.6: Decomposition of an elastic task execution in the different modes

We need to determine the timing parameters of the two subtasks. We need to ensure that subtasks provide the required execution time when the LO elastic task needs it. Hence,

© 2017 Romain GRATIA

we set the period of the subtask τ^{NMC} to the greater common multiple of the periods of LO and HI modes of the LO elastic task. The period of the τ^{DIS} is equal to the period in LO mode of the LO elastic task. We compute first the additional budget required in LO mode to schedule the LO elastic task in LO mode. It corresponds to the budget of the subtask τ^{DIS} and it is computed such that the subtask can be executed in the slack time of a set of modal servers. Then we compute the required budget to correctly execute the LO elastic task in HI mode. It corresponds to the budget of the subtask τ^{NMC}. This budget is computed such that the execution of both τ^{NMC} and τ^{DIS} ensures the scheduling of the LO elastic task as pictured in figure 4.6. To ensure that a LO elastic task complete execution is assured by τ^{NMC} and τ^{DIS} in LO mode, τ^{NMC} and τ^{DIS} are executed sequentially.

Finally, subtasks τ^{DIS} are partitioned into modal servers as LO discarding tasks.

4.5 Conclusion

In this chapter, we presented our solutions to tackle the different issues we identified in the previous chapter when designing a mixed–criticality scheduling algorithm.

Our approach to design our mixed–criticality scheduling algorithm for multi–processor platforms is to decompose the scheduling problem. We first design a mixed–criticality scheduling policy for uniprocessor. This mixed–criticality scheduling policy uses the slack time of HI task to either schedules LO tasks or to complete HI task execution depending on the active mode. It requires to partition the task set into several uniprocessor problems. Then a multi–processor scheduling policy is used to schedule the different uniprocessor problems. This is achieved by executing each uniprocessor scheduling problem into servers called **modal servers**. The use of these two policies form a hierarchical scheduling policy and yield our mixed–criticality scheduling algorithm for multi–processor platforms. The details of our hierarchical scheduling framework and of our mixed–criticality scheduling policy are given in chapter 5.

The performance of our mixed–criticality scheduling policy depends on two aspects. First, the choice of the multi–processor scheduling algorithm. Then, the effectiveness of the use of the slack time of HI tasks. This depends on how efficiently we perform the partitioning of the tasks in modal servers. This is an optimisation problem. We address the performance issues in chapter 6.

We also generalise our mixed–criticality scheduling algorithm to system with more than two criticality levels. This is done by performing an inductive process that considers each criticality level from the lowest criticality level to the highest one. It basically splits the task set in two groups based on criticality levels: one group contains tasks of the lowest

© 2017 Romain GRATIA

criticality levels, the other all other tasks. Tasks in the first group are scheduled in the slack time of tasks in the second group. This inductive process is presented in chapter 7.

Finally, we explain how we extend our approach to handle a different model of the degradation of the task execution, the elastic task model. It is achieved by performing the execution of LO elastic tasks in two different subtasks providing the minimal execution requirements of HI mode and the additional execution requirements of LO mode. We use slack time of a HI task to provide additional execution requirements in LO mode. This is addressed in chapter 8.

© 2017 Romain GRATIA

5 The structure of the mixed–criticality scheduling policy

Table of Contents

For the design of our mixed–criticality scheduling algorithm for multi–processor platforms, we proposed in the previous chapter to decompose the scheduling problem into two distinct problems. The mixed–criticality scheduling problem is handled as uniprocessor problems in modal servers that are scheduled by a multi–processor real–time scheduling algorithm. The justification for this approach is twofold.

First, it eases the conception of the overall scheduling algorithm. With such framework we still have to design a mixed–criticality scheduling policy for uniprocessor but we can use an existing multi–processor real–time scheduling algorithm. Designing a uniprocessor scheduling policy is easier than designing a multi–processor one [31].

Secondly, we choose to base our mixed–criticality scheduling policy on the use of the HI task slack time to schedule LO tasks. The use of the slack time aims at reducing the number of processors required, as LO tasks scheduled in HI task slack time do not require their own execution resources. But it requires to reserve the slack time for these LO tasks and prevent other tasks from executing in it. Furthermore, we have to prove prior to the execution that scheduling of LO tasks is ensured. This can only be achieved if we know when and how much slack time is available, that is what are the available execution resources to execute these LO tasks. The usual technique to purposely provide a determined and constant amount of execution resources to a subset of tasks following a particular scheduling policy is to use execution servers. These execution servers require to be scheduled by a second scheduling policy forming a hierarchical scheduling framework.

Yet, several issues remain to be addressed. We have to design a mixed–criticality scheduling policy. Indeed, we need a mixed–criticality scheduling policy that enables the scheduling of tasks in the slack time of other tasks while respecting their hard real–time constraints.

The correctness of the scheduling produced by this hierarchical scheduling framework is to be ensured. This notably requires specific schedulability tests for the scheduling of LO tasks in HI task slack time that takes into account the limitation of the execution resources provided.

In this chapter, we address the scheduling problem that concerns the mixed–criticality aspect of the scheduling but only with two criticality levels. The second scheduling problem concerns the scheduling on multi–processors, and is addressed in chapter 6. The generalisation to mixed–criticality systems with more than two criticality levels is addressed in chapter 7.

In this chapter, we lay the foundations of our mixed–criticality scheduling algorithm. We first explain and justify why the hierarchical scheduling framework gives the expected mixed–criticality scheduling policy for multi–processor and on which conditions

© 2017 Romain GRATIA

this scheduling is correct. Then we introduce **modal servers**. After providing a general definition of modal servers, we describe the three ways to use these modal servers to schedule a mixed–criticality system. First, modal servers enforce our mixed–criticality scheduling policy that schedules LO tasks in the slack time of a single HI task, we name such modal servers **slackful modal server**. We also provide schedulability tests that ensures that a LO task is schedulable in the slack time of a HI task. Second, modal servers are used to schedule LO tasks that cannot be scheduled in HI task slack time, we refer to these modal servers as **slackless modal server**. Lastly, slack time can be scattered among several HI tasks and this can prevent from scheduling any LO task in slack time while the overall available slack time is large. Therefore, we merge several modal servers to form what we call **aggregated modal servers** enabling the scheduling of LO tasks in the slack time of several HI tasks.

5.1 Laying the foundation of our mixed–criticality scheduling policy for multi–processors

In this section, we detail the formation of our mixed–criticality scheduling policy for multi–processor and the conditions to fulfil to schedule a system with such a hierarchical scheduling framework. Then, we determine how much slack time is available and present the issue of its use.

5.1.1 Hierarchical scheduling framework

The decomposition of the mixed–criticality scheduling problem on multi–processor platforms into two problems, one for the mixed–criticality scheduling on uniprocessor, the other for multi–processor one, aims at easing the design of our scheduling algorithm. Each scheduling problem is addressed by using a dedicated scheduling policy. Then the combination of these two scheduling policies form what is called a **hierarchical scheduling framework**.

Hierarchical scheduling frameworks have been used to ease the conception of systems with applications developed independently. These applications come with their own scheduling policies to schedule their own task sets, designated as **low level scheduling algorithm**. A **top level scheduling algorithm** is used to schedule these applications. When an application is executed it enforces its low level scheduling policy to elect the tasks to execute.

© 2017 Romain GRATIA

A key element to design such scheduling framework is the *execution server*, that is defined as follows:

Definition 12. *An execution server is a sequence of operations aiming at scheduling tasks. It is characterized by a resource model, a task set and a scheduling policy. An execution server uses the execution resources described by its resource model to execute its task set following its scheduling policy.*

Low level schedulers in servers can be any scheduling policy even a policy handling a different model of tasks.

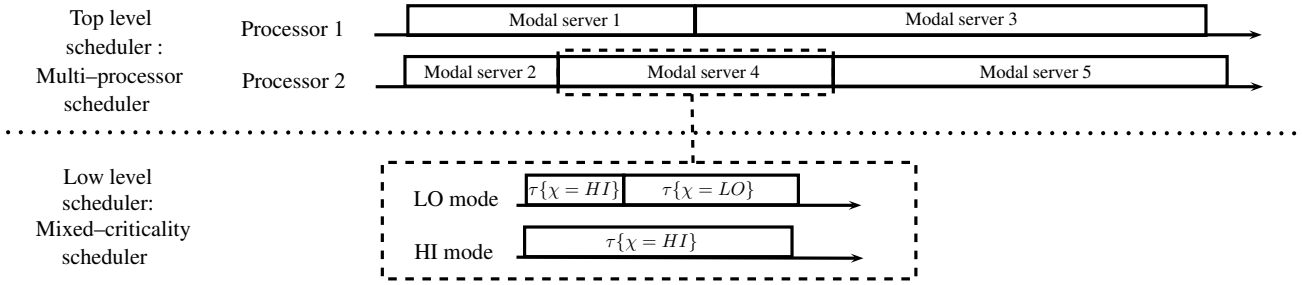


Figure 5.1: Representation of our hierarchical scheduling framework for mixed–criticality scheduling on multi–processor platforms

For instance servers are used to schedule aperiodic tasks with periodic tasks. Consequently, we can use servers to schedule mixed–criticality tasks and let a multi–processor real–time scheduling algorithm schedule these servers as represented in figure 5.1. This requires that the resource model of the execution server fits with the task model of real–time scheduling algorithm and in particular unchanging execution requirements. Then, scheduling an execution servers is identical to schedule a task.

Such hierarchical scheduling framework can also be used to compose several different scheduling policies to yield a specific scheduling policy. An example is the Reduction to UNiprocessor scheduling algorithm [42]. In RUN, two different scheduling policies are used in a hierarchical structure whose overall functioning gives an optimal scheduling algorithm for multi–processor platforms. Hence, by composing our two scheduling policies for multi–processor and mixed–criticality systems in a hierarchical framework, we obtain our mixed–criticality scheduling algorithm for multi–processor platforms by organising it as pictured in figure 5.1. The mixed–criticality scheduling policy is enforced at low level of the hierarchical framework and the multi–processor at its top level.

The schedulability of the whole system is ensured if the schedulability in each server is ensured and if the schedulability at the top level scheduling algorithm is ensured, that is

© 2017 Romain GRATIA

if all execution servers are correctly scheduled. A correctly scheduled execution server is defined as follows:

Definition 13. *An execution server is correctly scheduled when it receives execution resources as described by its resource model.*

The following theorem gives the conditions for a task set to be schedulable in an execution server:

Theorem 1. *Let S be an execution server with a resource model,* Γ *its task set and* $\mathcal{A}$ *its scheduling policy. If server S is correctly scheduled and* Γ *fulfils the schedulability test associated to scheduling policy* $\mathcal{A}$ *to be scheduled in server S then task set* Γ *is schedulable by server S.*

Proof. We prove this theorem by contradiction. Consider that there is a server S with a task set Γ scheduled following a scheduling policy $\mathcal{A}$. This server S is correctly scheduled and Γ fulfils the schedulability test associated to $\mathcal{A}$. Yet, a task in Γ still misses its deadline. Two cases: either schedulability conditions to be scheduled in server are met but server S does not provide expected execution time to the task set Γ or server S provides expected execution but schedulability conditions are not met. The first case contradicts the first assumption of the theorem stating that the server is correctly scheduled. The second case opposes the second assumption of the theorem that assumes task set Γ passes a schedulability test and hence server S provides sufficient execution resource to schedule it. □

The correctness of the scheduling of the task sets of servers depends on the sufficiency of the execution resources received by execution servers and provided by them to their task sets. The next step is to determine the resource model of the execution servers enforcing our mixed–criticality scheduling policy that we call modal servers. This resource model has to enable the allocation of the slack time.

5.1.2 Allocation of the slack time

Before introducing modal servers we determine the resource model that best suits to the slack time and the precautions required when allocating LO tasks to this slack time.

A mixed–criticality system is expected to execute in LO mode, HI tasks executed with their budgets of LO mode. They fully use their budgets of HI mode only after a mode change occurred, which is considered to be a very rare event. Now, let us assume we execute HI tasks with their budgets of HI mode in both LO and HI modes. Then HI tasks generate what we call *slack time* defined as follows:

© 2017 Romain GRATIA

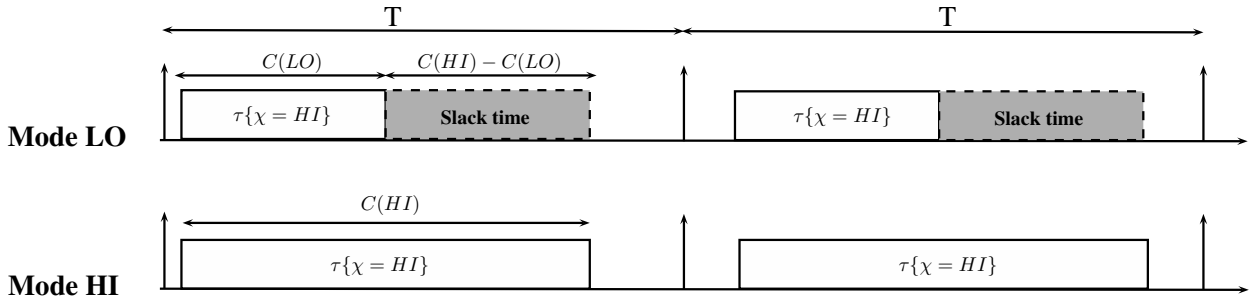


Figure 5.2: Origin of the slack time

Definition 14. *Slack time is idle time that originates from unused part of a task budget.*

For a HI task, this slack time amounts to $C(HI) - C(LO)$ as represented in figure 5.2. It is provided each time a HI task is executed as long as LO mode is active. Thus, the slack time of each HI task is provided every period of that task in LO mode, meaning that slack time is a periodic resource [66].

Task	Period	Criticality	C(LO)	C(HI)	U(LO)	U(HI)
τ_1	5	HI	1	3	0.2	0.6
τ_2	2	HI	0.5	1.5	0.25	0.75
τ_3	8	HI	0.8	3.2	0.1	0.4
τ_4	8	LO	3.2	3.2	0.4	0.4
τ_5	15	LO	3.75	3.75	0.25	0.25
τ_6	12	LO	2.4	2.4	0.2	0.2
τ_7	12	LO	2.4	2.4	0.2	0.2

Table 5.1: Example of MC task set with two criticality levels

For example, consider a typical mixed–criticality task set with two criticality levels presented in table 5.1, we shall use it to illustrate issues or to detail our solutions for the scheduling of systems with two criticality levels in the following. Task τ_1 slack time amounts to 2 time units and is replenished every $T_1 = 5$ time units and corresponds to a utilisation equal to 0.4.

Besides, we define the availability of the slack time as follows:

Definition 15 (Slack time availability)**.** *Slack time of task τ_i is said available when it is not used to execute task τ_i.*

Now that we have determined the resource model of the slack time, we can determine how to use it by taking into account two constraints.

First, the execution requirements of HI tasks: slack time must always be available to complete HI tasks execution.

© 2017 Romain GRATIA

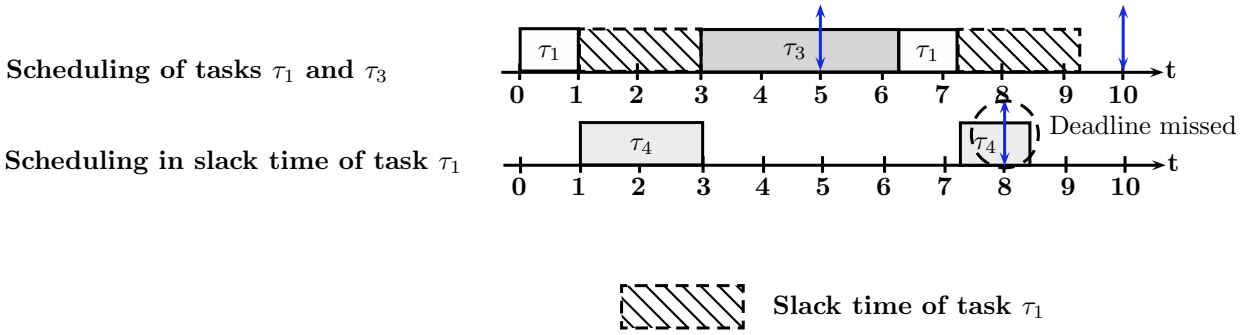


Figure 5.3: Availability of the slack time: unsuccessful execution

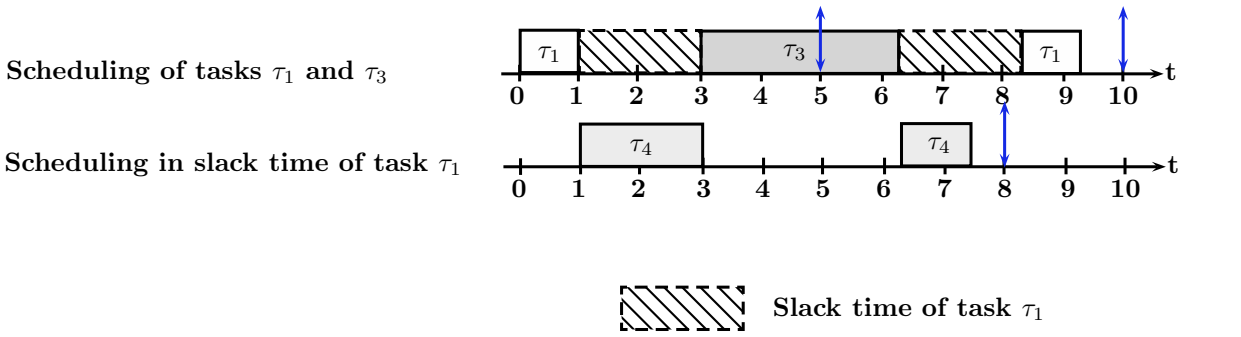


Figure 5.4: Availability of the slack time: successful execution

Second, the limited amount of slack time available and its period of replenishment. It requires to check that there is enough slack time to schedule a LO task and that this slack time is available when the LO task needs it. Therefore, ensuring that LO tasks are correctly scheduled in slack time requires the use of proper schedulability tests, that are presented in section 5.3.3 p 70.

To illustrate this issue, consider the figure 5.3. Task τ_4 from table 5.1 is executed within τ_1 slack time. But as can be seen in this figure, if the slack time of task τ_1 is made available only once τ_1 has completed its execution, then τ_4 misses its deadline at time 8. The only way to ensure that task τ_4 respects its deadline is to perform its execution before τ_1 has completed as described in figure 5.4. But then a potential mode change could not be correctly performed for task τ_1, as slack time has been partly used to execute the LO task while task τ_1 needs the whole slack time to complete its execution in HI mode.

In the following sections, we first introduce a general definition of modal servers and their mixed–criticality scheduling policy to schedule a set of LO and HI tasks. Then, we present how to use this modal server to schedule a single HI task and use its slack time to schedule LO tasks. After presenting schedulability tests to ensure the scheduling of LO task in HI task slack time, we present how modal servers are used to schedule LO tasks

© 2017 Romain GRATIA

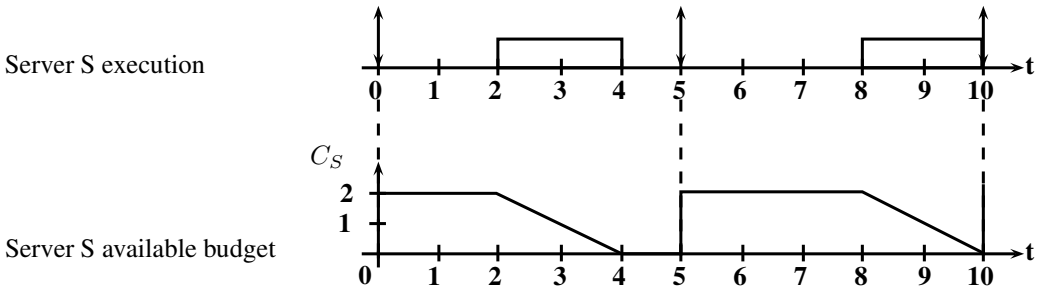


Figure 5.5: Budget replenishment of a periodic server S with budget $C_S = 2$ and period $T_S = 5$

that cannot be scheduled in any HI task slack time. Finally, we describe how to schedule a set of LO tasks in the slack time of several HI tasks with aggregated modal servers.

5.2 Modal server

Modal servers are introduced to enforce our mixed–criticality scheduling policy. In this section we present the general definition of a modal server.

Modal server are used to schedule periodic task. To fit with this model we introduce periodic servers:

Definition 16. *A periodic server S is an execution server whose resource model is described by a period T_S and a budget C_S, replenished every period T_S, and a utilisation $U_S = \frac{C_S}{T_S}$.*

Periodic servers behave as periodic tasks with implicit deadlines: they execute periodically for as long as their budgets as pictured in figure 5.5 and each of their executions must be completed before their next activation, that represents its deadline.

As for tasks, we can define the utilisation U_{Γ_S} of a set of periodic servers Γ_S such that:

$$U_{\Gamma_S} = \sum_{S_i \in \Gamma_S} U_i \tag{5.1}$$

Note that periodic servers present unchanging execution requirements, characterized by a single budget and a single period. We make explicit for such periodic severs what it means to be correctly scheduled:

Definition 17. *A periodic server S of budget C_S and period T_S is said to be correctly scheduled if it effectively executes C_S time units every T_S time units.*

© 2017 Romain GRATIA

But because of the mixed–criticality context modal server definition slightly differ from regular periodic servers or execution servers and are defined as follows:

Definition 18. *A modal server MS is a periodic server with two sets of tasks:*

- *A set of HI tasks* $\Gamma_{MS}(HI)$.
- *A set of LO tasks noted* $\Gamma_{MS}(LO)$.

These two task sets are scheduled as follows:

- *In LO mode, HI tasks in* $\Gamma_{MS}(HI)$ *are scheduled first and once they are completed, tasks in* $\Gamma_{MS}(LO)$ *are scheduled following a uniprocessor scheduling policy.*
- *In HI mode, only HI tasks in* $\Gamma_{MS}(HI)$ *are scheduled.*

In our case the uniprocessor scheduling policy used to schedule the LO tasks is Earliest Deadline First.

However, if we use this definition of the modal server, we cannot prove the correctness of the mixed–criticality scheduling policy enforce in these servers. Indeed, in LO mode, LO tasks scheduling may not be ensured since it depends of the completion of all the HI tasks first. Therefore, we use our modal servers with at most one HI task executed in each modal server. In the remainder of this chapter, we present the way of using our modal servers to produce a correct mixed–criticality scheduling.

5.3 Slackful modal server

We define what is a slackful modal server and prove the correctness of our mixed–criticality scheduling policy in this context. This modal server allows to take advantage of HI task slack time to schedule LO tasks.

5.3.1 Definition of slackful modal server

A slackful modal server is defined as follows:

Definition 19. *A slackful modal server is a modal server whose task set of HI tasks* $\Gamma(HI)$ *is constituted of a single HI task* τ_h.

Hence a slackful modal server is a particular case of modal servers. We call the HI task the **providing task**, as it provides the slack time. To represent a modal server, and a slackful modal server in particular, we use a representation as in the figure 5.6. What is

© 2017 Romain GRATIA

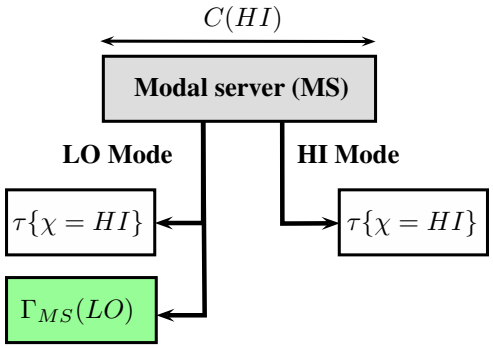


Figure 5.6: Task sets scheduled by slackful modal server

scheduled in LO mode is designated by the left arrow. The HI task $\tau(HI)$ is represented first indicating that it is executed first. Then the set of LO tasks $\Gamma_{MS}(LO)$ is below the HI task indicating it is executed after the HI task. The right arrow designate what is scheduled in HI mode: the HI task.

Before proving the correctness of the mixed–criticality scheduling for slackful modal server, we describe how we compute the timing parameters of such modal servers. First, we remind the definition of a divisor:

Definition 20. *Let n and m be two integers. n is a divisor of m, noted $n \mid m$, if there exists an integer k such that $m = nk$.*

The first objective of a slackful modal server is to ensure the scheduling of its HI task. Hence slackful modal server timing parameters should be set to schedule its HI task the most efficiently as possible.

Davis and Burns [67] show that the utilisation of an execution server required to scheduled a set of tasks is lower when the server period is a divisor of the period of one of the task. Following their results, a modal server should have a period that is a divisor of its HI task period.

Now, assume that the period set for a slackful modal server is a divisor of its HI task period. We still have to compute the budget of the slackful modal server. This budget must be sufficient to schedule the HI task in both HI and LO modes and as low as possible, in order to minimise the required processing power to schedule it. The following theorem gives a schedulability condition to schedule a task in a periodic server with a period multiple of the task period:

Theorem 2. *If a periodic task τ_i and a periodic server S are such that $T_S \mid T_i$ and $U_i \leq U_S$, then S ensures τ_i schedulability.*

Proof. It is akin to perform a period transformation [68; 16] on the HI task. □

© 2017 Romain GRATIA

This theorem can be used to compute the required budget of our slackful modal server: the budget must be such that the resulting utilisation of the slackful modal server is equal to utilisation in HI mode of its HI task. And it is the minimal value of utilisation that allows the correct execution of the HI task. This minimal utilisation value ensures the scheduling of the HI task whenever the period of the execution server is a divisor of the HI task period.

But for a same utilisation, the smaller the period of the slackful modal server is the smaller its budget. Smaller budget means that the execution of the HI task is split as several executions of the slackful modal server is required to complete the execution of a job of the HI task. But we have to take into account that each time a server is scheduled it entails the same overheads than a task, e.g context switching among others, and also the overheads originating from the use of servers [67]. Therefore, the choice of the period of the slackful modal server among all the divisors of the period of the HI task should be the period that allows to complete the execution of the HI in the smallest number of executions of the slackful modal server. The period of the slackful modal server should be the greatest divisor of the HI task period that ensures the correct scheduling of the HI task, that is the period of the HI task.

In conclusion, a slackful modal server has the same budget C and period T than its HI tasks in HI mode.

5.3.2 Proof of the correctness of the scheduling

A slackful modal server enforces a particular mixed–criticality scheduling policy on its HI task and LO tasks. Recall, that when LO mode is active, slackful modal server executes first its HI task and then schedule its sets $\Gamma_{MS}(LO)$ with Earliest Deadline First (EDF). When HI mode is active, a slackful modal server schedules only its HI tasks, as LO tasks are no longer executed.

As slackful modal server definition slightly differs from execution servers we have to adapt the theorem 1 on the schedulability of a task set in an execution servers to slackful modal server specificities:

Theorem 3. *Let $\Gamma(LO)$ be a set of LO tasks, τ_h a HI task and MS a slackful modal server. If the three following conditions are fulfilled:*

1. *slackful modal server MS is correctly scheduled .*
2. *HI task τ_h is schedulable by the slackful modal server MS in HI mode.*
3. *All tasks in $\Gamma(LO)$ are schedulable in slack time of task τ_h in LO mode.*

© 2017 Romain GRATIA

then τ_h and tasks in $\Gamma(LO)$ are schedulable by our mixed–criticality scheduling policy in slackful modal server MS in HI and LO modes.

Before proving this theorem we need to introduce the following lemma:

Lemma 1 (Criticality precedence condition). *A LO task $\tau_j \in \Gamma_{MS}(LO)$ of a slackful modal server, that schedules a HI task τ_h, is executed in interval $I = [k \cdot T_h, (k+1) \cdot T_h]$ if and only if τ_h has completed its $(k+1)^{th}$ execution without triggering a TFE in I.*

Proof. It is assured by the definition of slackful modal servers which first execute HI task and only then LO tasks. If a TFE is triggered then the system switches to the HI mode and LO tasks are not executed. □

Lemma 1 ensures that slack time of a HI task in LO mode is used to schedule LO tasks only when the HI task has completed its execution without triggering a TFE. It is now possible to prove the correctness of the theorem 3:

Proof. We prove this theorem by contradiction. Consider that there are a slackful modal server MS, a task set $\Gamma(LO)$ and a HI task τ_h such that theorem 3 assumptions are true but a task, either 1) τ_h or 2) a LO task in $\Gamma(LO)$, still misses its deadline.

1) If τ_h missed its deadline, two cases are possible. Either one or several LO tasks executed before τ_h but this is not possible because of lemma 1. Or slackful modal server did not executed long enough. Again two cases are possible. First, the slackful modal server was not correctly scheduled which contradicts our first assumption. Second, slackful modal server provides not enough execution time to HI task τ_h but this contradicts our second assumption.

2) This case either contradicts our third assumptions or HI τ_h executed for longer than its budget of LO mode. In that latter case, LO tasks are no longer executed as it would trigger a mode change. Hence the theorem holds. □

5.3.3 Schedulability tests to schedule LO tasks in HI task slack time

In this section, we provide schedulability tests to check whether LO tasks in slackful modal server $\Gamma_{MS}(LO)$ task set are schedulable in the slack time left by the HI task. To this end particular schedulability tests are required when allocating LO tasks to slackful modal server for two reasons. First, slack time provided by HI tasks is limited. Second, slackful modal server replenishment times are defined independently of the periods of the LO tasks it schedules, as illustrated in section 5.1.2 p 63. Two schedulability tests are provided and can be used for any periodic servers such as slackful modal server. The first test can

© 2017 Romain GRATIA

Task	Period	Criticality	C(LO)	C(HI)	U(LO)	U(HI)
τ_1	5	HI	1	3	0.2	0.6
τ_2	2	HI	0.5	1.5	0.25	0.75
τ_3	8	HI	0.8	3.2	0.1	0.4
τ_4	8	LO	3.2	3.2	0.4	0.4
τ_5	15	LO	3.75	3.75	0.25	0.25
τ_6	12	LO	2.4	2.4	0.2	0.2
τ_7	12	LO	2.4	2.4	0.2	0.2

Table 5.1: Example of MC task set with two criticality levels (same table that on page 64)

only be used if task periods are multiples of the server period, but it efficiently uses slack time provided. The second one is more complex and somehow pessimistic, but makes no particular assumptions.

First, we present the schedulability test when periods of LO tasks are multiple of the period of the periodic server. It extends theorem 2 by considering several tasks to be scheduled in a periodic server. Let a server S be a periodic server of period T_S and budget C_S and a set of tasks Γ to be scheduled by S.

Theorem 4. *If* $(\forall \tau_l \in \Gamma, T_S \mid T_l)$*, and* $U_\Gamma \leq U_S$*, then S ensures* Γ *schedulability.*

Yet, this condition on periods may not always be met. In this case, another schedulability test based on *Supply Bound Functions* (SBFs) [66] and the *Demand Bound Functions* (DBFs) is proposed.

A SBF returns the minimum amount of execution time supplied by a periodic server in an interval of length t and is defined as follows:

Definition 21 (Supply Bound Function [66])**.** *Let t be a duration. Let S be a periodic server with budget C_S and of period T_S. For any $t \geq 0$, the Supply Bound Function of S is defined as follows:*

$$SBF_S(t) = \lfloor \frac{t-(T_S-C_S)}{T_S} \rfloor \cdot C_S + \varepsilon(t) \tag{5.2}$$

with

$$\varepsilon(t) = max(t - 2(T_S - C_S) - T_S \cdot \lfloor \tfrac{t-(T_S-C_S)}{T_S} \rfloor, 0)$$

In our case the SBF is used to determined the minimal amount of execution provided by the slack time of HI tasks in slackful modal servers. The SBF from the slack time of HI task τ_1 in table 5.1, corresponding to a budget $C = 2$ every period $T = 5$, is pictured in figure 5.7.

The DBF function of a task or task set returns the amount of CPU time required by the task or tasks to meet its deadline from the beginning of its execution up to time t. Yet,

© 2017 Romain GRATIA

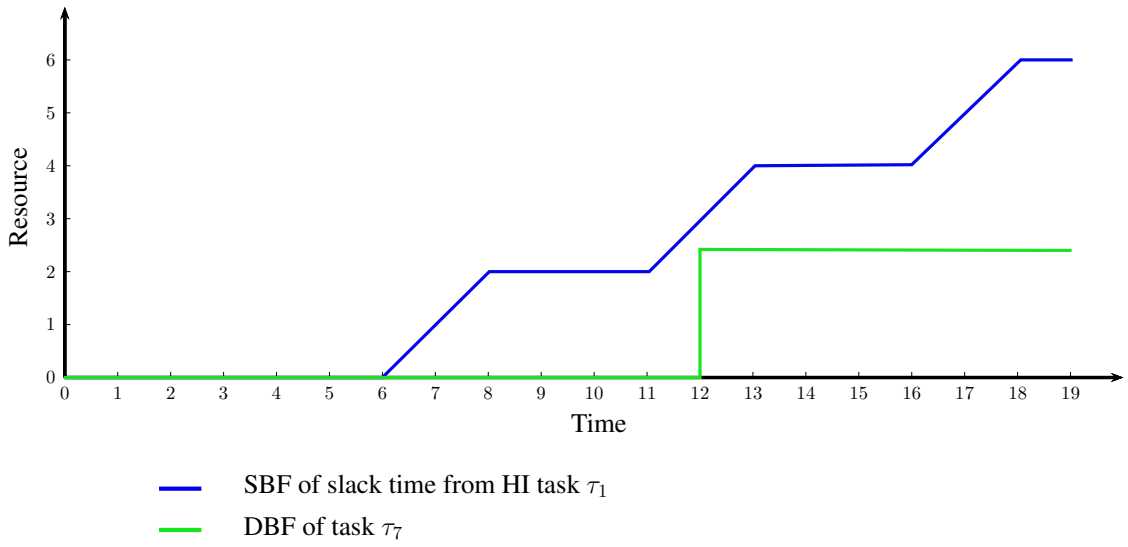


Figure 5.7: Example of SBF and DBF

the usual definition has to be slightly modified to account for parameters of the different modes.

Definition 22. *Let τ_i be a mixed–criticality implicit deadline periodic tasks of criticality level $L \in LO, HI$. For any $t \geq 0$, the Demand Bound Function of τ_i at criticality level L is defined as follows*

$$DBF_i(t,L) = \lfloor \frac{t}{T_i} \rfloor \cdot C_i(L) \tag{5.3}$$

As for utilization, the DBF of a task set is equal to the sum of DBFs of each task of the set. The DBF can be computed for each criticality level. But in our case, it is only relevant for LO tasks and hence requires only to be computed with budget of LO mode of LO tasks, therefore we omit to indicate the criticality level of the DBF and simply write DBF(t) instead of DBF(t,LO). The DBF of LO task τ_7 in table 5.1 is pictured in figure 5.7.

SBF and DBFs are then compared to check whether a slackful modal server provides enough execution time to its task set $\Gamma_{MS}(LO)$. Let $LCM(\Gamma \cup \{S\})$ be the Least Common Multiple of periods of tasks in a task set Γ and period of server S. As explained in [66], the condition for a periodic server S to ensure its task set schedulability is:

Theorem 5. *Given a set of tasks Γ if $\forall t \in [0; LCM(\Gamma \cup \{S\})]$, $DBF_\Gamma(t) \leq SBF_S(t))$, then S ensures Γ schedulability.*

This test may be seen as intractable but the number of comparisons to carry out can be reduced. Indeed, it is sufficient to only compare the DBF and the SBF at each deadline

© 2017 Romain GRATIA

of each task in Γ over the hyper–period, instead of comparing these two functions at each instant t over the hyper–period.

5.4 Slackless modal server

As said the slack time of HI task is limited. This limitation can prevent LO tasks from being scheduled in the slack time of any HI task. Still we have to ensure the scheduling of such tasks in LO mode and stop their executions in HI mode.

To this end we use modal server designated as **slackless modal server** defined as follows:

Definition 23. *A slackless modal server is a modal server MS whose set $\Gamma_{MS}(HI)$ is empty and set $\Gamma_{MS}(LO)$ contains a single LO task.*

We previously saw in section 5.3.1 p 67 that a server has the lowest utilisation to schedule its task set when it has the period of the tasks it must schedule. Therefore, to choose the timing parameters of a slackless modal server, we proceed as for the selection of the timing parameters for slackful modal server. A slackless modal server timing parameters are those of the LO task it schedules.

The scheduling policy of slackless modal server is left unchanged. In LO mode, LO tasks are scheduled as soon as the modal server is executed, as it contains no HI task. In HI mode it schedules nothing, as LO tasks are no longer executed.

5.5 Aggregated modal server

The issue is that each HI task may only provide small amount of slack time while the overall slack time of HI tasks is large. Due to this scattering of the slack time, no LO task may be schedulable in the slack time of HI tasks. To avoid that situation the idea is to gather as much as possible this slack time so that some LO tasks are scheduled in the slack time of several HI tasks. We achieve that by forming what we call **aggregated modal servers**.

5.5.1 Definition of aggregating modal servers

Aggregated modal servers are defined as follows:

© 2017 Romain GRATIA

Definition 24. *An aggregated modal server AMS is a set of slackful modal servers such that these slackful modal servers are executed sequentially and the set of LO tasks* $\Gamma_{AMS}(LO)$ *scheduled by this AMS is such that* $\Gamma_{AMS}(LO) = \cup_{MS_i \in AMS} \Gamma_{MS_i}(LO)$.

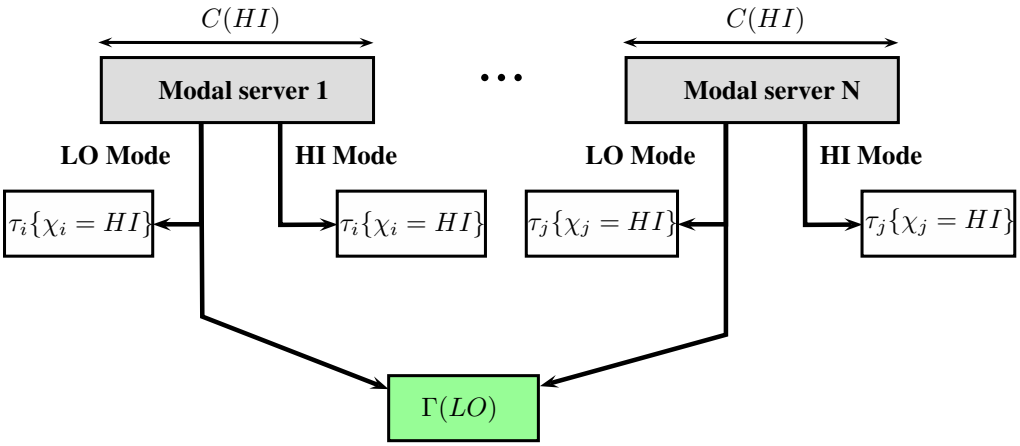


Figure 5.8: Aggregated modal servers task sets scheduling

The set scheduled in each mode by an aggregated modal server is pictured in figure 5.8. Determining the available execution time is too complex, if modal servers can execute in parallel. Executing sequentially all slackful modal servers of an aggregated modal server ensures that the available slack time from all modal servers is exactly the sum of their slack time in LO mode. The slack time of the HI tasks of these slackful modal servers can be added up and made available in their aggregated modal server.

Hence it supposes that slackful modal servers forming an aggregated modal server have to be schedulable on a uniprocessor. Therefore, the sums of the utilisations of these servers forming an aggregated modal server must be lower or equal to 1. In the remainder of this document, we refer to this condition as the **sequential condition**.

Because several slackful modal servers are used to schedule a same set of LO tasks, we have to extend schedulability tests used in the case of aggregated modal servers.

5.5.2 Schedulability tests associated to aggregated modal server

We present extensions of the schedulability tests described in section 5.3.3 p 70 to ensure that a set of LO tasks $\Gamma(LO)$ can be properly scheduled by an aggregated modal server AMS.

We start by extending the schedulability tests with a condition on periods. Let AMS be an aggregated modal server.

