Deployment of mixed criticality and data driven systems on multi-cores architectures

Roberto Medina

université PARIS-SACLAY

NNT : 2019SACLT004

Thèse de doctorat

Déploiement de Systèmes à Flots de Données en Criticité Mixte pour Architectures Multi-cœurs

Deployment of Mixed-Criticality and Data-Driven Systems on Multi-core Architectures

Thèse de doctorat de l'Université Paris-Saclay
préparée à TELECOM ParisTech

École doctorale n°580 : Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Programmation : Modèles, Algorithmes, Langages, Architecture

Thèse présentée et soutenue à Paris, le 30 Janvier 2019, par

Roberto MEDINA

Composition du Jury :

Alix MUNIER-KORDON Professeure, Sorbonne Université (LIP6)	Présidente
Liliana CUCU-GROSJEAN Chargée de Recherche, INRIA de Paris (Kopernic)	Rapporteuse
Laurent GEORGE Professeur, ESIEE (LIGM)	Rapporteur
Arvind EASWARAN Maître de conférences, Nanyang Technological University	Examinateur
Eric GOUBAULT Professeur, Ecole Polytechnique (LIX)	Examinateur
Emmanuel LEDINOT Responsable recherche et technologie, Dassault Aviation	Examinateur
Isabelle PUAUT Professeure, Université Rennes 1 (IRISA)	Examinatrice
Laurent PAUTET Professeur, TELECOM ParisTech (LTCI)	Directeur de thèse
Etienne BORDE Maître de conférences, TELECOM ParisTech (LTCI)	Co-Encadrant de thèse

Acknowledgments

First and foremost, I would like to thank my advisors Laurent Pautet and Etienne Borde. I feel really lucky to have worked with them during these past 3 years and 4 months. Laurent was the person that introduced me to real-time systems and the interesting complex problems that the community is facing. His insight was always helpful to identify and tackle challenging difficulties during this research. Etienne was an excellent supervisor. His constant encouragement, feedback and trust in my research kept me motivated and allowed me to complete this work. I am thankful to both of them for putting their trust in me and allowing me to pursue my career ambition of preparing a PhD.

Besides my advisors, I would like to thank Liliana Cucu-Grosjean and Laurent George for accepting to examine my manuscript. Their feedback was really helpful and valuable. I extend my appreciation to the rest of my PhD committee members Arvind Easwaran, Eric Goubault, Emmanuel Ledinot, Alix Munier-Kordon and Isabelle Puaut for their interest in my research.

Then I would like to thank the people I met at TELECOM ParisTech during these past few years. Especially, I want to thank my former colleagues Dragutin B., Robin D., Romain G., Mohamed Tahar H., Jad K., Jean-Philippe M., Edouard R., and Kristen V., for all the conversations we had (even if it was just to chat about anime, films, TV series or video games).

Heartfelt thanks go to my friends for their unconditional support and friendship. I especially want to thank Alexis, Anthony, Alexandre, Gabriela, Irene, Nathalie and Paulina for sticking with me and always supporting me.

Finally, I would like to thank my parents, Dora and Hernán, and my sister Antonella for their unconditional support, love and care.

Résumé

De nos jours, la conception de systèmes critiques va de plus en plus vers l'intégration de différents composants système sur une unique plate-forme de calcul. Les systèmes à criticité mixte permettent aux composants critiques ayant un degré élevé de confiance (c.-à-d. une faible probabilité de défaillance) de partager des ressources de calcul avec des composants moins critiques sans nécessiter des mécanismes lourds d'isolation logicielle.

Traditionnellement, les systèmes critiques sont conçus à l'aide de modèles de calcul comme les graphes data-flow et l'ordonnancement temps-réel pour fournir un comportement logique et temporel correct. Néanmoins, les ressources allouées aux data-flows et aux ordonnanceurs temps-réel sont fondées sur l'analyse du pire cas, ce qui conduit souvent à une sous-utilisation des processeurs. Les ressources allouées ne sont ainsi pas toujours entièrement utilisées. Cette sous-utilisation devient plus remarquable sur les architectures multi-cœurs où la différence entre le meilleur et le pire cas est encore plus significative.

Le modèle d'exécution à criticité mixte propose une solution au problème susmentionné. Afin d'allouer efficacement les ressources tout en assurant une exécution correcte des composants critiques, les ressources sont allouées en fonction du mode opérationnel du système. Tant que des capacités de calcul suffisantes sont disponibles pour respecter toutes les échéances, le système est dans un mode opérationnel de « basse criticité ». Cependant, si la charge du système augmente, les composants critiques sont priorisés pour respecter leurs échéances, leurs ressources de calcul augmentent et les composants moins/non critiques sont pénalisés. Le système passe alors à un mode opérationnel de « haute criticité ».

L' intégration des aspects de criticité mixte dans le modèle data-flow est néanmoins un problème difficile à résoudre. Des nouvelles méthodes d'ordonnancement capables de gérer des contraintes de précédences et des variations sur les budgets de temps doivent être définies.

Bien que plusieurs contributions sur l'ordonnancement à criticité mixte aient été proposées, l'ordonnancement avec contraintes de précédences sur multi-processeurs a rarement été étudié. Les méthodes existantes conduisent à une sous-utilisation des ressources, ce qui contredit l'objectif principal de la criticité mixte. Pour cette raison, nous définissons des nouvelles méthodes d'ordonnancement efficaces basées sur une méta-heuristique produisant des tables d'ordonnancement pour chaque mode opérationnel du système. Ces

tables sont correctes : lorsque la charge du système augmente, les composants critiques ne manqueront jamais leurs échéances. Deux implémentations basées sur des algorithmes globaux préemptifs démontrent un gain significatif en ordonnançabilité et en utilisation des ressources : plus de 60 % de systèmes ordonnançables sur une architecture donnée par rapport aux méthodes existantes.

Alors que le modèle de criticité mixte prétend que les composants critiques et non critiques peuvent partager la même plate-forme de calcul, l'interruption des composants non critiques réduit considérablement leur disponibilité. Ceci est un problème car les composants non critiques doivent offrir une degré minimum de service. C'est pourquoi nous définissons des méthodes pour évaluer la disponibilité de ces composants. A notre connaissance, nos évaluations sont les premières capables de quantifier la disponibilité. Nous proposons également des améliorations qui limitent l'impact des composants critiques sur les composants non critiques. Ces améliorations sont évaluées grâce à des automates probabilistes et démontrent une amélioration considérable de la disponibilité : plus de 2 % dans un contexte où des augmentations de l'ordre de 10^{-9} sont significatives.

Nos contributions ont été intégrées dans un framework open-source. Cet outil fournit également un générateur utilisé pour l'évaluation de nos méthodes d'ordonnancement.

Mots-clés: Système temps réel, criticité mixte, multi-processeurs, flux de données

Abstract

Nowadays, the design of modern Safety-critical systems is pushing towards the integration of multiple system components onto a single shared computation platform. Mixed-Criticality Systems in particular allow critical components with a high degree of confidence (*i.e.* low probability of failure) to share computation resources with less/non-critical components without requiring heavy software isolation mechanisms (as opposed to partitioned systems).

Traditionally, safety-critical systems have been conceived using models of computations like data-flow graphs and real-time scheduling to obtain logical and temporal correctness. Nonetheless, resources given to data-flow representations and real-time scheduling techniques are based on worst-case analysis which often leads to an under-utilization of the computation capacity. The allocated resources are not always completely used. This under-utilization becomes more notorious for multi-core architectures where the difference between best and worst-case performance is more significant.

The mixed-criticality execution model proposes a solution to the abovementioned problem. To efficiently allocate resources while ensuring safe execution of the most critical components, resources are allocated in function of the operational mode the system is in. As long as sufficient processing capabilities are available to respect deadlines, the system remains in a 'low-criticality' operational mode. Nonetheless, if the system demand increases, critical components are prioritized to meet their deadlines, their computation resources are increased and less/non-critical components are potentially penalized. The system is said to transition to a 'high-criticality' mode.

Yet, the incorporation of mixed-criticality aspects into the data-flow model of computation is a very difficult problem as it requires to define new scheduling methods capable of handling precedence constraints and variations in timing budgets.

Although mixed-criticality scheduling has been well studied for single and multi-core platforms, the problem of data-dependencies in multi-core platforms has been rarely considered. Existing methods lead to poor resource usage which contradicts the main purpose of mixed-criticality. For this reason, our first objective focuses on designing new efficient scheduling methods for data-driven mixed-criticality systems. We define a meta-heuristic producing scheduling tables for all operational modes of the system. These tables are proven to be correct, *i.e.* when the system demand increases, critical components will never miss a deadline. Two implementations based on existing preemptive global algo-

rithms were developed to gain in schedulability and resource usage. In some cases these implementations schedule more than 60% of systems compared to existing approaches.

While the mixed-criticality model claims that critical and non-critical components can share the same computation platform, the interruption of non-critical components degrades their availability significantly. This is a problem since non-critical components need to deliver a minimum service guarantee. In fact, recent works in mixed-criticality have recognized this limitation. For this reason, we define methods to evaluate the availability of non-critical components. To our knowledge, our evaluations are the first ones capable of quantifying availability. We also propose enhancements compatible with our scheduling methods, limiting the impact that critical components have on non-critical ones. These enhancements are evaluated thanks to probabilistic automata and have shown a considerable improvement in availability, *e.g.* improvements of over 2% in a context where 10^{-9} increases are significant.

Our contributions have been integrated into an open-source framework. This tool also provides an unbiased generator used to perform evaluations of scheduling methods for data-driven mixed-criticality systems.

Keywords: Safety-critical systems, mixed-criticality, multi-processors, directed acyclic graphs, data-driven

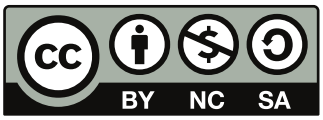


Table of Contents

List of figures

1 Introduction

TABLE OF CONTENTS

1.1 General context and motivation overview

Safety-critical software applications [1] are deployed in systems such as airborne, railroad, automotive and medical equipments. These systems must react to their environment and adapt their behavior according to conditions presented by the latter. Therefore, these systems have stringent time requirements: the response time of the system must respect time intervals imposed by its physical environment. For this reason, most safety-critical systems are also considered as *real-time systems*. The correctness of a real-time system depends not only on the logical correctness of its computations but also on the temporal correctness, *i.e.* the computation must complete within its pre-specified timing constraint (referred to as the *deadline*). In this dissertation, we focus on modern complex real-time systems which are often composed of software applications with different degrees of criticality. A deadline miss on a high critical component can have severe consequences. That is the main reason why these systems are categorized as safety-critical: a failure or a malfunction could cause catastrophic consequences to human lives or cause environmental harm. On the other hand, a deadline miss on a low critical/non-critical component, while it should occur on rare occasions, does not have catastrophic consequences on its environment, it reduces the quality of service of the system significantly.

Ensuring logical and time correctness is a challenge for system designers of safety-critical systems. The real-time scheduling theory has defined workload models, scheduling policies and schedulability tests, to deem if a system is *schedulable* (*i.e.* that respects

deadlines) or not. At the same time, programs often communicate, and share physical and logical resources: the interaction between these software components needs to be taken into account to guarantee timeliness in a safety-critical system.

Besides real-time schedulability, *logical correctness* is also necessary in safety-critical systems: the absence of deadlocks, priority inversions, buffer overflows, among others non-desired behaviors, is often verified at design-time. For these reasons system designers have opted to use models of computation like data-flow graphs or time triggered execution.

The data-flow model of computation [2; 3] has been widely used in the safety-critical domain to model and deploy various applications. This model defines *actors* that communicate with each other in order to make the system run: the system is said to be *data-driven*. The actors defined by this model can be tasks, jobs or pieces of code. An actor can only execute if all its predecessors have produced the required amount of data. When this requirement is met, the actor also produces a given amount of data that can be consumed by its successors. Therefore, actors have *data-dependencies* in their execution. Theory behind this model and its semantics provide interesting results in terms of logical correctness: deterministic execution, starvation freedom, bounded latency, are some examples of properties that can be formally proven thanks to data-flow graphs. Building upon these theoretical results, industrial tools like compilers, have been developed for safety-critical systems. Besides verifying logical correctness, these tools perform the *deployment* of the data-flow graph into the targeted architecture, *i.e.* how actors are scheduled and where they are placed in the executing platform.

Current trends in safety-critical systems: In the last decade, safety-critical systems have been facing issues related to stringent non-functional requirements like cost, size, weight, heat and power consumption. This has led to the inclusion of *multi-core architectures* and the *mixed-criticality scheduling model* [4] in the design of such systems.

The adoption of multi-core architectures in the real-time scheduling theory led to the adaptation and development of new scheduling policies [5]. Processing capabilities offered by multi-core architectures are quite appealing for safety-critical systems since there are important constraints in terms of power consumption and weight. Nonetheless, this type of architecture was designed to optimize the average performance and not the worst-case. Due to shared hardware resources, this architecture is hardly predictable. The difference between the best and worst-case becomes more significant. Since in hard real-time systems the Worst-Case Execution Time (WCET) is used to determine if a system is schedulable ensuring time correctness becomes harder when multi-core architectures are considered.

Safety standards are used in the safety-critical to certify systems requiring a certain degree of confidence. Criticality or assurance levels define the degree of confidence that is given to a software component. The higher the criticality level, the more conservative the verification process and hence the greater the WCET of tasks will be. For example the avionic DO-178B standard defines five criticality levels (also called assurance levels). Traditional safety-critical systems tend to isolate physically and logically programs that have different levels of *criticality*, by using different processors or by using software partitions. Conversely, Mixed-Criticality advocates for having components with different criticality levels on the same execution platform. Since WCETs are often overestimated for the most critical components, computation resources are often wasted. Software components with lower criticalities could benefit from those processing resources. To guarantee that the most critical components will always deliver their functionalities, in Mixed-Criticality the less critical components are penalized to increase the processing resources given to the most critical components. The fact that the execution platform can be shared between software components with different criticalities improves resource usage considerably, more so when multi-core architectures are considered (the WCET tends to be more overestimated for this type of architecture).

Multi-core mixed-criticality systems are a promising evolution of safety-critical systems. However, there are unsolved problems that need to be leveraged for the design of such systems, in particular when data-driven applications are considered. This thesis is an effort towards designing techniques for data-driven mixed-criticality multi-core systems that yield efficient resource usage, guarantee timing constraint and deliver a good quality of service.

The work presented in this dissertation was funded by the research chair in Complex Systems (DGA, Thales, DCNS, Dassault Aviation, Télécom ParisTech, ENSTA and École Polytechnique). This chair aims at developing research around the engineering of complex system such as safety-critical systems.

1.2 Contributions

Regarding the deployment of data-driven applications for mixed-criticality multi-core systems, our contributions are centered around two main axes. **(i) We begin by defining efficient methods to schedule data-dependent tasks in mixed-criticality multi-core systems**. At the same time, dependability is essential for safety-critical systems and while mixed-criticality has the advantage of improving resource usage, it compromises the *availability* (an attribute of dependability) of the less critical components. **(ii)** For this reason,

we define methods to evaluate and improve the availability of mixed-criticality systems.

The schedulability problem of real-time tasks in multi-core architectures is known to be NP-hard [6; 7]. When considering mixed-criticality multi-core systems, the problem holds its complexity [8]. Thus, in our contributions we have designed a meta-heuristic capable of computing scheduling tables that are deemed correct in the mixed-criticality context. This meta-heuristic called MH-MCDAG, led to a global and generic implementation presented in this dissertation. Existing scheduling approaches can be easily adapted with this generic implementation. Because we based our implementation in global approaches there is an improvement in terms of resource usage compared to approaches of the state-of-the-art. Since most industrial standards define more than two levels of criticality, the generalization of the scheduling meta-heuristic to handle more than two criticality modes is also a contribution presented in our works. This extension is a recursive approach of MH-MCDAG to which additional constraints are added in order to respect the schedulability of tasks executed in more than two criticality modes.

The second axis of contributions presented in this dissertation is related to the quality of service that needs to be delivered by safety-critical systems. In fact, all mixed-criticality models penalize the execution of tasks that are not considered with the highest criticality level, that way tasks that have a higher criticality can extend their timing budget in order to complete their execution. By guaranteeing that the highest criticality tasks will always have enough processing time to complete their execution, mixed-criticality ensures the time correctness of the system even under the most pessimistic conditions. While some tasks are not considered as high-criticality, they are still important for the system: their services are required to deliver a good quality of service. For this reason, we propose methods to analyze the availability of tasks executing in a mixed-criticality system. We have defined methods in order to estimate how often tasks are correctly executed (time-wise). Additionally, we propose various enhancements to multi-core mixed-criticality systems that considerably improve the availability of low criticality services. The inclusion of these enhancements has led to the development of translation rules to probabilistic automaton, that way system simulations can be performed and an availability rate can be estimated thanks to appropriate tools.

The final contribution we present in this dissertation is the open source framework we developed during our research works. This framework gathers the scheduling techniques we have developed in addition to the transformation rules that are used to estimate availability rates. Another key aspect of the framework was the development of an unbiased generator of data-driven mixed-criticality systems. In fact since our works are adjacent to

various research domains (real-time scheduling and operational research), we had to incorporate different methods to assess statistically our scheduling techniques. This framework allowed us to perform experimental evaluations. We statistically compared our scheduling techniques to the state-of-the-art in terms of acceptance rate and number of preemptions. We also present experimental results for generalized systems with more than two criticality levels. To our knowledge these experimentations are the first ones to consider generalized data-driven mixed-criticality systems.

1.3 Thesis outline

The organization and contents of the chapters of this thesis are summarized below.

- Chapter 2 presents industrial trends, background notions and related works. We start by describing the current industrial trends that led to the consideration of multi-core and mixed-criticality. Then, we briefly describe how time correctness is obtained in safety-critical systems. The third part of the chapter presents the data-flow model of computation that has been widely used in real-time applications to demonstrate logical correctness. Finally, a discussion about the importance of dependability aspects is developed. In particular we look into the influence that data-driven and mixed-criticality applications have in the dependability of the system.

- Chapter 3 defines problems and sub-problems that we have identified and addressed in this dissertation. The scheduling of mixed-criticality systems composed of data-dependent task has been rarely addressed in the literature and most approaches present limitations. At the same time, the most popular mixed-criticality execution model of the literature only copes with the schedulability of the system, whereas other aspects related to dependability, *e.g.* availability, are also important in the safety-critical domain.

- Chapter 4 introduces the task model model we consider, in addition to an overview of our contributions presented in this thesis. The MC-DAG model we define gathers all the relevant aspects related to our research works: data-dependencies with mixed-criticality real-time tasks. We briefly present how the contributions presented throughout this dissertation tackle the problems and sub-problems defined in Chapter 3.

- Chapter 5 presents our findings related to the scheduling of MC-DAGs in multi-core architectures. We begin by introducing the notion of MC-correctness for the

scheduling of MC-DAGs. Then, we present a necessary condition ensuring MC-correctness. Building upon this condition, we designed a MC-correct meta-heuristic: MH-McDAG. A generic and global implementation of MH-McDAG is also presented in this chapter: this implementation aims at solving the limits of existing approaches that have also tackled the problem of MC-DAG scheduling. The final part of this chapter presents a generalization of the necessary condition and the meta-heuristic to handle an arbitrary number of criticality levels.

- Chapter 6 studies the availability analysis problem of MC system executing MC-DAGs. We first introduce the necessary information required in order to perform availability analyses for MC systems. The second part of the chapter proposes to enhance the availability for MC systems in order to deliver an improved quality of service for tasks that are not consider has highly critical. Translation rules in order to obtain probabilistic automata have been defined. Probabilistic automata allow us to perform system simulations when the execution model of the system becomes complex due to the availability enhancements we want to deploy.

- Chapter 7 describes the open-source framework tool we have developed during this thesis. Existing contributions have mostly proposed theoretical results, therefore an objective during this thesis was to develop a tool allowing us to compare our contributions to existing scheduling techniques. To achieve this objective we developed an unbiased generator for the task model we have defined.

- Chapter 8 presents the validation of our contributions thanks to experimental results. First, an evaluation of the scheduling approaches presented in Chapter 5 is performed. We statistically assess the performances of our scheduling strategies compared to the existing approaches of the literature. Two important metrics for real-time schedulers are considered: the *acceptance rate* (*i.e.* the number of systems that are schedulable) and the *number of preemptions* entailed by the algorithms. Second, we study the impact of having more than two criticality levels: since few works have tackled the mixed-criticality scheduling problem with more than two levels of criticality, our results are the first ones to present experimental results considering data-driven applications in mixed-criticality multi-core systems. Many configurations for the generated systems were tested to carry out rigorous benchmarking.

- Chapter 9 is the conclusion of this dissertation. It summarizes our contributions and discusses future research perspectives.

2 Industrial needs and related works

Table of Contents

In this chapter we present the engineering and technological trends related to our research works. These trends are pushing towards the reduction of non-functional properties (*e.g.* cost, power, heat, *etc*) while still delivering more functionalities. This explains the motivation behind the *adoption of multi-core architectures* and *mixed-criticality systems*. We briefly recall principles of real-time scheduling, used to obtain temporal correctness. Then, we discuss related contributions dealing with mixed-criticality. To obtain logical correctness in safety-critical systems, we discuss works related to the design of data-flow/data-driven applications. For the final part of this chapter, we discuss the importance of dependability on safety-critical systems and demonstrate how mixed-criticality influences availability (a criteria of dependability). Our research works are related to all these topics and we have identified objectives that need to be fulfilled to deploy data-driven applications into mixed-criticality multi-core systems.

2.1 Industrial context and motivation

Safety-critical systems are nowadays confronted to new industrial trends: **(i)** embedded architectures used in the safety-critical domain, are putting efforts towards reducing size,

weight and power of computing elements; but at the same time **(ii)** more and more services are expected to be delivered by safety-critical systems. These two trends have led to innovation in terms of hardware (*e.g.* introduction of multi-core processors in embedded systems) and theory (*e.g.* mixed-criticality scheduling).

2.1.1 The increasing complexity of safety-critical systems

Nowadays, safety-critical systems are composed of many software and hardware components necessary to the *correct* execution of these systems into their physical environments. If we consider an airborne system for example, its services can be decomposed into the following categories: cockpit, flight control, cabin, fuel and propellers. Each one of this categories is decomposed into various software functionalities as well: camera systems, audio control, displays, monitoring, data recording, are some examples of functionalities included in the cockpit. The *design of safety-critical system is therefore very complex*. Safety standards have defined *criticality or assurance levels* for software and hardware components that are deployed into safety-critical systems. The consequence of missing a deadline vary based on the software component criticality level. Standards such as DO-254, DO-178B and DO-178C (used for airborne systems), define five assurance levels: each one of these levels is characterized by a failure rate (level A 10^{-9} $errors/h$, level B 10^{-7}, and so on). A deadline miss on a level A task might lead to catastrophic consequences, while a deadline miss on a level E task has no impact on the system's safety. Traditionally, services of different criticalities have been separated at a hardware level: in federated avionics for example, Line Replacement Units (LRUs) are used within the airborne system to deliver one specific functionality. By having such a space isolation, failure propagation is avoided. Modularity is also a key property achieved thanks to LRUs: a failing LRU can be replaced when needed.

Nevertheless, during the past few years, the safety-critical industry is putting efforts towards reducing size, weight and power consumption. This trend essentially advocates for the *integration of multiple functionalities on a common computing platform*. Some examples of this trend are the Integrated Modular Avionics (IMA) [9; 10] and the AUTomotive Open System ARchitecture (AUTOSAR) [11] initiatives that aim at the integration of functionalities onto common hardware, while maintaining compliance with safety standards. The main focus of these initiatives is to maintain benefits of the isolation offered by separated processing units but at a software level. In other words, a software component like a hypervisor [12] or a real-time operating system [13], will be responsible for executing and isolating the different functionalities required by the safety-critical system. This

integration has gained additional interest due to the constant innovation in integrated chips as well, in particular thanks to multi-core processors.

2.1.2 Adoption of multi-core architectures

Multi-core architectures were proposed to solve problems related to power consumption, heat dissipation, design costs and hardware bugs that mono-core processors were facing due to the constant miniaturization of transistors. *Computational power is improved by exploiting parallelism instead of increasing clock frequency of processors*. This prevents power consumption to grow and limits heat dissipation. Architectural design errors are also limited since in most multi-core architectures, all cores are identical.

Since manufacturers have decided to adopt multi-core architectures as the dominant architecture platform for embedded devices (for cost reasons, mainly due to scale factors in mobile phone industry), the *adoption of multi-core architectures is a necessity for the safety-critical domain*. Nevertheless, while multi-core architectures improve average performances, shared hardware resources like caches and memory buses *makes these architectures hardly predictable*. To ensure temporal correctness, this limitation has been solved by overestimating the worst behavior of applications deployed in the multi-core system. Real-time systems allocate processing resources to guarantee the execution of tasks even in their worst case.

In conclusion, to cope with current industrial needs, safety-critical systems need to: **(i)** execute *efficiently* in multi-core architectures. Efficiency in this case is related to number of cores necessary to schedule a system, most embedded architectures are limited by the number of processors that can be integrated. **(ii)** Make good use of processing resources by delivering as many functionalities as possible, even if these functionalities different criticalities. These two requirements need to respect temporal and logical correctness which are necessary in safety-critical systems. To do so, real-time scheduling and models of computation that assist system designers need to be adapted to these current trends.

In the next section we recall principles of real-time scheduling and present contributions that allowed systems designers to deploy real-time systems into mono and multi-core architectures.

2.2 Temporal correctness for safety-critical systems

Like we mentioned in Chapter 1, the correctness of a safety-critical system not only depends on the logical correctness of its outputs but also on timeliness. To satisfy deadlines imposed by their environments, safety-critical systems use models and theoretical results from real-time scheduling analyses and techniques. Nonetheless, resource allocation used by real-time schedulers is based on worst-case analysis which is very difficult to estimate, more so when multi-core architectures are considered.

2.2.1 Real-time scheduling

The workload models used in the real-time scheduling theory define timing constraints such as deadlines and resource requirements of the system. The workload model [14] we are interested in, also called task model, states that a program/function/piece of code becomes available at some instant, consumes a certain time duration for its execution and is required to complete its execution within a deadline.

The *periodic task model* [14] is the most widespread model in the conception of safety-critical systems. Each task τ_i, is instantiated several times during the life-time of the safety-critical system. These instances are called *jobs*.

Definition 1. ***Periodic task*** *A periodic task is defined through the following parameters:*

- ***Period*** T_i*: the delay between two consecutive job activations/releases of* τ_i*.*
- ***Execution time*** C_i*: the required time for a processor to execute an instance of the task* τ_i*. In general, this parameter is given by the Worst-Case Execution Time (WCET) of the task. The WCET represents an upper-bound of the task execution time.*
- ***Deadline*** D_i*: the relative date at which the task must complete its execution.*

Fig. 2.1 illustrates a Gantt diagram of a real-time task execution and its parameters. The actual execution of the task is represented with the blue box: two jobs are illustrated in the figure. These two jobs are released at the same the period of the task arrives and have to complete before their respective deadlines. The second job of the task is preempted at some point of its execution. Only one activation occurs during the T_i period.

The utilization factor U_i of a task τ_i is derived from these parameters:

$$U_i = \frac{C_i}{T_i}.$$

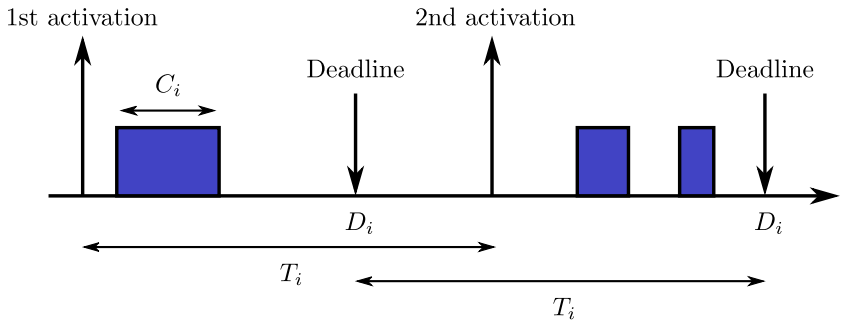


Figure 2.1: Real-time task characteristics

A real-time system is said to be composed of a *task set* τ, and its utilization can also be easily derived:

$$U = \sum_{\tau_i \in \tau} U_i.$$

Definition 2. ***Schedulability*** *A task set τ is said to be schedulable under a scheduling algorithm $\mathcal{A}$, if for all possible releases of a task $\tau_i \in \tau$, τ_i can execute for C_i time units and complete its execution within its deadline D_i.*

Therefore, the goal of real-time scheduling is to *allocate* tasks (*i.e.* give processing capabilities to a task for a specific amount of time) so as to not provoke deadlines misses. The real-time scheduling can be performed *off-line* (*i.e.* static scheduling tables are produced at design-time and used at runtime) or *on-line* (*i.e.* processing capabilities are allocated to tasks according to a scheduling policy during the execution of the system).

2.2.2 Off-line vs. on-line scheduling for real-time systems

To compute **off-line schedulers** for real-time systems, release times and deadlines of all tasks that constitute the system must be known a priori. A scheduling table needs to be computed respecting these constraints. Integer linear or constraint programming [15] are some of the techniques used to compute the scheduling tables. Once the table is obtained, the operating system uses it to allocate tasks into the computation platform. Some examples of approaches that compute static scheduling tables are Time-Triggered (TT) systems [16]. An advantage of TT scheduling is its complete determinism, which makes this approach easier to verify and certify: in the safety-critical domain systems need to be certified before they are deployed into their physical environment. Nonetheless, an important shortcoming of this approach is its flexibility: if new tasks need to be incorporated, a new table needs to be computed. The processing capabilities left cannot be easily used for extra services.

On-line schedulers generate the scheduling during runtime and are capable of incorporating new tasks on-the-fly. This type of schedulers can be classified into *fixed-priority* (FP) and *dynamic-priority* (DP). Some examples of well-known FP algorithms are Rate Monotonic Scheduling (RMS) and Deadline Monotonic Scheduling (DMS) [14]. These scheduling policies have defined sufficient conditions to determine if a task is schedulable a priori of runtime. If the utilization rate of the system is equal or less than $n(2^{1/n}-1)$, where n is the number of implicit-deadline tasks, the task set will be schedulable with RMS on mono-core architectures. Earliest Deadline First (EDF) [14] and Least-Laxity First (LLF) [17] on the other hand are DP schedulers. It is known that EDF is an optimal scheduling algorithm for single-core architectures with implicit deadline tasks [14], *i.e.* the utilization of the system needs to be inferior or equal to one ($U \leq 1$) to correctly schedule the system. LLF is also an optimal algorithm for mono-core processors [17] but it entails more preemptions than EDF.

Definition 3. ***Utilization bound [18]*** *A scheduling algorithm $\mathcal{A}$ can correctly schedule any set of periodic tasks if the total utilization of the tasks is equal or less than the utilization bound of the algorithm.*

In the case of constrained deadlines (*i.e.* the deadline is inferior to the period) the *demand bound function* can be used in order to test for schedulability and EDF is still optimal [19].

Different scheduling approaches to schedule a real-time task sets on a mono-core architecture have been proposed by the literature and optimality has proven to be obtainable. However, like we mentioned at the beginning of this chapter, current industrial needs are pushing towards the adoption of multi-core architectures.

2.2.3 Real-time scheduling on multi-core architectures

To support multi-core architectures, existing real-time scheduling algorithms developed for mono-core processors were adapted. This adaptation followed two main principles: the *partitioned* or the *global* approach. A survey presenting different methods to schedule real-time tasks on multi-core processors is presented in [7].

The **partitioned approach** consists in dividing the task set of the system into various partitions, and schedule them into a single core by applying a mono-core scheduling algorithm in this partition. Therefore, existing scheduling algorithms for mono-core processors can be used as-is in the partition created. How partitions are formed and distributed among cores is the main problem this approach needs to solve. It is widely known that such partitioning is equivalent to the bin-packing problem, and is therefore highly intractable:

NP-hard in the strong sense [20]. Optimal implementations are impossible to be designed. *Approximation algorithms* are therefore used to perform such partitioning. For example, Partitioned-EDF (P-EDF) using First-fit Decreasing, Best-fit Decreasing and Worst-fit Decreasing heuristics have shown to have a speedup factor no larger than 4/3 [20].

Definition 4. *(**Clairvoyant optimal algorithm [21]**) 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 algorithm cannot be implemented. Nevertheless, the speedup factor quantifies the distance from optimality of the algorithm's resource-usage efficiency.

Definition 5. *(**Speedup factor**) The speedup factor $\phi \geq 1$ for a scheduling algorithm $\mathcal{A}$ 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 $\mathcal{A}$.*

To avoid partitioning and its underlying bin-packing problem, **global approaches** have also been developed by the real-time community. Other advantages over partitioned approaches are the following: (i) spare capacity created when tasks execute for less than their WCET can be utilized by all other tasks, (ii) there is no need to run load balancing/task allocation algorithms when the task set changes. Some examples of global scheduling policies are Global-EDF (G-EDF), which has a speedup factor of $(2-1/m)$ [22] (where m is the number of processors). An adaptation to use static priorities under the global approach was proposed by Andersson *et al* [23] which has a utilization bound of $m^2/(3m-2)$. The Proportionate Fair (Pfair) [24] is based on the idea of fluid scheduling, where each task makes proportionate progress to its utilization. Pfair has been shown to be optimal for period tasksets with implicit deadlines and has a utilization bound of m [24]. Further improvements to the fluid model have been designed by the community to gain in efficiency [25] since a limitation of the fluid is the number of preemptions generated by this type of algorithm.

Efficient scheduling for multi-core real-time systems have been developed by the community either by adapting existing scheduler (*e.g.* P-EDF, G-EDF) or by proposing completely new models (*e.g.* Pfair). Nevertheless, real-time systems are dimensioned in function of the WCET of tasks. In fact, estimating precise and tight WCET for a task is very difficult and for safety reason it tends to be overestimated. This overestimation leads to poor resource usage since real-time systems are dimensioned taking into account these pessimistic WCETs. This overestimation is more remarkable for multi-core architectures.

2.2.4 The Worst-Case Execution Time estimation

To ensure temporal correctness in the safety-critical domain, systems are dimensioned in terms of tasks' WCET. That way, in the worst-case scenario a task will have enough timing budget to complete its execution. However, estimating the WCET of a task is a difficult problem [26]. The WCET estimation has to take into account many factors: the target architecture (*e.g.*, optimizations at the hardware level to improve performance), the complexity of software (*e.g.* loops and if-else branches can significantly change the execution time of a program), the shared resources (*e.g.* data caches shared among multi-core processors, software-level resources like semaphores and mutex), and so on. The estimation of WCET can be classified into three main categories: *static analysis*, *measurement-based* and *probabilistic* approaches.

Static analysis is a generic method to determine properties of the dynamic behavior of a task without actually executing it. Research around static analysis methods has shown to be adaptable to different types of hardware architectures, ranging from simple processors to complex out-of-order architectures. For example, WCET analysis taking into account pipeline behavior was studied in [27]. Due to the large processor-memory gap, all modern processors have opted to employ a memory hierarchy including caches. Predicting caches behavior has been a major research perspective for WCET analysis [28] since execution time of tasks is highly impacted by hits or misses on caches. Recent works are pushing towards the modularity of the WCET analysis. For instance, integer linear programming to analyze abstract pipeline graphs were studied in [29]. Another modular and reconfigurable method to perform the WCET analysis on different types of architectures was developed in [30]. The limit with static analysis comes from the model describing the software and the architecture that needs to match the reality. Obtaining a realistic model can be very difficult since it requires an extensive knowledge of the system components. Performing static analysis on hardware instructions or large programs becomes complex very easily as well, so this approach is not appropriate during early stages of the development phase.

Measurement-based approaches is an alternate solution to static analysis that does not require an extensive knowledge of the software and hardware architecture. The principle of measured-base analysis is to execute a given task on a given hardware or simulator to estimate bounds or distributions for the execution times of the task. Because the subset of measurements is not guaranteed to contain the worst-case, this approach often requires to perform many tests. Test have to cover all potential execution scenarios which can also be difficult since putting the hardware or software into specific states for testing might not be

straightforward. Hybrid approaches try to overcome these limitations [31] by combining static methods and completing them with measurement-based techniques.

Probabilistic timing analysis based on extreme value theory [32] aims at providing sound estimations in an efficient manner (*i.e.* with a low number of measurement runs). A probabilistic distribution which bounds the WCET is obtainable thanks to this method. This probabilistic WCET (pWCET) is derived for single and multi-path programs thanks to this method. The applicability of probabilistic timing analysis for IMA-based applications was demonstrated in [33]. The approach showed that tight pWCET estimates were obtainable and that the approach was scalable for large functions. Some of the main challenges that probabilistic timing analysis aims to solve nowadays are presented in [34]. Determining if input samples to derive pWCETs are representative, reliable parameters for extreme value models and the interpretation on the consequences of the obtained results are some of the main open problems for probabilistic analysis nowadays.

As we have demonstrated, the worst-case execution time is a difficult problem which *becomes even more difficult when multi-core architectures are considered.* Multi-cores are harder to analyze due to inter-thread interferences when accessing shared resources (*e.g.* shared bus or caches). The average performance is improved but the difference between best and worst-case execution becomes significant. To be compliant with safety standards, system designers are enforced to give a large WCET for tasks that have a high assurance level. Yet, most of the time, tasks will not use all their timing budgets, leading to poor resource usage.

2.3 The Mixed-Criticality scheduling model

The design of *Mixed-Criticality* (MC) systems is a consequence of the overestimation of WCET enforced by certification authorities and the need to integrate multiple functionalities of different criticalities on a common computing platform.

The seminal paper of Mixed-Criticality was presented by Vestal in [4]. This paper presents the following observation: the higher the criticality level, the more conservative the verification process and hence the greater the WCET of tasks will be. This is problematic since enforcing a "pessimistic" WCET leads to poor resource usage, nonetheless for safety reasons the WCET is used as the C_i timing budget in the periodic real-time task model. Due to the fact that systems are dimensioned considering this pessimistic WCET, tasks are often going to finish their execution before their timing budget is consumed. MC systems solve this issue since they are *capable of considering various WCETs for tasks in function of the criticality mode the system is in*. This behavior allows system designers to

incorporate more tasks in their systems and improve resource usage. The safety-critical system is enriched with the following parameters [35]:

- **Set of criticality levels**: a MC system is now composed of a *finite set of criticality levels*. The most common MC model defines two levels of criticality: HI and LO-criticality. The system is said to be executing in a given criticality level/mode. It is assumed the system starts its execution in the lower criticality level.

- **Criticality level of a task**: a task is said to belong to one of the criticality levels of the system. Depending on the criticality level the tasks belongs to, different timing budgets will be allocated to it. For example HI-criticality tasks in a dual-criticality system are executed in the LO and HI-criticality modes of the system, whereas LO-criticality tasks are often only executed in the LO-criticality mode [36; 8; 37] or have reduced service guarantees in the HI-criticality mode [38].

- **Set of timing budgets, periods and deadlines**: since the exact WCET is very difficult to estimate [26], MC systems define a vector of timing budgets for tasks that execute in more than one criticality level. $C_i(\chi_j)$ is the timing budget given to task τ_i in criticality mode χ_j. Periods can also change in function of the criticality mode. In general, the following constraints need to be true for any task τ_i [35]:

$$\chi_1 \succ \chi_2 \implies C_i(\chi_1) \geq C_i(\chi_2)$$
$$\chi_1 \succ \chi_2 \implies T_i(\chi_1) \leq T_i(\chi_2)$$

  for any two criticality levels χ_1 and χ_2. The completion of the model to make the deadline criticality-dependent has not been addressed in detail but greater or lower deadlines could be considered.

  If any job attempts to execute for a longer time than is acceptable in a given mode then a criticality mode change occurs. The majority of papers restrict the model to increase the criticality mode.

  Definition 6. ***Timing Failure Event*** *The time at which a job consumes its task LO time budget without completing its execution is called a Timing Failure Event (TFE).*

Definition 7. ***MC task set*** *A MC periodic task set τ, is defined by the tuple (χ_i, T_i, C_i, D_i).*

- *χ_i the criticality level of the task.*

- *T_i the set of period values corresponding to the criticality levels.*

- C_i *the set of execution time budgets of the task for the criticality levels of the system.*
- D_i *the set of deadlines corresponding to the criticality levels.*

Definition 8. ***MC-schedulable in dual-criticality systems*** *A MC task set* τ *is said to be MC-schedulable by a scheduling algorithm* $\mathcal{A}$ *if,*

- *LO-criticality guarantee: Each task in* τ *is able to complete its execution up to its* $C_i(LO)$ *within its deadline in LO-criticality mode* ($D_i(LO)$)*, and*
- *HI-criticality guarantee: Each task with a HI-criticality level is able to complete its execution up to its* $C_i(HI)$ *within its deadline in HI-criticality mode* ($D_i(HI)$)*.*

The scheduling of MC tasks is computationally intractable [8; 39] for mono and multi-core processors. **The problem is NP-hard in the strong sense.** In fact, the schedulability of the system has to be guaranteed in all the operational modes *but also when the system needs to perform a mode transition to the higher-criticality mode*. The variations in execution time can provoke a deadline miss if the scheduling is not performed correctly. Efficient (with polynomial or pseudo-polynomial complexity) scheduling algorithms have been designed by the community. Like when multi-core architectures where first introduced, existing real-time scheduling policies were adapted to support the MC model. Completely new approaches have also been developed in the literature of MC scheduling. Yet, most contributions have simplified the execution model by not considering communication between tasks, concurrent resource sharing or more than two criticality levels. A review of contributions related to Mixed-Criticality is maintained by Burns and Davis [35].

2.3.1 Mixed-Criticality mono-core scheduling

Similarly to the real-time scheduling policies, MC algorithms on mono-core processors can be performed with FP or DP.

In **FP scheduling**, Response Time Analysis (RTA) is used to determine if the system is schedulable. RTA determines the worst-case response time of a task. In the seminal work of MC, Vestal [4] demonstrated that neither rate monotonic nor deadline monotonic priority assignments were optimal for MC systems: Audsley's priority assignment algorithm [40] was found to be applicable for this model when mono-core architectures are considered. The FP approach gave birth to the Adaptive Mixed-Criticality (AMC) [37] which was extended many times to limit the number of preemptions [41] or gain in overall *quality* (*i.e.* improve the acceptance rate, decrease the speedup factor, among other qualitative metrics). Santy *et al*. [42] propose to delay as much as possible the mode transition

to allow the completion of LO-criticality tasks under a FP scheduling. They have also proposed to switch back to a lower criticality mode when an idle time (*i.e.* no tasks being scheduled on the system) occurs during the execution of the system.

Regarding **DP scheduling**, most MC scheduling approaches for mono-core architectures have been based on EDF. Baruah *et al.* [43] proposed a virtual deadline based EDF algorithm called EDF-VD. Virtual deadlines for HI-criticality tasks are computed so that their execution in the LO-criticality mode is performed sufficiently soon to be schedulable during the mode transition to the HI-criticality mode. It was demonstrated that EDF-VD has a speedup factor of 4/3, the optimal speedup factor for any MC scheduling algorithm [44]. Ekberg and Yi [45] introduced a schedulability test for constrained-deadline dual-criticality systems using a demand bound function. A deadline tightening strategy for HI-criticality tasks was also introduced, showing that their algorithm performs better than EDF-VD. Further improvements on the demand bound function test and in the deadline tightening strategy were proposed by Easwaran in [46].

While scheduling approaches have been designed for mono-core processors and have proven to be efficient (*i.e.* EDF-VD has the optimal speedup factor for mono-core MC systems), MC scheduling also had to be adapted for multi-core processors. To adopt MC scheduling in modern safety-critical system, multi-core execution needs to be supported.

2.3.2 Mixed-Criticality multi-core scheduling

To adopt MC scheduling in current safety-critical design, scheduling strategies need to be capable of allocating tasks into multi-core architectures. Most of the existing multi-core scheduling algorithms have been designed based on *partitioned* or *global* scheduling.

Similarly to the partitioned strategies that were presented before in this chapter, **partitioned MC scheduling** needs to statically assign tasks to cores with a *partitioning strategy*. Once the partitioning is performed, a single-core scheduling strategy is applied on the partition. We say that a partitioning strategy is *criticality-aware* when higher criticality tasks are assigned to cores before lower criticality tasks. An appealing feature of the partitioned approaches is that the mode transition to a higher criticality mode can be contained within the core that had the fault, limiting the fault propagation. The partitioning strategy is *criticality-unaware* tasks' criticalities do not have an influence on the partitioning policy. Baruah *et al.* [47] combined the EDF-VD schedulability test with a first-fit partitioning strategy to derive a scheduling algorithm with a speedup factor of 8/3. In [48], criticality-aware partitioning was shown to perform better than criticality-unaware partitioning for dual-criticality systems. Consequently, improvements to the criticality-aware

partitioning were designed by Gu *et al.* [49] and Han *et al.* [50]. Nonetheless, Ramanathan and Easwaran [51] recently developed a utilization-based partitioning strategy that gives better schedulability rates for criticality-aware and unaware partitioning. Gratia *et al.* [52] extended the single-core server based scheduling algorithm RUN, developed for classical real-time systems, to implicit-deadline dual-criticality periodic MC systems. Their experimental results demonstrated that GMC-RUN generated less preemptions than schedulers based on the fluid model [53].

Global MC scheduling allow task to execute on any of the processing cores and migrate between cores during runtime. While global MC scheduling approaches have proven to be less efficient in terms of schedulability compared to partitioned approaches, they are more flexible (*e.g.* no need to calculate new partitions when tasks are added to the system). Pathan [54] proposed a global FP scheduling algorithm based on RTA with a schedulability test based derived from Audsley's approach [40]. A global adaptation of the EDF-VD algorithm was designed by Li *et al.* [55]. Lee *et al.* [56; 53] have adapted the fluid scheduling model [24] to handle MC systems. Their contribution called MC-Fluid assigns two execution rates to each task in function of the criticality mode of the system. An optimal rate assignment algorithm for such systems was also proposed and showed that MC-Fluid has a speedup factor of $(\sqrt{5}+1)/2$. Baruah *et al.* [57] proposed a simplified execution rate assignment called MCF and proved that both MC-Fluid and MCF are speedup optimal with a speedup factor of 4/3.

MC semi-partitioned scheduling is an extension to partitioned scheduling that allows the migration of tasks between cores. The intention behind this approach is to improve the schedulability performance while maintaining the advantages of partitioning. Burns and Baruah [58] proposed a semi-partitioned scheduling of cyclic executives for dual-criticality MC systems. Awan *et al.* [59] extended semi-partitioned algorithms to constrained-deadline dual-criticality MC systems. LO-criticality tasks are allowed to migrate between cores for better utilization.

In conclusion, there are many scheduling strategies for MC systems on multi-cores [35]. Yet, most contributions employ a very pessimistic approach when a mode transition occurs: LO-criticality tasks are completely discarded/ignored. This is not suitable for many practical systems that require minimum service guarantees for LO-criticality tasks.

2.3.3 Degradation model

The most widespread model of MC is the discard model [36; 8; 37] where LO-criticality are completely discarded/ignored once the system makes a transition to the HI-criticality

mode. This approach has shown to give the best schedulability results compared to other existing degradation models but degrades the quality of service of LO-criticality tasks significantly. To overcome this problem, several techniques have been proposed in the past few years for mono and multi-core processors.

Selective Degradation is an improvement for MC systems first proposed by Ren *et al.* in [60]. Task grouping is performed to limit the mode switching consequences within the group that had the failure. Al-bayati *et al.* [61] propose semi-partitioned scheduling in which LO-criticality tasks are able to migrate once the mode transition to the HI-criticality mode occurred. The mode transition occurs for all cores simultaneously but LO-criticality tasks are able to migrate to a different core so they can complete their execution. Another take on semi-partitioned scheduling algorithms was proposed by Ramanathan and Easwaran [62], the utilization bound of their algorithm was derived and experimental results demonstrated that their method outperforms other semi-partitioned approaches [63].

New conditions to restart the LO-criticality mode execution have been proposed by Santy *et al.* [42] and Bate *et al.* [64; 65]. These contributions can be categorized as *Timely recovery* methods. By minimizing the duration in which the system is in the HI-criticality, the limited-service duration of LO-criticality tasks is reduced as much as possible so their availability increases.

Su *et al.* [38; 66] have extended the MC scheduling model with an *Elastic task model*. The LO-criticality tasks parameters are changed by increasing their period so that their utilization is diminished. At the same time, LO-criticality jobs are released earlier so the slack time that HI-criticality tasks can have is utilized by the LO-criticality tasks that have a minimum service guarantee. EDF-VD is adapted to support this execution model.

Many scheduling algorithms for MC systems have been proposed by the literature. Mono and multi-core scheduling ensuring time correctness and efficient resource usage can be obtained thanks to this model. There are limitations in terms of LO-criticality tasks execution identified by the community and new improvements are being proposed. Nevertheless, most existing scheduling approaches for MC systems have been developed for independent tasks sets, where no communication takes place. This is an important limitation regarding the adoption of MC scheduling for industrial safety-critical systems. In real applications tasks communicate or share resources. At the same time, industrial tools to design and verify safety-critical applications often describe software components thanks to graphs and data-flows with precedence constraints and data-dependencies.