© 2017 Romain GRATIA

Theorem 6. *If* $\forall MS_i \in AMS, \forall \tau_l \in \Gamma_{AMS}(LO), T_i \mid T_l$, *and* $U_{\Gamma_{AMS}(LO)} \leq \sum_{MS_i \in AMS} U_i$, *then* MS_i *ensures* $\Gamma_{MS_i}(LO)$ *schedulability.*

This condition on periods is only met for very specific cases. For more general cases, we can reuse and extend the schedulability test based SBF and DBF.

Authors in [69] extended the definition of SBF to two servers. They proved that the SBF of the aggregation of two periodic servers is the sum of their SBFs. We further extend this result to a set Γ_S of p periodic servers on a uniprocessor.

Lemma 2. *The* SBF_{Γ_S} *of the aggregation of p periodic servers* S_i *with* SBF_i *and forming a set* Γ_S *and executed on a uniprocessor is:*

$$SBF_{\Gamma_S}(t) = \sum_{i=0}^{p} SBF_i(t), \forall t \geq 0. \tag{5.4}$$

Proof. SBF_{Γ_S} gives the minimum amount of execution time provided by all periodic servers aggregated from Γ_S. Because these periodic servers are scheduled together on a uniprocessor, none of them is executed in parallel. Remind that each SBF_i corresponds to the minimum execution time provided by each periodic server S_i. Therefore the minimum amount of execution time provided by SBF_{Γ_S} is equal to the sum of those provided by each of the p servers S_i. □

Let Γ be a set of periodic tasks with implicit deadlines, and Γ_S a set of periodic servers. Then, the schedulability test presented in theorem 5, based on the DBF and SBF functions, can be extended as follows:

Theorem 7. Γ *is correctly scheduled by periodic servers in* Γ_S *if:*

$$\forall t \in [0; LCM(\Gamma \cup \Gamma_S)], DBF_{\Gamma}(t) \leq SBF_{\Gamma_S}(t) \tag{5.5}$$

With $LCM(\Gamma \cup \Gamma_S)$ *the least common multiple of periods of all tasks and servers respectively in* Γ *and* Γ_S.

This also holds in our case as we consider only modal servers respecting the sequential condition. As for theorem 5, SBFs and DBFs have only to be compared at the deadlines of the tasks in the task set over the hyper–period.

© 2017 Romain GRATIA

5.6 Conclusion

In this chapter, we have presented and justified the correctness of our hierarchical scheduling framework. We also justified the use of the periodic resource model to schedule tasks in the slack time of HI tasks.

Then we introduced a new kind of servers called modal servers. These servers are used in three different ways. First we presented slackful modal server that schedules a single HI task and a set of LO tasks in the slack time of the HI task. We provided the proof of the correctness of the mixed–criticality scheduling for slackful modal server and the schedulability tests to verify the schedulabiltiy of LO tasks in HI task slack time. Second, we presented slackless modal server used to schedule LO tasks not schedulable in any HI task slack time. Finally, we presented how the limitation of the slack time provided by each slackful modal server can be overcome by aggregating the slack time of these modal servers and presented their proper schedulability tests.

In the next chapter, we deal with the scheduling of these modal servers on a multi–processor platform. This is in done in two steps. First, we perform and justify the choice of the multi–processor real–time scheduling algorithm used to scheduled modal servers. Secondly, we present how we find a good partitioning of LO tasks in modal servers.

© 2017 Romain GRATIA

6 The performances of the mixed–criticality scheduling policy

Table of Contents

In the previous chapter, we described the structure of our mixed–criticality scheduling algorithm, that is a hierarchical scheduling framework. To enforce our mixed–criticality scheduling policy, that takes advantage of HI task slack time, we defined a new kind of execution server called modal servers. These servers are periodic servers, and as such can be scheduled as periodic tasks with implicit deadlines by a multi–processor real–time scheduling algorithm.

In this chapter, we complete the design of our hierarchical scheduling framework called GMC–RUN, for *Generalised Mixed–Criticality RUN*, to make it efficient. Two aspects of our approach still have to be specified: the top level scheduling algorithm and the allocation of LO tasks in slackful modal servers, aggregated or not. These two aspects have to be dealt with the aim to obtain the best schedulability performances, measured with theoretical and experimental metrics.

To ensure the efficiency of the top level of the hierarchical scheduling framework, we carefully choose its scheduling algorithm among several multi–processor real–time scheduling algorithms against several criteria. These criteria concern not only theoretical but also practical aspects of the multi–processor scheduling algorithms.

The efficiency of the low level scheduling algorithm depends on the efficient use of the HI task slack time, that is on how the allocation of LO tasks to modal servers is made. This allocation problem is an optimisation problem. Indeed, LO tasks executed in modal servers do not require additional execution resources. We have hence to find the allocation that reduces as much as possible the utilisation of the overall system, in order to reduce the number of processors. It corresponds to a difficult optimisation problem that we solve using an evolutionary algorithm.

We start this chapter with the choice of the top level scheduling algorithm of our hierarchical scheduling framework. Based on our criteria, that concern the number of preemptions entailed, the schedulability test among others, RUN has been chosen. We then present the most essential aspects of the algorithm RUN. Next, we present the conditions to use the different types of modal servers, i.e slackful modal servers, slackless modal servers and aggregated modal servers. Finally, we present the conditions for a set of modal servers to be schedulable by RUN.

In the second section of this chapter, we address the allocation of LO tasks in modal servers. We first describe our optimisation problem. We then give the principles of the evolutionary algorithm. Finally, we explain why they are an appropriate method to solve our optimisation problem and describe how we efficiently use them to find a good allocation.

© 2017 Romain GRATIA

Finally, theoretical and experimental assessments of GMC–RUN are carried out. Theoretical assessment is here to give an insight on the worst performances of our scheduling algorithm, while the experimental one gives an insight on its average performances. The theoretical assessment is twofold. First, we determine a lower and an upper bounds on the number of processors required by GMC–RUN. Second, we compute the speed–up factor of GMC–RUN. Experimental assessment is carried out by computing the schedulability ratio and the number of entailed preemptions from randomly generated task sets.

6.1 The choice of the top level scheduler

The role of the top level scheduling algorithm of the hierarchy is to ensure that modal servers are correctly executed. In our case, it is a multi–processor scheduling algorithm. The overall performances of our scheduling algorithm depend on the performances of the schedulers used at the top level and at the low level of the hierarchy. Therefore the choice of the top level scheduling algorithm is made based on a list of criteria. Based on our criteria, RUN [42] appears to be the best choice. After justifying this choice, we present the essential features of RUN to understand how the conditions of use of each kind of modal server, i.e slackful modal server, slackless modal server and aggregated modal servers, are fulfilled.

6.1.1 Justification of the choice

For the choice of our top level scheduling algorithm, we assess the global scheduling algorithms Reduction to UNiprocessor (RUN) [42], U–EDF [43] and DP–Fair [70] and partitioned EDF. These algorithms are the state–of–the–art multi–processor scheduling algorithms either for global scheduling algorithms or partitioned ones.

The first criterion used to select the possible scheduling algorithms is the ability to schedule periodic tasks with implicit deadlines and synchronously started, since modal servers behave as such. Indeed, modal servers have a periodic resource model, that behaves similarly to the periodic task model. All the considered scheduling algorithms fulfil that criterion.

To perform our choices we also use the following criteria:

1. Schedulability performances: to select our scheduling algorithm several indicators are used such as the speed–up factor or the ratio of randomly generated tasks successfully scheduled.

© 2017 Romain GRATIA

2. Number of preemptions and migrations per job entailed: it gives an insight prior to the implementation on the actual efficiency and practicability of the scheduling algorithm. Schedulability performance assessments generally do not take into account the overheads incurred by preemptions and migrations. A high number of preemptions per job negatively impacts the schedulability performances.

3. An algorithm whose implementation is feasible. Indeed, as said in the problem statement, there are algorithms based on concepts that are difficult to implement. One such algorithm is the fluid algorithm [61] that must provide at each instant execution time to each task proportionately to their utilisations. It requires the simultaneous use of a same processor by several tasks at each instant. This also concerns all the other overheads not originating from preemptions or migrations. For instance, if the time to carry out a scheduling decision or a task releasing is too large, it can nullify the schedulability performances even for a scheduling algorithm presenting the best theoretical schedulability performances.

We now compare all the considered scheduling algorithms against these three criteria.

1- Theoretical schedulability performances

All the considered global scheduling algorithms are optimal scheduling algorithms for multi–processor platforms. This means that they are able to schedule on a platform with m processors any task set whose utilisation is at most equal to m.

For its part, partitioned EDF has, as for all partitioned scheduling algorithms, a provable utilisation bound for periodic tasks with implicit deadlines of only $\frac{m+1}{2}$ [31]. This limit only ensures that task sets with a utilisation no higher than $\frac{m+1}{2}$ are schedulable on m processors.

On that criteria, global scheduling algorithms have a clear advantage over the partitioned one.

2- Number of preemptions and migrations

The negative impact of preemptions and migrations on schedulability performances becomes larger as their numbers increase. Therefore, scheduling algorithms that entail the smallest number of preemptions or migrations should be favoured.

The numbers of preemptions and migrations entailed are more generally estimated by simulating the execution of scheduling algorithms. Each considered scheduling algorithm has been presented [42; 43; 70] with a comparison based on the number of preemptions and migrations per job. From these comparisons the following conclusions can be drawn.

© 2017 Romain GRATIA

First, fair scheduling algorithms, such as DP–Fair, entail far more preemptions and migrations than any other considered scheduling algorithm, both partitioned and global scheduling algorithms. Besides, it significantly alters their schedulability performances. The impact on schedulability performances has been shown in [27], where it appears that partitioned EDF presents better schedulability performances than a fair optimal scheduling algorithm. And this, although that latter kind of scheduling algorithm is theoretically optimal and the former is not.

Second, partitioned EDF presents the best performances in terms of preemptions and does not entail any migration, which is an advantage when considering overheads [27; 36].

Third, RUN implementation [36] has been compared to partitioned EDF. This comparison has shown that for systems with a utilisation such that partitioned EDF presented no deadline misses, RUN entailed the same number of preemptions than partitioned EDF. RUN entailed more preemptions than partitioned EDF, only for systems with higher utilisations that are not all schedulable by partitioned EDF.

Fourth, U–EDF and RUN present very close performances on these two criteria [43]. U–EDF only slightly incurs more preemptions and migrations as the number of processors becomes larger compared to RUN.

Besides, these comparisons have also shown that RUN, and in a lesser extent U–EDF, are remarkably scalable as the numbers of preemptions and migrations change little with the size of the scheduled system, that is with the number of processors fully utilised.

3- Practicability of the algorithms

Preemptions and migrations are not the only sources of overheads than can degrade the performances of a scheduling algorithm. These performances can also be degraded by the time required to take a scheduling decision or the time required to process the releasing of a task.

RUN practicality has been questioned as RUN scheduling decisions are based on a complex structure derived offline called *reduction tree*. These decisions involve non trivial operations that have raised concerns on the complexity of the implementation of RUN and on its efficiency. But an implementation has been performed in [36]. Authors state that the RUN algorithm is implementable without too much complexity and does not entail prohibitive overheads. RUN is shown to be quite competitive against partitioned EDF.

On that criterion, partitioned EDF is shown to entail among the lowest overheads compared to all scheduling algorithms.

© 2017 Romain GRATIA

To our knowledge no implementation of DP–Fair and of U–EDF are currently available.

Conclusion

In conclusion, U–EDF, RUN and partitioned EDF present really interesting performances concerning preemptions and migrations. Yet, compared to partitioned EDF, U–EDF and RUN offer far better theoretical schedulability performances. RUN emerges as it entails slightly fewer preemptions than U–EDF. Furthermore, an implementation of RUN has been performed demonstrating its feasibility. For all these reasons we choose RUN as the top level scheduling algorithm of our hierarchical scheduling framework. Before presenting the conditions to fulfil to schedule modal servers with RUN, we need to present essential features of RUN.

6.1.2 RUN description

RUN is a hard multi–processor real–time scheduling algorithm that can schedule periodic tasks with implicit deadlines synchronously started. RUN approach is original as it mixes the principles of a partitioned algorithm with those of a global scheduling algorithm. But, it differs from semi–partitioned scheduling algorithms as task migrations are not limited. RUN scheduling algorithm is composed of two steps: one performed offline, another online. The first step consists in deriving offline a hierarchical structure called the *reduction tree*. This tree is based on particular task and server models called respectively fixed–rate tasks and fixed–rate servers.

Fixed–rate task and fixed–rate server

First, we introduce the fixed–rate task model used by RUN authors and upon which the fixed–rate server is based. We then describe how the more usual model of the periodic task with implicit deadlines is handled.

Fixed–rate task. Authors in RUN have their own model of tasks called fixed–rate tasks. Before giving a definition of such task, we introduce the definitions of a rate and of activation times.

Definition 25 (Rate)**.** *A rate $\mu \in [0,1]$ of a task is the proportion of a processor it needs to correctly execute.*

© 2017 Romain GRATIA

Hence, in case of periodic tasks the rate is equal to the utilisation of the task.

Definition 26 (Activation time)**.** *The activation times of a task are the instants when the task becomes ready to be executed.*

A fixed–rate task is defined as follows:

Definition 27. *A fixed–rate task is defined by a rate* $\mu \in [0,1]$ *and a set D containing all the task activation times and absolute deadlines.*

Fixed–rate task and periodic task with implicit deadlines. A periodic task with implicit deadlines can be modelled with a fixed–rate task. The rate of the fixed–rate task is equal to the utilisation of the periodic task. The set D of activation times and deadlines is equal to the set $D = \{j \cdot T, j \in \mathbb{N}\}$, where T is the period of the periodic task.

These fixed–rate tasks are then executed in RUN by servers called fixed–rate servers.

Fixed–rate server. A server is used to run tasks, that RUN authors refer to as its clients. When the server is executed, it selects one of its clients using a scheduling algorithm, and executes it. RUN authors use a particular model of a server called fixed–rate server:

Definition 28. *A fixed–rate server is defined through: a set of clients* Γ*, a set of replenishment times D, and an execution rate* $\mu \in [0,1]$*. Such a server is denoted* $S(\mu, D, \Gamma)$*. A server schedules the jobs of its client following its own scheduling policy.*

Sufficient conditions for clients schedulability in a fixed–rate server using Earliest Deadline First (EDF) are provided in theorem III.1 in [42]. The corresponding demonstration of the correctness of the scheduling follows the theorem. It first requires that servers are correctly scheduled. The server is correctly scheduled if for any two consecutive replenishment times t and t' of D, then the server is executed $(t' - t) \cdot \mu$ time units. Once a set of servers is proved to be schedulable, a schedulability test must be performed for each server to prove that its client set is also schedulable.

If the EDF fixed–rate server $S(\mu, D, \Gamma)$ has replenishment times equal to all activation times of tasks in Γ, then $U_\Gamma \leq \mu$ entails Γ is schedulable by S. Note that the clients of servers can be servers as well. An EDF fixed–rate server with a set of servers Γ_S as clients ensures its clients are schedulable if its rate is greater than the sum of the rates of its clients assuming $D = \cup_{S(\mu_j, D_j, \Gamma_j) \in \Gamma_S} D_j$, where D is the set of activation times of the server and D_j those of each client.

We can now, describe how the reduction tree is built offline.

© 2017 Romain GRATIA

Task	Period	Criticality	C(LO)	C(HI)	U(LO)	U(HI)
τ_1	5	HI	1	3	0.2	0.6
τ_2	2	HI	0.5	1.5	0.25	0.75
τ_3	8	HI	0.8	3.2	0.1	0.4
τ_4	8	LO	3.2	3.2	0.4	0.4
τ_5	15	LO	3.75	3.75	0.25	0.25
τ_6	12	LO	2.4	2.4	0.2	0.2
τ_7	12	LO	2.4	2.4	0.2	0.2

Table 6.1: Example of MC task set with two criticality levels (same table that on page 64)

Offline construction of the reduction tree

Upon this notion of fixed–rate server, RUN builds offline a hierarchy of servers called the reduction tree. The reduction tree is used online to elect tasks to execute. The first step to build this tree of servers is to partition the task set.

Partitioning the task set. RUN partitions a task set Γ into disjoint subsets. Each of these subsets Γ_j is scheduled in a EDF fixed–rate server. Hence, each subset must have a utilisation lower or equal to 1 in order to be correctly scheduled by its EDF fixed–rate server. This condition will be referred to as the **packing condition**. But these subsets must also respect a second condition: the sum of the utilisations of any two subsets must be strictly greater than 1. This ensures that no couple of subsets can be scheduled on a single processor. Each such subset represents a uniprocessor scheduling problem. Fixed–rate servers whose clients respect these constraints are referred to as **primal servers**.

The operation, that defines the set of primal servers, is called *PACK* [42]. This operation can have multiple possible solutions that are all correct. It is first applied on a set of tasks but can also be applied to a set of servers.

One possible result of the *PACK* operation for the task set presented in table 6.1 is given in figure 6.1. Primal servers are the servers S_{15}, S_{26} and S_{347}. Server S_{15} schedules tasks τ_1 and τ_5, S_{26} schedules τ_2 and τ_6 and server S_{347} schedules tasks τ_3, τ_4 and τ_7. Server S_{15} rate is equal to the sum of the utilisations of tasks τ_1 and τ_5 and its set of activation times is the union of those of tasks τ_1 and τ_5. The same holds for other servers with their corresponding tasks. Notice that server S_{347} has a rate equal to 1.0. During its execution, this server uses a complete processor.

Reducing the number of processors. The second operation aims at reducing the number of processors required to schedule all the created primal servers by enabling the sharing of processors between these primal servers. It is achieved by introducing a second kind

© 2017 Romain GRATIA

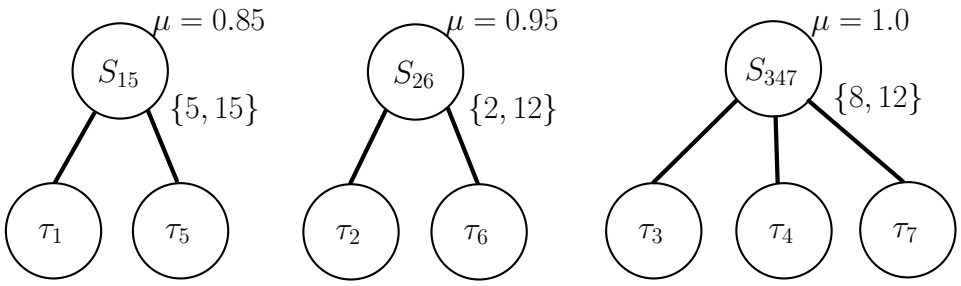


Figure 6.1: One possible result of the packing operation for task set presented in table 6.1

of fixed–rate servers called **dual servers**. These are created by a specific operation called *DUAL*. They are basically used to schedule the idle time of primal servers, that is the amount of time a processor remains idle if it was used to schedule a single primal server. The number of processors is reduced by merging the idle time of several primal servers. This is achieved by regrouping dual servers in other primal servers.

Generation of the reduction tree. The creation of primal and then dual servers is regrouped in one operation called *REDUCE*. The *REDUCE* operation results in the scheduling of tasks or of servers by two levels of servers: one with primal servers and one with dual ones. The two levels can be modelled by a tree structure in which a node represents a server or a task, and a branch binds a server to one of its clients. The *REDUCE* operation is first applied on the set of fixed–rate tasks. Then *REDUCE* is applied **iteratively** to the resulting servers. The algorithm stops when the *REDUCE* operation produces a single primal server. The tree structure produced by the sequence of the *REDUCE* operations is called the *reduction tree*. Primal servers that schedule fixed–rate tasks and not dual servers are called **leaf primal servers** .

As the *PACK* operation can have several possible results, the reduction tree is not unique for a task set. But all trees schedule the task set with the same number of processors.

One possible resulting reduction tree for the task set is presented in figure 6.2. The final primal server used to schedule the dual servers is called S_{root} and has a rate of 1.0. Note also that, based on the recommendation of RUN authors, we added an idle task τ_I. The purpose is to have an overall utilisation of the task set equal to an integer. In this example, it is equal to 3. The idle task set has a rate of 0.2 and an arbitrary deadline of 15. When this task is scheduled then the processor is idle. Servers marked with a "*" correspond to dual servers.

This reduction tree is then used online to determine the tasks to execute. For more details on the online behaviour of RUN see the article [42].

© 2017 Romain GRATIA

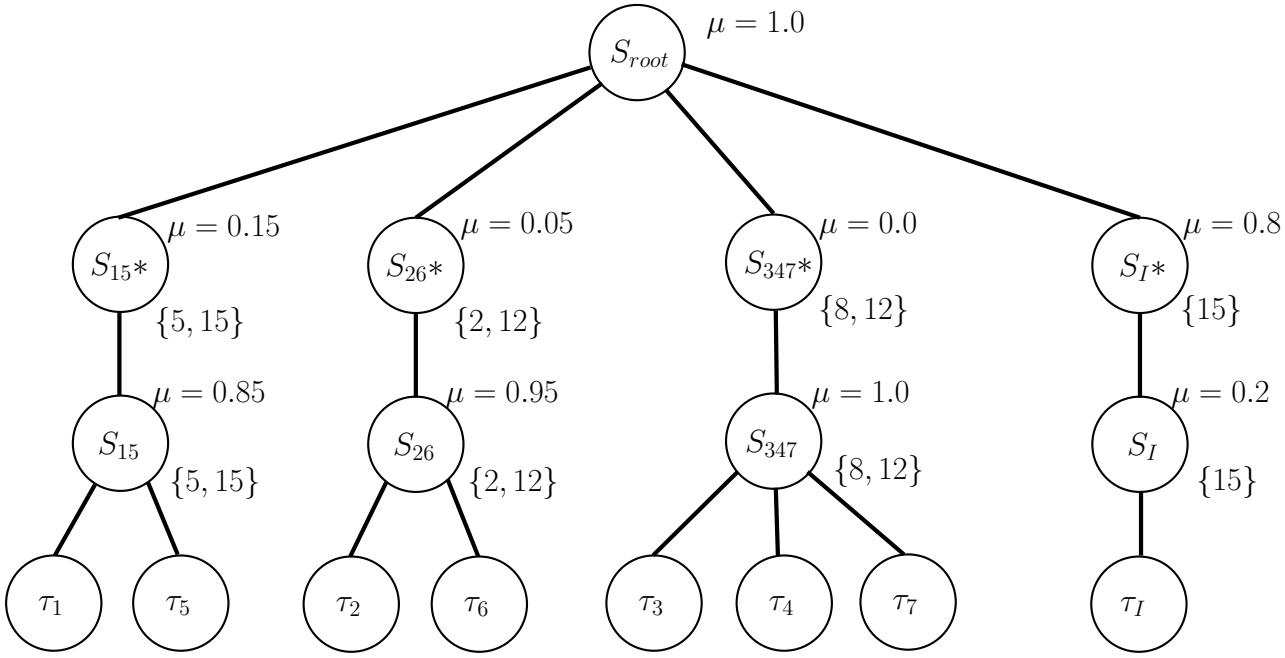


Figure 6.2: One possible reduction tree for task set presented in table 6.1

RUN schedulability test

A task set is schedulable by RUN on m processors if the following theorem is respected:

Theorem 8. *A task set* Γ *is schedulable on m processors by RUN if:*

$$U_\Gamma \leq m \tag{6.1}$$

A consequence of this theorem is that we can estimate the minimal number p of processors required to schedule a task set Γ with RUN from the task set utilisation U_Γ with the following formula:

$$p = \lceil U_\Gamma \rceil \tag{6.2}$$

Where $\lceil x \rceil$ gives the smallest integer larger or equal to x.

6.2 GMC–RUN: scheduling modal servers with RUN

In section 5.3.2 p 69 of the previous chapter, we presented the theorem 3 that states the conditions for the correct scheduling of mixed–criticality tasks in modal servers. One of the conditions of this theorem is the correct scheduling of modal servers.

In this section, we explicit the conditions to verify whether a set of modal servers is schedulable by RUN. It first requires that a reduction tree can be built from a set of modal

© 2017 Romain GRATIA

servers and thus respects the rules of the PACK and DUAL operations. Then, we define what an allocation is. Finally, we present the schedulability test for the scheduling of modal servers by GMC–RUN.

6.2.1 Conditions to schedule the different types of modal servers

In this section, we justify that the different types of modal servers can be scheduled with RUN. Indeed, the computation of the reduction for RUN requires the respect of the conditions for the PACK operation. In particular, the possibility of forming primal servers with utilisation lower or equal to 1.

Slackful and slackless modal servers

Slackful and slackless modal servers timing parameters are those of their providing tasks or of their LO tasks. Hence, the utilisation of these modal servers is at most equal to 1. Besides, as previously said, modal servers behave as periodic tasks. Therefore, RUN can schedule slackful modal servers and slackless modal servers.

Aggregated modal servers

An aggregated modal server is constituted of several slackful modal servers that have to be executed sequentially, as exposed in section 5.5 p 73. Their use with RUN requires to find a way to respect this sequential condition and the conditions of the PACK operation.

RUN decomposes the multi–processor scheduling problem into several uniprocessor ones thanks to the use of the primal servers. Within these primal servers, the tasks, or in our case modal servers, are executed as on a uniprocessor. They are hence executed sequentially. Thus, slackful modal servers executed in a same leaf primal server can be aggregated to form an aggregated modal server.

6.2.2 Allocation of tasks in modal servers

The scheduling of a mixed–criticality system with RUN requires that we have determined in which modal servers each LO task is executed. We need to introduce few notations to identify the set of servers that can be built given a mixed–criticality task set Γ.

The set of derived modal servers from Γ is the union of slackful modal servers derived from HI tasks of Γ noted $\mathcal{MS}_{SF}$, slackless modal servers derived from LO tasks of Γ, and the aggregated servers that can be built from HI tasks in Γ denoted $\mathcal{MS}_{AGG}$.

© 2017 Romain GRATIA

The relation binding LO task to modal server is called an allocation of LO tasks and is defined as follows:

Definition 29 (Allocation)**.** *An allocation of LO task from mixed–criticality task set* Γ *is a function that associates to each LO task, the modal server derived from* Γ.

Following this definition, it is possible to consider allocations that cannot be deployed. This situation occurs when the sum of utilisations in HI mode of the providing tasks of the aggregated modal server is higher than 1. Similarly, allocation may correspond to situations where LO task schedulability cannot be ensured as schedulability test applied to servers fails.

Once the allocation has been performed, the mixed–criticality system results in the set of modal servers noted $\Gamma_{GMC-RUN}$. If there are aggregated modal servers, there is also a set of constraints, noted $\mathcal{C}$. These constraints are imposed by the sequential condition of the aggregated modal servers. They have to be taken into account when performing the *PACK* operation that forms the leaf primal servers. The set of modal servers $\Gamma_{GMC-RUN}$ has then to be scheduled by RUN.

6.2.3 Schedulability test of GMC–RUN

In this section, we consider that the allocation of LO tasks into modal servers has been performed. This allocation has resulted in a set of modal servers noted $\Gamma_{GMC-RUN}$ and a set of constraints for the *PACK* operation noted $\mathcal{C}$. We note $U_{GMC-RUN}$ the utilisation of this set of modal servers and m the number of processors available. The following theorem gives the condition for the correct scheduling of the set of modal servers on the m processors:

Theorem 9. *Let* Γ *be a set of mixed–criticality tasks and* $\Gamma_{GMC-RUN}$ *be the set of modal servers resulting from the allocation of tasks in* Γ *to modal servers. Let* $U_{GMC-RUN}$ *be the utilisation of* $\Gamma_{GMC-RUN}$ *and* $\mathcal{C}$ *the set of constraints of the PACK operation resulting from the allocation. If* $U_{GMC-RUN} \leq m$ *and the set of constraints* $\mathcal{C}$ *are compatible with the rules of the PACK operation then RUN can schedule the set* $\Gamma_{GMC-RUN}$.

Proof. It follows from the respect of RUN conditions for the computation of its reduction tree by all types of modal servers and of theorem 8. □

© 2017 Romain GRATIA

6.3 Finding the best possible allocation

GMC–RUN schedulability performances depend on the amount of LO task utilisation we manage to allocate to slackful modal servers and aggregated modal servers. Indeed, the utilisation of the set of modal servers $\Gamma_{GMC-RUN}$ is equal to the sum of the utilisations of the slackful modal servers and of the aggregated modal servers and of the slackless modal servers needed to schedule the unallocated LO tasks. Hence, the more LO tasks are allocated to slackful modal servers or aggregated modal servers, the smaller the utilisation $U_{GMC-RUN}$ is.

We have hence to find the allocation of LO tasks with the largest utilisation of allocated LO tasks to slackful modal servers and aggregated modal servers.

We first define our optimisation problem and explain why we use an evolutionary algorithm to solve it. We then present the principles of the evolutionary algorithm and detail the mechanisms involved to find the best possible allocation.

6.3.1 Optimisation problem

We want to maximise the utilisation of LO tasks allocated to slackful modal servers and aggregated modal servers. Hence, let us first formalise this optimisation problem by describing its inputs, solution space, and function to maximise.

Roughly speaking problem inputs correspond to the definition of tasks in Γ. The solution space is the set of all possible allocations. In order to follow the usual definition of optimisation problem as assignment problem, it is possible to characterise an allocation through a vector of variables. Each variable represents the allocation of one LO task to a modal server. Hence, this variable value is either in $\mathcal{MS}_{SF}$, or $\mathcal{MS}_{AGG}$, or the symbol $\perp$ designating the task slackless modal server server. Thus, $Alloc_{\tau}$ denotes such a variable for the LO task τ. The solution space is the set of all assignments of $(Alloc_{\tau_i})_{\tau_i \in \Gamma(LO)}$ with values from $\mathcal{MS}_{SF} \cup \mathcal{MS}_{AGG} \cup \{\perp\}$.

Let $\Gamma_{Allocated}$ be the set of LO tasks that are not allocated to their respective slackless modal server. Then, the function to maximise is :

$$\sum_{\tau_i \in \Gamma_{Allocated}} U_i(LO)$$

Yet, each solution has to comply with additional constraints derived from the schedulabilty condition, and the sequential condition required to form aggregated servers. Before providing the complete definition, we introduce the following notion to identify the condi-

© 2017 Romain GRATIA

tion in which the schedulability test based on SBF and DBF should be used to determine task LO task schedulability in modal servers. For a slackful or aggregated modal server, its set of allocated LO tasks is said *harmonic* when each LO task period is a multiple of each providing task period in this server. When, such a condition is not satisfied, the SBF and DBF test needs to be used.

We sum up our optimisation problem in the following definition:

Definition 30. *input: a mixed criticality task set,* Γ

variables: the variable representing the allocation of each LO task of Γ *to a modal server:* $(Alloc_{\tau_i})_{\tau_i \in \Gamma LO}$ *with value in* $\mathcal{MS}_{SF} \cup \mathcal{MS}_{AGG} \cup \{\perp\}$.

Objective: maximise $\sum_{\tau_i \in \Gamma Allocated} U(\tau_i)$

Constraints: solutions need to respect:

For each $s \in \mathcal{MS}_{SF} \cup \mathcal{MS}_{AGG}$ *s.t.* $\Gamma_s(LO) \neq \emptyset \wedge \Gamma_s(LO)$ *is harmonic,*

$$\sum_{\tau_i \in \Gamma_s(LO)} U_i(LO) \leq \sum_{\tau_j \in \Gamma_s(HI)} (U_j(HI) - U_j(LO)) \tag{6.3}$$

For each $s \in \mathcal{MS}_{SF} \cup \mathcal{MS}_{AGG}$ *s.t.* $\Gamma_s(LO) \neq \emptyset \wedge \Gamma_s(LO)$ *is non-harmonic, for each* $t \in \{0, ..., LCM\}$*, LCM being the least common multiple of periods of LO and providing task of s:*

$$\sum_{\tau_i \in \Gamma_s(LO)} DBF_i(t) \leq \sum_{\tau_j \in \Gamma_s(HI)} SBF_j(t) \tag{6.4}$$

and for each $s \in \mathcal{MS}_{AGG}$*,*

$$\sum_{\tau_j \in \Gamma_s(HI)} U_j(HI) \leq 1 \tag{6.5}$$

Finding an allocation reaching the optimum of utilisation allocated to non slackless servers is a discrete optimisation problem. The choice of the approach depends usually on the size of the solution space (set of possible allocations), and the complexity of an computing exact solution. Concerning the size of the solution space, there are roughly as many slackful modal server and aggregated modal server than subsets of $\Gamma(HI)$ minus 1. Then, if we note n the number of LO tasks and k the number of modal servers, the number of possible allocations is $(2^k)^n$. For instance, with five LO tasks and five modal servers it makes 33,554,432 possible allocations

An exact solver tries to produce an actual optimum for the optimization problem. An alternative to exact solution is to use heuristics that at best offer local optima. Meta heuristic approaches are generic heuristics that need to be refined, or parametrized in order to

© 2017 Romain GRATIA

obtain a solver for the optimisation problem. Here follows some insights on why we have selected a meta heuristic approach to solve our optimisation problem. First, we need only to consider schedulable allocations. Finding such allocations involves for each task a schedulability test requiring to compare supply and demand bound functions (cost depending on period arithmetic properties). Such tests are thus rather complex. Moreover, recall that the problem suffers of a combinatorial explosion of the number of modal servers. It grows exponentially in the number of HI tasks. For these reasons, approaches based on heuristics appeared more tractable than exact solution computation.

Selecting or designing an heuristic efficiently requires some analysis of the features of the solution space. Finding its topological or structural properties (like orderings, vector space structure with independence, graph structure...), is usually the key to largely improve solution space exploration efficiency. It usually allows to design either dedicated solution space exploration, or reuse existing ones. Yet, our solution space has very poor structural properties: no total orders, no real vector space structure. It only has a partial order structure based on set inclusion comparison operator. Moreover, the set of schedulable allocation has no clear convexity properties, and tends to have a low density into the solution space. Finally, despite the utility criteria is a linear function, this property is hardly interesting given the lack of structure on the solution space. Hence, we decided to use meta-heuristics. A survey on such generic approaches to heuristic definition helped us select it, [71].

Among meta-heuristics, we preferred to avoid so called single trajectory approaches to maximize the likelihood to find at least a non trivial schedulable allocation. In single trajectory approaches, a solution is improved step by step through moves into the solution space. Alternatives are approaches that take advantage of improving simultaneously several candidate solutions. Yet, the challenge is to take advantage of the known properties of the solution space to refine the meta-heuristics. To our point of view, the evolutionary algorithm approach appears the most attractive.

6.3.2 Evolutionary algorithm

Evolutionary algorithms are stochastic optimisation algorithms. These algorithms find the best possible solution by iteratively and randomly exploring the space of solutions by modifying current known solutions.

© 2017 Romain GRATIA

Principles

We seek to find the allocation of LO tasks to modal servers that maximises the utilisation of allocated LO tasks through the use of an evolutionary algorithm. We describe here the general principles of these algorithms and how they can be used to solve our problem.

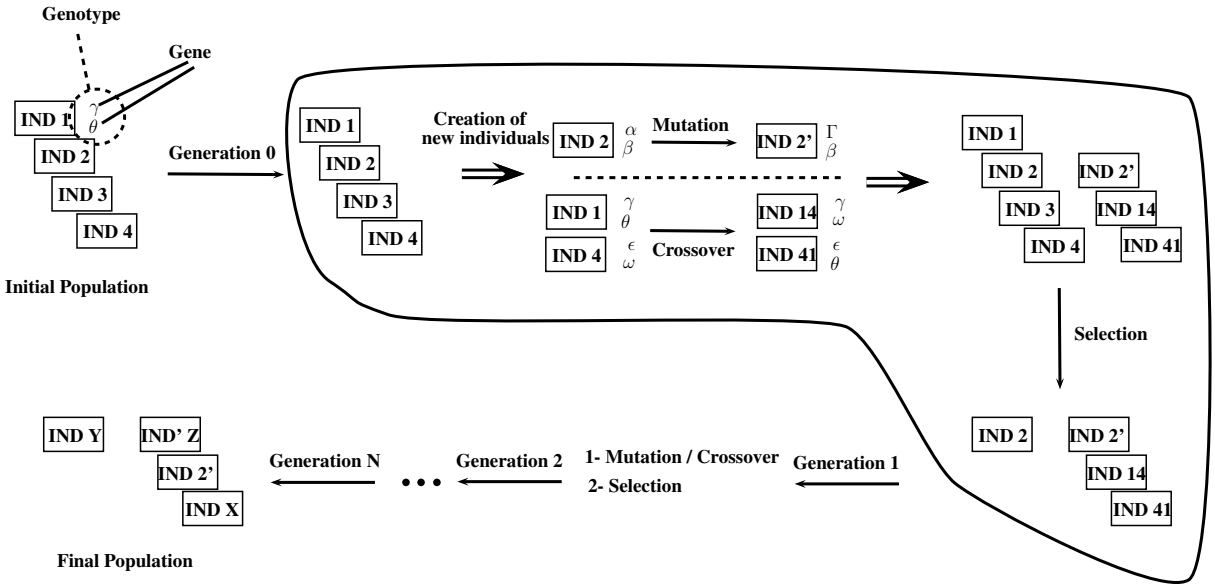


Figure 6.3: Proceeding of an evolutionary algorithm

The proceeding of an evolutionary algorithm is represented in figure 6.3.

In evolutionary algorithms, the representation of the solutions has to be carefully chosen to avoid considering results that are incorrect solutions to the problem. Each iteration, called generation, produces a subset of solutions called *population*. The population of the first generation, called initial population, is generated either randomly or composed of solutions already known. These populations are composed of *individuals*, that represent possible solutions of the problem. Each individual is described through a *genotype*, i.e a data structure, that encodes the solution. From the genotype of an individual, a *phenotype* can be decoded, that is a solution of the optimisation problem. If the genotype is not correctly chosen, the resulting phenotype may not be interpreted as a solution to the problem. In our case we have to ensure that the genotype of individual represents only the allocation of LO tasks to slackful modal server, or to aggregated modal server or to slackless modal server.

To find the best possible solution, the space of solutions has to be explored with efficient search and selection operators. Search operators produce new solutions by creating new individuals from existing ones with two objectives. The first is to ensure the conver-

© 2017 Romain GRATIA

gence towards the best possible solution. The second is to ensure that the many different subsets of solutions of the space of solutions are explored. This requires that the created individuals have a phenotype that can be interpreted as a solution of the problem. It also requires that they represent better solutions than the original individuals.

These operators are classified into two categories: the *mutation* and *crossover* operators. The role of the mutation operator is to explore the space of solutions in the "neighbourhood" of existing individuals. This is achieved by randomly altering the genotype of an existing individual. In our case, it would consist in either deallocating or allocating a LO task to a slackful modal server. The role of the crossover operator is to create potentially very different individuals from existing ones, allowing the exploration of different subsets of solutions. This is achieved by mixing the genotypes of several existing individuals. In our case, it would mean to mix the allocations of different tasks from different individuals. These new individuals are added to the population.

The selection operator has also the same two objectives. It ensures the convergence of the algorithm by selecting the best individuals to form a new population used for the next *generation*. This requires to be able to assess the solution expressed by the phenotype of each individual. In our case, the assessment of the individuals consists in computing the utilisation of the LO tasks allocated to slackful modal servers or aggregated modal servers. Second, it ensures the exploration of all subsets of solutions by ensuring the diversity of the individuals in the population, so that not same solutions are always considered. For our problem, it is ensured by keeping individuals that describe allocations that are not schedulable. We also keep individuals with aggregated modal servers that do not respect the sequential condition, i.e whose slackful modal servers cannot be sequentially executed. The resulting new population is then used in the next generations.

These operations are applied until a condition of termination is reached, that can be for instance a number of generations. The final solution to the optimisation problem is the individual with the best evaluation.

For a more detailed description of evolutionary algorithms, the reader can refer to the following documents [72; 73; 74].

Hence, the use of an evolutionary algorithm requires to define a representation of the individual and three operators: mutation, crossover and selection. But the representation used influences how the operators can be implemented. Besides, since these operators are applied on many individuals over all the iterations, they should be kept as simple as possible. Therefore, before choosing a representation for our individuals, we specify the operations required to perform our mutation, crossover and selection.