2.4 Logical correctness for safety-critical systems

To assist in the design, implementation, deployment and verification of safety-critical system, Models of Computation (MoC) like data-flow graphs have been used in the safety-critical domain. These MoCs help to guarantee the *logical correctness* of software deployed in safety-critical systems. Tools and programming languages based on these models have also been developed and are nowadays widely used by the industry. In this section we present main contributions related to the design of reactive safety-critical systems using this MoC. We also demonstrate that most data-flow MoCs are not adapted to handle the MC execution model. The few contributions that do support variations in execution times of tasks and mode transitions have poor resource usage which is in contradiction with the main purpose of the MC model.

2.4.1 The data-flow model of computation

The data-flow model of computation began with Kahn Process Networks (KPN) [2]. These networks model a collection of concurrent processes communicating with first-in, first-out (FIFO) channels. Data is represented as tokens that go through the network. A process blocks its execution if at least one of its input channels is empty. Once all inputs are present, the process is able to produce tokens in its output channels at any time. Channels are supposed to be unbounded (*i.e.* have infinite memory). Data Process Networks also known as Data-flow networks (DFN) are related to Kahn's denotational semantics and have been able to help the safety-critical community in different fields. As opposed to KPNs, due to their mathematical foundation, various properties can be checked and demonstrated in DFNs, *e.g.* check if any deadlock can occur or to prove that an invariant is respected, if the number of tokens in the graph is always the same, channels can be bounded, and so on), therefore modeling a program with a Data-flow graph can be interesting. Specific tools (eg. Prelude [67], SynDEx [68], Lustre-v4, Polychrony, ...) are able to generate code from a data-flow specification, making development of applications easier for certification and less error-prone.

The Synchronous Data-flow (SDF) model [3; 69] is widely used in safety-critical systems, in particular for reactive systems and signal processing because it provides a natural representation of data going throw processes. An application modeled with a SDF can be proven to be executable (indefinitely) in a periodic way.

Definition 9. ***Synchronous data-flow*** *A Synchronous data-flow graph is defined by a tuple* $G = (V, E, p, c)$*:*

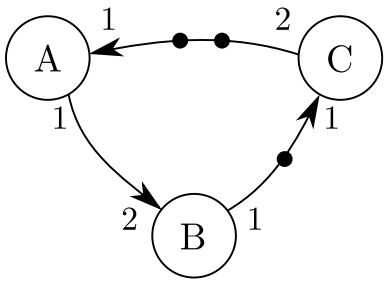


Figure 2.2: Example of a SDF graph

- *V is the vertex set of nodes also called* actors*. Vertices can also have execution times but some models consider instant execution of vertices.*
- $E \subseteq V \times V$ *is the edge set between vertices.*
- $p : E \to \mathbb{N}$ *is a function with* $p(e)$*, which gives the number of produced tokens by actor a for actor* a' *connected by edge e.*
- $c : E \to \mathbb{N}$ *is a function with* $c(e)$*, which gives the number of consumed tokens by actor a in its edge e, connected to actor* a'*.*

An example of SDF is presented in Fig. 2.2. This SDF is composed of three actors and three edges. The production and consumption rates are annotated next to the edges. The black dots in the vertices represent the initial marking of the graph: between actors *A* and *C* there are two tokens available and between actors *B* and *C* there is a single token available.

Mapping a SDF into a multi-core architecture is known to be a NP-complete complete problem and most approaches define heuristics or other approximation algorithms in order to maximize specific criteria, *e.g.* throughput, reliability, efficiency. A general survey of how graph based applications can be scheduled in many and multi-core architectures is presented in [70]. In our context, we focus on approaches that are capable of mapping data-flow with real-time constraints. That way, temporal and logical correctness are ensured.

2.4.2 Data-flow and real-time scheduling

Besides the logical correctness that the data-flow MoC can provide, we are interested in guaranteeing temporal correctness for software components deployed in safety-critical systems. The community around data-flow systems has also recognized the necessity to deliver services in a timely manner which led to the development of real-time scheduling techniques for data-flow MoCs.

Directed Acyclic Graph real-time scheduling

In the seminal work of SDF scheduling [69], Lee and Messerschmitt demonstrate how *the SDF graph can be transformed into an acyclic precedence graph*. This "unfolding" procedure can be done for a given number of periods of the SDF. The construction of a static schedule with acyclic precedence graph, also called Directed Acyclic Graph (DAG), is well known in the literature of operational research [71; 72]. For example techniques that aim at minimizing the *makespan* of the DAG (*i.e.* the timespan between beginning and the end of the execution of the DAG) can be applicable to respect a deadline. This transformation has the advantage of *opening the possibility to apply a large number of contributions that have been proposed for the scheduling and allocation of DAG models into computation platforms*.

Other approaches to schedule DAGs with real-time approaches have also been developed. For instance, sufficient conditions to schedule DAGs with real-time scheduling policies have been found. Fork-join DAG models and G-EDF scheduling was also tackled in [73] by Saifullah *et al.* Nevertheless, their scheduling method is restricted regarding the shape of the DAG that can be scheduled: not all DAGs can be represented with a fork-join model. In [74] Saifullah *et al.* proposed transformation techniques to schedule generalized DAG shapes with G-EDF. This approach is applicable to preemptive and non-preemptive G-EDF scheduling. The speedup factor of this scheduling approach is 4, which is larger in comparison to the scheduling algorithms presented in Section 2.2. *This shows that the scheduling problem with data-dependencies and real-time constraints is more complicated than independent real-time tasks.* Recent works in response time analysis [75] aim at reducing the pessimism on sufficient conditions for G-EDF and G-DM scheduling of DAG: these tests need to take into account the *interruption* that can be caused by vertices. Qamhieh *et al.* [76] have developed a method to deduce local offsets and deadlines for vertices of the DAG and apply G-EDF thanks to these parameters. A schedulability test is derived for their algorithm as well. Some optimizations to reduce the number of cores used to schedule DAGs with real-time constraints were also developed by Qamhieh *et al.* [77; 78]. Their stretching algorithm decomposes the sequential vertices to fill the slack that is left between the completion of the latest vertex of the DAG and the deadline.

Baruah has proposed scheduling techniques for DAGs composed of dual-criticality MC tasks for mono-core [79] and multi-core [80; 81] architectures. These approaches are based on the same principle: a priority ordering is computed for the HI and LO-criticality tasks independently (list scheduling [72] is used to obtain these orderings). To verify MC-schedulability, the HI-criticality mode is scheduled first. If a feasible schedule is ob-

tainable in the HI-criticality mode, then the LO-criticality mode is tested. The particularity of the LO-criticality mode schedule is that HI-criticality tasks will always be scheduled as soon as possible, potentially preempting LO-criticality tasks that were being allocated. The speedup bound of this scheduling approach is equal to $(2-1/m)$, inherited from the performances of list scheduling. Similar works to schedule MC DAGs on multi-core architectures have been proposed in [82; 83] but their model is restricted to the case where all vertices of a DAG have the same criticality level. Ekberg and Yi [84] have proposed a graph-based model called mode-switching digraph real-time (MS-DRT) task model. They have established schedulability analyses for their task model based on EDF under unicore processors.

Other data-flow scheduling algorithms

While the SDF model can be transformed into an acyclic precedence graph, this transformation tends to be costly in terms of memory. For this reason, scheduling approaches for the data-flow MoC have also been performed directly in the SDF graph.

Recent works to extend the SDF model in order to apply real-time scheduling policies have been presented in [85; 86; 87]. Since the classical SDF graph representation does not include any information about deadlines that need to be respected by tasks, a pre-processing phase is applied to the SDF by adding a source and destination node that execute exactly once per iteration. The idea is to specify the real-time latency constraint between the input and output of the SDF. Thanks to this transformation the SDF is transformed to a three-parameter sporadic task set. This task set can then be scheduled by uniprocessor real-time policies.

Other methods to schedule SDF with real-time constraints have used linear programming to find preemptive schedulers for dependent periodic tasks in [88]. Temporal isolation by adjusting offsets and deadlines for tasks are found, that way the dependent real-time task set is transformed into an independent one. Existing FP uniprocessors scheduling techniques are then applied on this newly generated task set. Linear programming was also used to schedule strictly period tasks modeled through SDF graphs without preemptions [89].

Dynamism on communication production and consumption rates

A major limitation of SDF graphs comes from their *static behavior*: communication rates, topology and execution times never change during the execution of the SDF. This is a

limitation for MC system which are based on modes of operations and changes between these modes.

Cyclo-static data-flows (CSDF) [90; 91] have been proved to be schedulable with static schedulers [90] and more recently with real-time policies [92]. CSDF differ from SDFs due to their cyclic changing behavior, *i.e.* rates can vary from one *iteration* of the graph to another. However dynamic changes are known at compile-time, actors can produce/-consume different amount of tokens at each iteration of the graph. In [92], the authors go a step further in their analysis of the CSDF graphs: they apply real-time scheduling policies for the produced task set and consider multi-core architectures as their platform. Their contribution also aims to minimize buffer size between actors, calculate the minimum number of processors and compute earliest starting times (*i.e.* start time for the actor). The CSDF is therefore a generalization of the SDF, however Bamakhrama *et al.* do not support loops in the graph. Complete design of the real-time application is done in [93], the authors transform the program specification until a CSDF is generated. From there, hard real-time scheduling policies are applied in order to schedule the actors of the graph in a multi-core architecture.

Affine Data-flow graphs (ADF) is another form of variation in communication rates. This model was first studied in [94]. An affine function gives a relation between the different clocks of actors in the data-flow graph, the graph becomes ultimately periodic after a certain amount of firings. There is a (n, φ, d) relation between the actors a_1 and a_2, where $n \in \mathbb{N}^*$ is the number of times actor a_1 fires for each $d \in \mathbb{N}^*$ activations of actor a_2 and φ is the instant the affine relation starts.

This MoC inherits interesting functional properties like deadlock freedom, boundedness in communication channels, starvation freedom, among other properties. Bouakaz *et al.* [95] also present a method to schedule ADF using partitioned multiprocessor scheduling. Given different data-flow graphs annotated with production and consumption rates, WCETs and user-provided timing requirements, the proposed method estimates periods, phases and process allocation in order to ensure that overflow/underflow is excluded, timing requirements are met and overall throughput (*i.e.* production of data tokens) is maximized. Since actors of the graphs are related by an affine relation a single variable needs to be computed. The main challenge is to find a suitable factor that will respect the constraint mentioned before. For each possible value a parametric EDF schedulability analysis is made, the analysis is made in two phases: a Quick convergence Processor demand Analysis is computed to find a schedulable configuration and then an optimization finds the best trade-off between utilization of the processor, feasibility and processor demand. To extend their approach to multi-core systems, a best-fit allocation is used as a partitioning scheme.

While CSDF and ADF are models capable of representing dynamic changes on communication rates, they are known at design-time. Therefore, while allocating these tasks into processors, system designers known exactly when changes are going to take place. At the same time, execution times for actors remains unchanged which is not the case for the MC scheduling model we want to adopt.

Variations in communications rates and execution times

Dynamism in communication rates and execution times have been included in Scenario-Aware Data-flow (SDAF) Graphs [96; 97]. The SDF model is enriched thanks to detectors that alert other vertices of changes in the behavior of the application. These vertices change their execution time and communication rate according to the scenario the system is in. SDAFs define semantics capable of performing the same formal verifications that SDFs have. However, regarding time analysis for this model, only the average execution time of the application can be performed and deadlines for vertices are not considered. This is a limitation on the applicability of this model to MC constraints.

The data-flow MoC has been widely used in the design and certification of safety-critical systems. Its mathematical foundation is interesting since properties like deadlock freedom, boundedness in communication channels, starvation freedom can be formally proven. In our context, we are interested in obtaining temporal correctness and resource efficient scheduling (*i.e.* MC scheduling) for this MoC as well. We have demonstrated that existing approaches are capable of scheduling data-flow graphs into mono and multi-core architectures with different constraints (*e.g.* with or without preemptions, with varying communication rates, *etc*). Nevertheless, the scheduling approaches defined for MC data-flow graphs lead to poor resource usage (*i.e.* federated approaches [81] require more processing capabilities than what is actually needed) or exhibit pessimistic assumptions for the LO-criticality tasks.

2.5 Safety-critical systems' dependability

Besides logical and temporal correctness an important aspect that needs to be considered for the development of safety-critical systems is *dependability* [1; 98].

2.5.1 Threats to dependability

Dependability [98] is a generic concept incorporating various attributes such as reliability, availability, safety, integrity and maintainability. While safety-critical systems can be designed in a way that logical and temporal correctness were proven to be obtainable (thanks to data-flow models and real-time scheduling for instance); in the real world systems are often confronted to *threats to dependability*.

In the context of MC scheduling: The discard MC model [36; 8; 37], the most common approach to schedule MC systems, advocates for the interruption of LO-criticality tasks after a timing failure occurs. This has a direct impact on the *availability* of LO-criticality tasks since they are not executed while the system is in the HI-criticality mode. While the discard MC approach has shown to give better schedulability results, delivering a minimum service guarantee is a necessity for the applicability of the MC model into the safety-critical domain. The literature around MC has recognized the necessity to improve the availability (also related to the quality of service) for LO-criticality tasks [99] which has been considered in recent contributions.

Other types of threats to dependability can come from different sources and some of them are naturally unavoidable. For instance, an important aspect that safety-critical system are confronted to, is the fact that these systems are often deployed in hazardous environments [1]. For example, satellites are exposed to radiation coming from the Sun and this has a direct impact on the usability/longevity of hardware components. If no measures are taken to handle these threats to dependability, then systems risk to produce errors that can have tragic consequences.

Therefore, safety-critical systems are confronted to different types of threats that affect the dependability of a system. Nonetheless, system designers are aware of this problem and have proposed means to improve dependability.

2.5.2 Means to improve dependability

One of the main arguments to use the data-flow MoC is actually related to the improvement on dependability it offers. *Errors can be limited during the design-phase* of safety-critical systems [100] thanks to the soundness of this MoC. This is actually one of the main reasons behind the success of industrial tools like SCADE or Simulink that use SDF models.

When we look into MC scheduling, the general solution that has been taken to improve dependability consists in *limiting the non-execution of LO-criticality*. By doing so, the availability rate of LO-criticality tasks remains acceptable for safety standards. We presented in Section 2.3, contributions that are capable of delivering a minimum service

guarantee for LO-criticality tasks. For example the elastic MC model [38; 66] changes the parameters of LO-criticality tasks in order to reduce their utilization in the HI-criticality mode. Semi-partitioned approaches [63; 61; 62] are capable of limiting the mode transition to certain cores and migrate LO-criticality tasks to other cores to allow them to complete their execution. The problem with these enhancements is that they have only been applied to independent task sets. For dependent task scheduling on MC with minimum service guarantee, Pathan [83] has designed a scheduling method that allow LO-criticality tasks to utilize the slack time left by HI-criticality tasks after they have been scheduled. Instead of performing a mode switch everytime a tasks has a TFE, in [101] Burns *et al.* propose to consider tasks that capable of dropping a job in order to balance the workload of the system. This enhancement has been applied to the AMC scheduler and they measure their impact on the acceptance rate of randomly generated task sets.

The approaches we have mentioned have proven to be effective but require to define new scheduling methods. Other means to improve dependability that are orthogonal to scheduling can also be considered in order to deliver a minimum service guarantee. For example in reactive systems where tasks are sampled at certain frequencies, incoherent or missing values are expected to be measured. Weakly-hard real-time tasks [102] can be used in those system since these tasks are capable of tolerating a fixed amount of failures given a number of successive executions. Finally, while recent contributions around MC claim that the quality of service of LO-criticality tasks is improved, there are no methods capable of quantifying an availability rate for these LO-criticality tasks.

In conclusion, safety-critical systems are unavoidably confronted to threats to dependability. Given the environment these systems are executed in, means to improve dependability are necessary. Design methodologies like the data-flow MoC we presented in Section 2.4 have been adopted by system designers to limit errors at design-time. While MC scheduling improves considerably resource usage and allows to incorporate various functionalities into the same computing platform, a minimum service guarantee needs to be ensured for practical safety-critical systems. The literature around MC systems has recognized this limitation. New task models have been proposed to cope with these requirements but new scheduling policies supporting these tasks models need to be developed. We also want to propose a generic method to evaluate the availability of less critical tasks on data-driven MC systems which has not been explored before.

2.6 Conclusion

This chapter presented the context and related works we consider in this dissertation.

To cope with current industrial needs, safety-critical systems need to **(i)** execute *efficiently* in multi-core architectures and **(ii)** make good use of precessing resources by *delivering as many functionalities as possible*. These two trends have a direct impact on the design and thus the deployment of safety-critical systems. Since logical and time correctness need to be ensured for this type of systems, innovation in real-time scheduling and data-flow MoCs have been proposed to cope with these new objectives.

Real-time scheduling has been adapted to support execution in mutli-core architectures and to incorporate tasks with different criticalities into the same execution platform. That way, both necessities for modern safety-critical systems are satisfied. Nevertheless, most contributions handling the MC model have been simplified and are not directly applicable to existing safety-critical systems design methodologies which often define data-dependencies.

On the other hand, while data-flow MoC are used to design and verify safety-critical systems, dynamism in their representation is very constrained. Incorporating mode transitions and different executions budgets to support the MC scheduling model has started to been addressed by the literature, yet current allocation policies often lead to poor resource usage, *e.g.* more processors than what is actually need are required to find feasible schedules.

For this reason, we need to define new scheduling methods capable of satisfying current industrial needs. These methods need to be *efficient*, *logical and temporal correct* and also capable to satisfying *data-dependencies*.

We have also demonstrated that MC scheduling can compromise the availability of the less critical services which is a problem for safety-critical system that require to deliver a minimum service guarantee. Methods to evaluate and enhance the availability for this type of tasks need to be developed, that way MC scheduling can be adopted in the safety-critical domain.

The next chapter establishes the problem statement of this dissertation, we explain and list the problems that need to be addressed in order to obtain an efficient deployment of data-driven applications into MC multi-core systems delivering a minimum service guarantee.

3 Problem statement

Table of Contents

The increasing popularity of the Mixed-Criticality (MC) model on safety-critical systems led to the development of many scheduling algorithms. Since these systems have to respect real-time constraints (*i.e.* deadlines, periods), to ensure a correct execution, many real-time schedulers were adapted to support the MC model. Like it was pointed out in Section 2.3, most MC scheduling approaches in the literature [35] have only considered the independent task set model.

On the contrary, we are interested in designing a scheduler for MC tasks with data dependencies. Incorporating data dependencies on MC scheduling raises new challenges regarding (i) the schedulability of the system in all execution modes and (ii) on mode transitions to higher criticality modes. At the same time, designing scheduling approaches has been the main focus in MC [35], yet we are also interested in the availability of non-critical tasks delivered by these systems. The principle of the MC model is to degrade the execution of non-critical tasks' in favor of HI-criticality tasks, their availability is therefore influenced by this behavior.

In most reactive safety-critical systems, tasks communicate with each other in order to inform different components of changes that occur during the execution of such systems.

For example, in a flight control system, data computed by a navigation functionality is sent to an altitude controller in order to adjust the speed of the propellers. We are interested in incorporating MC aspects to such systems in order to improve resource usage. Specially now that multi-core architectures are widely adopted in the industry, processing capabilities are very promising.

Regarding the availability of non-critical tasks, they are often in charge of delivering end-user functionalities, *e.g.* a satellite sending cellular signals, a drone sending images, *etc*. Executing them is therefore of prime importance to ensure a good quality-of-service for the system. However, due to mode transitions in MC, when the system makes a transition to a HI-criticality mode, all non-critical tasks are less or no longer executed in favor of critical tasks. In consequence, we aim at defining methods to estimate and improve the availability of non-critical tasks.

In this chapter we begin by defining the challenges related to the scheduling of MC data-dependent tasks on multi-core architectures. We then present the limitations we need to address when evaluating the availability of MC systems. Finally, we demonstrate that the discard MC model, widely used in the literature, has limitations when considering the availability of non-critical tasks.

3.1 Scheduling mixed-criticality data-dependent tasks on multi-core architectures

There are two main aspects we have to consider when proposing a MC scheduler for data-dependent tasks on multi-core architectures. On the one hand, data dependencies between tasks need to be respected, *i.e.* the predecessors of a task need to complete their execution before this task can be executed. While defining schedulers that satisfy this condition, we also need to ensure that deadlines for tasks are respected. On the other hand, we are interested in having a MC system with criticality modes and different timing budgets for tasks, in order to take advantage of the computation resources offered by multi-core architectures. Many schedulers satisfying data dependencies have been proposed in the literature and have shown good performances. Nonetheless, the key aspects of the MC model are often missing in these scheduling approaches.

In this section, we show the different approaches used to solve the data-dependent scheduling on multi-core architectures, known to be a *NP*-hard problem. We then demonstrate that scheduling MC tasks with data dependencies is also *NP*-hard. The challenges that are risen by considering the MC model are presented as well.

3.1.1 Data-dependent scheduling on multi-core architectures

To represent data dependencies between tasks, graphs are often used to describe applications: vertices represent tasks or jobs and edges represent dependencies between these jobs/tasks. Like we presented in 2.4, many graph models have been proposed in the literature. In safety-critical systems, the communication model that is often deployed, consists in having at least one execution of all the tasks' predecessor before such task can be executed. In other words, a task will not be able to execute if all its predecessors have not completed their execution. This communication model has been widely used in the Synchronous Dataflow (SDF) model [3] and in operational research with Directed Acyclic Graph (DAG) scheduling [103].

While SDF allocation on uni- and multi-core processors has seen many contributions since it was first introduced, deadlines and periods are rarely considered on multi-core architectures. Most allocation techniques optimize a given metric, like memory buffers [104] or throughput [105]. In our context, where deadlines and periods need to be respected, unfolding the SDF to obtain a DAG can be used to find feasible schedules where tasks respect their deadlines and periods [69]. As opposed to multimedia applications, the type of system's specifications we are targeting do not have large numbers of executions of tasks during one period of the application (tenths as opposed to thousands in multimedia applications). Therefore, we consider that applications running in the safety-critical system are or can be transformed into DAGs. Each vertex is a task of the application and edges are the dependencies between these tasks.

Deadline satisfaction problem of data-dependent tasks: Scheduling a DAG where tasks have an arbitrary execution time, in a multi-core architecture, while minimizing its completion time, is known to be a *NP*-complete problem [106; 6]. The fact that completion time of the DAG is minimized, is a desired aspect to design a scheduling approach. If the algorithm tries to minimize the completion time (C_{max}), it can also to satisfy a deadline (D) given to the graph, if we have $C_{max} \leq D$. The problem are not equivalent, but makespan minimization can respect deadlines. Additionally, *it has been proven that no optimal scheduler in polynomial time can be found to solve this problem* [106; 6].

On-line scheduling approaches: Existing approaches have tried to transform the problem of scheduling DAGs, into scheduling independent task sets. However this transformation requires more resources than what is actually needed to solve the original scheduling problem. By computing offsets and deadlines for each task of the DAG, a global real-time scheduler (*e.g.* G-EDF, G-RMS) can be used to schedule the system and still

satisfy the original data dependencies. Sufficient conditions to determine if a set of data-dependent tasks will be schedulable using these approaches have been defined [73; 74; 75]. Nonetheless, since the allocation of task to processor is not known in advance, the number of tasks that can interrupt the execution of another task has to be upper bounded, leading to an overestimation of tasks' response times. This overestimation leads to a poor resource usage of the platform, since to satisfy sufficient conditions, processor time where no tasks are running needs to be introduced in the system. While finding tighter bounds on the response times of tasks is a current research perspective [107], overestimated response time is problematic in our context of MC systems: we want to take advantage of the computation resources offered by the platform.

Off-line scheduling approaches: In 2.4.2 we presented another family of contributions that tackle the scheduling problem for data-dependent tasks, called List Scheduling (LS) heuristics. LS is an interesting solution since it can find feasible static schedules in pseudo-polynomial time. This family of scheduling techniques has a *greedy* behavior: a *ready* task will always be scheduled if there is an available processor. Consequently, LS minimizes the makespan of the scheduled DAG while still taking advantage of the platform's execution capabilities. LS computes static scheduling tables prior to runtime. Only the application of the LS heuristic will determine if the system is schedulable or not. Nevertheless, the heuristics are quite efficient even when the number of vertices and edges is large [72].

It has been shown that LS has a *worst-case approximation ratio* (or speedup factor) of $(2-\frac{1}{m})$, when comparing it to an optimal clairvoyant algorithm [103]. The approximation ratio is calculated between the completion time of the LS algorithm and an optimal clairvoyant algorithm. If an optimal algorithm can generate an m-processor schedule of with a completion time C_{max}, then LS generates a schedule of length $\leq (2-\frac{1}{m}) \times C_{max}$. A more recent result has demonstrated that $(2-\varsigma)$ is likely a lower bound on the worst-case approximation ratio of any polynomial-time heuristic, where ς is a constant close to 0. This makes LS the closest best known heuristic to solve the scheduling problem in polynomial time [108]. The benchmark studies in [72] list the most efficient LS heuristics and demonstrate that, LS can be adapted to efficiently solve different and more complex scheduling problems. Communications costs between each vertex, heterogeneous architectures, release dates, among other extensions, have been solved thanks to LS. However, most LS heuristics have not taken into consideration tasks with different timing budgets and systems with various modes of execution, which are the foundation of the MC model.

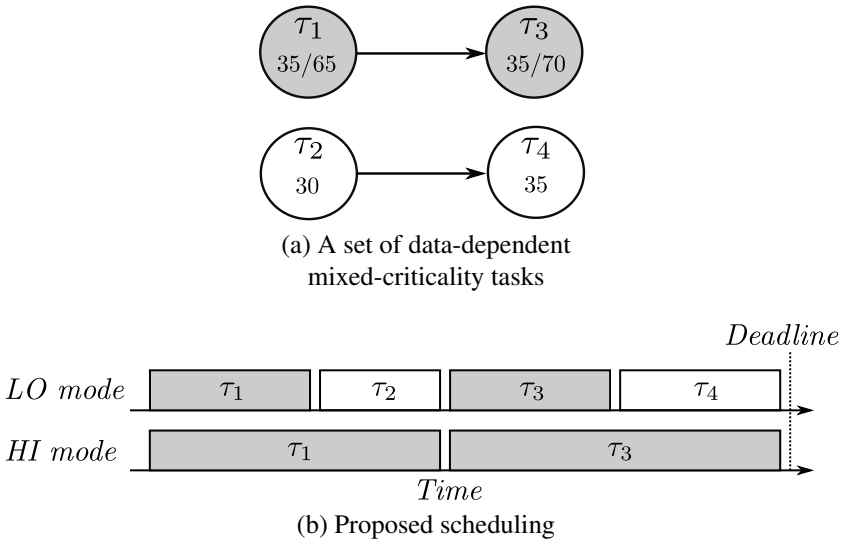
To solve the scheduling problem of DAGs, two main categories of contributions have shown promising results in the literature. The first category, transforms the DAG into

an independent tasks set by computing offsets and deadlines to implicitly respect data-dependencies [73; 74; 75; 107]. These approaches are interesting since the system is schedulable if the sufficient conditions are satisfied. Nonetheless, these approaches *require more processing capabilities than what is needed to satisfy sufficient conditions* for global real-time schedulers like G-EDF or G-DMS. This is an important limitation if a MC scheduler is to be defined using these approaches. The second category, the LS heuristics [103; 72], have interesting properties that are desirable when defining a MC scheduler. The heuristics are fast when trying to compute feasible schedules due to their polynomial complexity. The completion time of the DAG is minimized and therefore, it can implicitly respect a deadline. Finally, the greedy behavior of the heuristics takes advantage of the computation resources. By basing the MC scheduler on LS heuristics we can take advantage of the computation resources and still respect real-time constraints. Nonetheless, the problem of scheduling MC tasks with data dependencies is even more complex than the problem of scheduling DAGs, tasks have different timing budgets and mode transitions can occur during the system's execution.

3.1.2 Adoption of mixed-criticality aspects: modes of execution and different timing budgets

While existing LS heuristics have proven to be quite efficient when computing feasible schedules [72], only a few works have proposed to schedule data-dependent MC tasks using these heuristics [79; 80]. Like it was explained in Section 2.3, in the MC model, as defined by Vestal in [4], a system has different criticality levels which correspond to modes of execution. This decomposition allows to consider different timing budgets (corresponding to different WCET estimation) for tasks executed in the MC system. In a HI-criticality mode, tasks can have a larger timing budget compared to the one given in a lower criticality mode. In the previous section we explained that scheduling data-dependent tasks was already a very complex problem, therefore considering MC tasks adds a layer of complexity to the scheduling problem. Schedulability in all modes of execution but also during the transition to higher criticality modes need to be respected when defining a MC scheduler.

Problem 1 - Mixed-criticality scheduling problem: scheduling a system with data dependencies, in the form of DAGs, while respecting *MC constraints*, is *NP*-hard. We will demonstrate its complexity by showing how data dependencies only render the scheduling problem more difficult than regular MC scheduling.



(a) A set of data-dependent mixed-criticality tasks

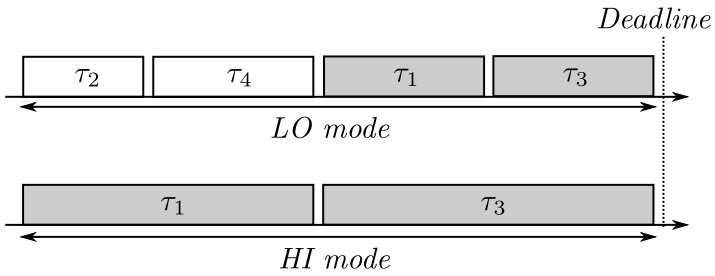
(b) Proposed scheduling

Figure 3.1: A task set schedulable in a dual-criticality system

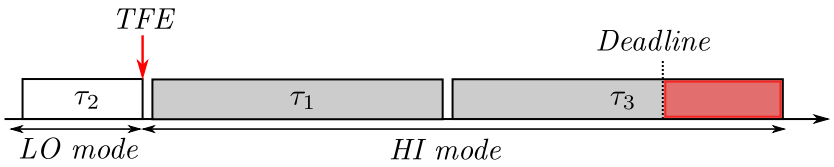
MC scheduling needs to respect the following constraints: first, the system needs to respect deadlines in all criticality modes, and second, the system must respect deadlines when the system makes a transition to a higher criticality mode.

Sub-problem 1.1 - Schedulability in all modes of execution: the systems needs to be schedulable in all its modes of execution. An example of a schedulable task set is illustrated in Fig. 3.1. This figure illustrates a dual-criticality system (Fig. 3.1a) with a task set composed of four data-dependent tasks. Edges represent dependencies between tasks and each task is annotated with its WCET(s). Tasks illustrated in gray are considered to be HI-criticality tasks and tasks illustrated in white are LO-criticality tasks. In the LO-criticality mode (LO mode), all four tasks are executed in the system, whereas in the HI-criticality mode (HI mode), only the gray tasks are scheduled with an extended timing budget (*i.e.* 65 TUs for τ_1 and 70 TUs for τ_2). The proposed scheduling of the MCS is presented in Fig. 3.1b, since the deadline is respected in the LO mode, but also in the HI mode, the system might be considered as schedulable. However, in the next paragraph we show that the schedulability in all criticality modes is not sufficient for MC systems.

Sub-problem 1.2 - Schedulability in case of a mode transition: mode transitions to higher criticality modes [109] often carry timing extension budgets for HI-criticality tasks, which could lead to a deadline miss. A mode transition to a higher criticality mode is triggered by a Timing Failure Event (TFE), *i.e.* if a task did not complete its execution within the timing budget that was given. Having *safe mode transitions* for the system, *i.e.* a mode transition that respects deadlines, is a desired property when designing a scheduling algorithm for MC systems. In Fig. 3.2, we illustrate the problem of safe mode transitions



(a) Proposed scheduling tables



(b) Scenario with a TFE in τ_2

Figure 3.2: A deadline miss due to a mode transition

when scheduling a MC system. We consider the same system of Fig. 3.1a, but this time, another proposed scheduling for the two modes of execution is presented in Fig. 3.2a, this scheduling respects precedence constraints since τ_1 (resp. τ_2) is executed before τ_3 (resp. τ_4). Nonetheless, with these new scheduling tables, if a TFE occurs on task τ_2 (Fig. 3.2b), the mode transition causes a deadline miss on task τ_3.

Proof. NP-hardness: the scheduling problem for a task set represented by a DAG, where vertices have an arbitrary execution times is known to be *NP*-complete [106; 6] and the problem of MC task scheduling is known to be *NP*-hard [8; 39].

Sub-problem 1.1 - When we consider a MC system, the schedulability of the DAG (or a subset of it, some tasks can be interrupted depending on the criticality mode) needs to be guaranteed on all modes of execution. In other words, we have to solve the scheduling problem for a DAG M times, where M is the number of criticality levels, known to be *NP*-complete.

Sub-problem 1.2 - In addition, when the system makes a mode transitions to higher criticality modes, tasks that are executed in both criticality modes need to respect deadlines and precedence constraints. This makes the scheduling more complex since timing budgets can increase from one criticality mode to another. Ensuring this condition has been shown to be *NP*-hard.

Therefore, scheduling MC data-dependent tasks, where sub-problem 1.1 and 1.2 need to be solved is *NP*-hard. □

To summarize, the proposed scheduler for data-dependent tasks in a MC systems needs to satisfy: (i) the schedulability in all the criticality levels and (ii) ensure safe mode tran-

sitions to higher criticality modes.

Scheduling approaches based on LS to schedule dual-criticality MC data-dependent tasks in the form of DAGs have been proposed in the literature [79; 80; 81]. Computing static scheduling tables by defining a priority ordering for MC tasks with precedence constraints is proposed in [79]. In this contribution, LS is used to find a priority ordering of tasks when the allocation decision is made. A priority ordering is found for HI-criticality tasks and for LO-criticality tasks as well. These orderings are independent from each other. Nonetheless, HI-criticality tasks' allocation is prioritized over LO-criticality tasks in the LO execution mode, ensuring safe mode transitions. However, this approach only considers mono-core architectures. The adaptation to multi-core architectures was proposed in [80]. The problem of scheduling multiple DAGs in a single MC system was tackled in [81]. The author proposes to transform the problem of scheduling multiple DAGs on a multi-core architecture, to schedule a single DAG on a cluster of cores of the multi-core architecture. The approach proposed in [81] relies on the method presented in [80] to schedule a DAG on its cluster.

Sub-problem 1.3 - Limits of existing approaches: as it is shown in [80; 81], adapting LS to handle MC aspects on data-dependent tasks is possible and has been proven to be correct (*i.e.* schedulability and safe mode transitions are delivered). Nonetheless, in these contributions only theoretical results have been presented, making it difficult to assess the performances of these approaches, *e.g.* checking if a scheduling can be found for different system configurations with various utilization rates. Prioritizing the allocation of HI-criticality tasks whenever they are ready is also a pessimistic condition. LO-criticality tasks also have dependencies among them, therefore if HI-criticality tasks preempt LO-criticality tasks too often, these preempted tasks (and their successors) will have a large response time potentially leading to a deadline miss. With the priority ordering defined in [80; 81], the preemption of LO-criticality tasks can occur frequently if HI-criticality are constantly being activated for example. This pessimistic condition has been used in order to ensure safe mode transitions, but we will demonstrate that this pessimism can be lifted.

Another aspect that we have to address is the generalization of the scheduling method to support an arbitrary number of criticality levels. This makes the scheduling problem more difficult, because safe mode transitions need to be ensured in more than two criticality modes. As a matter of fact, industrial standards often define more than two criticality levels. For example, railroad systems have four levels, while airborne systems have five.

In conclusion, a correct scheduling of MC data-dependent tasks on multi-core architectures is a complex problem: schedulability in all criticality modes needs to be respected

but also when mode transitions take place. While LS heuristics are a good comprise in terms of performance and optimality, existing approaches that have taken this direction are too restrictive: execution of HI-criticality tasks is prioritized systematically. Better conditions to promote HI-criticality tasks' allocation in the LO-criticality mode should be found. Additionally, existing approaches only consider dual-criticality systems, whereas standards for safety-critical systems often define four or more criticality levels. Therefore, we are also interested in proposing a scheduling method for a MC system having an arbitrary number of criticality levels.

3.2 Availability computation for safety-critical systems

While defining a correct scheduler for mixed-criticality data-dependent tasks is of prime importance for our works, we are also interested in the *availability* of services delivered by these tasks. As a matter of fact, most contributions regarding MC systems have been focused on scheduling methods considering the *MC discard* approach [36; 8; 37]. After a mode transition takes place in the system, timing budgets given to high criticality tasks increase. The MC discard approach has been the most dominant hypothesis considered when defining MC schedulers [35]. This approach has shown that interrupting the execution of non-critical tasks after the mode transition takes place gives a higher schedulability ratio for MC systems. While non-critical tasks are not considered as highly critical, they are often in charge of the quality-of-service (QoS) of the system: their execution is important as well [110]. In practice, interrupting non-critical services indefinitely gives a very poor availability for MC systems. Recent trends in MC scheduling acknowledge this limitation and propose to guarantee a minimal execution of LO-criticality tasks in the HI-criticality mode to deliver a minimum QoS [38; 65]. Nonetheless, these approaches have defined specific scheduling methods that are not applicable to data-dependent task sets. In this section we look into the requirements to define availability estimations for MC systems following the discard approach. At the same time, we show the limitations that the MC discard approach entails regarding the availability of non-critical tasks.

Problem 2 - Estimating availability rates: the evaluation of availability in MC systems can be performed in different manners, but it boils down to knowing how many times a task runs, divided by the number of times the system was executed (in LO and HI-criticality modes):

$$A(\tau_i) = \frac{\text{Executions of } \tau_i}{\text{Executions of the system}}. \tag{3.1}$$

This availability equation can be solved numerically if the model of the system allows it. Otherwise, we need to perform simulations of the system's execution in order to obtain an estimated value of the availability. In both cases, the most popular MC discard model of the literature [36; 8; 37] is missing necessary aspects to solve Eq. 3.1: (i) we need to know how often non-critical tasks are executed and interrupted, but (ii) we also need to propose a method to restore them in case a mode transition to a higher criticality mode occurred. The recovery of non-critical tasks is necessary because staying in a HI-criticality mode indefinitely would give very bad results on the availability of non-critical tasks. At the same time, safety-critical systems are often used in hazardous environments where human maintenance is impossible or very difficult.

In order to solve Eq. 3.1, the first information we need to have is *how many times a task τ_i is executed?* Since we are in a MC discard system, the execution of non-critical tasks is dependent on the criticality level the system is in, *i.e.* if the system is in (or makes the transition to) a HI-criticality mode, then non-critical tasks are not executed. Thus, we need to know *how often a TFE takes place in a MC system?*

Subproblem 2.1 - Incorporating a fault model: to determine how often a system switches to a HI-criticality mode, we need to incorporate a fault model for tasks executed in the MC system. This fault model should assign failure probabilities to each task, *i.e.* the probability a task will provoke a TFE and therefore a mode transition to a higher criticality mode. The occurrence of failures within a period of time is actually used in safety-critical standards to certify that a functionality meets a certain criticality level. For example, the airborne standard DO-178B [111] defines five software levels (also called "assurance" levels), where each one of them has a Failure Rate. Highest level A has a rate of $10^{-9}/h$, level B $10^{-7}/h$, and so on.

Sub-problem 2.2 - Incorporating a mode recovery: while having information about failure probabilities on tasks executed in the MC system is necessary for the availability computation, we also need to introduce a recovery mechanism for these tasks. Most approaches of the literature of MC systems only limit themselves to ensure the safe mode transition to a higher criticality level [35]. Nevertheless, for safety-critical systems, staying in the HI-criticality mode is a very important limitation since non-critical tasks are no longer executed, having an undesired impact on their availability. The complete interruption of non-critical tasks is a problem, since these tasks are often in charge of the QoS for safety-critical systems, *e.g.* if an exploration drone fails to send images/videos, the mission fails even if HI-criticality tasks allowed the drone to avoid crashing. We need to avoid human-maintenance since safety-critical systems are often used in hazardous environments, making this type of maintenance even impossible in some cases.

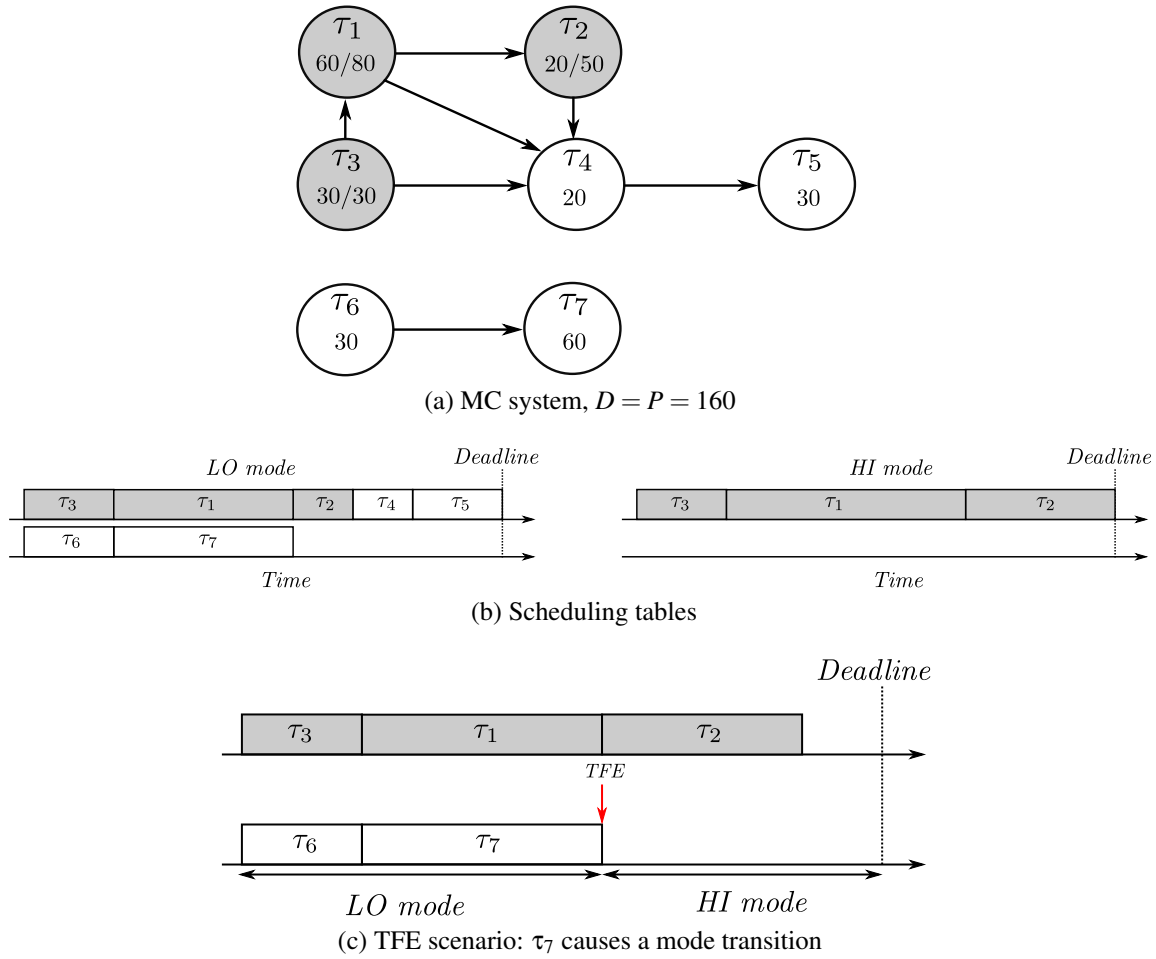
Defining the fault model and the recovery process in the MC system will allow us to estimate an availability rate for non-critical tasks quite efficiently: we can assess numerically the availability rates for non-critical tasks. Nonetheless, in most MC contributions [36; 8; 37], the fault propagation model considered is quite simplistic, in the sense that actual safety-critical systems often incorporate mechanisms to limit the impact of faults on the system. These mechanisms have a positive impact on the availability of non-critical tasks and we would like to incorporate them in our analysis as well.

3.3 Availability enhancements - Delivering an acceptable Quality of Service

With the necessary information regarding the system's failures and the recovery mechanism of non-critical tasks, we can propose methods to compute an availability rate. However, in the discard MC model [36; 8; 37] we see limitations regarding the obtainable availability.

Problem 3 - The discard MC model degrades availability significantly: sharing an architecture with functionalities of different criticalities has an impact on their availability. The problem is that even if functionalities are independent from each other, after a TFE occurs, the whole system makes a transition to a HI-criticality mode and LO-criticality tasks are no longer executed. Fig. 3.3 illustrates this scenario. We consider a dual-criticality system presented in Fig. 3.3a, the deadline and period of the system is set to 160 TUs. The scheduling tables for the HI and LO-criticality mode are presented in Fig. 3.3b, we have a dual-core processor to schedule the system. We suppose a TFE takes place when τ_7 is running (Fig. 3.3c), at this point the system switches to a HI-criticality mode, LO-criticality tasks are discarded and HI-criticality tasks have an extended timing budget. In this example, the problem of having a shared architecture can be seen with tasks τ_4 and τ_5 that are interrupted but are not dependent on the execution of task τ_7 (*i.e.* there are no data dependencies between τ_7 and τ_4, τ_5). In addition, the TFE took place during the execution of τ_7 that is a LO-criticality task but HI-criticality task τ_2, had its timing budget increased: it is possible that τ_2 did not need this timing extension. Having a mode transition whenever a task has a TFE is too pessimistic, specially when designers need to guarantee that an availability rate is met. It is thus necessary to limit fault propagation to limit the impact of resource sharing on services' availability.

Sub-problem 3.1 - Limiting the number of mode transitions to higher criticality modes: mode transitions to higher criticality modes are at the core of MC scheduling, and



(a) MC system, $D = P = 160$

(b) Scheduling tables

(c) TFE scenario: τ_7 causes a mode transition

Figure 3.3: Interruption of non-critical tasks after a TFE

need to be conserved in order to guarantee a safe execution of HI-criticality tasks. Nevertheless, incorporating a more precise fault propagation model would allow us to prevent unnecessary mode transitions to higher criticality modes. As a matter of fact, safety-critical system often integrate mechanisms to contain, mask or limit faults. For example Real-Time Operating Systems (RTOSes) [112] are conceived in a way that software components are isolated from each other, by partitioning the memory for example. That way, if a task has a failure, the kernel is capable of interrupting only faulty services and have the rest of components running normally. Other types of measures like using design patterns, allow safety-critical system to be fault tolerant.

Sub-problem 3.2 - Incorporating availability enhancements : Incorporating software or hardware mechanisms to improve the availability of tasks, thanks to fault tolerance, can have an impact on the system's model, invalidating previous methods that compute availability rates for non-critical tasks. If we consider for example, weakly hard real-time tasks [102], to know if the task is in an operational state we need to keep track

of a given number of executions. To estimate an availability rate for these cases, we need to adapt the methods proposed. For instance, simulations of the system's execution may be required.

3.4 Hypotheses regarding the execution model

The following section presents a summarized table of the hypotheses that are made in order to solve the abovementioned problems.

Table 3.1: Hypotheses of our execution model

Data-dependencies	
Graph	Directed acyclic graphs.
Vertices	Real-time tasks.
Edges	Precedence constraints.
Real-time	
Execution time	Tasks use all their timing budget.
Period	Tasks are periodic.
Deadline	Hard deadlines. Constrained or implicit.
Architecture	
Processors	Homogeneous.
Communication costs	Considered in tasks execution time.
Mixed-criticality	
Criticality levels	Two or more criticality levels.
Timing budgets	Monotonically increasing.
Degradation	Discarding lowest-criticality tasks.
Fault model	
Failures	Timing failure events.

A more detailed execution model capturing all the elements of our research context is presented in the next chapter (Section 4.1).

3.5 Conclusion

We have presented in this chapter the main issues regarding the scheduling of MC tasks with precedence constraints, as well as the importance of ensuring an availability rate to deliver a proper QoS.

We started by demonstrating that scheduling MC tasks with precedence constraints in a multi-core architecture is a *NP*-hard problem (**Problem 1**). Without MC constraints, scheduling data-dependent tasks was already *NP*-complete, we are therefore tackling a

more difficult problem. The schedulability in all criticality modes needs be ensured when designing a new MC scheduler (**Sub-problem 1.1**), but we also need to guarantee that mode transitions do not cause a deadline miss (**Sub-problem 1.2**). The most promising approaches of the literature have adapted a heuristic called List Scheduling in order to allocate MC tasks to multi-core processors. These heuristics are known to have the best performance in terms of time complexity (polynomial) while having a known speed-up factor of $(2 - \frac{1}{m})$. They also take advantage of the processing capabilities of the targeted architecture due to their work-conserving behavior. Nonetheless, existing contributions have some limitations we need to address (**Sub-problem 1.3**): (i) find better conditions to schedule HI-criticality tasks in LO-criticality modes, (ii) use less processing cores when the problem becomes more complex; and (iii) generalize the scheduling to support an arbitrary number of criticality levels for the system.