© 2017 Romain GRATIA

Designing the mutation, crossover and selection operators

The search and selection operators have to be designed with the objectives to explore different subsets of the space of solutions and to converge towards the best possible solution.

The allocation of a LO task to a modal server, as defined in definition 29, consists in adding the task to the set of LO tasks of the modal server. Therefore, operations on sets can be used to describe the mutation and crossover operators.

Mutation operator: the mutation operator is used to change the allocation of a single task in a single modal server. Since an individual represents the complete allocation of LO tasks in modal servers, a slight modification consists in only changing the allocation of a task in a slackful modal server. Hence, the mutation consists in our case to add or to remove a randomly chosen task from a randomly chosen slackful modal server.

Crossover operator: our crossover operator mixes the allocation of different tasks from several individuals. Our operator consists in taking the union of a subset of tasks noted Γ from a first individual with the complement of this task subset noted $\overline{\Gamma}$ from another individual to form a new individual. A second individual can be formed by making the opposite. That is, we take the complement of the subset $\overline{\Gamma}$ from the same first individual and the subset of tasks Γ from the same second individual.

Correction of individuals: yet, our two search operators as previously defined do not always produce individuals that can be interpreted as solutions of our optimisation problem. Indeed, our search operators can produce individuals, whose decoded phenotypes represent an allocation whose set consistency cannot be assessed. That is, they can produce individuals with aggregated modal servers whose sets of LO tasks are not equal to the union of the sets of LO tasks of each slackful modal server of the aggregation. Yet, our definition of aggregated modal server states that all modal servers have the exact same LO tasks in their LO task sets. Therefore, we have to correct these individuals to have a solution that is consistent with our definition of aggregated modal server.

To perform this correction, we compute the union of the sets of LO tasks of two slackful modal servers if a same LO task is allocated to the two slackful modal servers. The set produced by this union replaces the sets of the two slackful modal servers. It ensures that all slackful modal servers forming an aggregated modal server have the same set of LO tasks to schedule.

© 2017 Romain GRATIA

Task	Period	Criticality	C(LO)	C(HI)	U(LO)	U(HI)
τ_1	5	HI	1	3	0.2	0.6
τ_2	2	HI	0.5	1.5	0.25	0.75
τ_3	8	HI	0.8	3.2	0.1	0.4
τ_4	8	LO	3.2	3.2	0.4	0.4
τ_5	15	LO	3.75	3.75	0.25	0.25
τ_6	12	LO	2.4	2.4	0.2	0.2
τ_7	12	LO	2.4	2.4	0.2	0.2

Table 6.1: Example of MC task set with two criticality levels (same table that on page 64)

Selection operator: the selection operator has the same two objectives than the search operators. Through the selection of the best individuals of the population, it ensures the convergence of the evolutionary algorithm. It also promotes the exploration of different subsets of solutions by keeping some diversity in the population. This requires to assess the individuals.

First, the schedulability of the allocations has to be checked before. We have to identify the different allocations that have to be checked. This actually requires to identify the aggregated modal servers. This operation is similar to the one performed previously during the correction of individuals. The selection operator uses its result.

Once the different allocations of LO tasks decided, we can check their schedulability. And in case of aggregated modal servers, we also have to check that the sequential condition is respected. Then we evaluate the individuals. If an allocation has been found to be unschedulable, the evaluation of the individual is penalised. An individual is also penalised if it involves an aggregated modal server that does not fulfil the sequentiality condition.

Once the assessment performed, we select the individuals with the greatest evaluations. Selected individuals may represent unschedulable allocations or aggregated modal servers that do not respect the sequentiality condition. Indeed, the population may not be constituted of enough individuals representing schedulable individuals. These individuals ensure the diversity of the population.

Implementation of the evolutionary algorithm

Now that we know the requirements that the choice of the representation should respect, we can choose the structure of the genotype of our individuals. Then, we briefly describe how we implemented our search operators, the correction of individuals and the selection operators for the chosen representation.

© 2017 Romain GRATIA

Implementation of the genotype of the individual: the representation of our individuals has to represent only solutions to our problem and ease the application of the three operators. The allocation of tasks to slackful modal servers implies operations on sets, that is why we based our operators on set operations. A usual representation of sets is the vector. Hence, for each task, we use a binary vector of length equal to the number of slackful modal servers (or providing HI tasks). A "1" indicates that the task is allocated to the corresponding slackful modal server, otherwise a "0" is used. If a vector contains several 1s, it means that an aggregated modal server is formed. If a vector is full of 0s, then the task is allocated to a slackless modal server. The complete allocation of all LO tasks is represented by all the vectors which form a matrix. Hence, the genotype of our individuals are represented as matrices.

$$\begin{array}{c} \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccc} MS_1 & MS_2 & MS_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 6.4: Example of an individual for the task set in table 6.1

Figure 6.4 represents a possible individual for the task set in table 6.1. In this individual, task τ_5 is allocated to modal server MS_3, of providing task τ_3, that is the HI task whose slack time is used to schedule the LO task. Task τ_7 is allocated to the aggregated modal server formed with slackful modal servers MS_1 and MS_2 of providing tasks τ_1 and τ_2 respectively. Tasks τ_4 and τ_6 are allocated to slackless modal servers.

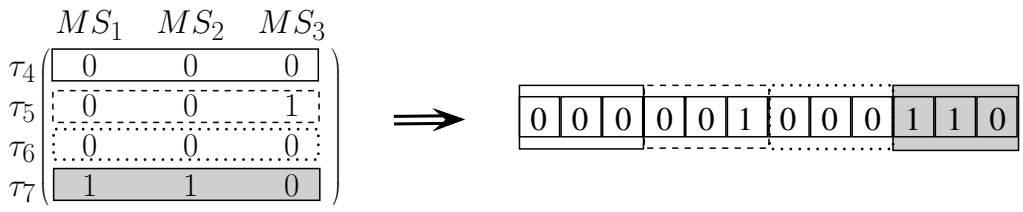


Figure 6.5: Change of the representation of the matrix of an individual

In the following, to ease the description of the action of our operators, we represent the genotype as a vector, as illustrated in figure 6.5. Each block of three in the vector represents the allocation of a task into the different slackful modal servers.

© 2017 Romain GRATIA

Implementation of the mutation operator: the mutation operator has to change the allocation of a single task to a single slackful modal server. This is achieved by randomly choosing an entry of the vector. The value of the corresponding entry is flipped to the opposite value. It means if the entry is a "1", then a new individual is created with a "0" at the same location and vice versa. The mutation hence results in the allocation or de–allocation of tasks from a slackful modal server. It can also form aggregated modal server, if the task was already allocated to a different slackful modal server.

An example is given in figure 6.6. In this example, the entry corresponding to the allocation of τ_7 to modal server MS_1, containing HI task τ_1, is changed from 1 to 0, in grey in the figure.

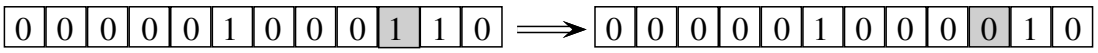


Figure 6.6: Example of a mutation

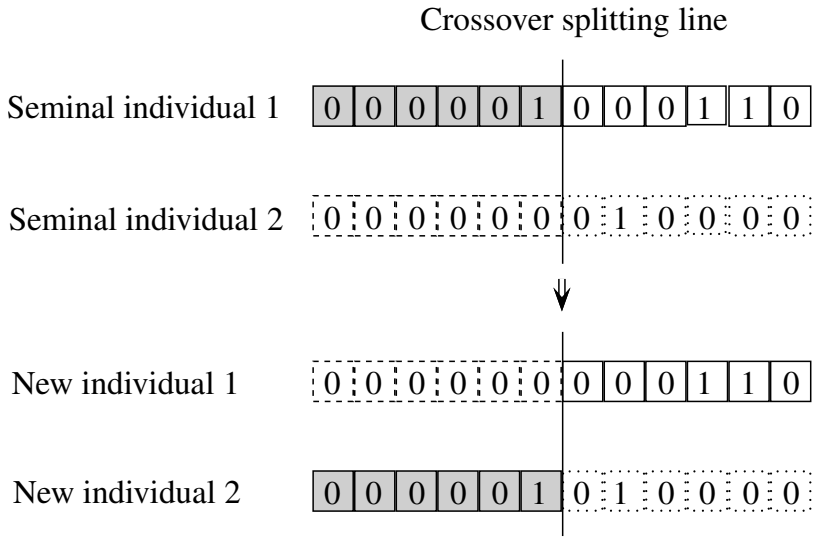


Figure 6.7: Example of a crossover

Implementation of the crossover operator The crossover operator has to form two new individuals from two existing ones by mixing their allocations. This mixing of the allocations is performed by using the union operation. We take the union of the allocations of a subset of LO tasks from a first individual with the complementary of this subset of LO tasks from another individual to form a new one. The second individual is formed by making the opposite. This operation is performed by splitting the vector of our seminal individuals in two parts and forming two new individuals. For each new individual, we use a different part of the vector from each original individual.

© 2017 Romain GRATIA

An example is given in figure 6.7. The vectors of our seminal individuals are split along the so called crossover splitting line, that corresponds to split the matrix between their 2^{nd} and the 3^{rd} rows. Then we use one part of each seminal individual to form a new one. A second individual is formed using the other part of each seminal individual.

Correction: the correction ensures that the individuals represent a solution consistent with our definition of the aggregated modal server. It has to ensure that each slackful modal servers of an aggregated modal server have the same LO tasks allocated. A binary AND operation allows to detect whether the tasks are allocated to a same slackful modal server. In this case, their vectors are replaced by their unions through the use of the binary OR operation. During this correction, we identify the different allocations represented by the individual.

We illustrate the result of the correction in figure 6.8. In the left of the figure, we have an individual with tasks τ_5 and τ_7 allocated to the slackful modal server MS_1. But τ_5 is also allocated to slackful modal server MS_3 and τ_7 to slackful modal server MS_2. This individual is inconsistent with our definition of the aggregated modal server. Indeed, two tasks allocated to a same slackful modal server are not allocated to all the same slackful modal servers. This individual has to be corrected and replaced by the individual represented on the right. In this individual, tasks τ_5 and τ_7 are allocated to all the same slackful modal servers MS_1, MS_2 and MS_3.

$$\begin{array}{c} \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccc} MS_1 & MS_2 & MS_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \quad \Longrightarrow \quad \begin{array}{c} \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccc} MS_1 & MS_2 & MS_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

Figure 6.8: Correction of individuals

Implementation of the selection operator: the selection operator selects the individuals to form the population for the next generation of the evolutionary algorithm. It requires to evaluate the individuals. To do so, it uses the identification of the different allocations from the correction. We first check the schedulability of each allocation. The schedulability test used depends on whether the periods of tasks can be divided by the periods of modal servers.

© 2017 Romain GRATIA

Task	Period	Criticality	C(LO)	C(HI)	U(LO)	U(HI)
τ_1	5	HI	1	3	0.2	0.6
τ_2	2	HI	0.5	1.5	0.25	0.75
τ_3	8	HI	0.8	3.2	0.1	0.4
τ_4	8	LO	3.2	3.2	0.4	0.4
τ_5	15	LO	3.75	3.75	0.25	0.25
τ_6	12	LO	2.4	2.4	0.2	0.2
τ_7	12	LO	2.4	2.4	0.2	0.2

Table 6.2: Example of MC task set with two criticality levels (same table that on page 64)

In the case that task periods can be divided, the test based on utilisation, from theorem 4, is used. It simply consists in computing the utilisations of the set of LO tasks and compare it with the utilisation of the whole slack time of slackful modal servers.

If the condition on periods is not met, we use the tests based on SBF and DBF functions described in theorem 5. We perform that test by comparing these two functions at all the deadlines of the tasks over the hyper–period, and not at all instants. This reduces the required number of comparisons.

The sequentiality condition, that states that slackful modal servers of aggregated modal servers have to be executed sequentially, is checked by summing the utilisations of the corresponding slackful modal servers. If that utilisation is lower or equal to one, the aggregated modal server is correct.

Once the schedulability of all allocations and sequential conditions for all aggregated modal servers have been checked, we evaluate the individual. It is performed by summing the utilisations of the LO tasks. If one allocation schedulability test failed or if the sequential condition of one aggregated modal server is not respected, this sum is multiplied by a fixed penalty factor of -10000.

Once all individuals of the population have been assessed, the selection operator selects a number of individuals. This number is equal to the sum of the number of modal servers and the number of LO tasks.

If we consider the individual in figure 6.4, it appears that the allocation of τ_5 in modal server MS_3 is unschedulable as it does not fulfil the theorem 7. Besides, the aggregated modal server does not fulfil the sequentiality condition. Indeed the sum of the utilisation of the two modal servers is greater than 1 (0.75 + 0.6 = 1.35 > 1). Hence, the overall evaluation of the individual is $(0.2+0.25)\cdot -10000 = -4500$.

© 2017 Romain GRATIA

Example: application of the evolutionary algorithm

Now that we know which scheduling algorithm is used as global scheduler and how to perform the allocation, we can apply our algorithm on the task set presented in table 6.2. The allocation results in a set of modal servers composed of one slackful modal server, one aggregated modal server and one slackless modal server.

With our evolutionary algorithm we find that τ_4 can be allocated to slackful modal server MS_2 and tasks τ_5 and τ_6 can be allocated to the aggregation of MS_1 and MS_3. Task τ_7 is allocated to the slackless modal server MS_4. With this allocation the overall utilisation of the system is reduced from 2.8 to 1.95, making this task set schedulable on 2 processors instead of 3. For the record, MC–Fluid [34] ensures this task set schedulability for no fewer than 3 processors.

In the next section, we assess GMC–RUN both theoretically and experimentally. First theoretically, we prove that GMC–RUN never uses more processors than a multi–criticality system scheduled by RUN and has a speed–up factor of 2. Then for experimental assessment on randomly generated task sets, we use our evolutionary algorithm to carry out our experiments.

6.4 Assessment of GMC–RUN

We assess GMC–RUN in two different ways, first theoretically. This gives us an indication on the worst case performances of our algorithm. In particular, it indicates us whether GMC–RUN performs better than if a mixed–criticality system was scheduled as a multi–criticality system. Recall, that a multi–criticality system is a mixed–criticality system but with tasks always executed with their HI mode timing parameters. Through the use of the speed–up, we can determine how far we are from a theoretical optimal clairvoyant scheduling algorithm. But since the worst case may only be reached for very specific and unlikely task sets, we also perform a second assessment that considers likelier cases.

Second, we assess it experimentally on randomly generated task sets. This assessment gives us an insight on the average performances of our algorithm. It also enables to consider only task sets that are likelier to be encountered in actual systems.

6.4.1 Theoretical assessment of GMC–RUN

Let us first recall some notations. $U_{\Gamma(HI)}(HI)$ is the utilisation in HI mode of HI tasks in task set Γ, $U_{\Gamma(HI)}(LO)$ is the utilisation in LO mode of HI tasks in task set Γ and $U_{\Gamma(LO)}(LO)$ the utilisation in LO mode of LO tasks in Γ. Finally, recall that a multi–

© 2017 Romain GRATIA

criticality task set is schedulable by any optimal scheduling algorithm on m processors as long as $U_{\Gamma(HI)}(HI) + U_{\Gamma(LO)}(LO) \leq m$.

The theoretical assessment of GMC–RUN is two fold.

First, we give an upper bound and a lower bound on the number of processors that GMC–RUN requires to schedule a mixed–criticality task set. We prove that GMC–RUN never requires more processors than the one required by RUN to schedule the same system but as a multi–criticality system.

Second, we compute its speed–up factor. This speed–up factor indicates how far the worst performances of GMC–RUN are from those of an optimal clairvoyant scheduling algorithm.

Number of processors

We are interested in determining whether GMC–RUN can require more processors than what would be required if the system was executed as a multi–criticality system. The following theorem gives an upper bound and a lower bound on the number of processors required to schedule a task set with GMC–RUN:

Theorem 10. *The number of processors P required to schedule a system verifies:*

$$\lceil max(U_{\Gamma(HI)}(HI), U_{\Gamma(LO)}(LO) + U_{\Gamma(HI)}(LO)) \rceil \leq P \leq \lceil U_{\Gamma}(HI) \rceil \tag{6.6}$$

Proof. **The upper bound** corresponds to the worst case execution of our partitioning algorithm of Γ. In the worst case, no LO tasks is executed in modal servers, corresponding to schedule the mixed–criticality system as a multi–criticality one. Hence, the sum of the utilisations of this multi–criticality system equals to $\sum_{\tau_i \in \Gamma} U_i(HI) = U_{\Gamma}(HI)$. **The lower bound** is proven considering that any correct mixed–criticality scheduler has to schedule any task set at least in each of its modes (LO and HI). We have to consider two executions: the first one in which the TFE never happens, and second in which it happens immediately. In LO mode, any task set cannot be scheduled with less than $\lceil U_{\Gamma(HI)}(LO) + U_{\Gamma(LO)}(LO) \rceil$ processors. Similarly, the same task set cannot be scheduled in HI mode with less than $\lceil U_{\Gamma(HI)}(HI) \rceil$ processors. To meet both conditions, any mixed–criticality scheduler uses for any task set at least $\lceil max(U_{\Gamma(HI)}(HI), U_{\Gamma(LO)}(LO) + U_{\Gamma(HI)}(LO)) \rceil$ processors. □

Notice that we have proved that GMC–RUN never requires more processors than the corresponding multi–criticality system scheduled by RUN or any other optimal real–time scheduling algorithm. This result is independent of the chosen method for the allocation of LO tasks to modal servers.

© 2017 Romain GRATIA

Speed–up factor: background

Recall from section 2.3.2 that the speed–up is regularly used to theoretically assess the performances of real–time scheduling algorithms. The principle of the speed–up factor is to compare the schedulability performances of the considered scheduling algorithm to the hypothetical performances of a clairvoyant optimal scheduling algorithm. Recall also, that the lower a speed–up is the better the performances of the considered algorithm are.

The speed–up factor was defined in [30] as the ratio s by which to increase the speed of processors such that the following inequality holds:

$$max_I \frac{S_s(I)}{A_1(I)} \leq c \tag{6.7}$$

Where $S_s(I)$ is the cost for the online scheduler to schedule the input I with processors of speed s. $A_1(I)$ is the cost for the clairvoyant scheduling algorithm to schedule the same input with processors of speed 1. c is a constant bounding the ratio.

It is generally sought to determine how much faster processors should be so that a considered algorithm can reach the best possible performances. We hence compare our algorithm to an optimal clairvoyant scheduling algorithm. This also means that we want a bounding constant equal to 1.

Speed–up factor for GMC–RUN

The following theorem gives the value of the speed–up factor of GMC–RUN:

Theorem 11. *The speed-up factor of GMC–RUN is 2.*

This speed–up factor is computed by comparing the performances of our algorithm GMC–RUN and an optimal clairvoyant scheduling algorithm.

Before beginning the proof, we recall the following mathematical property:

$$0 \leq x \leq y \Rightarrow \lceil x \rceil \leq \lceil y \rceil \tag{6.8}$$

Proof. In a mixed–criticality system, an optimal clairvoyant algorithm has the information of how long each job will execute. Therefore, with such algorithm we can determine prior execution whether the system will execute in HI or LO mode.

To express the cost to schedule an input I by each algorithm, we use the number of processors. For a given task set the maximum cost for any optimal algorithm is, as demonstrated in theorem 10 is $\lceil U_\Gamma(HI) \rceil$.

© 2017 Romain GRATIA

Since an optimal clairvoyant scheduling algorithm knows how the execution will unfold, it only need to use the maximum required number of processors for the mode that will be active. Hence its cost is given by:

$$max(\lceil U_{\Gamma(HI)}(HI)\rceil, \lceil U_{\Gamma(HI)}(LO) + U_{\Gamma(LO)}(LO)\rceil) \tag{6.9}$$

Using our cost function in inequality 6.7, we have to find a speed s such that the following inequality holds:

$$\frac{\lceil \frac{U_{\Gamma(HI)}(HI)+U_{\Gamma(LO)}(LO)}{s}\rceil}{max(\lceil U_{\Gamma(HI)}(HI)\rceil, \lceil U_{\Gamma(HI)}(LO) + U_{\Gamma(LO)}(LO)\rceil)} \leq 1 \tag{6.10}$$

We want to prove that this inequality holds for s = 2. The proof is divided in two parts. We first demonstrate this inequality holds for $s \geq 2$. Then we prove that for any s such that s < 2, the inequality is not true any more.

$\underline{s \geq 2}$**:**

Two cases can be identified.

First case, let assume

$$max(\lceil U_{\Gamma(HI)}(HI)\rceil, \lceil U_{\Gamma(HI)}(LO) + U_{\Gamma(LO)}(LO)\rceil) = \lceil U_{\Gamma(HI)}(HI)\rceil \tag{6.11}$$

We want to demonstrate that :

$$\lceil \frac{U_{\Gamma(HI)}(HI) + U_{\Gamma(LO)}(LO)}{2}\rceil \leq \lceil U_{\Gamma(HI)}(HI)\rceil \tag{6.12}$$

As $0 \leq U_{\Gamma(HI)}(LO)$, inequality (6.8) gives

$$U_{\Gamma(LO)}(LO) \leq \lceil U_{\Gamma(LO)}(LO)\rceil \leq \lceil U_{\Gamma(HI)}(LO) + U_{\Gamma(LO)}(LO)\rceil$$

Hence, from inequalities (6.11) and (6.8), we deduce that

$$U_{\Gamma(LO)}(LO) \leq \lceil U_{\Gamma(HI)}(LO) + U_{\Gamma(LO)}(LO)\rceil \leq \lceil U_{\Gamma(HI)}(HI)\rceil \tag{6.13}$$

$$\begin{aligned} \frac{U_{\Gamma(LO)}(LO) + U_{\Gamma(HI)}(HI)}{2} \leq \frac{\lceil U_{\Gamma(HI)}(HI)\rceil + U_{\Gamma(HI)}(HI)}{2} \Rightarrow \\ \frac{U_{\Gamma(LO)}(LO) + U_{\Gamma(HI)}(HI)}{2} \leq \lceil U_{\Gamma(HI)}(HI)\rceil \end{aligned} \tag{6.14}$$

$$\lceil \frac{U_{\Gamma(LO)}(LO) + U_{\Gamma(HI)}(HI)}{2}\rceil \leq \lceil \lceil U_{\Gamma(HI)}(HI)\rceil\rceil = \lceil U_{\Gamma(HI)}(HI)\rceil \tag{6.15}$$

© 2017 Romain GRATIA

Therefore, the inequality (6.12) is verified.
Second case, let assume

$$max(\lceil U_{\Gamma(HI)}(HI)\rceil, \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil) = \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \quad (6.16)$$

We want to demonstrate that :

$$\lceil \frac{U_{\Gamma(HI)}(HI)+U_{\Gamma(LO)}(LO)}{2}\rceil \leq \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \quad (6.17)$$

From inequalities (6.16) and (6.8), we deduce :

$$U_{\Gamma(HI)}(HI) \leq \lceil U_{\Gamma(HI)}(HI)\rceil \leq \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \quad (6.18)$$

$$U_{\Gamma(LO)}(LO)+U_{\Gamma(HI)}(HI) \leq U_{\Gamma(LO)}(LO)+\lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \quad (6.19)$$

$$\begin{aligned} U_{\Gamma(LO)}(LO) \leq \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \Rightarrow U_{\Gamma(LO)}(LO)+U_{\Gamma(HI)}(HI) \\ \leq 2\times(\lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil) \end{aligned} \quad (6.20)$$

$$\frac{U_{\Gamma(HI)}(HI)+U_{\Gamma(LO)}(LO)}{2} \leq \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \quad (6.21)$$

$$\lceil \frac{U_{\Gamma(HI)}(HI)+U_{\Gamma(LO)}(LO)}{2}\rceil \leq \lceil \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \rceil = \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil \quad (6.22)$$

Inequality (6.17) is also verified. Hence, 2 is a speed-up factor of GMC–RUN.

$\underline{s < 2}$**:**

Now, we show that 2 is the lower bound of the speed-up factor of our adaptation. To prove that let assume $s = 2-\varepsilon$ with $\varepsilon \to 0^+$. We can exhibit a counter example such that :

$$\lceil \frac{U_{\Gamma(HI)}(HI)+U_{\Gamma(LO)}(LO)}{2-\varepsilon}\rceil \geq max(\lceil U_{\Gamma(HI)}(HI)\rceil, \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil) \quad (6.23)$$

Lets assume we have a task set such that :
$U_{\Gamma(LO)}(LO) = U_{\Gamma(HI)}(HI) = \lceil U_{\Gamma(HI)}(HI)\rceil$ and $U_{\Gamma(HI)}(LO) = 0$
Then, $max(\lceil U_{\Gamma(HI)}(HI)\rceil; \lceil U_{\Gamma(HI)}(LO)+U_{\Gamma(LO)}(LO)\rceil) = \lceil U_{\Gamma(HI)}(HI)\rceil$
and inequality (6.23) becomes :

$$\lceil \frac{2}{2-\varepsilon}\times U_{\Gamma(HI)}(HI)\rceil > \lceil U_{\Gamma(HI)}(HI)\rceil \quad (6.24)$$

© 2017 Romain GRATIA

As $\frac{2}{2-\varepsilon} > 1$, inequality (6.24) can be made true for $U_{\Gamma(HI)}(HI)$ large enough. This demonstrates we can exhibit a counter example for any $s < 2$.

□

This value of 2 is the second best value among current global mixed–criticality scheduling algorithms. Note that this result is true for any optimal multi–processor scheduling processor used as top level scheduling algorithm, not only for RUN.

6.4.2 Experimental assessment

In this section, we experimentally evaluate the performances of GMC–RUN on two criteria. First, we estimate the schedulability efficiency on randomly generated task sets. Second, we measure the average number of preemptions per job. Our approach is mainly compared to MC–DP–Fair [34], a variant of a MC–Fluid, known for its schedulability efficiency. Evaluation for GMC–RUN has been done using both aggregated modal servers and evolutionary algorithm. The implementation of our evolutionary algorithm is based on the Distributed Evolutionary Algorithms in Python (DEAP) framework [75].

Sampling

We need representative task set samples to estimate average performances for various combinations of periods, utilisations and proportions of HI tasks. We generate random task sets following the method described in [53]. With this method, the proportion of HI tasks P_{HI} and the maximal utilisation of modes U_{Bound} and the minimal and maximal ratio between utilisations of LO and HI modes can be set.

For a given task set Γ, let U_{Bound} be $max(U_{\Gamma(HI)}(HI), U_{\Gamma}(LO))$ and let P_{HI} be the proportion of HI tasks in Γ. The normalised utilisation $U_{norm} = \frac{U_{Bound}}{m}$ takes its values in $(0.05\mathbb{N} \cap [0.6, 1.0])$, with m the number of processors whose values are successively taken in $\{2, 4, 8\}$. For each value of U_{norm}, 500 sample task sets are generated. For each sample, the generation procedure ensures the task set parameters are consistent with U_{Bound}. The algorithm 1 show the overall procedure to create one task set.

For each new task τ_{new}, we first determine its criticality level χ_{new}. It is performed by generating randomly a variable *hi_or_low* assuming a uniform distribution between 0 and 1. If $hi\_or\_low < P_{HI}$ then it is a HI task, otherwise it is a LO one.

If χ_{new} is set to HI, we first determine the utilisation in HI mode $U_{new}(HI)$ of the task. To ensure that no execution modes has an overall utilisation greater than U_{bound}, the utilisations of a HI task is the minimum between two values. The first value is randomly drawn in $[0.02, 0.7]$ assuming a uniform distribution. The second is the difference between

© 2017 Romain GRATIA

the utilisation limit U_{bound} and the current value of the utilisation of the HI mode of the task set $U_\Gamma(HI)$. $U_\Gamma(HI)$ is then updated with the utilisation of the new task in HI mode.

Then a ratio r between task utilization in LO and HI modes, $U_{new}(HI)/U_{new}(LO)$, has to be determined. This parameter is drawn from a uniform distribution in $[1,4]$. To avoid exceeding U_{Bound}, the LO utilisation value $U_{new}(LO)$ is chosen among two values. Either it is equal to the utilisation $U_{new}(HI)$ divided by r. Or it is equal to the difference between the utilisation limit U_{bound} and the current value of the utilisation in LO mode of the task set $U_\Gamma(LO)$. $U_\Gamma(LO)$ is then updated with the utilisation of the new task in LO mode.

If χ_{new} is set to LO. The utilization value in LO mode $U_{new}(LO)$ needs to be determined. Its value is the minimum between a value randomly drawn in $[0.02,0.7]$ assuming a uniform distribution or the difference between the utilisation limit U_{bound} and the current value of the utilisation in LO mode of the task set $U_\Gamma(LO)$. $U_\Gamma(LO)$ is then updated with the utilisation of the new task in LO mode.

Finally, the period is drawn in an interval from a log uniform distribution as described in [76]. The chosen interval is $[10,100]$ with a step of 10.

The number of tasks in the task set is not fixed in advance. A task τ_{new} is generated as long as *continue* is true. The variable *continue* is checked with the function $CheckUtilisation()$ at the end of each creation of a task. It stays true until one of the utilization $U_{\Gamma(HI)}(HI)$ or $U_\Gamma(LO)$ reaches U_{Bound}.

These task sets were first used to measure schedulability ratio and the weighted schedulability ratio. Then, they were used to measure the number of preemptions entailed per job. These two metrics were also used to compare GMC–RUN with MC–DP–Fair [34], and fpEDF–VD [53].

Result exploitation

Schedulability performances: ratio of task set successfully scheduled The acceptance ratio is the percentage of task sets deemed schedulable by a scheduling algorithm. We compare schedulability efficiency computing the acceptance ratio of GMC–RUN, fpEDF–VD and MC–DP–Fair. Figures 6.9, 6.10 and 6.11 show acceptance ratio varying for normalized values of U_{Bound}/m on 2, 4 and 8 processors respectively.

Results show that GMC–RUN outperforms fpEDF–VD as GMC–RUN exhibits a schedulability ratio close to MC–DP–Fair efficiency for any number of processors and proportion of HI tasks. For instance, on 2 processors with $P_{HI} = 0.3$ and for a normalized utilization value of 0.7 GMC–RUN manages to schedule more than 95% of task sets compared with 100% for MC–DP–Fair and less than 40% for fpEDF–VD.

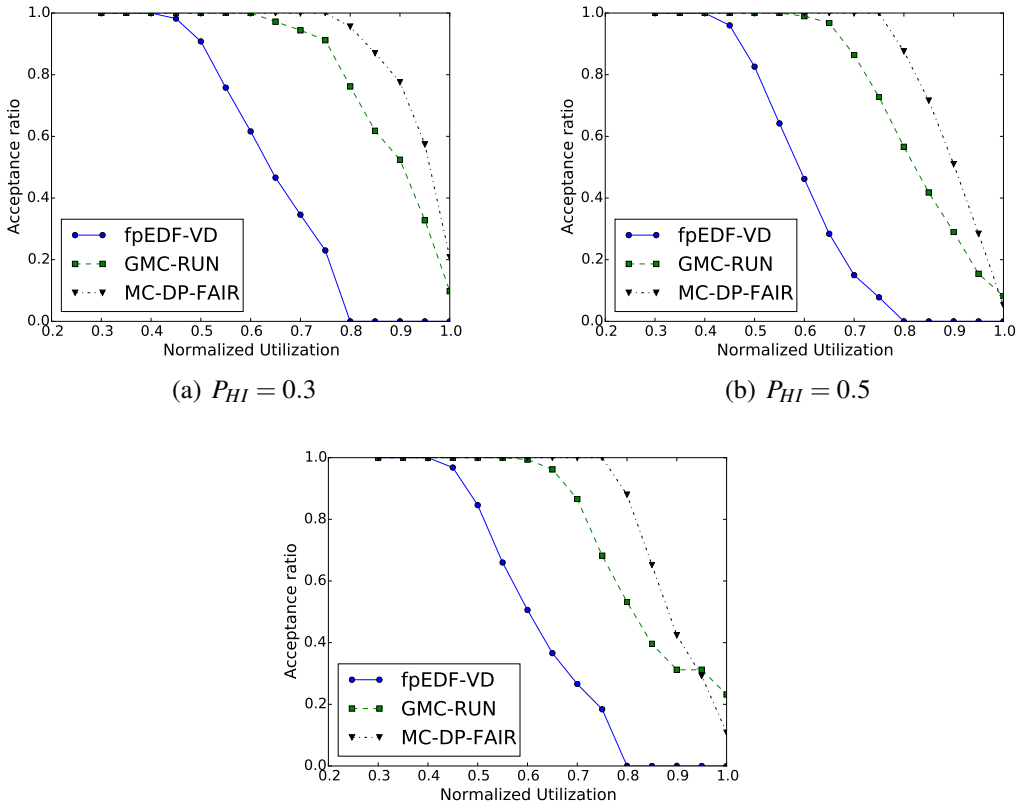
© 2017 Romain GRATIA

Algorithm 1 Task generation for experiments with two criticality levels

Input: U_{min}, U_{max}, r_{min}, r_{max}, T_{min}, T_{max}, U_{bound}, P_{HI}
Output: Γ

1:
2: $\Gamma \leftarrow \{\}$
3: $continue \leftarrow True$
4: **while** $continue = True$ **do**
5: $\quad \tau_{new} = NewTask()$
6: $\quad hi\_or\_low \leftarrow UniformDistribution(0,1)$
7: $\quad$ **if** $hi\_or\_low \leq P_{HI}$ **then**
8: $\quad\quad \chi_{new} = HI$
9: $\quad\quad U_{new}(HI) = min(UniformDistribution(U_{min}, U_{max}), U_{Bound} - U_{\Gamma(HI)}(HI))$
10: $\quad\quad U_{\Gamma(HI)}(HI) \leftarrow U_{\Gamma(HI)}(HI) + U_{new}(HI)$
11: $\quad\quad r \leftarrow UniformDistribution(r_{min}, r_{max})$
12: $\quad\quad U_{new}(LO) = min(U_{new}(HI), \frac{U_{new}(HI)}{r}, U_{Bound} - U_{\Gamma}(LO))$
13: $\quad\quad U_{\Gamma}(LO) \leftarrow U_{\Gamma}(LO) + U_{new}(LO)$
14: $\quad$ **else**
15: $\quad\quad U_{new}(LO) = min(UniformDistribution(U_{min}, U_{max}), U_{Bound} - U_{\Gamma}(LO))$
16: $\quad\quad \chi_{new} = LO$
17: $\quad\quad U_{\Gamma}(LO) \leftarrow U_{\Gamma}(LO) + U_{new}(LO)$
18: $\quad$ **end if**
19: $\quad T_{new} = GenerateTaskPeriod(T_{min}, T_{max})$
20: $\quad \Gamma \leftarrow \Gamma \cup \{\tau_{new}\}$
21: $\quad continue \leftarrow CheckUtilisation()$
22: **end while**

© 2017 Romain GRATIA



(a) $P_{HI} = 0.3$ (b) $P_{HI} = 0.5$

(c) $P_{HI} = 0.7$

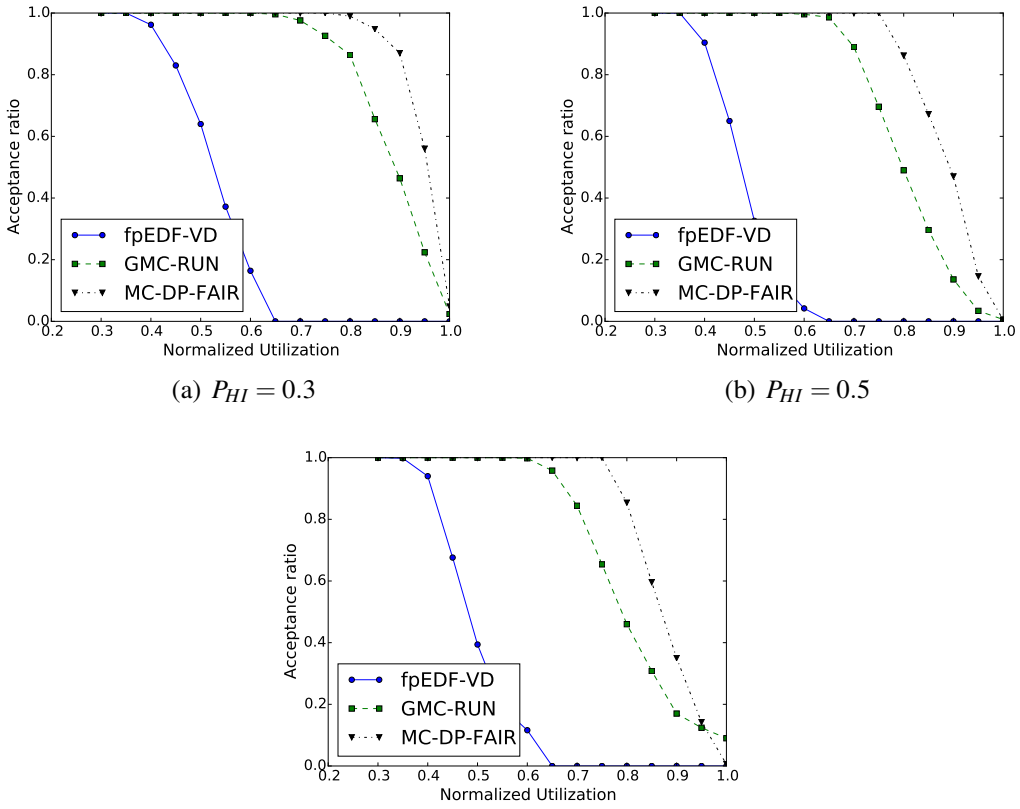
Figure 6.9: Acceptance ratio with normalized utilization (U_{Bound}/m) for different proportions of HI tasks on 2 processors.

GMC–RUN remains close to MC–DP–Fair on 2,4 and 8 processors. It presents as good performances as MC–DP–Fair up to a normalised utilisation of 0.7. Yet, GMC–RUN performs better than MC–DP–Fair in high utilisations for proportions of HI tasks equal to 0.7 for two and four processors.

The better performances of GMC–RUN and MC–DP–Fair compared to fpEDF–VD support our initial intuition that using the best multi–processor real–time scheduling algorithms to design a mixed–criticality one. Indeed, both algorithms, GMC–RUN and MC–DP–Fair, have designed based on an optimal multi–processor real–time scheduling algorithm.

Then the better results of GMC–RUN for high proportions of HI tasks in HI utilisation may seem surprising. But the explanation lies in task set samples properties. Higher proportions of HI tasks means that the task set utilisation is most likely equal to HI task utilisation alone. Hence, a lot of slack time is likely to be available in LO mode, meaning that modal servers have large budgets. Therefore, the allocation of LO tasks in modal

© 2017 Romain GRATIA



(a) $P_{HI} = 0.3$

(b) $P_{HI} = 0.5$

(c) $P_{HI} = 0.7$

Figure 6.10: Acceptance ratio with normalized utilization (U_{Bound}/m) for different proportions of HI tasks on 4 processors.

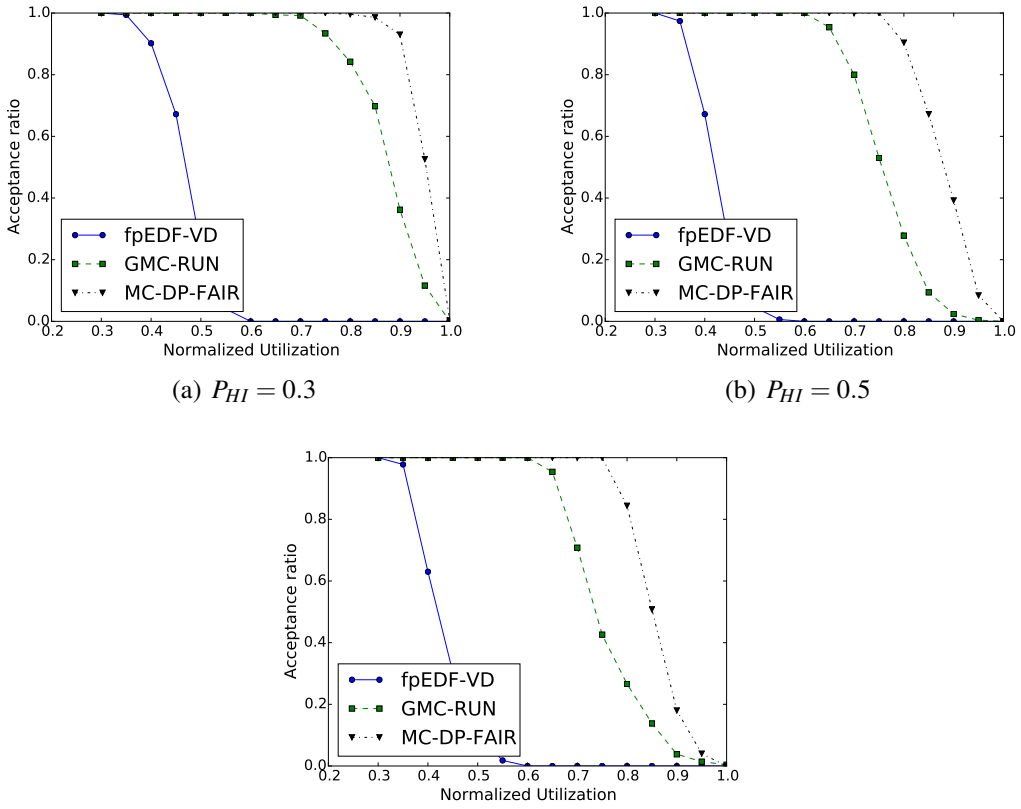
servers is more likely and thus reduces in larger proportion the overall utilisation of the system. For these cases, since the proportion of HI tasks is large, we manage with our approach to reduce task set utilisations to utilisation of the HI mode.

Then to have a better understanding of the influence of the proportion of HI tasks on the acceptance ratio, we compute the weighted acceptance ratio [77]. Let $R(U, P_{HI})$ be the acceptance ratio for a normalised utilisation U and a proportion of HI task P_{HI} and $\mathcal{U}_m$ be the set of normalised utilisations used for m processors with $m \in \{2,4,8\}$. Then the weighted acceptance ratio $A(P_{HI})$ is defined as follows:

$$A(P_{HI}) = \frac{\Sigma_{U \in \mathcal{U}_m} U \cdot R(U, P_{HI})}{\Sigma_{U \in \mathcal{U}_m} U} \tag{6.25}$$

The results are represented in figure 6.12. The common trend of these curbs is that the acceptance ratio first decreases until reaching proportion of HI tasks of 0.5/0.6 and a weighted acceptance ratio of 0.65 for 2 processors, 0.55 for 4 processors and 0.5 for

© 2017 Romain GRATIA



(a) $P_{HI} = 0.3$

(b) $P_{HI} = 0.5$

(c) $P_{HI} = 0.7$

Figure 6.11: Acceptance ratio with normalized utilization (U_{Bound}/m) for different proportions of HI tasks on 8 processors.

8 processors. Then acceptance ratio increases. These graphs also show that the fewer processors there are the larger the weighted acceptance ratio is. Note also that these trends hold for fpEDF–VD and MC–DP–Fair.

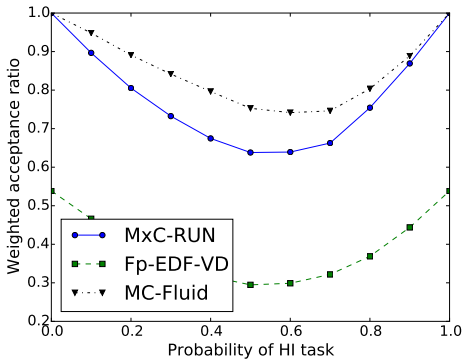
An explanation to this is that as the proportion of HI tasks increases before reaching 0.5, more HI tasks are to be scheduled. Still, they do not supply sufficient slack time to schedule a sufficient amount of LO task utilisation to sufficiently reduce the overall utilisation of the task set and make it schedulable. Then as HI tasks become the majority, slack time becomes more available allowing more LO tasks to be scheduled in it.

These experiments show that GMC–RUN has very satisfactory performances. It has schedulability efficiency comparable to MC–DP–Fair in average cases.

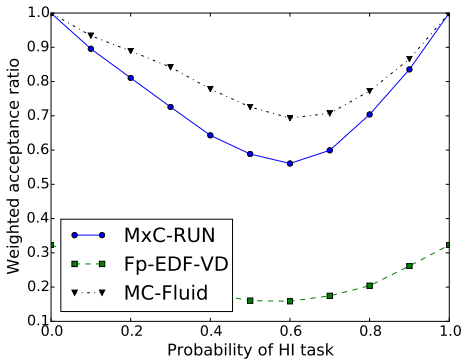
Practicality: number of preemptions entailed

The number of preemptions entailed by a scheduling algorithm impact the performances of a scheduling algorithm. The choice of RUN has been partly based on its ability to entail

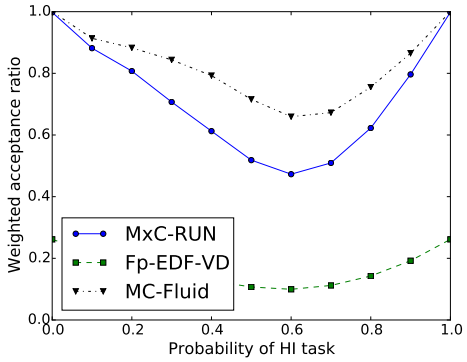
© 2017 Romain GRATIA



(a) For 2 processors



(b) For 4 processors



(c) For 8 processors

Figure 6.12: Weighted acceptance ratio with different values of HI task proportion for different number of processors.

few preemptions. But because modal servers have limited budgets their use can split the execution of their allocated tasks. Hence, it might generate preemptions that would not occur without the use of modal servers.

We propose to compare the average number of preemptions per job that GMC–RUN and MC–DP–Fair entail. We use the method described in RUN to count preemptions: per job, and ignoring those due to task activation and completion. We only count the number of preemptions in LO mode, as scheduling in HI mode is very similar to non–mixed–criticality case for both algorithms. The comparison for this mode will therefore yield the same results than those shown in [42], and that shown that RUN outperforms a version of DP–Fair scheduling algorithm. We filter out samples for which one of GMC–RUN or MC–DP–Fair cannot ensure schedulability on 2 processors. Results are pictured in figure 6.13.

The average number of preemptions is drawn with task numbers as abscissa. As expected, GMC–RUN entails at least five times fewer preemptions in average than MC–

© 2017 Romain GRATIA

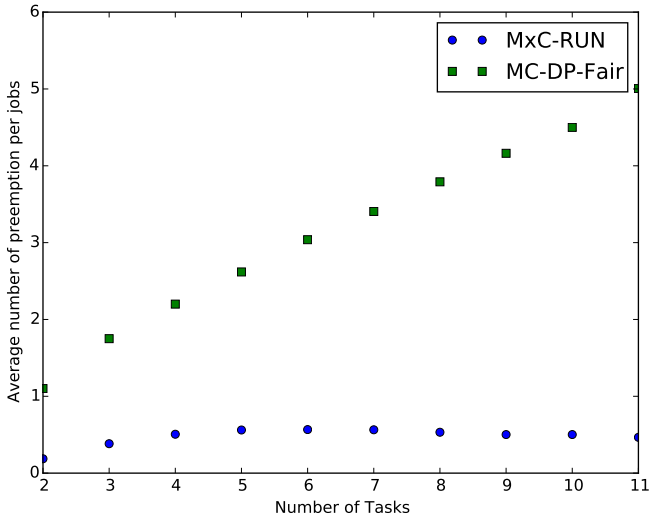


Figure 6.13: Average number of preemptions per job for task sets with different number of tasks

DP–Fair. Note that the gap is quite impressive for reasonable task set sizes and increases steadily. In this situation, GMC–RUN clearly dominates MC–DP–Fair.

6.5 Conclusion

In this chapter, we have achieved the design of our hierarchical scheduling framework with the choice of its top level scheduling algorithm. We based our choice on a list of performance criteria, that consider both theoretical and practical performance aspects of scheduling algorithms. We compared several possible multi–processor real–time scheduling algorithm against these criteria. It ultimately lead to the choice of the RUN algorithm. This choice led to name our hierarchical scheduling framework GMC–RUN.

We then described how the use of the different kinds of modal servers could be used with RUN. In particular, we described how the sequential condition of the aggregated modal servers could be respected through the use of RUN primal servers.

Next, we described the optimisation problem that represents the allocation of tasks in modal servers. The problem is to find the allocation with the largest utilisation of LO tasks allocated to modal servers. We explained why and how evolutionary algorithms are used to solve this optimisation problem.

Finally, we evaluate GMC–RUN both theoretically and experimentally. We first proved that GMC–RUN never requires more processors than the number of processors required by RUN when scheduling a mixed–criticality system as a multi–criticality one. Then, we

© 2017 Romain GRATIA

showed that GMC–RUN has the second best speed–up factor for mixed–criticality multi–processor scheduling algorithms, with a value of 2. We then measured the schedulability ratio on randomly generated task sets and compared GMC–RUN to fpEDF–VD and MC–DP–Fair, the two best mixed–criticality global scheduling algorithms. It results that GMC–RUN presents the second best performances and is closer to MC–DP–Fair, the best of the three algorithms, than to fpEDF–VD. At last, we computed the number of preemptions per job for GMC–RUN and MC–DP–Fair. On this criteria GMC–RUN outperforms MC–DP–Fair by entailing at least five times fewer preemptions.

In the next section, we describe our method to schedule mixed–criticality systems with more than two criticality levels with GMC–RUN. We notably present an iterative process that allow to correctly allocate tasks in modal servers for any number of criticality levels.

© 2017 Romain GRATIA

© 2017 Romain GRATIA

7 Extension to N criticality levels

Table of Contents

In previous chapters, we presented our mixed–criticality scheduling algorithm for systems with two criticality levels. Our algorithm is a hierarchical scheduling framework based on modal servers. These servers enforce a mixed–criticality scheduling policy that enables the scheduling of LO tasks in HI task slack time. It requires to allocate LO tasks to modal servers. This operation is performed by using an evolutionary algorithm.

Yet, in section 3.1.3 p 39 of the problem statement, we explained that the handling of only two criticality levels is not sufficient. The use of only two criticality levels aims at simplifying the mixed–criticality scheduling problem. But, industrial norms and standards define up to five criticality levels. Therefore, the handling of more than two criticality levels by mixed–criticality scheduling algorithms becomes a must. Moreover, handling more criticality levels enables system engineers to adjust more precisely the availability of the different tasks.

The generalisation to more than two criticality levels requires to handle more modes and consequently to ensure possibly more than one mode change. Thus it requires the coordinated scheduling of all modes so that mode changes are correctly handled. Recall in particular, that we have to ensure that we handle the disruption problem, exposed in section 3.2.2 p 41 of the problem statement, that can lead to missed deadlines after a mode change.

Besides, our algorithm is based on the use of the slack time of tasks. The introduction of more criticality levels results in potentially more tasks with available slack time. Moreover, the multiplication of modes makes the availability of the slack time, and thus its use, more complex compared to systems with two criticality levels. Indeed in dual–criticality systems, only the HI tasks have available slack time that is no longer available after a single mode change.

The multiplication of modes is first dealt with by the generalisation of the task model. Following this generalisation, we also have to adapt the definition of the modal server to ensure both all possible mode changes and the use of the slack time.

Since, there are more criticality levels and modes, the identification of which tasks can be executed in other task slack time is less obvious than for dual–criticality systems. We therefore need a clear rule to state whether the slack time of a task is available to execute another task. Hence, we have to revisit how we perform the allocation of tasks in modal servers. Since slack time availability depends on the active mode, the tasks with available slack time and those without can be defined mode by mode. Therefore, for each mode we have the same situation than for dual–criticality systems where tasks can be into two groups: those with slack time and those without. The idea is hence to perform the

© 2017 Romain GRATIA

allocation as with dual–criticality systems by considering each mode at a time through an inductive process.

In this chapter, we first describe the extension of the model and the notations of the mixed–criticality task. Next, we explicit the sources, the amount and the availability of the slack time in systems with more than two criticality levels. Then, we present how our modal servers can handle task set with tasks belonging to more than two criticality levels. Finally, we justify why our hierarchical scheduling framework yields a correct scheduling of mixed–criticality systems with any number of criticality levels. We then explain how our evolutionary algorithm is reused in an inductive process to perform the allocation of tasks in modal servers. Finally, we experimentally assess the schedulability performances of our scheduling algorithm by measuring the schedulability ratio for task sets scheduled with either three or two criticality levels.

7.1 Generalisation of the system model

In this section, we present the notations used for mixed–criticality tasks classified in more than two criticality levels. We then determine the amount of available slack time for such tasks per mode and state how it can be safely used.

7.1.1 Generalisation of the task model

This section deals with the description of the generalisation of the model of the mixed–criticality periodic task with implicit deadlines to any number of criticality levels. The notations used until now to indicate the criticality level of the tasks is not practical beyond two criticality levels. Since we aim at handling systems with more than two criticality levels, we need a different notation.

A practical and straightforward notation for denoting criticality levels is to use numbers. For system with N criticality levels, we use numbers from 1 to N and note $\mathcal{CL} = \{1,2,...,N\}$ the set of criticality levels. N represents the number of criticality levels as well as the highest possible criticality level.

In such mixed–criticality system, a task set Γ contains n tasks $\tau_1,...,\tau_n$ with each task $\tau_i \in \Gamma$ characterised by $(T_i,\chi_i,C_i(1),C_i(2),...,C_i(N))$. These tasks are synchronously started at time $t = 0$. T_i is still task τ_i period and its implicit deadline in all modes.

χ_i denotes τ_i criticality level in $\mathcal{CL}$. A task of criticality level χ_i is referred to as a CL–χ_i task or noted $\tau_i\{\chi = \chi_i\}$. Let τ_i and τ_j be two tasks of criticality level χ_i and χ_j respectively. If $\chi_i > \chi_j$ then τ_i is more critical than τ_j.

© 2017 Romain GRATIA

$C_i(1),...,C_i(N)$ are the budgets of τ_i for criticality levels 1 to N, respectively. We assume that $\forall\chi \in \mathcal{CL}$, $C_i(\chi) \leq T_i$. Moreover $\forall\chi$, $\chi_i < \chi$ implies $C_i(\chi) = C_i(\chi_i)$, and for any χ and χ' in $\mathcal{CL}$, $\chi \leq \chi' \leq \chi_i$ implies $C_i(\chi) \leq C_i(\chi') \leq C_i(\chi_i)$. We assume that all jobs of CL–N tasks always complete their executions in no more than $C_i(N)$. From the period and the budget of a task τ_i, we can derive its utilisation, $U_i = \frac{C_i}{T_i}$.

As for budgets there are as many task utilisations and Demand Bound Functions, defined in definition 22 in section 5.3.3 p 72, as there are criticality levels. Hence, for a criticality level $\chi \in \mathcal{CL}$, $U_i(\chi) = \frac{C_i(\chi)}{T_i}$ and $DBF_i(t,\chi) = \lfloor \frac{t}{T_i} \rfloor \cdot C_i(\chi)$.

The subset of tasks in Γ with criticality level χ is denoted: $\Gamma(\chi) = \{\tau_i \in \Gamma |\ \chi_i = \chi\}$. Similarly, $\Gamma(< \chi)$, $\Gamma(\leq \chi)$, $\Gamma(\geq \chi)$ $\Gamma(> \chi)$ denote subsets of Γ such that tasks are of criticality levels smaller, smaller or equal, greater or equal, and greater than χ, respectively. Besides, we define the maximal criticality level of a task set as follows:

Definition 31 (Maximal criticality level of a task set)**.** *The maximal criticality level of a task set* Γ *is defined as:* $\max_{\tau_i \in \Gamma} \chi_i$

We finally use the following notation Γ^M to designate a task set of maximal criticality level M.

Utilisation definition can be extended to a task set Γ: $U_\Gamma = \sum_{\tau_i \in \Gamma} U_i$. It can be generalised to define the utilisation of the task set Γ at level χ as: $U_\Gamma(\chi) = \sum_{\tau_i \in \Gamma} U_i(\chi)$. Finally, we note $U_{\Gamma(\chi_1)}(\chi_2)$ the utilisation of tasks of criticality level χ_1 in Γ at criticality level χ_2.

We define what an execution mode is with this model of mixed–criticality systems.

Definition 32 (Execution mode)**.** *An execution mode is a configuration of the scheduler characterised by the following elements;*

- *A criticality level* $\chi \in \mathcal{CL}$
- *Task set* $\Gamma(\geq \chi)$ *whose tasks are scheduled using their budgets C(*χ*)*

Execution mode corresponding to the criticality level χ, is referred to as mode χ. A mode χ is said lower (respectively, higher) than an another mode χ' if χ is smaller (respectively, larger) than χ'. N execution modes are defined for a system with N criticality levels. A mixed–criticality system can be executed in any of the N modes. Hence a mode define the scheduling objectives to reach for each task and requires mechanisms to enforce a mode change.

We consider the system starts executing in mode 1, and changes its mode whenever a Timing Failure Event occurs (TFE).

© 2017 Romain GRATIA

Definition 33 (Timing Failure Event)**.** *A Timing Failure Event is the instant at which a task job has been executed for as long as its C(χ) without completing its execution, while the system is in mode χ.*

If a TFE occurs while the system is in mode χ, then a mode change is performed from mode χ to mode $\chi+1$. CL–χ Tasks $\tau_i \in \Gamma(\chi)$ are no longer executed. A job of τ_i respects its timing parameters of level χ if its execution time is lower than $C_i(\chi)$ and thus does not trigger a TFE. Mode χ is said active as long as all jobs respect their timing parameters of criticality level χ. The system current execution mode is the mode with the lowest criticality level that has not suffered a TFE.

7.1.2 Determination of the slack time per mode

The task model being generalised to handle more criticality levels, we have to determine the available slack time before performing the allocation of tasks. It consists in checking whether the slack is still a periodic resource, quantifying the slack time and determining which tasks can be scheduled in that slack time.

Although the task model has been generalised, the periodic activation of the tasks remains. Hence, the slack time is still a periodic resource (see section 5.1.2 p 63).

In dual–criticality systems, slack time originates in LO mode from the difference between budgets of HI and LO modes as HI tasks are scheduled with their HI mode budgets. Following similar reasoning for systems with more than two criticality levels, we execute all tasks with their budgets at their criticality levels. Now that tasks have potentially more than two budgets, there are several intervals of slack time: one for each difference between budgets of two consecutive modes as pictured in figure 7.1. Each interval of slack time can be associated to a mode.

Definition 34 (Interval of slack time)**.** *An interval of slack time of mode M, or simply slack time of mode M, is a portion of the slack time of a task that is available until mode M becomes active.*

Recall the availability of slack time has been defined in definition 15 page 64. For example, the slack time of mode 3 is equal to the difference between the budgets of mode 3 and 2: $C(3)-C(2)$. Hence, a CL–N task executed with its budget of mode N, C(N), has $N-1$ intervals of slack time, as shown in figure 7.1.

After a TFE, several intervals of slack time of a task that can still be available. The availability of the slack time depends on the value of its mode and of the criticality level of the active mode. When mode M becomes active, the slack time of mode M of a CL–N task,

© 2017 Romain GRATIA

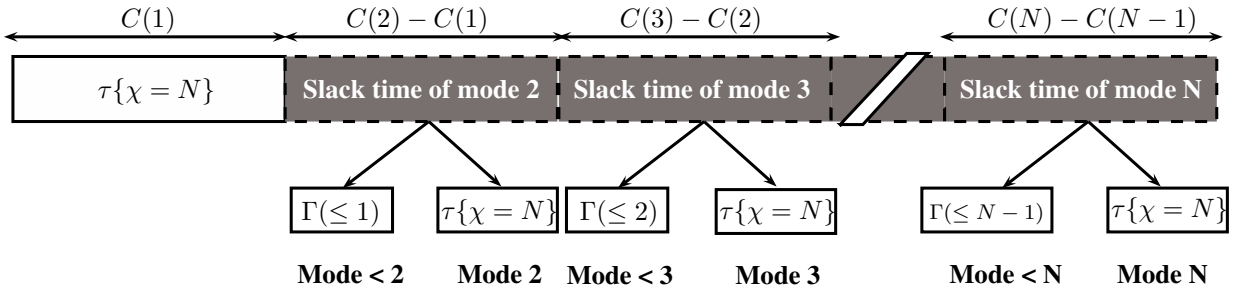


Figure 7.1: Intervals of the slack time of a CL–N task

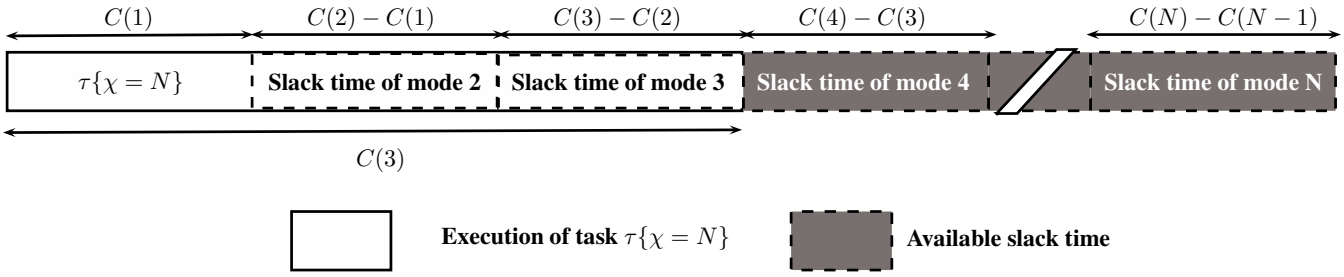


Figure 7.2: Availability of the intervals of slack time of a CL–N task

with N > M, is used to complete the CL–N task. But only intervals of slack time of modes M and lower are not available and used to execute the CL–N task in mode M. Meaning that the intervals of slack time from higher modes are still available. To illustrate this, observe figure 7.2. In this figure, the active mode is mode 3, as the CL–N task executes for as long as its budget of mode 3. This budget can be decomposed as the budget of mode 1 completed by the slack time of modes 2 and 3. These two intervals of slack time are hence no longer left unused in mode 3. But slack time of higher modes, from 4 to N, are still left unused. Therefore, they can be used to schedule other tasks.

As for dual–criticality systems, a set of tasks can be executed in the slack time of a task τ_i as long as this slack time is not used to complete task τ_i. These systems are constituted with a single criticality level of tasks providing slack time and a single criticality level of tasks schedulable in that slack time. In systems with more criticality levels, the number of criticality levels and of intervals of slack time of different modes are larger. Besides tasks can be scheduled in the slack time of all tasks. This depends on the modes of the slack time and of the criticality levels of tasks.

But we have to ensure that the interval of slack time of a task τ_i is never used to simultaneously complete the task τ_i and execute a set of other tasks. To avoid that situation, we have to respect the following theorem:

© 2017 Romain GRATIA

Theorem 12. *Let* Γ *be set of tasks of maximum criticality level K. Consider an interval of slack time of mode M of a CL–N task. If* $K < M$ *then the interval of slack time is available for the tasks in* Γ.

Proof. We prove this theorem by contradiction. Assume a CL–N task τ has a slack time of mode M and that a set of tasks Γ is composed of tasks with a maximum criticality levels equal to $M-1$. Still when a mode K becomes active that slack time has to schedule simultaneously τ and the set of tasks Γ.

If K is such that $K \geq M$. The interval of slack time completes task τ when mode M or higher is active. But when mode M becomes active none of the tasks in Γ is scheduled as they are all of criticality lower than M. Hence, in such case, task τ and tasks in Γ are not scheduled simultaneously in the slack time.

If K is such that $K < M$. The interval of slack time schedules the task set Γ when modes lower than M are active. Task τ executes for as long as a budget of mode lower than M and does not need the slack time of mode M.

This proves the theorem.

□

Now that we know when and how much slack time is available and the criticality levels of the tasks that can be scheduled in it, we extend the definition of modal servers.

Task	Period	Criticality	C(1)	C(2)	C(3)
τ_1	50	3	16	16	25.6
τ_2	50	1	17.28	17.28	17.28
τ_3	100	3	12.8	13.12	18.56
τ_4	50	1	23.04	23.04	23.04
τ_5	500	2	9.6	9.92	9.92
τ_6	50	3	2.56	10.88	31.36
τ_7	1000	1	13.12	13.12	13.12
τ_8	100	1	7.36	7.36	7.36
τ_9	100	1	16.32	16.32	16.32
τ_{10}	500	1	28.48	28.48	28.48
τ_{11}	50	2	14.08	35.84	35.84
τ_{12}	50	2	3.84	18.88	18.88
τ_{13}	100	2	72.64	74.24	74.24
τ_{14}	100	1	23.68	23.68	23.68

Table 7.1: Example of a set of tasks with three criticality levels.

© 2017 Romain GRATIA

7.2 Generalisation of the modal server

As for dual–criticality systems, modal servers are used to take advantage of the slack time produced by tasks to schedule other tasks. But, in systems with more than two criticality levels, we saw that tasks can have more than one interval of available slack time. Yet, modal servers have been defined to only handle one such interval of slack time. In this section, We extend here the definitions of modal servers, slackful modal servers, slackless modal servers and aggregated modal servers. Besides, to exemplify our definitions, we use the task set described in table 7.1. This task set is derived from tasks provided by our industrial partners of the ELA project. It is composed of tasks with three criticality levels and their timing parameters can be considered as representative of what can be encountered in the automotive industry.

7.2.1 General definition of a modal server

Recall, that in dual–criticality systems, the slack time of modal servers originated of a HI tasks, that were called providing tasks. We reuse this concept of providing tasks to generalise the definition of modal servers to systems with more than two criticality levels:

Definition 35. *In a system with N criticality levels, a modal server MS is characterised by a budget C_{MS}, a period P_{MS}, a set of providing tasks $\mathcal{P}_{MS}$ and N-1 task sets noted Γ^1_{MS},...,Γ^{N-1}_{MS}. We note M the maximal criticality level of $\mathcal{P}_{MS}$. Each task set Γ^1_{MS},...,Γ^{M-1}_{MS} are such that task set Γ^x, with $x \in [1,..,M-1]$, contains only tasks of criticality level lower or equal to x. The remaining task sets Γ^M_{MS},...,Γ^{N-1}_{MS} are left empty.*

A modal server periodically executes for C_{MS} time units every P_{MS} time units. During its execution it schedules its task sets as follows:

1. *If active mode K is such that $K < M$, schedule first tasks in task set $\mathcal{P}_{MS}$ until completion of all its tasks. Then $\forall x \in [K,...,M-1]$ schedule following a uniprocessor scheduling policy all task set Γ^x_{MS} in the increasing order of x for a duration equal to the interval of slack of time of mode x+1.*

2. *If $K = M$, schedule only task set $\mathcal{P}_{MS}$, following a uniprocessor scheduling policy.*

3. *If active mode K is such that $K > M$, schedule nothing.*

A modal server has still a periodic execution, since the slack time is still a periodic resource. The uniprocessor scheduling policy used is Earliest Deadline First. With, this definition of the modal server respects theorem 12 p 121, that gives a rule that ensures

© 2017 Romain GRATIA

the slack time is never used to simultaneously execute the providing tasks and other tasks. Its set Γ^1_{MS}, containing only CL–1 tasks, is scheduled in interval of slack time of mode 2. CL–2 and CL–1 tasks contained in Γ^2_{MS} are scheduled in interval of slack time of mode 3... And when mode M becomes active, only providing tasks in $\mathcal{P}_{MS}$ are scheduled, that is tasks of criticality level equal to M.

Since we have generalise the definition of the slack time and of the modal server, we now have to generalise the definitions of the slackful modal servers, slackless modal servers and aggregated modal servers.

Definition of a slackful modal server

A slackful modal server is defined as follows:

Definition 36. *A slackful modal server is a modal server MS whose providing task set $\mathcal{P}_{MS}$ contains a single task.*

For the same reasons than those exposed in section 5.3.1 p 67, a slackful modal server takes the same timing parameters than those of its providing task. A slackful modal server is represented in figure 7.3.

The correctness of the scheduling in these slackful modal servers is ensured for the same reasons than those exposed in section 5.3.2 p 69. We assume that the modal server is correctly scheduled. The modal server has the same timing parameters than those of the CL–M task. Besides this task is completed before scheduling the other task sets. It hence ensures that the CL–M task is correctly scheduled. The correctness of the scheduling of tasks in the other task sets of the modal servers is proved by using the schedulability tests presented in theorems 4 and 5, that can be found at pages 71 and 72, respectively.

Consider the task set described in table 7.1. We can form two slackful modal servers from tasks τ_{11} and τ_6. τ_{11} is of criticality level 2 and is added to the task set Γ^2 of its slackful modal server, noted MS_{11}. τ_6 is of criticality level 3 and is added to the task set Γ^3 of its slackful modal server, noted MS_6. τ_{11} has a slack time of mode 2 with a budget of 21.76. τ_6 has a slack time of mode 2 with a budget of 8.32 and a slack time of mode 3 of budget 20.48. We can allocate task τ_2 to the slackful modal server MS_{11} and more precisely in its slack time of mode 2. Similarly, we can allocate task τ_8 in the slack time of mode 2 and tasks τ_5 and τ_{12} in the slack time of mode 3 of the slackful modal server MS_6.

Definition of a slackless modal server

A slackless modal server is defined as follows:

© 2017 Romain GRATIA

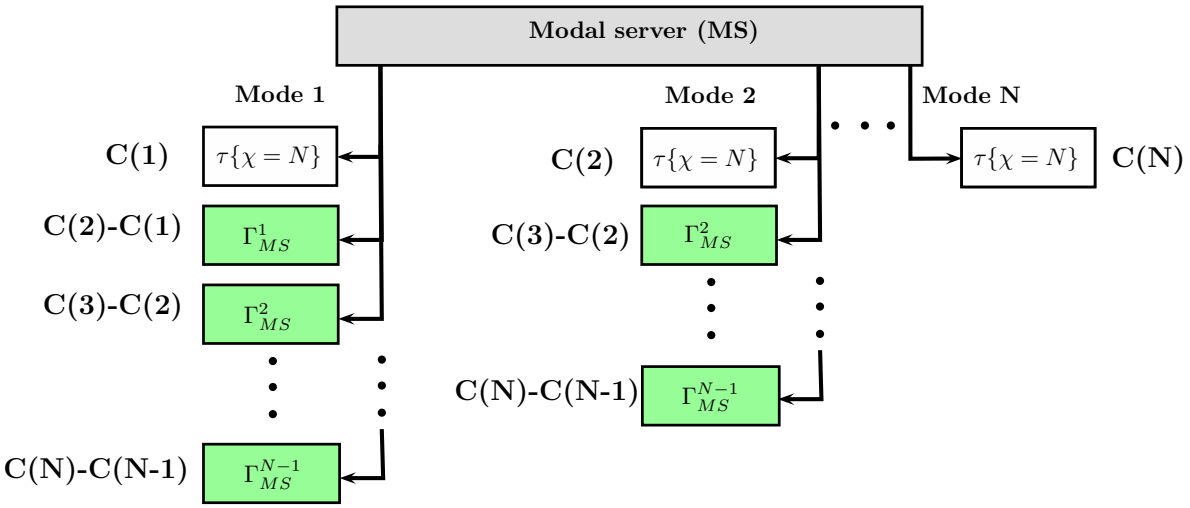


Figure 7.3: A slackful modal server with its task sets to execute in each mode with the given budgets

Definition 37. *A slackless modal server is a modal server MS whose task set $\mathcal{P}_{MS}$ has a single task and all its task sets Γ^1_{MS},...,Γ^{N-1}_{MS} are empty.*

Identically, the timing parameters of such modal server are those of the task at its criticality level.

Such modal server schedules its non empty task set until the active mode of the system is larger than the criticality level of the task and then nothing.

When considering the task set described in table 7.1, task τ_4 can not be allocated to any slackful modal server, hence it is scheduled in a slackless modal server.

Definition of an aggregated modal server

An aggregated modal server is defined as follows:

Definition 38. *An aggregated modal server AMS is a set of slackful modal servers MS_i such that these slackful modal servers are executed sequentially and one or several of their task sets $\Gamma^1_{MS_i}$,...,$\Gamma^{N-1}_{MS_i}$ contains the same tasks.*

The use of aggregated modal servers requires to respect the sequential condition presented in section 5.5.1 p 74.

When considering the task set described in table 7.1, an aggregation of slackful modal servers can be formed with the providing tasks τ_3 and τ_{12} to schedule task τ_{10} and τ_{14}. τ_{10} and τ_{14} being executed in the slack time of mode 2 for both providing tasks.

© 2017 Romain GRATIA

7.2.2 Scheduling generalised modal servers with GMC–RUN

In chapters 5 and 6, we explained that the peculiarities of the mixed–criticality scheduling were contained in modal servers. Besides, as model servers are periodic servers, they are schedulable as regular periodic tasks with implicit deadlines. Thus, modal servers can be scheduled by a multi–processor real–time scheduling algorithm.

Our extension of the modal server aimed at taking into account the changes brought by the use of more criticality levels. This affected the number of task sets in modal servers and the scheduling policy enforced in modal servers. Yet, it did not affect its periodic execution similar to that of a regular periodic server or task. Hence, mixed–criticality peculiarities are still handled in modal servers, they are thus still schedulable by a multi–processor real–time scheduling algorithm.

Assuming that the allocation of tasks in the different modal servers has been performed. It resulted in a set of modal servers noted Γ_{MS} of utilisation U_{MS} and a set of constraints $\mathcal{C}$ due to the potential formation of aggregated modal servers. Then this set of modal servers is schedulable by GMC–RUN if the conditions presented in theorem 9, page 88, are respected. That is if U_{MS} is lower or equal to the number of processors and the set constraints $\mathcal{C}$ respect the rules of the RUN PACK operation.

But it assumes that we have performed the allocation of tasks in modal servers. Our method described in section 6.3 p 89 was designed to handle dual–criticality systems not systems with more criticality levels. We describe in the following section, how we perform this allocation for mixed–criticality systems with more than two criticality levels.

7.3 Inductive allocation

This section describes how the allocation of tasks in modal servers is performed for mixed–criticality systems with more than two criticality levels. We explain how an inductive process allows to perform the allocation for such systems. This inductive process consists first in splitting the set of tasks into two groups based on the constraints imposed by theorem 12 p 121, that gives a rule that ensures the slack time is never used to simultaneously execute the providing tasks and other tasks.. These two groups can then be handled as a task set with two criticality levels to perform the allocation. These operations are repeated as many times as there are slack time of different modes, that is $N-1$ times for a system with N criticality levels.

To exemplify our approach we use the task set described in table 7.1. But, for the sake of clarity, we only consider tasks τ_4, τ_6, τ_8, τ_{10}, τ_{12} and τ_{13} of this task set. We represented

© 2017 Romain GRATIA

these tasks in figure 7.4, classified in criticality levels and with the available intervals of slack time in each mode.

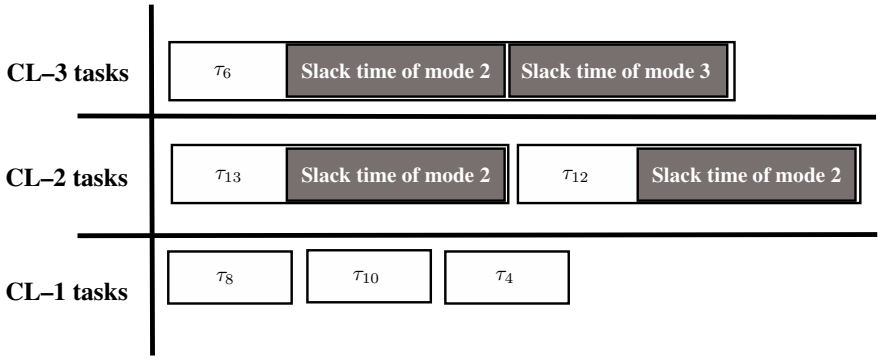


Figure 7.4: Representation of the considered tasks with their slack time of different modes

For dual–criticality systems, the allocation of tasks in modal servers is achieved by splitting the task set into two groups. A first group is constituted of tasks with available slack time, i.e the HI tasks. These tasks are used as providing tasks to form slackful modal servers. The second group is composed of tasks that have no available slack time. These tasks can hence be scheduled in slack time of the tasks of the first group, i.e the LO tasks. Tasks in the second group are allocated to slackful modal servers and aggregated modal servers, when they are schedulable. Otherwise, they are allocated to slackless modal servers. The result is a set of modal servers that are schedulable by a non–mixed–criticality scheduling algorithm such as RUN. The allocation in modal servers has the effect of reducing the number of criticality levels from two to one, i.e to a system schedulable by a real–time scheduling algorithm.

Following the dual–criticality system case, we start the allocation for a system with N criticality levels by splitting the task sets into two groups. The first group contains all tasks that have an interval of available slack time of mode K. The second group is composed of tasks that have no available slack time of mode K. That is tasks that have a criticality level strictly lower than K. The inductive process consists in considering all the intervals of slack time in the increasing order of modes.

Hence, the first iteration of the inductive process considers slack time of mode 2. The first group is hence composed of all tasks with a criticality level greater than 1 since they all have an interval of slack time of mode 2. The second group is constituted of tasks that can be allocated to the slack time of mode 2 according to theorem 12 p 121, i.e the CL–1 tasks.

The allocation is performed on these two groups as for dual–criticality systems. The result is a set of CL–1 tasks allocated to a set of slackful modal servers. These slackful

© 2017 Romain GRATIA

modal servers are formed with providing tasks of any higher criticality levels. CL–1 tasks are added to sets Γ^1 of slackful modal servers. A set of CL–1 tasks can potentially remain unallocated.

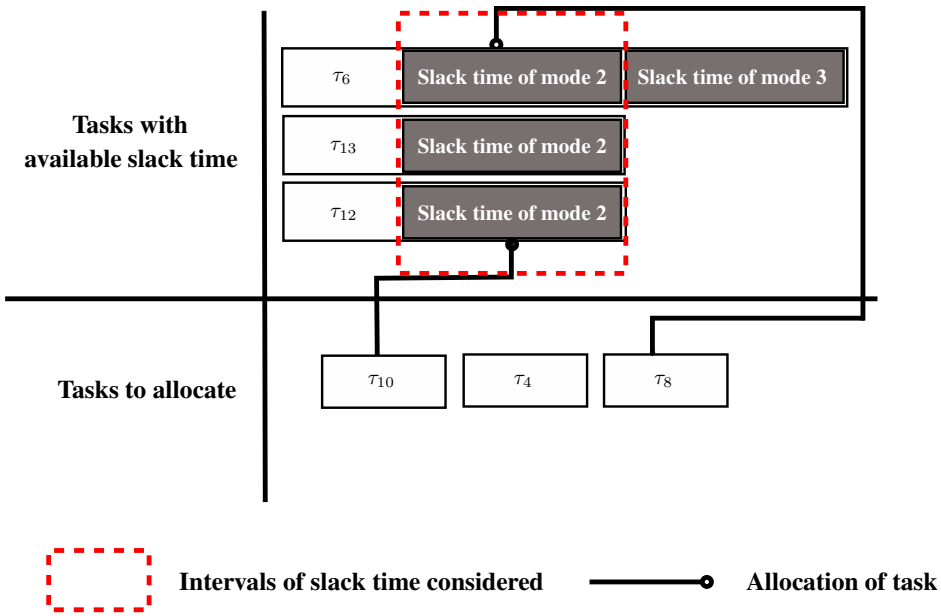


Figure 7.5: Representation of the allocation of system with three criticality levels: first iteration allocation in slice of slack time of mode 2

The splitting and the allocation of the tasks of our example are represented in figure 7.5. We seek to allocate CL–1 tasks, τ_4, τ_8 and τ_{10} into the slack time of mode 2 of tasks of higher criticality levels τ_6, τ_{12} and τ_{13}. We split these tasks in two groups and manage to allocate task τ_{10} into task τ_{12} slack time and task τ_8 into the slack time of task τ_6. This shows that our approach allows to use the slack time not only of CL–2 tasks to schedule CL–1 tasks but also slack time of tasks of higher criticality levels. The result of the first iteration of our approach for these tasks is pictured in figure 7.6.