While defining a correct and efficient scheduler is a main concern for our works, we are also interested in the availability offered by a MC systems (**Problem 2**). Non-critical tasks in safety-critical systems are often in charge of the QoS, for this reason we are interested in knowing how often these tasks are executed in MC systems. To define methods for estimating an availability rate, we need to know how often a non-critical task is interrupted (**Sub-problem 2.1**). In most MC contributions, after the system is a HI-criticality mode, there is no recovery mechanism for non-critical tasks (**Sub-problem 2.2)**.

In the discard MC model, when a timing fault occurs, the whole system makes a mode transition to a higher criticality mode, interrupting the execution of non-critical tasks. In the context of multi-core architectures this is very limiting since applications that are *non-dependent* on the failing tasks are also interrupted (**Problem 3**). We want to address this problem by proposing a more detailed fault propagation model that could allow us to limit the number of mode transitions (**Sub-problem 3.1**). Another aspect that has not been considered in the literature of MC systems, are reliability mechanisms often deployed to ensure that an availability rate for the system is met (**Sub-problem 3.2**). The necessity to evaluate and potentially improve the availability of non-critical tasks is a necessity for MC systems minimum service guarantees are required by safety-critical standards.

The next chapter presents our contributions tackling these problems: scheduling of MC data-dependent tasks, as well as methods to evaluate and improve availability on data-driven MC systems.

4 Contribution overview

Table of Contents

The previous chapter detailed the problem we need to address in order to define a correct and efficient MC scheduler for data-dependent tasks (**Problem 1**). We also presented the different requirements needed to compute (**Problem 2**) and improve the availability (**Problem 3**) of non-critical tasks.

In this chapter, we begin by defining the MC task model used throughout our contributions. This model, called Mixed-Criticality Directed Acyclic Graph (MC-DAG), takes into account the different elements and constraints of our context: data dependencies, MC timing budgets and real-time constraints (deadline and periods). In addition, our model allows us to propose evaluation methods for the availability rate of non-critical tasks by including a failure model.

An overview of our contributions is illustrated in Fig. 4.1. When it comes to MC systems our first contributions tackle the problem of scheduling data-dependent MC tasks on multi-core architectures, represented by the blue box in the figure. These contributions tackle **Problem 1**. We designed a meta-heuristic, called MH-McDAG, to find feasible and correct schedules of MC-DAGs. Few modifications are required in order to obtain different implementations of MH-McDAG. We have also managed to generalize our

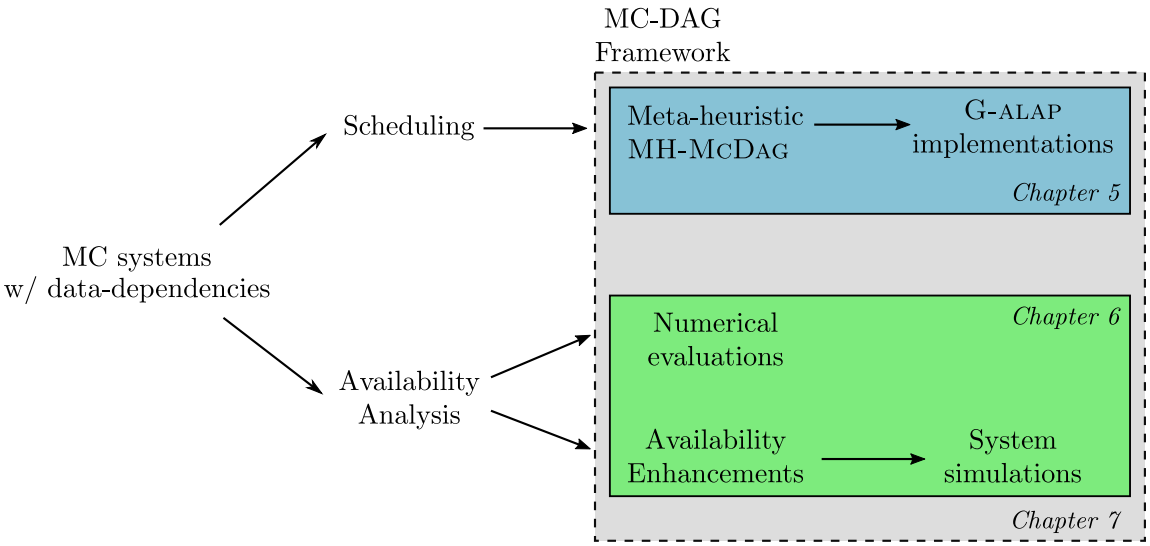


Figure 4.1: Contribution overview

multi-periodic MC-DAG scheduler to handle an arbitrary number of criticality levels: the meta-heuristic can be recursively generalized to support more than two levels of criticality.

Our methods developed to solve **Problem 2**, the evaluation of the availability for MC systems, are also described in this chapter. In Fig. 4.1, we represent this contribution with the green box. If the system allows it, we apply formulas to compute an availability rate: these are the numerical evaluations. These first evaluations led to the incorporation of enhancements to the MC model, in order to overcome **Problem 3**. These improvements are twofold: we incorporated a more detailed fault propagation model and considered fault tolerant mechanisms. Nonetheless since the execution model differs from simple real-time tasks, the availability evaluation requires us to perform system simulations.

Finally, we present the framework that was developed in order to experimentally evaluate our contributions. The framework is represented with the dotted box of Fig. 4.1 since it contains the implementations of our various contributions. Besides the implementations of our meta-heuristic, we have also implemented the scheduling approaches of the literature for MC-DAGs [80; 81] in order to compare us to the state-of-the-art. The framework also includes model transformation rules to perform system simulations in order to evaluate availability rates. A generator of random unbiased MC systems with data-dependent tasks is also included in the framework: this generator is of prime importance to assess the performances of our scheduling methods.

4.1 Consolidation of the research context: Mixed-Criticality Directed Acyclic Graph (MC-DAG)

In this section we present the model developed incorporating all the elements of our context: data-dependent tasks, MC timing budgets and our fault model. Like we explained in 3.1.1, to incorporate data-dependencies between tasks, we assume that applications can be modeled in the form of DAGs.

A Mixed-Criticality System (MCS) is defined by the tuple $\mathcal{S} = (\mathcal{G}, \mathcal{CL}, \Pi)$.

- $\mathcal{G}$ is the set of applications executed by the MCS. The definition of an *application* is given in the next paragraph.
- $\mathcal{CL} = \{\chi_1, \ldots, \chi_n\}$ is the set of criticality levels of the system. We consider this set is ordered, *i.e.* and operator $\prec$ can be defined such as $\chi_1 \prec \cdots \prec \chi_\ell \prec \chi_{\ell+1} \prec \cdots \prec \chi_n$. The transition to a higher criticality mode is progressive: if the system is in χ_ℓ mode, it will switch to $\chi_{\ell+1}$ mode.
- Π is the homogeneous multi-core processor, *i.e.* all processors have the same speed for all the tasks in the system. $|\Pi| = m$ is the number of cores available on this platform. Data can be sent through the cores thanks to an interconnect bus.

A Mixed-Criticality Directed Acyclic Graph (MC-DAG), $G_j \in \mathcal{G}$, represents an *application* being executed in the system $\mathcal{S}$. It is defined by the following tuple: $G_j = (V_j, E_j, D_j, P_j)$.

- V_j is the set of vertices of the MC-DAG. Each vertex is a MC task executed by the application.
- $E_j \subseteq (V \times V)$ is the set of edges between two tasks. If $(\tau_i, \tau_j) \in E_j$, then task τ_i must finish its execution before task τ_j can start. A vertex is ready to be executed as soon as all of its predecessors have been executed. We define $succ(\tau_i)$ (resp. $pred(\tau_i)$) the set of successors (resp. predecessors) of a task.

  The existing model of MC-DAGs described in [80; 79; 81] is closed to ours. Nonetheless, it has restricted communications coming from a lower-criticality level for safety reasons: if a low-criticality predecessor of a high-criticality task does not produce its output, then the high-criticality task would not be able to execute. However, low-to-high communications often take place in safety-critical systems, one example is the monitoring the execution of tasks to determine if there are errors on the values

computed by tasks. Therefore, in our model low-to-high communications are allowed and in Chapter 5 we explain how the scheduling can take into account this type of communication.

- $D_j \in \mathbb{N}^+$ is the deadline of the graph, *i.e.* all vertices of the MC-DAG need to be executed before this deadline.

- $T_j \in \mathbb{N}^+$ is the period of the graph, *i.e.* vertices without predecessors become active again once this period has been reached. We have the following relation $D_j \leq T_j$.

Each vertex of the MC-DAG corresponds to a MC task, $\tau_i \in V_j$, defined as follows $\tau_i = (\chi_i, C_i(\chi_1), \ldots, C_i(\chi_\ell), p_i(\chi_1), \ldots, p_i(\chi_{i-1}))$.

- $\chi_i \in \mathcal{CL}$ is the criticality level of the task.

- $C_i(\chi_1), ..., C_i(\chi_n)$ is the set of WCETs of the task. $\{C_i(\chi_\ell) \in \mathbb{N} \mid \forall \chi_\ell \succ \chi_i, C_i(\chi_\ell) = 0\}$ since we are in the discard MC model, the task is not executed on criticality levels that are higher than χ_i.

  We assume that $C_i(\chi_n)$ is monotonically increasing when n increases, this corresponds to the observation of [4], the higher the criticality level is, the more overestimated the WCET is. We also assume that the communication time is included in the WCET of a task.

- $p_i(\chi_1), \ldots, p_i(\chi_i)$ is the set of failure probabilities for task τ_i. Each $p_i(\chi_n) \in [0;1]$ is the failure probability of task τ_i in the mode $p_i(\chi_\ell)$, *i.e.* the probability the task can cause a Timing Failure Event (TFE). These probabilities are deduced from the WCET estimation analysis [113]. In order to perform the availability analysis for our MC systems, this information needs to be provided.

- $j_{i,k}$ is the *job* of task τ_i. In a multi-periodic system a task can have multiple activations, $j_{i,k}$ is the k-th activation of task τ_i.

In Fig. 4.2 we illustrate an example of a MCS system. For this example, we suppose we have the following criticality levels, $\mathcal{CL} = \{L_1, L_2, L_3\}$, L_3 being the most critical level. The system presented, is composed of two MC-DAGs, G_1 and G_2, each one of them has a different deadline and period. Tasks are annotated with their WCETs (if $C_i(\chi_\ell) = 0$, we do not represent the value) and their failure probabilities. The criticality level of a task is illustrated by the gray-scale: dark gray means the task has a L_3 criticality level, white means it has a L_1 criticality level. Edges represent the data dependencies between the

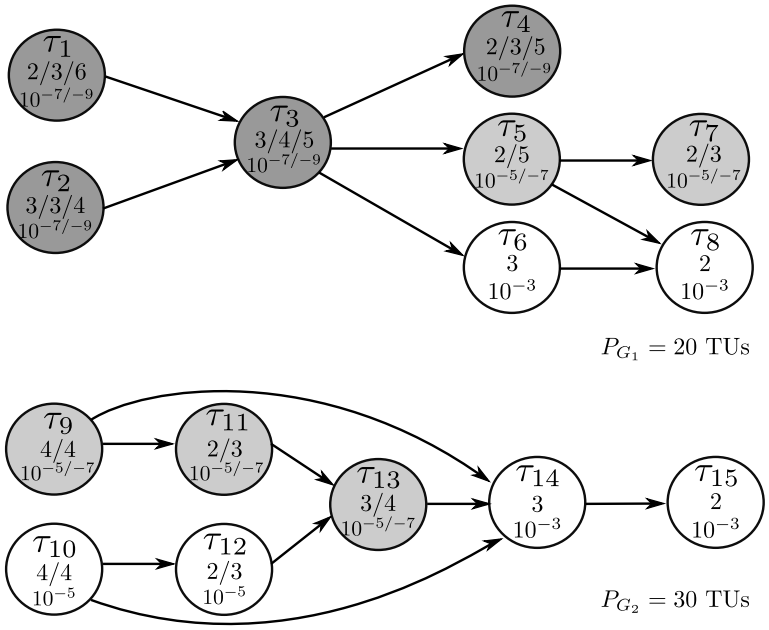


Figure 4.2: Example of a MCS $\mathcal{S}$ with two MC-DAGs

tasks. As we can see, we have a communication that go from tasks of level L_1 to tasks belonging to level L_2: τ_{12} communicates with task τ_{13} which would be restricted in the MC-DAG model of [80; 79; 81].

4.2 Scheduling approaches for MC-DAGs

Like we demonstrated in the previous chapter (section 3.1.2), the **MC Scheduling of MC-DAGs (Problem 1)** is a *NP*-hard problem. Promising approaches based on LS to solve this problem have been proposed in the literature [80; 81] they handle the **Schedulability in all modes of execution (Sub-problem 1.1)** and also the **Schedulability in case of a mode transition (Sub-problem 1.2)**. Nonetheless, there are **Limits of existing approaches (Sub-problem 1.3)**: **(i)** the priority ordering used to schedule the MC systems, systematically prioritize high-criticality tasks in all criticality modes. At the same time, **(ii)** when multiple MC-DAGs are being scheduled, the existing approaches create clusters of cores in order to reduce the problem of scheduling multiple MC-DAGs to scheduling a single MC-DAG in its cluster. This leads to poor resource usage since more cores are required in order to obtain feasible schedules. The last limitation of these approaches is the fact that they are **(iii)** applicable only to dual-criticality systems and in safety-critical standards more than two criticality levels are often used.

We designed a meta-heuristic, called MH-McDAG, capable of solving the scheduling problem by respecting the schedulability in all criticality modes and by enforcing HI tasks' execution at precise instants in order to have safe mode transitions. Our implementations of MH-McDAG compute static scheduling tables, one for each criticality mode: the HI-criticality mode is scheduled first, and in the LO-criticality scheduling, HI-criticality tasks can preempt LO-criticality tasks in order to have safe mode transitions. Similarly to the approaches of [80; 81], our scheduling strategies are list-based but the tables computed are Time Triggered (TT). We have improved the constraints enforced on high criticality tasks to ensure safe mode transitions. Chapter 5 gives a detailed description of the meta-heuristic and its implementations. The recursive generalization to support an arbitrary number of criticality levels is also described in that chapter.

Global generic implementations of MH-McDAG for $\mathcal{CL} = \{LO, HI\}$ and $|\mathcal{G}| = N$

To schedule an arbitrary number of MC-DAGs on a multi-core architecture, we implemented MH-McDAG following a global approach, *i.e.* all tasks of all MC-DAGs can be scheduled in all cores. The targeted architecture is homogeneous and communication costs are assumed to be accounted for in the execution time of tasks. The first instance of our implementation uses a priority ordering based on the *laxity* of each task. The laxity of a task is given by the difference between the deadline of the task, the current instant t and the remaining execution time. To prioritize tasks that have an important number of successors, we calculate virtual deadlines for each vertex of the MC-DAGs. To respect safe mode transitions to the HI-criticality mode, HI-criticality tasks can preempt LO-criticality tasks during the computation of the LO-criticality scheduling table. Our experimental results comparing our heuristic to the existing method of the literature [80; 81] showed that we outperform the state-of-the-art in terms of schedulability for randomly generated MC-DAGs. The comparison is done supposing a periodic activation of MC-DAGs which also benefits the federated scheduling approach [81]: offsets between MC-DAGs activations are non-existent which increases the resource usage of the platform. A limit to our laxity-based scheduler is the number of preemption it entails, for this reason the second instance of our meta-heuristic defines a priority ordering based on G-EDF. Experimental results have shown that this instance generates up to 100 times less preemptions and performs better than existing approaches of the state-of-the-art in the majority of cases.

Generalization of MH-McDAG for $|\mathcal{CL}| = M$ and $|\mathcal{G}| = N$

The meta-heuristic we developed to schedule MC-DAGs in dual-criticality systems can be generalized to support an arbitrary number of criticality levels. This generalization is done by recursively respecting activation times of high-criticality tasks. However, new

conditions on the finish time of high-criticality tasks need to be respected in order to allow timing budget extensions. We implemented this new meta-heuristic basing it on the laxity of each task. The generalization of our previous heuristic consists in computing the scheduling tables starting with the highest criticality levels first. By doing so, in the lowest criticality mode, the most critical tasks will have enough processing time to complete their execution if TFEs take place in all the criticality modes. To the best of our knowledge, our approach is the only one that has generalized the scheduling of multiple MC-DAGs on a system with an arbitrary number of criticality levels.

Works related to the scheduling of MC-DAGs into multi-core architectures have been published in two different conferences: at the International Conference on Reliable Software Technologies (Ada-Europe) in 2017 and at the Real-Time Systems Symposium (RTSS) in 2018. Details about these contributions are given in Chapter 5.

4.3 Availability analysis and improvements for Mixed - Criticality systems

To propose evaluation methods for availability analysis in order to overcome **Problem 2**, we considered a *fault model* for the MC system (**Sub-problem 2.1**). By including failure probabilities for each task and thanks to the static scheduling tables computed by our heuristics (Section 4.2), we are capable of defining methods to compute the availability rate of non-critical tasks numerically. The reincorporation of non-critical tasks is a necessity for the computation of availability rates: considering that the MC system stays in a high criticality mode indefinitely is too constraining for real industrial applications (**Sub-problem 2.2**). We considered a *recovery mechanism* of non-critical tasks once the system makes the transition to a high criticality mode: if all high criticality tasks are able to finish their execution within their low criticality WCET, we can start reincorporating low criticality tasks. Once all low criticality tasks have been reincorporated, we have made the transition back to the low criticality mode. The availability of a task is given by its failure probability, plus the failure probabilities of all tasks executed before it. This allows us to solve the availability formula of Eq. 3.1.

The results we obtained with this recovery mechanism showed the problem we have when considering the discard MC model [36; 8; 37]: since we are sharing a single execution platform, all tasks executing in the system have an influence on the availability of other tasks (**Problem 3**). Even if there are no data-dependencies between two tasks, mode transitions occur in a synchronous way for the whole system: a task with a higher failure

probability executing at the beginning of the scheduling table will influence the availability of a task executing at the end of the scheduling table. A TFE is more likely to occur during the execution of this first task.

To overcome this problem, we incorporated a more detailed *fault propagation model* that limits the interruption of non-critical tasks, and avoids unnecessary timing budget extensions for high criticality tasks (**Sub-problem 3.1**). At the same time, we decided to take into account mechanisms deployed into safety-critical systems that improve the availability of tasks (**Sub-problem 3.2**). For instance, we considered tasks that are weakly-hard real-time (they can tolerate a given number of faults within a number of successive executions) [102] and also design patterns that use redundancy to mask faults. Nonetheless, when these improvements are incorporated into the system, availability needs to be estimated thanks to simulations of the system's execution. To deal with this requirement, we defined translations rules to produce probabilistic automata that are then used in the PRISM model checker [114] to perform system's simulations.

These analyses and improvements led to two publications at the Symposium on Industrial Embedded Systems (SIES) in 2016 and at Design, Automation & Test in Europe Conference & Exhibition (DATE) in 2018. More details about these contributions are given in Chapter 6.

4.4 Implementation of our contributions and evaluation suite: the MC-DAG Framework

Our different contributions led to the development of an open source framework: the MC-DAG Framework[1]. This framework was developed in Java and is cross-platform. It is composed of a scheduling module, a translation module and a random MC-DAG generator. We briefly describe how these different modules work in this section.

Specification of a MC system - to model a MC system with all the elements described in section 4.1, we use XML files. The scheduling tables, computed by our schedulers can also be written to XML files. We chose this markup language because it has been used in similar tools like SDF³ [115], and in some RTOSes.

Scheduling module - this module uses the specification of a MC-DAG system and applies the requested scheduling heuristic using the number of cores that are given. If the heuristic managed to find a solution, the scheduling tables are written into a file. Otherwise an exception is thrown and an error message is returned to the user. The implementation

[1] MC-DAG framework - https://github.com/robertoxmed/MC-DAG

of the scheduling module is multithreaded: we can test the schedulability of various systems at the same time in order to perform benchmarks. These benchmarks are capable of comparing us to the state of the art [80; 81], but they also allow us to analyze the behavior of our heuristics.

PRISM translation rules - this module takes the system and the scheduling tables as an input in order to apply the translation rules to obtain PRISM automata. Two files need to be produced in this case, one for the model and another contains the formulas to evaluate the availability rate of non-critical tasks thanks to the simulations of the system.

Random MC-DAG generator - existing approaches of the literature to schedule MC-DAGs on multi-core architectures have only shown theoretical works. No benchmarking tools were developed which was an important limitation we wanted to address. To test our heuristics and compare our works to the state of the art, the MC-DAG framework includes a random MC-DAG generator. The generation of random MC-DAG has to be unbiased nonetheless. In order to achieve this unbiased generation, we incorporated existing works of the literature regarding the generation of DAGs [116] but also the distribution of timing budgets for tasks for MC systems [5; 55]. Nevertheless, we want to be able to control some parameters of this generation to assess statically our contributions.

4.5 Conclusion

This chapter presented an overview of our contributions that address the problems we presented in Chapter 3. Our first contributions are related to the scheduling of MC-DAGs on multi-core architectures. Since delivering a minimum service guarantee for the less critical components is a necessity in safety-critical systems, we have also developed methods to perform availability analyses.

To solve the scheduling problem of data-dependent tasks using the MC model, we designed a list-based meta-heuristic capable of computing scheduling tables for data-driven MC systems. As opposed to existing approaches of the literature that have taken the same route, we defined improved conditions for the execution of HI-criticality tasks in the LO-criticality mode, in order to keep safe mode transitions. We have also implemented our meta-heuristic following a global approach which improves our schedulability rate compared to existing approaches of the literature. Our scheduling approach has been generalized to support an arbitrary number of criticality levels as well. Details about our scheduling approaches are given on Chapter 5.

The availability analysis and proposed improvements for MC systems are presented in Chapter 6. We explain in detail how we calculate the availability of non-critical tasks

thanks to our fault model. Our results also demonstrate that the discard MC model, where LO-criticality tasks are discarded after a mode transition occurs, delivers a poor availability rate for LO-criticality tasks. For this reason, we propose to include availability enhancements on MC systems. Our availability analysis is extended to support these enhancements in the system.

The framework we developed incorporating all our contributions is presented in Chapter 7. In order to compare our methods to existing approaches of the literature, this framework includes a random MC-DAG generator, capable of generating unbiased graphs.

Finally, we present the experimental validations of our contributions in Chapter 8. We compare the performances of our scheduling heuristics to the existing methods of the literature [80; 81]. This comparison uses metrics like acceptance rates and number of preemptions generated by the scheduling methods we propose. We also present a study on the schedulability of MC systems that integrate more than two criticality levels.

5 Scheduling MC-DAGs on Multi-core Architectures

TABLE OF CONTENTS

Many scheduling algorithms have been proposed for the MC model, nevertheless taking into account data dependencies (or precedence constraints) between tasks has rarely been addressed [35]. In chapter 3, we demonstrated that **MC Scheduling of MC-DAGs (Problem 1)** is a *NP*-hard problem. In this chapter we present the contributions related to solving this problem.

We first present a meta-heuristic capable of scheduling MC systems executing applications in the form of MC-DAGs on a multi-core architecture: MH-MCDAG. The meta-heuristic tries to compute feasible schedules to satisfy **Schedulability in all modes of execution (Sub-problem 1.1)**. Nonetheless, when the low-criticality mode scheduling is computed, we need to take particular care of mode transitions to higher criticality modes since the **Schedulability in case of a mode transition (Sub-problem 1.2)** needs to be ensured. In order to do so, we introduce a property that needs to be respected for high-criticality tasks when we are computing the low-criticality scheduling table. By doing so, we are sure to obtain *MC-correct* schedulers (solving Sub-problem 1.1 and 1.2) for any implementation of the meta-heuristic. Therefore, we propose a generic approach to solve the **MC Scheduling of MC-DAGs (Problem 1)**.

Works in the literature have tackled the problem of scheduling MC-DAGs in mono and multi-core architectures [80; 79; 81; 82; 83]. We have categorized and proposed solutions to **Limits of existing approaches (Sub-problem 1.3)** throughout this chapter.

First of all, **(i)** the *execution of high-criticality tasks is heavily constrained* since these tasks are systematically prioritized even when the system is in low-criticality mode. By doing so, low-criticality tasks' execution could be considerably delayed, potentially causing a deadline miss. Therefore, we propose to *relax the execution of high-criticality tasks in high-criticality mode* in order to obtain better schedulability while still guaranteeing MC-correctness. This relaxation is also motivated by the fact that *communication between low and high-criticality tasks* are more likely to be satisfied when the scheduling of the system is less constrained.

Approaches in [81; 82; 83] advocate for the creation of core clusters when multiple *sporadic* MC-DAGs need to be scheduled: each cluster schedules a single MC-DAG. **(ii)** *Clustering often leads to a poor resource usage* when MC-DAGs are activated in a periodic manner. It is very likely that a MC-DAG will not use all the processing capabilities offered by the cores of their clusters. To overcome this problem, we developed a *global and generic* implementation of our meta-heuristic MH-McDAG. Since the scheduling problem is complex, we want to be able to solve it efficiently (*e.g.* in pseudo-polynomial time) while still ensuring good resource usage of the targeted architecture. Two different real-time schedulers where adapted to fit this implementation. The scheduler builds tables off-line, similarly to Time-Triggered (TT) execution [16].

The final part of this chapter presents a generalization of MH-McDAG to support an arbitrary number of criticality levels: **(iii)** existing *scheduling strategies for MC-DAGs are limited to dual-criticality systems*. New conditions on high-criticality tasks activations need to be introduced in order to have safe mode transitions to higher criticality modes. The generalization of the scheduling approach is motivated by the fact that industrial standards often define more than two criticality levels for their systems.

5.1 Meta-heuristic to schedule MC-DAGs

When designing a scheduling approach for MC-DAGs executing in multi-core architectures, we need to make sure that the scheduling computed respects deadlines, data-dependencies and mode transitions to the higher-criticality mode.

In this section we define a meta-heuristic to schedule MC-DAGs: MH-McDAG. This meta-heuristic is applicable to schedule dual-criticality systems, *i.e.* MC systems with two levels of criticality: HI and LO. In order to achieve this, we define a condition on

high-criticality tasks execution, that when respected, ensures that the system is capable of switching to the high-criticality mode without causing a deadline miss.

5.1.1 Mixed-Criticality correctness for MC-DAGs

To define a scheduling strategy for MC systems executing MC-DAGs, we begin by characterizing properties to be respected in order to satisfy **Schedulability in all modes of execution (Sub-problem 1.1)** and **Schedulability in case of a mode transition (Sub-problem 1.2)**. In [81], Baruah introduced the notion of a *MC-correct* scheduling strategy for MC-DAGs:

Definition 10. *A **MC-correct** schedule is one which guarantees*

1. ***Condition LO-Mode**: If no vertex τ_i, of any MC-DAG in $\mathcal{G}$ executes beyond its $C_i(LO)$ then all the vertices complete execution by the deadlines; and*
2. ***Condition HI-Mode**: If no vertex τ_i, of any MC-DAG in $\mathcal{G}$ executes beyond its $C_i(HI)$ then all the vertices that are designated as being of HI-criticality complete execution by their deadlines.*

Condition LO-Mode in Definition 10 ensures the schedulability in LO mode: as long as all tasks execute within their $C_i(LO)$ (*i.e.* no TFE occurs), a MC-correct scheduling satisfies task deadlines and precedence constraints. **Condition HI-Mode** in Definition 10 states that when HI-criticality vertices need to execute until their $C_i(HI)$ (*i.e.* after a mode transition has occurred), a MC-correct scheduling satisfies deadlines and precedence constraints for HI tasks. **Condition HI-Mode** needs to be ensure at all times for the MC system.

Guaranteeing **Condition LO-Mode** is possible as long as a *correct* scheduler for the MC system is found in the LO-criticality mode.

Definition 11. *A **correct** schedule in a given criticality mode $\chi \in \mathcal{CL}$, is a schedule that respects the deadline and precedence constraints on all task jobs of the MCS, considering their $C_i(\chi)$ as execution time.*

The difficulty in guaranteeing MC-correctness comes from the fact that **Condition HI-Mode** needs to be respected. In fact, having two independent *correct* schedulers for both criticality modes is not sufficient to guarantee MC-correctness: in other words having a correct schedule in HI mode and another in LO mode is not sufficient to have MC-correctness. The HI-criticality mode scheduling needs to be correct allowing the system

to switch to the HI-criticality mode without missing a deadline for HI-criticality tasks. This problem was demonstrated in Section 3.1.2 (Fig. 3.2), the scheduling of tasks in the LO-criticality mode plays a major role in mode transitions. If HI-criticality tasks are activated too late in the LO-criticality mode, then a possible deadline miss can occur. To avoid this problem, we want to know *when does a HI-criticality task need to be executed in the LO-criticality mode, in order to not miss a deadline if a TFE occurs?*

To answer this question, we define **Safe Transition Property**, a sufficient property on HI-criticality tasks execution in the LO-criticality mode. Respecting the property in the LO-criticality mode guarantees that the timing budget allocated to HI-criticality tasks is large enough so they can complete their execution even after a mode transition. This can be done by executing HI-criticality tasks in the LO-criticality mode, before or at the same time they are executed in the HI-criticality mode.

To formalize this concept, we start by defining the function ψ_i^{χ} as follows:

$$\psi_i^{\chi}(t_1,t_2) = \sum_{s=t_1}^{t2} \delta_i^{\chi}(s). \tag{5.1}$$

where

$$\delta_i^{\chi}(s) = \begin{cases} 1 & \text{if } \tau_i \text{ is running at time } s \text{ in mode } \chi, \\ 0 & \text{otherwise} \end{cases}.$$

This function defines the *cumulated execution time* allocated to task τ_i in χ mode from time t_1 to time t_2. In other words it counts how much time was allocated to task τ_i in the χ mode during the timespan $[t_1;t_2]$. Thanks to this function we are now capable of defining the sufficient property to have MC-correct scheduling.

Definition 12. ***Safe Transition Property (Safe Trans. Prop.)***

$$\psi_i^{LO}(r_{i,k},t) < C_i(LO) \implies \psi_i^{LO}(r_{i,k},t) \geq \psi_i^{HI}(r_{i,k},t). \tag{5.2}$$

where $r_{i,k}$ is the release date of the job k of task τ_i. Since we are considering real-time periodic systems, tasks can have multiple activations.

Safe Trans. Prop. states that, while the k-th activation of HI task τ_i has not been fully allocated in LO mode, the budget allocated to this job in LO mode must be greater than the one allocated to it in HI mode. Intuitively this guarantees that whenever a TFE occurs, the final budget allocated to the job of τ_i is at least equal to its WCET in HI mode.

Theorem 1. *To ensure MC-correctness (Definition 10), it is sufficient to obtain a correct schedule in HI-criticality mode and from this, define a correct schedule in LO-criticality mode respecting* ***Safe Trans. Prop.***

Proof. We suppose that we have computed two scheduling tables using Theorem 1 for a MC system $\mathcal{S}$. Let us prove that the scheduling tables obtained respect *MC-correctness*.

Condition LO-Mode is respected since the LO-criticality scheduling table obtained through Theorem 1 is correct (Definition 11). Therefore tasks are able to complete their execution within their $C_i(LO)$ and no deadline is missed.

Condition HI-Mode can be decomposed into two parts: **(i)** the scheduling table for the HI-criticality is correct by construction (*i.e. data-dependencies and deadlines are respected*), but we need to prove that **(ii)** mode transitions to the HI-criticality mode do not provoke deadline misses for HI-criticality tasks. To prove the second point, we need to demonstrate that HI-criticality tasks will have enough processing time to complete their execution without missing a deadline in case of a TFE. Let us assume a TFE occurs at time t, and let us consider the job $j_{i,k}$ of any HI-criticality task τ_i. At time t, $j_{i,k}$ has been executed for $\psi_i^{LO}(r_{i,k},t)$ (see Equation 5.1).

Case 1. If $\psi_i^{LO}(r_{i,k},t) = C_i(LO)$, $j_{i,k}$ has been fully allocated by the scheduler at time t. τ_i was completely executed in LO mode and met its deadline. Indeed, the scheduling in LO mode is correct and ensures that all tasks meet their deadlines if they are executed within their $C_i(LO)$.

Case 2. If $\psi_i^{LO}(r_{i,k},t) < C_i(LO)$, as a TFE occurs, the scheduling strategy triggers the HI mode. Basically, it stops the LO-criticality scheduling to start the HI-criticality at time instant t. The WCET of $j_{i,k}$ is also updated to $C_i(HI)$. At time instant t, job $j_{i,k}$ in LO mode has already been executed for $\psi_i^{LO}(r_{i,k},t)$ and has $C_i(HI) - \psi_i^{HI}(r_{i,k},t)$ of execution time available to complete its execution in HI mode. We want to know if the allocated budget is large enough to respect the deadline $d_{i,k}$ after the mode switch to the HI-criticality mode.

We define $B_{i,k}(t)$ as the budget that would be allocated between the LO and the HI-criticality mode for job $j_{i,k}$. This budget is decomposed in two parts. The LO mode part allocated during time interval $[r_{i,k},t]$ and the HI mode part allocated during time interval $[t,d_{i,k}]$. More formally,

$$B_{i,k}(t) = \psi_i^{LO}(r_{i,k},t) + \psi_i^{HI}(t,d_{i,k})$$

Since $C_i(HI) = \psi_i^{HI}(r_{i,k},t) + \psi_i^{HI}(t,d_{i,k})$, we have:

$$B_{i,k}(t) = \psi_i^{LO}(r_{i,k},t) + C_i(HI) - \psi_i^{HI}(r_{i,k},t)$$

Enforcing **Safe Trans. Prop.** in LO mode, we know that $\psi_i^{LO}(r_{i,k},t) \geq \psi_i^{HI}(r_{i,k},t)$. Therefore:

$$\begin{aligned} B_{i,k}(t) &\geq \psi_i^{HI}(r_{i,k},t) + C_i(HI) - \psi_i^{HI}(r_{i,k},t) \\ &\geq C_i(HI). \end{aligned}$$

We conclude that the budget allocated to job $j_{i,k}$ when a TFE occurs, is large enough to complete its execution within its HI criticality WCET.

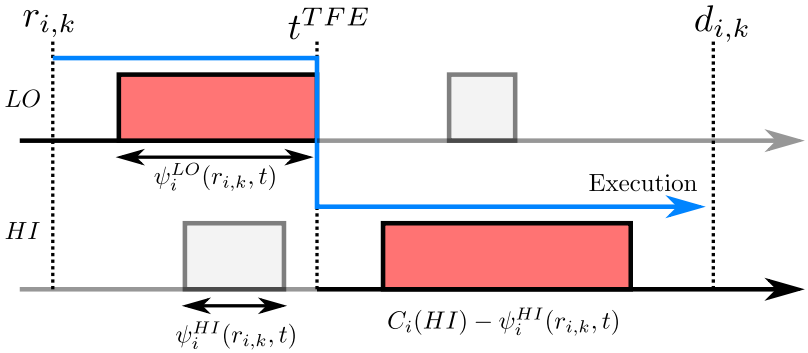


Figure 5.1: Illustration of case 2: $\psi_i^{LO}(r_{i,k},t) < C_i(LO)$.

To better understand the proof, Fig. 5.1 illustrates the behavior of the scheduling strategy respecting **Safe Trans. Prop.** and being schedulable in HI and LO modes. The red rectangles represent the available computation time for a job $j_{i,k}$ when a TFE occurs. It is clear from the figure that this computation time is large enough for the job to complete its execution within its $C_i(HI)$.

□

To design a scheduling approach that is MC-correct and therefore that respects conditions **Condition LO-Mode**, **Condition HI-Mode** and correct execution in both criticality modes, we have introduced the sufficient property: **Safe Trans. Prop.** Building on this condition, we present MH-McDAG, a meta-heuristic capable of scheduling MC systems composed of MC-DAGs.

5.1.2 MH-McDAG, a meta-heuristic to schedule MC-DAGs

In this subsection we define MH-McDAG, a meta-heuristic to schedule MC systems executing multi-periodic MC-DAGs. This meta-heuristic is decomposed in two steps: the scheduling of the system in the HI-criticality mode, and the scheduling of the system in the LO-criticality mode enforcing **Safe Trans. Prop.** MH-McDAG schedules the system in HI-criticality mode using an adaptation of a suitable scheduling algorithm for real-time

tasks with data dependencies. Global-Earliest Deadline First (G-EDF) can be used for instance. If the schedule is correct in HI-criticality mode, we then proceed to schedule the system in LO-criticality mode. The same scheduling algorithm can be used again for the LO-criticality mode, but we enforce **Safe Trans. Prop.** to guarantee **Condition HI-Mode** of MC-correctness (Definition 10). If the resulting schedule is correct in LO-criticality mode, **Condition LO-Mode** of MC-correctness is also satisfied and therefore the schedule is MC-correct.

Definition 13. MH-MCDAG *Meta-heuristic for multi-periodic MC-DAG scheduling*

1. *Schedule tasks in HI-criticality mode and check its correctness.*

2. *Schedule tasks in LO-criticality mode, enforcing **Safe Trans. Prop.** Check correctness of the schedule in LO-criticality mode.*

The main advantage of defining a meta-heuristic is its *genericity*: any implementation of MH-MCDAG will be able to obtain MC-correct schedulers for MC-DAGs. In fact, we simply need to adapt a scheduler to take into account real-time deadlines and data-dependencies. Different aspects can be considered when adapting a scheduler in order to follow MH-MCDAG. If the user is looking to improve the*acceptance rate* for example, solutions based on the *laxity* of a task, like G-LLF, have shown to be efficient [117]. On the other hand, if the user is concerned about the *number of preemptions* entailed by the adapted scheduler, another algorithm like G-EDF would be more suitable than laxity-based algorithms. The only requirement for the adaptation is that **Safe Trans. Prop.** needs to be ensured, this is not suitable for all scheduling algorithms since we need to count the timing budget that is allocated for tasks in both criticality modes.

In the next section we begin to tackle the **Limits of existing approaches (Sub-problem 1.3)**. As a matter of fact, existing approaches that schedule MC-DAGs [79; 80; 81] can be considered as implementations of MH-MCDAG. They implicitly enforce **Safe Trans. Prop.**. In these existing approaches, HI-criticality tasks are systematically prioritized during the scheduling of the two-criticality modes, that way HI-criticality tasks complete their execution in the low-criticality mode before or at least at the same time than in the HI-criticality mode. We demonstrate that by doing so, HI-criticality tasks' execution in the low-criticality mode is too constrained which makes the MC scheduling problem harder than it should be.

5.2 Scheduling HI-criticality tasks

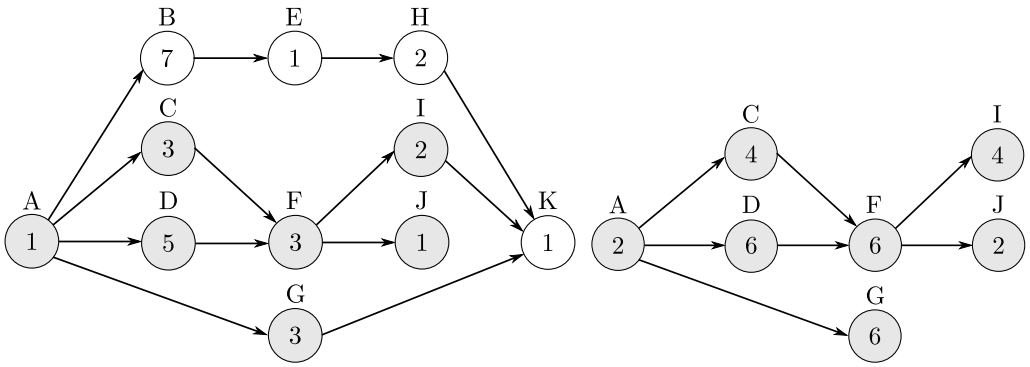
Existing approaches of the literature to schedule MC-DAGs [79; 80; 81] on dual-criticality systems (*i.e.* system having a LO and HI execution mode) are implicit implementations of MH-McDAG. These approaches, based on List Scheduling (LS), compute two separate priority orderings for HI and LO-criticality tasks. **Condition HI-Mode** of MC-correctness (Definition 10) is ensured by *systematically prioritizing the execution of HI-criticality tasks in both criticality modes*. In this section we explain **(i)** why constraining HI-criticality task this way is detrimental for the schedulability of the MC system. Thus, **(ii)** we propose to compute the HI-criticality scheduling by maximizing their completion time. This relaxation of HI-criticality tasks' execution also allows to **(iii)** incorporate low-to-high communication more easily than in the existing approaches.

5.2.1 Limits of existing approaches: as soon as possible scheduling for HI-criticality tasks

The adoption of LS heuristics to schedule a MC-DAG into a multi-core processor was first proposed by Baruah in [80]. This contribution proposes a method to schedule a MC-DAG using the MC discard model with two levels of criticality: HI and LO. The motivation behind the contribution is to keep the performances of LS scheduling (*i.e.* polynomial complexity to find feasible solutions) but include constraints on HI-criticality tasks in order to have safe mode transitions.

In order to obtain the scheduling for a MC-DAG, [80] establishes a priority ordering for HI-criticality tasks: this priority ordering is used to sort a list of *ready* tasks (*i.e.* tasks that have met their precedence constraints). The order gives more or less priority to tasks when the allocation decision is made, *i.e.* when a task is scheduled in a core for a given timeslot. In [80] proved that *the heuristic is valid for any priority ordering obtained by LS algorithms*: in other words, the approach is generic as long as a LS algorithm is used to obtain the priority ordering of vertices for the MC-DAG. A HI-criticality scheduling table (S^{HI}) is then computed by picking the top elements of the list. The list is updated progressively when tasks finish their execution or when tasks have met their precedence constraints and become ready. In the HI-criticality mode, tasks cannot be preempted.

If the system is schedulable in the HI-criticality mode, then the LO-criticality scheduling table is computed by considering tasks with their $C_i(LO)$. Another priority ordering is calculated for all tasks but LO and HI-criticality priority orderings are independent. While computing the scheduling table, HI-criticality tasks always have a greater priority than LO-



(a) MC-DAG in LO mode (b) MC-DAG in HI mdoe

Figure 5.2: Example of MC-DAG

criticality tasks. In order to have safe mode transitions to the higher criticality mode, HI-criticality tasks can also preempt LO-criticality tasks. In other words, HI-criticality tasks are executed As Soon As Possible (ASAP), while LO-criticality tasks can be preempted. Proofs of MC-correctness for the heuristic are included in the publications supporting the contribution [80; 81].

Thanks to an example of a MC-DAG (illustrated in Fig. 5.2) we explain how the algorithm of [80] computes the scheduling tables for the MC system. We aim to demonstrate how HI-criticality tasks' execution affects the scheduling in the LO-criticality mode, more precisely how LO-criticality tasks can be affected by the ASAP execution of HI-criticality tasks. While the algorithm of [80] might declare a system as non-schedulable, we show that a MC-correct schedule can be obtained if HI-criticality tasks are less constrained.

The MC-DAG in LO mode is represented in Fig. 5.2a, and the HI mode is shown in Fig. 5.2b. Gray vertices represent HI-criticality tasks and white vertices LO-criticality tasks. The number labels on the vertices represent the WCET in LO mode ($C_i(LO)$) and in HI mode ($C_i(HI)$). The WCET is represented in Time Units (TUs). We consider that the deadline and period of this MC-DAG are equal to 18 TUs. We define the *utilization* for a MC-DAG on criticality level χ_ℓ as:

$$U_{\chi_\ell}(\mathcal{G}) = \sum_{\tau_i \in V} \frac{C_i(\chi_\ell)}{T_j}. \tag{5.3}$$

The utilization rate can be generalized for the whole system $\mathcal{S}$:

$$U_{\chi_\ell}(\mathcal{S}) = \sum_{G_j \in \mathcal{G}} U_{\chi_\ell}(G_j). \tag{5.4}$$

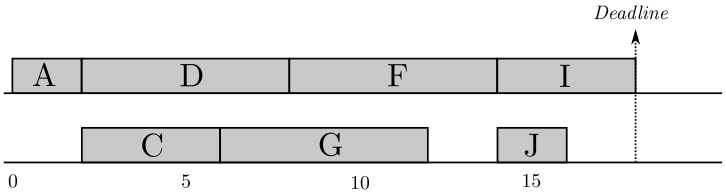
We define $U_{max}(\mathcal{S}) = \max\{U_{\chi_\ell}(\mathcal{S}) \mid \forall \chi_\ell \in \mathcal{CL}\}$. This value allows us to deduce the lower bound for the number of cores required to schedule the system $\mathcal{S}$: $m_{min} = \lceil U_{max}(\mathcal{S}) \rceil$. If $U_{max}(\mathcal{S}) > m$, (where m is the number of cores available in the architecture) we know for a fact that the processor Π does not have enough processors to schedule the system. In our example (Fig. 5.2), the utilization in LO mode is obtained with Eq. 5.3: $U(LO) = \frac{1+7+3+5+1+3+3+2+2+1+1}{18} \approx 1.61$ and $U(HI) \approx 1.67$. Therefore we need at least $m_{min} = \lceil \max(1.61, 1.67) \rceil = 2$ cores to schedule the system.

As we mentioned before, the contribution in [80] demonstrated that any priority ordering can be used for the LS heuristic to compute the scheduling tables for the MC-DAG. With the execution model that we have, Highest Level First with Estimated Times (HLFET) has shown to be the most efficient in terms of complexity and makespan minimization. Therefore, HLFET has better chances to find a feasible schedule with the deadline that was given to the DAG [72]. With this priority ordering and thanks to the example of Fig. 5.2, we demonstrate how LO-criticality tasks' execution is affected by the constant preemption caused by HI-criticality tasks.

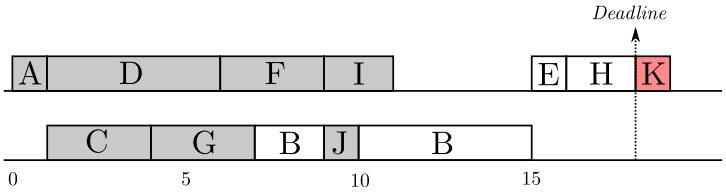
HLFET bases its priority ordering on a *level* coefficient, given by the longest path to an exit vertex, considering timing budgets given to tasks. This path is known in graph theory as the **critical path** from a vertex to an sink vertex. The critical path of a MC-DAG is the largest critical path from an entry vertex to a sink vertex. For example in Fig. 5.2a, the HLFET level/critical path of vertices K and J in the LO-criticality mode is 10, because they are exit vertices with no successors. The HLFET level of A in the LO mode is 110, and so on. When we apply the heuristic proposed in [80] using HLFET as the priority ordering, we obtain the scheduling tables presented in Fig. 5.3 considering a dual-core architecture. The priority orderings calculated thanks to HLFET are the following. For the HI-criticality mode: : $\langle A \prec_{HI} D \prec_{HI} C \prec_{HI} F \prec_{HI} G \prec_{HI} I \prec_{HI} J \rangle$. For the LO-criticality mode: $\langle A \prec_{LO} C \prec_{LO} D \prec_{LO} G \prec_{LO} F \prec_{LO} I \prec_{LO} J \rangle$ for HI tasks and $\langle B \prec_{LO} E \prec_{LO} H \prec_{LO} K \rangle$ for LO tasks.

As we can see the system is schedulable in the HI-criticality mode (Fig. 5.3a): the deadline is respected as well as data-dependencies between tasks. Nonetheless, the S^{LO} table in Fig. 5.3b shows that the deadline given to the MC-DAG is not respected for task K. The missed deadline is due to the fact that task B is being constantly preempted by HI-criticality tasks. Task B becomes ready right after the execution of task A, *i.e.* at time slot 10, but it only starts its execution at time slot 70. HI-criticality tasks C, F, G, I and J are being allocated before task B because they are HI-criticality tasks.

Intuitively, we can see that some HI-criticality tasks like I and J do not need to be scheduled as soon as they become ready to respect their deadlines and have safe mode tran-



(a) S^{HI} scheduling table



(b) S^{LO} scheduling table

Figure 5.3: Scheduling tables for the MC-DAG of Fig. 5.2

sitions. We can also conclude that LO-criticality tasks might need to be scheduled before some HI-criticality tasks in order to respect the deadline. If a LO-criticality task response time is too large, then its successors might be activated too late. While the approach [80] advocates for the systematic execution of HI-criticality tasks in the LO-criticality mode, we demonstrate in the next subsection that this constraint can be relaxed. The relaxation of HI-criticality tasks' execution tends to improve LO-criticality tasks response times and results in a better acceptance rate for MC systems. This latest affirmation will be demonstrated in Chapter 8, where we measure acceptance rates for our scheduling method and the scheduling method of [80].