With this first iteration, we have performed the allocation in the slack time of mode 2. The next iteration is to consider the slack time of mode 3. We repeat the same operations. Tasks are split into two groups, one group contains the tasks with available slack time of mode 3. The other group is composed of tasks that can be allocated to this slack time. This second group is composed of CL–2 tasks and also includes the set of unallocated CL–1 tasks originating from the first iteration of the allocation. This second iteration results in the allocation of CL–2 tasks into task sets Γ^2 of slackful modal servers and potentially in a set of unallocated CL–2 and CL–1 tasks.

Note that during this second iteration, the formation of aggregated modal servers has to take into account the potential aggregated modal servers from the first iteration. Indeed,

© 2017 Romain GRATIA

it may happen that during the first iteration two slackful modal servers MS_α and MS_β form an aggregated modal server. Then, during the second iteration, one the two modal servers MS_α form another aggregated modal with a different slackful modal server MS_γ. This aggregated modal server is correct not only if MS_α and MS_γ respect the sequentiality condition, that is if they can be scheduled sequentially. But it is correct only if MS_α, and MS_β, and MS_γ respect the sequentiality condition. Therefore, during an iteration of the inductive process the formation of aggregated modal servers must take into account the potential aggregated modal servers from previous iterations.

Note also that some CL–2 tasks may be providing tasks of slackful modal servers. Hence, these slackful modal servers may have CL–1 tasks in their sets Γ^1. These same CL–2 tasks may then be allocated to other slackful modal servers to be scheduled in an interval of slack time of mode 3, or higher. Thus, a slackful modal server can be allocated to another slackful modal server. It still yields a correct scheduling as long as the conditions of theorem 1 presented in section 5.1.1, p 63, are respected. This theorem states that a set of periodic tasks with implicit deadlines is schedulable by a server if the server is correctly scheduled and tasks passed a schedulability test. Since, modal servers can be scheduled as regular periodic tasks, this theorem also applies to allocated slackful modal servers. Hence, if the slackful modal servers are correctly scheduled and its allocated slackful modal servers respect a schedulability tests, then the allocated slackful modal servers are schedulable. Then, tasks allocated to allocated slackful modal servers stay schedulable since the slackful modal servers are correctly scheduled and the tasks has passed a schedulability test. As a slackful modal server and its providing task have the same timing parameters, we can indistinctly speak of the allocation of a slackful modal server or of the corresponding providing task.

This possibility happens in our example as shown in figure 7.7. During this second iteration, we considered the allocation of CL–2 tasks τ_{12} and τ_{13} and of the CL–1 task τ_4, that remained unallocated from the first iteration. Task τ_{12} is allocated to the slack of time of mode 3 of task τ_6, while τ_{12} execute in its slack time of mode 2 task τ_{10}. We have hence the allocation of a slackful modal server into another slackful modal server.

Once, the second iteration is achieved, we perform the next iteration. During the K^{th} iteration, with $K < N-1$, the two groups are constituted as follows:

- A first group of tasks with available slack time of mode $K+1$, that corresponds to tasks of criticality level greater or equal to $K+1$.

© 2017 Romain GRATIA

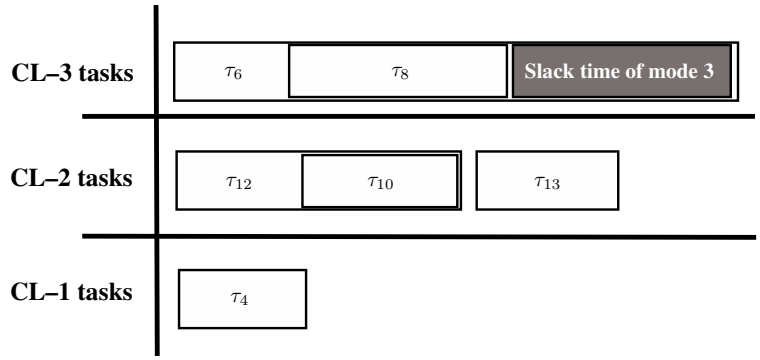


Figure 7.6: Result of the considered task set after the first allocation

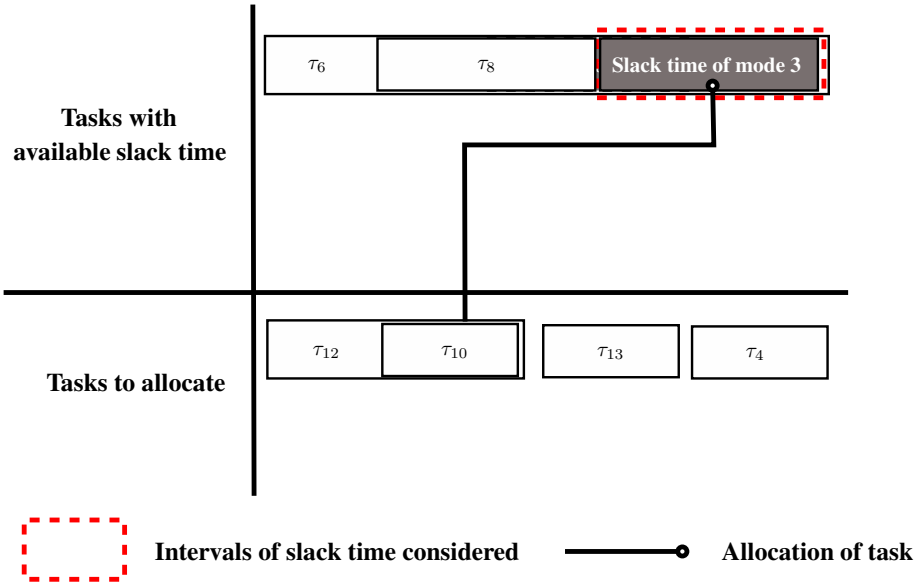


Figure 7.7: Representation of the allocation of system with three criticality levels: second iteration allocation in slice of slack time of mode 3

- A second group constituted of tasks of criticality level equal to K and potentially of all unallocated tasks of criticality levels lower than K originating from all previous iterations.

The tasks in the second groups are to be allocated in the slack time of mode $K+1$ of the tasks of the first group. They will be added to the sets Γ^K of the corresponding slackful modal servers.

Once all the iterations of the inductive process have been performed, the remaining unallocated tasks are each scheduled in its own slackless modal server. The resulting set of slackful modal servers and slackless modal servers can then be scheduled if it respects

© 2017 Romain GRATIA

the condition of theorem 9 p 63 that gives the conditions for a set of modal servers to be schedulable by GMC–RUN.

Providing task	Allocation in slack time of mode 2	Allocation in slack time of mode 3
τ_1	$\emptyset$	τ_9
τ_3	τ_{10},τ_{14}	τ_7
τ_4	•	•
τ_6	τ_8	τ_5, τ_{12}
τ_{11}	τ_2	•
τ_{12}	τ_{10},τ_{14}	•
τ_{13}	$\emptyset$	•

Table 7.2: Result of the allocation of the task set described in table 7.1.

The final result of the allocation of the whole task set described in table 7.1 is presented in table 7.2. In table 7.2, tasks in column "Providing task" are the tasks scheduled either in slackful modal servers or in slackless modal servers at the end of the allocation process. The tasks appearing in the other two columns are those scheduled in the slack time of other tasks. In these same columns, the "•" means that the task has no available slack time in that mode. The symbol "$\emptyset$" means that the task has slack time but no tasks has been allocated. Thanks to our approach, we are able to reduce the overall utilisation from 4.53 to 3.24. We can hence schedule this task set on four processors instead of five.

7.4 Experimental assessment

The generalisation of our approach to systems with more than two criticality levels offers system engineers the possibility to adjust more precisely the availability of tasks. Indeed, by adding more criticality levels, they can more finely classify its tasks in the different criticality levels. In this section, we want to assess the impact on the schedulability performances of the use of more than two criticality levels. To this end we measure and compare the schedulability ratio of GMC–RUN using the same task sets but in different configurations with two or three criticality levels. We first describe how these task sets are generated.

Considered scenario

We consider the case where system engineers have tasks classified in three criticality levels. For each task they dispose of as many budgets value as the task criticality level. But

© 2017 Romain GRATIA

since they dispose of only mixed–criticality scheduling algorithm that handled two criticality levels, they have to convert their three–criticality–level system into a dual–criticality one. A first mode is used to schedule all tasks. The second one is used to schedule only the most critical tasks that is those of criticality level 3.

In the second mode, CL–3 tasks are executed with their safest budgets C(3).

For the first mode, system engineers have several possible configurations. CL–1 tasks have a single possible budget C(1). CL–2 tasks can be executed with their budgets of their criticality levels C(2) or of criticality level 1, C(1). The use of C(2) can decrease the likelihood of a CL–2 task triggering a TFE, it is hence more conservative. The use of C(1) requires fewer processing power, it is hence more optimistic. For CL–3 tasks, they can also decide between their budgets C(1) or C(2). The use of C(2) provides the same advantage than for CL–2 tasks. The use of the budget C(1) results in more available slack time. We hence consider the following configurations with two criticality levels:

- CL–1 tasks with C(1), CL–2 tasks with C(1) and CL–3 tasks with C(1) and C(3). As it uses the optimistic (Opt) budgets for CL–2 and CL–3 tasks, it is called configuration Opt2Opt3.
- CL–1 tasks with C(1), CL–2 tasks with C(2) and CL–3 tasks with C(1) and C(3). As it uses the conservative (Cons) budget for CL–2 and the optimistic (Opt) one for CL–3 tasks, it is called configuration Cons2Opt3.
- CL–1 tasks with C(1), CL–2 tasks with C(2) and CL–3 tasks with C(2) and C(3). As it uses the conservative (Cons) budgets for CL–2 and CL–3 tasks, it is called configuration Cons2Cons3.

But the use of these configurations with two criticality levels has an impact on the availability of tasks. We compare the guarantees of availability of the configurations with two criticality levels with those of the configuration with three criticality levels in table 7.3. A $\downarrow$ means that the availability of tasks is degraded and a $\searrow$ means it is slightly degraded. A $\longrightarrow$ means that the availability in not impacted. A $\uparrow$ means that the availability is highly improved and a $\nearrow$ means it is slightly improved.

The impact on the availability of tasks in table 7.3 is assessed by considering the likelihood that the use of a budget triggers a TFE. A TFE is likelier with the use of budgets C(1) than with budgets C(2) or even more than with budgets C(3). Similarly, a TFE is likelier with the use of budgets C(2) than with budgets C(3). Recall also that a TFE is supposed not to be possible with budgets C(3).

First consider the CL–1 tasks. These tasks are in configuration Opt2Opt3 as likely to be stopped as in the system with three criticality levels. Indeed, in these configurations,

© 2017 Romain GRATIA

	CL–1 availability	CL–2 availability	CL–3 availability
Configuration Opt2Opt3	$\longrightarrow$	$\downarrow$	$\longrightarrow$
Configuration Cons2Opt3	$\nearrow$	$\downarrow$	$\longrightarrow$
Configuration Cons2Cons3	$\uparrow$	$\searrow$	$\longrightarrow$

Table 7.3: Impact on the availability of tasks in the different configurations compared to the three criticality level configuration

these tasks are dropped as soon as a C(1) is exceeded. In the other two configurations, Cons2Opt3 and Cons2Cons3, CL–1 task availability is improved, as the likelihood that a TFE is triggered by a CL–2 or CL–3 task is lowered. Indeed, in configuration Cons2Opt3, CL–2 tasks trigger a TFE if they exceed their budgets C(2) and not C(1). And for configuration Cons2Cons3, CL–1 tasks are stopped if CL–2 or CL–3 task triggers a TFE if they exceed their budgets C(2) and not C(1).

Now consider CL–2 tasks. In all configurations these tasks see their availability degraded as they can be stopped as soon as a CL–1 task exceed its C(1) budget. While, in the configuration with three criticality levels, these tasks are dropped only if a CL–2 or a CL–3 task exceeds its budget C(2). Availability is a bit better in configuration Cons2Cons3, since CL–3 tasks are executed with their budgets C(2). We hence see that all the configurations with two criticality levels degrade the availability of CL–2 tasks.

The availability of CL–3 tasks is unaffected in all configurations since their execution is always ensured by the mode 2 of the configurations with two criticality levels.

We now compare the schedulability performances of these three possible configurations with two criticality levels with those of the the configuration with three criticality levels, called configuration 3Crit.

Task set generation

The generation of the task sets is performed with the objective to evaluate the impact of the use of three criticality levels on the schedulability performances. To that aim, we need representative task set samples to estimate average performances for various combinations of task parameters with three criticality levels. To generate our task sets, we extend the method described in [53] used for generating dual–criticality task sets. With this extended method, the maximal utilisation of modes U_{Bound} and the minimal and maximal ratio between utilisations of modes can be set for each task set with three criticality levels. We

© 2017 Romain GRATIA

generate 500 task sets for each value of $U_{Bound} \in [0.6, m]$, with m the number of processors such that $m \in \{2, 4, 8\}$.

The comparison with optimal mixed–criticality scheduling algorithm is impossible since none can be designed. In the case of mixed–criticality systems, we know that no optimal scheduling algorithm exists [49]. Hence another method has to be found to measure how far the experimental performances of an algorithm are from those of an optimal scheduling algorithm. Yet, we know that an optimal mixed–criticality scheduling algorithm is able to schedule a mixed–criticality task set if the largest utilisation of the modes is lower than the number of processors. That is for a system with three modes if $max_{1 \leq l \leq 3} U_{\Gamma(\geq l)}(l) \leq m$, with m the number of processors, then the task set is schedulable by the optimal scheduling algorithm. We hence generate task sets such that tasks are created as long as $max_{1 \leq l \leq 3} U_{\Gamma(\geq l)}(l) \leq U_{Bound}$.

To that aim we first determine the criticality level of a task following a uniform distribution. We then draw the utilisation at the criticality level of the task. This utilisation value is limited to interval $[0.02, 0.7]$ to avoid tasks with a too large utilisation. Once this utilisation is drawn, we have to determine the other values for the tasks with a criticality level larger than 1.

We want to compute these utilisations while limiting the difference between the highest mode utilisation of the task and that of the lowest mode. Therefore, to compute the utilisation corresponding to the mode 1 of these tasks, we randomly draw a ratio between these two utilisations. Yet, we consider that this ratio can be larger for CL–3 tasks than for CL–2 tasks. Indeed, the safety requirements are more stringent for the former than for the latter. Hence, the ratio is drawn between 1 and a value that depends on the criticality level of the task. The different values used are presented in table 7.4. A third value is then required for the CL–3 tasks. It is randomly drawn such that in average the slack time is evenly distributed between the slack time of mode 2 and mode 3.

But, we want to respect the utilisation limit U_{Bound}. Therefore, for each utilisation value we take the minimum between the randomly drawn value and the difference between the utilisation U_{Bound} and the current utilisation of the corresponding mode. This ensures that the utilisation limit U_{Bound} is not exceeded.

Finally, a period is drawn following a log distribution as described in [76].

Once the task sets with three criticality levels have been created, we create those for the configurations with two criticality levels. We simply take the task sets with three criticality levels and transform them into dual–criticality task sets. The CL–1 and CL–2 tasks become CL–1 tasks and CL–3 tasks become CL–2 tasks in dual–criticality task

© 2017 Romain GRATIA

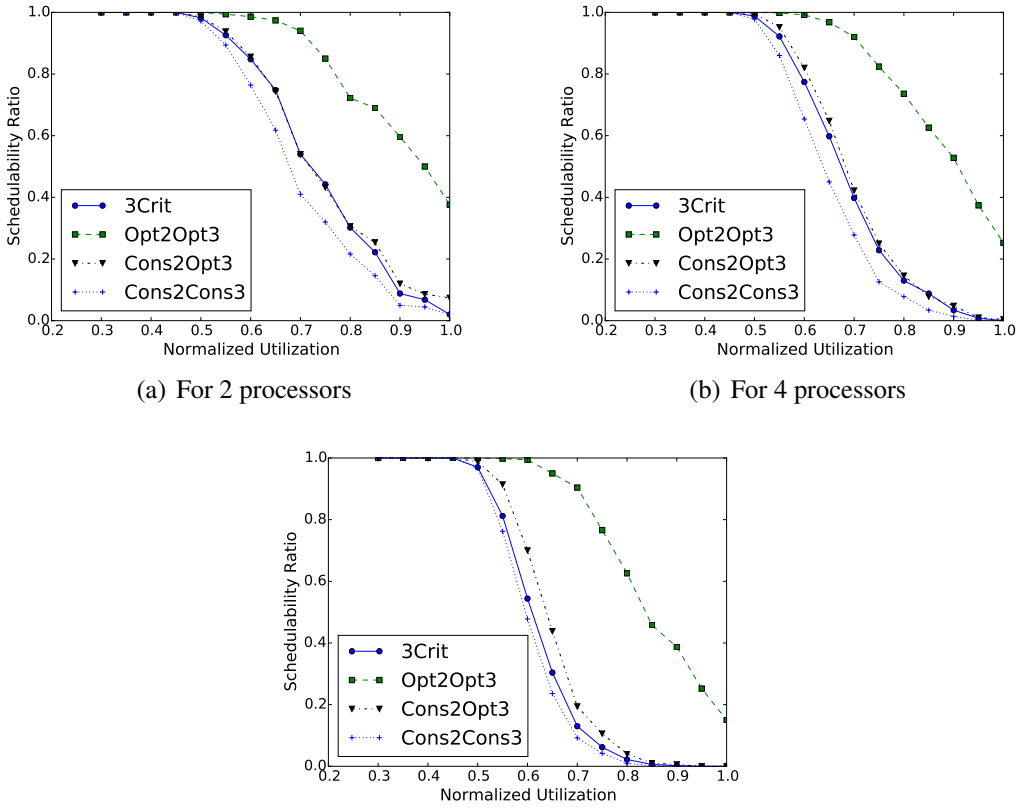
Criticality level	r_{max}
2	5
3	10

Table 7.4: Values of maximal ratio for different criticality levels

sets. The chosen values for their utilisations is performed as previously described for each configuration.

We then look for the allocations for all these task sets and measure the schedulability ratio for each couples of U_{Bound} and m.

Result exploitation



(a) For 2 processors

(b) For 4 processors

(c) For 8 processors

Figure 7.8: Schedulability ratio of GMC–RUN using three criticality levels or two criticality levels.

We assessed the three possible configurations with two criticality levels against the configuration with three criticality levels. The results are presented in figure 7.8. We recall

© 2017 Romain GRATIA

	CL–1 availability	CL–2 availability	CL–3 availability
Configuration Opt2Opt3	$\longrightarrow$	$\downarrow$	$\longrightarrow$
Configuration Cons2Opt3	$\nearrow$	$\downarrow$	$\longrightarrow$
Configuration Cons2Cons3	$\uparrow$	$\searrow$	$\longrightarrow$

Table 7.5: Impact on the availability of tasks in the different configurations compared to the three criticality level configuration

the impact on the availability of each configuration with two criticality levels compared to the configuration with three criticality levels in table 7.5.

The configuration with three criticality levels performs as well as the configurations Cons2Opt3 and Cons2Cons3 while offering better availability for CL–2 tasks. Indeed the schedulability ratio of the configuration with three criticality levels is almost equal to the schedulability ratio of the configuration Cons2Opt3 for two and four processors. With a difference between schedulability ratio limited to 5% with 2 and 4 processors for a normalised utilisation of 1.0 and 0.6, respectively. With 8 processors, this difference rises at up to 15% for a normalised utilisation of 0.6.

The difference, with the configuration Cons2Cons3 is far larger. With 2 and 4 processors the difference can go as high as 13% and 14%, respectively, for a normalised utilisation of 0.65. With 8 processors, the difference is limited to 5% at most.

The degradation of the schedulability performances of the 3Crit configuration with 8 processors can be explained by the combination of two factors. First, compared to systems with 2 and 4 processors, the number of possible allocations is far larger. In section 6.3.1 p 89, we presented that the number possible allocations can be computed with $(2^k)^n$, where n is the number of tasks to allocate and k the number of modal servers. The average number of tasks for systems with 8 processors is 25 against 12 for those with 4 processors. If we consider that for 4 processors we have in average 4 slackful modal servers and 8 tasks to allocate, it gives 4,294,967,296 possible allocations. Then for 8 processors, with an average of 8 slackful modal servers and 17 tasks, it gives $\backsim 8 \cdot 10^{40}$ possibilities. The second factor is that in the 3Crit configuration, modal servers have in average intervals of slack time of a utilisation 0.18 against 0.29 for the configuration Cons2Opt3. This smaller utilisation could be mitigated by finding the best possible allocation for each slackful modal server or by forming aggregated modal servers for system with 2 and 4 processors. But with a larger number of tasks, these allocations becomes far harder to find with 8 processors.

© 2017 Romain GRATIA

In terms of schedulability performances only, the configuration Opt2Opt3 is clearly the best but it is also the worse in terms of availability for the CL–2 tasks. Hence, if the system engineers only seek schedulability performances it is the configuration it should use. With 4 processors the difference between schedulability ratio of the configuration 3Crit and Opt2Opt3 can go as high as 60%.

7.5 Conclusion

In this chapter, we presented our approach to schedule mixed–criticality systems with more than two criticality levels. We first presented our notations to model such systems for any number of criticality levels. We also described the complexity to use the slack time in such systems. Indeed, in such system more tasks have available slack time and whose available amount depends on the active mode of the system. We provided a clear rule to ensure its safe use. We then extended the definition of modal servers for such systems that enables to take advantage of slack time of task of any criticality level.

Next, we described how to perform the allocation of tasks to modal servers. This is achieved through an inductive process that consider each mode at a time. During each iteration, the allocation is performed as for dual–criticality systems with an evolutionary algorithm.

Finally, we assessed the impact of the use of three or two criticality levels in different configuration to schedule a system whose tasks are classified in three criticality levels. These configurations offered different levels of availability guarantee for lower criticality tasks, with the configuration with three criticality levels providing the best guarantee for CL–2 tasks. It resulted that using the configuration with three criticality levels presents in most cases schedulability performances as good as those of two configurations with two criticality levels. Although, the configuration offering the best schedulablity performances is a configuration with two criticality levels, it is one the configurations offering the lowest guarantees for the availability of CL–2 tasks. Hence, our approach to schedule systems with more than two criticality levels can offer better availability without highly impacting the schedulability performances.

In the next chapter, we present our approach to schedule elastic task in dual–criticality systems.

© 2017 Romain GRATIA

8 The scheduling of elastic tasks

Table of Contents

In previous chapters, we presented our mixed–criticality scheduling algorithm GMC–RUN to schedule mixed–criticality systems using the discarding degradation model of task execution. First with two criticality levels and then for any number of criticality levels. But the discarding degradation model of task execution may not fit the industrial needs. Industrial partners may require that tasks are less frequently executed instead of being totally stopped as exposed in section 3.3. This corresponds to the elastic task model.

In this chapter, we seek to schedule LO elastic tasks with a minimum number of processors and as few preemptions as possible. However, our previous approach can not be used as is. Indeed, compared to discarding tasks, elastic tasks do not have decreasing budgets but have increasing periods. Besides, although LO elastic tasks have smaller execution requirements in HI mode, a minimal execution has yet to be ensured. These two differences make our approach unusable as is. Indeed, our approach requires that LO tasks are not executed in HI mode to be allocated in modal servers. Moreover, slackful modal servers and aggregated modal servers, are designed to provide additional execution time budget, not to execute tasks with different frequencies.

Hence to schedule a task set with the elastic tasks model, we first transform it in an equivalent task set compliant with the discarding model. This adaptation aims at switching from a change of periods to a change of budgets between modes. It is performed by splitting each LO elastic task in two subtasks. The first subtask ensures the minimal execution requirements of HI mode and is called **non mixed–criticality subtask**. The second one, only executed in LO mode, provides the additional execution requirement needed in LO mode compared to HI mode, and is called **discarding subtask**. The computation of the timing parameters of the subtasks is done in such way that the scheduling in each mode is ensured for all tasks even in case of a Timing Failure Event (TFE). Besides, they have to be computed such that the resources needed to execute the system are reduced. Therefore, they are computed such that their utilisations are not equal to the sum of the utilisations of LO and HI modes of the LO elastic task, but equal to its utilisation in LO mode. It at least ensures that we do not perform worse than if we executed the LO tasks with their LO mode timing parameters. Then, we find the best possible allocation of the discarding subtasks in slackful modal servers, so that dedicated execution resources are only needed for non mixed–criticality subtasks. Note that if a LO elastic task can not be allocated to any modal server, then it is always executed with its timing parameters of LO mode in slackless modal server.

However, the allocation of discarding subtasks in slackful modal servers requires that they are completely schedulable in these servers. If they are not, they are executed in slackless modal servers, and our approach is doubly penalised. Indeed, the execution

© 2017 Romain GRATIA

resources needed to execute the LO elastic task are those of the LO mode even in HI mode. And the slack time of HI tasks is left unused. Hence, the idea to limit this double penalty. Instead of seeking to completely execute a discarding subtask in slackful modal servers, only a part of its execution is allocated to such servers. We call such allocations **partial allocations**. It is performed by determining the maximum execution time that slackful modal servers provide. Yet, it may happen that the budget that can be provided by slackful modal servers is lower than the budget required to meet the execution requirements of the elastic task. In that case, the missing execution time budget has to be provided by the non mixed–criticality subtask. The principle is that the resulting allocation still enables the execution of LO elastic tasks with fewer execution resources than if LO elastic tasks were executed using only their LO mode timing parameters.

In this chapter, we first present the peculiarities of the elastic task model. Then, we explain how the slack time of HI tasks can be used to schedule the additional execution requirements of LO elastic tasks in LO mode. It is achieved by decomposing LO elastic tasks into two subtasks called non mixed–criticality subtask and discarding subtask. We also justify and explicit the conditions for the correctness of our approach. Next, we explain how we determine the periods and give the inequalities to compute the budget of each subtask. Finally, we describe how, during the allocation, we compute the budgets of each subtask by taking into account the budgets of modal servers.

8.1 The elastic task model

The elastic task model was first proposed by Buttazzo and al. in [19]. This model enables the adaptation of the rate of execution of a task, in particular in case of overloaded situations. The aim is to always provide minimum execution time to all tasks. In this section, we explicit the notations used for elastic tasks and the changes on the execution following a mode change.

An elastic task can be defined as follows:

Definition 39 (Elastic task)**.** *An elastic task is a mixed–criticality periodic task with implicit deadlines with two different periods. It is characterised by the following timing parameters:*

- *A criticality level* $\chi \in \{LO, HI\}$.
- *Two periods noted T(LO) and T(HI) such that* $T_i(LO) \leq T_i(HI)$*, if* $\chi = LO$*, and* $T_i(LO) = T_i(HI)$*, if* $\chi = HI$.

© 2017 Romain GRATIA

Task name	T(LO)	T(HI)	χ	C(LO)	C(HI)
τ_1	5	10	LO	2	2

Table 8.1: Example of an elastic task

- *Two execution budgets noted C(LO) and C(HI) such that $C_i(LO) = C_i(HI)$, if $\chi =$ LO, and $C_i(LO) \geq C_i(HI)$, if $\chi = HI$.*

The discarding task model can be seen as a particular case of the elastic task model. A discarding task would be an elastic task with an infinite period in HI mode.

An example of such elastic task is given in table 8.1, we use this task throughout this chapter to illustrate the issues to address and to detail our solutions. The execution of this task in each mode is presented in figure 8.1.

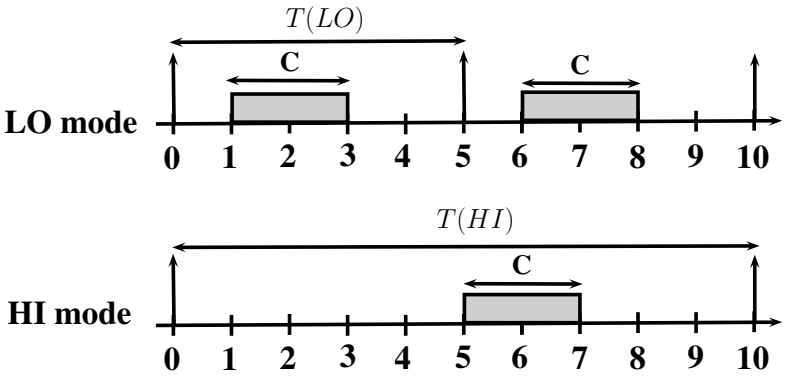


Figure 8.1: Representation of the execution of task τ_1 from table 8.1

Because of the two possible values for the period, we have to explicit the computation of the utilisation for each criticality level: $U_i(LO) = \frac{C_i(LO)}{T_i(LO)}$ and $U_i(HI) = \frac{C_i(HI)}{T_i(HI)}$. We also use the following notations: $U_{\Gamma(LO)}(LO) = \sum_{\tau_i \in \Gamma(LO)} U_i(LO)$, $U_{\Gamma(HI)}(LO) = \sum_{\tau_i \in \Gamma(HI)} U_i(LO)$, $U_{\Gamma(HI)}(HI) = \sum_{\tau_i \in \Gamma(HI)} U_i(HI)$.

In this chapter, we only consider systems with two criticality levels, composed of a set of elastic tasks Γ with n independent tasks, synchronously started, $\tau_1, ..., \tau_n$ and described by using two modes, called LO and HI modes. We consider the system starts executing in LO mode. This mode stays active, as long as each task τ_i completes its jobs in a time smaller than $C_i(LO)$. It changes its mode whenever a Timing Failure Event occurs that is when a task has executed for as long as its LO mode budget without completing its execution. In that case, the system changes of mode from LO mode to HI mode.

The scheduler has to ensure for the HI tasks the same scheduling objectives than with the discarding task model:

© 2017 Romain GRATIA

1. HI task jobs not yet completed can continue to execute for up to their $C(HI)$ without missing their deadlines

2. All following new HI task jobs can execute for as long as their $C(HI)$

But for the LO tasks these objectives have now changed:

1. LO task jobs not yet completed and started at a time t_s are executed up to C(HI) time. The deadline of the jobs are switched from deadlines computed using the periods of LO mode, $t_s + T(LO)$, to deadlines computed by using the periods of HI mode, $t_s + T(HI)$.

2. New jobs of LO tasks are executed up to $C(HI)$ every $T(HI)$.

Hence, a mode change notably impacts the activation times of elastic tasks. After, a mode change, LO tasks activation times are shifted by an offset equal to the difference between their periods of HI and LO modes. They no longer correspond to the activation times computed with their periods of LO mode were used. Nor are they those computed from their periods of HI mode. Hence, after a mode change, tasks can no longer be considered as synchronously started. Let demonstrate this property as the activation times and deadlines can be computed in each mode.

We now provide formula to compute the activation times and deadlines of jobs during which a TFE occurs, in LO mode and in HI mode. Assume that a Timing Failure Event (TFE) occurs at time t_{TFE}. The last activated LO (LAL) job of a LO task is the j_{LAL}^{th} job such that $j_{LAL} = \lfloor \frac{t_{TFE}}{T(LO)} \rfloor + 1$. The activation time of this job, noted act_{LAL}, can be computed by:

$$act(j_{LAL}) = \lfloor \frac{t_{TFE}}{T(LO)} \rfloor \cdot T(LO) = (j_{LAL} - 1) \cdot T(LO) \tag{8.1}$$

In LO mode, the activation time of the k^{th} job of a LO or HI task τ, such that $k < j_{LAL}$, and assuming a synchronous start of all tasks, is equal to:

$$act_{LO}(k) = (k-1) \cdot T(LO) \tag{8.2}$$

The corresponding deadline of this job is given by:

$$ddl_{LO}(k) = k \cdot T(LO) \tag{8.3}$$

In HI mode, the activation time of the k^{th} job of LO task τ, such that $k \geq j_{LAL}$, is given by:

$$act_{HI}(k) = act_{LAL} + (k - j_{LAL}) \cdot T(HI) \tag{8.4}$$

© 2017 Romain GRATIA

The corresponding deadline of this job is given by:

$$ddl_{HI}(k) = act_{LAL} + (k + 1 - j_{LAL}) \cdot T(HI) \tag{8.5}$$

For HI tasks, the equations for the computation of the release time and deadline remain the same than in LO mode since for these tasks $T(LO) = T(HI)$.

To exemplify this impact, consider the execution of task τ_1 in figure 8.2. At time $t = 8$ a TFE is detected. Hence, the activation of the job following the LAL job of task τ_1, that corresponds to its third job, is given by:

$$act_{LAL} + (k - j_{LAL}) \cdot T(HI) = \lfloor \frac{8}{5} \rfloor \cdot 5 + (3 - (\lfloor \frac{8}{5} \rfloor + 1)) \cdot 10 = 15 \tag{8.6}$$

The deadline of the third job is given by:

$$\lfloor \frac{8}{5} \rfloor \cdot 5 + (3 + 1 - (\lfloor \frac{8}{5} \rfloor + 1)) \cdot 10 = 25 \tag{8.7}$$

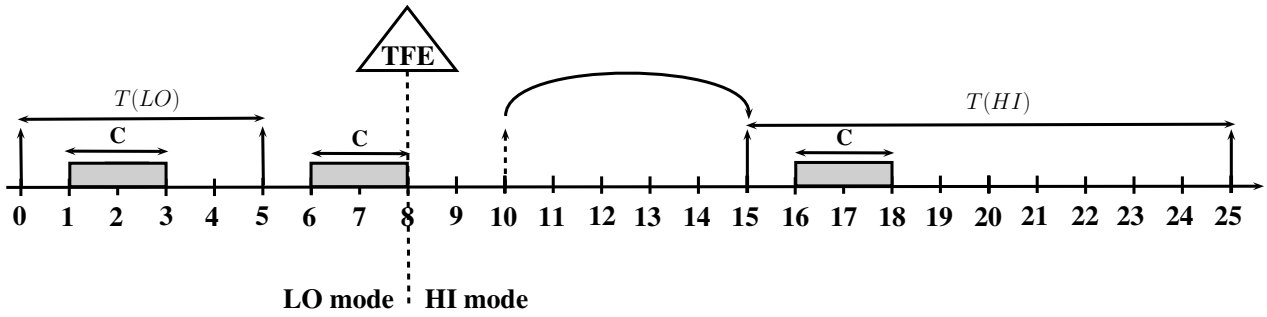


Figure 8.2: A mode change with elastic tasks

It follows the definition of the scheduling of a mixed–criticality system composed of elastic tasks:

Definition 40 (Scheduling objectives)**.** *A mixed–criticality system with elastic task is correctly scheduled if:*

1. *As long as the LO mode is active, all jobs complete their executions before their deadlines of LO mode and using at most their budgets of this mode.*

2. *As long as the HI mode is active, all jobs complete their executions before their deadlines of HI mode and using at most their budgets of this mode.*

These changes in the scheduling objectives of the LO task prevent the reuse as is of the slackful modal servers. Indeed, their use assumes HI tasks completely use the budget

© 2017 Romain GRATIA

of slackful modal servers and aggregated modal servers. Hence, with the current approach of GMC–RUN, LO elastic tasks are only schedulable in slackless modal servers, making the reduction of the execution requirements of the system impossible. In the following section, we present how we manage to use slackful modal servers to schedule LO elastic tasks and make possible the reduction of the execution requirements of the system.

8.2 Task elasticity and slack time

In this section, we describe how the decomposition of an elastic task into two subtasks enables their scheduling in HI task slack time. We also expose the conditions so that this decomposition produces a correct scheduling.

8.2.1 Decomposition in subtasks

We aim at finding how to schedule LO elastic tasks in slackful modal servers, aggregated or not. The issue is that slack time of HI tasks is available in LO mode only, while LO elastic tasks are executed in both LO and HI modes. To make the use of modal servers possible, we decompose each LO elastic task into two subtasks.

As previously said, LO elastic tasks have lower utilisations in HI mode than in LO mode. To take advantage of this observation, we divide the tasks into two subtasks one mandatory and one optional as presented in [78]. This technique as been already used for mixed–criticality systems in [79] but for HI tasks only. In this paper, subtasks are used to represent an increase in execution requirements of HI task in HI mode.

In our approach, we use this technique to model the increase of the execution requirements of LO elastic tasks in LO mode. Such decomposition is drawn in figure 8.3, for an elastic task of budget C and of periods T(LO) and T(HI). The decomposition is performed by defining two subtasks. A first one ensures the execution requirements of the HI mode and is thus always executed. It is referred to as non mixed–criticality subtask and noted τ^{NMC}. A second subtask provides the additional execution time needed in LO mode in addition to the first subtask. As it is only executed in LO mode, it executes as a LO discarding task, and as such it can be executed in slackful modal servers. Hence, it is referred to as discarding subtask and noted τ^{DIS}.

These subtasks are used to schedule another task, they hence are executions servers. These servers are then supposed to be scheduled by modal servers. Therefore, they have to execute as a periodic task with implicit deadlines. That is why the subtasks are modelled as periodic servers.

© 2017 Romain GRATIA

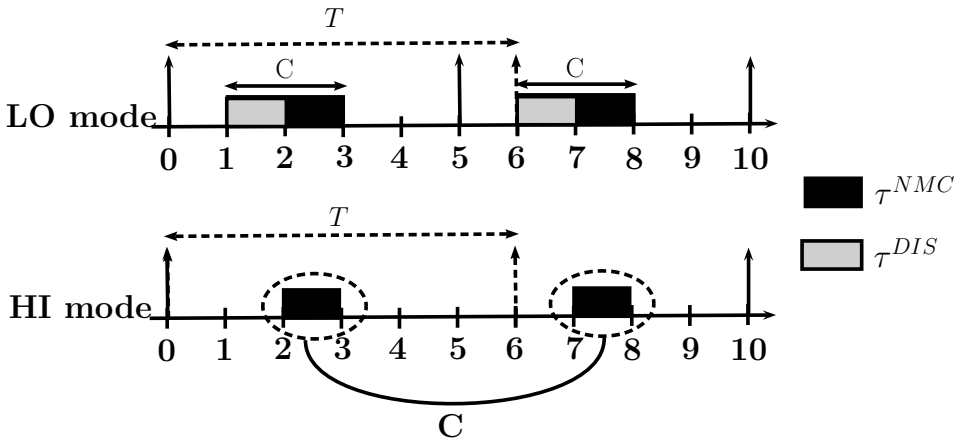


Figure 8.3: Decomposition of an elastic task execution in each mode

Now that we have determined how to decompose the execution of the elastic task, we have to compute the timing parameters of each subtask. Since, they are periodic servers, we have to determine a budget and a period for each subtask. The computation of these timing parameters have to fulfil several requirements:

1. They must be computed such that the execution of the LO elastic task is correct in each mode even in the case of a TFE.

2. We aim at reducing the overall execution requirements of the system. Hence, the scheduling of discarding subtask and non mixed–criticality subtask should not require more processing power than the scheduling the LO elastic task in LO mode.

3. The resulting timing parameters should limit the number of added preemptions. In particular, we want to avoid to have subtasks with a period of 1. Indeed, with periods of 1 our approach would mimic the execution produced by a fluid algorithm and would thus produce lots of preemptions. We can reduce the number of preemptions by limiting the number of executions of the subtasks required to complete one execution of the LO elastic task. We have hence to find the largest periods for the two subtasks such that the two other constraints are respected.

4. The subtasks are then to be scheduled by modal servers. It requires that tasks have a null offset, even after a TFE. These subtasks must hence have a single period and start executing with a null offset.

To ease the computation of these timing parameters and because the initial LO elastic task can execute only on one processor at a time, we assume that the two subtasks are

© 2017 Romain GRATIA

never executed in parallel. In the case of GMC–RUN, this requires to execute the two subtasks of a same elastic task in the same primal server.