5.2.2 Relaxing HI tasks execution: As Late As Possible scheduling in HI-criticality mode

The scheduling approach proposed in [80] constrains the activation and execution of HI-criticality tasks As Soon As Possible (ASAP) in the LO-criticality mode, in order to ensure MC-correctness. In Section 5.1.1 we demonstrated that, as long as the **Safe Trans. Prop.** is respected in the LO-criticality mode, the scheduling computed by the scheduling strategy will be MC-correct. The *key* difference with the existing approach [80], is that HI-criticality tasks in our implementations, are not systematically prioritized during the computation of the LO-criticality scheduling. The *promotion* of HI-criticality tasks only takes place if **Safe Trans. Prop.** would not be respected during the allocation.

To ease the computation of the LO scheduling table, we want to ease the enforcement **Safe Trans. Prop.** To do so, $\psi_i^{HI}(r_{i,k},t)$ should be kept minimal as long as $\psi_i^{LO}(r_{i,k},t) < C_i(LO)$. In other words, **HI-criticality tasks in HI mode should be scheduled As Late**

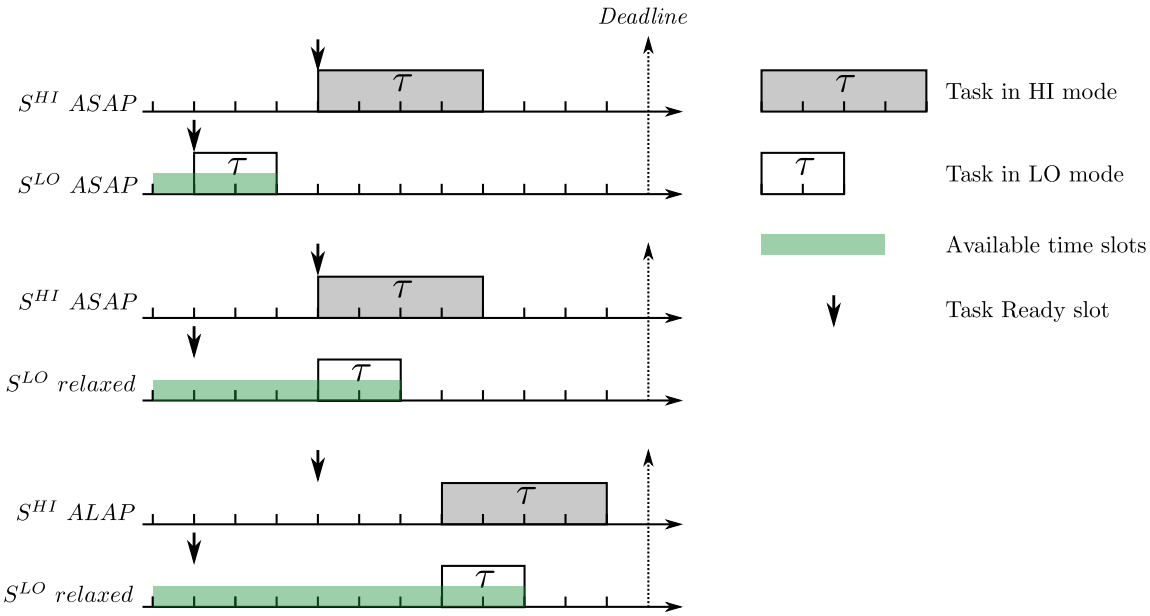


Figure 5.4: Usable time slots for a HI-criticality task: ASAP vs. ALAP scheduling in HI and LO-criticality mode

As Possible (ALAP) and not as soon as they become ready. In Fig. 5.4 we illustrate how a HI-criticality task τ is constrained in three different cases: (i) when the HI-criticality task is scheduled ASAP in HI and LO-criticality modes [80], (ii) when the HI-criticality tasks is scheduled ASAP in the HI-criticality mode but only **Safe Trans. Prop.** is enforced; and (iii) when the HI-criticality tasks is scheduled ALAP in the HI-criticality mode and **Safe Trans. Prop.** is enforced.

The down arrow in Fig. 5.4 represents the time slot at which τ becomes ready (*i.e.* its data dependencies have been met). This time slot is different in the HI and LO-criticality mode since tasks have different timing budgets in both criticality mode and data dependencies can be satisfied earlier in the LO-criticality mode (we have $C_i(LO) \leq C_i(HI)$). We highlighted in green the time slots that task τ can use to execute. The first two timetables represent the scheduling that would be performed by the approach in [80]: HI-criticality tasks are scheduled ASAP in both criticality modes, leaving very few usable slots for task τ. The two middle timetables present the execution of τ that respects **Safe Trans. Prop.** with HI-criticality tasks scheduled ASAP in the HI-criticality mode. For the LO-criticality mode, we lift the ASAP constraint on HI-criticality tasks and show where task τ needs to be scheduled in order to respect **Safe Trans. Prop.** Finally, the last two timetables demonstrate the interest of the approach we chose: if HI-criticality tasks are scheduled ALAP in the HI-criticality mode, when the LO-criticality scheduling table respecting **Safe Trans. Prop.** is computed, task τ has a larger amount of usable time slots for its execution.

The scheduling approaches we propose throughout this chapter are based on the third timetables: we execute HI-criticality tasks ALAP in order to obtain more usable time slots for HI-criticality tasks to execute.

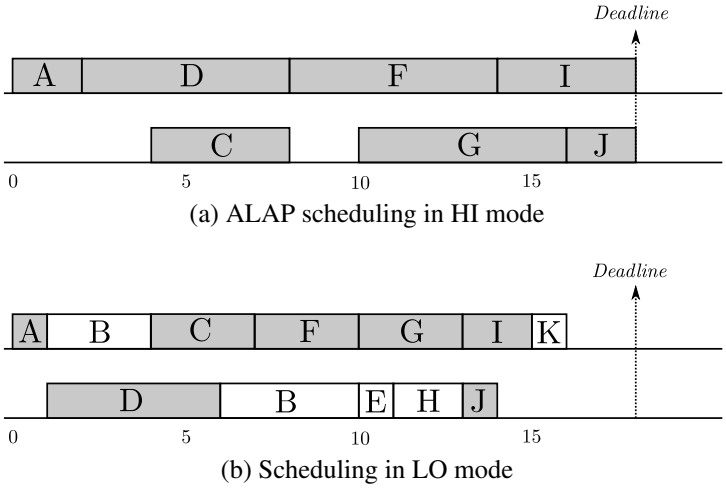
In [118], we proposed a scheduling approach for single MC-DAGs based on LS as well. The difference with [80] is the relaxation of HI-criticality tasks in the LO-criticality mode. The algorithm has the following differences:

- HI-criticality tasks are scheduled ALAP in the HI-criticality mode.
- We use a single priority ordering, $\prec$, for all tasks independently of their criticality level in the LO-criticality mode. **Safe Trans. Prop.** is enforced in the LO-criticality mode, *i.e.* HI-criticality tasks can preempt LO-criticality tasks to respect the property for every time slot.

When we apply the HLFET priority ordering to our scheduling approach, we obtain the following orders, in the HI-criticality mode: $\langle A \prec D \prec C \prec F \prec G \prec I \prec J \rangle$ and in the LO-criticality mode: $\langle A \prec B \prec D \prec C \prec F \prec E \prec G \prec H \prec I \prec J \prec K \rangle$. The HI-criticality scheduling table is presented in Fig. 5.5a. As we mentioned, tasks are scheduled ALAP but the scheduling needs to remain *correct*, *i.e.* data-dependencies and deadlines need to be respected. The LO-criticality scheduling table is presented in Fig. 5.5b. A major difference when comparing our HI-criticality scheduling table to the one in Fig. 5.3a is that task B starts its execution at time slot 10 (as opposed to 7). This allows LO-criticality tasks to not miss the deadline in our case. We can also notice that task B is preempted at time slot 4, this is due to the fact that **Safe Trans. Prop.** needs to be respected in order to have MC-correct scheduling tables. In Chapter 8 we present a comparison in terms of acceptance rate for our scheduling approach and the one proposed in [80]. The experiments confirm the statistical gain of our approach due to the relaxation of HI-criticality tasks execution in the LO-criticality mode.

5.2.3 Considering low-to-high criticality communications

Another motivation to have an ALAP scheduling for HI-criticality tasks in the HI-criticality mode is related to the LO-to-HI communications that we consider in our execution model (Chapter 4 Section 4.1). The execution model adopted in [79; 80; 81], restricts the communication LO-to-HI tasks. This could be in fact be a problem: LO-criticality components do not go through the same certification process than HI-criticality components do, therefore a failure on their execution is more likely to happen. If a HI-criticality task needs the input of the LO-criticality tasks, then the HI-criticality task risks to not be capable of executing.



(a) ALAP scheduling in HI mode

(b) Scheduling in LO mode

Figure 5.5: Improved scheduling of MC-DAG

Nonetheless, in safety-critical systems HI-criticality components could be in charge of monitoring LO-criticality tasks: their outputs are used by the HI-criticality components. For example a HI-criticality task can be in charge of controlling values computed by a LO-criticality tasks in order to detect if these values are correct, *e.g* values are between bounds or do not have drastic changes. In this subsection we answer the following problem: *what are the implications of the LO-to-HI communication in terms of scheduling?*

Characterization of the LO-to-HI communication

For LO-to-HI communications, we need to respect the precedence constraint: the LO-criticality task needs to be scheduled before the HI-criticality successor. Nonetheless, the HI-criticality task *is capable of fulfilling its execution even if the LO-criticality task did not produce any data* (caused by the a TFE and the mode transition to the HI-criticality mode for example). The semantic for this type of communication is defined as follows:

Definition 14. *A **soft data-dependency** is used when the transmitting task has a lower criticality level than the receiving task. While the precedence constraint needs to be respected in all criticality modes where both tasks are executed on, the receiving tasks is capable of completing its execution even if the transmitting task was not able to produce its outputs (due to a TFE for example).*

In Fig. 5.6 we represent a MC-DAG with a LO-to-HI communication. HI-criticality tasks are represented in gray (τ_1 and τ_3). These tasks are annotated with their WCET in LO and HI-criticality mode. τ_2 is a LO-criticality task represented in white, annotated with its WCET in LO mode. The communication between τ_2 and τ_3 is considered to be

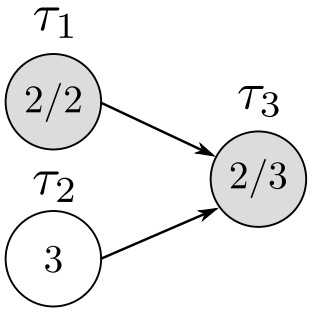


Figure 5.6: A MC-DAG with LO-to-HI communications

a *soft data-dependency*. The deadline/period for the MC-DAG is equal to 5 TUs. Thanks to this example we explain how the ASAP strategy is often not capable of scheduling the system whereas a MC-correct schedule for this system exists.

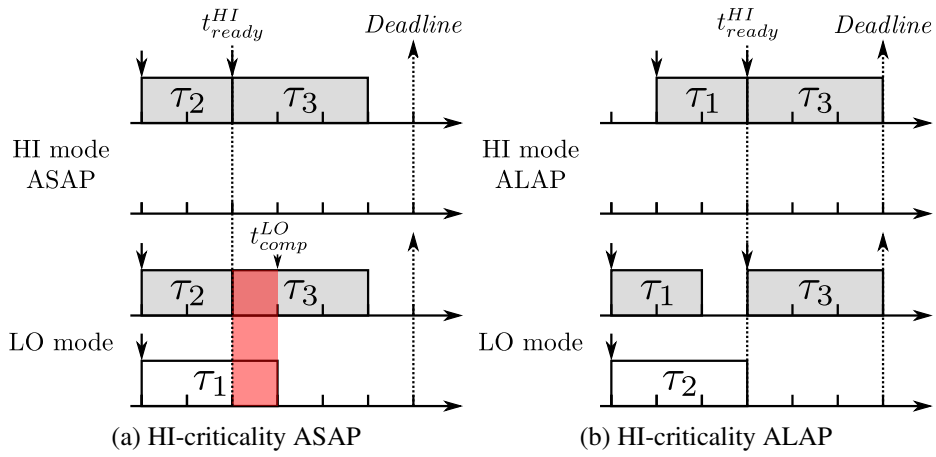
Constraints on the scheduling

When it comes to scheduling MC-DAGs containing LO-to-HI communications, we can characterize a necessary condition for LO-criticality tasks that emit data to HI-criticality tasks. This condition applies to off-line scheduling methods. The following condition needs to be respected in the LO-criticality mode:

$$t^{LO}_{comp}(j_{i,k}) \leq \min\left(d_{i,k}, \min_{\tau_j \in succ(\tau_i)}\left(t^{HI}_{ready}(\tau_j)\right)\right). \tag{5.5}$$

The condition states that the completion time of a job $j_{i,k}$ in the LO-criticality mode, noted $t^{LO}_{comp}(j_{i,k})$, needs to be inferior or equal to: the minimum between deadline of the job $d_{i,k}$ and; the minimum of the time at which HI-criticality task become ready in the HI-criticality mode. Respecting this conditions ensures that data produced by the LO-criticality tasks will be available when the HI-criticality task is executed. Also it guarantees that the HI-criticality task can execute in the HI-criticality mode without causing deadline misses.

We illustrate this necessary condition in Fig. 5.7 by scheduling the system illustrated in Fig. 5.6 on a dual-core architecture. The scheduling tables obtained by using the ASAP scheduling of [80] are illustrated in Fig. 5.7a. As we can see, the system is schedulable in the HI-criticality mode: deadlines are respected, as well as precedence constraints. Nonetheless, in the LO-criticality mode there is an incoherence in the scheduling tables: at time slot 3, τ_2 is still being scheduled while τ_3 starts its execution because it needs to respect **Safe Trans. Prop.**. *The precedence constraint between τ_2 and τ_3 is not respected in this case*. In Fig. 5.7b, we illustrate the scheduling tables using an ALAP strategy for



(a) HI-criticality ASAP (b) HI-criticality ALAP

Figure 5.7: ASAP vs. ALAP with LO-to-HI communications

HI-criticality tasks. Again the system is schedulable in the HI-criticality mode. This time in the LO-criticality mode, there are no incoherence because τ_3 is capable of starting its execution at a later time slot.

In this section we have presented the limits regarding the scheduling of HI-criticality tasks on existing approaches of the literature. Because HI-criticality tasks are scheduled ASAP in all criticality modes, systems are deemed as non-schedulable when feasible and MC-correct solutions exist. *Relaxing* the execution of HI-criticality tasks in the HI-criticality mode by using an ALAP strategy allows us to gain in schedulability and also allows systems to have LO-to-HI communication. The next section presents our contribution related to the scheduling of multiple MC-DAGs: existing approaches have reduced the multiple MC-DAG scheduling problem to the scheduling of a single MC-DAG on a cluster of cores. This reduction often leads to poor resource usage.

5.3 Global implementations to schedule multiple MC-DAGs

In this section we present our global implementation of MH-McDAG tackling the problem of *multiple periodic MC-DAG scheduling* for multi-core architecture. Existing contributions [81; 82; 83] scheduling multiple MC-DAGs have chosen to follow the *federated* approach. The idea behind federated approaches is to create clusters of cores for DAGs that have a high utilization, *i.e.* $U_{max}(G_j) > 1$. A MC-DAG then has *exclusive* access to $\lceil U_{max}(G_j) \rceil$ cores. By doing so, the problem of scheduling multiple MC-DAGs into a multi-core architecture is transformed into single MC-DAG scheduling (in a set of cores). For MC-DAGs with $U_{max}(G_j) \leq 1$, they are transformed into sequential tasks and are ex-

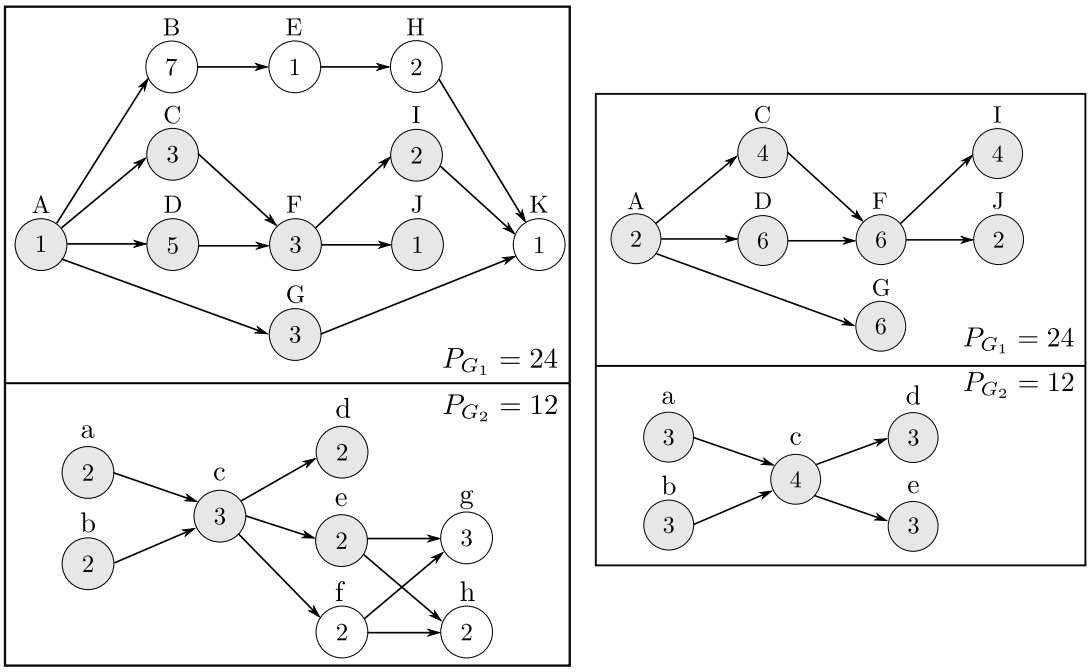
ecuted with a MC scheduling policy [35] on the remaining cores after the clustering has taken place.

While federated approaches have the advantage of being quite simple and being capable of supporting sporadic activations of MC-DAGs, they lead to poor resource usage. For example in a system composed of two MC-DAGs $\mathcal{G}_1$ and $\mathcal{G}_2$ with the following utilization rates $U_{max}(G_1) = 3.1$ and $U_{max}(G_2) = 1.2$, the federated approach of [81] needs at least 6 cores to schedule the system, $\lceil U_{max}(G_1) \rceil + \lceil U_{max}(G_2) \rceil = 4 + 2 = 6$. Nonetheless, since the utilization of the system is closer to 5: $U_{max}(\mathcal{S}) = 3.1 + 1.2 = 4.5$, we would like to define a scheduling approach capable of computing tables with the least amount of cores required.

To overcome poor resource usage, we have designed a global version of MH-McDAG for multi-periodic MC-DAGs. Multi-periodicity of software components is often used in the design of reactive safety-critical systems and better scheduling methods can be defined. The implementation of MH-McDAG, called G-ALAP, is based on the principle that HI-criticality tasks are scheduled ALAP in the HI-criticality mode and that *all* cores can be used during the allocation of the vertices. G-ALAP computes static scheduling tables off-line that are MC-correct. This implementation has been instantiated three times by using different real-time scheduling algorithm: *because the implementation is generic we can easily choose one of the three adaptations*. We instantiate our algorithm by adapting real-time scheduling algorithms instead of using LS heuristics like HLFET. *HLFET is not applicable for the case when multiple MC-DAGs are considered* since it only uses execution times to calculate the priority ordering of tasks. Conversely, we need to respect periods and deadlines of tasks as well.

By choosing global scheduling policies we are capable of taking into account deadlines and also allocate tasks on *all* the cores that are available in the architecture. The first instance of G-ALAP is based on the Global Least-Laxity First (G-LLF) algorithm known to give good performances in terms of acceptance rates. The second instance is based on Global Earliest Deadline First (G-EDF), it entails less preemptions and migrations which are relevant aspects when designing schedulers for safety-critical systems. The third instance of our scheduling algorithm combines both G-EDF and G-LLF. We limit the number of preemptions in the HI-criticality mode by using G-EDF and improve schedulability in the LO-criticality mode by using G-LLF. Other adaptations of MH-McDAG could also be considered and we show that few modifications are required in order to obtain them.

In Fig. 5.8 we present a MC system composed of two MC-DAGs. Fig. 5.8a represents the system in the LO-criticality mode, numbers correspond to the $C_i(LO)$ of tasks. Fig. 5.8b represents the system in HI-criticality mode. This system includes the previous



(a) LO-criticality mode

(b) HI-criticality mode

Figure 5.8: MC system with two MC-DAGs

MC-DAG presented in Fig. 5.2, noted G_1, with a new period/deadline of 24 TUs. We consider MC-DAGs have implicit deadlines deduced by their periods. The second MC-DAG, G_2, has a period/deadline of 12 TUs. Since the system is multi-periodic due to the two MC-DAGs, scheduling tables need to be computed for the *hyper-period* of the system. The hyper-period H is equal to the least common multiplier of all the periods of the MC-DAGs. In this example $lcm(24, 12) = 24$. Therefore, MC-DAG G_2 will have two executions, while MC-DAG G_1 will have only one execution during the hyper-period.

5.3.1 Global as late as possible Least-Laxity First - G-ALAP-LLF

In this subsection we present the global implementation of MH-MCDAG, called G-ALAP. We detail how the algorithm works considering the G-LLF instance of the algorithm. G-ALAP is decomposed in two steps: (i) compute the HI-criticality scheduling table; and (ii) compute the LO-criticality scheduling table enforcing **Safe Trans. Prop.**

HI-criticality scheduling table: Algorithm 1 presents the computation of the HI-criticality scheduling table (S^{HI}). Like we mentioned in the previous section, in order to ease the computation of the LO-criticality scheduling table for HI-criticality tasks, we are going to allocate HI-criticality tasks ALAP in the HI-criticality mode. An easy way to obtain such behavior is to produce **dual task graphs** of the MC-DAGs and schedule this new

set using an ASAP strategy. A dual task graph is obtained by **inversing the precedence constraints**. Obtain a dual for each graph of the system can also be performed, we use the following notation: $\mathcal{S}^\star$ for the dual of the system and $G^\star$ for the dual of a single MC-DAG. Once a *correct* scheduling is found for the dual graphs, the final step of the algorithm inverses the table: we horizontally "flip" the scheduling table, since data-dependencies were respected for the dual system, once the horizontal flip is performed data-dependencies will be respected for the normal system. The transformation of the system to its dual is done in line 2 of Alg. 1. The inversion of the scheduling table is done in line 27.

The rest of the algorithm is straightforward. We begin by initializing the remaining time of all HI-criticality tasks and by inserting source vertices into the ready list (l. 5-8). R_i^{HI} is the variable that stores the remaining time that needs to be allocated for a task job. The next step consists in building the scheduling table slot by slot until the hyper-period. Before the scheduling decisions for each core are made, we sort the ready list. The SORTHI (l. 9) function is the core of our scheduling strategy: jobs that are in the ready list are sorted according to a specific priority ordering. In the G-ALAP-LLF implementation, SORTHI order is determined by the laxity of each task, therefore SORTHI needs to compute laxities of ready tasks to then sort them in ascending order.

To obtain a scheduler based on G-LLF we need to define a laxity formula in the context of tasks with precedence constraints. We consider **critical path** (CP_i^χ) of a vertex τ_i in mode χ. The laxity of the k-th job of task τ_i (noted $j_{i,k}$), at time slot t, in the criticality mode χ is defined as follows:

$$L_{i,k}^\chi(t) = d_{i,k} - t - (CP_i^\chi + R_{i,k}^\chi). \tag{5.6}$$

$d_{i,k}$ is the deadline of job $j_{i,k}$. $R_{i,k}^\chi$ is the remaining execution time of the job $j_{i,k}$, *i.e.* $R_{i,k}^\chi = C_i(\chi) - \psi_i^\chi(r_{i,k}, t-1)$. Like for the normal G-LLF, a lower laxity leads to a higher priority of the task allocation.

Once laxities are updated and the list is sorted, the function VERIFYCONSTRAINTS is called (l. 10). This utility function needs to be defined differently depending on the scheduling policy used to implement MH-MCDAG. In the case of G-ALAP-LLF, the following constraints have to always be respected:

- **Number of tasks with zero-laxities**: the number of tasks with zero-laxities needs to be inferior or equal to the number of cores available.
- **Non-negative laxities**: if a task has a negative laxity it means that it has accumulated some delay. Due to precedence constraints this cannot be tolerated (delays could be propagated to successors of tasks, leading to a deadline miss).

Algorithm 1 Computation of the HI scheduling table with G-ALAP

function CALCSHI($\mathcal{S}$: MC system to schedule)
  Transform system $\mathcal{S}$ to $\mathcal{S}^\star$
  $Ready \leftarrow \emptyset$ ▷ List of ready tasks

  for all HI tasks τ_i **do**
    $R_i^{HI} \leftarrow C_i(HI)$ ▷ Remaining execution time
    $Ready \leftarrow Ready \cup \{\tau_i \mid pred(\tau_i) = \emptyset\}$ ▷ Add source vertices
  end for

  for all timeslots $t < H$ **do**
    SORTHI($Ready, t$) ▷ Updates ready list order if necessary
    if VERIFYCONSTRAINTS($Ready, t$) $= \perp$ **then**
      return NOTSCHEDULABLE
    end if
    for all cores $c \in \Pi$ **do**
      $\tau_i \leftarrow$ head of $Ready$ not being allocated
      $S^{HI}[t][c] \leftarrow \tau_i$ ▷ Allocate task τ_i
      $R_i^{HI} \leftarrow R_i^{HI} - 1$
    end for
    $Ready \leftarrow Ready \cup \{succ(\tau_i) \mid \forall \tau_j \in pred(succ(\tau_i)), R_i^{HI} = 0\}$
    $Ready \leftarrow Ready \setminus \{\tau_i \in Ready \mid R_i^{HI} = 0\}$

    for all $G_j \in \mathcal{G}$ **do**
      if $t+1 \mod T_j = 0$ **then** ▷ Reactivation of G_j
        $\forall \tau_i \in G_j,\ R_i^{HI} \leftarrow C_i(HI)$
        $Ready \leftarrow Ready \cup \{\tau_i \in G_j \mid pred(\tau_i) = \emptyset\}$ ▷ Add source vertices
      end if
    end for

  end for
  Reverse the scheduling table S^{HI}
  return S^{HI} scheduling table
end function

- **Number of slots left in the scheduling table**: the number of available time slots is equal to the difference between the hyper-period and the current time slot t multiplied by the number of cores: $(H-t)\times m$. If the total remaining times of all ready tasks is superior than this value, then there are not enough time slots left in the scheduling table to allocate all tasks.

If one of these constraints is not respected, VERIFYCONSTRAINTS returns a false boolean and we stop the scheduling heuristic, declaring the system as NOTSCHEDULABLE.

The next step in the algorithm is to allocate tasks to cores (l. 15-19). We just need to grab up to m (m being the number of available cores of Π) tasks that are in *Ready* and allocate them to the available cores. The remaining time for each of the allocated task is updated as well (l. 18).

Once tasks are allocated to the available cores, the algorithm can perform two types of updates on the ready list. The first one consists in activating new jobs (l. 18), if all the predecessors of a task have been scheduled then the task's job becomes ready and is added to the list. The second update removes tasks that have been fully allocated, *i.e.* their remaining time reached zero (l. 19).

After updating the ready list, we verify if any MC-DAG of the system has a new activation (l. 22-27). Since the scheduling table is computed during the hyper-period, MC-DAGs can have multiple activations during the computation of the table. If the period of a MC-DAG is reached, then we reinitialize the remaining time of tasks (l. 24) and reincorporate jobs of the source vertices of the MC-DAG (l. 25). Finally, if the algorithm was capable of going through all timeslots until the hyper-period and VERIFYCONSTRAINTS was never false, then we reverse the scheduling table calculated for $\mathcal{S}^\star$ (l. 27) and return it (l. 30).

Application of Alg. 1 with G-ALAP-LLF in an example: Let us demonstrate the application of the algorithm with the MC system presented in Fig. 5.8. We have the following utilization rates for the MC-DAGs: for G_1 $U_{LO}(G_1)\approx 1.21$ and $U_{HI}(G_1)=1.25$, for G_2 $U_{LO}(G_2)=1.5$ and $U_{HI}(G_2)\approx 1.33$. The maximum utilization rate for the system is therefore $U_{max}(\mathcal{S})\approx 2.71$ and the lower bound of of cores required is $m_{min}=\lceil 2.71\rceil=3$. We apply the Alg. 1 on system $\mathcal{S}$ of Fig. 5.8 with a tri-core architecture.

The transformation of the system to its dual is presented in Fig. 5.9. Like we mentioned, the transformation consists of inversing data dependencies: source vertices become exit vertices and vice versa. With the dual system $\mathcal{S}^\star$, we initialize remaining times for all the HI-criticality task jobs and add the first vertices to the ready list, task jobs $j_{I,0}, j_{J,0}, j_{d,0}$ and $j_{e,0}$ in this case.

The main loop is then reached, and we start the allocation process core by core. The SORTHI function starts by calculating laxities of the ready tasks for time slot 0:

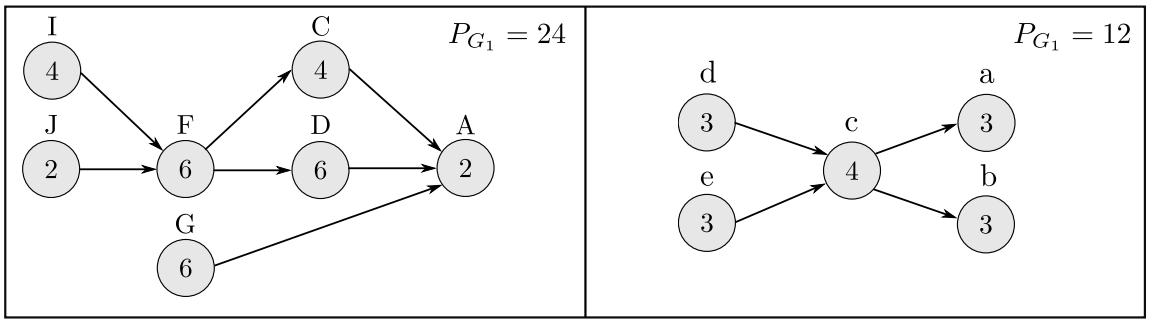
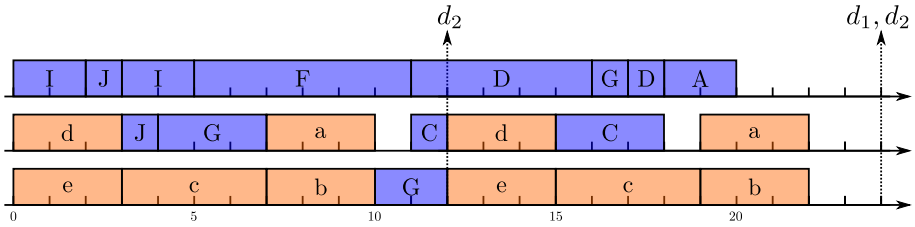
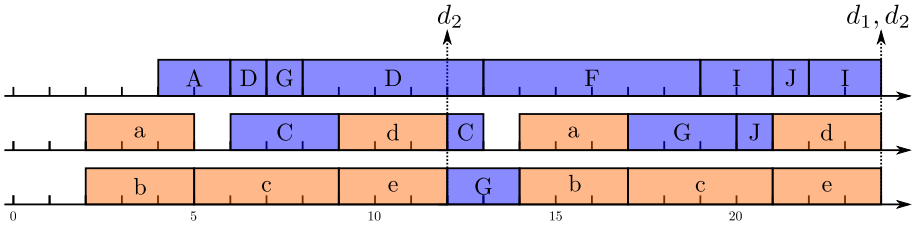


Figure 5.9: Transformation of the system $\mathcal{S}$ to its dual $\mathcal{S}^\star$



(a) HI-criticality scheduling for $\mathcal{S}^\star$



(b) HI-criticality scheduling with ALAP execution

Figure 5.10: HI-criticality scheduling tables for the system of Fig. 5.8

$L^{HI}_{d,1}(0) = 2$, $L^{HI}_{e,1}(0) = 2$, $L^{HI}_{I,1}(0) = 6$ and $L^{HI}_{J,1}(0) = 8$. None of the constraints of VERIFY-CONSTRAINTS is violated so the function does not return false. Since we only have four ready tasks and three available cores, only tasks d, e and I are allocated at this time slot. No updates are performed to the ready list because none of the task finished its execution. Also we have not reached the period of another MC-DAG. Moving to the next slot $t = 1$, we have the following laxities: $L^{HI}_{d,1}(1) = 2$, $L^{HI}_{e,1}(1) = 2$, $L^{HI}_{I,1}(1) = 6$ and $L^{HI}_{J,1}(1) = 7$. The same allocation as the previous slot takes place. At time slot $t = 2$ however, tasks I and J will have the same laxity, the algorithm has an arbitrary order when tasks have the same laxities so in this case, tasks d, e and J are allocated. Also tasks d and e finish their execution thus the ready list is updated: tasks c has its dependencies met therefore it is added to the ready list, and tasks d and e are removed from the ready list. These steps are reproduced until time slot 12: G_2 is then reactivated because we have reached its period.

The final scheduling tables produced by Alg. 1 for the system of Fig. 5.8 are presented in Fig. 5.10. Blue boxes represent the execution MC-DAG G_1 and its tasks. Orange boxes represent the execution of MC-DAG G_2. For this example, the system is in fact schedulable, constraints in VERIFYCONSTRAINTS were never violated during the computation of the scheduling table. Fig. 5.10a presents the scheduling tables of the dual system created by the inversion of dependencies. The final table for system $\mathcal{S}$ is presented in Fig. 5.10b.

LO-criticality scheduling table: Algorithm 2 presents the computation of the LO-criticality scheduling table (S^{LO}). This table is computed almost the same way as the HI-criticality table (Alg. 1). However, for this criticality mode we do not perform a system transformation. The only requirement in order to obtain a MC-correct scheduling is to respect **Safe Trans. Prop.** and obtain a *correct* scheduler in LO-criticality mode.

Like for Alg. 1, we start by initializing remaining times and adding source vertices to the ready list (l. 4-7). Then the main loop allocates tasks to cores slot by slot until it reaches the hyper-period. The sorting function SORTLO, is called for each slot (l. 8): laxities are calculated and the ready list is sorted. However, the main difference between SORTHI and this function, is the fact that **Safe Trans. Prop.** will be enforced by SORTLO. Thus, in order to respect **Safe Trans. Prop.** (Eq. 5.2), if at the current time slot the condition does not hold, then the task is promoted with the highest priority and put at the beginning of the ready list. The rest of the function is similar to Alg. 1: the ready list is updated by adding newly activated tasks (l. 19) and by removing tasks that finished their execution (l. 20), MC-DAGs are reactivated when we reach their period (l. 21-26) and, the final step returns the scheduling table if VERIFYCONSTRAINTS has not been violated.

Going back to the example presented in Fig. 5.8, we are going to apply Alg. 2 on that system. The ready list is initialized and the source vertices A, a and b are added first. Their laxities are the following: $L^{LO}_{a,1}(0) = 2$, $L^{LO}_{b,1}(0) = 2$ and $L^{LO}_{A,1}(0) = 12$. Once task A is fully allocated at time slot 10, tasks B, C, D and G are activated and added to the ready list. We have the following laxities $L^{LO}_{a,1}(1) = 2$, $L^{LO}_{b,1}(1) = 2$, $L^{LO}_{B,1}(1) = 12$, $L^{LO}_{D,1}(1) = 12$, $L^{LO}_{C,1}(1) = 14$ and $L^{LO}_{G,1}(1) = 19$. Tasks are allocated according to their laxities until time slot 7, where tasks C and G are promoted (*i.e.* they are considered as zero-laxity tasks for that time slot and are put at the top of the ready list) in order to respect **Safe Trans. Prop.**. Other promotions occur at time slots 8 (task C), 10 (task D), 11 (task D), 14 (task F), 15 (task F) and 20 (task J).

The final LO-criticality scheduling table is illustrated in Fig. 5.11. Again, blue boxes represent the execution of MC-DAG G_1 and orange boxes represent the execution of MC-DAG G_2. The system is schedulable in LO-criticality mode and thanks to the task pro-

Algorithm 2 Computation of the LO scheduling table with G-ALAP

1: **function** CALCSLO($\mathcal{S}$: MC system to schedule)
2: $Ready \leftarrow \emptyset$ ▷ List of ready tasks sorted by laxities
3: **for all** τ_i **do**
4: $R_i^{LO} \leftarrow C_i(LO)$ ▷ Remaining execution time
5: $Ready \leftarrow Ready \cup \{\tau_i \mid pred(\tau_i) = \emptyset\}$ ▷ Add source vertices
6: **end for**
7: **for all** timeslots $t < H$ **do**
8: SORTLO($Ready, t$)
9: **if** VERIFYCONSTRAINTS($Ready$) $= \perp$ **then**
10: **return** NOTSCHEDULABLE
11: **end if**
12: **for all** cores $c \in \Pi$ **do**
13: $\tau_i \leftarrow$ head of $Ready$ not being allocated
14: $S^{LO}[t][c] \leftarrow \tau_i$ ▷ Allocate task τ_i
15: $R_i^{LO} \leftarrow R_i^{LO} - 1$
16: **end for**
17: $Ready \leftarrow Ready \cup \{succ(\tau_i) \mid \forall \tau_j \in pred(succ(\tau_i)), R_i^{LO} = 0\}$
18: $Ready \leftarrow Ready \setminus \{\tau_i \in Ready \mid R_i^{LO} = 0\}$
19: **for all** $G_j \in \mathcal{G}$ **do**
20: **if** $t+1 \mod T_j = 0$ **then** ▷ Reactivation of G_j
21: $\forall \tau_i \in G_j,\ R_i^{LO} \leftarrow C_i(LO)$
22: $Ready \leftarrow Ready \cup \{\tau_i \in G_j \mid pred(\tau_i) = \emptyset\}$ ▷ Add source vertices
23: **end if**
24: **end for**
25: **end for**
26: **return** S^{LO} scheduling table
27: **end function**

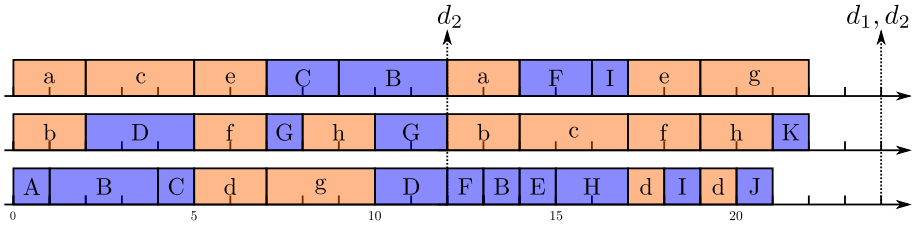


Figure 5.11: LO-criticality scheduling table for the system of Fig. 5.8

motions that were performed by the algorithm, **Safe Trans. Prop.** (Eq. 5.2) is respected. Thus, by construction the scheduling tables that were produced are MC-correct.

While the G-LLF adaptation of the G-ALAP algorithm is capable of finding MC-correct scheduling tables, it can entail an important number of preemptions. In the example we have presented, the scheduling produces 13 preemptions for 44 activations of tasks for both criticality modes. The next subsection presents another adaptation of G-ALAP that has a lowest acceptance ratio but it entails less preemptions.

5.3.2 Global as late as possible Earliest Deadline First - G-ALAP-EDF

A known limitation for LLF is the fact that the strategy usually generates numerous preemptions and migrations. This can be seen in Fig. 5.11 during time slots 11-15 where tasks B and F preempt each other. Improvements to avoid this type of behavior have been proposed in the literature [119] and can be included in our heuristic. Nonetheless, in order to demonstrate the genericity of our approach, we detail how we can obtain a version of G-ALAP based on G-EDF, we refer to this adaptation as G-ALAP-EDF.

To adapt the implementation of the meta-heuristic we define a function for the priority ordering of tasks that will be used for the SORTHI and SORTLO functions of Alg. 1 and Alg. 2. Since we are basing the algorithm in G-EDF, the priority ordering needs deadlines for each task. While we could choose the deadline of the MC-DAG (D_j), due to precedence constraints this is not a good alternative: we would like to prioritize tasks that are predecessors to have better chances of finding a correct scheduler for example. Therefore, we define d_i^χ as the *virtual* deadline of task τ_i in criticality mode χ:

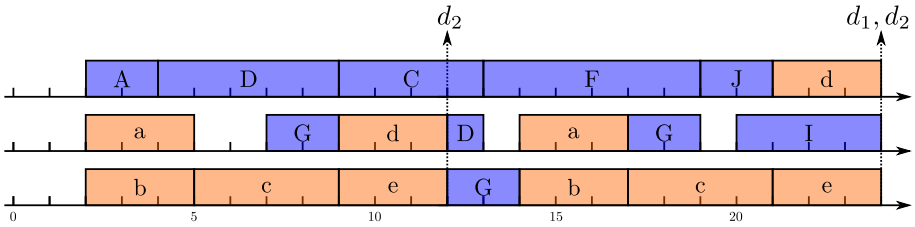
$$d_i^\chi = D_j - CP_i^\chi. \tag{5.7}$$

Where D_j is the deadline of the MC-DAG G_j, and CP_i^χ is the critical path of the vertex' successors.

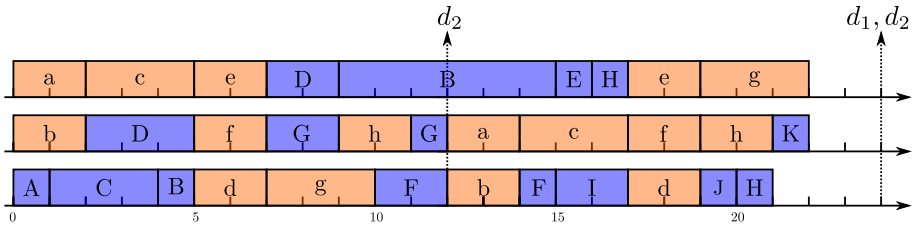
The constraints that has to be respected during the computation of the scheduling tables and that are verified by VERIFYCONSTRAINTS are the following:

- **Respecting virtual deadlines**: tasks must always respect their virtual deadlines, otherwise their successors could miss the deadline for the MC-DAG, D_j.

- **Number of slots left**: the sum of remaining times of ready tasks needs to always be inferior or equal to the number of slots available.



(a) HI-criticality mode



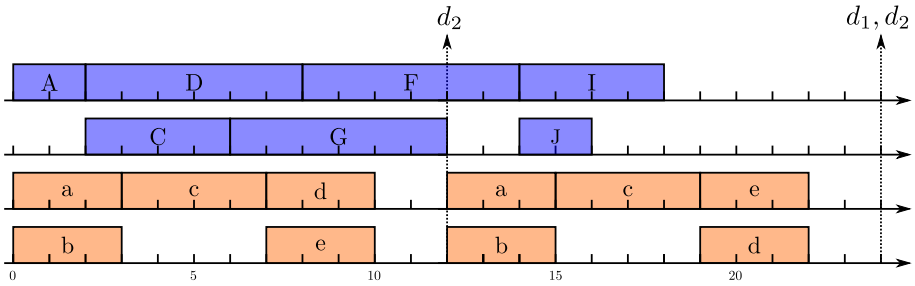
(b) LO-criticality mode

Figure 5.12: Scheduling of the system in Fig. 5.8 with G-ALAP-EDF

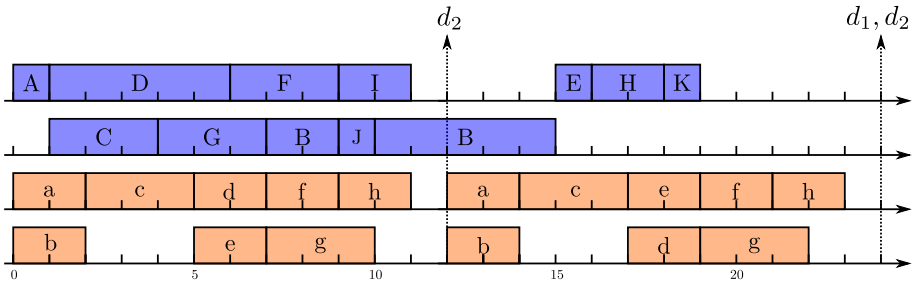
For the SORTHI and SORTLO functions the difference with the previous implementation is that, tasks are sorted in ascending order in function of their virtual deadlines. For the HI-criticality mode, the priority ordering of the ready list does not change in function of the time slot that is being allocated. Nonetheless, preemption occurs when a task with an earlier deadline is activated at some point. In the LO-criticality mode, virtual deadlines are also used and preemption can occur as well. Nonetheless, like for the SORTLO version of G-ALAP-LLF, we need to ensure that **Safe Trans. Prop.** is respected, therefore if a task can potentially violate Eq. 5.2, then it is promoted and it is put at the beginning of the ready list. Because G-ALAP-EDF is an implementation of MH-MCDAG, by construction we will obtain MC-correct scheduling tables.

In Fig. 5.12 we represent the scheduling tables that are found for the system of Fig. 5.8 using G-ALAP-EDF. Fig. 5.12a shows that a correct scheduler for the HI-criticality mode can be found. The S^{LO} table is represented in Fig. 5.12b, the system is also schedulable in this criticality mode and task promotions took place for time slots 7, 8 (tasks D and G) and 19 (task J). As we can see, the number of preemptions is also lower to the number of preemptions produced by G-ALAP-LLF: we have 8 preemptions for the 44 activations of tasks. Chapter 8 presents detailed results in terms of acceptance rate for both G-ALAP-LLF, G-ALAP-EDF and G-ALAP-HYB.

G-ALAP-HYB is the third instance of our algorithms combining both G-EDF for the HI-criticality mode and G-LLF for the LO-criticality mode. For the SORTHI function Eq. 5.7 is used to order tasks in the ready list, and SORTLO uses Eq. 5.6 to sort ready lists in addition to **Safe Trans. Prop.**



(a) Federated approach: HI-criticality mode



(b) Federated approach: LO-criticality mode

Figure 5.13: Scheduling with the federated approach

The Federated approach

The scheduling tables that are computed by the federated approach [81] are illustrated in Fig. 5.13. Each MC-DAG of Fig. 5.8, has a maximum utilization rate greater than 1 and lower than 2, therefore we need two clusters of two cores to schedule each of these MC-DAGs. Like we explained in Section 5.2, in [79; 81] the author advocates for the execution of HI-criticality tasks as soon as they become ready in order to ensure MC-correctness. Nonetheless, like we demonstrated throughout this section, G-ALAP-LLF and G-ALAP-EDF were capable of finding MC-correct schedules for the same system with only three cores, which corresponds to the lower bound of required cores to schedule the system.

In this section we have presented our global and generic implementation of the MH-MCDAG meta-heuristic: G-ALAP. The main objective of this implementations was to overcome the limitation regarding the scheduling of multiple MC-DAGs: the creation of clusters to schedule MC-DAGs leads to poor performance usage. To achieve improved performances in terms of resource usage and acceptance rate we adapted global real-time scheduler (G-LLF and G-EDF). In Chapter 8 we experimentally assess the performances of our heuristics and we compare them to the state of the art [81]. In the next section we tackle the final limitation we have identified for existing approaches: safety-critical systems often define more than two levels of criticalities. We propose a generalization of our meta-heuristic to handle an arbitrary number of criticality levels.

5.4 Generalized N-level scheduling

In safety-critical systems, certifications standards usually define more than two criticality levels, as opposed to most contribution in MC [35]. For example, for railroad systems four different Safety Integrity Levels (SIL) are used to categorize applications that are hosted in the system. In airborne systems, the DO-178B certification defines five different Software levels, depending on the effects of a failure condition in the system. Therefore, our final objective when it comes to scheduling MC-DAGs in multi-core architectures, is to define an approach capable of handling an arbitrary number of criticality levels.

We begin by extending the definition of MC-correctness, firstly introduced by Baruah in [81]. Thanks to this definition, we design a new meta-heuristic to schedule MC-DAGs with an arbitrary number of criticality levels in a multi-core architecture. At the end of this section, we briefly describe a N-criticality levels implementation of the meta-heuristic based on G-LLF.

5.4.1 Generalized MC-correctness

To define a scheduling strategy for a generalized MC system, we have to introduce new properties in order to satisfy **Schedulability in all modes of execution (Sub-problem 1.1)** and **Schedulability in case of a mode transition (Sub-problem 1.2)**. Considering more than two criticality modes makes the MC scheduling problem of the system more difficult: the scheduling needs to be compatible in all criticality modes.

In Chapter 4, we defined the task model for a MC system with N criticality modes of execution. We remind some key aspects of the generalization:

- Criticality modes can be ordered as a set of N elements: $\chi_1 \prec \chi_2 \prec \cdots \prec \chi_N$. When a TFE occurs in mode χ_ℓ the system makes a transition to mode $\chi_{\ell+1}$. The system always starts its execution in mode χ_1.
- Timing budgets are monotonically increasing, *i.e.* $C_i(\chi_1) \leq C_i(\chi_2) \leq \cdots \leq C_i(\chi_N)$.
- A task is considered to be of χ_ℓ-criticality if it executes in modes $\{\chi_1, \ldots, \chi_\ell\}$.