Once the computation of the timing parameters has been performed for each subtask of the LO elastic tasks, we have to find the best possible allocation of the discarding subtasks in modal servers. It still is performed by using an evolutionary algorithms. Yet, we have to ensure that the two subtasks are not executed in parallel once the discarding subtask is allocated. Therefore, we have to check that the non mixed–criticality subtasks can be sequentially executed with the eventual slackful modal servers in which the discarding subtask has been allocated.

Finally, the modal servers resulting of the allocation process are then scheduled by RUN as they were with discarding tasks. This because they are regular periodic servers with a null offset.

In the next section, we give the conditions to ensure the scheduling of LO elastic tasks using our approach.

8.2.2 Correctness of the approach

In this section, we detail the conditions for the correctness of the scheduling of a LO elastic task in its discarding subtask and its non mixed–criticality subtask. These conditions are listed in the following theorem:

Theorem 13. *Let τ be a LO elastic task and τ^{DIS} its discarding subtask and τ^{NMC} its non mixed–criticality subtask. If the following conditions are fulfilled:*

1. *τ^{DIS} and τ^{NMC} are executed sequentially.*
2. *τ^{DIS} and τ^{NMC} are correctly scheduled.*
3. *The execution of both τ^{DIS} and τ^{NMC} ensure the scheduling of τ in LO mode.*
4. *The execution of τ^{NMC} only ensures the scheduling of τ in HI mode.*

then task τ is schedulable by τ^{DIS} and τ^{NMC} in LO and HI modes.

Proof. We prove this theorem by contradiction. Assume a task τ and its discarding subtask τ^{DIS} and its non mixed–criticality subtask τ^{NMC} such that the conditions of theorem 13 are respected but still task τ misses a deadline.

In LO mode, if task τ misses a deadline then τ^{DIS} and τ^{NMC} did not provide enough execution time. Either they were not correctly scheduled but that contradicts our second hypothesis, or their added up budget is not sufficient. That latter possibility contradicts

© 2017 Romain GRATIA

either our first assumption meaning that a part of the budget was lost because τ^{DIS} and τ^{NMC} executed in parallel while τ can execute in only one server at a time. Or our third assumption is violated, meaning that τ^{DIS} and τ^{NMC} can not provide sufficient execution time to the elastic task.

In HI mode, if τ misses a deadline then either τ^{NMC} can not correctly scheduled τ in HI mode but it contradicts our fourth assumption or τ^{NMC} is not correctly scheduled in contradiction with our second assumption. □

This theorem assumes our capacity to correctly compute the timing parameters of the discarding subtask and non mixed–criticality subtask.

8.3 Determining the timing parameters of the subtasks

In this section, we describe how we perform the computation of the timing parameters of the discarding subtask and of the non mixed–criticality subtask.

The computation of the timing parameters has several objectives to meet. It first has to ensure the correct execution of the LO elastic task. It also aims at minimising the utilisations of both the non mixed–criticality subtask and of the discarding subtask. And finally, it should limit the number of preemptions by finding the largest possible periods.

The computation of the timing parameters is performed through the use of a schedulability test. It is either based on the utilisations (see theorem 4 in section 5.3.3), or on the Supply Bound Function and the Demand Bound Function (see theorem 5 in section 5.3.3).

However, the schedulability test that can be used depends on a property on periods of the LO elastic task and on periods of its subtasks. That is whether LO elastic task periods are a multiple of the periods of the discarding subtask and of the non mixed–criticality subtask. If we assumed that periods are never multiples, we could only use the test based on SBF and DBF. This would give poor results as this test can reject cases that pass the schedulability test based on utilisations.

Therefore, it is critical to first determine the periods of the subtasks. Once these periods are determined we present the inequalities that budgets of the discarding subtask and non mixed–criticality subtask have to respect.

8.3.1 Selecting of the periods

In this section, we determine the periods of the two subtasks used to decompose an elastic task. The subtasks are periodic servers used to schedule a single task, which is a LO elastic task.

© 2017 Romain GRATIA

We showed in section 5.3.1, that the required utilisation of a server to schedule a task set is the lowest when its period is a divisor of the period of the tasks to schedule. We reuse this property in the case of the subtasks. Subtasks are periodic servers that execute a single task. Hence, to minimise the utilisations of both the discarding subtask and the non mixed–criticality subtask, their periods have to be divisors of the periods of the LO elastic task.

The non mixed–criticality subtask has to ensure the scheduling of the LO elastic task in LO and HI modes and when a mode change occurs. The simplest way to ensure the correct scheduling of a LO elastic task within its subtask is to choose its period T^{NMC} such that the subtask and the task have the same activation times. This can be ensured by choosing its period such that it divides the periods of the LO elastic task. Since, the non mixed–criticality subtask schedules the LO elastic task in both LO and HI modes it should divide both periods. But then, there may be several common divisors of the LO and HI periods.

To choose T^{NMC} among them, we take into account the fact that the lower the period of the subtask the more the execution of the LO elastic task will be split, resulting in more preemptions. To limit the number of additional preemptions the period of the non mixed–criticality subtask should be as large as possible. It means to take the greatest common divisor of the periods of the LO elastic task in LO and HI modes.

We follow the same reasoning for the period T^{DIS} of the discarding subtask. But this subtask has only to ensure the scheduling of the LO elastic task in LO mode. Hence, its period has only to be a divisor of the period of the LO elastic task in LO mode. To avoid unnecessary preemptions, we favour its largest divisor and choose the period in LO mode of the elastic task.

8.3.2 Finding subtask budgets

In this section, we give the inequalities used to compute the budgets of the discarding subtask and non mixed–criticality subtask.

To ensure the scheduling of a LO elastic task, the utilisations of its discarding subtask and of its non mixed–criticality subtask have to respect the conditions of the following theorem:

Theorem 14. *Let τ be a LO elastic task of utilisations U(LO) and U(HI), and of periods T(LO) and T(HI) for LO and HI modes respectively. Let τ^{NMC} be its non mixed–criticality subtask and τ^{DIS} its discarding subtask. Let C^{NMC} and C^{DIS} be the budgets of τ^{NMC} and τ^{DIS} respectively. Let T^{NMC} and T^{DIS} be the periods of τ^{NMC} and τ^{DIS}. If the following*

© 2017 Romain GRATIA

equations holds:

$$\frac{C^{NMC}}{T^{NMC}} + \frac{C^{DIS}}{T^{DIS}} \geq U(LO) \tag{8.8}$$

$$\frac{C^{NMC}}{T^{NMC}} \geq U(HI) \tag{8.9}$$

and τ^{NMC} *and* τ^{DIS} *are executed sequentially then* τ *is correctly scheduled.*

Proof. We have to prove that the two subtasks of an elastic task always ensure its scheduling. We decompose our proof in three cases: when the system is in LO mode, when it is in HI mode and when a TFE occurs.

In LO mode, we have to prove that the two subtasks ensure the correct scheduling of LO elastic tasks. The schedulability is ensured by theorem 4 p 71. Indeed, this theorem concerns the case where server periods divide the periods of the tasks they have to schedule. Since, this property holds here we can apply it. Besides, the two subtasks never execute in parallel. Indeed, T^{NMC} and T^{DIS} have been selected so that they divide T(LO). The tasks are schedulable if their utilisations is lower than the utilisation of the servers. Hence the sum of the utilisations of the two subtasks have to be larger than the utilisation of LO mode U(LO) of the elastic task.

We now consider the case of the job during which a TFE occurs at time $t_{TFE} \geq 0$. This job has eventually to respect the execution requirements of the LO elastic task in HI mode. We hence have to verify that only the execution of the non mixed–criticality subtask τ^{NMC} can ensure the scheduling of the LO elastic task. Indeed, after a TFE, the discarding subtask is no longer executed.

Assume, this job is the j^{th} job of the elastic task, $j \geq 1$. This job has to be executed up to C(HI). It executes in the interval $[t_{start}, t_{deadline}[$ with $t_{start} = (j-1) \cdot T(LO)$ and $t_{deadline} = (j-1) \cdot T(LO) + T(HI)$. The subtask τ^{NMC} is also released at time t_{start}, since T^{NMC} divides the period T(LO). It has also a deadline at $t_{deadline}$, since T^{NMC} divides $t_{deadline}$. Hence, over the interval $[t_{start}, t_{deadline}[$, τ^{NMC} executes $\alpha = \frac{T(HI)}{T^{NMC}} \geq 1$ times. From inequality 8.9 it holds that:

$$\frac{C^{NMC}}{T^{NMC}} \geq U(HI) \tag{8.10}$$

$$\iff \frac{C^{NMC}}{T^{NMC}} \geq \frac{C(HI)}{T(HI)} \tag{8.11}$$

© 2017 Romain GRATIA

$$\iff \frac{T(HI)}{T^{NMC}} \cdot C^{NMC} \geq C(HI) \tag{8.12}$$

$$\iff \alpha \cdot C^{NMC} \geq C(HI) \tag{8.13}$$

This holds for any job and for any t_{TFE}.

In HI mode, after a mode change occurred at time $t_{TFE} \geq 0$, the released jobs now suffer from an offset. We have hence to prove that the non mixed–criticality subtask can still ensure the scheduling of the LO elastic task.

The k^{th} job of an elastic task starts executing at $t_{start} = \lfloor \frac{t_{TFE}}{T(LO)} \rfloor \cdot T(LO) + (k - (\lfloor \frac{t_{TFE}}{T(LO)} \rfloor + 1)) \cdot T(HI)$, with $k \geq 1$. Its absolute deadline is $t_{deadline} = \lfloor \frac{t_{TFE}}{T(LO)} \rfloor \cdot T(LO) + (k + 1 - (\lfloor \frac{t_{TFE}}{T(LO)} \rfloor + 1)) \cdot T(HI)$. Over the interval $[t_{start}, t_{deadline}[$, the elastic task needs to execute for C(HI) units of time.

The non mixed–criticality subtask τ^{NMC} is also activated at time t_{start}. Indeed, T^{NMC} divides t_{start}, for any t_{TFE} and k, since T^{NMC} divides T(HI). Besides, in interval $[t_{start}, t_{deadline}[$, subtask τ^{NMC} executes $\alpha = \frac{T(HI)}{T^{NMC}} \geq 1$ times. Therefore, subtask τ^{NMC} executes for $\alpha \cdot C^{NMC}$ units of time over the interval $[t_{start}, t_{deadline}[$.

From inequality 8.9 it holds that:

$$\frac{C^{NMC}}{T^{NMC}} \geq U(HI) \tag{8.14}$$

$$\iff \alpha \cdot C^{NMC} \geq C(HI) \tag{8.15}$$

This holds for any job started in HI mode and for any t_{TFE}. Hence, the schedulability of the elastic in HI mode is ensured.

This proves theorem 14. □

This theorem gives us the inequalities to compute the budgets of the two subtasks. Yet, there is some scope in their computations that can be taken advantage of to improve schedulability performances.

Balancing the budgets

Observe in theorem 14 that there is a possible balance between the budget of the non mixed–criticality subtask and the budget of the discarding subtask. Indeed, in inequality 8.8, the larger the budget of the non mixed–criticality subtask is, the smaller the budget

© 2017 Romain GRATIA

of the discarding subtask can be. Instead of considering that the non mixed–criticality subtask has only to ensure the execution requirements of the HI mode, it can also provide some of the additional execution requirements needed in LO mode. This enables to reduce the budget of the discarding subtask. This can be taken advantage of when performing the allocation of discarding subtask.

So far, when we performed the allocation of a LO task in a modal server, i.e in the slack time of a HI task, the slack time provided by the HI task could only be used to completely execute the LO tasks. If it is not possible, that is if no schedulability test is passed, our approach is imposed a double penalty. First, the LO task is provided with a dedicated execution time budget that covers its whole execution requirements. Second, HI task slack time remains totally unused. That case makes our use of the slack time inefficient. The idea to mitigate these two drawbacks is to only allocate to modal servers what they can actually execute, this is what we call a partially allocated task:

Definition 41 (Partially allocated task)**.** *A partially allocated task is a task whose only a part of its execution is performed in a modal server.*

In the case of elastic task, it consists in computing the budget of the discarding subtask such that it fits in a slackful modal server or an aggregated modal server.

Recall that what can be executed in slackful modal servers or aggregated modal servers does not have to be accounted for when sizing the required processing power to execute the system. Hence, we should compute the maximum budget that can be provided by a slackful modal server or an aggregated one, this corresponds to an optimisation problem.

8.4 Resolution of the optimisation problem

In this section, we expose our solutions to resolve our optimisation problems. The first problem, is to compute the maximum budgets of discarding subtasks that can be provided by a slackful modal server or an aggregated modal server. The second problem is to find the allocation that minimises the required processing power to execute a system.

We first explain how we intend to solve these two problems. Then, we explain how the evolutionary algorithm is used to find a good allocation. Finally, we describe how the budgets of the subtasks are computed for a given allocation of tasks to a modal server.

8.4.1 Decomposition of the optimisation problem

The two problems cannot be solved independently. Indeed, the maximal budgets of discarding subtasks that can be allocated to modal server depends on which modal server is

 © 2017 Romain GRATIA

allocated discarding subtasks. Therefore, for each allocation we need to compute the maximal budgets of the discarding subtasks. But because these two problems have different properties we do not use the a unique method to solve them.

The problem of finding the allocation of LO elastic tasks with the largest utilisation of discarding subtasks in modal servers has the same properties than for discarding tasks. Therefore, we also solve this problem by using evolutionary algorithm. We exposed these properties and justify why we considered that evolutionary algorithm is the best method to resolve it in section 6.3.1 p 89.

Concerning the computation of the maximum budget a different method can be used. Indeed, budgets are continuous variables and not discrete as for the problem of finding the allocation. Therefore, more classical resolution methods for continuous optimisation problems can be used. These methods are more efficient than the use of an evolutionary algorithm.

In the next sections, we describe how the evolutionary algorithm we used for discarding tasks is refined for elastic tasks. Then, we present the inequalities to solve in order to find the maximal budgets of discarding subtasks for a given allocation of tasks to a modal server. Two cases are considered, when the modal server period divides the periods of the discarding subtasks and when it does not.

The individuals are evaluated by computing the sum of the utilisations of the discarding subtask. This individual is penalised if the the corresponding non mixed–criticality subtasks and the modal server cannot be executed sequentially.

8.4.2 Refining the evolutionary algorithm

The allocation of the LO elastic tasks in modal servers is also performed by using an evolutionary algorithm. Although the genotype does not change, only the discarding subtask of a LO elastic task is allocated to modal servers and not the whole LO task.

Concerning the operators, only the selection operator has to be adapted. Indeed, for each allocation we do not perform a schedulability test to check that a set of discarding subtasks is schedulable in a modal server. Instead, we compute their budgets such that they are schedulable in the modal server. Then, we compute the budgets of the non mixed–criticality subtasks with the inequalities of the theorem 14 ensuring the schedulability of the LO elastic task. Besides, we have to check that the corresponding non mixed–criticality subtask is never executed in parallel of the modal server. With GMC–RUN, that latter condition is checked by verifying that the non mixed–criticality subtask can be scheduled in the same RUN primal server than the slackful modal server. The individuals

© 2017 Romain GRATIA

are evaluated by computing the sum of the utilisations of the discarding subtask. This individual is penalised if the the corresponding non mixed–criticality subtasks and the modal server cannot be executed sequentially.

8.4.3 Computation of the budgets of the subtasks

For a given set of discarding subtasks allocated to a slackful modal server or an aggregated modal server, we seek to maximise the utilisation of discarding subtasks scheduled in it. When performing the allocation, there might be tasks in this set that can be completely executed in it while others might only be partially allocated. We have hence to decide which tasks should be completely executed in the modal server and which should only be partially allocated. It is hence an optimisation problem. We distinguish two cases when resolving this problem: when modal server periods divide the periods of the discarding subtasks and when they do not.

Task periods are multiple of modal server periods

We want to maximise the utilisation of discarding subtasks allocated to the slackful modal servers or the aggregated modal servers. In the case modal server period divides those of the discarding subtasks, we can use the theorem 4 based on the utilisation and presented p 71. This maximum can be found by solving the following optimisation problem:

Definition 42. *Let Γ^{DIS} be a set of discarding subtasks, and Γ^{NMC} be the set of corresponding non mixed–criticality subtasks of a set Γ of LO elastic tasks. Let MS be a modal server. The period of MS divides all periods of the discarding subtasks in Γ^{DIS}. We note U_{MS} the utilisation of MS and U_{MS}^{slack} the overall utilisation of the slack time provided by MS. We note $U_i^{DIS/alloc}$ the utilisation of the task $\tau_i^{DIS} \in \Gamma^{DIS}$ that can be allocated in modal servers MS. We also note U_i^{NMC} the utilisation of the corresponding non mixed–criticality subtask in Γ^{NMC}. We finally note $U_i(LO)$ and $U_i(HI)$ the utilisations in LO and HI of the corresponding LO elastic task. To find the maximal utilisation $U_i^{DIS/alloc}$ of each discarding subtasks in Γ^{DIS}, we have to solve the following optimisation problem:*

$$
\begin{array}{ll}
Maximise & \sum_{\tau_i \in \Gamma^{DIS}} U_i^{DIS/alloc} \\
Subject\ to & \\
 & \sum_{\tau_i \in \Gamma^{DIS}} U_i^{DIS/alloc} \leq U_{MS}^{slack} \\
 & U_i^{DIS/alloc} \leq U_i(LO) - U_i(HI)\ \forall \tau_i^{DIS} \in \Gamma^{DIS} \\
 & \sum_{\tau_i \in \Gamma^{NMC}} U_i^{NMC} + U_{MS} \leq 1
\end{array}
$$

© 2017 Romain GRATIA

The inequalities $\sum_{\tau_i \in \Gamma^{DIS}} U_i^{DIS/alloc} \leq U_{MS}^{slack}$ ensure that the discarding subtasks are schedulable in MS.

The inequalities $U_i^{DIS/alloc} \leq U_i(LO) - U_i(HI)$ are here to upper bound the budgets of the discarding subtasks. It is not relevant to have a discarding subtask with a higher utilisation than the difference of the utilisations between modes. Indeed, a non mixed–criticality subtask has a utilisation at least equal to the utilisation in HI mode of the LO elastic task. Therefore, it is only interesting to allocate an utilisation equal to up the difference of utilisations between modes LO and HI.

The inequalities $\sum_{\tau_i \in \Gamma^{NMC}} U_i^{NMC} + U_{MS} \leq 1$ ensure that the non mixed–criticality subtasks and the modal server can be executed sequentially.

The budgets of non mixed–criticality subtasks can then be computed using the inequalities presented in theorem 14, that gives the conditions on subtask budgets to schedule a LO elastic task.

Task periods are not multiple of modal server periods

We now compute the maximum budgets of discarding subtasks that can be allocated to a modal server, in the case where task and modal server periods are not multiples. In that case, we have to solve the following optimisation problem:

Definition 43. *Let Γ be a set of LO elastic tasks. We note $U_i(LO)$ and $U_i(HI)$ the utilisations in LO and HI of the LO elastic task $\tau_i \in \Gamma$. Let Γ^{DIS} be the corresponding set of discarding subtasks to be allocated in a modal server MS of utilisation U_{MS}. We note Γ^{NMC} the set of corresponding non mixed–criticality subtasks and U_i^{NMC} the utilisation of a subtask $\tau_i^{NMC} \in \Gamma^{NMC}$. We want to determine the budget $C_i^{DIS/alloc}$ for each discarding subtask $\tau_i^{DIS} \in \Gamma^{DIS}$ of period T_i^{DIS} such that the utilisation of allocated discarding subtasks is maximised. This requires to solve the following optimisation problem:*

$$
\begin{aligned}
&\textit{Maximise} \quad \sum_{\tau_i \in \Gamma^{DIS}} \frac{C_i^{DIS/alloc}}{T_i^{DIS}} \\
&\textit{Subject to} \\
&\qquad \sum_{\tau_i \in \Gamma^{DIS}} \left\lfloor \frac{t}{T_i^{DIS}} \right\rfloor C_i^{DIS/alloc} \leq \sum_{MS_j \in \Gamma_{MS}} SBF_j(t) \;\forall\, t \in [0, LCM] \\
&\qquad \frac{C_i^{DIS/alloc}}{T_i^{DIS}} \leq U_i(LO) - U_i(HI) \;\forall \tau_i^{DIS} \in \Gamma^{DIS} \\
&\qquad \sum_{\tau_i \in \Gamma^{NMC}} U_i^{NMC} + U_{MS} \leq 1
\end{aligned}
$$

Where LCM is the hyper–period.

© 2017 Romain GRATIA

The inequalities with the DBF and the SBF ensure the schedulability of the discarding subtasks.

The inequalities $\frac{C_i^{DIS/alloc}}{T_i^{DIS}} \leq U_i(LO) - U_i(HI)$ set an upper bound on the budgets of the discarding subtasks. It is not relevant to have a discarding subtask with a higher utilisation than the difference of the utilisations between modes. Indeed, non mixed–criticality subtask have a utilisation at least equal to the utilisation in HI mode of the LO elastic task.

The inequalities $\sum_{\tau_i \in \Gamma^{NMC}} U_i^{NMC} + U_{MS} \leq 1$ ensure that the non mixed–criticality subtasks and the modal server can be executed sequentially.

The budgets of non mixed–criticality subtasks can then be computed using the inequalities presented in theorem 14, that gives the conditions on subtask budgets to schedule a LO elastic task.

8.5 Discussion

In this section, we discuss the advantages of our approach to schedule elastic tasks compared to existing works.

Originality: it is, to our knowledge, the first global scheduling algorithm for the elastic task model. The only multi–processor scheduling algorithms for the elastic task model are partitioned algorithms [57; 59].

Advantages: our approach presents several advantages compared to existing scheduling algorithms for the elastic task model.

First our task model is more general than those used in other scheduling algorithms. It does not propose a variety of period values as large as in [19; 80], where periods can take any value in an interval of values, but a larger choice of values for the budgets can be used. Indeed, our approach can handle increasing or constant periods with constant or decreasing budgets when changing from LO mode to HI mode. A model with decreasing budgets for LO task was proposed in [81] and applied in [82], though only for uniprocessors. This is possible because our approach enables a flexible distribution of the budgets between the two subtasks.

Besides, our task model is sufficient to handle actual systems. Indeed, in actual systems the periodicity of tasks cannot take any value as they can be constrained by other fields such as the control theory for instance. The impact of the objectives pursued by control engineers on the choice that can be made computer engineers is described in [83]. In this

© 2017 Romain GRATIA

paper is described a method to implement a Flight Management System modelled with SIMULINK on many–core platforms and notably for the choice of the frequencies of the execution of tasks.

Finally, compared to the approach in [57], we give prior the execution of the system the guarantee that LO tasks can effectively execute in LO mode. Indeed, in [57], the schedulability test only ensures that the HI mode is schedulable. It is only during the execution of the system that it is determined if LO tasks can execute more frequently. This is determined at precise instants, called early release points. If, at these instants, HI tasks have left enough slack time, a LO task can be executed earlier. But there is no methods to know prior execution whether this will actually happen. On the contrary, our approach ensures that the LO mode is schedulable and it will stay active as long as no Timing Failure Event occurs.

8.6 Conclusion

In this chapter, we have described how to schedule elastic tasks within modal servers. It is performed by decomposing each elastic task into two subtasks. One subtask, called non mixed–criticality subtask, is executed in both LO and HI modes. It must at least ensure the correct execution of the elastic task in HI mode. A second subtask, called discarding subtask, is executed only in LO mode. It has to provide the additional execution time required in LO mode. With the non mixed–criticality subtask, it ensures that the LO elastic task is correctly scheduled in LO mode. We made explicit the conditions to ensure a correct scheduling of an elastic task. In particular, it requires the sequential execution of the discarding subtask and of the non mixed–criticality subtask. We explained how to determine the period of each subtask, and presented the inequalities that their budgets have to respect.

Finally, we described how we take advantage of modal server budgets during the allocation to maximise the allocated utilisation of LO elastic tasks through the use of the partial allocation.

Our approach is to our knowledge the first global scheduling algorithm for elastic tasks. The elastic task model is sufficient for actual systems. Besides, it can easily be extended to handle elastic task with a decreasing budget and an increasing period and ensure that LO mode will be active. It also ensures that LO tasks can be effectively executed with their LO mode timing parameters before a TFE occurs.

© 2017 Romain GRATIA

© 2017 Romain GRATIA

9 Conclusion

TABLE OF CONTENTS

This chapter concludes this dissertation and identifies future research opportunities.

9.1 Conclusion

In this thesis, we have proposed a versatile approach to efficiently schedule mixed–criticality systems. Mixed–criticality systems are composed of several tasks with different criticality levels. The scheduling of these systems can be efficiently performed by accepting to degrade the availability of tasks with lower criticality levels.

There exist several mixed–criticality task models to perform this degradation. In this thesis, we studied two task models. The most commonly used results in the complete stop of the execution of a task and is called the discarding task model. We first considered it with two criticality levels. We then considered this task model with more than two criticality levels. Using more criticality levels enables to more finely classify tasks, and thus to more finely degrade the availability of tasks. The last studied model is the elastic task model with two criticality levels. With that model, task executions are more gracefully degraded by simply reducing the frequency of their executions instead of stopping them completely.

Our contributions enable to schedule mixed–criticality system efficiently while giving the system designers sufficient scope to adjust the availability of each task to meet its objectives. These contributions of this thesis are the following:

- We first introduce a new kind of execution servers called modal servers for mixed–criticality systems; we first apply them to schedule systems with two criticality lev-

els composed of discarding tasks. These servers enforce a uniprocessor mixed–criticality scheduling policy that takes advantage of the slack time of HI tasks to schedule LO ones. Each time a LO task is executed in the slack time of a HI task, the processing power required by the system, measured through its utilisation, is reduced. They handle the mixed–criticality system peculiarities while executing as regular periodic servers. We described three types of modal servers: slackful modal servers, aggregated modal servers and slackless modal servers. We provided the theorems to ensure the correct scheduling of tasks in these modal servers.

- These modal servers are then used in a hierarchical scheduling framework. Modal servers are scheduled using any multi–processor hard real–time scheduling algorithm, since they execute as regular periodic servers. We explain how we chose the Reduction to UNiprocessor (RUN) scheduling algorithm for its theoretical and practical performances. We therefore called our hierarchical scheduling framework GMC–RUN. The efficiency of GMC–RUN depends on our ability to execute the largest amount of LO task utilisation in HI task slack time. It corresponds to a difficult optimisation problem, that we resolved through the use of an evolutionary algorithm.

- We then assessed GMC–RUN. First theoretically, by proving it has a speed–up factor of 2. It is the second best speed–up among mixed–criticality scheduling algorithms for multi–processors. Secondly, experimentally on randomly generated task sets. We measured the schedulability ratio of GMC–RUN and compared it with two others global mixed–criticality scheduling algorithms: MC–DP–Fair and fpEDF–VD. Again GMC–RUN presents the second best performances, while being closer to the best algorithm MC–DP–Fair than to fpEDF–VD. Finally, we measured the number of preemptions per job for MC–DP–Fair and GMC–RUN. GMC–RUN entails at least five times fewer preemptions than MC–DP–Fair.

- We then proposed a simple inductive process to schedule, with GMC–RUN, mixed–criticality systems with more than two criticality levels. We compared the performances of our scheduling algorithms with tasks sets scheduled using three or two criticality levels. It results that using three criticality levels gives the best compromise between availability and schedulability efficiency.

- We then explained a method to schedule elastic tasks with GMC–RUN. We schedule elastic tasks into two kinds of execution servers: a modal server and a regular periodic server. The regular periodic server provides the required execution in HI

© 2017 Romain GRATIA

mode. The modal server provides the additional execution required in LO mode. Our approach is the first global scheduling algorithm for elastic tasks. Compared to other elastic scheduling algorithms, it allows to consider a more general elastic task model, guarantees the LO tasks availability and as a global scheduling algorithm should provide good efficiency performances.

- We finally introduced the partial allocation of tasks in modal servers. This allows to use the slack time of HI tasks that was not used because not large enough. Indeed, we now seek to execute in a modal server only what it can schedule of a task instead of seeking to completely execute the task in it. Besides, it can be used for both the elastic and the discarding task models. It should thus improve the efficiency of our approach for both models.

Our contributions form a scheduling framework that is both efficient and versatile. Yet, it also creates new research opportunities.

9.2 Future Work

We identified several research opportunities to complete our contributions.

Elastic tasks

Elastic tasks scheduling : The performance assessment of our approach to schedule systems composed of elastic tasks with two criticality levels would first require to determine how to properly evaluate the different approaches to compare them. We saw in section 8.5 p 154, that our approach ensures the scheduling of LO mode prior the execution of the system while another research work does not. The advantages or disadvantages of each approach can not be measured with the mere comparison of schedulability ratio as it is usually performed. Indeed, this performance criteria does not allow to evaluate the task availability ensured by each approach.

Elastic task model with more than two criticality levels : The elastic task model used in our approach only use two periods. Yet, system designers could be interested to use more than two periods. This would offer system designers more latitude to adjust the availability of each task. This extension could be considered as a generalisation of our approach to elastic task model with more than two criticality levels.

© 2017 Romain GRATIA

Implementation of GMC–RUN

Attempts to implement GMC–RUN have been carried out from the existing RUN implementation [36]. However, we realised that the existing implementation was a very early prototype and was not in compliance with what was described in [36]. Due to these technical issues, we were not able to achieve the implementation of GMC–RUN before the end of this thesis. The interest for an implementation remains, since we designed our algorithm with the objective to make it practicable.

Partial allocation in practice

We have to find an efficient method to perform the partial allocation of tasks and integrate in our tool to find allocations. We intend to implement this approach not only for elastic tasks but also for discarding tasks. Thanks to the partial allocation a higher share of the available slack time will be used and it should hence improve our schedulability performances. We will also observe the effect of the use of this approach on the number of preemptions, since it results in splitting the execution of allocated LO tasks between at least two servers.

Design process

Deriving a complete conception process can help system designers finding the best trade–off between availability and efficiency as it has been performed in [65]. This process would make use of the different task models, discarding and elastic, with potentially more than two criticality levels. This would require to be able to determine the gain in availability that each possibility offers. It would also require methods to determine the task budgets for each criticality level potentially associated to a confidence level, that would indicate the probability that the budget has to be exceeded. This could be achieved by taking advantage of the probabilistic WCETs [84; 85], that aims computing WCET associated with a probability of being exceeded.

© 2017 Romain GRATIA

10 Résumé en français de la thèse

Table of Contents

10.1 Introduction

Ce document résume les travaux effectués dans le cadre de la thèse de Romain GRATIA sous la supervision de Laurent PAUTET et Thomas ROBERT de Télécom ParisTech. Cette thèse a été effectuée au sein de l'Institut de Recherche Technologique SystemX (IRT SystemX) dans le cadre du projet Électronique et Logiciel pour l'Automobile (ELA). Ce projet a regroupé de nombreux acteurs de la filière automobile française, comme Renault, le groupe PSA, Continental et Valéo ainsi que l'Institut Mines–Télécom, l'Université Paris Sud, l'Université Pierre et Marie Curie et des PMEs comme Smile (anciennement OpenWide) et Intempora.

10.2 Contexte industrielle et motivations

L'industrie automobile est confrontée depuis quelques années aux défis posés par l'intégration de plus en plus d'applications dans une voiture. Cette tendance est par ailleurs appelée à se renforcer du fait du développement de la voiture autonome. Elle a pour conséquence de forcer les concepteurs des systèmes embarqués à repenser leurs pratiques, afin de maintenir des coûts de conception et de production à des niveaux raisonnables.

Une voiture représente ce qui est appelé un système critique, car son dysfonctionnement, ou d'un de ses sous–systèmes, peut engendrer des conséquences graves pour ses utilisateurs et son environnement. Au sein de ce système, certaines applications peuvent être considérées comme plus critiques que d'autres, c'est-à-dire, dont le dysfonctionnement peut avoir des conséquences plus graves que d'autres. Il est donc possible de classer les applications dans différents niveaux de criticité. Différentes méthodes de classification existent, souvent propre à chaque secteur industriel. Dans l'automobile cette classification est décrite dans la norme ISO26262, qui classe les applications dans 4 *Automotive Safety Criticality Levels*. Chaque niveau de criticité est associé avec un certain nombre de précautions à prendre lors de la conception et de l'exécution des applications correspondantes. Ces précautions étant de plus en plus contraignantes à mesure que le niveau de criticité augmente.

Une des difficultés rencontrées lors de la conception de tel système est le dimensionnement de la puissance de calcul nécessaire permettant d'assurer l'exécution correcte de toutes les applications. Cette puissance de calcul doit être dimensionnée au plus juste afin d'éviter des surcoûts inutiles. De fait, des architectures et des méthodes de conceptions permettent de faciliter la conception de tels systèmes.

Dans un premier temps, dans le but de faciliter le dimensionnement et la correction de l'exécution des applications, celles–ci sont découpées en éléments exécutables séquentiellement appelés tâches. A chaque tâche est associé un morceau de code exécutable séquentiellement et un modèle d'exécution. Il existe plusieurs modèles de tâches qui se distinguent sur trois aspects: les conditions d'activations, les contraintes temporelles sur l'exécution et les besoins en puissance de calcul. Une tâche peut être soit activée si certains événement se produisent, soit activée à des dates pré–déterminées. Les contraintes se caractérisent souvent par des échéances, c'est-à-dire des instants auxquels l'exécution d'une tâche doit être terminée. Enfin, les besoins en puissance de calcul correspondent à une estimation du temps nécessaire pour qu'une tâche termine une exécution.

Une fois cette décomposition en tâches effectuée, la théorie de l'ordonnancement permet de déterminer si un ensemble d'applications peut être correctement exécuté sur un

© 2017 Romain GRATIA

ensemble de processeurs, et plus particulièrement la théorie de l'ordonnancement des systèmes temps réel. Il existe pour cela des procédures de vérifications, appelées tests d'ordonnançabilité, qui permettent de s'assurer que l'exécution de chaque tâche respectera les caractéristiques de son modèle d'exécution. Afin de limiter le nombre de choix de conception, un certain nombre d'architectures ont été définies. Celles–ci précisent notamment la répartition des applications sur les calculateurs.

L'architecture actuellement utilisée est l'architecture fédérée. Celle–ci consiste à exécuter chaque application sur des processeurs dédiés. Elle présente l'avantage d'éviter que l'exécution des tâches d'une application soit perturbée par l'exécution des tâches d'une autre application. Cela est particulièrement intéressant dans le cas où d'applications avec des niveaux de criticité différents. Cependant, avec le nombre croissant d'applications à embarquer, la pertinence de cette architecture est de plus en plus discutable. Actuellement elle se traduit déjà par l'intégration de près de 100 Electonic Control Units (ECU). Or plus le nombre d'ECUs embarqués est grand plus la vérification de la correction de l'exécution des tâches est compliquée [4]. En effet, cette architecture requiert l'échange de messages sur un réseau (de type CAN) reliant les ECUs. La complexification du réseau avec le nombre grandissant d'ECU augmente les chances d'un manque de bande passante sur le réseau ou de grande latence dans la réception des messages. Ces deux phénomènes pouvant alors avoir un impact sur l'exécution des tâches d'une application.

Une architecture alternative, appelée architecture intégrée, est actuellement considérée. Elle consiste à partager les processeurs entre différentes applications, ce partage visant à utiliser plus efficacement la puissance de calcul des processeurs. Cependant, du fait des niveaux de criticité différents, les possibilités offertes par cette architecture ne sont utilisées qu'à moitié. En effet, pour l'instant seules les applications de même criticité peuvent être exécutées sur les mêmes processeurs, limitant ainsi les gains potentiels. Cependant cette approche n'est plus tenable pour les années à venir.

De cette description, il faut en effet retenir qu'un élément clé de la réussite ou non de l'exécution d'applications sur un ensemble de processeurs est l'estimation des besoins en puissance de calcul des tâches. Celle–ci est effectuée pour des tâches temps réel par l'estimation du pire temps d'exécution (WCET).

Cependant, si l'utilisation des WCETs pour estimer les besoins en temps d'exécution des tâches assure une exécution sûre du système, elle conduit aussi à une surestimation des besoins en puissance de calcul. En effet, ces WCETs sont en général beaucoup plus grands que les temps d'exécution moyens des tâches. Ceci s'explique par la très grande variabilité du temps d'exécution des tâches qui est due aux différents chemins possibles dans le code d'une tâche. Ces différents chemins se traduisant par l'exécution de dif-

© 2017 Romain GRATIA

férentes instructions dont les temps d'exécution eux–même sont différents et variables d'une exécution à une autre. Cette variabilité est encore plus marquée pour les multi–processeurs du fait du partage de ressources entre processeurs, comme les caches ou le bus d'accès à la mémoire [10; 11; 9]. Ces variabilités font qu'aucune méthode d'estimation du WCET d'une tâche ne fournit d'estimation précise. Le manque de maîtrise de ces variabilités est alors compensée par l'usage de marges lors de l'estimation des WCETs, afin d'assurer l'exécution correcte des tâches. Ces marges se traduisent par une surestimation des WCETs et sont d'autant plus importantes que la tâche est critique.

Des travaux ont visé à essayer de gérer cette variabilité, et deux tendances se dégagent. D'un côté, certains travaux essayent d'éliminer cette variabilité grâce à des méthodes de partitionnement [12; 13] ou basées sur le principe du Time Division Multiple Access (TDMA) [14; 15]. Cependant, ces méthodes ont tendance à augmenter les temps d'accès aux ressources et demandent une connaissance fine de l'architecture du matériel utilisé. De telles méthodes sont inadaptées au contexte automobile de cette thèse.

L'autre approche possible est de tolérer cette variabilité. La première solution est d'intégrer cette variabilité dans les WCETs des tâches, mais cette solution est inefficace du fait du sur–dimensionnement de la puissance de calcul nécessaire qu'elle induit. Une autre approche consistant à surveiller l'exécution des tâches et à mettre en place des mécanismes évitant la propagation des anomalies lorsqu'elles se produisent. Cette dernière approche est considérée dans le cadre de l'ordonnancement temps réel.

L'une de ces approches a été proposées par Vestal [16]. Elle vise à adapter l'estimation des WCETs des tâches à leur niveau de criticité tout en partageant de mêmes processeurs pour exécuter des tâches avec des niveaux de criticité différents. Cette proposition a donné lieu à la création des systèmes dits à criticité mixte et à des algorithmes d'ordonnancement dédiés. Ceux–ci visent à trouver un meilleur compromis entre l'usage efficace de la puissance de calcul des processeurs et la disponibilité des tâches, en particulier des moins critiques.

Nos travaux ont porté dans ce contexte à développer de nouvelles méthodes d'ordonnancement pour les systèmes temps réel critiques à criticité mixte exécutés sur des processeurs multi–processeurs.

© 2017 Romain GRATIA

10.3 Ordonnancement des systèmes à criticité mixte: modèles et état de l'art

Dans cette section, nous présentons les modèles de systèmes à criticité mixte et les algorithmes d'ordonnancement dédiés à ces systèmes existants. L'ordonnancement des systèmes à criticité mixte visent en effet à utiliser plus efficacement la puissance de calcul des processeurs sans compromettre le sûreté de l'exécution des tâches les plus critiques. Ils requièrent donc des modèles spécifiques et des algorithmes d'ordonnancement associés.

10.3.1 Mixed–criticality task model

Le modèle des tâches à criticité mixte [16] est une extension des tâches temps réel classiques.

Definition 44 (Mixed Criticality Task Set)**.** *Dans un lot de tâches Γ à criticité mixte périodiques, $\tau_1, ..., \tau_n$, chaque tâche τ_i est caractérisée par 5 paramètres, $(T_i(LO), T_i(HI), \chi_i, C_i(LO), C_i(HI))$ tels que:*

- *χ_i est le niveau de criticité, ou criticité, de la tâche pris dans $\{LO, HI\}$, où HI dénote une criticité plus importante que LO, ce qui est noté $HI > LO$.*
- *$T_i(LO)$, $T_i(HI)$ sont les périodes pour les niveaux LO et HI.*
- *$C_i(LO), C_i(HI)$ les budgets temps d'exécutions pour les niveaux LO et HI.*

Nous utilisons aussi les notations suivantes:

- Une tâche LO et une tâche HI son respectivement des tâches avec une criticité LO et HI respectivement.
- Si une tâche τ_i est suivi du suffixe $\{\chi_i = LO\}$, or $\{\chi_i = HI\}$, cela signifie que la tâche τ_i est une tâche LO ou HI respectivement.
- L'utilisation U d'une tâche périodique de période T et de budget C est $U = \frac{C}{T}$. Dans le cas des systèmes à criticité mixte, il existe autant d'utilisations que de niveaux de criticité: $U_i(LO) = \frac{C_i(LO)}{T_i(LO)}$ et $U_i(HI) = \frac{C_i(HI)}{T_i(HI)}$ pour les niveaux LO et HI respectivement. L'utilisation peut par ailleurs est calculée pour un lot de tâches Γ comme suit: $U_\Gamma = \sum_{\tau_i \in \Gamma} U_i$.

© 2017 Romain GRATIA

Les paramètres utilisés lors de l'exécution d'un lot de tâches dépendent des temps d'exécution effectifs des tâches. Tant que les paramètres temporels du niveau LO sont respectés, ceux–ci sont utilisés notamment pour l'activation et les échéances des tâches. Si un job s'est exécuté pendant aussi longtemps que le budget de niveau LO de la tâche sans avoir pour autant terminé son exécution, alors les paramètres utilisés changent. Ce comportement est appelé une *Timing Failure Event* (TFE):

Definition 45 (Timing Failure Event)**.** *L'instant auquel un job dépasse le budget du niveau LO de sa tâche sans avoir terminé son exécution est appelé une Timing Failure Event (TFE).*

Nous faisons l'hypothèse dans la thèse, que le budget de mode HI d'une tâche HI est sûr, et ne peut donc pas être dépassé. Lorsqu'une TFE se produit, les paramètres temporels utilisés par les tâches changent, c'est ce qui est appelé un changement de mode [18]. Nous pouvons alors définir au moins deux modes d'exécution. Ceux–ci sont nommés d'après les criticité car leur définition en dépend:

Definition 46 (Mode d'exécution)**.** *Soit L une criticité, le mode d'exécution L, noté mode L, nécessite que toutes les tâches τ_i, avec une criticité χ_i telle que $\chi_i \geq L$, soient ordonnancées et respectent leurs paramètres temporels de mode L.*

Durant l'exécution d'un système, si un mode est dit actif à un instant t, cela signifie alors que toutes les tâches de criticité au moins L doivent respecter leurs temps d'activation et leurs échéances définis à partir de leurs paramètres temporels de mode L. Le mode actif à un instant t correspond au mode avec la criticité la plus petite, telle qu'aucun job ne s'est exécuté pendant un temps plus long ou égal que le budget de cette criticité. En conséquence, le mode LO est toujours le mode initial. Par ailleurs, l'ordonnancement de ces tâches nécessitent de pouvoir assurer que les échéances des tâches HI dans le nouveau mode seront respectées si une TFE se produit. Et cela, même si ce changement se traduit par une augmentation de budgets pour les tâches LO et HI. C'est ce que nous appelons la **propriété de continuité de l'ordonnancement**.

L'introduction des modes nécessitent l'introduction de notations supplémentaires. $\Gamma(LO)$ et $\Gamma(HI)$ représentent le lot de tâches LO et de tâches HI dans Γ respectivement. La notation $U_{\Gamma(X)}(Y)$ représente l'utilisation du lot de tâches $\Gamma(X)$ à la criticité Y, avec X étant aussi une criticité. Par exemple $U_{\Gamma(LO)}(LO)$ représente l'utilisation des tâches LO en mode LO.

Si un changement de mode se traduit potentiellement par un changement de tous les paramètres temporels d'une tâche, dans les faits, seuls quelques paramètres changent.

© 2017 Romain GRATIA

Dans le modèle le plus répandue, seuls les budgets changent pour les tâches HI de la manière suivante:

- $T_i(LO) = T_i(HI)$
- $C_i(LO) \leq C_i(HI)$.

En ce qui concerne les tâches LO, plusieurs modèles de dégradation de leur exécution existent et sont appelés modèles de dégradation de l'exécution des tâches LO. Nous en considérons deux modèles: le modèle à annulation et le modèle de la tâche élastique.

Modèle de dégradation d'exécution

L'ordonnancement des systèmes à criticité mixte visent à utiliser plus efficacement la puissance de calcul des processeurs sans compromettre le sûreté de l'exécution des tâches les plus critiques. Ceci n'est possible qu'en acceptant que les tâches de plus basse criticité voient leur exécution dégradée pour permettre l'exécution en mode HI des tâches HI. Deux modèles sont considérés dans la thèse pour effectuer cette dégradation.

Le modèle de dégradation à annulation. Lorsqu'une TFE se produit, tous les jobs actuels et futurs des tâches LO sont arrêtés. Nous appelons ce modèle le modèle de tâche à annulation. C'est le modèle le plus couramment utilisé actuellement. Il en résulte que les périodes et les budgets d'une tâche LO τ_i sont constants: $T_i(LO) = T_i(HI)$, and $C_i(LO) = C_i(HI)$. Mais ce modèle peut être considéré comme trop définitif dans la dégradation des tâches LO.

Le modèle des tâches élastiques. Ce modèle suppose que les tâches LO doivent toujours être exécutées. Cependant leurs paramètres de mode LO sont leurs paramètres de références avec lesquels elles doivent être exécutées. Le mode HI leur offrant seulement une exécution dégradée du fait de l'utilisation d'une période plus grande. Nous avons les hypothèses suivantes sur les paramètres d'une tâche LO élastique τ_i: $T_i(LO) \geq T_i(HI)$ and $C_i(LO) = C_i(HI)$.

Noter qu'un système à criticité mixte peut être transformé en un **système à criticité multiple**. Cela correspond à un système dans lequel les tâches LO et HI sont toujours exécutées avec leurs paramètres temporels correspondant à leur criticité.

La modèlisation des systèmes à criticité mixte étant très différente des systèmes temps réel classique ceux–ci nécessitent des algorithmes d'ordonnancement adaptés.

© 2017 Romain GRATIA

10.3.2 Algorithmes d'ordonnancement pour les tâches dites à annulation

Dans cette section, nous présentons brièvement les algorithmes d'ordonnancement existants pour les systèmes à criticité mixte pour multi–processeurs. Une présentation plus complète de l'ensemble des résultats pour ces systèmes est disponible dans l'état de l'art écrit par Burns [45]. Nous classons ces algorithmes selon qu'ils sont capables de gérer deux niveaux de criticité ou plus pour le modèle à annulation et considérons enfin les algorithmes pour le modèle élastique. Mais tout d'abord, nous présentons brièvement les critères de performances utilisés pour comparer des algorithmes d'ordonnancement.

Critères de performance

Les performances d'un algorithme d'ordonnancement sont mesurées par sa capacité à pouvoir ordonnancer un grand nombre de lots de tâches pour un nombre de processeurs donné. Il existe différentes manières de mesurer cette capacité: certaines sont théoriques d'autres expérimentales.

Critère théorique Un algorithme capable d'ordonnancer tous les lots de tâches ordonnançables pour un certain de processeurs est dit *optimal*.

Cependant il n'est pas toujours possible de développer un algorithme optimal. C'est pourquoi il est alors intéressant de déterminer l'écart entre les performances d'un algorithme et celles d'un algorithme optimal, même théorique. Cela est possible grâce au facteur speed–up [30].

Le facteur speed–up est régulièrement utilisé pour évaluer théoriquement les performances des algorithmes d'ordonnancement temps réel. Le principe de ce facteur est de comparer les performances de l'algorithme étudié avec celles d'un hypothétique algorithme optimal et clairvoyant. Plus ce facteur est petit plus l'algorithme a de bonnes performances.

Le facteur speed–up a été défini comme le ratio s par lequel la fréquence des processeurs doit augmenté de sorte à ce que un algorithme S ait les mêmes performances qu'un algorithme d'ordonnancement optimal clairvoyant. Il calculé de sorte à ce que l'inégalité suivante soit respectée:

$$max_I \frac{S_s(I)}{A_1(I)} \leq c \tag{10.1}$$

© 2017 Romain GRATIA

Avec $S_s(I)$ le coût de l'ordonnancement par l'algorithme S de l'entrée I avec des processeurs de fréquence s. $A_1(I)$ est le coût de l'ordonnancement par l'algorithme clairvoyant pour les mêmes entrées et avec des processeurs de fréquence 1. Et c est une constante, généralement prise égale à 1 pour déterminer à quelle vitesse l'algorithme S a les mêmes performances que l'algorithme clairvoyant. Un algorithme optimal possède un facteur speed–up de 1. Cependant, le facteur speed–up ne permet que d'avoir une indication sur les pires performances d'un algorithme. Par ailleurs, il ne prend pas en compte les surcoûts nécessaires pour appliquer la politique d'ordonnancement.

Critères empiriques Ces critères servent à donner une vision sur les performances moyennes d'un algorithme d'ordonnancement. Il consiste généralement à générer aléatoirement des lot de tâches censés être représentatifs de ceux qui peuvent être rencontrés dans des systèmes industriels. Plusieurs méthodes de génération de lot de tâches ont été proposées pour les systèmes à criticité mixte [32; 33; 34]. Puis, des indicateurs statistiques sont calculés à partir de ces lots de tâches, comme la probabilité qu'un lot de tâche soit ordonnançable par un algorithme donné. L'un de ces indicateurs est le taux de succès d'ordonnançabilité, qui consiste à compter le nombre de lot de tâches ordonnançables.

Estimation des surcoûts. La génération de ces lots de tâches permet par ailleurs de simuler leur ordonnancement, permettant ainsi de compte le nombre de préemptions et de migrations. Ces événements ayant un impact non négligeable sur les performances d'un algorithme d'ordonnancement.

Une autre solution est d'implémenter l'algorithme dans un système d'exploitation. Ce prototype est ensuite évalué en vérifiant que les tâches sont correctement exécutées et en mesurant le temps d'exécution de l'ordonnanceur. Plusieurs implémentations ont été faites [26; 35; 36] et ont permis de déterminer si les algorithmes étudiés pouvaient être implémentés efficacement.

Algorithmes pour les systèmes avec deux niveaux de criticité

Algorithmes d'ordonnancement partitionnés. Dans leur article [46], les auteurs proposent une extension de l'algorithme *zero-slack rate-monotonic* (ZSRM) [47]. Cet algorithme se base sur le calcul d'instants appelés *zero slack instants*. Ce sont les derniers instants à partir desquels une tâche HI ne peut plus terminer son exécution en utilisant son budget de mode HI même si les tâches LO sont arrêtées. Leur approche consiste donc à les arrêter au plus tard lorsque ces instants sont atteints. Ces instants peuvent être utilisés avec une politique d'ordonnancement monoprocesseur telle que EDF ou RM. Pour leur

© 2017 Romain GRATIA

extension aux multi–processeurs, ils utilisent des heuristiques pour l'allocation des tâches aux processeurs telles que Best Fit et Worst Fit.

L'algorithme monoprocesseur EDF–VD a aussi été étendue aux multi–processeurs grâce à l'approche partitionée [48]. EDF–VD, pour *EDF with Virtual Deadlines*, est le meilleur algorithme monoprocesseur pour les systèmes à criticité mixte [49], lorsque les facteurs speed–up sont considérés. Cet algorithme consiste à calculer des échéances virtuelles pour les tâches HI. Celles–ci sont calculées de sorte à ce que les tâches HI soient exécutées plus tôt en mode LO pour qu'elles puissent s'exécuter pendant aussi longtemps que leur de budget de mode HI en cas de changement de mode. Ces échéances virtuelles sont calculées grâce à un facteur qui diminue l'échéance des tâches HI en mode LO. La version partitionnée de cet algorithme a facteur speed–up de $\frac{8m-4}{3m}$, où m représente le nombre de processeurs.

Cependant les algorithmes partitionnés sont connus pour ne pas avoir les meilleures performances d'ordonnançabilité.

Algorithmes d'ordonnancement semi–partitionnés. Dans leur article [50], les auteurs proposent un algorithme semi-partitionné. Les changements de mode sont faits processeurs par processeurs, c'est-à-dire que si une anomalie est détectée sur un processeur, seul ce dernier change de mode. Les tâches LO de ce processeur migrent alors vers un autre processeur. Le test d'ordonnançabilité est basé sur le test *Adaptive Mixed Criticality* (AMC) [51]. C'est un test basé sur l'analyse du temps de réponse. Des mécanismes pour retourner vers le mode LO sont aussi proposés. Mais cette approche étant basée sur une approche partitionnée n'autorisant les migrations que dans le cadre des changements de mode, ses performances doivent être similaires à un algorithme partitionné.

Une extension aux systèmes à criticité mixte de NPS–F [40] est proposée dans l'article [52] . Les tâches HI sont allouées sur l'ensemble des processeurs disponibles. La capacité restante de chaque processeur est ensuite utilisée pour exécuter des tâches LO. Cependant la description complète de l'algorithme reste à faire.

Algorithmes d'ordonnancement globaux. Le principe des échéances virtuelles est réutilisé dans deux algorithmes globaux.

Le premier se base sur l'algorithme fpEDF et est appelé fpEDF–VD [53]. Le calcul des échéances virtuelles est fait d'une manière analogue à celle présentée pour EDF–VD, simplement le test d'ordonnançabilité est remplacé par celui de fpEDF. Cet algorithme un facteur speed–up de $\sqrt{5}+1$, ce qui ne représente pas la meilleur performance pour ces algorithmes.

© 2017 Romain GRATIA

Le second algorithme est MC–DP–Fair [34], qui est actuellement l'algorithme avec le meilleur facteur speed–up et qui présente aussi les meilleurs taux d'ordonnançabilité. Les auteurs ont d'abord adapté le concept d'algorithme fluide aux systèmes à criticité mixte donnant l'algorithme MC–Fluid. Il consiste à calculer des taux d'utilisations pour les tâches HI de sorte à ce qu'elles s'exécutent suffisamment en mode LO pour pouvoir s'exécuter avec leur budget de mode HI d'une façon sûre après un changement de mode. MC–DP–Fair est l'adaptation de DP–Fair qui reprend le calcul de nouveaux taux d'utilisation mais utilise aussi les échéances virtuelles. Ces deux algorithmes ont un facteur speed–up $\frac{4}{3}$ [54], qui est la valeur optimale pour un algorithme d'ordonnancement pour systèmes à criticité mixte [49]. Cependant, ces algorithmes basés sur l'ordonnancement fluide sont connus pour engendrer un très grand nombres de migrations et de préemptions.

Algorithmes pour les systèmes avec plus de deux niveaux de criticité.

Dans cette section, nous considérons les algorithmes capable de gérer plus de deux niveaux de criticité.

Le premier algorithme pour ces systèmes est MC^2 [56], qui peut gérer 5 niveaux de criticité. C'est un algorithme hiérarchique à 5 niveaux, un pour chaque niveau de criticité. Chaque niveau a une politique d'ordonnancement dédiée. Pour les tâches les plus critiques, l'ordonnancement est assuré par la génération d'une table d'ordonnancement. Ensuite, les tâches avec le second niveau de criticité sont ordonnancées par l'algorithme EDF partitionné. Puis les tâches des troisième et quatrième niveaux de criticité sont ordonnancées avec global EDF. Le dernier niveau de criticité est ordonnancé avec un algorithme d'ordonnancement *best effort*. Sur chaque processeurs les tâches sont ordonnancées dans l'ordre décroissant de leur niveau de criticité. Bien qu'implémentable [32], cet algorithme requiert cependant que les périodes des tâches du second niveau de criticité soient des multiples de celles des tâches du premier niveau. Il n'est donc utilisable que pour des cas bien spécifiques.

10.3.3 Algorithmes adaptés au modèle de tâches élastiques

Nous présentons dans cette section, les algorithmes multi–processeurs dédiés à l'ordonnancement des tâches élastiques sur multi–processeurs.

Le premier algorithme a été présenté dans [57]. Les auteurs présentent un algorithme semi–partitionné. Les tâche LO et HI sont allouées aux processeurs en ne considérant que leurs paramètres temporels de mode HI. L'ordonnancement se fait ensuite selon early–release EDF [58]. Cette version modifiée de EDF permet d'exécuter les tâches avec une

© 2017 Romain GRATIA

période plus petite que celle du mode HI si les tâches HI n'utilisent pas tout leur budget de mode HI. La détermination si une tâche LO peut se faire avec une plus petite période se fait à des instants nommés *early release points*. Cependant, il est impossible de savoir avant l'exécution que le lot de tâches pourra vraiment être exécuté avec les paramètres de mode LO.

Un second algorithme a été présenté dans [59]. Cet algorithme est un algorithme partitionné. Deux allocations des tâches sont faites: une pour le mode LO, une pour le mode HI. Dans chaque allocation les tâches HI restent sur les mêmes processeurs. Par contre, les tâches LO peuvent être amenées à changer de processeur si un changement de mode se produit. Le test d'ordonnançabilité utilisé pour allouer les tâches aux processeurs se base sur l'analyse du temps de réponse, qui sert aussi à déterminer les priorités fixes des tâches. Cependant, bien que cet algorithme assure que le mode LO peut être exécuté, le changement de mode requiert d'arrêter momentanément les tâches LO. Elles sont arrêtées tant que les tâches LO, qui doivent migrer, n'ont pas migré et que les jobs des tâches HI au moment du changement de mode n'ont pas terminé leur exécution.

10.4 Problématique

Nombre d'algorithmes d'ordonnancement pour systèmes à criticité mixte ont été présentés notamment pour les multi–processeurs. Cependant, comme présenté dans la section précédente, chacun de ces algorithmes présente des défauts soit en terme d'efficacité soit sur leur capacité à être utilisé dans des systèmes réels. Ces limitations ayant trait aux nombres de niveaux de criticité ordonnaçables et ou l'utilisation exclusive au modèle de tâche à annulation. Nous visons à proposer une approche pour l'ordonnancement de ces systèmes qui soit efficace et utilisable. Comme pour les algorithmes actuels, nous avons adapté un algorithme d'ordonnancement temps réel aux systèmes à criticité mixte. Mais, pour éviter les défauts des algorithmes actuels et nous assurer que nous obtenons bien les performances voulues, nous avons conçu notre approche étape par étape.

La première étape a visé à éviter que l'approche finale exhibe les mêmes défauts que les algorithmes actuels. C'est pourquoi nous effectuons l'adaptation d'un algorithme en gardant en tête les objectifs que l'ont souhaite atteindre. Cela commence lors du choix de l'algorithme à adapter qui doit présenter les caractéristiques visées. Il s'agit ensuite d'évaluer et anticiper les surcoûts que l'adaptation engendrera et essayer de les limiter. Enfin, il faut aussi garder en tête que l'ordonnancement des systèmes à criticité mixte avec plus de deux niveaux de criticité est nécessaire mais qu'il n'est pas forcément aisé de passer de l'ordonnancement à deux niveaux vers plus de deux.

© 2017 Romain GRATIA

Ensuite, il s'agit de s'assurer que l'adaptation respecte bien les objectifs d'ordonnancement des systèmes à criticité mixte. Notamment l'adaptation doit pouvoir assurer l'ordonnancement des différents modes et assurer la continuité de l'ordonnancement lorsqu'un changement de mode se produit. En effet, celui–ci se traduit par une augmentation du budget des tâches HI qui doivent pouvoir l'utiliser avant leurs échéances. Enfin il faut que cette adaptation soit faite de manière efficace afin d'assurer de meilleurs performances d'ordonnançabilité.

La dernière étape a consister à permettre l'ordonnancement d'un autre modèle de dégradation que celui dit de l'annulation. En l'occurrence, nous avons considéré le modèle des tâches élastiques. Or ce modèle requiert de continuer à exécuter les tâches LO en mode HI et ce avec des période différentes entre les mode LO et HI, ce qui introduit un certains de différences qu'il faut gérer par rapport au modèle de dit de l'annulation.

10.5 GMC–RUN: un algorithme d'ordonnancement multi–processeur pour système à criticité mixte

Dans cette section, nous présentons notre algorithme GMC–RUN, pour *Generalised Mixed–Criticality–RUN*. Nous décrivons d'abord son fonctionnement global. Puis, nous décrivons comment nous parvenons à le rendre efficace.

10.5.1 Décomposition du problème d'ordonnancement

Pour concevoir notre algorithme d'ordonnancement à criticité mixte sur multi–processeur, nous avons divisé le problème d'ordonnancement en deux sous problèmes. Le premier sous problème concerne le problème d'ordonnancement à criticité mixte. Le second concerne le problème d'ordonnancement sur multi–processeurs.

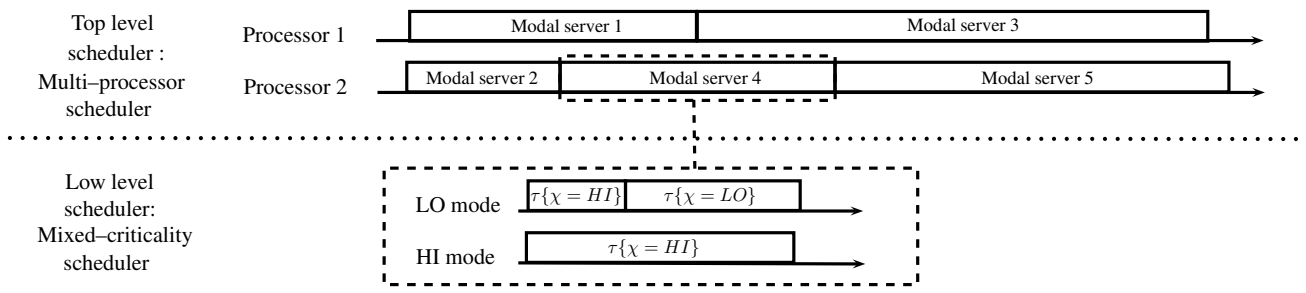


Figure 10.1: Représentation de l'algorithme hiérarchique

© 2017 Romain GRATIA

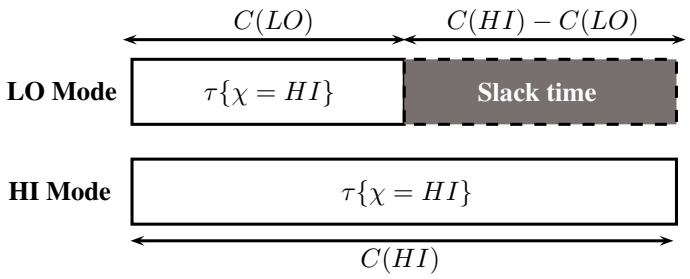


Figure 10.2: Origine du slack time.

Pour mettre en application cette décomposition du problème en deux sous problème nous avons utilisé des serveurs d'exécution défini comme suit:

Definition 47. *Un serveur d'exécution est une séquence d'opérations qui vise à ordonnancer des tâches. Il se caractérise par un modèle de ressource, un lot de tâches et une politique d'ordonnancement. Un serveur d'exécution utilise ses ressources d'exécution décrit par son modèle de ressource pour exécuter le lot de tâches selon sa politique d'ordonnancement.*

Les serveurs sont couramment utilisés pour appliquer une politique d'ordonnancement spécifique à un sous lot de tâches. L'utilisation de ces serveurs nous permet de décomposer de le problème d'ordonnancement comme présenté dans la figure 10.1.

Dans la partie inférieure de cette figure, les serveurs sont utilisés pour appliquer une politique d'ordonnancement à criticité mixte monoprocesseur. Elle s'applique à un sous ensemble de tâches du système.

Ces serveurs sont ensuite ordonnancés selon une politique d'ordonnancement multi–processeur temps réel comme représenté dans la partie supérieure de la figure 10.1. L'emploi de ces deux algorithmes forment ce qui est appelé un algorithme d'ordonnancement hiérarchique à deux niveaux. Le niveau du bas, applique la politique d'ordonnancement à criticité mixte dans les serveurs. Le niveau du haut, applique la politique d'ordonnancement multi–processeur aux serveurs. Il nous reste maintenant à préciser ces deux politiques d'ordonnancement.

Les systèmes à criticité mixte sont définis avec deux modes d'exécution, les modes LO et HI. Dans chaque mode est associé aux tâches un lot de paramètres temporels propre à un mode. Si les tâches HI sont exécutées avec leur budget de mode HI, même dans le mode LO, alors une part de ce budget reste inutilisée tant que le mode LO est actif. Cette partie inutilisée du budget est appelée *slack time*. La quantité de slack time disponible par tâche est estimable avant son exécution, comme le montre la figure 10.2. La quantité de slack time disponible est égal à la différence entre les budgets des modes HI et LO pour

© 2017 Romain GRATIA

chaque tâche HI. Selon les modes il peut être utilisé pour terminer l'exécution de la tâche HI ou pour exécuter des tâches LO.

Pour cela, il nous faut concevoir une politique d'ordonnancement qui permet d'ordonnancer correctement les tâches LO et HI dans ce slack time. Celle–ci doit assurer que les tâches HI peuvent terminer leur exécution avant leurs échéances même en cas de changement de mode. C'est pourquoi notre politique termine d'abord l'exécution des tâches HI, nous permettant de nous assurer que celles–ci n'ont pas besoin de leur slack time. Ce qui nous permet alors de pouvoir ordonnancer des tâches LO d'une manière sûre dans le slack time des tâches HI. Étant donné que l'ordonnancement dans le slack time des tâches HI est similaire à ordonnancer des tâches sur un monoprocesseur, nous utilisons une politique d'ordonnancement monoprocesseur.

Nous avons besoin d'un modèle de ressource pour nos serveurs qui nous permet d'utiliser le slack time des tâches HI. Le modèle de ressource utilisé correspond à un modèle de ressource utilisé pour les systèmes temps réel classique: avec un WCET et une période. Un tel modèle de ressource est le modèle des ressources périodiques [66].

Dans ce modèles une ressource est accessible pour une durée fixée par un budget qui est réapprovisionné périodiquement. Par ailleurs, ce modèle présente l'avantage d'être très répandu et compatible avec de nombreux algorithmes d'ordonnancement temps réel classiques.

Ces serveurs appliquent une politique d'ordonnancement à criticité mixte qui tire profit du slack time laissé par les tâches HI. Ce slack time est soit utilisé pour ordonnancé un lot de tâches LO ou pour terminer l'exécution des tâches HI en mode HI. Ainsi ces serveurs sont appelés des serveurs modaux définis comme suit:

Definition 48. *Un serveur modal MS est un serveur dont le modèle de ressource est défini par une période T_S, un budget C_S et une utilisation $U_S = \frac{C_S}{T_S}$. Ce budget est abondé à chaque période T_S. Il possède deux lots de tâches:*

- *Un lot de tâches HI $\Gamma_{MS}(HI)$.*
- *Un lot de tâches LO $\Gamma_{MS}(LO)$.*

Ces lots de tâches sont ordonnancés selon politique d'ordonnancement suivante:

- *En mode LO, les tâches HI dans $\Gamma_{MS}(HI)$ sont ordonnancées en premier. Une fois ces tâches terminées, les tâches LO dans $\Gamma_{MS}(LO)$ sont ordonnancées. Les ordonnancements de ces deux lots de tâches est fait suivant une politique d'ordonnancement monoprocesseur.*

© 2017 Romain GRATIA

- *En mode HI, seules les tâches HI dans $\Gamma_{MS}(HI)$ sont ordonnancées.*

On appelle serveur modal slackful, les serveurs modaux avec une seule tâche HI dans le lot $\Gamma_{MS}(HI)$. Les serveurs ne disposent que d'une quantité limitée mais constante de slack time pour exécuter un lot de tâches LO et qui n'est disponible que dans un intervalle de temps bien défini. Par ailleurs, le slack time disponible peut être globalement grand mais distribué entre beaucoup de tâches. La quantité de slack time disponible par tâche HI peut donc être très petite, ne permettant d'exécuter aucune tâche LO. Afin de limiter cet éparpillement du slack time, nous utilisons le slack time de plusieurs tâches HI pour exécuter un même lot de tâches LO. Les serveurs modaux exécutant chacune de ces tâches formant alors ce qu'on appelle des **serveurs modaux agrégés**. Enfin, les tâches LO non allouées dans le slack time d'un serveur modal slackful ou agrégé sont exécutées dans des serveurs modaux slackless. Ces serveurs sont tels que que $\Gamma_{MS}(HI)$ est vide et $\Gamma_{MS}(LO)$ contient une seule tâche LO. Il y a donc autant de serveur modaux slackless qu'il y a de tâches LO non allouées.

Quoi qu'il en soit, agrégés ou non, les serveurs modaux ne disposent que d'une quantité limitée de slack time. Il se peut donc que ce slack time soit insuffisant pour correctement exécuter des tâches LO. C'est pourquoi nous avons besoin de test d'ordonnançabilité pour s'assurer avant l'exécution du système que les tâches LO exécutées dans le slack d'une tâche HI le seront correctement. Si une tâche passe avec succès un test d'ordonnancement celle–ci est dit allouée au serveur modal correspondant. Deux tests d'ordonnançabilité sont utilisés: un basé sur l'utilisation et l'autre sur les *demand bound function* et *supply bound function* [66].

Les particularités des systèmes à criticité mixte sont gérées par la politique d'ordonnancement des serveurs modaux. Ceux–ci ont des paramètres temporels constants, i.e. ils ne changent pas de paramètres temporels si un changement de mode se produit. Les serveurs modaux sont donc ordonnançables par des algorithmes d'ordonnancement classiques.

La détermination de quelles tâches LO sont exécutées dans le slack time de quelles tâches HI est effectuée hors ligne. Elle répond à un certains nombre d'objectifs de performance tout comme le choix de l'algorithme multi–processeur.

10.5.2 Efficacité de l'approche

Les performances d'un algorithme hiérarchique, comme le nôtre, dépendent des performances de chaque algorithme qui compose cette hiérarchie. Ainsi, les algorithmes utilisés pour former cette hiérarchie doivent présenter des performances qui soient en adéquation

© 2017 Romain GRATIA

avec nos objectifs, à savoir un nombre faible de préemptions engendrées et un grand taux de succès à l'ordonnancement.

Chaque algorithme composant la hiérarchie a un impact sur le nombre de préemptions engendrées. Dans le cas d'un algorithme hiérarchique, ce nombre est au plus égal à la somme des préemptions engendrées par chaque algorithme de la hiérarchie. En effet, si une préemption est engendrée par l'algorithme de premier niveau, alors il se peut que cela se traduise au second niveau par une préemption qui ne se serait pas produit si l'algorithme de ce niveau était utilisé seul. Le cas inverse peut aussi se produire. Une préemption peut être engendrée par l'algorithme du second niveau et ne l'aurait pas été si l'algorithme de premier niveau avait été utilisé seul.

Par ailleurs, le test d'ordonnançabilité de la hiérarchie se fait par l'application des tests d'ordonnançabilité de chaque niveau de la hiérarchie. Dans notre cas, un système à criticité mixte est ordonnançable si deux conditions sont remplies. Premièrement, un lot de tâches est ordonnançable dans un serveur si celui–ci passe avec succès le test d'ordonnançabilité associé à notre politique d'ordonnancement à criticité mixte. La seconde condition est que les serveurs modaux soient ordonnançables par la politique d'ordonnancement multi–processeur. Si les deux politiques d'ordonnancement ont des tests d'ordonnancement pessimistes, avec par exemple une limite d'utilisation basse, alors la hiérarchie d'ordonnancement aura de mauvaises performances.

Il résulte que le choix des algorithmes d'ordonnancement utilisés pour chaque niveau doit prendre en compte ces aspects. La prise en compte de ces contraintes nous a amené à choisir RUN [42] pour effectuer l'ordonnancement des serveurs modaux sur le multi–processeur. Nous avons par ailleurs choisi *Earliest Deadline First* (EDF) pour effectuer l'ordonnancement des tâches LO dans le slack time des tâches HI. En effet, ces deux algorithmes engendrent très peu de préemptions [60; 42] et sont tous les deux optimaux pour les mutli–processeurs et les monoprocesseurs respectivement. En particulier, un lot de serveurs modaux est ordonnançable sur m processeurs par RUN si son utilisation est inférieure ou égale à m.

Enfin, les performances de notre algorithme hiérarchique peuvent aussi être affectées par la répartition des tâches LO dans les serveurs modaux. Cette répartition consiste à déterminer quelles tâches LO sont exécutées dans le slack time de quelles tâches HI en allouant ces tâches LO aux serveurs modaux. Comme une tâche LO exécutée dans le slack time d'une tâche HI n'a pas besoin de budget temps d'exécution propre, plus il y a de tâches LO allouées dans le slack time, plus l'utilisation du lot de serveurs modaux est faible. Nous avons donc à résoudre un problème d'optimisation dont l'objectif est de maximiser le taux d'utilisation des tâches LO exécutées dans le slack time des tâches HI.

© 2017 Romain GRATIA

Task	Period	Criticality	C(LO)	C(HI)	U(LO)	U(HI)
τ_1	5	HI	1	3	0.2	0.6
τ_2	2	HI	0.5	1.5	0.25	0.75
τ_3	8	HI	0.8	3.2	0.1	0.4
τ_4	8	LO	3.2	3.2	0.4	0.4
τ_5	15	LO	3.75	3.75	0.25	0.25
τ_6	12	LO	2.4	2.4	0.2	0.2
τ_7	12	LO	2.4	2.4	0.2	0.2

Table 10.1: Exemple d'un lot de tâches à criticité mixte.

Ce problème d'optimisation est résolu par l'utilisation d'un algorithme évolutionnaire. En effet, le nombre de solutions possibles et la complexité des tests d'ordonnançabilité font que l'utilisation de méthode de résolution exacte est inefficace.

Exemple: application de l'algorithme évolutionnaire Maintenant que nous avons notre algorithme au complet, à savoir un politique d'ordonnancement à criticité mixte, une politique d'ordonnancement multi–processeur et un méthode pour trouver une bonne allocation des tâches LO dans les serveurs modaux. Nous pouvons appliquer notre approche sur le lot de tâches présenté dans le tableau 10.1. Le résultat de l'allocation suite à l'application de notre algorithme évolutionnaire donne un serveur modal dit slackful, un agrégé et un serveur modal slackless.

La tâche τ_4 est allouée au serveur modal slackful MS_2 quant aux tâches τ_5 et τ_6 elles sont allouées à l'agrégé formé avec MS_1 et MS_3. La tâche τ_7 est allouée à serveur modal slackless MS_4. Cette allocation permet de réduire l'utilisation du système de 2,8 à 1,95 et peut donc être ordonnancé sur 2 processeurs au lieu de 3. Remarquer, que MC–fluid [34] nécessite 3 processeurs pour ordonnancer ce lot de tâches.

Dans la section suivante nous évaluons d'une manière plus exhaustive les performances de GMC–RUN.

10.6 Évaluation de GMC–RUN

GMC–RUN a été évalué de deux manières différentes. D'abord d'une manière théorique, ce qui nous donne une indication des pires performances possibles de notre algorithme. En particulier, nous prouvons que GMC–RUN n'a jamais besoin de plus de processeurs pour ordonnancer un système à criticité mixte que si celui–ci est ordonnancé comme un système à criticité multiple. Puis en calculant le facteur speed–up, nous déterminons l'écart entre les performances de notre algorithme et un algorithme optimal clairvoyant. Cependant,

© 2017 Romain GRATIA

cet indicateur de performance est pessimiste puisque cet écart peut n'arriver que dans le cas de systèmes très particuliers et non représentatifs des systèmes pouvant être rencontrés dans l'industrie.

C'est pourquoi nous évaluons aussi expérimentalement notre algorithme afin d'avoir une idée sur les performances de notre algorithme pour des lots de tâches que nous considérons comme représentatifs.

10.6.1 Évaluation théorique

Nous rappelons d'abord un certain nombre de notations. $U_{\Gamma(HI)}(HI)$ est l'utilisation en mode HI des tâches HI du lot Γ. $U_{\Gamma(HI)}(LO)$ est l'utilisation en mode LO des tâches HI du lot Γ. $U_{\Gamma(LO)}(LO)$ est l'utilisation en mode LO des tâches LO du lot Γ. Enfin, un système à multiple criticités est ordonnançable par un algorithme d'ordonnancement multi–processeur optimal sur m processeurs si $U_{\Gamma(HI)}(HI) + U_{\Gamma(LO)}(LO) \leq m$.

L'évaluation théorique de GMC–RUN se fait en deux temps.

D'abord, nous donnons et prouvons la borne supérieure du nombre de processeurs requis pour ordonnancer un système à criticité mixte. Puis, nous calculons son facteur speed–up. Ce facteur donne une indication sur l'éloignement des performances de l'algorithme de celles d'un algorithme optimal clairvoyant.

Nombre de processeurs

Nous nous intéressons ici à déterminer si pour ordonnancer un système à criticité mixte, GMC–RUN peut avoir besoin de plus de processeurs que si le système est ordonnancé comme un système à criticité multiple. Le théorème suivant donne une borne supérieure et une borne inférieure sur le nombre de processeurs requis pour ordonnancer un système à criticité mixte avec GMC–RUN:

Theorem 15. *Le nombre de processeurs P requis pour ordonnancer un système à criticité mixte vérifie les inégalités suivantes:*

$$\lceil max(U_{\Gamma(HI)}(HI), U_{\Gamma(LO)}(LO) + U_{\Gamma(HI)}(LO)) \rceil \leq P \leq \lceil U_{\Gamma}(HI) \rceil \tag{10.2}$$

Remarquer que nous avons prouvé que GMC–RUN ne requiert jamais plus de processeurs que l'ordonnancement du système à criticité multiple ordonnancé par un algorithme d'ordonnancement temps réel optimal classique. Ce résultat est par ailleurs indépendant de la méthode choisie pour effectuer l'allocation des tâches LO dans les serveurs modaux.

© 2017 Romain GRATIA

Facteur speed–up

Le théorème suivant donne la valeur du facteur speed–up pour GMC–RUN:

Theorem 16. *Le facteur speed–up de GMC–RUN est 2.*

Ce facteur speed–up est le deuxième meilleur facteur speed–up parmi les algorithmes d'ordonnancement pour systèmes à criticité mixte. Elle est par ailleurs vraie pour n'importe quel algorithme optimal multi–processeur utilisé dans notre hiérarchie d'ordonnancement, et pas seulement pour RUN.

10.6.2 Évaluation expérimentale

Nous avons ensuite évalué expérimentalement GMC–RUN et l'avons comparé avec d'autres algorithmes d'ordonnancement à criticité mixte, à savoir MC–DP–Fair [34] et fpEDF–VD [53]. Cette évaluation se base sur deux critères. Le premier est le taux de succès d'ordonnançabilité. Le second le nombre de préemptions engendrées. Nous avons pour cela généré aléatoirement des lots de tâches. L'implémentation de notre algorithme évolutionnaire est basé sur le framework Distributed Evolutionary Algorithms pour Python (DEAP) [75].

La génération des lots de tâches est basée sur une méthode décrite dans [53]. Cette méthode permet de générer des lots de tâches à partir de différentes valeurs de période, taux d'utilisation et proportions de tâches HI.