Definition 15. ***Generalized MC-correctness*** *- A generalized MC-correct scheduling is one that guarantees:*

1. *If all vertices of any MC-DAG execute within their $C_i(\chi_1)$, then all vertices complete their execution by the deadlines; and*

2. *If no vertex of any MC-DAG executes beyond its $C_i(\chi_\ell)$, then all vertices that are designed as being of χ_ℓ-criticality complete their execution by their deadlines.*

The first point of Definition 15 ensures the schedulability in the lowest-criticality mode: if all tasks complete their execution within their $C_i(\chi_1)$ then the system must satisfy all deadlines. The second point states that when the system is operating in the χ_ℓ mode, then all tasks that belong to that criticality level are capable of completing their execution within their $C_i(\chi_\ell)$. This definition is a simple induction of the MC-correct definition (Definition 10). We now define the necessary property that allows MC system with N-criticality modes to have safe mode transitions to the higher criticality level.

Definition 16. ***Generalized Safe Trans. Prop.***

$$\psi_i^{\chi_\ell}(r_{i,k},t) < C_i(\chi_\ell) \implies \psi_i^{\chi_\ell}(r_{i,k},t) \geq \psi_i^{\chi_{\ell+1}}(r_{i,k},t). \tag{5.8}$$

The principle of **Safe Trans. Prop.** (Eq. 5.2) is generalized by induction with this definition. Its generalization states that a $\chi_{\ell+1}$-criticality task should not have executed for less time in the χ_ℓ than it has executed in $\chi_{\ell+1}$ mode. Respecting this condition will guarantee the second part of Definition 15, since $\chi_{\ell+1}$-criticality tasks will have enough time to extend their timing budget and complete their execution within their $C_i(\chi_{\ell+1})$.

5.4.2 Generalized meta-heuristic: N-MH-McDAG

Thanks to Definition 15 and 16, we can design a new meta-heuristic to schedule multiple MC-DAG with an arbitrary number of criticality modes.

Definition 17. N-MH-McDAG *Meta-heuristic for multi-periodic MC-DAG scheduling with an arbitrary number of criticality levels.*

1. *Schedule the MC system in descending order: starting by the highest criticality mode and going towards the lowest criticality mode.*

2. *While scheduling criticality modes that are not the highest-criticality mode, enforce* ***Generalized Safe Trans. Prop.***.

Due to the fact that N-MH-McDAG enforces **Generalized Safe Trans. Prop.** during the computation of the criticality tables, we obtain Generalized MC-correctness for the scheduling produced by this meta-heuristic.

In the generalization of our scheduling heuristics, we also aim at relaxing the execution of tasks that are executed in more than one criticality mode. This can be done by scheduling these tasks ALAP for all the criticality modes they are executed in.

Safe Transition Property for Dual Graphs

When scheduling tasks ALAP in more than one criticality, we need to take particular care of **Generalized Safe Trans. Prop.** We define an equivalent necessary property that needs to be respected while we are scheduling the dual of the system. Respecting the property ensures that **Generalized Safe Trans. Prop.** is satisfied once we perform the horizontal flip of the scheduling tables.

Definition 18. ***Dual Graph Generalized Safe Transition Property**:*

$$\psi_i^{\chi_{\ell+1}}(r'_{i,k},t') \leq C_i(\chi_{\ell+1}) - C_i(\chi_\ell) \implies \psi_i^{\chi_\ell}(r'_{i,k},t') = 0. \tag{5.9}$$

This property states that while the timing budget $C_i(\chi_{\ell+1}) - C_i(\chi_\ell)$ of task τ_i has not been fully allocated in the $\chi_{\ell+1}$ mode, then the task start its execution in the χ_ℓ mode.

Theorem 2. *Respecting **Dual Graph Generalized Safe Transition Property** on the scheduling of the dual MC-DAGs in all criticality modes is equivalent to respecting **Generalized Safe Trans. Prop.** on the normal MC-DAGs.*

Proof. We recall the definition of $\psi_i^{\chi_\ell}(t_1,t_2) = \sum_{s=t_1}^{t2} \delta_i^{\chi_\ell}(s)$, where

$$\delta_i^{\chi_\ell}(s) = \begin{cases} 1 & \text{if } \tau_i \text{ is running at time } s \text{ in mode } \chi_\ell, \\ 0 & \text{otherwise} \end{cases}.$$

The dual operation on the system transform a job $j_{i,k}$ into a dual job $j^\star_{i,k^\star}$ of a task of period P as follows, $n \times P$ being the hyper-period:

A job $j_{i,k}$ with the following release date and deadline $[r_{i,k} = k \times P,\ d_{i,k} = (k+1) \times P]$, is transformed into the dual job $j^\star_{i,k^\star}$ with parameter $[r_{i,k^\star} = k^\star \times P,\ d_{i,k^\star} = (k^\star+1) \times P]$ with $k^\star = n-1-k$.

For instance, if during the hyper-period a task of period 10 has 3 jobs (0, 1, 2), the event consisting in the release of job $k = 2$ occurs at time slot 10 and in the dual space, it corresponds to the deadline of job $k^\star = 0$ at time 10.

This allows us to define the following properties for any task job $j_{i,k}$:

1. $\psi_i^{\chi_\ell}(r_{i,k},t) + \psi_i^{\chi_\ell}(t,d_{i,k}) = C_i(\chi_\ell)$.
2. $\psi_i^{\chi_\ell}(t,d_{i,k}) = \psi_i^{\chi_\ell}(r_{i,k^\star},t^\star)$ and $\psi_i^{\chi_\ell}(r_{i,k},t) = \psi_i^{\chi_\ell}(t^\star,d_{i,k^\star})$ where $t^\star = n \times P - t$.

We simplify the following notations for clarity in the proof $\psi_i^{\chi_\ell} \equiv \psi^{\chi_\ell}$, $r_{i,k^\star} \equiv r^\star$, $r_{i,k} \equiv r$, $d_{i,k^\star} \equiv d^\star$, $C_i(\chi_\ell) \equiv C(\chi_\ell)$.

Generalized Safe Trans. Prop. states:

$$\begin{aligned}
\psi^{\chi_\ell}(r,t) < C(\chi_\ell) &\implies \psi^{\chi_\ell}(r,t) \geq \psi^{\chi_{\ell+1}}(r,t) \\
C(\chi_\ell) - \psi^{\chi_\ell}(t,d) < C(\chi_\ell) &\implies C(\chi_\ell) - \psi^{\chi_\ell}(t,d) \geq C(\chi_{\ell+1}) - \psi^{\chi_{\ell+1}}(t,d) \text{ with Prop. 1} \\
\psi^{\chi_\ell}(t,d) > 0 &\implies \psi^{\chi_{\ell+1}}(t,d) - \psi^{\chi_\ell}(t,d) \geq C(\chi_{\ell+1}) - C(\chi_\ell) \\
\psi^{\chi_\ell}(r^\star,t^\star) > 0 &\implies \psi^{\chi_{\ell+1}}(r^\star,t^\star) - \psi^{\chi_\ell}(r^\star,t^\star) \geq C(\chi_{\ell+1}) - C(\chi_\ell) \text{ with Prop. 2.}
\end{aligned}$$

This can be used as is. If we take the contraposition it states that:

$$\psi^{\chi_{\ell+1}}(r^\star,t^\star) - \psi^{\chi_\ell}(r^\star,t^\star) \leq C(\chi_{\ell+1}) - C(\chi_\ell) \implies \psi^{\chi_\ell}(r^\star,t^\star) \leq 0$$

but we have $\psi^{\chi_\ell}(t_1,t_2) \geq 0$ for any t_1, t_2, thus:

$$\psi^{\chi_{\ell+1}}(r^\star,t^\star) - \psi^{\chi_\ell}(r^\star,t^\star) \leq C(\chi_{\ell+1}) - C(\chi_\ell) \implies \psi^{\chi_\ell}(r^\star,t^\star) = 0.$$

□

ALAP strategy with N criticality modes

In Section 5.2, we explained the motivation behind the idea of executing HI-criticality tasks ALAP in the HI-criticality mode. By doing so, the usable slots for all tasks during the computation of the LO-criticality scheduling table is significantly larger. This leads to a better schedulability of MC-systems.

In practice, we will apply the same type of task graph transformation we had in Section 5.3: by inverting data dependencies on MC-DAGs and scheduling the system with an ASAP strategy, we obtain an ALAP behavior for the scheduling tables once we inverse it. We will compute scheduling tables on the dual of the system: $\mathcal{S}^\star$, for all criticality modes that are not the lowest criticality mode. Nonetheless, by doing so we need to enforce the **Dual Graph Generalized Safe Transition Property** for tasks that are executed in more than one criticality level.

In Fig. 5.14, we illustrate how the scheduling of the dual graph is constrained during the allocation of a task τ for the criticality modes $\chi_{\ell+1}, \chi_\ell$ and $\chi_{\ell-1}$. The first time diagram represents the scheduling obtained for the $\chi_{\ell+1}$ mode, the task can be executed at all time slots that are represented. Once the scheduling in the $\chi_{\ell+1}$ mode is found, we proceed to schedule the task in the χ_ℓ mode. The time slots highlighted in red represent slots that cannot be used by the τ task, if the task is scheduled in those slots then **Dual Graph Generalized Safe Transition Property** is not respected. Once the χ_ℓ mode has been

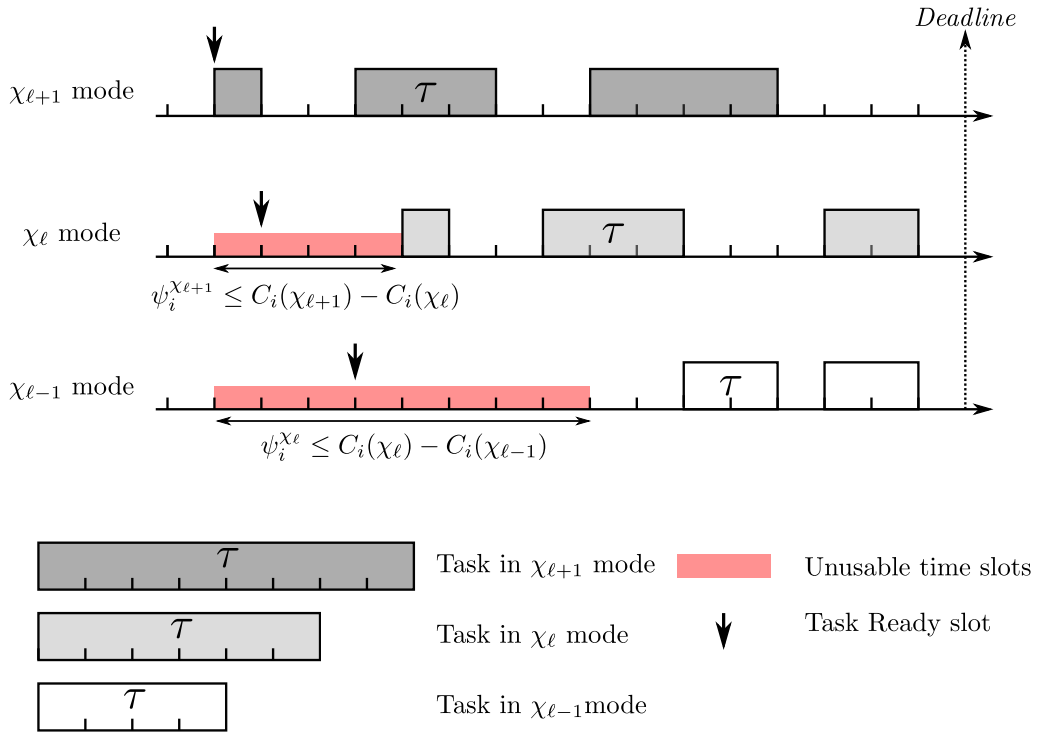


Figure 5.14: Representation of unusable time slots for a task τ_i in a generalized MC scheduling with ASAP execution

scheduled, then we can proceed to the $\chi_{\ell-1}$ mode. Again slots highlighted in red cannot be used by τ.

5.4.3 Generalized implementations of N-MH-McDAG

Like for MH-McDAG, we have implemented a generic and global version of N-MH-McDAG. Like the implementation of MH-McDAG we presented in this chapter, the algorithm produces static scheduling tables for all criticality modes. We also have adapted the implementation to use priority orderings based on G-LLF and G-EDF. For both adaptations the equations for the priority orderings of jobs remain unchanged (Eq. 5.6 for G-LLF and Eq. 5.7 for G-EDF).

The algorithms to schedule the system are very similar to Alg. 1 and 2 as well. To schedule the system in criticality modes that are not lowest criticality mode, we have almost the same algorithm than Alg. 1: the scheduling is performed with the $\mathcal{S}^\star$ system, and it is done slot by slot. The sorting function needs more tuning however, we need to enforce **Dual Graph Generalized Safe Transition Property** in addition to sort jobs according to their laxities or deadlines. Therefore tasks can be *suspended* as long as Eq. 5.9 holds: this is the behavior we represented in Fig. 5.14 with the time slots being highlighted

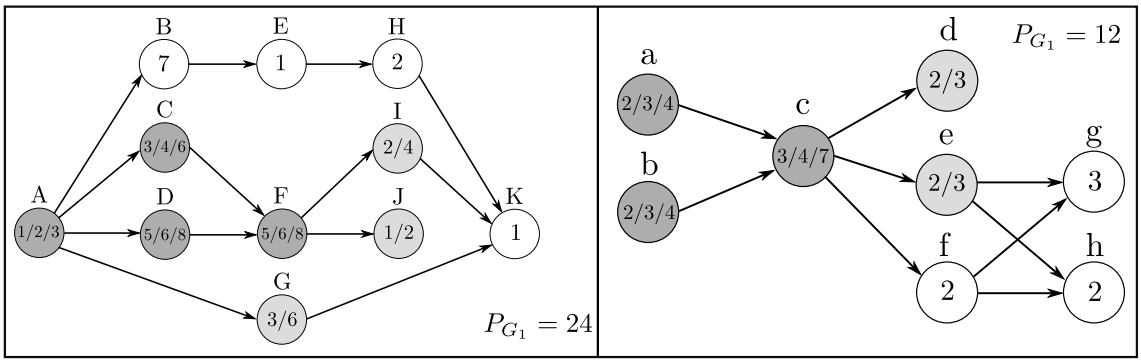


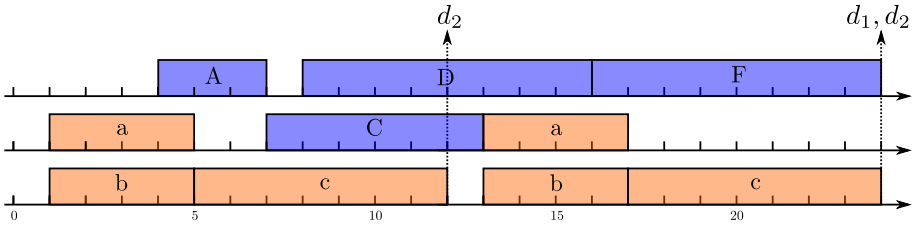
Figure 5.15: MC system with two MC-DAGs and three criticality modes

in red. Once the tables for the modes have been computed we can perform the horizontal flip in order to obtain the ALAP behavior in all tables. For the lowest criticality mode, the algorithm is almost identical to Alg. 2, the sorting function uses the chosen priority ordering for the ready list and tasks' jobs are promoted in order to respect **Generalized Safe Trans. Prop.** (Eq. 5.8).

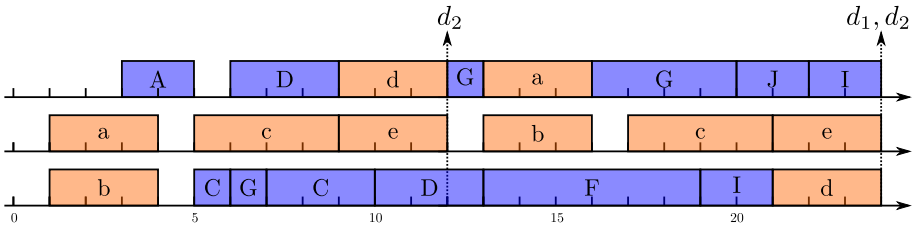
The VERIFYCONSTRAINTS function of Alg. 1 and Alg. 2 are the same that were presented on the dual-criticality version of the algorithms. For the G-LLF adaptation: jobs must not have negative laxities, not more than m tasks have zero laxities. For the G-EDF adaptation: jobs must always respect their virtual deadlines. And for both implementations the number of available slots always needs to be superior or equal to the remaining time to be allocated for jobs in the ready list.

To illustrate how the generalized scheduling works, we schedule the system presented in Fig. 5.15 with the G-LLF adaptation of the generalized algorithm. This system is composed of three criticality modes and it is based on the system of Fig. 5.8. The gray-scale represent the criticality a task belongs to: dark gray means the task belongs to the highest-criticality mode, light gray to the second higest-criticality mode and white tasks are only executed in the lowest-criticality mode. Vertices are annotated with the WCETs in all criticality modes.

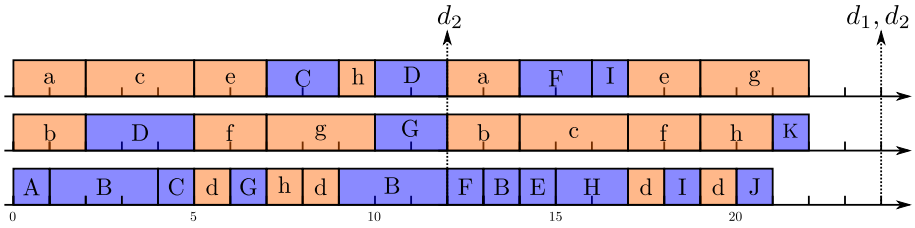
The scheduling tables produced by the G-LLF adaptation are illustrated in Fig. 5.16. We assume the system has three criticality levels: $\chi_1 \lhd \chi_2 \lhd \chi_3$. Like we explained before, in order to compute generalized MC-correct scheduling tables, the heuristics starts by the higher-criticality level, which is χ_3 in this case. During the computation of this criticality mode, there are no constraints to be respected, jobs are ordered simply according to their laxities. The scheduling table is illustrated in Fig. 5.16a. Once the χ_3 has been scheduled, the algorithm proceeds to compute the scheduling for the χ_2 mode. For this mode, **Dual**



(a) Scheduling in the χ_3 mode



(b) Scheduling in the χ_2 mode



(c) Sscheduling in the χ_1 mode

Figure 5.16: Scheduling tables for the system of Fig. 5.15 with three criticality levels

Graph Generalized Safe Transition Property is enforced. The final scheduling table computed by the algorithm is presented in Fig. 5.16b.

The final scheduling table is computed for the χ_1 mode, where an ASAP strategy is used. In this case, we schedule the normal system and not its dual. We also enforce **Generalized Safe Trans. Prop.** in this case. The table is illustrated in Fig. 5.16c.

5.5 Conclusion

This chapter presented our contributions regarding the **MC Scheduling of MC-DAGs (Problem 1)**. Like it was stated in Chapter 3, the MC scheduling problem can be decomposed into two different sub-problems: the **Schedulability in all modes of execution (Sub-problem 1.1)** and the **Schedulability in case of a mode transition (Sub-problem 1.2)**. In the first part of this chapter, we defined a property to characterize when a system is capable of performing a safe mode transition: **Safe Trans. Prop.** Building upon this property, we designed a meta-heuristic called MH-McDAG capable of producing MC-correct scheduling tables. To solve the **Schedulability in all modes of execution (Sub-problem 1.1)**, *correct* scheduling (*i.e.* a scheduler that respects deadlines and data dependencies) are produced for the HI and LO-criticality mode. To solve **Schedulability in case of a mode transition (Sub-problem 1.2)**, HI-criticality tasks can preempt LO-criticality tasks in the LO-criticality mode to respect **Safe Trans. Prop.**.

While MH-McDAG solves **Sub-problem 1.1** and **1.2**, there are **Limits of existing approaches (Sub-problem 1.3)** to schedule MC-DAGs into multi-core architectures [80; 81; 82]. These limitations can be listed as follows: • HI-criticality tasks are too constrained on their execution in LO-criticality mode, • poor resource usage of the architecture when multiple MC-DAGs are considered; and • only dual-criticality systems are considered by these approaches.

Our global implementation of MH-McDAG presented in this chapter, G-ALAP deals with the first two limitations. Our improvements over the state-of-the-art come from the fact that HI-criticality tasks execution in the LO-criticality is less constrained. We scheduled HI-criticality tasks as late as possible in the HI-criticality mode to allow LO-criticality tasks to complete their execution even when HI-criticality tasks are ready to be allocated. Another advantage of our heuristics is that they improve resource usage due to the fact that they are based on *global schedulers*: G-ALAP has been adapted to use the G-LLF and G-EDF real-time schedulers. The federated approaches [81; 82] advocate for the creation of core-clusters to schedule MC-DAGs while we are capable of using all cores for all tasks of MC-DAGs of the system.

The final part of this chapter deals with the last limitation of existing approaches: the generalization of the scheduling of MC-DAGs to support an arbitrary number of criticality levels. By applying a simple induction of MH-McDAG we are able to schedule MC-DAGs with an arbitrary number of criticality levels on multi-core architectures. The definition of MC-correctness was generalized and we also established a new necessary property: **Generalized Safe Trans. Prop.**, so that tasks executing in more than one criticality level will have enough time to extend their timing budget and not miss the deadline. Besides defining this condition we introduced its equivalent for the dual MC-system (*i.e.* a system with inversed data-dependencies). The scheduling on the dual system needs to be performed in order to constraint as less as possible the allocation of tasks during the computation of the scheduling tables. An implementation of the generalized MH-McDAG based on global algorithms has also been developed and is used as an example in this chapter.

6 Availability on data-dependent Mixed-Criticality systems

TABLE OF CONTENTS

In the previous chapter we characterized MC-correctness of scheduling strategies for MC-DAGs executing in a multi-core architecture: the system needs to be schedulable in all criticality modes and we have to be capable of making the transition to the higher criticality mode without missing deadlines for tasks. We also developed a meta-heuristic guaranteeing MC-correctness thanks to the enforcement of a safe transition property. The genericity of this meta-heuristic was demonstrated thanks to the different implementations presented in the chapter.

While the main focus of research related to mixed-criticality has been the development of scheduling policies; in our works, we are also interested in the Quality of Service (QoS) of these systems. In this chapter we look into the *availability of mixed-criticality systems*, in particular the availability of non-critical tasks: how often non-critical services are executed defines the QoS of the safety-critical system. In fact, high-criticality tasks are mostly reserved to guarantee the system's safety since they often go through costly certification processes: for example functions to avoid that the drone or the satellite crashes will be high-criticality. Nevertheless, in a safety-critical system, the end-user applications are often executed by tasks characterized as non-critical (or low-criticality): for example recording video on a exploration drone, sending telecommunication signals to mobile

phones on a satellite, *etc*. While maintaining the system from crashing is very important, delivering end-user functionalities is the main reason the system was deployed, therefore the availability of non-critical tasks is also important.

As we explained in Chapter 2, recent trends in mixed-criticality systems recognize the necessity to guarantee a minimal QoS for non-critical tasks [99; 38]. Therefore, our first objective consists in **Estimating availability rates (Problem 2)**: this can be done either numerically or experimentally, but in both cases the MC model is lacking information. To overcome this problem, we introduce in this chapter a **Fault model (Sub-problem 2.1)** allowing us to estimate how often mode transitions to higher criticality modes occur. At the same time, in most MC models once the transition to the high-criticality mode is made, there is no recovery method to reincorporate low-criticality tasks: we also propose in this chapter a **Recovery mechanism (Sub-problem 2.2)** for low-criticality tasks.

The final part of this chapter demonstrates how the **Discard MC model degrades availability (Problem 3)** of the system. To address this limitation we propose different enhancements for MC systems: the first enhancement **Limiting the number of mode transitions to higher criticality modes (Sub-problem 3.1)** and the second enhancement consists in **Incorporate mechanisms for availability enhancement (Sub-problem 3.2)** to re-evaluate the availability of the system.

6.1 Problem overview

Commercialized safety-critical systems are subject to meticulous development cycles, tests and certification processes in order to guarantee reliability, availability, maintainability, safety, security, (RAMS) properties [110]. When we look into MC systems, safety is ensured thanks to high-criticality execution modes: the most critical software components are capable of extending their timing budget in order to complete their execution, guaranteeing that their services will be available in case of a TFE. As a matter of fact, in the literature of MC systems, most contributions are focused on the scheduling problem [35] and in order to schedule such systems, the discard approach [36; 8; 37] has been the prevailing model. Nonetheless, since the discard model advocates for the interruption of LO-criticality tasks when a TFE occurs, the availability of these tasks is affected. Even if tasks are considered as LO-criticality, their execution on a safety-critical system is also important: most functionalities that are not related to the system's safety will be executed by LO-criticality tasks. For example video recording for an exploration drone will most likely be executed by LO-criticality tasks while the autopilot software will be of HI-criticality to avoid crashing or losing the drone. The main propose of the drone is

to collect exploration data by the means of video, yet tasks fulfilling this duty are considered as LO-criticality and can potentially be interrupted if HI-criticality tasks need more processing time. At the same time, it is unrealistic to assign the highest criticality to all tasks in a system since certification processes are very costly and computation resources would be wasted.

In this chapter we want to answer the following question: *how available are LO-criticality tasks in a MC system?* Calculating the availability of a task boils down to determining a ratio between the expected uptime of a task (*i.e.* the time the task executes without failures) to the aggregate of expected values of up and downtime (*i.e.* the time the task was supposed to be executed but was discarded). The availability of a task τ_i is measured for a given timelapse. Its formula is given by:

$$A_i(t_1,t_2) = \frac{Uptime_i}{Uptime_i + Downtime_i}. \tag{6.1}$$

When we consider periodic systems, like the MC-DAGs we introduced in Chapter 4 (Section 4.1), the availability can be measured in function of the hyper-period of the system. At each hyper-period we can verify which tasks were and were not executed. In fact, during the hyper-period, a task is supposed to be executed a fixed number of times: HP/P_i. The availability of a task is therefore given by the following equation:

$$A_i(t_1,t_2) = \frac{N_i}{N \times \frac{HP}{P_i}}.$$

The availability of a task is measured during the time interval $[t_1,t_2]$. N_i is the number of times task τ_i is executed during the interval specified. N is the number of times the system was executed during its hyper-period between t_1 and t_2. Therefore $N \times \frac{HP}{P_i}$ is the number of times the task was supposed to be executed during the N hyper-periods that took place in the interval $[t_1,t_2]$. Nonetheless, a TFE can be triggered by any task at some point during the execution of the system, potentially interrupting the execution of τ_i: this affects the value N_i, while the value N keeps increasing since the system keeps its execution in a higher-criticality mode. Therefore, we need to know how often mode transitions are triggered in the MC system by incorporating a **Fault model (Sub-problem 2.1)** . Most MC contributions only consider the degradation of LO-criticality tasks in favor of HI-criticality tasks, for this reason, we want to incorporate a **Recovery mechanism (Sub-problem 2.2)** of LO-criticality tasks, *i.e.* a mechanisms that would allow the system to switch from the HI-criticality to the LO-criticality mode. Without a recovery mechanism

the availability of LO-criticality tasks becomes negligible since only the N value increases as time progresses.

Our contributions related the availability of MC systems demonstrate that the **Discard MC model degrades availability (Problem 3)**. Since we are considering multi-core architectures for our works, when a TFE occurs, the system changes to the HI-criticality mode in a synchronous way, *i.e.* all cores start executing tasks in the HI-criticality mode and LO-criticality tasks are discarded. This is a limitation in terms of availability since LO-criticality tasks can be interrupted by independent tasks that were executed in other cores, in other words, some tasks are being interrupted while they could finish their execution. Therefore, **Limiting the number of mode transitions to higher criticality modes (Sub-problem 3.1)** is a relevant enhancement that can be considered. We propose a more detailed fault propagation model in order to contain faults and limit the number of mode transitions. This same principle of limiting the interruption of LO-criticality tasks has been presented in partitioned [61] and semi-partitioned [63] MC models but for independent tasks and performing tasks migrations. The approach that we have taken is closer to the *selective degradation* proposed in [60], where tasks are grouped in order to reduce the impact of mode switch within that group. Finally, commercialized safety-critical systems **Incorporate mechanisms for availability enhancement (Sub-problem 3.2)** and often define more than two criticality levels. We propose to look into two different fault-tolerance mechanisms: design patterns using replicas with voting mechanisms [120] and the $(m-k)$ firm tasks [102]. The incorporation of such mechanisms changes the execution model of the system and analyzing the availability of non-critical tasks considering more than two criticality levels forces us to run systems simulations in order to estimate availability rates. To perform these simulations we transform the system and its scheduling tables into probabilistic automata. Then the PRISM model checker [114] is capable of performing the system simulations to estimate the availability.

6.2 Availability analysis for data-dependent MC systems

In this section we present how we perform the availability analysis for non-critical tasks in MC systems (**Problem 2**). We begin by presenting the fault model we consider in order to estimate how often mode transitions to the higher criticality mode occur (**Sub-problem 2.1**). Then we explain the mode recovery mechanism that can be used in order to reincorporate non-critical tasks and switch back to the low-criticality mode (**Sub-problem 2.2**). We are only interested in *the availability of LO-criticality outputs* since data produced by these tasks are used by the end-user or other devices.

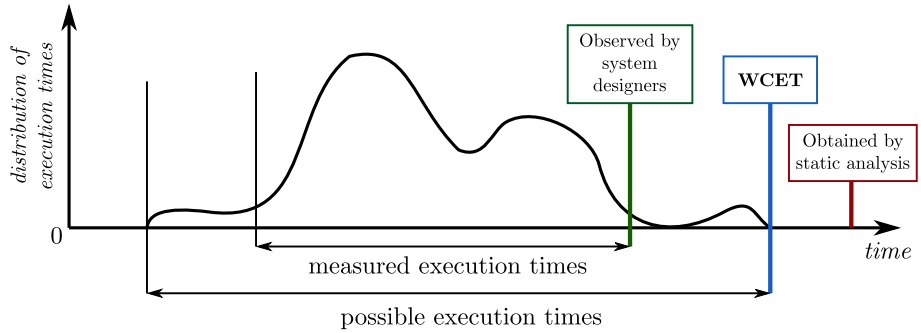


Figure 6.1: Execution time distributions for a task

6.2.1 Probabilistic fault model

The works of this thesis are based on the MC model first introduced by Vestal [4], the basis of this model relies on the fact that different *timing budgets* can be given to a task depending on the *assurance level* that needs to be ensured: the higher the assurance level is, the more conservative the verification process and hence the greater the timing budget will be. Criticality modes were introduced [36; 8; 37] for dual-criticality systems, in order to allow tasks to switch from one timing budget to another: this change occurs whenever a HI-criticality task needs more timing budget to complete its execution (or if a LO-criticality task did not complete its execution). The fault model that we present in this section only takes into account timing failures that would cause mode transitions, we called them Timing Failure Events (TFEs) in the previous chapters. These failures can be caused by various things like execution paths in the software, shared caches and data buses in the architecture, *etc*. We explain how we incorporate and the logic behind the *probabilistic fault mode*l for MC systems that we consider. Other types of failures are out-of-scope for this thesis but can be considered as perspectives for future works related to the availability analysis of MC systems.

During the conception of a safety-critical system, system designers need to estimate the WCET of tasks that will be executed in the system. Like we explained in Chapter 2, finding a tight WCET is very difficult [26] and due to certification processes, upper-bounds for WCETs are often used. This procedure often leads to poor resource usage since the upper-bounded WCET is almost never consumed by the task during its execution.

Fig. 6.1 (based on [26]) presents the detailed execution time distribution for a task. Depending on the method applied to estimate the WCET, the range of *observed* execution times can vary. For instance the execution time range covered by measurement techniques, *e.g.* Measured Based Timing Analysis (MBTA) for instance, tend to be more restricted than the range produced by static methods. In Fig. 6.1 we can see that "measured execu-

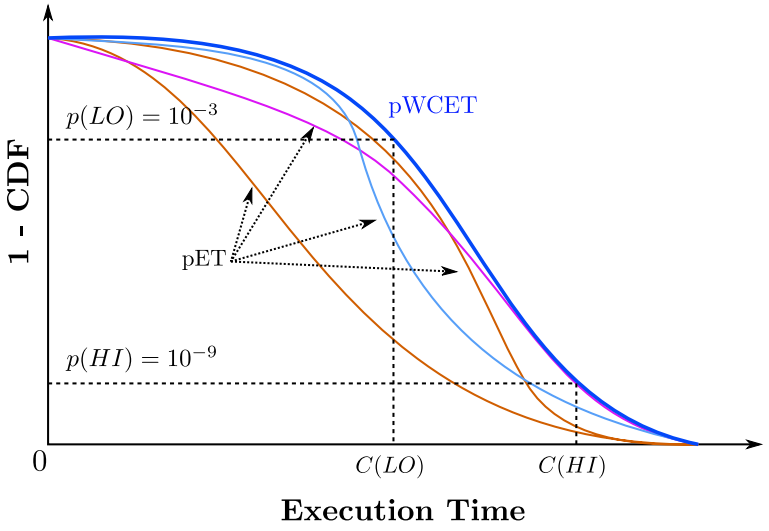


Figure 6.2: Exceedance function for the pWCET distribution of a task

tion times"', typically obtained by MBTA, do not include the actual WCET of the task. What we want to demonstrate is that *the estimation of the WCET is related to the probability of having a TFE*. For example if the system designer decides to use the maximal observed time as the timing budget for its task, then there is a non-negligible probability that a TFE will occur because the task needs more processing time in order to complete its execution. This can be clearly seen in Fig. 6.1: the execution time distribution goes beyond the maximal observed execution time. With the MC model, the maximal measured execution time and the upper-bounded WCET can be used: typically one would be the $C_i(LO)$ capable of satisfying the execution for most cases; and the other would be the $C_i(HI)$ for the rare cases when the task needs more timing budget.

In the context of safety-critical systems, standards define *levels of assurance* for tasks executed in the system. A failure in the higher levels can cause important losses, therefore lower failure probability are ensured, *e.g.* on airborne systems Level A of the DO-178B certification [111] has a failure rate of $10^{-9}/h$, while Level C has a failure rate of $10^{-5}/h$. The certification process for a Level A task will most likely demand that an upper-bounded WCET is used, and therefore the $10^{-9}/h$ failure rate can always be guaranteed. On the other hand, if the maximal observed execution is used then the failure rate would be closer to $10^{-7}/h$. Therefore we can *associate failure probabilities to tasks in function of their timing budget*. This conjecture is also used in the works from [113], where a failure probability can be deduced in function of the WCET that is assigned to a task. The objective is to choose a WCET accordingly to the failure probability and therefore the failure rate we need to guarantee.

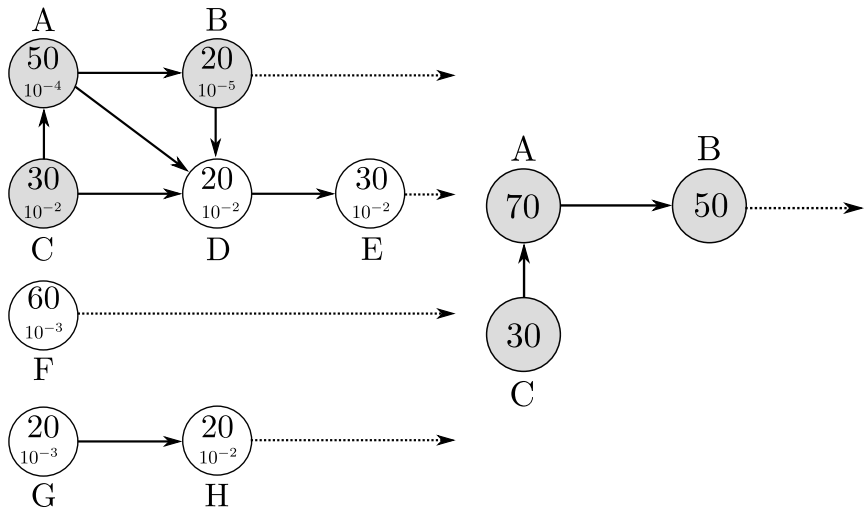
In Fig. 6.2 (extracted from [113]), the exceedance function for the probabilistic-WCET (pWCET) of a task is illustrated. From this function, an execution time that has no higher probability of being exceeded can be deduced. The figure shows that the selected $C_i(LO)$ has a probability $p_i(LO) = 10^{-8}$ of causing a TFE. For the $C_i(HI)$ the probability is equal to 10^{-12} (a lower probability than the failure rate for the Level A of the DO-178B standard).

Complementing the task model we presented in Chapter 4, a task τ_i is therefore characterized by a set of failure probabilities, $p_i(\chi_\ell)$ is the timing failure probability of task τ_i for the χ_ℓ criticality mode. Timing failure probabilities are monotonically decreasing following the observation from the seminal work on MC [4]: higher assurance level implicate that lower failure probabilities are enforced. This set of timing failure probabilities is our probabilistic fault model for MC systems and solves **Sub-problem 2.1** since we can now estimate how often a mode transition to the higher criticality mode occurs.

6.2.2 Mode recovery mechanism

With the probabilistic fault model we have defined, we have the first requirement in order to estimate the availability rates thanks to Eq. 6.1. Nonetheless, few MC scheduling techniques are capable of reincorporating LO-criticality tasks after the system has switched to the HI-criticality mode. Most works interested in the QoS for LO-criticality tasks propose to degrade the execution of these tasks either by changing their periods [38], migrating them to other cores [61; 63] or interrupt only a subgroup of tasks [60]. The problem with these approaches is that they rely on the hypothesis that enough processing power is available to not drop completely the execution of LO-criticality tasks. Reincorporating LO-criticality tasks after they have been dropped due to the mode transition to has been proposed for the Adaptive Mixed-Criticality scheduler and independent task sets [64; 65]. Since our works are based on data-dependent tasks, we need to consider a different recovery mechanism.

The mode recovery we consider in our works consists in *resuming the execution of the MC system in the LO-criticality mode at the hyper-period*. Since our scheduling methods are work-conserving it is rare that idle instants will occur during the execution of the system. Assuming that the scheduling strategy is **MC-correct**, the system can start allocating LO-criticality tasks once again when the hyper-period is reached. If other failures occur after the system is executing in the LO-criticality mode, then the mode transition is guaranteed to not miss a deadline (characteristic guaranteed by the MC-correctness).



(a) LO-criticality mode (b) HI-criticality mode

Figure 6.3: MC system example for availability analysis, $D = 150$ TUs.

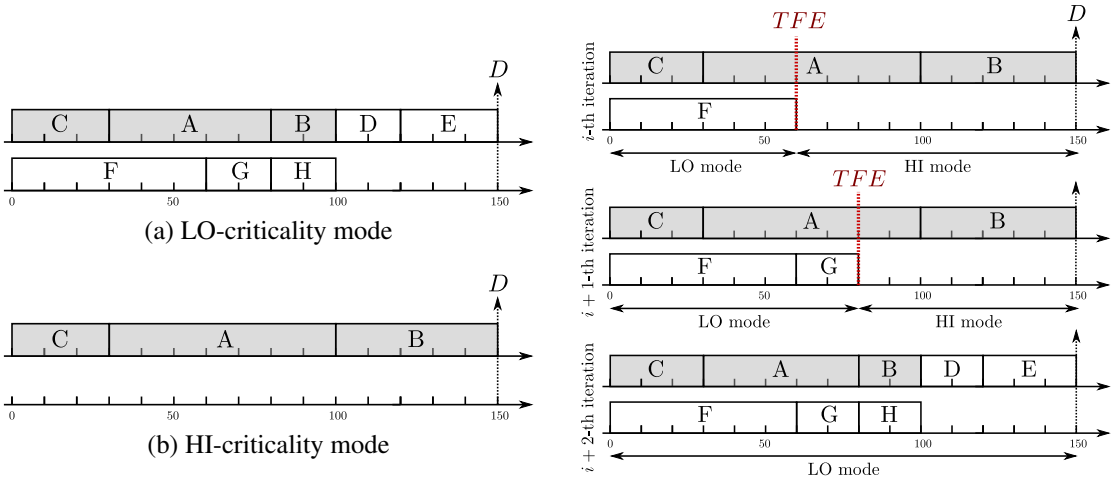
Thanks to the **Recovery mechanism (Sub-problem 2.2)** we have introduced and to the **Fault model (Sub-problem 2.1)** , the availability of LO-criticality tasks can be estimated. The number of task executions (N_i in Eq. 6.1) will not remain constant if a TFE takes place. As a matter of fact, the availability for a k-th job $j_{i,k}$ of task τ_i in a dual-criticality system in the LO-criticality mode can be estimated with the following formula, for $t \in [0; +\infty[$:

$$A_{i,k}^{LO} = 1 - \left(p_i(LO) + \sum_{\tau_j \in before(j_{i,k})} p_j(LO) \right). \tag{6.2}$$

Where $before(j_{i,k})$ is the set of tasks executed *before* the task job $j_{i,k}$ by the scheduling strategy. The availability of the task itself is equal to the average of jobs' availability.

The formula states that the availability of a task is equal to the probability of the task not failing in the LO mode ($1 - p_i(LO)$), to which we need to add the failure probability of tasks executed before τ_i in the scheduling table ($\sum_{\tau_j \in before(j_{i,k})} p_j(LO)$). The recovery mechanism allows tasks to execute again after the hyper-period has been reached.

Let us consider the MC system presented in Fig. 6.3, we illustrate our recovery mechanism and how we are capable of obtaining the availability rates for LO-criticality outputs thanks to this example: we limit the example to a single dual-criticality MC-DAG for clarity, nonetheless multiple MC-DAGs can be considered for the analysis. The system presented in the figure has three HI-criticality tasks represented in gray, and five LO-criticality tasks represented in white. The LO-criticality mode is illustrated in Fig. 6.3a and the HI-criticality mode in Fig. 6.3b. Full arrowed lines represent data dependencies



(a) LO-criticality mode

(b) HI-criticality mode

(c) Mode recovery explanation

Figure 6.4: Scheduling tables for the system of Fig. 6.3

among tasks and dotted lines represent output data production (*i.e.* data produced by these tasks is used by the user or other devices). For the LO-criticality mode (Fig. 6.3a), timing budgets are annotated by the upper numbers on the vertices. The second number included in the vertex is the timing failure probability of a task: like we explained in the previous subsection, the failure probability is related to the timing budget that was given to the task. For the HI-criticality mode (Fig. 6.3b), only the timing budgets are given: the system cannot make an additional transition to a higher-criticality mode. The deadline for the system is 150 TUs. The utilization in LO mode is $U_{LO}(\mathcal{S}) \approx 1.67$ and in HI mode $U_{HI}(\mathcal{S}) = 1$. Hence, at least 2 cores are needed in order to schedule the system.

For this example, we apply the G-ALAP-EDF our scheduling algorithm presented in Chapter 5 since it entails less preemptions than G-ALAP-LLF. The scheduling tables obtained are illustrated in Fig. 6.4 for the LO-criticality mode (Fig. 6.4a) and the HI-criticality mode (Fig. 6.4b). The hyper-period of the system is at 150 TUs and is also called an *iteration*.

Let us now consider a scenario where task F has a TFE for an iteration i. Fig. 6.4c represents such scenario. At 60 TUs, the expected completion time for task F, the OS detects a failure and the system changes to a HI-criticality mode. Since we are in a discard MC system, LO-criticality tasks are dropped (*i.e.* they are no longer executed) and HI-criticality tasks execute with an extended timing budget. The dashed line in red illustrates the TFE that occurred during the execution of task F. In order to reestablish the execution of LO-criticality tasks, for the $i+1$-th iteration of the system, we start executing the LO-criticality scheduling table again. The reasoning behind this approach is that, if tasks are

Table 6.1: Availability rates with the discard MC approach

	Availability Rate
Outputs	Discard MC
E	95.789%
F	98.9%
H	97.79%

capable of completing their execution within their $C_i(LO)$, then processing time for LO-criticality tasks becomes available. After all, the LO-criticality table was built in a way that all tasks are schedulable within their $C_i(LO)$ and we can switch to the HI-criticality mode safely if another TFE takes place (properties demonstrated for MH-McDAG in Chapter 5). We can see in Fig. 6.4c that the system tries to restart the scheduling in the LO-criticality mode and successfully executes task F this time. Nonetheless, during the execution of task A, another TFE occurs and the system makes the transition to the HI-criticality mode once again. It is only at iteration $i+2$ that the system is capable of executing all tasks within their $C_i(LO)$: we consider that the LO-criticality mode has been reestablished when this happens.

6.2.3 Evaluation of the availability of LO-criticality outputs

Availability rates obtained by our analysis are presented in Table 6.1. Like we mentioned, we are only interested in the LO-criticality outputs, tasks E, F and H in the system of Fig. 6.3. The rate of task H is obtained with the following formula:

$$\begin{aligned} A_{H,1}^{LO} &= 1 - \left(p_H(LO) + \sum_{\tau_j \in before(j_{H,1})} p_j(LO) \right) \\ &= 1 - (p_H(LO) + p_A(LO) + p_C(LO) + p_F(LO) + p_G(LO)) \\ &= 1 - (10^{-2} + 10^{-4} + 10^{-2} + 10^{-3} + 10^{-3}) \\ &= 0.9779. \end{aligned}$$

The $before(H)$ set is composed of tasks A, B, C, F and G since they are executed before task H in the S^{LO} scheduling table (Fig. 6.4a). The same formula is applied for the other LO-criticality outputs. Nevertheless, the difference of availability between tasks is highly dependent on the scheduling of the system: all tasks that are executed before a given task, degrade the availability since more potential TFEs can occur during the execution. *E.g.* for task E of Fig. 6.3, since it is scheduled at the end of the LO scheduling table (Fig. 6.4a),

its availability is lower than task H even if both have the same failure probability. This difference of almost 2% is quite significative for safety-critical systems, like we explained, failure rates are measured up to the 10^{-9}.

As we have demonstrated in this section, thanks to the probabilistic **Fault model (Sub-problem 2.1)** and to the **Recovery mechanism (Sub-problem 2.2)**, we are capable of **Estimating availability rates (Problem 2)** for LO-criticality tasks. In the next section we present enhancements that can be deployed on MC systems in order to improve the availability of LO-criticality tasks since the **Discard MC model degrades availability (Problem 3)**.

6.3 Enhancements in availability and simulation analysis

In the previous section we enriched the MC-DAG model with a probabilistic failure model to represent potential TFEs that can occur during the execution of the system. At the same time, we introduced a recovery mechanism allowing us to perform an availability analysis and concluded that the discard MC model [36; 8; 37] degrades the availability of LO-criticality tasks significantly. In this section we detail the limits in terms of availability of the MC discard approach, to then propose two different types of enhancements allowing us to improve significantly the availability of LO-criticality tasks. The inclusion of enhancements to MC systems led to the necessity of defining translation rules to probabilistic automata: the production of these automata is defined for the generalized case where systems have more than two criticality levels.

6.3.1 Limits of the discard MC approach

The MC discard approach [36; 8; 37] has been the most dominant model for MC scheduling due to its simplicity and its results in terms of acceptance rate. Because LO-criticality tasks are completely interrupted when the system makes the transition and are not executed in the HI-criticality mode, HI-criticality tasks have better chances of being schedulable since the whole architecture is available to them. This approach is only focused on safety, *i.e.* guaranteeing that HI-criticality tasks will be able to complete their execution, to the detriment of availability for LO-criticality tasks. Some recent contributions in MC acknowledge that the complete interruption of LO-criticality tasks affects the QoS [99; 38], which is a main concern for safety-critical systems. For example the elastic MC model [38; 66] proposes to change periods of LO-criticality tasks when the system is in a HI-criticality mode in order to execute them less often, the utilization

of LO-criticality tasks is reduced by applying this period transformation. Other type of service guarantee schedulers have been proposed since then, partitioned and semi-partitioned systems are capable of limiting the mode transition to their partitions and migrate LO-criticality tasks to other cores so they can complete their execution [61; 63]. Other adaptations of LO-criticality tasks execution have also been proposed [49; 83]. Our approach is similar to the selective degradation proposed in [60], the impact of the TFE will only affect a subgroup of tasks. Our goal is to *deploy enhancements to the system while still conserving the execution model we introduced in Chapter 4: thus, we want to incorporate enhancements to the MC discard model.*

The limits we have identified thanks to our availability analysis are the following:

- Mode transitions to higher criticality modes are made in a synchronous manner where all cores switch to the higher criticality scheduling table. While this is necessary in most cases to guarantee MC correctness, tasks can be interrupted by any task that was executed before, regardless of their criticality. For example, since LO-criticality tasks are less constrained in terms of service guarantee, it is natural that their failure rate is greater than HI-criticality tasks' rates: a TFE coming from a LO-criticality task triggers a mode transition, yet HI-criticality tasks should be more capable of completing their execution within their $C_i(LO)$. In other words, HI-criticality will have their timing budget extended, but it is very likely that it was not necessary since the TFE was caused by a LO-criticality task. **Limiting the number of mode transitions to higher criticality modes (Sub-problem 3.1)** is the first objective we want to solve, in order to avoid discarding non-criticality tasks.

- Safety-critical systems often **Incorporate mechanisms for availability enhancement (Sub-problem 3.2)**. The notion of robustness on MC systems is studied in [101]. In this section, we explain how these enhancements can be deployed into the system and how the availability rate can be estimated. These mechanisms should be compatible with the MC discard approach. We focus only on two mechanisms to demonstrate the interest of the enhancements. Due to the execution behavior of these robust mechanisms we are forced to estimate availability thanks to system's simulations. Future works can be built on our results in order to consider other types of enhancements.

We present in the next subsections our contributions in terms of availability enhancements for non-criticality tasks. These contributions are described for a generalized MC system with an arbitrary number of criticality levels. For clarity, throughout the examples we just present results on a dual-criticality system. We begin by describing a fault

propagation model capable of limiting the number of mode transitions to higher criticality modes. We then incorporate design patterns like Triple Modular Redundancy [120] and weakly hard real-time tasks [102] as additional enhancements to the overall system. In order to estimate the availability of a system enriched with these enhancements and with more than two criticality levels, we have defined translation rules in order to obtain PRISM probabilistic automata [114].

6.3.2 Fault propagation model

The first enhancement for MC system we consider is a *detailed fault propagation model*. In fact, safety-critical systems are equipped with software or hardware mechanisms that limit fault propagations to other components so as to improve dependability [121; 122; 123]. Instead of switching systematically to the higher criticality mode whenever a task has a timing failure, we differentiate TFEs coming from a task with the same criticality level the system is executed in, or a TFE coming for a task with a higher criticality level than the system is executed in. The main objective of this fault propagation model is to avoid dropping the execution of tasks that could complete their execution by performing unnecessary mode transitions to the higher criticality mode. This principle of limiting the fault propagation for MC systems has been initially studied in [60]. Nonetheless, it differs from our fault model since we try to limit the number of mode transitions in function of tasks' criticalities, whereas the proposed approach performs the mode transitions whenever a TFE takes place.

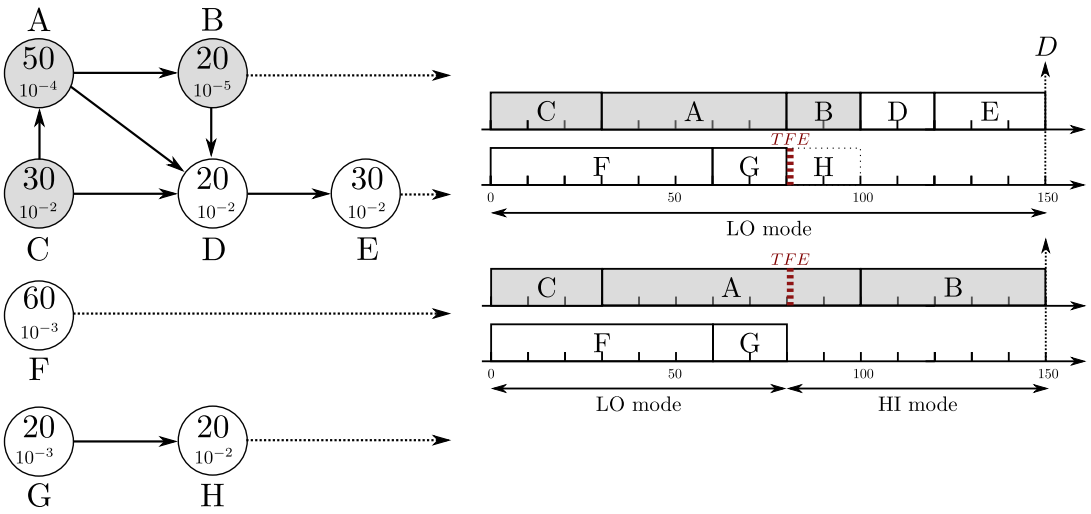
The fault propagation model we define has the two following behaviors:

1. **When system is executing in the χ_ℓ mode and a $\chi_{\ell+1}$-criticality task has a timing failure**: make the mode transition to the $\chi_{\ell+1}$-criticality mode.

2. **When the system is executing in the χ_ℓ mode and a χ_ℓ-criticality task has a timing failure**: interrupt only the successors of the task that had a failure (*i.e.* tasks that depend on the data produced by the failing task). $\chi_{\ell+1}$-criticality task (or higher) and unaffected χ_ℓ-criticality tasks can finish their execution within their $C_i(\chi_\ell)$.

By considering this fault propagation model in our MC systems, the availability formula changes for the dual-criticality case:

$$A_{i,k}^{LO} = 1 - \left(p_i(LO) + \sum_{\tau_j \in pred(\tau_i) \cup beforeHI(j_{i,k})} p_j(LO) \right). \tag{6.3}$$



(a) MC system

(b) Scenario with TFEs on LO and HI-criticality tasks

Figure 6.5: Fault propagation model with an example

The set considered of tasks that can affect the availability tends to be smaller: $pred(\tau_i)$ is the set of predecessors of the task (predecessors in the DAG, *i.e.* tasks that send data to τ_i) and $before_{HI}(\tau_i)$ is the set of HI-criticality tasks executed before τ_i. If any of these tasks fail then τ_i is interrupted. This set covers all tasks that have an influence on task τ_i execution, the others are not communicating directly or indirectly with it, so they can be ignored by the availability formula.

Fig. 6.5 illustrates our fault propagation model. The iteration of the system illustrated in Fig. 6.5b, shows the behavior of our fault propagation model when a LO-criticality task has a TFE. In the example, task *G* had a failure and task *H* depends on its output, as shown in Fig. 6.5a, we just discard task *H* (represented with the dotted box). In order to keep LO-criticality tasks running, we discard only tasks that have dependencies with the failing task, in other words the successors of the task that had a failure are no longer executed. At the same time, since HI-criticality tasks are capable of completing their execution even when they communicate with LO-criticality tasks (Chapter 5 Section 5.2.3), we allow them to complete their execution within their $C_i(LO)$.

The second iteration of the system illustrated in Fig. 6.5b shows the behavior of the failure propagation model when a HI-criticality task has a timing failure. In this case, task *A* has a TFE and the system makes a transition to the HI-criticality mode (dropping LO-criticality tasks and extending timing budgets for HI tasks). Since HI-criticality tasks must always complete their execution within their deadlines even in the case of a TFE, we drop all LO-criticality tasks, the timing budget that was previously used by LO-criticality tasks can potentially be used by HI-criticality tasks.

The availability of task H of Fig. 6.5a can be calculated as follows:

$$\begin{aligned} A_{H,1}^{LO} &= 1 - \left(p_H(LO) + \sum_{\tau_j \in pred(H) \cup before_{HI}(j_{H,1})} p_j(LO) \right) \\ &= 1 - (p_H(LO) + p_A(LO) + p_C(LO) + p_G(LO)) \\ &= 1 - (10^{-2} + 10^{-4} + 10^{-2} + 10^{-3}) \\ &= 0.9889. \end{aligned}$$

As opposed to the previous calculus now the availability of task H is only dependent on the execution of tasks A, C and G (as opposed to A, C, F and G). The availability was increased of 1.1% thanks to the fault propagation model. We provide results for the rest of LO-criticality outputs in the next subsection where we present the fault-tolerant mechanisms we consider in our availability analysis.

6.3.3 Fault tolerance in embedded systems

Fault-tolerant mechanisms have been widely deployed into safety-critical systems in order to improve dependability. In this subsection we describe how a MC system can incorporate design patterns like Triple Modular Redundancy (TMR) [120] and/or weakly hard real-time tasks [102] in order to improve the availability of LO-criticality tasks. We present the deployment of these mechanisms with the example of the system presented in Fig. 6.3, our goal is to improve the availability of output E, which has the lowest availability rate out of all LO-criticality outputs (as shown in Table 6.1). We detail the changes that take place in the execution model of the system and our contributions in order to estimate the availability rate of the MC system with these new enhancements.

Triple Modular Redundancy

Design patterns like Triple Modular Redundancy (TMR) [120] are widely used in safety-critical systems in order to improve RAMS properties. We focus only on one design pattern as a motivating example to demonstrate the interest of considering such design patterns during the conception of MC systems.

The TMR consists in replicating a software (or hardware) component three times and comparing the produced results of these replicas thanks to voting mechanisms. If at least two out-of three results are the same, then the correct output can be deduced: the majority has the advantage. The idea is to mask faults that can occur during the execution of one of the replicas. In fact, having a failure in one of the replicas already occurs on rare occasions

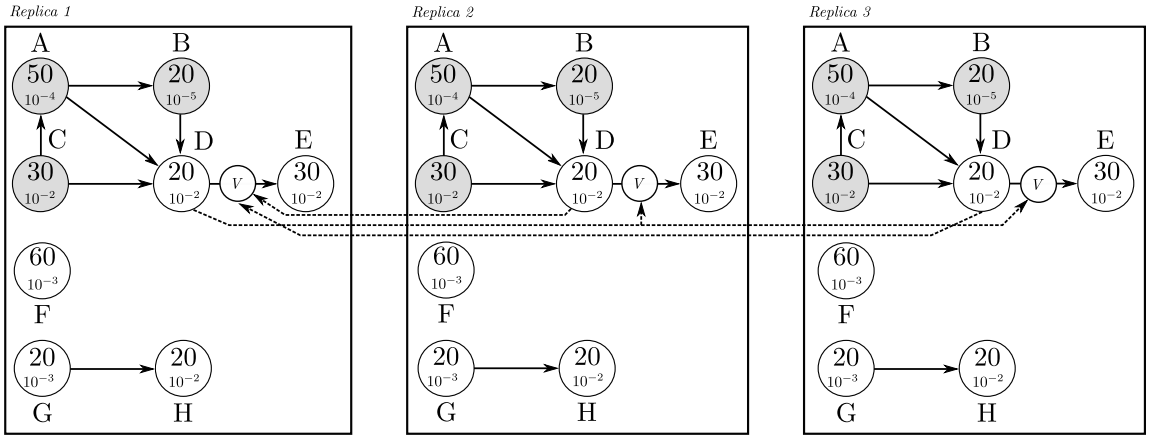


Figure 6.6: A TMR of the MC system of Fig. 6.3

for a safety-critical system, consequently having two or more replicas in a failing state at the same time is almost impossible.

To take into account the presence of replicas on the availability rates we apply a probability tree to deduce the following formula:

$$A_i^{LO,TMR} = (A_i^{LO})^3 + 3 \times \left((A_i^{LO})^2 \times (1 - A_i^{LO}) \right). \tag{6.4}$$

The availability of a task with a TMR is the equal to the combination of the three replicas being in a functional state $(A_i^{LO})^3$, plus the states when at least two replicas are in a functional state: $3 \times \left((A_i^{LO})^2 \times (1 - A_i^{LO}) \right)$.

The application of a TMR design pattern is presented in Fig. 6.6. We incorporated voting mechanisms, vertices labeled as the V task, responsible for checking outputs produced by task D and its replicas. We consider the execution of these voting mechanisms is negligible compared to the timing budget of tasks. The failure probabilities for tasks F and G remains unchanged in each one of the replicas, since the TMR does not affect them (*i.e.* none of these tasks communicate with the voting mechanism and are executed beforehand). For the E task however, its availability changes since the probability that task D has a failure is now different:

$$\begin{aligned} p_D^{TMR}(LO) &= 3 \times \left((1 - 10^{-2}) \times (10^{-2})^2 \right) + (10^{-2})^3 \\ &= 0.000298. \end{aligned}$$

The equation is again an application of a probability tree to deduce the states where task D has a failure even with the voting mechanism. D is said to be in a failing state when at least two of the replicas are also in a failing state: $3 \times \left((1 - 10^{-2}) \times (10^{-2})^2 \right)$ or when

Table 6.2: Availability rates for LO-criticality outputs with availability enhancements

	Availability Rate		
Outputs	Discard MC	Enhanced FP	Enhanced FP + TMR
E	95.78900%	96.98900%	99.87675%
F	98.90000%	98.90000%	99.96397%
H	97.79000%	98.89000%	99.96331%

the three replicas are simultaneously in a failure state $(10^{-2})^3$. We can see that the failure probability was reduced considerably (it used to be 0.01%).

The availability obtained for the LO-criticality outputs using the TMR design pattern are resumed in Table 6.2. We present the availability values for all LO-criticality outputs considering the normal discard MC approach (column "Discard MC"), the enhanced model with fault propagation (column "Enhanced FP") and the enhanced fault propagation model with TMR (column "Enhanced FP + TMR"). These results clearly show the improvements in availability that we can obtain using this type of fault tolerance mechanisms: for example, task *E* which has the lower criticality, increased its availability by over 1% with our enhanced fault propagation model and by over 4% with the TMR. Nevertheless, using a TMR might not always be possible due to budget, space, or power constraints. For this reason we look into another method of fault tolerance in order to improve the availability of LO-criticality outputs.

Weakly hard real-time task

Tolerating a given number of deadline misses on hard real-time tasks was presented in [102]. On monitoring systems or automatic control systems for example, sampling periods need to be set. These periods tend to be set as fast as possible in order to detect changes that the system might need to handle. Nonetheless, for safety reasons periods tend to be faster than what is actually required. Therefore, missing one or more measurements within a fixed number of executions of the sampler can actually be tolerated, as long as the action from monitoring can be done within the delay caused by the missing measurements. In other words, provided that the effect on such a delay does not have severe consequences, deadlines can be missed.

We represent the behavior of a Weakly Hard Real-time Task (WHRT) in Fig. 6.7. The task represented in a $(1-2)$-firm task which means that out-of two successive executions, we tolerate one deadline miss. The state machine representing its behavior is composed of four states: white states in the figure indicate that the task is *functional*, the red state represents that the task is in a failure state. Each state is annotated with a bitset, this bitset

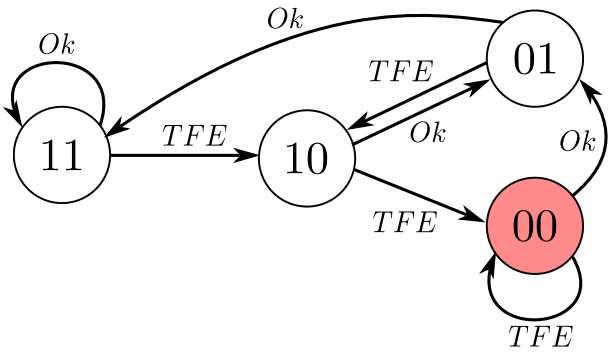
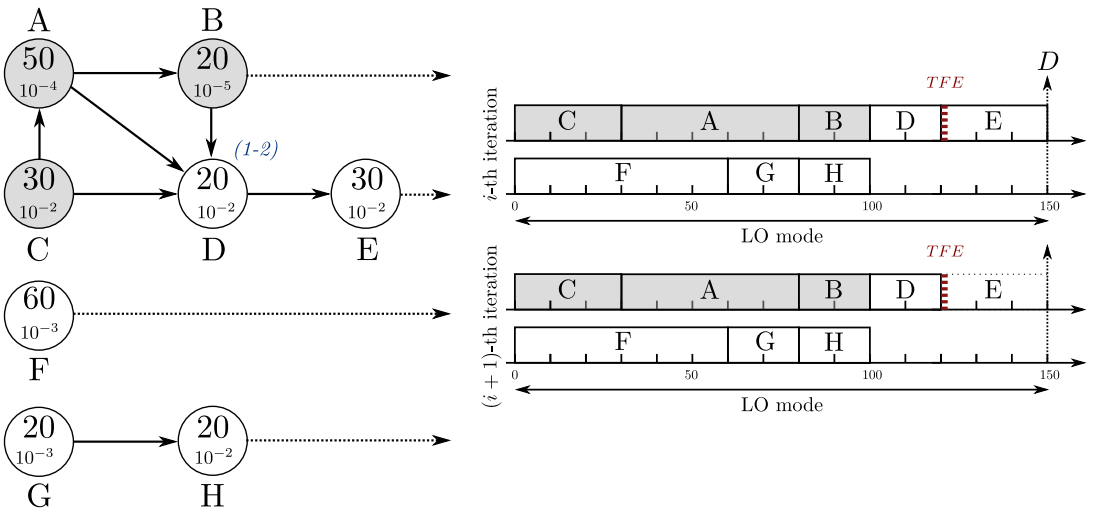


Figure 6.7: $(1-2)$-firm task state machine



(a) MC system

(b) Scenario with TFEs on LO and HI-criticality tasks

Figure 6.8: $(m-k)$-firm task execution example

records the execution history for the number of successive executions we are interested in: 1's are recorded when the task completed its execution within its $C_i(LO)$ and 0's are recorded when a TFE occurred. Arrows between states are labeled with "TFE" or "Ok" indicating what state is reached after the task has a TFE or when the task completed its execution with the timing budget given.

Let us now consider a WHRT in the system we have presented throughout this chapter. In Fig. 6.8 we have represented the WHRT with a blue label: task D is now considered to be a $(1-2)$-firm task, *i.e.* one fault is tolerated out of two successive executions (Fig. 6.8a). The time diagrams represented in Fig. 6.8b show how the task is capable of maintaining its execution on iteration i even after a TFE occurred. Nevertheless, on iteration $i+1$, the second successive TFE provokes the interruption of the successors of the task, in this case task E. Due to the fact that the functional (or non-functional) state of the task is now dependent on a series of executions, the availability formula (Eq. 6.3) can no longer be used to

estimate the availability rate: the failure probability of a task is more complex to estimate. In order to solve this problem we propose to transform the system and its scheduling to perform system simulations. These simulations are performed thanks to the PRISM [114] model checker since it allows us to incorporate all the aspects presented throughout the chapter: failure model, recovery mechanism and the fault propagation model.

6.3.4 Translation rules to PRISM automaton

The incorporation of WHRT tasks changes the execution behavior of the MC system quite significantly, since the failure probability of a task is now dependent on a sequence of executions. In order to estimate the availability rate when this type of task is deployed in the system *we need to perform system simulations*. This type of analysis is completely new to the domain of MC systems with data-dependent tasks. We need to choose an adapted tool capable of modeling all the aspects that we have presented in this chapter until now. The challenges of this procedure are the following: **(i)** there has to be a way to keep track of tasks' executions in order to estimate their availability rate, **(ii)** failure probabilities need to be considered, **(iii)** the fault propagation model we propose capable of limiting the number of mode transitions also needs to be taken into account, and **(iv)** a record of sequential executions of the system needs to be stored.

The tool that allows us to perform system simulations considering all the abovementioned aspects is the PRISM model checker [114]. We introduce translations rules that take the MC system specification and the scheduling tables as an input to produce *probabilistic automaton*. This probabilistic automaton represents the state of the multi-core processors, the criticality mode the system is in and the tasks that are being executed on it. The translation rules have been defined for the generalized case where systems can have more than two criticality levels: when more than two criticality levels are considered *the number of scenarios that lead to the execution or interruption of a task increases significantly* which also motivates the introduction of translation rules. The availability of a task is therefore a measurement when we consider the automaton we produce: by simulating the execution of the system for a long period of time, we estimate what the availability rate is.

The implementation under PRISM uses the following aspects offered by the tool:

- States in the automaton allow us to represent the state of the processor, *i.e.* what tasks are running in the architecture at a certain instant. By doing so we can represent the system running in any χ_ℓ-criticality mode.

- Probabilistic automata allow us to encapsulate our failure model: probabilistic transitions are used to represent if the task did complete its execution within its $C_i(\chi_\ell)$ or if it had a TFE.
- The fault propagation model is captured by the use of boolean variables in the automaton. If the output of a task was produced, the boolean is marked as true, otherwise it is marked as false. To detect if an output is available, we check if all the predecessors of the output produced data, *i.e.* we verify that the conjunction of booleans is true.
- The behavior of the WHRT is simulated thanks to a bit set. This bit set will be updated according to the task's behavior: a 1 is added at the end of the bit set if data was produced, otherwise a 0 is added. Once the bit set is updated, we check the number of bits marked with 1 to deduce if the task is a failure state or not.
- The fault recovery process can be easily represented thanks to a transition that goes to the first state representing the beginning of the lowest criticality scheduling table.
- Transitions can update counters used to measure the number of times a tasks is executed and the number of times the system is executed. The formula of Eq. 6.1 is then applied to obtain the availability rates of outputs.

General principle: The idea behind the translation of scheduling tables to a probabilistic automaton (PA) is to represent the state of the processor, *i.e.* which task is running and in which mode. Since we are executing MC-DAGs in multi-core architectures, we could transform the scheduling table of each core into an automaton. However, having parallel automata in PRISM increases significantly the simulation time due to the large number of possible states. To limit the number of states on the PA and because **TFEs can only be detected at the the end of the timing budget given to a task**, *we only include states that represent the end of the timing budget given to a task for a given criticality mode*. This means that if a task job is preempted several times during its execution, in the automaton we will only have one state representing the end of its timing budget. Since the completion of all tasks need to be checked, doing it in a sequential manner does not change the final result for the availability rates but significantly reduces the simulation time.

Translation rules: To check the completion of tasks, we *sort them by considering their end of time window in the scheduling tables*. The first step is to create ordered list of tasks for each criticality level. For each task τ_i in this list (where i is the index of this task in the list) we create a state $S_i^{\chi_\ell}$ and a boolean variable b_i. The state represents the end of the timing budget given to a task job for its execution at that time instant, and

the boolean is used to notify the successors that output was produced or not. Tasks that are executed in more than one criticality level have more than one state in the automaton: each state represents the execution of the task in the criticality modes it is executed in. Due to parallelism obtainable thanks to multi-core processors, tasks can have the same end of time window. This has to be taken into consideration for the availability computation since mode transitions can be triggered by tasks that have a higher criticality level than the level the system is executed in; but outputs can be available at the same time slot. Ties in the list need to be broken with the following rules:

1. Tasks that are exit nodes in the mode considered go first: we need to know if their output is available.

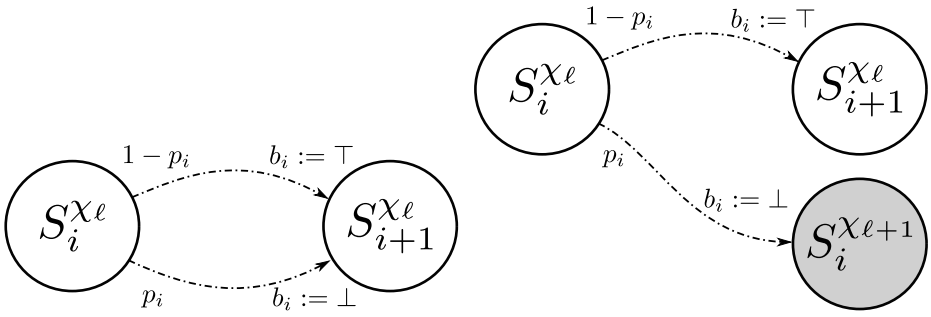
2. Tasks with a higher criticality level than the mode considered go afterwards, a mode transition occurs when they have a TFE.

3. Other tasks are checked at the end, no mode transition occurs even if they fail. In reality, some tasks could have been dropped because one (or more) of their predecessors failed to complete their execution. While the state is always presented in the automata, detecting faulty predecessors takes place in the transition that is taken to leave the state.

After generating the states representing the execution of tasks in all criticality modes, we connect them with the rules illustrated in Fig. 6.9. How tasks are connected is essential for the PA, the rules presented avoid deadlocks, are coherent with the execution of the system and represent the potential TFEs that can occur during the system's execution.

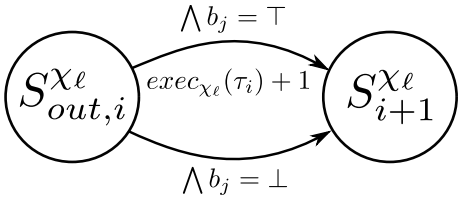
1. From a state $S_i^{\chi_\ell}$ when the task TFE cannot cause a mode transition to higher criticality mode: we add two probabilistic transitions connecting $S_i^{\chi_\ell}$ to $S_{i+1}^{\chi_\ell}$. One of the probabilistic transition represents the occurrence of a TFE (with a probability p_i) and updates b_i to false, whereas the other (with a probability $1-p_i$) updates b_i to true. This rule is illustrated in Fig. 6.9a, the dotted/dashed lines represent the probabilistic transitions, the left label is the probability and the right label is the assignment of the boolean b_i.

2. From a state $S_i^{\chi_\ell}$ when the criticality mode considered is *not* the highest criticality mode the task it is executed in: we add a probabilistic transition with a probability $1-p_i$ to the next state $S_{i+1}^{\chi_\ell}$, boolean b_i is marked as true. A second probabilistic transition with a probability p_i is added to a state $S_i^{\chi_{\ell+1}}$: the system performs a mode transition to the higher criticality mode. This translation rule is represented in Fig. 6.9b.

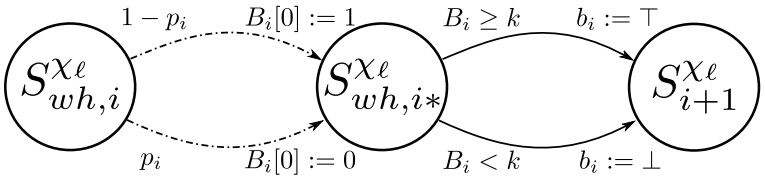


(a) Same criticality level rule

(b) Higher criticality level rule



(c) Output rule



(d) WHRT rule

Figure 6.9: PRISM translation rules for availability analysis

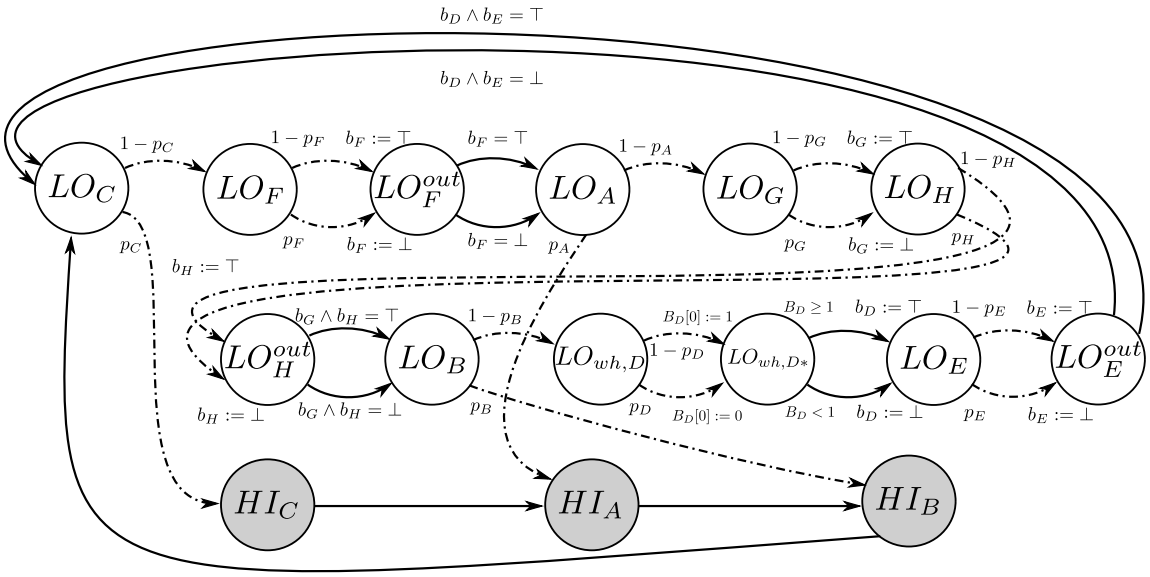


Figure 6.10: PA of the system presented in Fig.6.8

3. The translation rule of Fig. 6.9c is used when a task is also an output for the criticality mode considered. After applying the rule of Fig. 6.9a, we need to add an extra state, named $S^{\chi_\ell}_{out,i}$, that checks if all the predecessors of the task were able to complete their execution ($\bigwedge_{\tau_j \in pred(\tau_i)} b_j$). If that was the case, then the availability of the output is incremented thanks to PRISM counters: each output has its own counter and the system has a counter to check if the system completed its execution in the LO or HI-criticality mode. If the conjunction of booleans is false, the availability counter for the LO-criticality output is not incremented.

4. The final translation rule, illustrated in Fig. 6.9d represents a WHRT task. Like we mentioned before, to capture the behavior of the WHRT task we use a bit set that keeps track of the successive executions of the task. The size of the bit set naturally keeps track of the right amount of successive executions we are interested in. Like it is represented in the figure, the probabilistic transitions are in charge of updating the bit set: if the task did not fail, a 1 is added at the end of the bit set; and if there is a TFE, a 0 is added at the end of the bit set. The oldest bit is erased after during the update. A state, $S^{\chi_\ell}_{wh,i*}$ is added for WHRT tasks in order to check the status of the task, if the number of bits equal to 1 is superior to the number tolerated faults, then the task is in a functional state and its boolean b_i is marked as true. Otherwise the task is considered to be in a failing state and its boolean is marked as false.

Application to the example and results: When we apply the translation rules to the system and the scheduling tables (Fig. 6.7 and Fig. 6.4), we obtain the automaton

Table 6.3: Availability rates for LO-criticality outputs with availability enhancements

	Availability Rate		
Outputs	Discard MC	Enhanced FP	Enhanced FP + WHRT
E	95.78900%	96.98900%	97.90121%
F	98.90000%	98.90000%	98.90099%
H	97.79000%	98.89000%	98.89441%

presented in Fig. 6.10. White states represent the system's execution in the LO-criticality mode, while gray states represent the HI-criticality mode. Like we mention, the first step of the translation, creates the states representing the completion time of tasks for each operational mode. Afterwards, rules to connect each state are applied like we previously explained. Once this automaton is obtained, we can use it in the PRISM model checker in order to estimate the availability rate.

Table 6.3 presents the results obtained for the LO-criticality outputs when we consider a WHRT task. Again we show the gains in availability for all outputs of the system. First, the values obtained with the classic MC discard model are presented. Like for Table 6.2, we present the results obtained when we consider the fault propagation model we described: there is an increase in availability for tasks E and H of 1.2% and 1.1% respectively. The last column presents the results obtained with the PRISM model checker after performing simulations of the system. As we can see, the experimental results are equivalent to the numerical values obtained for tasks F and H. For task E, since its predecessor task D is a $(1-2)$-firm task, we can see another improvement in its availability: we gained almost 1%, demonstrating the interest of including WHRT tasks in the system in order to gain in availability for LO-criticality tasks.

An example of the generalized translation: Fig. 6.11 presents a simplified representation of a probabilistic automaton for a system with N-levels of criticality. This illustration demonstrates the interest of performing translation rules for systems composed of more than two levels of criticality. As an example let us assume we are interested in the availability of task τ_{i_2}. In the automaton we can see that this task is executed in modes χ_0 and χ_1. Thus, in order to evaluate the availability of τ_{i_2}, we need to know how many times this task was executed in both operational modes. The lines illustrated in red represent the different paths leading to the execution of task τ_{i_2} for modes χ_0 and χ_1. As we can see there are four possible combinations that lead to the complete execution of τ_{i_2}. Nonetheless for tasks that are further in the scheduling table and that executed in various criticality levels, the number of possible paths that lead to the complete execution of a task grows rapidly. While enumerating the possible execution paths for the system leading to the execution of

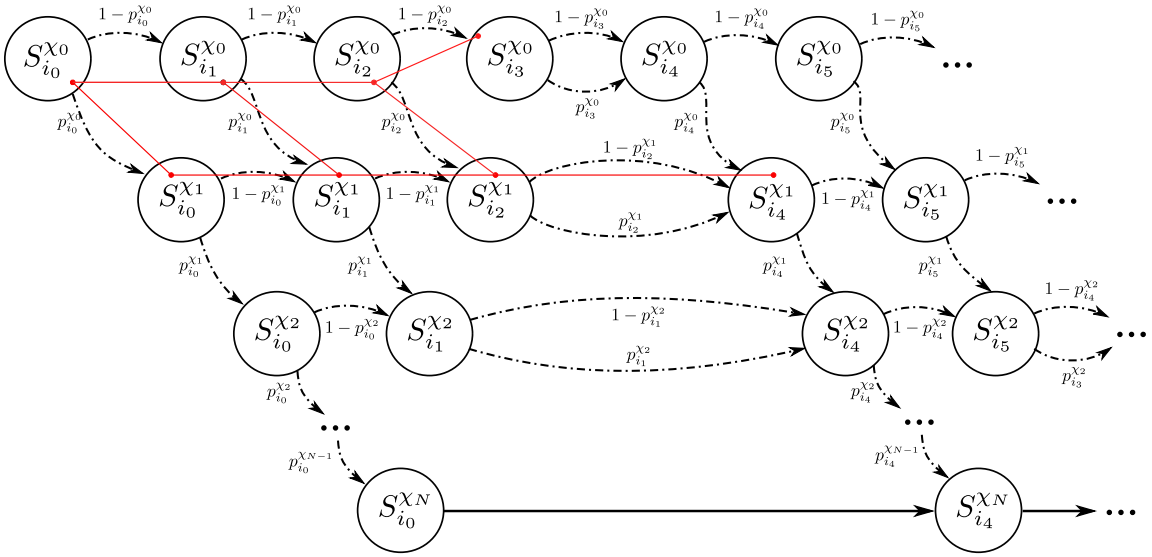


Figure 6.11: Illustration of a generalized probabilistic automaton

a task is possible, it easily becomes error prone. In addition, the translation rules can be combined, so we can have WHRT tasks in addition to an arbitrary number of criticality levels. For this reason translation rules are also interesting when more than two criticality levels are used in the MC system.

In this section we presented our solutions tackling the following problem: **Discard MC model degrades availability (Problem 3)**. We started by introducing a more detailed fault propagation model aiming at **Limiting the number of mode transitions to higher criticality modes (Sub-problem 3.1)** . Thanks to this model, outputs that are unaffected by the TFE can complete their execution. The other type of enhancement we considered, is the fact that safety-critical systems **Incorporate mechanisms for availability enhancement (Sub-problem 3.2)**. While the TMR has shown to greatly improve the availability rate for all LO-criticality tasks, it might not always be possible to deploy replicas in the targeted system. For this reason we have also considered WHRT tasks and proposed translation rules that allow us to perform simulations of the system in order to estimate availability rates.

6.4 Conclusion

In this chapter we presented our availability analysis for MC systems. We suppose MC systems are described following the MC-DAG model presented in Chapter 4 (Section 4.1) and are scheduled with static scheduling tables considered to be MC-correct: for exam-

ple our implementations of the MH-McDAG meta-heuristic (Chapter 5) can be used to schedule these systems.

Thanks to these hypotheses we can **Estimate availability rates (Problem 2)** for LO-criticality outputs. This quantification of availability has not been considered before for MC systems. The scheduling methods we consider is based on the discard MC model [36; 8; 37] of literature: LO-criticality tasks are discarded once a mode transition to the HI-criticality mode has to be performed. We defined a probabilistic **Fault model (Sub-problem 2.1)** in order to estimate how often LO-criticality tasks are discarded and a **Recovery mechanism (Sub-problem 2.2)** to reincorporate the discarded tasks once processing time is available once again. Availability formulas for LO-criticality outputs are then established thanks to the fault model and the recovery process.

While these enrichments to the execution model of MC-DAGs allowed us to calculate the availability rate for LO-criticality outputs, we demonstrated that the **Discard MC model degrades availability (Problem 3)**. This is due to the fact that any TFE occurring in the system provokes a mode transition to the HI-criticality mode, and therefore tasks that are executed last in the scheduling table have the lowest availability rate. We solved this limitation in two ways: first of all, we incorporated a fault propagation model that contains faults in order to **Limit the number of mode transitions to higher criticality modes (Sub-problem 3.1)**. Second, since safety-critical systems often **Incorporate mechanisms for availability enhancement (Sub-problem 3.2)**, we considered them into our analysis. Throughout the chapter we demonstrated the significant improvements in the availability rate of LO-criticality tasks that were obtained thanks to our enhancements. In the example considered, the fault propagation model increased the availability of some tasks by over 1%. When we incorporated fault tolerance mechanisms in addition to the fault propagation model, we increased between 1% and 4% of availability. This changes are quite significant since certification processes measure the availability of tasks up to 10^{-9}.

In this chapter we have also presented translation rules to obtain PRISM automata from MC systems specifications and scheduling tables. These translation rules are applied when the execution model of the system becomes too complex, *i.e.* when enhancements like weakly hard real-time tasks are incorporated in the system, or when the system has more than two criticality levels. The automata produced by our rules simulates the state of the processor during its execution. Availability rates are estimated thanks to execution simulations of this representation.

In the next chapter, we present the MC-DAG framework. This open source tool we developed incorporates the model transformation rules we defined in Section 6.3.3, used

to estimate availability rates thanks to simulations of the system. The framework is also capable of scheduling systems and performing benchmarks in order to assess statically our contribution regarding the scheduling of MC-DAGs in multi-core architectures.

7 Evaluation suite: the MC-DAG framework

TABLE OF CONTENTS

In this chapter, we present the MC-DAG framework, an open-source tool developed to perform the scheduling and availability analyses of MC data-driven systems. This tool consolidates all our contributions and is composed of four different modules:

- A scheduling module capable of applying the G-ALAP-LLF, G-ALAP-HYB and G-ALAP-EDF heuristics to schedule a system (contributions presented in Chapter 5).
- An unbiased and random MC-DAG generator. This module has been developed in order to compare our scheduling methods to existing approaches of the literature. Since different aspects need to be considered in order to obtain an unbiased MC-DAG we had to combine different generation methods of the literature.
- A benchmark suite to measure acceptance rates and number of preemptions for the scheduling heuristic we have developed, but also to compare them to scheduling policies of the state-of-the-art [80; 81].
- An availability module that applies transformation rules to obtain PRISM automata from the system and its scheduling tables (this contribution is presented in Chapter 6).

We begin by presenting the motivation and an overview of the features included in the MC-DAG framework. Details about how we developed the unbiased MC-DAG generation are presented afterwards. The final part of this chapter explains how benchmarks are done in order to compare our contributions to the state-of-the-art.

7.1 Motivation and features overview

When we look at existing approaches capable of scheduling MC-DAGs on multi-core architectures [80; 81; 82; 83], two of them only present theoretical results and the other two have a restricted execution model where all vertices of the MC-DAG belong to the same criticality mode. The scheduling approaches of MC-DAGs [82; 83] have shown experimental results, nevertheless their evaluation framework has not been made publicly available.

The MC-DAG framework we have developed provides the abovementioned feature: scheduling for MC data-driven systems. The framework is hosted on GitHub [2] under the Apache 2 license. It was developed in Java in order to be cross-platform and different JAR files can be created to execute the modules included in the tool.

One JAR module handles the scheduling of MC systems. This module takes a MC system specification as an input. To specify a MC systems a XML file needs to be created following the specification described in the GitHub wiki (https://github.com/robertoxmed/MC-DAG/wiki/Declaring-a-XML-MxC-System). The MC-DAG specification includes the vertices, edges, timing budgets in all criticality modes, number of available cores and number of criticality levels. If the debugging flag is turned on, then all the steps of the scheduling table computation are displayed. The user can choose between the three heuristics we have implemented (G-ALAP-LLF, G-ALAP-HYB or G-ALAP-EDF) in order to schedule the system. Whenever VERIFYCONSTRAINTS of Alg. 1 or Alg. 2 returns false, the programs stops and returns an exception since scheduling tables could not be computed. When the scheduling of the MC system in all criticality modes succeeds, the scheduling tables are written in a XML output file and are displayed if debugging is on.

To statistically assess our contribution, we need to perform benchmarks. Nonetheless, publicly available tools usually only deal with one of the aspects of our research: for example we can find random DAG generators (*e.g.* TGFF [124], GGEN [116]) but the execution time allocated to vertices in the graphs is not controlled like for task sets when real-time scheduling policies are evaluated. On the other hand, task set generators are not

[2] MC-DAG Framework - https://www.github.com/robertoxmed/MC-DAG

capable of creating MC-DAGs. For these reasons one of our objectives was to integrate a generator of MC-DAGs that would be publicly available and reusable by the community. The MC-DAG generator and the benchmark suite are also contained in the framework. Many parameters need to be set for the MC-DAG generator, in the next section we explain how we achieve an unbiased generation. The program can generate an important number of random systems with the parameters chosen by the end-user. Generated systems are written into XML files so they can be read by the scheduling module if the user desires. To generate a graphical representation of the system that was generated, a `.dot` file is written so it can be transformed into post script using Graphviz tools[3]. With these generated systems, the benchmark suite is capable of performing tests to evaluate acceptance rates and measure the number of preemptions generated by different scheduling policies. Since these tests can take a large amount of time, the suite is fully automated for the generation and the performance evaluation.

Additionally, when we look into availability or quality-of-service aspects related to MC systems the same problem also exists: there are not publicly available tools in order to evaluate the availability of non-critical tasks of MC systems. For the availability analysis, the module takes the MC system specification as an input file as well. This JAR file can use scheduling tables given as an additional input or it can calculate the scheduling tables on the fly thanks to our heuristics. With these two inputs, the transformation rules defined in Chapter 6 are applied and the automata are build. Two output files are created: one contains the probabilistic automata (a `.pm` file) and the other contains formulas to measure the availability rates of the non-critical tasks (a `.pctl` file). With these two files the PRISM model checker can be run from the command line or by using the GUI to obtain the availability rates for the non-critical tasks.

7.2 Unbiased MC-DAG generation

In this section we describe how we perform an unbiased generation for systems composed of MC-DAGs. We developed this module of the framework in order to perform statistical analyses. In particular, we planned to compare our scheduling methods to existing approaches of the state-of-the-art [80; 81]. Like we mentioned before, a tool capable of generating DAGs with MC tasks was not available publicly. In order to overcome this limitation we incorporated different aspects related to the generation of unbiased DAGs and the generation of task sets to evaluate real-time scheduling policies.

[3] Graphviz layout programs - http://www.graphviz.org

First of all, when it comes to DAG generation, in [116] Cordeiro *et al.* describe various algorithms capable of creating DAGs with random topologies. We decided to extend the **`Layer-by-Layer`** [125] method due to its simplicity and effectiveness. The idea behind this procedure is to progressively create vertices by assigning them a "layer" level. For each layer level a given number of vertices is created. The number of vertices and layers is parameterized by the user. As layers are created, edges are formed between vertices following a probability rate, *i.e.* there is a probability e of having an edge between two vertices of different layers. To avoid cycles during the formation of edges, an edge can only be added if the source vertex has a lower layer level than the destination vertex.

The second part of the generation, is related to the real-time aspect of our heuristics. We want to tests the limits of our scheduling method by having control over the *utilization rate* of the system. Scheduling policies are often judged based on how they can perform when a system has a high utilization rate: high utilization rates translate to a more difficult scheduling problem. As a matter of fact, many contributions have been proposed in the literature to perform evaluations of real-time scheduling policies. Nonetheless, UUNIFAST [126] has proven to be the most used generation method due to the fact that an uniform distribution of utilization rates can be obtain easily and quickly. Therefore in order to distribute an utilization rate among tasks we use UUNIFAST and its multi-core adaptation UUNIFASTDISCARD [5]. UUNIFAST is used to distribute an overall utilization rate among a given number of MC-DAGs included in the system. Since the utilization rate of a MC-DAG can be greater than 1, we do not need to filter utilization rate sets like UUNIFASTDISCARD does. Nevertheless, when we distribute the utilization rate among vertices we use UUNIFASTDISCARD, we cannot have a vertex with a utilization greater than 1 since it would implicate that more than one processor is needed to scheduled a sequential task.

Generation of a MC system

Algorithm 3 presents the random system generation of our tool. Seven parameters need to be given by the user:

- $U_{max} \in \mathbb{R}_+^*$ the maximum utilization of the system in all the criticality modes.
- $n_{\mathcal{G}} \in \mathbb{N}$ the number of MC-DAGs that will be included in the system.
- $\{\chi_1, \ldots, \chi_N\}$ the set of criticality levels.
- n_V the number of vertices that need to be create for each MC-DAG.

Algorithm 3 Unbiased System generation

```
1:  function GENERATESYSTEM(U_max: utilization of the system in all modes,
        n_G: number of DAGs to create,
        {χ_1, ..., χ_N}: set of criticality levels,
        n_V: number of vertices to create,
        e: probability of having an edge,
        p: vertices/layer lower-bound,
        r: reduction factor)
2:      S ← new MC system
3:      |S.Π| ← ⌈U_max⌉                          ▷ Number of available cores
4:      S.χ ← {χ_1, ..., χ_N}                    ▷ Set of criticality levels
5:      U_Gset[] ← new set of utilizations for DAGs
6:      UUNIFAST(U_Gset, U_max)                  ▷ Distribute utilization rates
7:      for all i < n_G do
8:          S.G ← S.G ∪ GENERATEGRAPH(U_Gset[i], S.χ, n_V, e, p, r)
9:      end for
10:     return S
11: end function
```

- e the probability of having an edge between two vertices.
- p an lower-bound on the number of vertices that belong to a layer.
- r a reduction factor for the utilization of tasks executed in more than one criticality mode.

After the system is instantiated and initialized with the architecture and the criticality modes (l. 2-4), an array of utilization rates is created (l. 5). This array $U_{\mathcal{G}set}$ of size $n_{\mathcal{G}}$, is used by UUNIFAST to distribute the utilization rate of the system to each MC-DAG (l. 6). The idea behind UUNIFAST is to distribute the utilization U_{max} among the slots of $U_{\mathcal{G}set}$ uniformly. Once the distribution is performed, we proceed to create the number of MC-DAGs requested by the user (l. 7-8).

Generation of a MC-DAG

The function GENERATEGRAPH is detailed in Algorithm 4. The **`Layer-by-layer`** procedure described in [125; 116] is used as a basis for the algorithm. The timing budget distribution and assignment to each vertex of the MC-DAG is done before edges are created.

The parameters used by the function are almost identical to parameters used by GENERATESYSTEM. The only difference is the utilization rate that will be used by the vertices of the graph: $U_{Gset} \in \mathbb{R}_+^*$.

Algorithm 4 Unbiased MC-DAG generation

function GENERATEGRAPH(U_G: DAG utilization,
χ: criticality levels,
n_V: number of tasks,
e: edge probability,
p: vertices/layer upper-bound,
r: reduction factor)

$G \leftarrow$ **new** MC-DAG ▷ Initialization block
$D \leftarrow$ random deadline from pre-defined list
$budget[] \leftarrow \lfloor D \times U_G \rfloor$
$prevLayer \leftarrow 0$
for all $\chi_\ell \in \chi$ in decreasing order **do** ▷ Generation
  $layer \leftarrow \text{rand}(0, prevLayer)$
  $U_{set}[] \leftarrow$ array of utilization rates for each task

  repeat ▷ Generate the utilization for each task
    $return_{UUD} \leftarrow$ UUNIFASTDISCARD$(U_{set}, budget[\chi_\ell]/D)$
  until $return_{UUD} = \top$

  $tasksToGen \leftarrow n$
  while $budget[\chi_\ell] > 0 \ \wedge \ tasksToGen > 0$ **do** ▷ Vertices generation phase
    $count \leftarrow 0$
    while $count \leq p \ \wedge \ tasksToGen > 0$ **do**
      $\tau_i \leftarrow$ **new** vertex
      $C_i(\chi^\ell) \leftarrow \lceil D \times U_{set}[i] \rceil$
      $budget[\chi_\ell] \leftarrow budget[\chi_\ell] - C_i(\chi^\ell)$
      $\tau_i.layer \leftarrow layer$

      if $layer \neq 0$ **then** ▷ Edges incorporation phase
        for all $\{\tau_j \in G \mid \tau_j.layer > \tau_i.layer\}$ **do**
          if $\text{rand}(0,1) \leq e \ \wedge \ CP_j^{\chi_\ell} + C_i(\chi_\ell) \leq D$ **then**
            $G.E \leftarrow G.E \cup (\tau_j, \tau_i)$
          end if
        end for
      end if

      $G.V \leftarrow G.V \cup \{\tau_i\}$
      $tasksToGen \leftarrow tasksToGen - 1$
      $count \leftarrow count + 1$
    end while
    $layer \leftarrow layer + 1$
  end while

```
33:         if χ_ℓ ≥ χ_2 then                                  ▷ Reduction phase
34:             U_reduc ← U_G/r                                ▷ Reduce by the reduction factor
35:             budget_reduc ← U_reduc × D
36:             repeat
37:                 for all τ_i ∈ G do
38:                     C_i(χ_{ℓ−1}) ← rand(1, C_i(χ_ℓ))
39:                 end for
40:             until budget_reduc < Σ_{τ_i∈G} C_i(χ_{ℓ−1})
41:             budget[χ_{ℓ−1}] ← budget[χ_{ℓ−1}] − budget_reduc
42:         end if
43:         prevLayer ← layer
44:     end for

45:     return G
46: end function
```

Initialization phase (l. 2-5): The function starts by creating a MC-DAG and assigning it a deadline (l. 3). This deadline is randomly selected from a pre-defined list: {100, 120, 150, 180, 200, 220, 250, 300, 400, 500}. Since industrial implementations often avoid large hyper-periods caused by prime numbers between deadlines of applications, we follow the same principle with this list of possible deadlines. With the utilization and the deadline of the MC-DAG, we can deduce the timing budgets that needs to be distributed among tasks in all criticality levels (l. 4). Budgets are stored in an array of size $|\chi|$, the cardinality of the criticality levels. The timing budget is used for all tasks that are executed in a given criticality level: this includes tasks that are executed in more than one criticality level. For example, let us consider a system of three criticality modes composed of 30 vertices/tasks: in the highest criticality mode, the timing budget is distributed among 10 tasks. Once we move to the second highest criticality level, the timing budget is divided among 20 tasks, the 10 tasks that were created in the previous criticality mode (with a reduced timing budget since $C_i(\chi_\ell) \leq C_i(\chi_{\ell+1})$), in addition to the 10 new tasks that need to be added. The final level will distribute the timing budget between 30 tasks.

Generation initialization (l.7-11): The generation phase of Alg. 4 (l. 6-44) then starts by looping over the criticality levels in descending order. A layer value is randomly selected (l. 7): in the highest criticality level, we are forced to start with layer number 0, but for subsequent criticality levels, we can start in a deeper layer. This prevents biases that could be caused by having too many source vertices and MC-DAGs with very few consecutive tasks. The next step consists in creating an array of utilization ranges. This array U_{set} of size n, is used to store the utilization given to each task created in the χ_ℓ level. In order to do so, we use the adaptation of UUniFast for multi-core architectures:

UUNIFASTDISCARD [5]. The idea behind this method is to use UUNIFAST but discard a set of utilization rates where one (or more) value is greater than one. This needs to be avoided since it would mean that a *sequential* task would need more than 1 CPU in order to complete its execution, which is impossible. Since UUNIFASTDISCARD can take several tries to create an acceptable utilization array, the procedure is called as many time as it is required (l. 9-11). (For Alg. 3, regular UUNIFAST is used since a MC-DAG can have a utilization greater than one).

Vertex generation phase (l.12-32): With the utilization rates for each task being created, the vertex generation phase starts. We need to create n vertices per level, and up to p vertices per layer. After the vertex is created (l. 16), we transform the utilization rate obtained thanks to UUNIFASTDISCARD to timing budget $C_i(\chi_\ell)$ (l. 17), the overall budget for the criticality level is then updated (l. 18) and the current layer level is given to the task (l. 19).

Incorporation of edges (l.20-26): If the current layer is not the 0-layer, then we try to add edges between the newly created vertex and all the vertices that have a lower layer level (l. 20-26). By only having edges that go from a low-level layer to a high-level layer we prevent the creation of cycles in the graph: a cycle would imply that an edge goes from a high layer communicates with a low layer vertex at some point. We do follow the probability e to have an edge but also verify that adding an edge would not create a critical path bigger than the deadline assigned to the MC-DAG: $CP_j^{\chi_\ell} + C_i(\chi_\ell) \leq D$ (l. 22).

Once the edges are created, we add this new vertex to the graph (l. 27), update the number of tasks to create (l. 28) and the number of tasks created in the layer (l. 29). This procedure is repeated until the timing budget has been distributed.

Reduction phase (l.33-42): Once all tasks for the current criticality level have been created, we check if we are not in the lowest-criticality level (l. 33). If that is not the case, the reduction phase takes place. This reduction phase creates timing budgets for tasks executed in the χ_ℓ criticality mode for the $\chi_{\ell-1}$ mode. We start by dividing the level utilization, $U_\mathcal{G}$, by the reduction factor r (l. 34) and we obtain the targeted timing budget for all tasks afterwards (l. 35). The process to reduce timing budgets for the $\chi_{\ell-1}$-criticality mode then starts: we randomly generate the $C_i(\chi_{\ell-1})$ timing budget to avoid an homogeneous reduction that could also introduce biases for the scheduling (l. 38). The only restriction is that timing budgets need to be monotonically decreasing in function of the criticality-level: $C_i(\chi_\ell) \leq C_i(\chi_{\ell+1})$. This reduction is applied to all tasks until the targeted budget is reached (l. 36-40). The *budget* array is also updated (l. 41) for tasks that will be exclusive to the $\chi_{\ell-1}$-criticality mode.

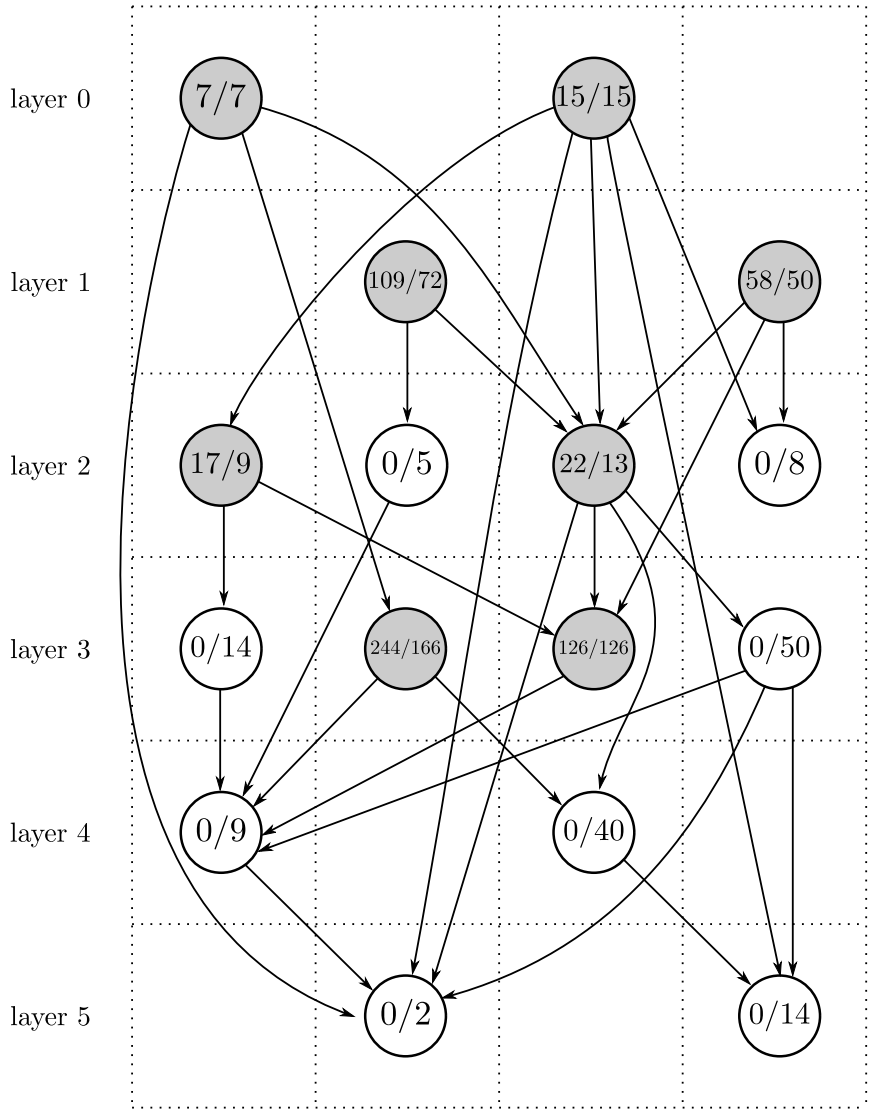


Figure 7.1: Example of a dual-criticality randomly generated MC-DAG

After the generator has iterated through all levels and through all layers, the generated MC-DAG is returned (l. 45) and is added into the system (Alg. 3 l. 8).

An example of the random generation

Like for the scheduling module, the random generator is capable of generating multiple systems in parallel. This feature is very convenient for our benchmarking suite. An example of generated system is illustrated in Fig. 7.1. The generator was set to use the following parameters: $n_{\mathcal{G}} = 1$, $U_G = 2$, $\chi = \{HI, LO\}$, $n = 16$, $e = 20\%$, $p = 2$, $r = 2$. Vertices are annotated with their IDs and with their timing budgets: HI-criticality tasks have two non-

null timing budgets, while LO-criticality tasks have one non-null timing budget. Edges represent the data-dependencies among vertices.

Thanks to Alg. 3 and Alg. 4 we can generate a large number of unbiased and random MC-DAGs in order to perform benchmarks of our scheduling algorithm. This generation is generalized and MC systems with more than two criticality levels can be created. In the next section, we present the benchmark suite we have developed, as well as the metrics that can be obtained thanks to this module of the framework.

7.3 Benchmark suite

Like we explained at the beginning of this chapter, most contributions that tackle the problem of MC-DAG scheduling have only presented theoretical work [80; 79; 81] or have not published open tools [82; 83] that could be used by the community. To overcome this limitation and in order to evaluate our contributions by comparing them to the state-of-the-art, our MC-DAG framework includes a benchmark suite.

The suite is a combination of a Python script and two Java JAR files. One JAR file contains the generation tool we described in the previous section. The other JAR file performs schedulability tests and measures the number of preemptions for G-ALAP-LLF, G-ALAP-HYB, G-ALAP-EDF and FEDMCDAG from [80; 81]. The script was created in order to perform and automatize a large number of tests.

In the script, the user needs to set the following parameters: number of tasks per MC-DAG (n_V), number of MC-DAGs ($n_\mathcal{G}$), number of cores, the probabilities of having an edge between two vertices (e), the number of threads that are used by the benchmark suite and the number of files to create for each combination of parameters. All combinations of these parameters are used by the script in order to generate unbiased random MC-DAGs thanks to Alg. 3 and Alg. 4; and to perform benchmarks on these generated systems. Extensive tests are needed in order to cover a large space of possible MC system configurations. For example in Chapter 8 where we compare our results to existing approaches, we tested 500 systems for a single point of some graphs and tests took about 8hrs for over 350,000 files.

The benchmark module (the second JAR file of the benchmark suite) takes an arbitrary number of input files (XML MC system specifications) to perform tests, the user can choose which scheduling algorithms are used by the tool and needs to specify an output path to write detailed results in addition to an output path were the summarized results are written. Detailed and summarized results are written in the form of CSV files in order to plot curves afterwards.

7.4 Conclusion

The MC-DAG framework, an open-source tool developed in order to have a platform capable of analyzing MC systems composed of MC-DAGs was presented in this chapter. Since existing works scheduling MC-DAGs have not released publicly available tools and since availability analyses for MC-DAGs has never been performed before, we developed this tool so the community can use it and extend it at will.

This framework is decomposed in different independent modules. The scheduling module contains our scheduling heuristics (contribution detailed in Chapter 5). An unbiased random MC-DAG generator is also included in the framework. Finally, transformation rules to obtain probabilistic automata from MC systems are included on another module (contribution presented in Chapter 6).

While two modules are simply the implementations of our contributions (Chapter 5 and 6), in this chapter we have presented another key contribution included in our framework: the unbiased MC-DAG generator. By combining different methods of the literature regarding DAG generation (the **`Layer-by-Layer`** method [116; 125]) and task set utilization distribution (UUNIFAST [126] and UUNIFASTDISCARD [5]), we have developed a random MC-DAG generator. Thanks to this generator we can evaluate the performances of our scheduling methods and compare them to existing methods of the state-of-the-art.

The benchmark suite included in this framework was used for the experiments presented in the following chapter. We give detailed results of our scheduling heuristics regarding acceptance rates and number of preemptions for randomly generated systems.

8 Experimental validation

Table of Contents

This chapter presents our experimental validation for the scheduling contributions presented in this dissertation. We begin by describing our experimental setup. Many parameters have to be considered for the generation of MC-DAGs so we describe the rationale behind the setup that we use. The second section of this chapter aims at demonstrating how our scheduling approaches for a dual-criticality system outperform the state-of-the-art [80; 81]. We measure the acceptance rate and the number of preemptions for the G-ALAP implementations of MH-McDAG and for the FedMcDag algorithm of [81]. These two metrics are decisive when judging the *quality* of real-time scheduling policies. Finally, we present experiments of our scheduling approach when we consider MC systems composed of more than two levels of criticality. To best of our knowledge this is the first statistical analysis performed on MC data-driven systems executing MC-DAGs where vertices can have different criticality levels. The influence that the number of criticality levels have on the acceptance rate is demonstrated.

8.1 Experimental setup

In the previous chapter we presented our contribution regarding the generation of unbiased MC-DAGs. In order to statistically assess the performance of our scheduling algorithms we need to see the *influence of generation parameters* on the performance criteria we use:

acceptance rate and number of preemptions. The following list gives the parameters that we varied in order to perform the assessment of our scheduling approaches:

- **Utilization of the system**: the main parameter used in order to assess the quality of real-time schedulers is the utilization of the system. Having a higher utilization makes the scheduling problem more difficult as the processor is more demanded. Therefore, a scheduling approach is efficient if it continues to find feasible schedules as utilization increases.

- **Probability of having an edge between two vertices**: the *density* of graphs we try to schedule has a big influence on the scheduling problem. Increasing the number of data dependencies makes the scheduling problem more difficult. For our experimentations we chose to set this parameter to 20% or 40%. Having denser graphs is not a problem for the generator of MC-DAGs but does not represent the normal amount of precedence constraints in industrial safety-critical systems.

- **Number of tasks per DAG**: each MC-DAG is fragmented into a set of tasks and the size of this set has an influence on the scheduling problem. The overall timing budget that needs to be distributed among tasks is fixed once the utilization of the system and the deadline of the graph have been set. Having a small number of tasks implies that each task has a large timing budget C_i. The scheduling problem tends to become easier when the number of tasks increases given a set utilization rate and deadline as timing budgets become smaller.

- **Number of DAGs**: having various MC-DAGs in the system implies that tasks have different periods and need to be activated more than once during the hyper-period of the system. This has an impact on the scheduling problem since from one activation to the other, the number of jobs that need to be scheduled can be different.

- **Number of cores for a given architecture**: by increasing the number of cores considered in the architecture, the scheduling problem also becomes more difficult. For a given utilization rate, timing budgets that need to be distributed among tasks become greater, subsequently increasing tasks' timing budgets. Again allocating 'larger" tasks is more difficult for the approximation algorithms we study. We consider architectures having four and eight cores for our experiments. Considering architectures with more than eight cores switching to higher criticality mode in a synchronous manner is unrealistic [127]. There is latency in messages provoked by the network-on-chip.

- **Number of criticality levels**: intuitively by increasing the number of levels the scheduling problem becomes more complex. More tables need to be computed and have to be schedulable. At the same time safe transitions need to be ensured between more criticality modes.

The generation of MC-DAGs is a combination of all these parameters and in order to cover as much as possible all configurations, we need to generate an important number of test files. The influence of other generation parameters was also studied but the variations on acceptance rates and number of preemptions were not relevant enough to present them in this manuscript. The next section evaluates the behavior and performances of our scheduling approaches compared to existing works of the literature [80; 81] for dual-criticality systems.

8.2 Acceptance rates on dual-criticality systems

The first experiments we present are centered around dual-criticality systems. Like we have mentioned before, existing approaches capable of scheduling MC-DAGs have only been developed for dual-criticality systems. In our experimentations we want to evaluate the performance gain of our scheduling approach compared to schedulers defined in [80; 81]. This evaluation remains experimental and we do not provide a theoretical proof of dominance over the existing approaches of the literature. Nonetheless, we explain the key aspects that differentiate our implementations of MH-McDAG from the ones proposed in [80; 81].

Our study is divided in two parts: we begin by performing an analysis on MC systems composed of a single DAG, to then evaluate performances on multiple MC-DAG systems. For all experimentations we have instantiated the G-ALAP-LLF, G-ALAP-EDF, G-ALAP-HYB version of our meta-heuristic and GENSCHEDMULTIPROC [80] / FEDMCDAG [81] to compare results.

8.2.1 Single MC-DAG scheduling

In our first experiments, we compare our scheduling heuristic to the approach presented in [80; 81] for a system composed of a single MC-DAG with a utilization superior to one: this implies that more than one processor is needed to schedule the MC-DAG. Like we explain in the Chapter 5, the approach presented in [80; 81] uses LS to find a priority ordering for HI and LO-criticality tasks. The two priority orderings are independent. HI-criticality tasks' allocation is systematically prioritized in order to obtain MC-correct schedules, *i.e.*

when a HI-criticality task becomes active it will be executed ASAP potentially preempting any LO-criticality task that was running in the architecture. On the other hand, we have proposed to schedule HI-criticality tasks ALAP and only preempt LO-criticality tasks in order to respect **Safe Trans. Prop.** (see Definition 12 in Chapter 5). The idea is to constraint as less as possible the computation of the LO-criticality scheduling table.

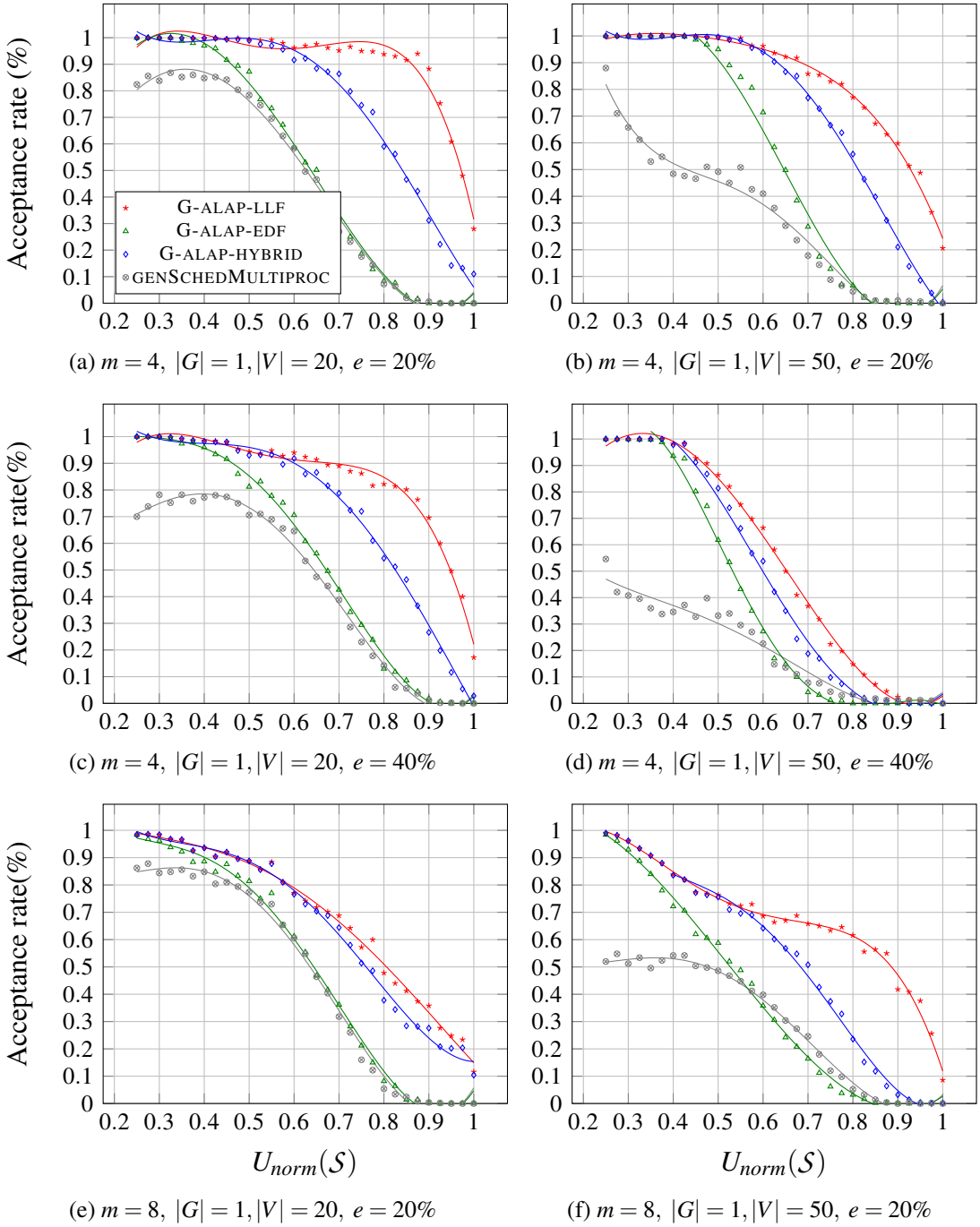
Besides the comparison to the state-of-the-art, we show the impact of some generation parameters like **(i)** the number of tasks per DAG, **(ii)** the probability to have an edge between two vertices and **(ii)** the number of cores available in the architecture. Therefore, the first experiments we present are configured as follows:

- A single MC-DAG is generated for the system.
- The number of tasks progressively changes between 20 and 50.
- The probability to have an edge between two vertices is set to 20% or 40%.
- The number of cores increases from four to eight.
- The reduction factor is set to two: the utilization rate given to HI-criticality task is reduced by half when the timing budgets for the LO-criticality mode are generated. Other reduction factors can be considered but a reducing budgets by half ensures that LO-criticality tasks will have large timing budgets and HI-criticality tasks in LO mode will also have an impact on the scheduling.
- The ratio between HI and LO-criticality tasks is set to 1:1, the same amount of LO and HI-criticality tasks is created for each MC-DAG. Like for the reduction factor, other values could have been considered, but the idea is to have both types of tasks for all the experiments.

Acceptance rate

Fig. 8.1, presents our results in terms of acceptance rate (*i.e.* the percentage of systems that were schedulable) for our scheduling heuristics compared to the heuristic of the state-of-the-art. In the plots that we present, we have represented the measured rates in addition to their polynomial fitting curve for readability reasons.The X-axis represents the normalized utilization while the Y-axis is the acceptance rate. For each point of the graph we generated 500 random MC systems.

For GENSCHEDMULTIPROC of [80], we implemented it using the most efficient LS heuristic to minimize the makespan considering our hypotheses for the MC system: Highest Level First with Estimated Times (HLFET). This heuristic is based on the longest



(a) $m = 4,\ |G| = 1, |V| = 20,\ e = 20\%$

(b) $m = 4,\ |G| = 1, |V| = 50,\ e = 20\%$

(c) $m = 4,\ |G| = 1, |V| = 20,\ e = 40\%$

(d) $m = 4,\ |G| = 1, |V| = 50,\ e = 40\%$

(e) $m = 8,\ |G| = 1, |V| = 20,\ e = 20\%$

(f) $m = 8,\ |G| = 1, |V| = 50,\ e = 20\%$

Figure 8.1: Measured acceptance rate for different single MC-DAG scheduling heuristic

cumulative path, considering timing budgets from an exit vertex. In [72] it was demonstrated that this heuristic is the most efficient when homogeneous architectures and no communication costs are considered. Also it holds the lowest complexity which makes it an efficient algorithm.

In Fig. 8.1a we show the results obtained for a MC system composed of a single MC-DAG with 20 vertices in a four cores architecture. The scheduler that has the best performance is G-ALAP-LLF, followed by G-ALAP-HYB. We recall the principle of G-ALAP-HYB: the HI-criticality mode is scheduled with the EDF priority ordering we defined thanks to Eq.5.7 (see Chapter 5) and in the LO-criticality mode we use the LLF priority ordering (Eq. 5.6 of Chapter 5). For G-ALAP-LLF the acceptance rate is very good since more than 90% of the generated systems are schedulable until the utilization reaches 90%. For G-ALAP-HYB, the acceptance rate degrades faster compared to G-ALAP-LLF. After the system is at 65% of its utilization, the acceptance rate drops below 90%. Nevertheless, results are still good: when the system has a utilization of 80%, almost 60% of systems remain schedulable. When we look into the results obtained by G-ALAP-EDF and GENSCHEDMULTIPROC, their performances are similar to each other, specially after the utilization reaches 60%. These two approaches present the lowest acceptance rate: less than 60% when the utilization of the system is at 60%. G-ALAP-EDF and GENSCHEDMULTIPROC are very similar when the utilization of the system is high, due to the fact that HLFET and EDF define a similar priority ordering (based on critical paths) and HI-criticality tasks become bigger so executing them ALAP or ASAP does not influence on the acceptance rate anymore.

Influence on the number of tasks: In Fig. 8.1b, we present the results obtained by the schedulers when the number of tasks is incremented from 20 to 50. The fact that more tasks are created implicates that potentially more precedence constraints need to be respected. However, these tasks tend to have a smaller execution time, so for certain utilization rates the scheduling problem becomes easier.

Again the best performances are obtained by the G-ALAP-LLF implementation. Yet, when the normalized utilization reaches 70%, the acceptance rate goes below 90% and progressively degrades. The degradation on the acceptance rate is more notorious than when 20 tasks were created for each MC-DAG. Nonetheless, the performance is still good since over 70% of systems were schedulable when the utilization is below 80%.

For the G-ALAP-HYB implementation, once the utilization is over 65%, the acceptance rate degrades faster than in Fig. 8.1a. For example in Fig. 8.1b when $U_{norm}(\mathcal{S}) = 0.8$ the acceptance rate is at 54% as opposed to 60% in Fig. 8.1a However, since the difference between Fig. 8.1a and Fig. 8.1b is around 5%, results are still interesting for this heuristic.

The behavior of G-ALAP-EDF is a bit different: the degradation on the acceptance rate is only visible when the system reaches a utilization of 50%, as opposed to 40% in Fig. 8.1a. There is an improvement in the acceptance rate due to the fact that the utilization is distributed among more tasks. This be seen when the system's utilization is at 0.6: 60% of schedulable systems for Fig. 8.1a whereas in in Fig. 8.1b 70% of systems are schedulable. Once the utilization goes beyond 70% there is a slight degradation in terms of acceptance rate, around 5%, between Fig. 8.1a and Fig. 8.1b.

Finally, the results of GENSCHEDMULTIPROC show that having more tasks greatly affects the performances of the algorithm. For example when $U_{norm}(\mathcal{S}) = 0.4$, the acceptance rate drops from over 80%, to a bit over 50%. In this case, our heuristics clearly outperform the existing approach, until the utilization reaches 70% where G-ALAP-EDF and GENSCHEDMULTIPROC perform almost identically. There are two main reasons that explain this behavior. **(i)** HI-criticality tasks are scheduled until their completion by GENSCHEDMULTIPROC and because we incremented the number of tasks we also incremented the number of HI-criticality tasks. Our approach on the other hand, can preempt HI-criticality if a task with a higher priority becomes active and does not violate **Safe Trans. Prop. (ii)** A second source of optimization comes from the fact that HI-criticality tasks are scheduled ALAP in the HI-criticality mode, which relaxes the execution of LO-criticality tasks. This difference in scheduling becomes negligible after the utilization of the system reaches 70% but this is due to the fact that HI-criticality tasks become bigger and scheduling them ASAP or ALAP does not have an impact anymore.

Influence of the graph's density: In Fig. 8.1c and Fig. 8.1d we evaluated the impact of the probability to have a data dependency between two vertices. This probability increased from 20% to 40%.

Between Fig. 8.1c and Fig. 8.1a the only parameter that changed was the probability to have an edge. The overall behavior of the four scheduling heuristics is the same between the two plots that we mentioned: G-ALAP-LLF shows the best performance, followed by G-ALAP-HYB, and G-ALAP-EDF is comparable to GENSCHEDMULTIPROC. Nonetheless, there is a general deterioration on performances of all heuristics, which was expected since graphs are denser (*i.e.* more dependencies need to be considered).

The results of Fig. 8.1d are quite different from the behavior observed in Fig. 8.1b: for these two plots we set the number of tasks to 50. Because the number of tasks is bigger and due to the increment in data dependencies, the results of our three implementations are "closer", *e.g.* for G-ALAP-LLF and G-ALAP-HYB there is only a 10% difference in the acceptance rate. The gap between G-ALAP-HYB and G-ALAP-EDF is also tinner, we have on average a 20% difference between both curves. This behavior can be explained by

the increment of data dependencies and tasks which makes the scheduling problem quite difficult to solve.

For GENSCHEDMULTIPROC we can also see a general deterioration on its performance due to the numerous precedence constraints that need to be considered to compute the scheduling tables. Nonetheless when the system has a utilization higher than 65%, GENSCHEDMULTIPROC has a better performance than G-ALAP-EDF. This can be explained by the fact that the priority ordering we defined based on G-EDF is too permissive for some tasks that can have the exact same deadline but not the same execution time. Nonetheless, GENSCHEDMULTIPROC is only better than G-ALAP-EDF by a very small margin.

Influence of the targeted architecture: The final experiments presented in Fig. 8.1 demonstrate the effects of the number of cores considered in the targeted architecture. We increased this number from four to eight. The probability to have a data dependency is set to 20% and the number of tasks varies from 20 (Fig. 8.1e) to 50 (Fig. 8.1f).

Because the number of cores has increased, the timing budget that is given to tasks also increases in function of the normalized utilization. As we can see, this has an impact on the performance of all the scheduling heuristics. While G-ALAP-LLF still shows the best performance in this case, its results are not that different from G-ALAP-HYB. The deterioration in its performance is also visible: for a system with a utilization of 80% in four cores, 90% of systems were schedulable (Fig. 8.1a) as opposed to 48% (Fig. 8.1e). Nevertheless, when we look into the results obtained in Fig. 8.1f, G-ALAP-LLF has a better acceptance ratio. Tasks with smaller execution times explain this improvement but is limited when the utilization increases: 60% of systems are schedulable when $U_{norm}(\mathcal{S}) = 0.8$ as opposed to 80% (Fig. 8.1b).

The performance of G-ALAP-HYB also suffers from a general deterioration but less visible than results obtained by G-ALAP-LLF. Considering the case where a MC-DAG has 20 vertices, when the utilization is at 80%, the acceptance rate diminished from 60% (Fig. 8.1a) to almost 40% (Fig. 8.1e). This decrement is also visible when a MC-DAG has 50 vertices: for $U_{norm}(\mathcal{S}) = 0.8$ we have a 25% of acceptance rate (Fig. 8.1f) as opposed to 55% (Fig. 8.1b).

When we look into the results obtained by G-ALAP-EDF, we notice that there is almost no impact on the acceptance rate when the number of cores increases and MC-DAGs have 20 vertices. The curves and values from Fig. 8.1a and Fig. 8.1e are almost the same. However, when the number of cores and the number of tasks increases this is no longer the case. The acceptance rate drops almost in a linear fashion for G-ALAP-EDF when MC-DAGs have 50 vertices and the architecture has eight cores (Fig. 8.1f). A similar linear

degradation is seen when the architecture has only four cores (Fig. 8.1b) but it occurs at a later point in Fig. 8.1f, *i.e.* when the utilization is over 40%.

For GENSCHEDMULTIPROC, when the number of tasks is set to 20 (Fig. 8.1e) the results are similar to what we observed for G-ALAP-EDF: the results are quite similar between four (Fig. 8.1a) and eight (Fig. 8.1e) cores. Nonetheless, when the number of tasks and cores increases (Fig. 8.2c) there is a more significant difference specially when the utilization of the system is low (below 50%). We can also observe the fact that GENSCHEDMULTIPROC outperforms G-ALAP-EDF when the utilization of the system is higher than 60%, this is due to the fact the the G-EDF priority ordering of tasks is more permissive (the same behavior was observed in Fig. 8.1d).

Conclusion on single MC-DAG scheduling

The following conclusions can be established thanks to our experimental results:

- G-ALAP-LLF gives the better performance in terms of acceptance rates which is in accordance with the result presented in [117]: G-EDF is dominated by G-LLF. G-ALAP-HYB has the second best performance followed by G-ALAP-EDF and GENSCHEDMULTIPROC. In some situations GENSCHEDMULTIPROC has better performances than G-ALAP-EDF.

- Increasing the number of tasks tends to make the scheduling more difficult when the utilization of the system is elevated. Conversely when the utilization of the system is low, having more vertices makes the scheduling problem easier: the timing budgets are smaller so allocating them is easier.

- Increasing the probability to have an edge between two vertices is one of the main parameters affecting the scheduling problem of MC-DAGs. When the number of tasks is not very high (20 or less for example), the degradation is not very consequent. However if a lot of tasks need to be scheduled by the system and there is a high probability to have an edge between them, the performances of our scheduling heuristics were clearly affected (*e.g.* there is 60% decrease in the acceptance rate for G-ALAP-LLF when $U_{norm}(\mathcal{S}) = 0.8$).

- Considering a bigger architecture with more cores has an implication on the timing budgets that tasks will have. This has shown to have consequences on the performance of G-ALAP-LLF rending the scheduling problem more difficult for this scheduler. G-ALAP-EDF and GENSCHEDMULTIPROC seem to be less affected by this change when 20 tasks are on the system. Nonetheless, when more cores and

tasks are considered(eight cores and 50 tasks), G-ALAP-EDF and GENSCHEDMULTIPROC see a degradation on their performances.

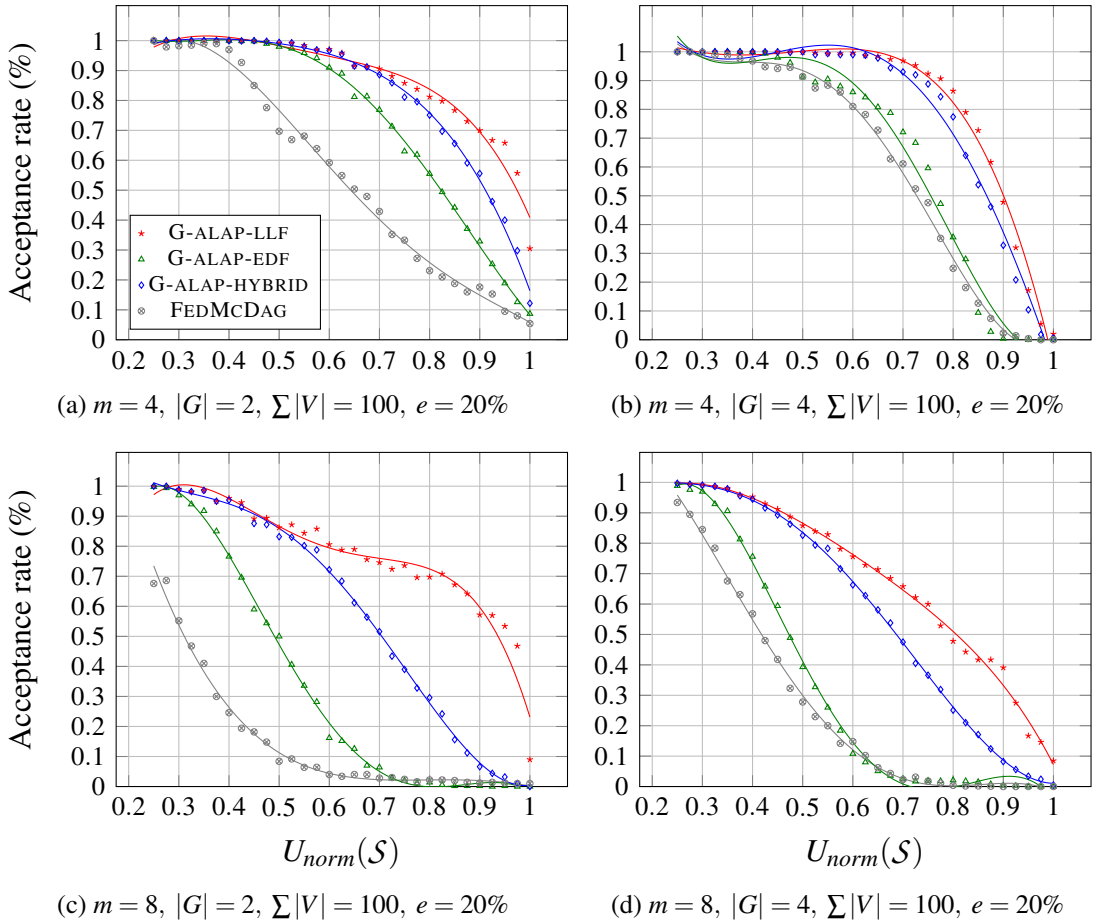
In the next section we present a study comparing our scheduling heuristics to the method presented in the state-of-the-art [81] when multiple MC-DAGs with different periods are on the same system . We include results presenting the number of preemptions per job generated for each heuristic: *we did not present these results for single MC-DAG scheduling since they had the same order of magnitude* and conclusions can be drawn from our experiments with multiple MC-DAGs. While G-ALAP-LLF might give better performances in terms of acceptance rate, it is known that solutions based on laxities entail an important number of preemptions which could compromise the feasibility of the scheduler.

8.2.2 Multiple MC-DAG scheduling

We now present the results of our analysis on MC systems composed of more than one MC-DAG. Our main objective is to statistically evaluate how our heuristic outperform the federated approach of the literature [81]. Like we explained in Chapter 5, the federated approach [81] creates clusters of cores to reduce the multiple MC-DAG scheduling to various single MC-DAG scheduling sub-problems. The advantage of this approach is that sporadic activations of MC-DAGs are possible and therefore the execution model is more generic than our global implementations of MH-MCDAG. Nonetheless, by considering periodic activations of MC-DAGs, the federated approach has better resource utilization since there is no offset between MC-DAG activations. At the same time, the type of data-driven reactive systems we are interested in (*e.g.* flight control systems) follow periodic activations of software components. Finally, to the best of our knowledge only the federated approach [81] is the only existing algorithm capable of scheduling multiple MC-DAGs in a single architecture.

We measured the acceptance rate of our heuristics (G-ALAP-LLF, G-ALAP-HYB and G-ALAP-EDF) compared to FEDMCDAG [81]. We also provide an analysis on the number of preemptions that are produced by all scheduling heuristics since it is an important aspect to consider for the feasibility of the schedulers. The generation parameters for the random MC systems are the following:

- Multi-core architectures have four or eight cores.
- We increase the number of MC-DAGs from two to four. The periods given to the MC-DAGs are chosen randomly from a pre-defined list of periods (like it was explained in Section 7.2, Chapter 7).



(a) $m = 4,\ |G| = 2,\ \sum|V| = 100,\ e = 20\%$

(b) $m = 4,\ |G| = 4,\ \sum|V| = 100,\ e = 20\%$

(c) $m = 8,\ |G| = 2,\ \sum|V| = 100,\ e = 20\%$

(d) $m = 8,\ |G| = 4,\ \sum|V| = 100,\ e = 20\%$

Figure 8.2: Comparison to existing multiple MC-DAG scheduling approach

- The number of tasks is fixed to 100, *i.e.* between all the MC-DAGs that are created we have 100 tasks in total.
- The probability to have an edge between two vertices was set to 20% since it represents the normal amount of data-dependencies that are included in industrial safety-critical applications.

This evaluation method is therefore focused on the effects of multi-periodicity caused by the inclusion of more than one MC-DAG in a single MC system.

Acceptance rate

Fig. 8.2 presents the results obtained in terms of acceptance rate for our scheduling heuristics compared to the federated approach of literature. Points in the curve present the measurements obtained by the scheduling heuristics and for clarity we have also presented the polynomial fitting curves (like we did for Fig. 8.1). 500 random MC systems were

generated for each experiment. The Y-axis represents the acceptance rate for randomly generated MC systems, while the X-axis gives the normalized utilization of the architecture considered for the generation.

Like we mentioned at the beginning of this subsection, our study is mostly focused on the effects of multi-periodicity: tasks now have to be scheduled multiple times during the hyper-period. Fig. 8.2a and Fig. 8.2c presents the results obtained when 2 MC-DAGs are considered in the system and both of them have 50 vertices. We increased the number of MC-DAGs to four in Fig. 8.2b and Fig. 8.2d.

Like for single MC-DAG scheduling the general observation in terms of performance remains the same: it is G-ALAP-LLF that shows the best performances, followed by G-ALAP-HYB, then G-ALAP-EDF and finally FEDMCDAG.

For G-ALAP-LLF, the performances obtained by the scheduling heuristic are quite good when two MC-DAGs are deployed into a quad-core architecture (Fig. 8.2a): over 80% of systems are schedulable when $U_{norm} = 0.8$ with this heuristic. If we increase the number of MC-DAGs considering the same hardware architecture (Fig. 8.2b) and the same cumulative number of tasks ($\sum |V| = 100$): the performance of the heuristic is similar to the previous case up to the point when the system is at 80% of its utilization, afterwards the degradation is more significant. However, when we increase the number of cores from four to eight, for two (Fig. 8.2c) and four MC-DAGs (Fig. 8.2d) performances change more significantly between both experiments. Comparing to the case when the architecture had only four cores, we can draw the following conclusion: the performance is significantly degraded because the timing budgets for tasks becomes bigger. For instance between Fig. 8.2a and Fig. 8.2c, instead of having 80% of schedulable systems in four cores, we have 70% in eight cores. In fact, the degradation is more noticeable when more MC-DAGs are in the system and the architecture has more processors. This last observation is due to the fact that tasks have more than one activation during the computation of the scheduling table and their timing budget is larger (Fig. 8.2d).

When we look into the behavior of G-ALAP-HYB, its performance is almost comparable to what G-ALAP-LLF produced when the system has four cores (Fig. 8.2a and Fig. 8.2c). The biggest difference in terms of acceptance rate is seen when two MC-DAGs are considered in the system and the utilization is over 80% (Fig. 8.2a): on average G-ALAP-LLF schedules 15% more systems than G-ALAP-HYB. This difference is less remarkable when four MC-DAGs are considered in the architecture, after the system reaches 70% G-ALAP-LLF only outperforms G-ALAP-HYB by a 10% margin. The fact that G-ALAP-LLF dominates G-ALAP-HYB can be explained by the fact that scheduling the system in the HI-criticality mode is also a difficult problem, which G-LLF tends to

do better than G-EDF. When the number of cores increases to eight, G-ALAP-HYB shows a degraded performance and the difference with G-ALAP-LLF is less important. When two MC-DAGs are on the system (Fig. 8.2c), after the system reaches a utilization rate of 55%, G-ALAP-HYB degrades in a linear fashion. The same type of degradation is visible for MC systems with four MC-DAGs and when the utilization is over 50% (Fig. 8.2d). However, the difference between G-ALAP-LLF and G-ALAP-HYB is less visible in this case.

The analysis of G-ALAP-EDF is the following: when the system considered has four cores its performances are good until the point where the system has a utilization inferior to 70%. There is a degradation in performance after this point for both cases, when two MC-DAGs are in the system (Fig. 8.2a) G-ALAP-EDF follows a polynomial decrease while when four MC-DAGs are in the system there is linear degradation (Fig. 8.2b). Like in single MC-DAG results, G-ALAP-EDF suffers a performance loss when the architecture has more cores (Fig. 8.2c and Fig. 8.2d). The scheduling problem does become more difficult when more MC-DAGs are in the system, but the difference is very slight (only 5% difference in acceptance rate). Another important observation concerns results produced by G-ALAP-EDF and by the approach from the literature [81]. G-ALAP-EDF clearly outperforms FEDMCDAG when only two MC-DAGs are in the system and the architecture has four or eight cores (Fig. 8.2a and Fig. 8.2c).

The principle of FEDMCDAG to create clusters of cores for MC-DAGs that have a high utilization (greater than one), has the advantage of being quite simple and inherit from the correctness of single MC-DAG scheduling. However, the results shown in Fig. 8.2 clearly shows that our heuristics outperform this method in terms of acceptance rate. For example when only two MC-DAGs are in the system (Fig. 8.2a and Fig. 8.2c), the results of FEDMCDAG are up to 50% less effective than G-ALAP-LLF with four cores(Fig. 8.2a) or even 70% when eight cores are considered (Fig. 8.2c). The fact that our approach is global and not federated is why systems are more schedulable in our case. There is an improvement in the results delivered by the approach when the number of MC-DAGs increases (Fig. 8.2b and Fig. 8.2d) which was expected because MC-DAGs with a low utilization (lower than one) are transformed into sequential tasks and isolated into a single cluster. These results are also comparable to the performance of G-ALAP-EDF which is in accordance to the previous results we had for single MC-DAG scheduling.

Considering multiple MC-DAGs in a single system does in fact make the problem more difficult for most of the heuristics that we study. The general observation in terms of performance between the heuristics is the same that we established for single MC-DAG scheduling: G-ALAP-LLF gives the best results, followed by G-ALAP-HYB, G-

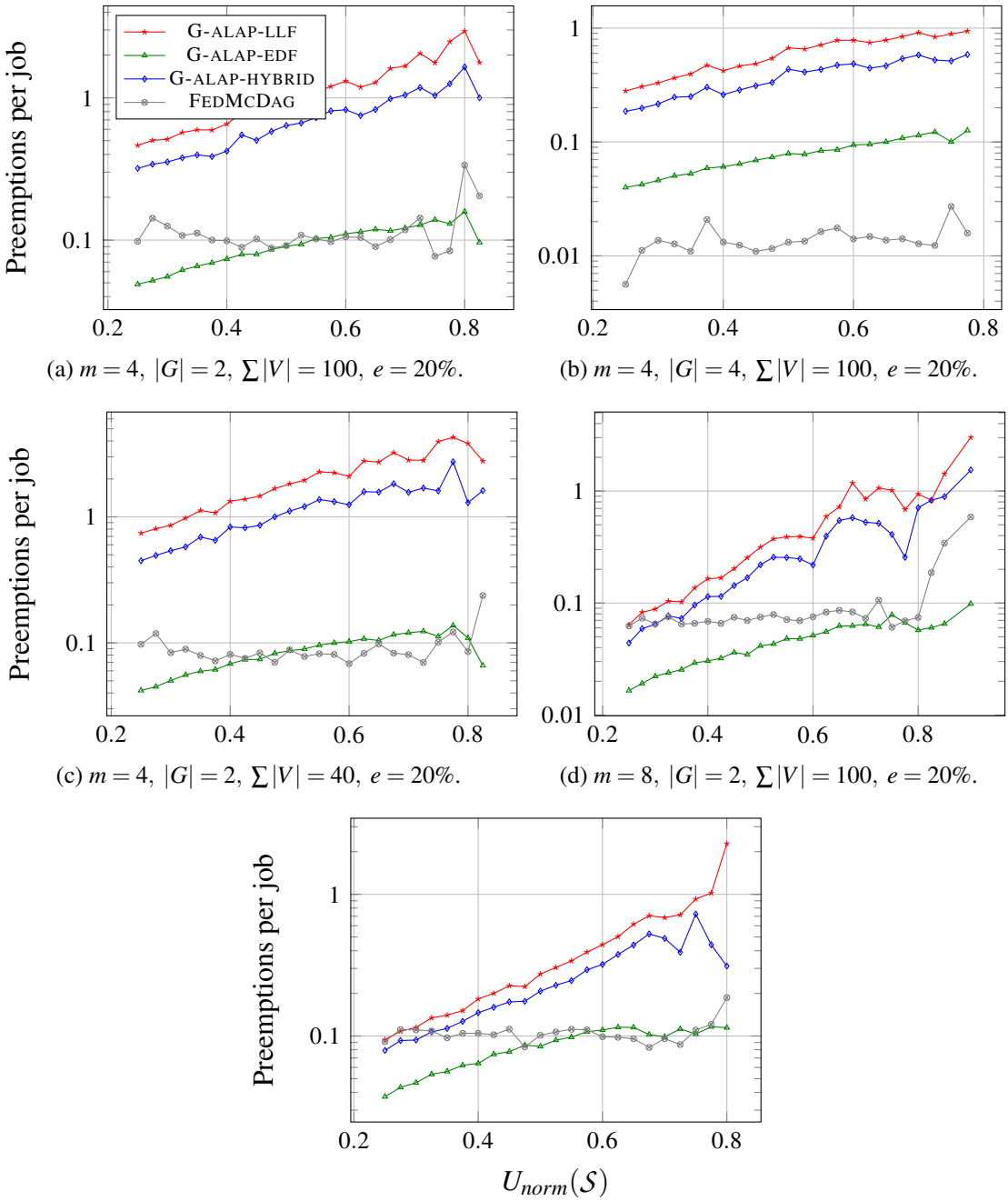
ALAP-EDF and FEDMCDAG. For some configurations, the last two are almost identical when the utilization of the system increases. When MC systems are composed of MC-DAGs with a high utilization, FEDMCDAG tends to give the worst performances. On the other hand, when the utilization of the system is decomposed into more MC-DAGs, since their utilization can be inferior to one, they are transformed into sequential tasks by FEDMCDAG. By doing so, the schedulability of this algorithm improves but it defeats the purpose of having MC-DAGs and parallel execution in applications in the first place.

Entailed number of preemptions

Besides the acceptance rate obtained by a scheduling approach, another metric that needs to be taken into account is the *number of preemptions* for jobs. In fact, having too many preemptions in the system could invalidate the theoretical results obtained since the overhead caused by context switches and migrations would not be negligible and potential deadline misses could occur. For this reason we look into the number of preemptions per job produced by each one of the heuristics we have presented. We limit the results to the most relevant cases since the magnitude order across all results was similar. For single MC-DAG scheduling the results also showed the same magnitude order so we have not included them in this chapter. We have also limited our study to the cases where all scheduling approaches were able to schedule a system. For instance, when the utilization is high and G-ALAP-LLF is the only heuristic capable of scheduling the system, the number of preemptions is not comparable since the other algorithms were unable to find a feasible MC correct schedule.

The results regarding the number of preemptions are illustrated in Fig. 8.3. Like for the acceptance rate experiments, we generated 500 random MC systems and tried to schedule them with the four heuristics presented in this chapter. Points in the plots represent an average of the number of preemptions for schedulable systems out of the 500 randomly generated test files. We aim to demonstrate the impact that some generation parameters have on the number of preemptions for each heuristic we have studied. The parameter changes we present are the following:

- Fig. 8.3a is used as a basis for our comparisons.
- The number of DAGs is increased from two to four in Fig. 8.3b.
- The number of tasks per DAG is decreased from 50 to 20 in Fig. 8.3c.
- The number of cores is increased from four to eight in Fig. 8.3d.



(a) $m = 4,\ |G| = 2,\ \sum|V| = 100,\ e = 20\%$.

(b) $m = 4,\ |G| = 4,\ \sum|V| = 100,\ e = 20\%$.

(c) $m = 4,\ |G| = 2,\ \sum|V| = 40,\ e = 20\%$.

(d) $m = 8,\ |G| = 2,\ \sum|V| = 100,\ e = 20\%$.

(e) $m = 4,\ |G| = 2,\ \sum|V| = 100,\ e = 40\%$.

Figure 8.3: Number of preemptions per job

- The density becomes greater (from 20% to 40%) in the graph of Fig. 8.3e.

For all the plots in Fig. 8.3, the X-axis is the normalized utilization of the system. The Y-axis gives the average number of preemptions per job: it is the division between the number of preemptions and the number of jobs activations. **We use a logarithmic scale on the Y-axis to illustrate the results.**

Solutions based on G-LLF: Like it was expected, solutions based on the G-LLF scheduler entail more preemptions: between 0.4 to 4 preemptions per job for G-ALAP-LLF and between 0.3 to 1 for G-ALAP-HYB. When the utilization of the system increases, the average number of preemptions increases as well. This increment is caused by tasks with an increased timing budget which causes them to be executed for a longer period of time and therefore the chances to be preempted increase as well. This type of behavior can be seen for the set of plots presented in Fig. 8.3.

An important result we want to insist on is the number of preemptions that G-ALAP-HYB produces compared to G-ALAP-LLF. The purpose of a hybrid algorithm combining G-EDF for the HI-criticality mode and G-LLF for the LO-criticality mode was to limit the number of preemptions that can be generated while the algorithm computes the HI-criticality scheduling table. The results of Fig. 8.3 show that we were in fact capable of limiting the number of preemptions thanks to G-ALAP-HYB. At the same time, while G-ALAP-LLF and G-ALAP-HYB generate more preemptions they tend to use less cores to find feasible schedules since clusters are not necessary for the scheduling of the system.

G-ALAP-EDF vs. Federated approach: When comparing the results in terms of number of preemptions between G-ALAP-EDF and FEDMCDAG, the results are more sparse and different conclusions can be established.

When we look into the results presented Fig. 8.3a, Fig. 8.3c and Fig. 8.3e, we have the same behavior for both heuristics: at the beginning FEDMCDAG generates more preemptions but as utilization increases, G-ALAP-EDF takes the upper hand and entails more preemptions compared to FEDMCDAG. Nevertheless, we can see that while G-ALAP-EDF tends to generate more preemptions, both heuristics are placed in the same threshold varying from 0.01 to 0.4 average number of preemptions per job.

In the last two figures we have two different behaviors. When the number of MC-DAGs is increased to four, Fig. 8.3b shows that G-ALAP-EDF has more preemptions than FEDMCDAG: since FEDMCDAG transforms MC-DAGs with a low utilization into a single bigger sequential task, the number of jobs decreases and therefore the average number of preemptions is inferior in this case. Also the clustering of MC-DAGs gives exclusive cores to tasks which limits the cases when tasks need to be preempted. In Fig. 8.3d on the other hand, we can see that FEDMCDAG creates the most preemptions when the number of

cores increases to eight. This can be explained by the fact that HI-criticality tasks are being scheduled ASAP and potentially preempting LO-criticality that were being executed. This does not happen for G-ALAP-EDF at the beginning but as utilization increases we can see that both curves tend to group around the same value.

While G-ALAP-LLF and G-ALAP-HYB give the better results in terms of acceptance rate (Fig. 8.1 and Fig. 8.2), we have seen that these solutions tend to entail an important number of preemptions (Fig. 8.3). In comparison to G-ALAP-EDF and FEDMCDAG, our solution based on G-LLF generates on average 100 times more preemptions per job. However, without knowing the implications that a preemption and context switch has on the targeted system/architecture it is impossible to draw a conclusion on the applicability of our scheduling heuristics. We are conscious of this limitation and is the main reason we have proposed solutions using both G-LLF and G-EDF or only G-EDF to schedule MC-DAGs on multi-core architectures.

8.3 A study on generalized MC-DAG systems

In this section we present a study on the schedulability of MC systems composed of MC-DAGs with more than two criticality levels. To the best of our knowledge, our heuristics (G-ALAP-LLF, G-ALAP-HYBand G-ALAP-EDF) are the only ones capable of performing this type of scheduling. Other approaches considering more than two criticality levels have not been adapted to handle precedence constraints. We aim at analyzing the impact that criticality levels have on the scheduling problem of MC-DAGs on multi-core processors.

8.3.1 Generalized single MC-DAG scheduling

In the previous section we demonstrated the influence that generation parameters have on the acceptance rate and number of preemptions for our heuristics. In order to analyze exclusively the effects that multiple criticality levels have on the acceptance rate, we have decided to fix most of the generation parameters.

The experimental evaluation is set as follows:

- There is a single MC-DAG in the system.
- The number of vertices for the MC-DAG is set to 20.
- The number of cores is set to eight.

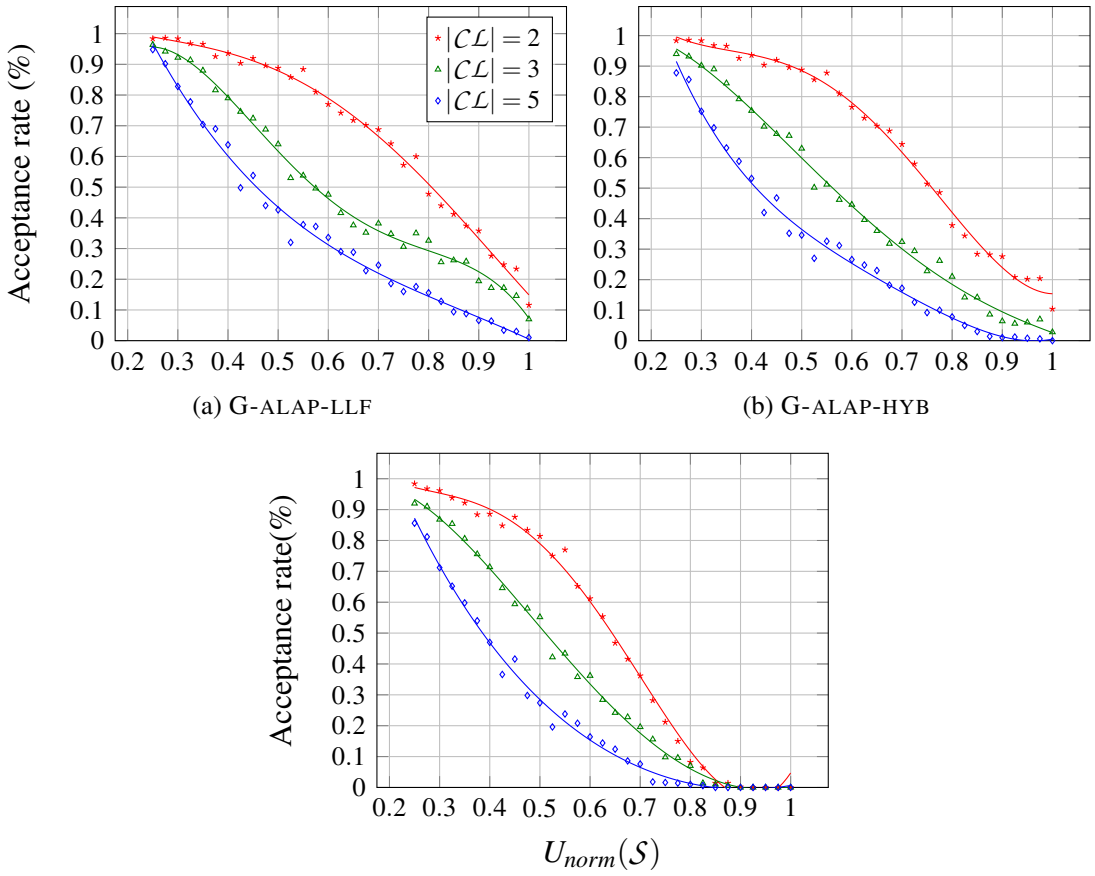
- The probability to have an edge between two is set to 20%.
- Criticality levels are set to two, three and five. In airborne systems the maximum levels of criticality is five, so we do not test systems with more than five criticality levels. It is very unlikely that a single architecture will host five different criticality levels as well.
- The reduction factor is set to two: the utilization rate occupied by tasks that have a higher level than the level considered during the generation is divided by two.
- The number of tasks that are created at each level is proportional to the number of levels, *i.e.* when five criticality levels are considered, four tasks have the highest criticality levels and four tasks have the lowest criticality level.

Acceptance rate

Fig. 8.4 presents the results we have obtained for our three heuristics in terms of acceptance rate (Y-axis) in function of the normalized utilization (X-axis). Each point that is presented in the graphs represents the acceptance rate of 500 randomly generated MC systems.

The behavior of G-ALAP-LLF when the MC systems has two or more criticality levels is presented in Fig. 8.4a. Each curve in the plot gives the acceptance rate for systems with two, three and five criticality levels. For the dual-criticality case, we already saw the performances of G-ALAP-LLF in Fig. 8.1e: the acceptance rate is quite good since more than 60% of systems are schedulable when the utilization is lower than 75%. The green polynomial approximation with the triangle marks represent the acceptance for three criticality levels: like we anticipated the scheduling problem becomes more complex and therefore an important degradation in the acceptance rate is visible, when $U_{norm}(\mathcal{S}) = 0.6$ the acceptance rate was almost at 78% as opposed to 49%. When five criticality levels are considered for the MC system, the acceptance rate drops to 33% for $U_{norm}(\mathcal{S}) = 0.6$.

The effects of multiple criticality levels on G-ALAP-HYB are illustrated in Fig. 8.4b. G-ALAP-HYB schedules all the criticality modes that are not the lowest one using the G-EDF priority ordering we defined, it is only at the lowest criticality mode that the heuristic applies the G-LLF priority ordering. Again the behavior for dual-criticality systems was shown in Fig. 8.1e, and while results were inferior to G-ALAP-LLF, the acceptance rate was still quite good: over 60% of schedulable system when the utilization is lower than 70%. We can see there is an important degradation when three criticality levels: on average G-ALAP-HYB schedules 30% less MC systems. This performance loss is also visible

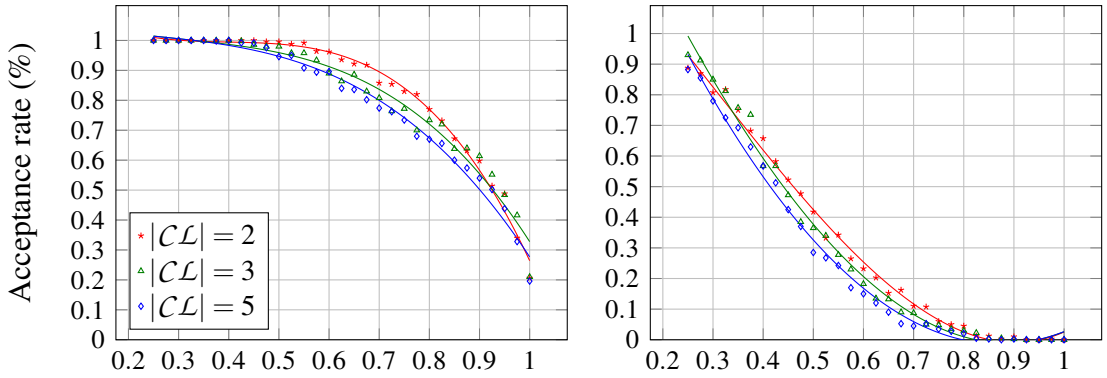


(a) G-ALAP-LLF

(b) G-ALAP-HYB

(c) G-ALAP-EDF

Figure 8.4: Impact of having multiple criticality levels on single MC-DAG systems. $m = 8,\ |G| = 1,\ |V| = 20,\ e = 20\%$.



(a) G-ALAP-LLF
$m = 4,\ |G| = 1,\ |V| = 50,\ e = 20\%.$

(b) G-ALAP-EDF
$m = 8,\ |G| = 4,\ \sum|V| = 80,\ e = 40\%.$

Figure 8.5: Saturation problem in generation parameters

between three and five criticality levels, on average 20% less of MC systems are schedulable with five criticality levels.

Fig. 8.4c presents the results obtained for G-ALAP-EDF when two and more criticality levels are considered in the system. Like in the two previous cases, G-ALAP-EDF has more troubles to schedule systems when the number of criticality levels increases. The difference is nonetheless less remarkable compared to results obtained by G-ALAP-LLF and G-ALAP-HYB: the performance loss is around 20% when we compare two criticality levels to three levels, and also around 20% when the system has five criticality levels.

Like it was stated, the fact that more criticality levels are included in the systems makes the scheduling of MC-DAGs more difficult. The performance loss for all our heuristics was expected, nonetheless the curves follow a polynomial performance loss.

It was not possible to establish a conclusion regarding the number of preemptions. Because the scheduling problem becomes more difficult as the number of criticality levels increases there are less schedulable systems. Therefore the average number of preemptions entailed by our algorithms also decreased. This gives the impression that by increasing the number of criticality levels, we are decreasing the average number of preemptions needed to find a feasible MC correct schedule. This may not be true since more tasks need respect the **Generalized Safe Trans. Prop.** which implies more preemptions.

In the next part of our analysis for generalized MC-DAGs, we present the effects that some generation parameters had on the schedulability of the system.

8.3.2 The parameter saturation problem

For our evaluations on the generalized scheduling of MC systems, we tested many combinations of generation parameters for MC-DAGs. Like we stated at the beginning of this chapter, there are many configurations that can be considered for the generation of MC systems. We wanted to test as many configurations as possible to establish a conclusion on the influence on the number of criticality levels a system can have. Nonetheless, after performing various experimentations, we realized that some of these parameters tend to produce specific MC-DAG shapes influencing the scheduling problem to a point where criticality levels do not play a major role on schedulability. We refer to this phenomena as the *parameter saturation problem*.

As a matter of fact, in Fig. 8.1d (one MC-DAG, with $e = 40\%$ and $|V| = 50$), we started to see this problem when the number of tasks and the graph's density increased in a dual-criticality system: the acceptance rates for all heuristics were less sparse because the only schedulable cases were specific to certain MC-DAG shapes that were easier to schedule (*e.g.* all HI-criticality vertices are sources or are always scheduled before LO-criticality tasks). Fig. 8.5 shows a couple of examples of the parameter saturation problem. When we increased the number of tasks and had an architecture of only four cores, for G-ALAP-LLF the difference in terms of acceptance rate between two, three and five levels of criticality was almost negligible (Fig. 8.5a). We can see that there is still a degradation in the acceptance rate when the number of criticality levels increases, but as opposed to the results obtained in Fig. 8.4, this difference is very small: between 1 and 3%, as opposed to 30%. In Fig. 8.5b, we can also see that increasing the number of criticality levels has almost no influence in the acceptance rate of G-ALAP-EDF. In this case, we considered a MC system with four MC-DAGs containing each 20 vertices, a probability to have an edge of 40% and an octo-core architecture. The two results that were presented are just some examples to illustrate the saturation generation problem, other configurations faced the same limitation.

8.4 Conclusion

The experimental validation of our contributions related to the scheduling of MC-DAGs on multi-core processors was presented in this chapter.

We started by comparing our scheduling heuristics to the methods proposed by the state-of-the-art [80; 81]. This dual-criticality study was decomposed into *single* and *multiple* MC-DAG scheduling. Throughout all experimentations we confirmed that our G-

LLF-based solutions have a better performance in terms of acceptance rate compared to solutions based on G-EDF and to the state-of-the-art. In most cases, G-ALAP-EDF gave better results in terms of acceptance rate compared to the state-of-the-art [80; 81]. And even in configurations where this was no longer the case, the difference in acceptance rate between G-ALAP-EDF and existing contributions [80; 81] remained small: between 1% and 5%.

We analyzed the influence that certain parameters have on the schedulability of MC-DAGs: the number of vertices, the number of cores in the architecture and the graph's density (*i.e.* the probability to have an edge between two vertices) have an important impact on the difficulty of the scheduling problem. We found that graph's density and the targeted architecture in particular have a major impact in the schedulability problem for the heuristics that were studied.

The difference in terms of acceptance rate between our global approaches and the federated approach [81] was more notorious when multiple MC-DAGs with different periods were considered in the system. This was expected since the federated approach reduces the multiple MC-DAG scheduling problem into the scheduling of a single MC-DAG in a cluster of cores. In many cases, the cluster will not be used to its full capacity and more cores than what is actual needed are demanded by the federated algorithm. This performance difference was quite notorious under certain hypotheses, *e.g.* our approach was capable of scheduling over 60% more systems than the federated approach. For the multiple MC-DAG scheduling problem we also provided results on the number of preemptions per job that are generated by each one of the scheduling heuristic we analyzed. It is well-known that laxity based schedulers tend to generate more preemptions than deadline based schedulers. This was in fact confirmed by our experimentations since G-ALAP-LLF would generate 100 times more preemptions per job than our solution based on G-EDF. If the number of preemptions is a limiting factor for system designers our two other heuristics G-ALAP-HYB and G-ALAP-EDF have demonstrated to give better schedulability results with less preemptions.

The final part of this chapter presented an evaluation of our scheduling heuristics for MC systems with an increasing number of criticality levels. The scheduling problem of MC-DAGs into multi-core architectures becomes more difficult since more tables need to be computed and they need to be compatible in order to ensure safe mode transitions to higher criticality levels. We saw that all our heuristic suffered from a performance loss when the criticality levels increased from two to three, and from three to five. This performance loss was not less than 20% on average.

This concludes our exhaustive statistical validation of our scheduling heuristics defined in Chapter 5. Many configurations for the generated systems were tested and the influence of the generation parameters were presented throughout the chapter. Overall our heuristics have proven to be more efficient in terms of acceptance rate. Therefore, system designers can choose a solution among the heuristics we have presented in function of their needs. For example, systems with a high utilization are more likely to be schedulable with our G-LLF adaptations. If the number of preemptions needs to be kept minimal, G-ALAP-EDF can be tested first to see if feasible schedulers can be found. This latest heuristic entails a comparable number of preemptions to approaches of the state-of-the-art [80; 81].

9 Conclusion and Research Perspectives

TABLE OF CONTENTS

This chapter presents the conclusion for this dissertation. We also proceed to identify future research perspectives.

9.1 Conclusions

Safety-critical systems are facing two major trends in their design and deployment. First of all, manufacturers are pushing towards the reduction of non-stringent functional constraints like power consumption, size, weight, price and heat. Thanks to processing capabilities offered by multi-core architectures this objective is tangible. Conversely, the demand to deliver additional functionalities on safety-critical systems keeps increasing. Research around the Mixed-Criticality execution model has given promising results to achieve this last objective since temporal correctness is ensured while additional functionalities are delivered.

Logical correctness required in the safety-critical domain is often achieved thanks to computation models like data-flow graphs which have data dependencies and are said to be data-driven. For this reason, in this thesis we have studied the problem of deploying Mixed-Criticality (MC) data-driven applications into multi-core architectures.

To propose solutions to this complex problem, we started by looking into the scheduling of MC applications with precedence constraints into multi-core processors. At the same time, while the MC model claims that less critical services become available thanks to this execution model, there has not been contributions capable of quantifying the avail-

ability of less critical software components. In case the minimum service guarantee is not satisfied, means to improve availability can be used to achieve this objective.

The contributions that have been presented in this manuscript are the following:

- The task model we defined is presented in Chapter 4. The MC-DAG model incorporates all the necessary aspects related to our research works: the MC execution model, precedence constraints in the form of graphs and a fault model to perform availability analyses.

- In Chapter 5, we presented our contributions related to the scheduling of MC-DAGs into multi-core architectures. The scheduling problem for MC dual-criticality systems and data-dependent applications is known to be very complex (*NP*-hard in the strong sense).

  To propose an efficient approximate solution, we started by defining a property to ensure safe mode transitions to the HI-criticality mode: **Safe Trans. Prop.** This property states that, as long as a sufficient timing budget was given to HI-criticality tasks in the LO-criticality mode, the mode transition to the HI-criticality mode can take place without missing any deadline. Building upon this property **we defined a generic meta-heuristic to produce MC-correct scheduling tables for MC-DAGs: MH-McDAG**. The genericity of our approach allowed us to adapt efficient real-time scheduling algorithms for multi-core architectures.

  Existing approaches capable of scheduling MC-DAGs on multi-core architectures are implicit implementations of MH-McDAG. Nevertheless, they show the following limitations: (i) HI-criticality tasks are scheduled as soon as possible which constraints the execution of LO-criticality tasks significantly. (ii) When multiple MC-DAGs with different periods have to be scheduled, the problem is reduced to various single MC-DAG scheduling on a cluster of cores. Therefore, this scheduling approach often needs more cores than what is actually needed to find MC-correct scheduling tables.

  To overcome these limitations our implementations of MH-McDAG are based on the two following principles: (i) **the scheduling of HI-criticality tasks is done as late as possible in the HI-criticality mode** in order to constrain as less as possible the LO-criticality tasks scheduling. LO-criticality tasks will only be preempted if a HI-criticality tasks has to be scheduled in order to respect **Safe Trans. Prop.** (ii) **Our implementation of MH-McDAG is global**, *i.e.* all tasks can be executed on all cores. We defined a priority ordering based on G-LLF since it is known to

give good performances in terms of acceptance rate. We have also defined a priority ordering based on G-EDF which generates less preemptions.

The final part of the chapter presents a **generalization of our scheduling heuristic to support an arbitrary number of criticality levels**. The generalization is an inductive application of the scheduling method we designed for the dual-criticality case. Nonetheless, to guarantee safe mode transitions and maintain an "as late as possible" behavior in more than two criticality levels, we had to introduce a new constraint on the scheduling of tasks to guarantee MC-correctness in the generalized system.

- We defined techniques to perform the availability analysis of MC systems executing MC-DAGs in Chapter 6. This chapter also presented different improvements we have considered in order to enhance availability. To the best of our knowledge, the contributions presented in this chapter are the first ones to actually quantify the availability of LO-criticality tasks in MC systems.

  By defining a *fault model for tasks* (*i.e.* how often a TFE takes place) and a *recovery mechanism for LO-criticality tasks* after the system has performed a mode transition to the HI-criticality mode, we were able to **estimate the availability rate for LO-criticality tasks under the discard MC model** (the most widespread MC model of the literature). Thanks to this numerical evaluation, we concluded that the discard MC model degrades the availability of LO-criticality tasks significantly. This is an important limitation to the adoption of MC on practical safety-critical systems since a minimum service guarantee is expected to be delivered even for the less critical software components.

  In order to limit the degradation caused by the discard MC model, we have proposed to incorporate two different types of enhancements on MC systems. The first enhancement consists in **defining a more precise fault propagation model**, *i.e.* instead of systematically switching to the HI-criticality mode whenever a LO-criticality task produces a timing failure, only successors of this task are interrupted. By doing so, independent LO-criticality tasks are capable of completing their execution even after a TFE occurs in a LO-criticality task. This enhancement showed a significant improvement on the availability rate for the LO-criticality tasks, *e.g.* up to 1% in a context where availability is measured up to 10^{-9}.

  The second type of enhancement is the **incorporation of additional components to improve availability**. For instance availability formulas we defined in this chapter can be easily adapted to consider design patterns used in safety-critical systems like

Triple Modular Redundancy. This replication of components in addition to our fault propagation model improved the availability of LO-criticality tasks even further: up to a 4% increase. However, some design patterns are not applicable to systems limited in power consumption or weight. In this case, extensions to the real-time execution model can be considered. For instance, we looked into weakly-hard real-time tasks which are capable of missing a fixed number of deadlines within a fixed amount of sequential executions. To estimate the availability rate when this type of task is deployed in the system, we developed **translation rules to obtain probabilistic automata**. The availability equations we previously defined do not take into account previous states the task was in, which forces us to perform simulations. Probabilistic automata are capable of capturing all the elements related to the execution model of the system: the fault and fault propagation model, in addition to the recovery process. The PRISM model checker is used to perform system's simulations and obtain an estimation of the availability rate. This latest enhancement showed an improved of over 2% for tasks that were the most impacted by the discard MC model.

These **translation rules can be applied to MC systems with more than two criticality levels**. The possible execution states that the system can be in when more than two levels of criticality are considered in the system, is another reason to use probabilistic automata to estimate the availability rate of MC systems.

- The abovementioned contributions have been integrated into an open-source tool: the MC-DAG framework presented in Chapter 7. This tool is decomposed into three modules: a scheduling module (contributions presented in Chapter 5), an availability module to obtain probabilistic automata (contributions of Chapter 6) and an **unbiased random MC-DAG generator**.

  To achieve an unbiased generation for MC-DAGs we had to combine different contributions of the literature for DAG scheduling and real-time systems. This generation is unbiased in the sense that shapes that influence DAG's schedulability are avoided. At the same time, since the utilization given to the system and its task set is one of the main attributes used to judge the quality of an scheduling algorithm, we used existing works of the literature to uniformly distribute this utilization among the vertices of the MC-DAGs we create.

- The experimental validation of our scheduling contributions performed thanks to the MC-DAG framework is discussed in Chapter 8. The evaluation was divided into

dual-criticality assessment and a study of schedulability of generalized MC systems (*i.e.* systems with more than two criticality levels).

In the dual-criticality assessment we compared the implementations of our meta-heuristic to the existing methods of the state-of-art [80; 81]. Like we expected, **our G-LLF based heuristics outperforms the state-of-the-art in terms of acceptance rate**: the difference is notorious and goes up to a 60% in some cases. On the other hand, while our implementation based on tends to outperform the state-of-the-art, for systems with a high utilization the priority ordering can be too permissive. In those cases, the existing approaches [80; 81] perform better than our heuristic by a small margin. The number of preemptions was also measured for our scheduling methods and for the state-of-art. While it is true that laxity-based heuristics generate more preemptions, the acceptance rate obtained was significantly superior. The hybrid approach using the G-EDF priority ordering in the HI-criticality mode and the G-LLF priority ordering in the LO-criticality mode, can be used as a middle-ground solution to achieve a good acceptance rate and limit the number of preemptions.

The **incidence of generation parameters on the acceptance rate and on the number of preemptions was also demonstrated in our experimental evaluations**: increasing the density of the graph or the number of cores in the architecture considered have a major impact on the performances of all the scheduling heuristics.

To the best of our knowledge, **our experimental evaluations on generalized MC-DAGs is the first one to consider data-dependent MC tasks with an arbitrary number of criticality levels**. Like it was expected, considering multiple criticality levels increases the complexity of the scheduling problem: we saw an average of 20-30% degradation in the performance of our three heuristics as we increased the number of criticality to three and then five.

This dissertation presented two complementary contributions: **(i)** a scheduling algorithm and **(ii)** methods to evaluate and enhance availability for data-driven MC systems. These contributions were generalized to MC systems with an arbitrary number of criticality levels and were validated thanks to rigorous and precise evaluations.

9.2 Open problems and research perspectives

In this section we present some open problems and research perspectives for future works.

9.2.1 Generalization of the data-driven MC execution model

Considering different periods on vertices

Some reactive safety-critical systems have incorporated periods within vertices of a graph. This is has been seen in Flight Control Systems for example. Defining feasible and MC-correct schedulers in this case is more complicated in this case: firings of vertices cannot occur as soon as input data is available, the vertex can only execute after its release date and needs to complete its execution before its next period. In other words, *the data-dependency needs to satisfied within a timespan*. We pictured two possibilities that can be explored to solve this issue: (i) include "virtual" vertices that are allocated into "virtual" cores to enforce the interarrival time between activations of a vertex; or (ii) adopt a method to transform the SDF graph into a classic sporadic MC real-time task set.

Considering elastic tasks

We demonstrated that the discard MC model degrades the availability of LO-criticality tasks significantly. In order to improve and deliver a minimum service guarantee for LO-criticality tasks, the elastic task model [38; 66] is another solution that has been proposed in the literature. By decreasing the timing budget and increasing the period of LO-criticality tasks in the HI-criticality mode, some applications are capable of delivering a degraded service. The *implications of such a transformation in the context of data-driven applications has not been explored yet* but is an interesting research perspective. Our means to improve availability might not be applicable to all safety-critical systems so having an execution model with elastic tasks is another way to improve availability of MC systems.

9.2.2 Availability vs. schedulability: a dimensioning problem

In Chapter 6 we presented the dimensioning problem of safety-critical systems. An over-estimated WCET leads to less timing failure events but is also responsible for the under-utilization of the system. There is a trade-off to consider when systems are dimensioned. If system designers incorporate as many tasks as possible by reducing their timing budgets, tasks are prone to overrun and more mode transitions will occur. Availability is degraded in this case. On the other hand, incorporating less tasks with an increased timing budget will limit the number of mode transitions. Less LO-criticality services will be incorporated but will be more available in this case. *Defining architectural exploration techniques*

that take into account the trade-off between availability and schedulability can be another research perspective of our works.

9.2.3 MC-DAG scheduling notions in other domains

Data-driven models of computation have been used in other domains than safety-critical systems. For example DAG representations are used in scientific workflows. These workflows are then executed by distributed systems. Some aspects about workflow computing are adjacent to real-time scheduling. For instance, trying to respect deadlines or minimizing response time are some problems that this domain tries to solve. We believe *our contributions defining mode transitions and different execution times for vertices can be of use in workflow computing as well*. Computation processes of the scientific workflow can require more or less time depending on previous results or on system demand.

List of Publications

[MBP16] Roberto Medina, Etienne Borde, and Laurent Pautet. Availability analysis for synchronous data-flow graphs in mixed-criticality systems. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6. IEEE, 2016.

[MBP17] Roberto Medina, Etienne Borde, and Laurent Pautet. Directed acyclic graph scheduling for mixed-criticality systems. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 217–232. Springer, 2017.

[MBP18a] Roberto Medina, Etienne Borde, and Laurent Pautet. Availability enhancement and analysis for mixed-criticality systems on multi-core. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 1271–1276. IEEE, 2018.

[MBP18b] Roberto Medina, Etienne Borde, and Laurent Pautet. Scheduling multi-periodic mixed criticality dags on multi-core architectures. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018.

Bibliography

[1] N. R. Storey, *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[2] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[3] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[4] 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, IEEE, 2007.

[5] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 398–409, IEEE, 2009.

[6] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 29. wh freeman New York, 2002.

[7] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[8] S. Baruah, "Mixed criticality schedulability analysis is highly intractable," 2009.

[9] P. J. Prisaznuk, "Integrated modular avionics," in *Aerospace and electronics conference, 1992. naecon 1992., proceedings of the ieee 1992 national*, pp. 39–45, IEEE, 1992.

[10] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Digital Avionics Systems Conference, 2007. DASC'07. IEEE/AIAA 26th*, pp. 2–A, IEEE, 2007.

[11] "Autosar consortium. automotive open system architecture (autosar)." https://www.autosar.org/.

[12] X. Jean, *Maîtrise de la couche hyperviseur sur les architectures multi-coeurs COTS dans un contexte avionique*. PhD thesis, Télécom ParisTech, 2015.

[13] R. Kaiser and S. Wagner, "Evolution of the pikeos microkernel," in *First International Workshop on Microkernels for Embedded Systems*, p. 50, 2007.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[15] K. Schild and J. Würtz, "Scheduling of time-triggered real-time systems," *Operating Systems of the 90s and Beyond*, vol. 5, no. 4, pp. 335–357, 2000.

[16] H. Kopetz, "The time-triggered model of computation," in *Real-Time Systems Symposium*, 1998.

[17] A. K.-L. Mok, *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[18] F. Liu, A. Narayanan, and Q. Bai, *Real-time systems*. Citeseer, 2000.

[19] S. Baruah and J. Goossens, "Scheduling real-time tasks: Algorithms and complexity," *Handbook of scheduling: Algorithms, models, and performance analysis*, vol. 3, 2004.

[20] S. Baruah, "Partitioned edf scheduling: a closer look," *Real-Time Systems*, vol. 49, no. 6, pp. 715–729, 2013.

[21] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM (JACM)*, vol. 47, no. 4, pp. 617–643, 2000.

[22] C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 140–149, ACM, 1997.

[23] 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, IEEE, 2001.

[24] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[25] J. H. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," *Journal of Computer and System Sciences*, vol. 68, no. 1, pp. 157–204, 2004.

[26] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[27] J. Engblom, *Processor pipelines and static worst-case execution time analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2002.

[28] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[29] I. J. Stein, *ILP-based path analysis on abstract pipeline state graphs*. epubli, 2010.

[30] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based wcet analysis," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pp. 37–44, IEEE, 2001.

[31] A. Betts, N. Merriam, and G. Bernat, "Hybrid measurement-based wcet analysis at the source level using object-level traces," in *OASIcs-OpenAccess Series in Informatics*, vol. 15, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[32] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 91–101, IEEE, 2012.

[33] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, *et al.*, "Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study," in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pp. 241–248, IEEE, 2013.

[34] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, "Open challenges for probabilistic measurement-based worst-case execution time," *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 69–72, 2017.

[35] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, pp. 1–69, 2013.

[36] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," in *International Symposium on Mathematical Foundations of Computer Science*, pp. 90–101, Springer, 2010.

[37] 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, IEEE, 2011.

[38] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 147–152, EDA Consortium, 2013.

[39] K. Agrawal and S. Baruah, "Intractability issues in mixed-criticality scheduling," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 106, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[40] N. C. Audsley, "On priority asignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[41] Q. Zhao, Z. Gu, and H. Zeng, "Pt-amc: integrating preemption thresholds into mixed-criticality scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 141–146, EDA Consortium, 2013.

[42] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 155–165, IEEE, 2012.

[43] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "Mixed-criticality scheduling of sporadic task systems," in *European Symposium on Algorithms*, pp. 555–566, Springer, 2011.

[44] 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 *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 145–154, IEEE, 2012.

[45] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-time systems*, vol. 50, no. 1, pp. 48–86, 2014.

[46] A. Easwaran, "Demand-based scheduling of mixed-criticality sporadic tasks on one processor," in *2013 IEEE 34th Real-Time Systems Symposium*, pp. 78–87, IEEE, 2013.

[47] 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.

[48] P. Rodriguez, L. George, Y. Abdeddaïm, and J. Goossens, "Multicriteria evaluation of partitioned edf-vd for mixed-criticality systems upon identical processors," in *Workshop on Mixed Criticality Systems*, 2013.

[49] X. Gu and A. Easwaran, "Dynamic budget management with service guarantees for mixed-criticality systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, pp. 47–56, IEEE, 2016.

[50] J.-J. Han, X. Tao, D. Zhu, and H. Aydin, "Criticality-aware partitioning for multi-core mixed-criticality systems," in *Parallel Processing (ICPP), 2016 45th International Conference on*, pp. 227–235, IEEE, 2016.

[51] S. Ramanathan and A. Easwaran, "Utilization difference based partitioned scheduling of mixed-criticality systems," in *Proceedings of the Conference on Design, Automation & Test in Europe*, pp. 238–243, European Design and Automation Association, 2017.

[52] R. Gratia, T. Robert, and L. Pautet, "Generalized mixed-criticality scheduling based on run," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pp. 267–276, ACM, 2015.

[53] J. Lee, S. Ramanathan, K.-M. Phan, A. Easwaran, I. Shin, and I. Lee, "Mc-fluid: Multi-core fluid-based mixed-criticality scheduling," *IEEE Transactions on Computers*, no. 1, pp. 1–1, 2018.

[54] R. M. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *2012 24th Euromicro Conference on Real-Time Systems*, pp. 309–320, IEEE, 2012.

[55] H. Li and S. Baruah, "Outstanding paper award: Global mixed-criticality scheduling on multiprocessors," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 166–175, IEEE, 2012.

[56] 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*, IEEE, 2014.

[57] S. Baruah, A. Eswaran, and Z. Guo, "Mc-fluid: simplified and optimally quantified," in *Real-Time Systems Symposium, 2015 IEEE*, pp. 327–337, IEEE, 2015.

[58] A. Burns and S. Baruah, "Semi-partitioned cyclic executives for mixed criticality systems," in *Proc. 3rd Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 36–41, 2015.

[59] M. A. Awan, K. Bletsas, P. F. Souto, and E. Tovar, "Semi-partitioned mixed-criticality scheduling," in *International Conference on Architecture of Computing Systems*, pp. 205–218, Springer, 2017.

[60] J. Ren and L. T. X. Phan, "Mixed-criticality scheduling on multiprocessors using task grouping," in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pp. 25–34, IEEE, 2015.

[61] Z. Al-bayati, Q. Zhao, A. Youssef, H. Zeng, and Z. Gu, "Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pp. 630–635, IEEE, 2015.

[62] S. Ramanathan and A. Easwaran, "Mixed-criticality scheduling on multiprocessors with service guarantees," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 17–24, IEEE, 2018.

[63] 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*, pp. 257–266, ACM, 2015.

[64] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pp. 259–268, IEEE, 2015.

[65] I. Bate, A. Burns, and R. I. Davis, "An enhanced bailout protocol for mixed criticality embedded software," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 298–320, 2017.

[66] H. Su, N. Guan, and D. Zhu, "Service guarantee exploration for mixed-criticality systems," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–10, IEEE, 2014.

[67] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, "Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset," in *19th International Conference on Real-Time and Network Systems*, 2011.

[68] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine, "The syndex software environment for real-time distributed systems design and implementation," in *European Control Conference*, vol. 2, pp. 1684–1689, 1991.

[69] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[70] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *Design automation conference (dac), 2013 50th acm/edac/ieee*, pp. 1–10, IEEE, 2013.

[71] T. L. Adam, K. M. Chandy, and J. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[72] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[73] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[74] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of dags," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3242–3252, 2014.

[75] A. Parri, A. Biondi, and M. Marinoni, "Response time analysis for g-edf and g-dm scheduling of sporadic dag-tasks with arbitrary deadline," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pp. 205–214, ACM, 2015.

[76] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global edf scheduling of directed acyclic graphs on multiprocessor systems," in *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pp. 287–296, ACM, 2013.

[77] M. Qamhieh, L. George, and S. Midonnet, "A stretching algorithm for parallel real-time dag tasks on multiprocessor systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, p. 13, ACM, 2014.

[78] M. Qamhieh, L. George, and S. Midonnet, "Stretching algorithm for global scheduling of real-time dag tasks," *Real-Time Systems*, pp. 1–31, 2018.

[79] S. Baruah, "Implementing mixed-criticality synchronous reactive programs upon uniprocessor platforms," *Real-Time Systems*, vol. 50, no. 3, pp. 317–341, 2014.

[80] S. Baruah, "Implementing mixed criticality synchronous reactive systems upon multiprocessor platforms," *The University of North Carolina at Chapel Hill, Tech. Rep*, 2013.

[81] S. Baruah, "The federated scheduling of systems of mixed-criticality sporadic dag tasks," in *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pp. 227–236, IEEE, 2016.

[82] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu, "Mixed-criticality federated scheduling for parallel real-time tasks," *Real-Time Systems*, vol. 53, no. 5, pp. 760–811, 2017.

[83] R. M. Pathan, "Improving the schedulability and quality of service for federated scheduling of parallel mixed-criticality tasks on multiprocessors," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 106, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[84] P. Ekberg and W. Yi, "Schedulability analysis of a graph-based task model for mixed-criticality systems," *Real-Time Systems*, vol. 52, pp. 1–37, jan 2016.

[85] A. Singh, P. Ekberg, and S. Baruah, "Applying real-time scheduling theory to the synchronous data flow model of computation," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 76, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[86] A. Singh, P. Ekberg, and S. Baruah, "Applying real-time scheduling theory to the synchronous data flow model of computation," in *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)* (M. Bertogna, ed.), vol. 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 8:1–8:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[87] A. Singh, P. Ekberg, and S. Baruah, "Uniprocessor scheduling of real-time synchronous dataflow tasks," *Real-Time Systems*, pp. 1–31, 2018.

[88] E. C. Klikpo and A. Munier-Kordon, "Preemptive scheduling of dependent periodic tasks modeled by synchronous dataflow graphs," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pp. 77–86, ACM, 2016.

[89] K. Jad, *Modeling and scheduling embedded real-time systems using Synchronous Data Flow Graphs*. PhD thesis, Sorbonne Université, 2018.

[90] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Static scheduling of multi-rate and cyclo-static dsp-applications," in *VLSI Signal Processing, VII, 1994.,[Workshop on]*, pp. 137–146, IEEE, 1994.

[91] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *IEEE Transactions on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.

[92] M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *Proceedings of the ninth ACM international conference on Embedded software*, pp. 195–204, ACM, 2011.

[93] M. A. Bamakhrama, J. T. Zhai, H. Nikolov, and T. Stefanov, "A methodology for automated design of hard-real-time embedded streaming systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 941–946, EDA Consortium, 2012.

[94] A. Bouakaz, J.-P. Talpin, and J. Vitek, "Affine data-flow graphs for the synthesis of hard real-time applications," in *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pp. 183–192, IEEE, 2012.

[95] A. Bouakaz, T. Gautier, and J.-P. Talpin, "Earliest-deadline first scheduling of multiple independent dataflow graphs.," in *SiPS*, pp. 292–297, Citeseer, 2014.

[96] B. D. Theelen, M. C. Geilen, T. Basten, J. P. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on*, pp. 185–194, IEEE, 2006.

[97] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pp. 404–411, IEEE, 2011.

[98] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[99] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

[100] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[101] A. Burns, R. I. Davis, S. Baruah, and I. J. Bate, "Robust mixed-criticality systems," *IEEE Transactions on Computers*, 2018.

[102] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[103] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell Labs Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[104] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the 42nd annual Design Automation Conference*, pp. 819–824, ACM, 2005.

[105] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proceedings of the 44th annual Design Automation Conference*, pp. 777–782, ACM, 2007.

[106] M. R. Garey and D. S. Johnson, ""strong"np-completeness results: Motivation, examples, and implications," *Journal of the ACM (JACM)*, vol. 25, no. 3, pp. 499–508, 1978.

[107] H. S. Chwa, J. Lee, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for parallel tasks on multi-core platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1331–1345, 2017.

[108] O. Svensson, "Conditional hardness of precedence constrained scheduling on identical machines," in *Proceedings of the forty-second ACM symposium on Theory of computing*, pp. 745–754, ACM, 2010.

[109] A. Burns, "System mode changes-general and criticality-based," in *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, pp. 3–8, 2014.

[110] D. L. Parnas, A. J. Van Schouwen, and S. P. Kwan, "Evaluation of safety-critical software," *Communications of the ACM*, vol. 33, no. 6, pp. 636–648, 1990.

[111] L. A. Johnson *et al.*, "Do-178b, software considerations in airborne systems and equipment certification," *Crosstalk, October*, vol. 199, 1998.

[112] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, ACM, 2009.

[113] D. Maxim, R. I. Davis, L. Cucu-Grosjean, and A. Easwaran, "Probabilistic analysis for mixed criticality systems using fixed priority preemptive scheduling," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pp. 237–246, ACM, 2017.

[114] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *LNCS*, pp. 585–591, Springer, 2011.

[115] S. Stuijk, M. Geilen, and T. Basten, "Sdf^ 3: Sdf for free," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp. 276–278, IEEE, 2006.

[116] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proceedings of the 3rd international ICST conference on simulation tools and techniques*, p. 60, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[117] J. Lee, A. Easwaran, I. Shin, and I. Lee, "Zero-laxity based real-time multiprocessor scheduling," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2324–2333, 2011.

[118] R. Medina, E. Borde, and L. Pautet, "Directed acyclic graph scheduling for mixed-criticality systems," in *Ada-Europe International Conference on Reliable Software Technologies*, pp. 217–232, Springer, Cham, 2017.

[119] S.-H. Oh and S.-M. Yang, "A modified least-laxity-first scheduling algorithm for real-time tasks," in *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pp. 31–36, IEEE, 1998.

[120] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[121] J. Rushby, "Bus architectures for safety-critical embedded systems," in *International Workshop on Embedded Software*, pp. 306–323, Springer, 2001.

[122] H. Kopetz, "Fault containment and error detection in the time-triggered architecture," in *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on*, pp. 139–146, IEEE, 2003.

[123] A. Goldberg and G. Horvath, "Software fault protection with arinc 653," in *Aerospace Conference, 2007 IEEE*, pp. 1–11, IEEE, 2007.

[124] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proceedings of the 6th international workshop on Hardware/software codesign*, pp. 97–101, IEEE Computer Society, 1998.

[125] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.

[126] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[127] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, vol. 52, no. 4, pp. 399–449, 2016.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Déploiement de Systèmes à Flots de Données en Criticité Mixte pour Architectures Multi-cœurs

Mots clés : Ordonnancement, temps-réel, criticité mixte, architectures multi-cœurs, flux de données

Résumé : De nos jours, la conception de systèmes critiques va de plus en plus vers l'intégration de différents composants système sur une unique plate-forme de calcul dans le but de réduire la taille, poid, coût et diffusion de chaleur de ces systèmes.
Traditionnellement, les systèmes critiques sont conçus à l'aide de modèles de calcul comme les graphes data-flow et l'ordonnancement temps-réel pour fournir un comportement logique et temporel correct. Néanmoins, les ressources allouées aux data-flows et aux ordonnanceurs temps-réel sont fondées sur l'analyse du pire cas, ce qui conduit souvent à une sous-utilisation des processeurs. Cette sous-utilisation devient plus remarquable sur les architectures multi-cœurs où la différence entre le meilleur et le pire cas est encore plus significative.
Le modèle d'exécution à criticité mixte propose une solution au problème susmentionné : les ressources sont allouées en fonction du mode opérationnel du système. Tant que des capacités de calcul suffisantes sont disponibles pour respecter toutes les échéances, le système est dans un mode opérationnel de « basse criticité ». Cependant, si la charge du système augmente, les composants critiques sont priorisés pour respecter leurs échéances, leurs ressources de calcul augmentent et les composants moins/non critiques sont pénalisés. Le système passe alors à un mode opérationnel de « haute criticité ».
L' intégration des aspects de criticité mixte dans le modèle data-flow est néanmoins un problème difficile à résoudre. Des nouvelles méthodes d'ordonnancement capables de gérer des contraintes de précédences et des variations sur les budgets de temps doivent être définies.
Alors que le modèle de criticité mixte prétend que les composants critiques et non critiques peuvent partager la même plate-forme de calcul, l'interruption des composants non critiques réduit considérablement leur disponibilité. Ceci est un problème car les composants non critiques doivent offrir une degré minimum de service. C'est pourquoi nous définissons des méthodes pour évaluer et améliorer la disponibilité de ces composants.

Title : Deployment of Mixed-Criticality and Data-Driven Systems on Multi-core Architectures

Keywords : Scheduling theory, real-time systems, mixed-criticality, multi-core architectures, data-driven

Abstract : Nowadays, the design of modern Safety-critical systems is pushing towards the integration of multiple system components onto a single shared computation platform in order to reduce cost, size, weight, heat and power consumption.
Traditionally, safety-critical systems have been conceived using models of computations like data-flow graphs and real-time scheduling to obtain logical and temporal correctness. Nonetheless, resources given to data-flow representations and real-time scheduling techniques are based on worst-case analysis which often leads to an under-utilization of the computation capacity. The allocated resources are not always completely used. This under-utilization becomes more notorious for multi-core architectures where the difference between best and worst-case performance is more significant.
The mixed-criticality execution model proposes a solution to the abovementioned problem. To efficiently allocate resources while ensuring safe execution of the most critical components, resources are allocated in function of the operational mode the system is in. As long as sufficient processing capabilities are available to respect deadlines, the system remains in a 'low-criticality' operational mode. Nonetheless, if the system demand increases, critical components are prioritized to meet their deadlines, their computation resources are increased and less/non-critical components are potentially penalized. The system is said to transition to a 'high-criticality' operational mode.
Yet, the incorporation of mixed-criticality aspects into the data-flow model of computation is a very difficult problem as it requires to define new scheduling methods capable of handling precedence constraints and variations in timing budgets.
While the mixed-criticality model claims that critical and non-critical components can share the same computation platform, the interruption of non-critical components degrades their availability significantly. This is a problem since non-critical components need to deliver a minimum service guarantee. In fact, recent works in mixed-criticality have recognized this limitation. For this reason, we define methods to evaluate and improve the availability of non-critical components.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France