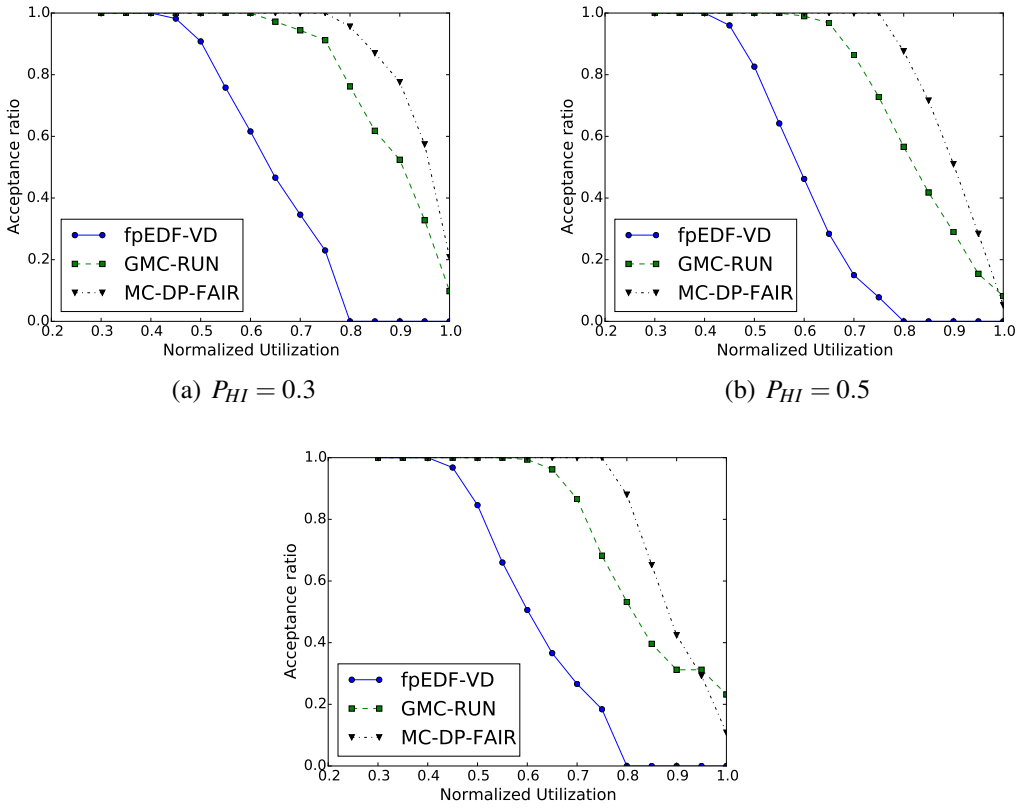
Le taux de succès d'ordonnançabilité

Le ratio d'acceptation est le pourcentage de lot de tâches considéré ordonnançable par un algorithme d'ordonnancement. Nous avons comparé les ratios d'acceptation de GMC–RUN, fpEDF–VD et MC–DP–Fair. La figure 10.3 montre le taux de succès d'ordonnançabilité pour différentes utilisations normalisées U_{Bound}/m pour 2 processeurs.

Les résultats montrent que GMC–RUN a de meilleures ratio d'acceptation que fpEDF–VD et qui sont proches de ceux de MC–DP–Fair, quelque soit la proportion de tâches HI. Par exemple, avec deux processeurs, 30% de tâches HI et une utilisation normalisée de 0.7, GMC–RUN ordonnance 95% des lots de tâches contre 100% pour MC–DP–Fair et moins de 40% pour fpEDF–VD. Cependant, GMC–RUN est meilleur que MC–DP–Fair pour des hautes utilisations et pour des une proportion de tâches HI égale à 70%.

Les meilleurs performances de GMC–RUN et MC–DP–Fair par rapport à celles de fpEDF–VD confirment notre intuition initiale que les meilleurs algorithmes d'ordonnancement

© 2017 Romain GRATIA



(a) $P_{HI} = 0.3$ (b) $P_{HI} = 0.5$

(c) $P_{HI} = 0.7$

Figure 10.3: Taux de succès d'ordonnançabilité pour des utilisation normalisées (U_{Bound}/m) pour différentes proportions de tâches HI sur 2 processeurs.

à criticité se base sur les meilleurs algorithmes d'ordonnancement classiques. En effet, GMC–RUN et MC–DP–Fair ont été conçus à partir d'algorithme d'ordonnancement multi–processeur optimaux classiques.

Les meilleurs résultats de GMC–RUN pour des proportions de tâches HI élevées peuvent s'expliquer par le fait que plus la proportion de tâches HI est élevée plus le taux d'utilisation du lot de tâches correspond à celui des tâches HI seules. Par ailleurs, une grande proportion de slack time devient disponible en mode LO, ce qui facilite l'allocation de tâches LO dans les serveurs modaux et donc de réduire l'utilisation du système.

Le nombre de préemptions engendrées

Le nombre de préemptions engendrées par un algorithme d'ordonnancement influence sa capacité à pouvoir ordonnancer ou non un lot de tâches. Le choix de RUN s'est fait en parti sur ce critère, et pour lequel il excelle en engendrant très peu de préemptions. Mais comme les serveurs modaux ont des budgets limités, leur utilisation peut découper

© 2017 Romain GRATIA

l'exécution des tâches qui leur sont allouées. Ainsi, leur utilisation peut engendrer des préemptions supplémentaires.

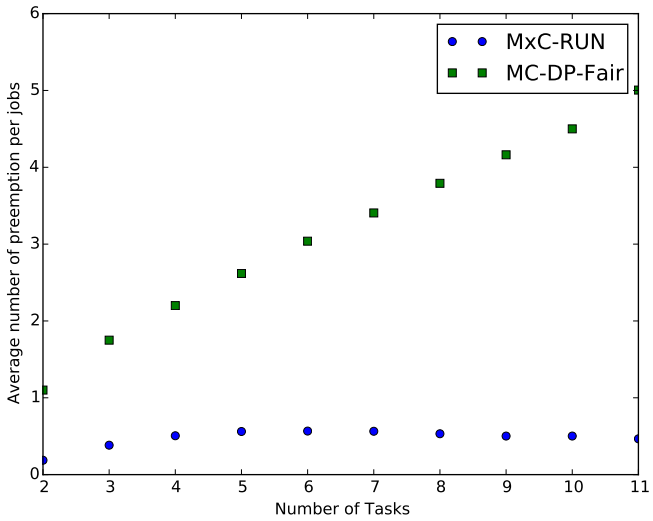


Figure 10.4: Nombre moyen de préemptions par job pour des lots constitués d'un nombre variable de tâches.

Nous avons comparé le nombre de préemptions engendrées par job pour GMC–RUN et MC–DP–Fair. Pour cela nous avons utilisé la même méthode de comptage que celle décrite dans le papier sur RUN: pour chaque job et en ignorant celles dues à une activation ou à la complétion d'un job. Nous n'avons effectué ce comptage que pour le mode LO, car le mode HI revient à ordonnancer les tâches HI avec RUN et donnerait des résultats similaires à ceux présentés dans [42] et qui montrent que RUN est bien meilleur qu'un algorithme similaire à DP–Fair. Les résultats sont présentés dans le figure 10.4.

Le nombre moyen de préemptions est reproduit en fonction du nombre de tâches. Comme attendu, GMC–RUN engendre en moyenne au moins cinq fois moins de préemptions que MC–DP–Fair. Par ailleurs, l'écart se creuse à mesure que le nombre de tâches augmente.

10.7 Ordonnancement des systèmes à criticité mixte avec plus de deux niveaux de criticité

Cependant, la plupart des algorithmes d'ordonnancement à criticité mixte ne se limite à l'ordonnancement que des systèmes avec deux niveaux de criticité. L'extension à plus de niveaux de criticité n'est souvent pas faite et pas forcément évidente. Or, l'utilisation

© 2017 Romain GRATIA

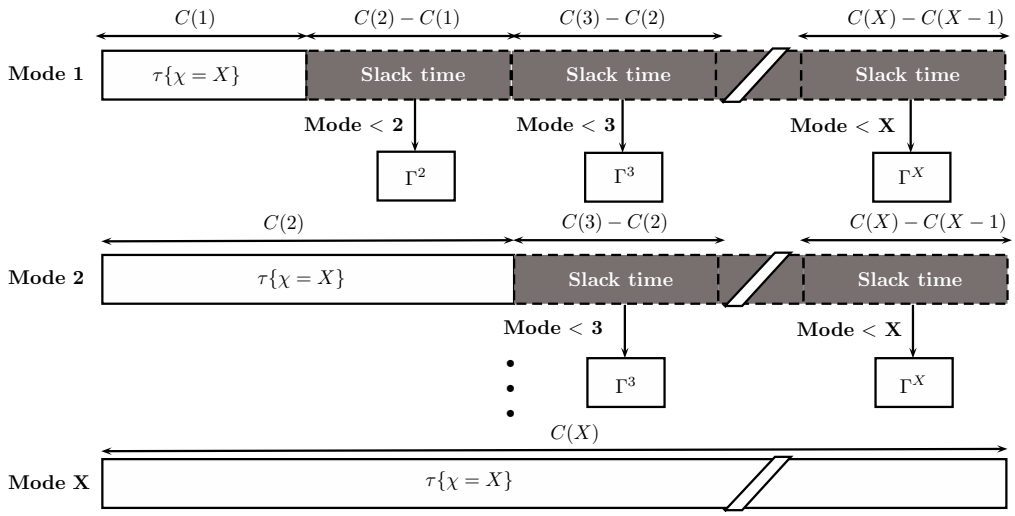


Figure 10.5: Slack time disponible dans chaque mode pour une tâche CL–X.

de plus de deux niveaux de criticité permet de gérer plus finement la dégradation de l'exécution des tâches de criticité moyenne. Dans notre cas, nous y parvenons en allouant les tâches dans les serveurs modaux par un processus itératif.

Les notations doivent tout d'abord être adaptées au cas des systèmes avec plus de deux niveaux de criticité. Nous nous basons pour ces systèmes sur une notation à base de chiffres. Pour une tâches de criticité X nous utilisons la notation suivante: tâche CL–X. Par exemple, une tâche de criticité 2 est notée tâche CL–2. Plus X est grand, plus la tâche est critique. De la même manière, nous notons mode X, le mode dans lequel toutes les tâches de criticité plus grande ou égale à X sont exécutées avec leur budget correspondant à la criticité X.

Lorsqu'une tâche de criticité X est exécutée avec son budget de mode X, même dans les modes inférieurs à X, alors la tâche n'utilise pas tout son budget et il y a alors du slack time. Ceci est représenté dans la figure 10.5. Ce slack time disponible est d'autant plus important que le mode est bas. Nous pouvons donc de la même que pour les systèmes avec deux niveaux de criticité utiliser ce slack time pour exécuter d'autres tâches et ce en utilisant des serveurs modaux. La différence ici est qu'une tâche de criticité plus grande que 2 peut encore avoir du slack time de disponible après la détection d'une TFE. Après une seule TFE, seulement le slack time correspondant à la différence entre les budgets des modes 2 et 1 n'est plus disponible. Par exemple, dans la figure 10.5, lorsque le système passe du mode 2 vers le mode 3, le slack time qui est réutilisé pour exécuter la tâche CL–X est égal à $C(3) - C(2)$. On dit alors que ce slack time est de mode 3, comme il est utilisé pour exécuter la tâche de plus haute criticité quand le mode 3 devient actif. Par conséquent, une tâche CL–X peut utiliser son slack time pour ordonnancer $X - 1$ lots de

© 2017 Romain GRATIA

tâches de plus basses criticités Γ^l, avec $l \in [2;X]$. Cela en fait un pour chaque mode, du mode 2 au mode X.

Chaque lot Γ^l est ordonnancé dans le slack time égal à la différence des budgets de deux modes consécutifs. Dans la figure 10.5, le lot Γ^3 est ordonnancé dans le slack time de mode 3. Ce slack time de mode 3 est utilisé dans les modes 1 et 2 pour exécuter des tâches de plus basses criticités. Mais en mode 3, il doit servir à terminer l'exécution de la tâche CL–X. Un serveur modal doit donc maintenant gérer plusieurs lots de tâches.

Pour déterminer la criticité des tâches qui peuvent être ordonnancées dans chacun de ces lots, nous examinons quand le slack time est utilisé pour exécuter la tâche de plus haute criticité. Un lot de tâches de plus basses criticités ne doit pas être ordonnancé si une tâche CL–X a besoin de compléter son exécution en utilisant le slack time. Ceci arrive quand une TFE se produit et un mode plus élevé devient actif. Ainsi, chacun de ces lots de tâches ne doit contenir que des tâches qui n'ont plus à être ordonnancées lorsque la tâche CL–X doit utiliser le slack time pour compléter son exécution. Dans le cas du slack time dont la tâche CL–X a besoin quand le mode 3 devient actif, seules les tâches avec une criticité strictement inférieure à 3 peuvent y être exécutées. En effet, lorsque la tâche CL–X a besoin de ce slack time de mode 3, elle peut l'utiliser sans empêcher d'autres tâches de s'exécuter. Afin de toujours respecter cette règle, nous avons conçu un processus d'allocation itératif.

Afin d'assurer que chaque slack time de mode M+1 n'a à exécuter que des tâches avec une criticité strictement inférieure à M+1, nous séparons les tâches selon leur criticité et formons deux groupes de tâches comme représenté dans la figure 10.6. Le premier groupe, dans la partie inférieure de la figure, est composé de tâches de criticité strictement inférieure à M+1. Le second groupe, dans la partie supérieure de la figure, contient toutes les tâches de criticité supérieure ou égale à M+1. Une fois ces deux groupes formés, nous répartissons les tâches du premier groupe dans les serveurs modaux formés à partir des tâches du second groupe. Cette répartition vise à exécuter les tâches de plus basse criticité dans le slack time de mode M+1 des serveurs modaux. Dans la figure 10.6, les tâches de plus basses criticité sont allouées dans le slack time de mode M+1 dont le budget est égal à $C(M+1) - C(M)$. Une fois cette répartition terminée, les tâches de basse criticité allouées sont dans le lot de tâches correspondant à ce slack time de mode M+1.

Tous les niveaux de criticité sont ensuite considérés en répétant ces opérations de formation des deux groupes et de l'allocation des tâches du premier groupe dans les serveurs modaux formés à partir des tâches du second groupe. Chaque étape se termine par la réduction d'un niveau de criticité grâce à l'allocation des tâches dans les serveurs modaux (slackful, agrégés ou slackless). Une fois tous les niveaux de criticité traités, le résultat

© 2017 Romain GRATIA

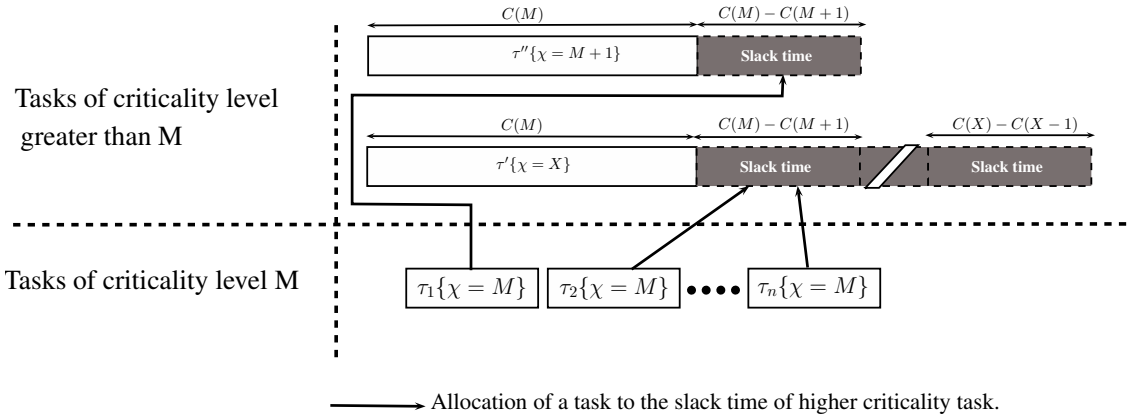


Figure 10.6: Allocation des tâches CL–L dans le slack time de tâches plus critiques.

est un lot de serveurs modaux ordonnançable par un algorithme d'ordonnancement temps réel classique comme dans le cas des systèmes à criticité mixte avec deux niveaux.

L'ordonnancement dans les serveurs modaux suit la même logique que dans le cas des systèmes avec deux niveaux. Chaque serveur modal exécute d'abord la tâche avec la plus haute criticité, et à partir de laquelle il a été crée, puis le serveur modal ordonnance les lot de tâches moins critiques.

10.8 Ordonnancement des tâches élastiques

Lorsqu'une TFE se produit, l'exécution des tâches LO est dégradée. La dégradation subie dépend du modèle de dégradation utilisé. Le plus couramment utilisé parmi les algorithmes d'ordonnancement à criticité mixte est ce qu'on appelle le modèle à annulation. Dans ce modèle, les tâches LO sont totalement arrêtées après un changement de mode, ce qui rend ce modèle peu intéressant pour des applications industrielles. En effet, nos partenaires industriels ont besoin de pouvoir assurer une exécution minimale pour des tâches de plus basse criticité. C'est pourquoi, nous considérons l'utilisation d'un autre modèle de dégradation de l'exécution des tâches LO, appelé le modèle des tâches élastiques. Dans ce modèle, les tâches LO ne sont pas totalement arrêtées mais leur exécution est ralentie par l'utilisation d'une période et d'une échéance plus grandes comme représenté dans la figure 10.7. Dans cette figure, une tâche LO est exécutée C unités de temps avec une périodicité T(LO) en mode LO. Quand le mode HI devient actif cette même tâche est toujours exécutée avec un budget C mais avec une périodicité T(HI) = 2·T(LO). Cependant, du fait de l'exécution des tâches LO en mode HI, l'utilisation des serveurs modaux requiert ne peut pas se faire telle quelle.

© 2017 Romain GRATIA

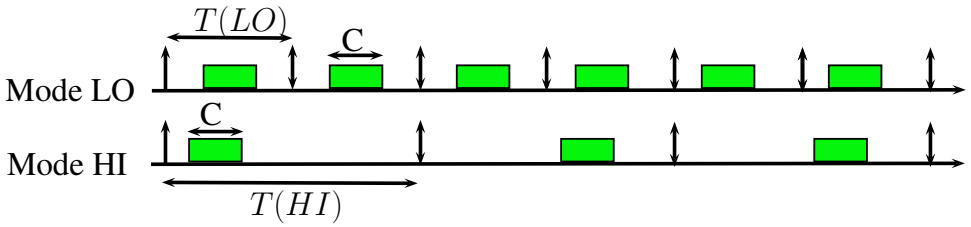


Figure 10.7: Exécution d'une tâche élastique dans les différents modes.

En mode HI, les tâches LO élastiques reçoivent une exécution minimale. En mode LO, elles sont exécutées d'une manière nominale. Un changement de mode se traduit donc pour ces tâches par une réduction des besoins d'exécution des tâches LO. Afin de pouvoir réutiliser notre approche basée sur les serveurs modaux, nous calculons les besoins d'exécution minimaux du mode HI et les besoins supplémentaires requis en mode LO comparés à ceux du mode HI. Ces besoins d'exécution supplémentaires, seulement nécessaires en mode LO, peuvent alors être fournis par le slack time de serveur modaux.

Ceci nécessite de transformer le modèle de la tâche élastique. Pour cela, au lieu d'exécuter une tâche LO avec une période plus petite et un même budget, nous cherchons à exécuter la tâche LO avec une même période mais un budget plus petit en mode HI. Nous transformons donc un changement de périodes en un changement de budgets.

En faisant cela, un tâche LO élastique τ peut être décomposée en deux sous tâches comme représenté dans la figure 10.8. Ces deux sous tâches sont deux serveurs qui sont utilisés pour exécuter la tâche LO élastique chacun avec les objectifs d'ordonnancement suivants:

1. La sous tâche τ^{NMC} fournit le temps d'exécution requis en mode HI. Il assure que la tâche LO élastique reçoit un budget C avec une périodicité T(HI) en mode HI. Elle est appelée τ^{NMC}, NMC signifiant *Non Mixed–Criticality*, étant donné qu'elle est toujours exécutée.

2. La sous tâche τ^{DIS} vient en supplément de la tâche NMC et fournit le temps d'exécution additionnel en mode LO afin d'assurer l'exécution de la tâche LO en mode LO. Elle est notée DIS pour *discarding*, car elle est arrêtée après un changement de mode.

Nous avons enfin besoin de déterminer les paramètres temporels des deux sous tâches. Ces paramètres temporels doivent assurer l'ordonnancement correct de la tâche LO dans les deux modes. Pour cela, nous fixons la période de la sous tâche τ^{NMC} égal au plus grand multiple commun des périodes du mode LO et HI de la tâche LO, tandis que la période de la tâche τ^{DIS} est égale à sa période en mode LO. Ensuite, nous commençons par calculer le budget de la tâche τ^{DIS}. Son budget est calculé de sorte à ce que la sous tâche τ^{DIS} soit

© 2017 Romain GRATIA

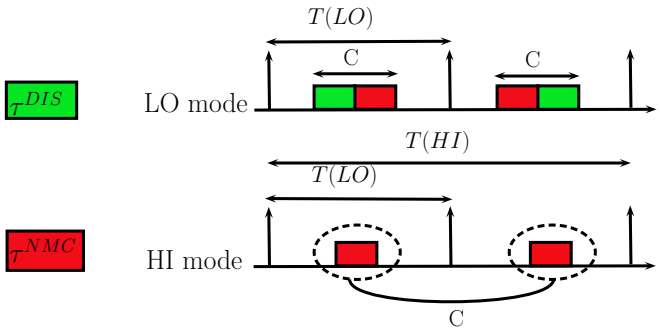


Figure 10.8: Décomposition de l'exécution d'une tâche élastique dans les différents modes.

ordonnançable dans le slack time d'un lot de serveurs modaux. Pour cela, ce calcul est effectué en même temps que l'allocation des sous tâches τ^{DIS} dans les serveurs modaux. Puis nous calculons le budget de la sous tâches τ^{NMC} de telle sorte que l'ordonnancement de la tâche LO en mode HI soit assuré et que l'exécution combinée des deux sous tâches τ^{DIS} et τ^{NMC} assure l'ordonnancement de la tâche LO. Pour cela, on s'assure que les deux sous tâches ne sont jamais exécutées en parallèle.

10.9 Conclusion

Dans cette thèse, nous avons proposé une approche versatile et efficace pour l'ordonnancement des systèmes à criticité mixte. Ces systèmes sont composés de tâches avec différentes criticité et peuvent être ordonnancées efficacement en acceptant d'éventuellement dégrader l'exécution des tâches les moins critiques.

Il existe de nombreux modèles de dégradation. Dans cette thèse nous en avons étudié deux. Le plus couramment utilisé est le modèle dit de l'annulation qui consiste à arrêté complètement les tâches LO après un changement de mode. Nous l'avons d'abord considéré pour des systèmes avec deux niveaux de criticité puis l'avons ensuite étudié pour des systèmes constitués de plus de deux niveaux. L'utilisation de plus de niveaux de criticité permettant de gérer l'exécution des tâches d'une manière plus fine, car il permet de dégrader l'exécution des tâches d'une manière plus progressive. Le second modèle de dégradation étudié est le modèle de la tâche élastique avec deux niveaux de criticité. Avec ce modèle, l'exécution des tâches est dégradée moins définitivement en augmentant leur périodicité plus qu'en les arrêtant totalement.

Nos contributions permettent l'ordonnancement de systèmes à criticité mixte d'une manière efficace tout en donnant aux concepteurs une marge de manœuvre pour ajuster l'exécution des tâches à leur besoin. Les contributions de cette thèse sont les suivantes:

© 2017 Romain GRATIA

- Introduction d'un nouveau type de serveur d'exécution appelé serveur modal pour les systèmes à criticité mixte. Ces serveurs ont d'abord été appliqués aux systèmes avec deux niveaux de criticité composé de tâches basées sur le modèle de l'annulation. Ces serveurs appliquent une politique d'ordonnancement à criticité mixte monoprocesseur qui tire profit du slack time des tâches HI pour ordonnancer des tâches LO.

- Ces serveurs modaux sont ensuite ordonnancés par un algorithme d'ordonnancement multi–processeur classique, qui dans notre cas est RUN. Les serveurs et cet algorithme forment un algorithme d'ordonnancement dit hiérarchique que nous avons appelé GMC–RUN. Les performances de GMC–RUN dépendent de la résolution efficace du problème d'allocation des tâches LO dans les serveurs modaux. Ce problème d'optimisation a été résolu grâce à l'utilisation d'un algorithme évolutionnaire.

- Nous avons ensuite évalué GMC–RUN. D'abord théoriquement en calculant son facteur speed–up que nous avons prouvé être de 2. Ensuite expérimentalement sur des lots de tâches générés aléatoirement. Nous avons mesuré le taux de succès d'ordonnançabilité et le nombre de préemptions moyen par job. Il se trouve que sur le premier critère GMC–RUN présente des performances proches de celles de MC–DP–Fair mais engendre beaucoup moins de préemptions que ce dernier.

- Nous avons ensuite présenté un processus itératif simple pour ordonnancer des systèmes avec plus de deux niveaux de criticité.

- Enfin nous avons présenté une méthode pour permettre l'ordonnancement de tâches élastiques avec GMC–RUN. Cette méthode consiste à décomposer l'exécution des tâches LO en deux sous tâches. Le calcul des budgets de ces sous tâches tirant profit du slack time disponible d'un lot de serveurs modaux donné. Par ailleurs elle est la première approche pour ces tâches sur multi–processeur permettant d'assurer l'exécution du système dans le mode LO.

© 2017 Romain GRATIA

List of Publications

[GRP14] Romain Gratia, Thomas Robert, and Laurent Pautet. Adaptation of run to mixed-criticality systems. In *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing, JRWRTC'2014*, pages 25–28, 2014.

[GRP15a] Romain Gratia, Thomas Robert, and Laurent Pautet. Generalized mixed-criticality scheduling based on RUN. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 267–276, 2015.

[GRP15b] Romain Gratia, Thomas Robert, and Laurent Pautet. Scheduling of mixed-criticality systems with RUN. In *20th IEEE Conference on Emerging Technologies & Factory Automation, ETFA 2015, Luxembourg, September 8-11, 2015*, pages 1–8, 2015.

© 2017 Romain GRATIA

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, pp. 11–33, Jan. 2004.

[2] R. W. Stence, "Digital by-wire replaces mechanical systems in cars," in *SAE Technical Paper*, SAE International, 10 2004.

[3] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007.

[4] M. Broy, "Automotive software and systems engineering," in *Formal Methods and Models for Co-Design, 2005. MEMOCODE'05. Proceedings. Third ACM and IEEE International Conference on*, pp. 143–149, 2005.

[5] A. Zahir and P. Palmieri, "Osek/vdx-operating systems for automotive applications," in *OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar*, pp. 4–1, IET, 1998.

[6] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A Definition and Classification of Timing Anomalies," in *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)* (F. Mueller, ed.), vol. 4 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006.

[7] H. Shah, K. Huang, and A. Knoll, "Timing anomalies in multi-core architectures due to the interference on the shared resources," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 708–713, Jan 2014.

[8] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem overview of methods and

survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[9] A. Blin, C. Courtaud, J. Sopena, J. Lawall, and G. Muller, "Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System," in *28th EUROMICRO Conference on Real-Time Systems (ECRTS'16)*, (Toulouse, France), July 2016.

[10] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, (Berkeley, CA, USA), pp. 18:1–18:18, USENIX Association, 2007.

[11] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Dependable Computing Conference (EDCC), 2012 Ninth European*, pp. 132–143, May 2012.

[12] J. Rushby, "Partitioning for safety and security: Requirements, mechanisms, and assurance," NASA Contractor Report CR-1999-209347, NASA Langley Research Center, june 1999. Also to be issued by the FAA.

[13] P. J. Prisaznuk, "Arinc 653 role in integrated modular avionics (ima)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pp. 1–E, IEEE, 2008.

[14] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[15] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, "Ensuring robust partitioning in multicore platforms for ima systems," in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pp. 7A4–1–7A4–9, Oct 2012.

[16] S. 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*, pp. 239–243, Dec 2007.

[17] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "How realistic is the mixed-criticality real-time system model?," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, (New York, NY, USA), pp. 139–148, ACM, 2015.

© 2017 Romain GRATIA

[18] J. Real and A. Crespo, “Mode change protocols for real-time systems: A survey and a new proposal,” *Real-Time Syst.*, vol. 26, pp. 161–197, Mar. 2004.

[19] G. C. Buttazzo, G. Lipari, and L. Abeni, “Elastic task model for adaptive rate control,” in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pp. 286–295, Dec 1998.

[20] F. Santy, L. George, P. Thierry, and J. Goossens, “Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp,” in *2012 24th Euromicro Conference on Real-Time Systems*, pp. 155–165, July 2012.

[21] F. Santy, G. Raravi, G. Nelissen, V. Nelis, P. Kumar, J. Goossens, and E. Tovar, “Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems,” in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS ’13, (New York, NY, USA), pp. 183–192, ACM, 2013.

[22] I. Bate, A. Burns, and R. I. Davis, “A bailout protocol for mixed criticality systems,” in *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, ECRTS ’15, (Washington, DC, USA), pp. 259–268, IEEE Computer Society, 2015.

[23] C. L. L. Sudarshan K. Dhall, “On a real-time scheduling problem,” *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[24] L. George, P. Courbin, and Y. Sorel, “Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling,” *Journal of Systems Architecture*, vol. 57, no. 5, pp. 518 – 535, 2011. Special Issue on Multiprocessor Real-time Scheduling.

[25] S. Funk and S. K. Baruah, “Restricting edf migration on uniform heterogeneous multiprocessors,” 2005.

[26] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, “Litmus rt : A testbed for empirically comparing real-time multiprocessor schedulers,” in *2006 27th IEEE International Real-Time Systems Symposium (RTSS’06)*, pp. 111–126, Dec 2006.

[27] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, “On the scalability of real-time scheduling algorithms on multicore platforms: A case study,” in *Real-Time Systems Symposium, 2008*, pp. 157–169, Nov 2008.

© 2017 Romain GRATIA

[28] B. B. Brandenburg and J. H. Anderson, "On the implementation of global real-time schedulers," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, (Washington, DC, USA), pp. 214–224, IEEE Computer Society, 2009.

[29] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Is semi-partitioned scheduling practical?," in *2011 23rd Euromicro Conference on Real-Time Systems*, pp. 125–135, July 2011.

[30] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM*, vol. 47, pp. 617–643, July 2000.

[31] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, pp. 35:1–35:44, Oct. 2011.

[32] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "Rtos support for multicore mixed-criticality systems," in *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, (Washington, DC, USA), pp. 197–208, IEEE Computer Society, 2012.

[33] P. Rodriguez, L. George, Y. Abdeddaïm, and J. Goossens, "Multicriteria evaluation of partitioned edf-vd for mixed-criticality systems upon identical processors," in *1st Workshop on Mixed Criticality Systems (WMC), IEEE Real-Time Systems Symposium*, pp. 49–54, December 2013.

[34] J. Lee, K. M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "Mc-fluid: Fluid model-based mixed-criticality scheduling on multiprocessors," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pp. 41–52, Dec 2014.

[35] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *2010 31st IEEE Real-Time Systems Symposium*, pp. 14–24, Nov 2010.

[36] D. Compagnin, E. Mezzetti, and T. Vardanega, "Putting run into practice: Implementation and evaluation," in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 75–84, July 2014.

[37] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.

© 2017 Romain GRATIA

[38] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pp. 193–202, Dec 2001.

[39] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, pp. 322–334, 2006.

[40] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, (Washington, DC, USA), pp. 447–456, IEEE Computer Society, 2009.

[41] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, (New York, NY, USA), pp. 345–354, ACM, 1993.

[42] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 104–115, Nov 2011.

[43] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, "U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *2012 24th Euromicro Conference on Real-Time Systems*, pp. 13–23, July 2012.

[44] S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Transactions on Computers*, vol. 53, pp. 781–784, June 2004.

[45] A. Burns and R. Davis, "Mixed criticality systems: a review.." www-users.cs.york.ac.uk/ burns/review.pdf .

[46] K. Lakshmanan, D. d. Niz, R. R. Rajkumar, and G. Moreno, "Resource allocation in distributed mixed-criticality cyber-physical systems," in *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, (Washington, DC, USA), pp. 169–178, IEEE Computer Society, 2010.

[47] D. d. Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 291–300, Dec 2009.

© 2017 Romain GRATIA

[48] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[49] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *2012 24th Euromicro Conference on Real-Time Systems*, pp. 145–154, July 2012.

[50] H. Xu and A. Burns, "Semi-partitioned model for dual-core mixed criticality system," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, (New York, NY, USA), pp. 257–266, ACM, 2015.

[51] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 34–43, Nov 2011.

[52] K. Bletsas and S. M. Petters, "Using nps-f for mixed-criticality multicore systems," *SIGBED Rev.*, vol. 10, pp. 36–36, July 2013.

[53] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *ECRTS'2012*, pp. 166–175, July 2012.

[54] S. Baruah, A. Eswaran, and Z. Guo, "Mc-fluid: Simplified and optimally quantified," in *Real-Time Systems Symposium, 2015 IEEE*, pp. 327–337, Dec 2015.

[55] S. Ramanathan and A. Easwaran, "Mc-fluid: rate assignment strategies," in *3rd Workshop on Mixed Criticality Systems (WMC), IEEE Real-Time Systems Symposium*, pp. 6–11, December 2015.

[56] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, (Washington, DC, USA), pp. 1864–1871, IEEE Computer Society, 2010.

[57] H. Su, D. Zhu, and D. Mossé, "Scheduling algorithms for elastic mixed-criticality tasks in multicore systems," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 352–357, Aug 2013.

[58] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 147–152, March 2013.

© 2017 Romain GRATIA

[59] Z. Al-bayati, Q. Zhao, A. Youssef, H. Zeng, and Z. Gu, "Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms," in *The 20th Asia and South Pacific Design Automation Conference*, pp. 630–635, Jan 2015.

[60] G. C. Buttazzo, "Rate monotonic vs. edf: Judgment day," *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.

[61] P. Holman and J. H. Anderson, "Adapting pfair scheduling for symmetric multiprocessors," *J. Embedded Comput.*, vol. 1, pp. 543–564, Dec. 2005.

[62] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Transactions on Computers*, vol. 61, pp. 1140–1152, Aug 2012.

[63] A. Burns, "An augmented model for mixed criticality," *Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121)*, vol. 5, no. 3, 2015.

[64] S. Baruah and Z. Guo, "Mixed-criticality job models: a comparison," in *3rd Workshop on Mixed Criticality Systems (WMC), IEEE Real-Time Systems Symposium*, pp. 1–5, december 2015.

[65] D. Tămaş–Selicean, *Design of Mixed-Criticality Applications on Distributed Real-Time Systems*. PhD thesis, 2014.

[66] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pp. 2–13, Dec 2003.

[67] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pp. 10 pp.–398, Dec 2005.

[68] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for some practical problems in prioritized preemptive scheduling," in *Proceedings of the 7th IEEE Real-Time Systems Symposium (RTSS'86), December 2-4, 1986, New Orleans, Louisiana, USA*, pp. 181–191, 1986.

[69] J. Lee, L. T. X. Phan, S. Chen, O. Sokolsky, and I. Lee, "Improving resource utilization for compositional scheduling using dprm interfaces," *SIGBED Rev.*, vol. 8, pp. 38–45, Mar. 2011.

© 2017 Romain GRATIA

[70] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Systems*, vol. 47, no. 5, pp. 389–429, 2011.

[71] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez, "Metaheuristic optimization frameworks: a survey and benchmarking," *Soft Computing*, vol. 16, no. 3, pp. 527–561, 2012.

[72] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[73] S. Luke, *Essentials of Metaheuristics*. Lulu, second ed., 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[74] J. Dréo, A. Petrowski, P. Siarry, and E. Taillard, *Métaheuristiques pour l'optimisation difficile*. Eyrolles, first ed., 2003.

[75] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[76] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS'2010*, pp. 6–11, July 2010.

[77] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability."

[78] J. W. Liu, K.-J. Lin, W.-K. Shih, A. C.-s. Yu, J.-Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *Computer*, vol. 24, no. 5, pp. 58–68, 1991.

[79] J. Theis, G. Fohler, and S. Baruah, "Schedule table generation for time-triggered mixed criticality systems," in *1st Workshop on Mixed Criticality Systems (WMC), IEEE Real-Time Systems Symposium*, pp. 79–84, December 2013.

[80] M. Jan, L. Zaourar, and M. Pitel, "Maximizing the execution rate of low-criticality tasks in mixed criticality systems," in *1st Workshop on Mixed Criticality Systems (WMC), IEEE Real-Time Systems Symposium)*, pp. 43–48, Dec. 2013.

[81] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *1st Workshop on Mixed Criticality Systems (WMC), IEEE Real-Time Systems Symposium*, pp. 1–6, december 2013.

© 2017 Romain GRATIA

[82] S. Baruah, A. Burns, and Z. Guo, “Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors,” in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 131–138, July 2016.

[83] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, “The rosace case study: From simulink specification to multi/many-core execution,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 309–318, April 2014.

[84] S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis, “Static probabilistic timing analysis for real-time systems using random replacement caches,” vol. 51, pp. 77–123, 2015.

[85] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, “Proartis: Probabilistically analyzable real-time systems,” vol. 12, (New York, NY, USA), pp. 94:1–94:26, ACM, May 2013.

© 2017 Romain GRATIA

Une Approche Efficace et Polyvalente pour l'Ordonnancement de Systèmes à Criticité Mixte sur Processeur Multi-Coeurs

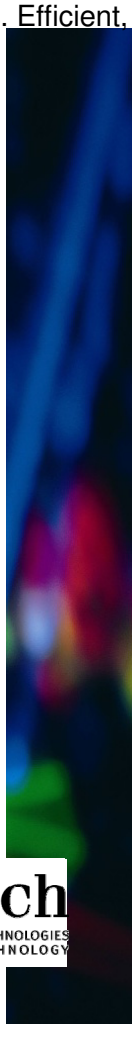
Romain Gratia

RÉSUMÉ : Dans cette thèse, nous nous intéressons à l'ordonnancement des systèmes à criticité mixte sur des multi-processeurs. Dans ces systèmes des applications de différents niveaux de criticité sont exécutées sur une même plate-forme d'exécution. Ces niveaux de criticité indiquent l'importance de chaque application pour le fonctionnement sûr du système et auxquels sont associées des exigences de sûreté de fonctionnement. Le respect de ces exigences est actuellement assuré par l'emploi de méthodes de conception, qui sont connues pour leur faible efficacité en particulier pour les multi-processeurs. Nous proposons donc dans cette thèse une approche efficace et générique d'ordonnancement des systèmes à criticité mixte sur multi-processeurs. Efficace, car les comparaisons avec les solutions existantes ont montré que notre approche permet d'ordonnancer plus de systèmes tout en générant très peu de préemptions. Nous parvenons à ces résultats en tirant profit de la différence de temps d'exécution entre niveaux de criticité que nous utilisons pour exécuter plus d'applications critiques ou non critiques selon le comportement du système. Générique, car elle permet d'ordonnancer des systèmes à criticité mixte composés d'un nombre quelconque de niveaux de criticité ainsi que deux modèles de systèmes à criticité mixte différents. Cette généricité nous permet d'adapter notre approche aux objectifs de disponibilité de chaque application et ainsi de mieux répondre aux besoins industriels.

MOTS-CLEFS : Ordonnancement temps réel, multi-processeurs, systèmes à criticité mixte.

ABSTRACT : This thesis focuses on the scheduling of mixed-criticality systems on multi-processors. In these systems, several applications with different criticality levels are executed on a same execution platform. The criticality levels denote the importance of the applications for the correct execution of the system. To each criticality level is associated safety requirements. These safety requirements are currently satisfied by using methods known to be inefficient, particularly for multi-processor platforms. We therefore propose in this thesis an efficient and versatile approach to schedule these mixed-criticality systems on multi-processors. Efficient, as our experiments and comparisons with existing solutions show that our approach schedules more systems while entailing fewer preemptions. This is achieved by taking advantage of the difference of execution times between criticality levels to either execute more the non critical or the critical applications depending on the behaviour of the system. Versatile, since our approach schedules mixed-criticality systems composed of any number of criticality levels and two different mixed-criticality task models. Thanks to this versatility, our approach can be adapted to meet the availability objectives of the applications and thus better meet the industrial needs.

KEY-WORDS : real–time scheduling, multi-processors, mixed–criticality systems.



TELECOM ParisTech

EDITE

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY