Gestion de l'énergie renouvelable et ordonnancement temps réel dans les systèmes embarqués

Younès Chandarli

UNIVERSITÉ PARIS-EST

THÈSE

en vue de l'obtention du titre de

DOCTEUR
de l'Université Paris-Est

Spécialité : INFORMATIQUE

École doctorale : MSTIC

YOUNES CHANDARLI

Real-time Scheduling for Energy-Harvesting Embedded Systems

---

soutenue le 2 Décembre 2014

Jury

Directeur :	LAURENT GEORGE	–	ESIEE Paris (LIGM), France
Encadrant :	DAMIEN MASSON	–	ESIEE Paris (LIGM), France
Rapporteurs :	DANIEL MOSSÉ	–	University of Pittsburgh, U.S.A
	HAKAN AYDIN	–	George Mason University, U.S.A
Examinateurs :	LAURENT PAUTET	–	Télécom Paris-Tech (LTCI), France
	MARYLINE CHETTO	–	Université de Nantes (IRCCyN), France

LABORATOIRE D'INFORMATIQUE
GASPARD-MONGE

Sous la co-tutelle de :
CNRS
ÉCOLE DES PONTS PARISTECH
ESIEE PARIS
UPEM • UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

PhD prepared at
LIGM (UMR 8049)
Laboratoire d'Informatique Gaspard-Monge
Cité Descartes, Bâtiment Copernic
5, boulevard Descartes
Champs-sur-Marne 77454 Marne-la-Vallée Cedex 2

ESIEE PARIS L'ÉCOLE DE L'INNOVATION TECHNOLOGIQUE

PhD in collaboration with
ESIEE Paris
Systems Engineering Department
2, boulevard Blaise Pascal
93162 Noisy le Grand CEDEX

À mes parents,
À mes grands parents,
À ma sœur et mon frère,
À toute ma famille,
À tous mes amis,
À toutes les personnes que j'ai eu le privilège de croiser.

Parce que chaque personne rencontrée, même brièvement,
est une occasion exceptionnelle de se redécouvrir
et de s'émerveiller.

To my parents,
To my grandparents,
To my brother and my sister,
To all my family,
To all my friends,
To all the people who have graced my life.

Because each encounter, even a brief one,
is an occasion to re-imagine oneself
and to marvel.

"The greatest part of a writer's time is spent in reading, in order to write: a man will turn over half a library to make one book". Samuel Johnson.

Author's publication list

International Conference Papers

ECRTS'2013 Yasmina Abdeddaïm, Younès Chandarli, and Damien Masson. "The Optimality of PFP_{ASAP} Algorithm for Fixed-Priority Energy-Harvesting Real-Time Systems". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 2013

RTNS'2014 Yasmina Abdeddaïm, Younès Chandarli, R.I Davis, and Damien Masson. "Schedulability Analysis for Fixed Priority Real-Time Systems with Energy-Harvesting". In: *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*. 2014, pages 67–76
RTNS 2014 Best Paper Award

ISORC'2015 Younès Chandarli, Nathan Fisher, and Damien Masson. "Response Time Analysis for Thermal-Aware Real-Time Systems Under Fixed-Priority Scheduling". In: *Proceedings of the IEEE Symposium On Real-time Computing (ISORC)*. 2015

Workshop and WIP[1] Papers

WATERS'2012 Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms". In: *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2012, pages 21–26

WIP RTCSA'2012 Younès Chandarli, Yasmina Abdeddaïm, and Damien Masson. "The Fixed Priority Scheduling Problem for Energy Harvesting Real-Time Systems". In: *Proceedings of the work in progress session of the the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2012

WIP RTAS'2013 Yasmina Abdeddaïm, Younès Chandarli, and Damien Masson. "Toward an Optimal Fixed-Priority Algorithm for Energy-Harvesting Real-Time Systems". In: *Proceedings of the Work in progress session of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pages 45–48

[1] Work In Progress

Technical Reports

1. Younès Chandarli, Manar Qamhieh, and Frédérique Fauberteau. *YARTISS: A Generic, Modular and Energy-Aware Scheduling Simulator for Real-Time Multiprocessor Systems*. technical report. Paris, France, 2013. URL: http://hal.archives-ouvertes.fr/hal-01076022

2. Yasmina Abdeddaïm and Younès Chandarli. *Optimal Real-Time Scheduling Algorithm for Fixed-Priority Energy-Harvesting Systems*. technical report. Paris, France, 2013. URL: http://hal.archives-ouvertes.fr/hal-01076021

Acknowledgment

I am using this opportunity to express my gratitude to everyone who supported me throughout this thesis. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the thesis.

I have to thank my research supervisors, Mr. Laurent George and Mr. Damien Masson. Without their assistance and dedicated involvement in every step throughout the process, this work would have never been accomplished. I would like to thank you very much for your support and understanding over these past three years.

I would also like to show gratitude to my committee, including Mr. Hakan Aydin, Mr. Daniel Mossé, Mrs. Maryline Chetto and Mr. Laurent Pautet.

I express my warm thanks to Mrs. Yasmina Abdeddaïm, Mr. Rob Davis and Nathan Fisher for their collaboration, support and guidance.

Getting through my thesis required more than academic support, and I have many, many people to thank for listening to and, at times, having to tolerate me over the past three years. I express all my gratitude and appreciation for their friendship. My colleagues have been unwavering in their personal and professional support during the time I spent at the University.

Last but not the least, I would like to thank my family: my parents for giving birth to me at the first place and supporting me throughout my life.

Contents

Abstract

In this thesis, we are interested in the real-time fixed-priority scheduling problem of energy-harvesting systems.

An energy-harvesting system is a system that can collect the energy from the environment in order to store it in an energy storage device and then to use it to supply an electronic device. This technology is more and more used in small embedded systems that are required to run autonomously for a very long lifespan. Wireless sensor networks and medical implants are typical applications of this technology.

Moreover, most of devices that work with energy-harvesting have to execute many recurrent tasks within a limited time. Thus, these energy-harvesting devices are subject to real-time constraints where the correctness of the system depends not only on the correctness of the results but also on the time in which they are delivered.

This thesis addresses the real-time scheduling layer of this kind of systems. More specifically, it focuses on a specific family of real-time scheduling: the preemptive fixed-task-priority scheduling for monoprocessor platforms. In this approach, each task of the system is assigned a static priority. Then, at any moment, the scheduler executes first the active task with the highest priority.

The problematic of such a scheduling approach for energy-harvesting systems is to find efficient scheduling algorithms and their associated schedulability analysis. The responsibility of a scheduling algorithm is to produce a valid and perpetual schedule where all the tasks respect their timing and energy constraints. In other words, all the deadlines must be met and the system must never run out of energy. Whereas a schedulability analysis provides schedulability tests and conditions that guarantee the schedulability or the unschedulability of a given task set in a given energy configuration. Furthermore, to guarantee such schedulability, the energy storage unit capacity must be sufficient to avoid energy wasting. Then, for each scheduling algorithm and schedulability analysis, we must specify the minimum battery capacity that keeps the schedulability of a given task set.

This dissertation starts with a brief state of the art of the existing real-time scheduling theory for energy-harvesting systems including energy-aware scheduling solutions that add more idle periods to replenish energy, and dynamic processor voltage and frequency scaling approaches that reduce the consumption rate of the processor. In this part, we show that in the case of energy-aware scheduling approach, the fixed-task-priority scheduling for energy-harvesting systems was not deeply studied.

The first contribution of this thesis is the proposition of the PFP_{ASAP} scheduling algorithm. It is an adaptation of the classical fixed-task-priority scheduling to the energy-harvesting context. It consists of executing tasks as soon as possible whenever the energy is sufficient to execute at least one time unit and replenishes otherwise. The

replenishment periods are as long as needed to execute one time unit. The schedule produced by this algorithm can be seen as the energy counter-part of the classical fixed-task-priority work-conserving scheduling where the processor cannot be idle while there are pending jobs and sufficient energy to execute. We prove in this dissertation that PFP_{ASAP} is optimal in the class of fixed-task-priority scheduling algorithms but only in the case of non-concrete systems where the first release time of tasks and the initial energy storage unit level are known at run-time and where all the tasks consume more energy than the replenishment during execution times. A sufficient and necessary schedulability condition for such systems is also proposed. It consists of a response time analysis that computes the longest response time of all tasks in the worst-case scenario. We prove also that this scenario happens whenever all the tasks are requested simultaneously while the energy storage unit level is at its minimum level.

Unfortunately, when we relax the assumption of tasks energy consumption profile, by considering both tasks that consume more energy than the replenishment and the ones that consume less than the replenishment, the PFP_{ASAP} is no longer optimal and the worst-case scenario is no longer the synchronous release of all the tasks, which makes the precedent schedulability test only necessary. To cope with this limitation, we propose in a second contribution to upper bound tasks worst-case response time in order to build sufficient schedulability conditions instead of exact ones. We propose two upper bounds based on the construction of virtual scenarios that maximize the number of interferences and the needed replenishment time.

Regarding the possibility of finding an optimal algorithm, we explore through this dissertation different ideas and approaches of scheduling policies in order to build an optimal algorithm for the general model of fixed-task-priority tasks by considering all types of task sets and energy consumption profiles. We show through some counter examples the difficulty of finding such an algorithm and we show that most of intuitive scheduling algorithms are not optimal. After that, we discuss the possibility of finding such an algorithm.

In order to better understand the scheduling problematic of fixed-priority scheduling for energy-harvesting systems, we also try to explore the solutions of similar scheduling problematics, especially the ones that delay executions in order to guarantee some requirements. The thermal-aware scheduling is one of these problematics. It consists of executing tasks such that a maximum temperature is never exceeded. This may lead to introduce additional idle times to cool down the system in order to prevent reaching the maximum temperature. As a first step, we propose in this thesis to adapt the solutions proposed for energy-harvesting systems to the thermal-aware model. Thus, we adapt the PFP_{ASAP} algorithm to respect the thermal constraints and we propose a sufficient schedulability analysis based on worst-case response time upper bounds.

Finally, we present *YARTISS*: the simulation tool used to evaluate the theoretical results presented in this dissertation.

Résumé

Dans cette thèse nous nous intéressons à la problématique de l'ordonnancement temps réel à priorité fixe des systèmes embarqués récupérant leur énergie de l'environnement.

Ces derniers collectent l'énergie ambiante de l'environnement et la stockent dans un réservoir d'énergie afin d'alimenter un appareil électronique. Cette technologie est de plus en plus utilisée dans les petits systèmes embarqués qui nécessitent une longue autonomie et une longue durée de vie. Les réseaux de capteurs et les implants médicaux sont des applications typiques de cette technologie.

De surcroît, dans la majorité des cas, les systèmes qui opèrent avec cette technologie doivent exécuter des tâches récurrentes dans un temps imparti. Ainsi, ces systèmes sont soumis à des contraintes dites temps réel où le respect des contraintes temporelles est aussi important que l'exactitude des résultats.

Cette thèse traite l'ordonnancement temps réel de ce genre de systèmes. Plus précisément, Nous nous intéressons à une famille spécifique de l'ordonnancement temps réel : l'ordonnancement préemptif à priorité fixe sur des plateformes monoprocesseur. Dans cette approche, chaque tâche se voit attribuer une priorité statique qui ne change pas tout au long de son cycle de vie. Ainsi, l'ordonnanceur exécute à tout moment la plus prioritaire des tâches actives.

La problématique d'ordonnancement sur les systèmes qui récupèrent l'énergie de l'environnement est de trouver des algorithmes performants ainsi que les outils d'analyses d'ordonnançabilité associés. L'algorithme d'ordonnancement est responsable de produire un ordonnancement perpétuel valide où toutes les contraintes temporelles et énergétiques sont respectées. En d'autres termes, les échéances de toutes les tâches doivent être respectées et le niveau du réservoir d'énergie ne doit jamais descendre en dessous de son seuil minimal. Une analyse d'ordonnançabilité fournit des tests et des conditions qui garantissent l'ordonnaçabilité d'un système de tâches donné dans une configuration d'énergie donnée. De plus, pour garantir une telle ordonnançabilité, la capacité du réservoir d'énergie doit être suffisante pour satisfaire sans perte la demande d'énergie des tâches. Pour cela, on doit pouvoir calculer pour chaque algorithme et pour chaque condition d'ordonnançabilité la taille minimale du réservoir associée.

Cette thèse commence par un bref état de l'art de la théorie existante de l'ordonnancement temps réel des systèmes décrits plus haut. Cela inclut deux approches : la première consiste à introduire des temps d'inactivité supplémentaires afin de recharger, la deuxième consiste à baisser la fréquence et/ou la tension du processeur afin de réduire le taux de consommation. Dans cette partie, Nous montrons que dans le cas de la première approche, l'ordonnancement à priorité fixe n'a pas été suffisamment étudié. Les résultats de cette thèse contribuent à apporter des éléments de réponse aux problématiques de cette famille d'ordonnancement.

La première contribution de cette thèse est la proposition de l'algorithme d'ordonnancement PFP_{ASAP}. Il s'agit d'une adaptation de l'ordonnancement préemptif classique à priorité fixe au modèle des systèmes qui récupèrent l'énergie de l'environnement. Cela consiste à exécuter les tâches au plus tôt dès que l'énergie est suffisante pour exécuter au moins une unité de temps et laisser le réservoir se recharger le cas échéant. La particularité ici est que les périodes de rechargement sont aussi longues que nécessaire pour pouvoir exécuter une seule unité de temps. Cet ordonnancement peut aussi être vu comme l'équivalent "énergie" de l'ordonnancement non-oisif à priorité fixe classique où le processeur ne peut pas être en mode inactif alors qu'il reste encore des tâches en attente et que l'énergie est suffisante pour exécuter. On prouve à travers cette dissertation que PFP_{ASAP} est optimal dans la classe d'algorithmes à priorité fixe mais uniquement dans le cas des systèmes dits non-concrets où la date de la première activation des tâches et le niveau initial du réservoir d'énergie ne sont connus qu'au moment de l'exécution, et quand toutes les tâches consomment plus d'énergie pendant leur exécution que le système n'en collecte. Une condition d'ordonnançabilité nécessaire et suffisante pour ce type de systèmes est également proposée. Il s'agit d'une analyse de temps de réponse des tâches qui calcule le plus long temps de réponse possible de chaque tâche qui se produit dans le pire scénario (l'instant critique). Nous prouvons aussi que ce scénario correspond à l'activation synchrone de toutes les tâches quand le niveau du réservoir d'énergie est au plus bas.

Malheureusement, si l'on relâche l'hypothèse sur le profil de consommation d'énergie des tâches, en considérant des tâches qui consomment plus que le rechargement et d'autres qui consomment moins, l'algorithme PFP_{ASAP} n'est plus optimal et l'activation synchrone des tâches n'est plus le pire scénario ce qui rend la condition d'ordonnançabilité précédemment citée seulement nécessaire. Pour contourner cet obstacle, nous proposons dans une seconde contribution de borner le pire temps de réponse des tâches afin de construire des conditions suffisantes au lieu des tests exacts. Nous proposons deux bornes supérieures construites sur des scénarios virtuels qui maximisent le nombre des interférences et le temps de rechargement nécessaire.

Concernant la possibilité de trouver un algorithme optimal, nous explorons à travers ce manuscrit différentes idées et approches d'ordonnancement dans le but de construire un algorithme optimal pour le modèle général des systèmes à priorité fixe en considérant tous les types de systèmes de tâches et tous les profils de consommation d'énergie. Nous montrons avec quelques contre exemples la difficulté de trouver un tel algorithme et aussi que la plupart des idées intuitives n'aboutissent pas à des algorithmes optimaux. Par la suite, nous discutons la possibilité de trouver un tel algorithme.

Dans le but de mieux comprendre la problématique d'ordonnancement à priorité fixe des systèmes collecteurs d'énergie, nous proposons d'explorer les solutions proposées pour des problématiques d'ordonnancement similaires, en particulier celles où le retardement des exécutions est parfois nécessaire pour respecter certaines contraintes. L'ordonnancement avec contraintes thermiques est l'une de ces problématiques. Cette dernière consiste à exécuter les tâches de telle sorte qu'une certaine température maxi-

male n'est jamais atteinte ou dépassée. Cette contrainte pousse le système à suspendre les exécutions de temps en temps pour rajouter des temps de refroidissement afin d'éviter que la température maximale ne soit atteinte. Comme première étape de ce travail, nous proposons dans cette thèse d'adapter les solutions proposées pour les systèmes à énergie renouvelable aux systèmes à contraintes thermiques. Ainsi, nous adaptons l'algorithme PFP_{ASAP} afin que la contrainte thermique soit respectée. Nous proposons également une analyse d'ordonnançabilité basée sur des bornes du pire temps de réponse des tâches.

Pour terminer, nous présentons *YARTISS* : l'outil de simulation développé pendant cette thèse pour en évaluer les résultats théoriques.

Introduction

Computing is one of the sciences that have revolutionized the world during the last century. Humans invented electronic computers for the first time during World War II. Using the Turing Machine model, these computers were able to perform complicated encryption/decryption computations. A computer is a device that can be programmed to perform a set of arithmetic or logical operations. Since a sequence of operations can be changed, the computer can solve more than one kind of problem.

Conventionally, a computer consists of at least one processing element, typically a **C**entral **P**rocessing **U**nit (CPU) and some form of memory. The processing element carries out arithmetic and logic operations, and a sequencing and control unit that can change the order of operations based on stored information. The first computers were the size of a large room, consuming as much power as several hundred modern **P**ersonal **C**omputers (PCs).

During the last decades, the use of computers in many applications in the modern life has increased dramatically the demand of processing power and efficiency. The progress of technology allowed modern computers based on integrated circuits to be millions to billions of times more capable than the early machines, and occupy a fraction of the space. Simple computers have become small enough to fit into mobile devices that can be powered by small energy sources. This progress allows new applications to emerge. Nowadays computers are capable to control even critical systems such as nuclear operations or airplane flight control. This kind of computers are mostly embedded in autonomous systems and can be found in many devices from MPEG Audio Layer 3 (MP3) players to spacecrafts and from toys to industrial robots.

The most important issue of these systems is determinism which means that their behavior has to be predictable prior deployment. These systems need predictability not only in term of correctness or prevision of the required results but also in term of the time that the computations take or the date at which the results are delivered. Unfortunately, nowadays computers are not deterministic and thereby they are not fully predictable. To cope with this limitation, the behavior of these systems is studied assuming the worst-case scenario. Therefore, the considered systems must deliver correct results within a bounded interval of time even in the worst-case scenario. Systems that are expected to respect such timing constraints are called ***R**eal-**T**ime **S**ystems (RTS)*.

Furthermore, with the increase of processors speed and the miniaturization of electronic devices, energy and heat management becomes one of the major issues to address. In fact, the aim of small devices is to provide autonomous services for a long lifespan like wireless sensor networks or medical implants. The challenge of such devices is to use the environmental energy to run for a very long time by eliminating maintenance operations, e.g. battery replacement. Many environmental energy sources can be exploited to achieve such an autonomy, e.g. solar, vibration, wind, thermal, etc. Each source is adapted to specific applications. For example, the solar energy can be

used for outdoor devices and the vibration energy for industrial monitoring. Systems that collect the energy from the environment and use it are called ***E**nergy-**H**arvesting **S**ystem (EHS)*.

Most of energy-harvesting systems have to execute recurrent tasks, e.g. data sensing and transmission operations, and have to guarantee a bounded delay of results delivery. In such systems, both energy and time constraints have to be respected. Indeed, when operating with a renewable energy, the system has to manage the energy collected from the source, the energy consumption of tasks, the capacity of the energy storage device and the time constraints of the associated real-time system. These systems are called ***R**eal-**T**ime **E**nergy-**H**arvesting **S**ystems (RTEHS)*.

During the last decade, the real-time behavior of energy-harvesting systems has attracted further interest. The challenge here is to find the right schedule of the real-time tasks while respecting energy limitations and time constraints. Some scheduling algorithms have been proposed in the literature but most of them focused on the **E**arliest **D**eadline **F**irst (*EDF*) rule which executes first the tasks with the earliest time constraint. However, even though *EDF* scheduling is efficient, there exists an other family of scheduling algorithms which is widely used in industry and not very well studied in this context up to now, it is the fixed-priority scheduling. In this kind of real-time scheduling, the tasks keep the same priorities all the time and the scheduling algorithm executes tasks according to their priority. This thesis addresses the problematic of this family of real-time scheduling algorithms. We study through this dissertation the different aspects of fixed-priority real-time scheduling for energy-harvesting systems. We first propose some partial solutions, namely a new scheduling algorithm and two associated schedulability analysis. Next, we explain the difficulty of finding optimal solutions for this problematic. Finally, we discuss the same scheduling problematic with a similar model, namely thermal-aware scheduling.

The remainder of this dissertation is composed of two main parts. Part I presents the necessary background of the classical real-time scheduling theory and the state of the art of real-time energy-harvesting systems. Part II details the contributions of this thesis.

Firstly, the state of the art part is structured as follows.

Chapter 1 introduces the classical real-time scheduling theory. In this chapter we define the different levels and components of a real-time system and we describe the properties and the notations used through this dissertation. Then, we review the main task models and their properties. After that we present the different families of scheduling algorithms and their associated schedulability and feasibility conditions.

Chapter 2 gives a brief overview of the different energy sources, their extraction methods and the main energy storage technologies. We first explore and compare the available technologies of energy-harvesting techniques. Then, we present a brief state of the art of the available energy storage technologies, namely batteries and supercapacitors, by showing their properties and the applications for which they are suitable.

Chapter 3 contains the state of the art of the different models and solutions available for the real-time scheduling of energy-harvesting systems. In this chapter, we start by defining and specifying the formal model of the targeted real-time energy harvesting systems that set the scope of this thesis and then, we explain the real-time scheduling problematic we are interested in, namely the fixed-priority scheduling. After that, we identify the different scheduling approaches for energy-harvesting systems, then, we list and explain the major real-time scheduling algorithms for energy-harvesting systems that have been proposed in the literature. At the end, we compare these algorithms to each other through simulations and we summarize the strengths and the weaknesses of each algorithm.

Secondly, the contributions part is organized as follows.

Chapter 4 describes in detail the **P**reemptive **F**ixed-Task-**P**riority **A**s **S**oon **A**s **P**ossible (PFP_{ASAP}) scheduling algorithm which is the first contribution of this thesis. In this chapter we first present a theoretical study of the algorithm by proving some properties like the optimality, and characterizing some aspects like the the worst-case scenario, the minimum battery capacity and the schedulability condition. Next, we compare PFP_{ASAP} to the algorithms presented in the state of the art through simulation and we discuss its limitations.

Chapter 5 presents a schedulability analysis of the PFP_{ASAP} algorithm where some assumptions are removed. This analysis is based on tasks response time approximation. In this chapter, we propose two upper bounds of tasks response time and we build two schedulability conditions with two levels of precision. Then, we validate this theoretical result with simulations.

Chapter 6 shows the difficulty of finding an optimal algorithm for fixed-priority energy-harvesting systems. We explain with counter examples why the algorithms proposed up to now and the intuitive algorithms are not optimal. We try also to build an optimal algorithm with an exponential complexity.

Chapter 7 studies the fixed-priority scheduling for thermal-aware systems which is a very close model to energy-harvesting's one. In this chapter we use the properties of the PFP_{ASAP} algorithm to build an optimal algorithm for the thermal-aware model and a schedulability analysis.

Chapter 8 presents **Y**et **A**n Other **R**eal-**Ti**me **S**ystems **S**imulator (YARTISS), the simulation tool used to perform the simulations in this dissertation. In this chapter, we present the architecture of this simulator and we show through some examples how it is extensible.

Finally, we conclude this dissertation by summarizing the contributions of this thesis and discuss the remaining open problems and the possible axes of future development.

Part I

State of The Art

Chapter 1

Real-time Scheduling

Contents

1.1 Introduction

Nowadays electronic devices are more and more used to control critical operations like nuclear reactions or flight commands. These devices are required to perform computations in order to take decisions. For most of such devices, the time at which the results of these computations are delivered is as important as their correctness.

The most important issue of these systems is determinism which means that the behavior of the considered systems can be predicted totally or partially prior deployment. These systems need predictability not only in term of correctness or prevision of the required results but also in term of the time that the computations take or the date at which the results are delivered. Systems that can guarantee such a predictability are called *Real-Time Systems (RTS)*.

Definition 1.1 (Real-Time Systems)**.**
In computer science, a system is considered as Real-Time if the correctness of the system depends not only on the logical result of the computation but also on the time at which the result is delivered [But11]. ■

Although the term of real-time is frequently used in many application fields, it is subject to different interpretations. One can say that an application operates in real-time if it is able to quickly react to external events. For example, one can use *real-time video streaming* to refer to a video that can be produced and viewed on the flight. This has become possible thanks to the increasing power of computers and telecommunication media. According to this interpretation, a system is considered to be real-time if it is fast. However, the term *fast* has relative meaning and does not fill the main property that characterizes a real-time system which is time predictability. The considered systems are often subject to environment interaction. Thus, it does not make sense to design a real-time computing system for flight control without considering the timing characteristics of the different critical computation activities of the aircraft.

Some people erroneously believe that it is not worth to invest in real-time research because advances in computer hardware will take care of any real-time requirement. Although this advance will improve processing speed, this does not guarantee that the timing constraints of an application will be met. In fact, while the aim of fast computing is to minimize the average response time of a given set of tasks, the aim of real-time computing is to meet the individual timing requirements of each task worst-case scenario. Hence, instead of being fast, a real-time system should be predictable. A safe way to achieve predictability is to investigate and employ new methodologies at every stage of the development of an application, from design to testing.

The main difference between a real-time and a non-real-time task is that a real-time task is characterized by a temporal constraint: the deadline, which is the maximum time before which the task must complete its execution. In critical applications, a result delivered after the deadline is not only late but wrong. Regardless of the application, a well-designed real-time system should eliminate or minimize temporal constraint violations.

Depending on the consequences that may occur because of a missed deadline, real-time systems are usually classified in two families *hard* and *soft*.

In a *hard real-time system* no deadline misses are tolerated because the penalty or the damage caused by missing a deadline may be catastrophic, e.g., danger for human health or the environment. For a hard real-time system to be temporally correct, each computation or task must successfully complete prior its deadline whatever the scenario the system can cross. The system must be tested in the worst-case scenario. Finding the worst-case scenario by testing all the possible scenarios may not be possible in practice. Instead, formal analysis techniques are necessary to ensure that the considered system is correct and predictable.

In contrast, in a *soft real-time system*, missing a deadline does not compromise the safety or the integrity of the system but degrades the **Q**uality **o**f **S**ervice (QoS). Therefore, the goal of a soft real-time system is to maximize the quality of service by minimizing deadline violations. In this kind of real-time systems, formal analysis can be applied to prove a certain quality of service instead of feasibility.

For a system to be proven temporally correct, three aspects of a real-time system must be considered in the formal analysis:

1. *Real-Time Workload*: the computation performed by the real-time system that must complete before its deadline. Usually, the workload is modeled using the notion of *recurring tasks*. A recurring task requests the execution of infinite sequential pieces of code called *jobs*. Each requested job is associated to a deadline.

2. *Processing Platform*: the set of hardware resources where the jobs of the workload are executed. These resources include the processor(s) (CPUs), the memories, the caches and the interconnection between them, etc. The architecture of the platform is very important because its performance influences directly the temporal behavior of the system.

3. *Scheduling Algorithm*: if the workload is composed of one task, there is not a scheduling problem because there is no concurrence between tasks. However, when the system is composed of several tasks with different deadlines, many jobs may by ready for execution at the same time and need to be executed as soon as possible to meet their deadlines. In this case, a scheduling algorithm is needed to decide, at any time, which jobs are executed on the processing platform. The choice of the scheduling policy is one of the main factors that impacts the temporal behavior of the system.

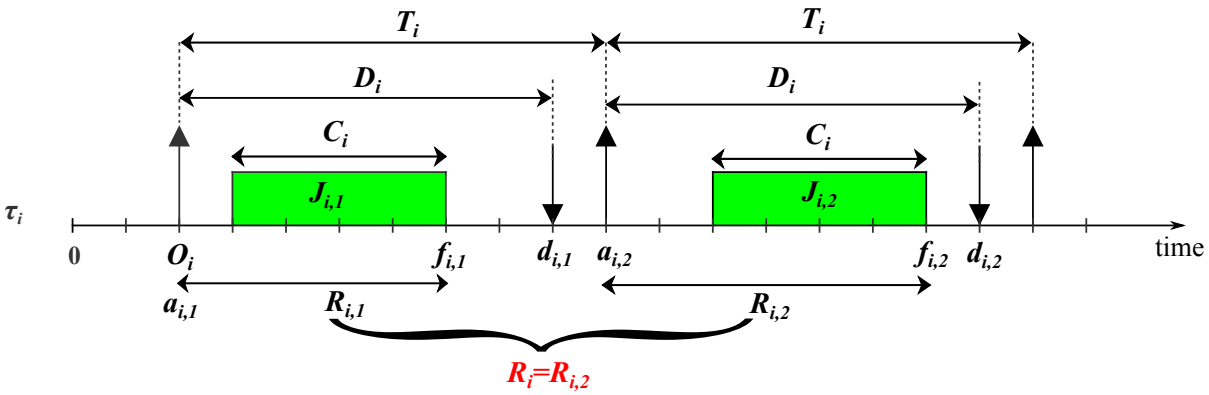


Figure 1.1: Task and job parameters

The aim of this chapter is to present the main concepts and definitions needed to understand the contributions of this dissertation. The remainder of this chapter is organized as follows, Section 1.2 formally describes the common models of real-time workload. Section 1.3 introduces briefly the processing platforms considered in this dissertation. Then, Section 1.4 enumerates the most common scheduling algorithms for real-time systems and some of their properties. Finally, the main formal analyses of the temporal correctness of real-time systems are presented in Section 1.5.

1.2 Workload Model

To ensure the predictability required for real-time systems, it is primordial to specify the parameters and the hypotheses of the considered systems. The task model or the workload model is the first aspect to be specified for a real-time system. It consists of setting the temporal parameters of each task, the relationship between them and the assumptions of the model.

A *task* is a software entity that performs computation for a specific function, e.g. checking sensors values or activating actuators. It consists of executing a sequence of instructions on the processing platform. These sequence of instructions provided by the task can be executed multiple times.

Most of real-time tasks have a cyclic structure in a way that they execute some code and then they block waiting for a periodic or a sporadic timer. For this reason, real-time tasks are usually modeled as a sequence of recurrent *jobs*. Hence, a *job* is in instance of a real-time task associated to a temporal deadline relative to its arrival time.

Since the seminal paper addressing real-time recurrent tasks, i.e. the work of Liu and Layland in [LL73], a real-time task τ_i is characterized by the following parameters.

- **Offset** O_i: the first time where task τ_i is activated, it coincides with the request

time of the first job of the task. It is also called *start time* or *first release time*. When all the tasks composing the system are started at the same time, i.e. $\forall i\ O_i = 0$, we say that the tasks are *synchronous*, and *asynchronous* otherwise,

- **Execution Time** C_i: the computation time required by each job of τ_i. Tasks may have variable execution times between different jobs. This is due to the fact that the path and the duration of instructions executions depend on many parameters like the type of the used memory (structure of caches) or the branches resulting of conditional instructions and loops. This makes the problem of keeping a constant execution time very difficult. Some studies have observed that effective execution times can vary up to 87% relative to their worst-case execution time [MW98]. This variation may compromise the global temporal predictability of the system when dealing with hard real-time systems. To cope with this problem, the notion of ***W**orst **C**ase **E**xecution **T**ime (WCET)* was proposed, it consists of considering only the longest possible execution time.

  In order to use, the WCET, the must guarantee some sustainability properties which means that if it works in the worst-case it must work in a more favorable case as well. Therefore, it is safe to study the real-time system with the WCET.

  The WCET is actually computed with different methods, it can be estimated simply by getting the longest value given by empirical measurement on the targeted platform, e.g. benchmarks, or formally by analyzing the code of the task to find the longest path of possible branches and loops in the worst state of memory and input data size. The computation of the exact WCET is one of the most important and active research fields within the real-time scheduling community. Many contributions are yielded every year to improve WCET predictability [HP13; Now+14; BD13]. Nevertheless, in soft real-time systems, the average execution time is sometimes used instead of WCET to make the formal analysis less pessimistic,

- **Period** T_i: this parameter models the recurrent aspect of real-time tasks, it represents the inter-arrival time between two successive jobs. According to the considered task model, this parameter can be fully specified with a constant value, partially with a lower bound, or not specified at all. The periodicity model has an important impact on the complexity of the scheduling problem, this point will be discussed in Section 1.5.2 on page 43,

- **Relative Deadline** D_i: the time allocated to task τ_i to complete a job execution. Each job has exactly D_i time units to finish executing after being activated. Exceeding this amount of time, the job violates its temporal constraint. The deadline is the parameter that characterizes real-time systems because it models their temporal constraints. A relative deadline is not an absolute time, it is the difference between a job's activation time and its absolute deadline. Relative deadline is usually associated to tasks and absolute deadline to jobs,

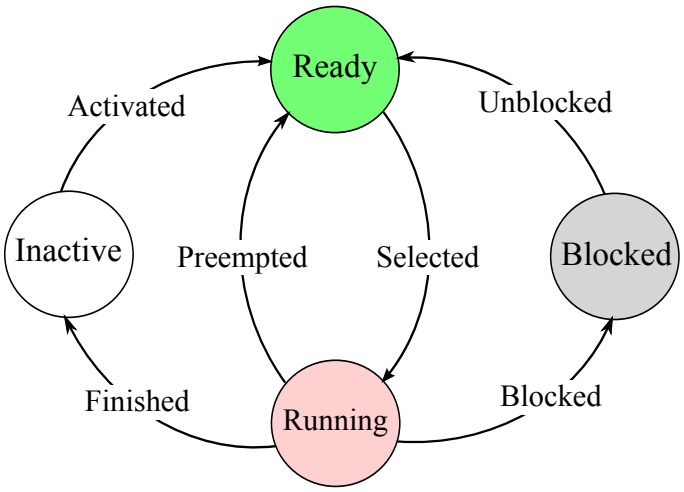


Figure 1.2: Task life cycle

- **Worst Case Response Time** R_i: the longest job response time of task τ_i, i.e. $R_i = max_{\forall j}(R_{i,j})$, where $R_{i,j}$ is the response time of the j^{th} job of task τ_i, i.e. the amount of time needed to complete its execution including potential blocking or waiting times caused by interferences and preemptions. Formally, it is the difference between the job's activation time and its termination time.

Regarding jobs, $J_{i,j}$ denotes the j-th job of task τ_i and is characterized with the following parameters.

- **Arrival time** $a_{i,j}$: the time at which job $J_{i,j}$ becomes ready for execution, it is also called request time or release time. This time depends on the periodicity model of the task,
- **Absolute Deadline** $d_{i,j}$: the effective time of the job's deadline, it can be obtained by adding D_i to its arrival time,
- **Termination time** $f_{i,j}$: the time at which job $J_{i,j}$ finishes its execution,
- **Response time** $R_{i,j}$: is the difference between job's completion time and its request time, i.e. $R_{i,j} = f_{i,j} - a_{i,j}$.

Figure 1.1 on page 30 illustrates these parameters.

1.2.1 Task Life Cycle

During the life span of a real-time system, multiple jobs of a task can be released. At a given moment of a task life cycle, it can be in one of the following states:

- ***Ready***: refers to the state in which the job has been activated but not yet executed.

- ***Running***: refers to the state in which the job is being executed on the processing platform.
- ***Blocked***: refers to the state in which the job is stopped by the system. This can happen in some real-time systems where there is for example shared resource with mutual exclusion, the job is stopped until it gets the access to the resource.
- ***Inactive***: in this state the job is not yet requested or has already finished its execution.

Figure 1.2 on the facing page shows the transitions between these states.

In some simple real-time systems, it may be possible to completely specify the parameters of each job prior to system run-time (i.e. the system designer has complete knowledge of each job $J_{i,j}$). However, in systems with a large (or infinite) number of jobs or systems that have dynamic behaviors, explicitly specifying each job, prior to system run-time, may be impossible or unreasonable. Based on this characteristic, we can classify task models as either completely specified or partially-specified. We now discuss the differences between these two types of systems.

1.2.2 Completely-Specified Recurrent Task Model

A real-time system is completely specified if the exact values of the temporal parameters and constraints of all jobs can be determined off-line or prior to system run-time. Formally, when the arrival-time and the absolute deadline of each job are known off-line. This implies that jobs offsets and inter-arrival times can also be determined prior to run-time. Typically, completely-specified task models are appropriate for hard real-time applications that have completely predictable executions. For example in an avionic control system, the control system will process the pilot's input command at strict time intervals to ensure that flight control does not degrade. Completely-specified systems are sometimes called *concrete* systems.

Periodic Task Model

In many real-time control applications, periodic activities represent the major computational demand in the system. Periodic tasks typically arise from sensory data acquisition, control loops, action planning and system monitoring. Such activities need to be cyclically executed at specific rates which can be derived from the application requirements. The *periodic task model* was proposed for the first time by Liu and Layland in 1973 [LL73]. It allows the specification of homogeneous sets of jobs that occur at strict intervals. A periodic task τ_i is specified by the temporal parameters described earlier, i.e. (O_i,C_i,T_i,D_i), where the inter-arrival time between two successive jobs T_i is set to a constant value. The first job of task τ_i is released at O_i, the second at $O_i + T_i$ and the j-th one at $O_i + (j-1) \times T_i$. Then, the time of jobs arrival and

their deadlines are fully predictable. Periodic tasks are also known as time-triggered tasks because their jobs arrival times are determined only by time.

1.2.3 Partially-Specified Recurrent Systems

For many real-time systems, it is not possible to know beforehand how many jobs will be generated by the system during run-time. Furthermore, completely specified systems such as periodic task systems are not able to handle changes of real-time workloads because of the restrictive constraint that jobs must be released at strict periodic intervals. Thus, for systems where the arrival times between jobs could change dynamically, e.g. packets in a network application, the periodic model may not be suitable. To cope with the inflexibility of completely-specified systems, one may instead consider partially-specified ones. The execution of the same partially-specified task set for the same period of time may lead to different workload with different job arrival times. Nevertheless, the specification of a partially-specified system includes a set of constraints that any execution must satisfy in order to ensure predictability. Partially-specified task systems are sometimes called *non-concrete* systems.

In this subsection, we will introduce more general models for partially-specified task systems: the sporadic and the aperiodic models. These general task models can be used to represent more complex applications than the restrictive periodic task model.

Sporadic Task Model

The sporadic task model proposed by Liu and Layland in [LL73] removes the restrictive assumption of the periodic task model where the jobs of a task are released at a strict periodic time interval. In addition, the offset parameter O_i is not specified and the inter-arrival parameter is lower bounded. Then, a sporadic task τ_i can be characterized by a tuple (C_i,T_i,D_i) where C_i is its worst-case execution time, D_i its relative deadline and T_i is the *minimum inter-arrival time* between two successive jobs of τ_i, i.e. $\forall\ (J_{i,j}, J_{i,j+1})$, $a_{i,j+1} - a_{i,j} \geq T_i$ (note that T_i denoted the exact inter-arrival time for periodic tasks). Then, at run-time, the number of jobs released during the same interval of time can vary according to dynamic inter-arrival time of jobs.

Aperiodic Task Model

Unlike periodic and sporadic task models that set some hypothesis on jobs inter-arrival times, the aperiodic task model removes completely the notion of period or inter-arrival time. The jobs can arrive at any time as well as non-predictable events. For this reason, aperiodic task systems are also called event-triggered systems. Then, an aperiodic task is characterized only with two parameters, namely the WCET, C_i, and the deadline D_i. Even though the aperiodic task model is the most general one, it is the less predictable model. Indeed, when dealing with unspecified jobs inter-arrival times, the time analysis of tasks worst-case response time is impossible because the amount

of workload within any time interval is neither known nor bounded which prevents the behavior of the system to be temporally predictable. Therefore, the aperiodic task model is not suitable for hard real-time systems that need a full temporal predictability. However, it could be very interesting to model some soft real-time applications that aim to increase the utilization of platforms with a certain QoS or mixed systems that are composed of periodic, sporadic and aperiodic tasks.

1.2.4 Tasks Independence

One of the most common hypotheses in real-time systems is *tasks independence*. This means that the only resource on which tasks are in concurrency is the computation units, i.e. the processors. Nevertheless, in many real-time applications, the tasks composing the system may be directly or indirectly dependent to other tasks. Indeed, tasks may be subject to input data dependency constraints, i.e. a task τ_1 may require the results produced by an other task τ_2, and thus, τ_1 cannot start executing until τ_2 has not yet finished. Dependencies between tasks could be modeled with a graph of jobs, e.g. Directed Acyclic Graphs (DAG) task model [Bar+12]. Furthermore, tasks dependency can also come from resource sharing (other resources than the computation units). Indeed, when some resources that cannot be accessed simultaneously are shared between tasks, the task that holds the resource blocks the execution of other tasks. Then, the execution duration of the waiting tasks depends on how long the task holding the resource will keep it. Therefore, concurrency and resource sharing may lead to indirect dependency between tasks.

In the case of several shared resources, deadlocks can occur if tasks are waiting for each other. This leads to a concurrency issue which is one of the widely discussed topics within the real-time systems community [But11]. The common solutions consist of bounding the blocking time and including it in the WCET.

The remainder of this dissertation focuses only on types of systems that are predictable and can be formally verified. We study especially periodic and sporadic independent systems for the basic model, then, other constraints and parameters will be added in order to study real-time scheduling for energy-harvesting systems.

1.3 Processing Platform

In contrast with real-time workload which is the software part of real-time systems, the processing platform is the computer hardware where real-time applications are executed. It consists of a collection of physical elements that constitutes the computer responsible of running the application such as the processor or the CPU, Input/Output devices, the different levels of memory and caches, etc. It is crucial for real-time applications design to consider the different components of the processing platform because they are part of the model and their performance and architecture influence directly the temporal behavior of the system.

The process of determining a task worst-case execution parameters is called timing analysis. Timing analysis must account for worst-case cache behavior, memory access time, program structure and worst-case execution paths.

The analysis for determining the contribution of each of these factors to the worst-case execution time depends on the specific system and the program. Other factors that can increase the worst-case execution time are job preemptions (i.e. a job is suspended while a different job is executing, and then, its execution is resumed at a later time). The context switch and scheduling algorithm overhead contribute to increase jobs response times. The total preemption cost is typically dependent on the processor architecture and the used scheduling algorithm.

In this dissertation, we will assume that the worst-case execution time of each task has already been determined. We will consider only monoprocessor platforms that has a unique speed. We will assume also that jobs are preemptable with a negligible cost.

1.4 Scheduling Algorithms

The role of a real-time scheduling algorithm within a real-time application is to determine which active jobs are executed on the processing platform at every time instant. In other words, it determines the intervals of execution for jobs on the processing platform. The sequence of execution intervals of a task set is known as a schedule. The goal of a real-time scheduling algorithm is to produce a schedule that ensures that every job is allocated processor slots (i.e. executes) to finish executing before its deadline.

In this section, we discuss the classification of real-time scheduling algorithms.

1.4.1 Online/Offline Scheduling

Scheduling real-time jobs is the responsibility of the scheduler, which can be part of the software layer that composes the real-time operating system or part of the hardware layer. The scheduler can be seen as a specific task that takes scheduling decisions by following a scheduling policy or a scheduling algorithm.

If the sequence of real-time jobs is specified prior to run-time or generated by a completely-specified task set, a scheduling algorithm can generate and store the schedule prior to run-time. This approach is called *static* or *offline* scheduling. In contrast, for systems that are partially-specified or have a schedule too large to be stored in the system's memory, an *online* algorithm is appropriate for this case. At time t, the online real-time scheduling algorithm decides the set of jobs to execute at time t based on the status of jobs released at or prior to t. An online scheduling algorithm does not have specific information about the release of jobs after time t (i.e. future jobs arrival times are unknown). This dissertation focuses on deterministic online real-time scheduling algorithms.

1.4.2 Work-Conserving/Non-work-conserving Scheduling

A real-time scheduling algorithm can produce a *non-work-conserving* schedule, i.e. a schedule where the processor is inactive while at least one job is ready for execution. This may seem as a counter-productive property, however, it can be very useful for adaptive scheduling algorithms that may delay jobs execution to avoid future failures as for energy-harvesting systems. In contrast, a *work-conserving* schedule does not allow the processor to be inactive while ready jobs are waiting for execution.

1.4.3 Preemptive/Non-Preemptive Scheduling

When several tasks are ready to be executed at the same time on a monoprocessor platform, two approaches can be applied: the *preemptive* or the *non-preemptive* approach. In the non-preemptive approach, when a task starts executing, it cannot be interrupted until it finishes its execution. With this approach, there is at most one context switch, i.e. the scheduler replaces the execution context of the finishing job with starting one. These context switches have a cost that has to be considered in the schedulability analysis, although the hypothesis of an negligible cost is often considered. In a preemptive scheduling, the scheduler can preempt a job to execute an other one. The advantages of this approach are reducing the response time of higher priority jobs, a best use of the processor and better rate of schedulability. With this approach, jobs are executed concurrently on the processor. In this dissertation, we focus only on preemptive scheduling algorithms.

1.4.4 Optimal Scheduling

The term of *optimality* in real-time theory is used for different properties. Here we are interested in the optimality of scheduling algorithms form schedulability point of view. For this sake, we use the definition proposed by Buttazzo in [But11]:

"A real-time scheduling algorithm is said to be optimal for a specific class of algorithms if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a valid schedule, then an algorithm is said to be optimal if it always finds a valid schedule whenever there exists a valid one. In other words, an algorithm is said to be optimal if it may fail to produce a valid schedule for a given task sets only if no other algorithms of the same class can do it."

In contrast, a non optimal algorithm may fail to schedule a given task set while a valid schedule exists.

Furthermore, the schedulability of a task set according to an optimal algorithm is a necessary and sufficient feasibility condition for the considered class of algorithms. See Section 1.5 on page 40 for more details.

1.4.5 Priority Driven Scheduling Algorithms

A possible approach for an online scheduling algorithm is to assign, at any given time t, each job $J_{i,j}$ a priority $P_{i,j}$ (which is assumed to be a positive integer). A priority-driven scheduling algorithm sorts at each time t the active jobs according to $P_{i,j}$ (in non-decreasing order) and schedules the highest-priority job on the processor.

In this subsection, we describe the classical priority-driven scheduling algorithms. These later differ in the manner that they assign a priority to jobs. In the following, we give three classifications of priority-driven scheduling algorithms. We follow the classification names used in [FBB06]. The three major classes of priority-driven scheduling algorithms are **F**ixed-**T**ask-**P**riority (FTP), **F**ixed-**J**ob-**P**riority (FJP), and **D**inamic **P**riority (DP).

1.4.6 Fixed Task-Priority Scheduling Algorithms (FTP)

In fixed task-priority scheduling, each task is assigned a static or fixed priority P_i prior to run-time and keeps the same value during run-time. Each job generated by that task inherits the same priority value. Thus, for a real-time system with n tasks, there are n distinct priorities (one for each task). We assume that the tasks are indexed in a non-decreasing order of priority. Therefore, for a task set $\Gamma=\{\tau_1, \ldots, \tau_n\}$, τ_1 has the highest priority and τ_n has the lowest one. In general, the task-priority assignment can be determined by the system designer. However, there are three well-studied task-priority assignment policies for sporadic task systems: **R**ate **M**onotonic (RM), **D**eadline **M**onotonic (DM) and Audsley's **O**ptimal **P**riority **A**ssignment (OPA).

Task Priority Assignment Policies

The tasks priority assignment has a significant effect on the schedulability of task sets in fixed-task-priority scheduling [Dav+13]. Hence, it is important to select the right priority assignment policy. In the following we describe briefly the main priority assignment policies.

Rate Monotonic: For rate monotonic priority assignment policy [LL73], each task τ_i is assigned a priority according to its period. The shorter the period is, the higher the priority of the task is. Rate Monotonic is optimal for non-concrete task sets with implicit deadlines, i.e. $O_i = 0$ and $D_i = T_i$. The intuition behind this optimality is that tasks can use all the available time because of the implicit deadlines and that high frequency tasks cannot support a lot of interferences in contrast of low frequency tasks. Then, in this case, assigning high priorities to high frequency tasks is the best that we can do. This is proved in [LL73].

Deadline Monotonic: The deadline monotonic policy [LL73] assigns to each task τ_i a priority according to its relative deadline parameter: the shorter the deadline is, the

Algorithm 1.1 Optimal Priority Assignment

```
for all priority level k, lowest first do
    for all unschedulable task τ_i do
        if τ_i is schedulable at priority k with all other unassigned tasks assumed to higher priorities then
            assign τ_i to priority k
            break (continue outer loop)
        end if
    end for
    return unschedulable
end for
return schedulable
```

higher the priority of the task is. Deadline Monotonic is optimal for non-concrete task sets with constrained deadlines [LL73], i.e. $O_i = 0$ and $D_i \leq T_i$. Intuitively this can be explained by the fact that with constrained deadlines tasks with short deadlines can support less interferences than ones with larger deadlines. Then, in this case, assigning high priorities to tasks with short deadlines is the best that we can do. The detailed proof is available in [LL73].

Optimal Priority Assignment: Note that the optimality of rate monotonic and deadline monotonic relies on two strong assumptions, namely tasks offsets and deadlines model. If we remove these assumptions RM and DM are no longer optimal for fixed-priority scheduling. To cope with this problem, Audsley proposed in [Aud91; Aud01] an optimal priority assignment OPA algorithm that provides an optimal priority assignment for sporadic task sets with arbitrary-deadlines and concrete offsets. It consists of testing greedily different priority assignments using a necessary schedulability test as shown in Algorithm 1.1. This approach requires at most $n(n+1)$ tests to determine a schedulable priority assignment whenever such an ordering exists. This tests much less priority assignment than $n!$ that would be otherwise required, using a brute force approach that checks every possible combination.

For a schedulability test to be OPA-Compatible [DB09], it must respect the following rules:

1. the schedulability of a task may, according to the test, be dependent on the set of higher priority tasks, but not on their relative priority ordering,

2. the schedulability of a task may, according to the test, be dependent on the set of lower priority tasks, but not on their relative priority ordering,

3. when the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority cannot become unschedulable according to the test, if it was previously schedulable at the lower priority.

1.4.7 Fixed Job-Priority Scheduling Algorithms (FJP)

For fixed job-priority scheduling, the restriction that a task's jobs have identical priority is removed. Instead, each job $J_{i,j}$ is assigned a single priority $P_{i,j}$ that does not change during its execution. The specific FJP scheduling algorithm determines the priority assignment for jobs. *Earliest-Deadline First* EDF is a well known FJP scheduling algorithm proposed by Liu and Layland in [LL73]. It assigns a priority to each job according to its absolute deadline. The earlier the deadline of the job is, the higher its priority is. In other words, EDF schedules among the set of jobs with remaining execution the job with the nearest deadline.

1.4.8 Dynamic Priority Scheduling Algorithms (DP)

The dynamic priority Scheduling algorithms classification is the most general. It removes the restriction that a job priority does not change. A job $J_{i,j}$ priority $P_{i,j}$ can now vary over time. A well-known example of DP scheduling algorithm is **L**east **L**axity **F**irst (LLF). At time t, the LLF scheduling algorithm assigns a priority to each active job according to to its laxity. The shorter the alxity of the job is, the higher the priority of the job is. The laxity of a job $J_{i,j}$ at time t is $(d_{i,j} - t) - c_{i,j}(t)$, where $c_{i,j}(t)$ is remaining processor cost of job $J_{i,j}$ at time t.

In this dissertation, we focus only on FTP scheduling for energy-harvesting systems.

1.5 Scheduling Analysis

To ensure the temporal correctness of a real-time system, it must be validated prior to run-time using formal verification techniques. These techniques must ensure that for all executions of the system, all jobs generated by the system will meet their deadlines. Two fundamental analyses in formal verification for real-time systems are considered: *feasibility analysis* and *schedulability analysis*. The feasibility analysis determines whether there exists a valid schedule of the system where all jobs meet their deadlines irrespective to the used scheduling algorithm. The schedulability analysis determines whether a given scheduling algorithm will always meet all jobs deadlines.

The remainder of this section first sets some definitions then it introduces the main feasibility and schedulability analysis techniques applied to the previously mentioned scheduling algorithms.

1.5.1 Definitions

In this subsection we define some properties and quantities to simplify the understanding of the schedulability analysis that follows.

Definition 1.2 (Valid Schedule)**.**
The schedule S produced by scheduling algorithm A is considered as valid if and only if all the deadlines are infinitely met. ■

Definition 1.3 (Worst-Case Scenario)**.**
The worst-case scenario of task τ_i is the configuration of the system that leads to the longest response time of τ_i. This configuration includes tasks parameters such as first release times and periods. ■

Definition 1.4 (Task Processor Utilization)**.**
The processor utilization of task τ_i denoted U_i is the ratio of time spent in the execution of τ_i's jobs. It can be obtained by Equation 1.1.

$$U_i = \frac{C_i}{T_i} \tag{1.1}$$

■

Definition 1.5 (System Processor Utilization)**.**
The system processor utilization denoted U is the ratio of time spent in the execution of the whole task set. It is the sum of all tasks processor utilization.

$$U = \sum_{i=1}^{n} U_i \tag{1.2}$$

■

Definition 1.6 (Task Density)**.**
The task density denoted Λ_i is an upper bound of the ratio of time spent in the execution of task τ_i jobs.

$$\Lambda_i = \frac{C_i}{\min(D_i, T_i)} \tag{1.3}$$

■

Definition 1.7 (System Density)**.**
The system processor density denoted Λ is an upper bound of the percentage of time spent in the execution of the task set. It is the sum of all tasks processor densities.

$$\Lambda = \sum_{i=1}^{n} \Lambda_i \tag{1.4}$$

■

Definition 1.8 (Task Workload Function)**.**
The workload of task τ_i within time interval $[t_1, t_2]$ denoted $W_i(t_1, t_2)$ is the amount of processor time needed to execute the jobs of τ_i that are requested within time interval $[t_1, t_2]$ or are pending at time t_1. ■

Definition 1.9 (System Workload Function)**.**
The workload of a set of tasks Γ within time interval $[t_1, t_2]$ denoted $W(t_1, t_2)$ is the sum of workload of all tasks composing Γ within $[t_1, t_2]$.

$$W(t_1, t_2) = \sum_{\tau_i \in \Gamma} W_i(t_1, t_2) \tag{1.5}$$

■

Tasks workload functions depend on the used scheduling algorithms. In the literature there are two main task workload functions usually used in schedulability and feasibility analysis: the request bound function and the demand bound function.

Definition 1.10 (**R**equest **B**ound **F**unction (RBF))**.**
The request bound function of recurrent task τ_i denoted $RBF_i(t)$ computes the worst cumulative workload generated by τ_i's jobs that are *requested* during time interval $[0, t[$. Lehoczky et al. [JLY89] proposed Equation 1.6 to compute $RBF_i(t)$.

$$RBF_i(t) = \max\left(\left\lceil \frac{t - O_i}{T_i} \right\rceil, 0\right) \times C_i \tag{1.6}$$

■

Definition 1.11 (**D**emand **B**ound **F**unction (DBF))**.**
The demand bound function of recurrent task τ_i denoted $DBF_i(t)$ is the worst cumulative workload generated by τ_i's jobs that are *requested and finished* during time interval $[0, t]$. Baruah et al. [BHR90a] proposed Equation 1.7 to compute $DBF_i(t)$.

$$DBF_i(t) = \max\left(\left\lfloor \frac{t - O_i - D_i}{T_i} \right\rfloor + 1, 0\right) \times C_i \tag{1.7}$$

■

Definition 1.12 (Busy-Period)**.**
A busy-period is an interval of time $[t_1, t_2]$ where the processor is continuously executing jobs while there are waiting jobs. It starts from time t_1 when the processor is idle for the last time and ends at time t_2 when all waiting jobs are finished. It is possible to have consecutive busy-periods since it depends on the waiting jobs and not only the processor. ■

Definition 1.13 (**H**yper-**P**eriod (HP))**.**
The hyper-period of a task set Γ is the smallest interval of time after which the global periodic pattern of all the tasks are repeated. It is typically defined as the **L**east **C**ommon **M**ultiple (LCM) of the periods of all the tasks of the system. ■

The aim of scheduling analysis is to provide conditions and tests that help designers to determine prior run-time whether a task set is feasible/schedulable or not. We can distinguish three kinds of conditions: necessary conditions, sufficient conditions and necessary and sufficient.

Definition 1.14 (Necessary Condition)**.**
A necessary condition means that a task set that fails the test is definitely not feasible or not schedulable. Though satisfying the condition does not mean that it is certainly feasible, more verification is needed to conclude. ■

Definition 1.15 (Sufficient Condition)**.**
A sufficient condition means that a task set that succeeds the test is definitely feasible or schedulable. In contrast, failing the test does not mean that task set is certainly not feasible or not schedulable, more verification is needed to conclude. ■

Definition 1.16 (Necessary and Sufficient Condition)**.**
A sufficient and necessary condition is an exact test that tells with certainty whether the task sets is schedulable/feasible or not. ■

1.5.2 Feasibility or Schedulability

These two notions have been defined differently in the literature. Some times the term *schedulable* is used to describe the property *feasible* and vice versa. Here, we use the definitions proposed by Davis and Burns in [DB11].

A feasibility test is a condition that tells whether at least one valid schedule exists for the given task set or not. This schedule may require a non-clairvoyant scheduler or a clairvoyant scheduler which does not necessarily exist. Feasibility conditions are independent form scheduling algorithms.

In contrast, a schedulability condition is a test that tells whether a valid schedule can be produced by the given algorithm for the given task set or not. In this case, the algorithm must exist.

Note that a feasibility test is more general than a schedulability one. A task set that is schedulable according to a specific algorithm, is necessarily feasible. However, the opposite way is not true: a task set can be feasible but not schedulable with a given algorithm. Indeed, a schedulability test becomes a feasibility one when the used algorithm is optimal because if a solution exists, the optimal algorithm will find it.

Many feasibility and schedulability conditions was proposed in the literature. Most of them are applicable only for specific kinds of task sets with specific assumptions. For example, the relationship between periods and deadlines or tasks first release times. Therefore, it is very important for designers to check the assumptions before using a test. In the following we list some existing tests for fixed-priority and dynamic-priority scheduling.

1.5.3 Schedulability Analysis for Fixed-Task-Priority

This subsection presents the classical feasibility and schedulability conditions for fixed-task-priority scheduling.

Maximum utilization feasibility test

One of the possible ways to check the feasibility or the non feasibility of a task set is to compare its processor utilization to the maximum one that one processor can support or to lower bound the utilization of feasible task sets. It is impossible to schedule a task set with a utilization higher than 1 on one processor whatever are deadlines and offsets models because the time needed to execute the workload of any time interval $W(t_1, t_2)$ is greater than the available time, i.e. $t_2 - t_1$. Therefore a necessary feasibility condition for monoprocessor platforms can be obtained with Equation 1.8.

$$U \leq 1 \tag{1.8}$$

Liu and Layland bound

For task sets composed of n periodic tasks with *implicit deadlines* and Rate Monotonic priority assignment, Liu and Layland proposed in [LL73] a sufficient schedulability condition by lower bounding the utilization of feasible task sets. This can be obtained by Equation 1.9.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(2^{1/n} - 1\right) \tag{1.9}$$

This test can be extended to task sets with *constrained deadlines* and *deadline monotonic* (DM) priority assignment by comparing the bound to the density instead of the utilization as shown in Equation 1.10.

$$\Lambda = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n \left(2^{1/n} - 1\right) \tag{1.10}$$

However, such a test is very pessimistic because it assumes that periods are shorter than the actual ones. The maximum achievable utilization or density with Liu and Layland bound can be obtained by computing its limit when n tends to infinity as show in Equation 1.11.

$$\lim_{n \to \infty} n \left(2^{1/n} - 1\right) = \ln 2 \simeq 0.69 \tag{1.11}$$

Hyperbolic bound

For task sets composed of n periodic tasks with *implicit deadlines* and Rate Monotonic priority assignment, Bini et al. proposed in [BBB03] a sufficient feasibility condition by refining Liu and Layland bound. This can be obtained by Equation 1.12.

$$\prod_{i=1}^{n} (U_i + 1) \leq 2 \tag{1.12}$$

We can also use this bound for task sets with *constrained deadlines* by using tasks densities Λ_i rather than utilizations, however, it is very pessimistic.

Worst-Case Response Time Analysis

Recall that except the maximum utilization feasibility test, all the above conditions are only sufficient. Then, a task set that fails these tests or does not fill the assumptions is not definitely unfeasible. To cope with this problem, the **W**orst **C**ase **R**esponse **T**ime (WCRT) analysis was proposed in the literature. Pandya et al. showed in [MPK86] that a necessary and sufficient schedulability test can be obtained by checking if the WCRT of each task is lower than its relative deadline, as expressed by Equation 1.13.

$$\forall\, \tau_i \in \Gamma,\ WCRT_i \leq D_i \tag{1.13}$$

For non-concrete task sets with *constrained or implicit deadlines* and deadline monotonic priority assignment, Audsley et al. proposed in [Aud+93] an algorithm to compute the worst-case response time of tasks that can be used to build a necessary and sufficient test. Equation 1.13 describes how to compute the WCRT of priority level-i. The termination time of the first job of task τ_i which coincides with the WCRT in the synchronous scenario can be computed with a fixed point iteration on Equation 1.14 using the iterative workload function W_i^m.

$$\begin{cases} W_i^0 & = & C_i \\ W_i^{m+1} & = & C_i + \displaystyle\sum_{j<i} RBF_j(W_i^m) \\ WCRT_i & = & \displaystyle\min_{m>0 \,\wedge\, W_i^{m+1}=W_i^m} (W_i^m) \end{cases} \tag{1.14}$$

Note that this algorithm assumes that all the tasks are requested simultaneously which is the worst-case scenario for Liu and Layland model. This scenario is also called the critical instant. Furthermore, this algorithm checks only the deadline of the first job. This is sufficient because first, the test considers only constrained or implicit deadlines which avoids the interference between the jobs of the same task, and second, the longest response time happens necessarily in the worst-case scenario. If the first job meets its deadline, the next ones will do also. If we remove the assumptions on deadlines model and tasks offsets, checking the deadline of first jobs in a synchronous activation is no longer sufficient and the test becomes only necessary.

1.5.4 Schedulability Analysis for Dynamic-Priority

This subsection presents the classical feasibility and schedulability conditions for fixed-job-priority and dynamic-priority scheduling.

Maximum utilization feasibility test

A task set cannot be scheduled on only one processor when its utilization ratio is greater than 1. Therefore a necessary feasibility test can be obtained by Equation 1.15.

$$U \leq 1 \tag{1.15}$$

EDF utilization schedulability test

For task sets with *implicit deadlines*, the previous test become necessary and sufficient for EDF scheduling [LL73].

EDF density schedulability test

For task sets with *constrained deadlines*, Liu [Liu00] proposed a sufficient schedulability test based on the density of the system instead of its utilization as shown by Equation 1.16.

$$\Lambda \leq 1 \tag{1.16}$$

EDF schedulability Arbitrary Deadlines

For task sets with arbitrary deadlines, Baruah et al. [BHR90b] proposed a sufficient and necessary schedulability test given by Equation 1.17.

$$\forall\, t > 0,\ \sum_{\tau_i \in \Gamma} DBF_i(t) \leq t \tag{1.17}$$

1.6 Conclusion

In this chapter we introduced the classical theory of real-time systems and the basic state of the art needed to understand the next chapters. We first presented the main task models and their categorizations. Second, we described briefly the processing platforms targeted by this dissertation, namely monoprocessor platforms, and we set hardware assumptions that will be considered along this dissertation. Finally, we presented the main real-time scheduling algorithms, their categorizations and the corresponding feasibility and schedulability conditions.

In the remainder of the dissertation, we focus only on preemptive fixed-task-priority scheduling on monoprocessor platforms with additional constraints.

Chapter 2

Energy-Harvesting Systems

Contents

2.1 Introduction

Due to the growing demand for smaller devices with longer battery life, energy management in embedded systems has become one of the most active research areas. Indeed, a naive use of the available energy can lead to failures or a short lifespan. However, most of the targeted embedded applications are required to operate for long time after deployment, for example, wireless sensor nodes. The extended lifespan of these electronic devices is of particular importance when they have limited accessibility. Thus, collecting energy from the ambient environment can be a very interesting solution, which is known as ***E**nergy-**H**arvesting (EH)*. In this process, energy is drawn from the environment and then converted and stored in order to supply the embedded electronic devices. Compared to classical energy storage devices, the environment proves to be an infinite source of available energy. Furthermore, the use of this kind of energy reduces the need to replace batteries periodically which constitutes a major part of maintenance.

Many environmental sources can be exploited, including thermal, solar, mechanical, fluid, etc. Energy sources should be chosen according to the characteristics of the

targeted application. Self powered sensors for medical implants and remote monitoring sensors in structures such as bridges or buildings are typical examples of possible applications. In addition, the applications running on power-limited systems can be subject to timing constraints. Consequently, real-time and energy-aware features are both highly desirable and sometimes crucial for such systems.

In this chapter, we define what is an energy-harvesting system and we describe in detail its components.

The remainder of this chapter is organized as follows: Section 2.2 introduces and defines formally energy-harvesting systems. Section 2.3 gives an overview of the existing energy-harvesting devices and technologies. Section 2.4 discusses the different types of energy storage units and their usage. Section 2.5 gives some examples of actual energy-harvesting systems. Finally, Section 2.6 summarizes and concludes the chapter.

2.2 What is an Energy-Harvesting System ?

In our daily lives, the term *harvesting* is usually used to refer to the process of gathering different mature corps from the fields at the end of the growing season, or the growing cycle. Using this word for energy may seem a little strange. In fact, in the domain of energy, the term of *Energy-Harvesting* refers to the process of collecting the ambient energy from the environment and storing it in order to supply low power and small autonomous devices, such as wireless sensor networks, and portable electronic equipments. This process is also known as *Energy-Scavenging* or *Power-Harvesting*. Moreover, an ***E**nergy-**H**arvesting **S**ystem (EHS)* is an electronic device that uses an energy-harvesting generator to supply its activity.

Usually, energy-harvesting systems are composed of three main parts as described in Figure 2.1 on the next page: an energy harvester, an energy storage unit and a computing unit.

The energy harvester is the part responsible of collecting the ambient energy. It converts different kinds of environmental energy into electrical energy. Several energy sources and harvesting techniques are possible nowadays, in Section 2.3 on the facing page we present a non exhaustive list of energy sources and their extraction technologies.

The energy storage unit is a device used to store the harvested energy. Usually rechargeable batteries or capacitors are used. The choice of the storage unit type and capacity depends on the targeted application. In Section 2.4 on page 55 we list the main energy storage units available in the industry.

The computing unit is the embedded application or the mission part of the system, it is usually composed of data sensors, a processing unit and a data transmission device. Most of computing units embedded in energy-harvesting systems are real-time systems.

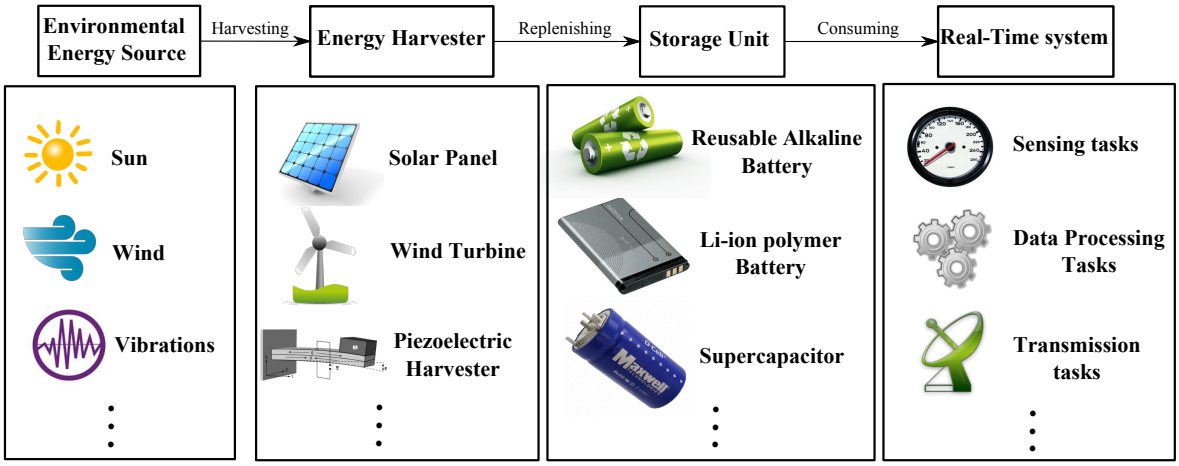


Figure 2.1: Energy-harvesting system components

In this dissertation, we focus on this part of energy-harvesting systems and we study the real-time scheduling for such systems.

2.3 Energy-Harvesting Technologies

Multiple sources are available for energy harvesting, including solar power, ocean waves, wind, vibrations, thermoelectricity, and human motions power. For example, vibration energy can be used to supply wireless monitoring sensors that are plugged inside the industrial machines. Prior research showed that these energy sources are not suitable for all applications, and that the energy sources must be chosen according to the application characteristics. In the following, we describe the main energy sources and their harvesting technologies.

2.3.1 Radiations Energy Harvesting

Solar Energy Conversion

Light is an ambient energy source that is available almost everywhere and can be used to supply electronic devices. Light energy harvesters convert light energy into electricity via photovoltaic cells. In a semi-conductor that is exposed to light, a photon with enough energy can extract an electron and create a gap. Usually, this electron finds quickly an other gap to fill, and the energy brought by the photon is dissipated. The principal of a photovoltaic cell is to put the free electrons in one side of the material and the gaps in the other one instead of filling the gaps of only one side. Therefore, a voltage difference and an electric tension appears like in a battery.

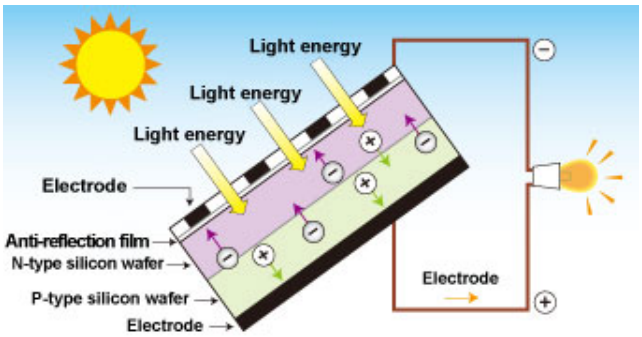


Figure 2.2: Operating principle of a photovoltaic cell

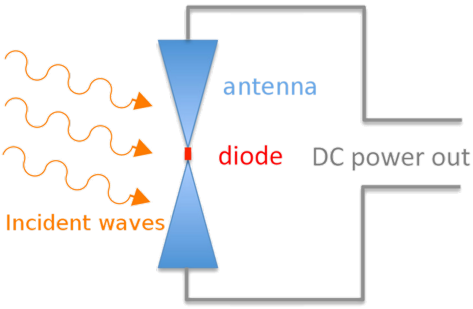


Figure 2.3: Radio waves energy-harvesting

Figure 2.2 illustrates the operating principle of a photovoltaic cell.

The efficiency of power conversion of a photovoltaic cell is given by the ratio between the input of photons energy and the output of the yielded energy. A solar cell of 100 cm^2 can generate $1.3W$ if the irradiation is 1000 W/cm^2 and the efficiency of such cell is 13 % [Mat+14]. The applications of photovoltaic cells covers a large specter of energy power, from microwatts for miniaturized applications to several kilowatts for solar power plants. Although the efficiency of commercial photovoltaic cells is still below 20%, there exists some prototypes that can reach 35%. The main limitation of solar energy harvesting is its sensibility to brightness intensity and cells size.

The use of solar energy is the topic of numerous research work and an important progress was done in this field. In [ECLEZ11] El Chaara et al propose a full review on the evolution of existing photovoltaic cells technologies in the last 30 years.

Radio Frequency Energy Conversion

The proliferation of radio transmitters in nowadays urban landscape, including mobile telecommunications and transmissions, leads to consider solutions where ambient radio frequency signals can be used as an attractive source of energy for small mobile applications. The most common method of distributing power wirelessly is to use **R**adio

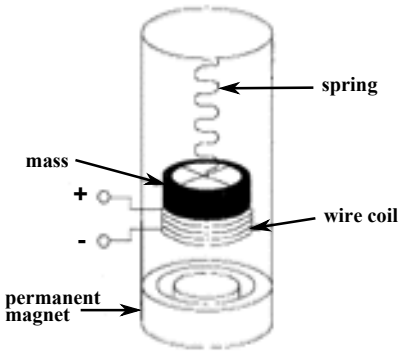


Figure 2.4: Electromagnetic vibration energy-harvesting method

Frequency (RF) radiation [SJ03]. It is now common to use RF energy-harvesting in passive **R**adio **F**requency **Id**entification (RFID) systems where RF energy is broadcasted to power remote small devices or tags in a range of less than 3 meters. The RFID systems exploit RF induction by using radio backscatter techniques which allow to consume very small power on the RFID tag during transmissions. Figure 2.3 on the preceding page illustrates with a simple drawing the RF energy-harvesting.

Using RF energy is interesting, however, RF energy harvesters have unfortunately limited power and need either large reception surface or to be very close to the transmitter. Yeatman [Yea04] remind that an electric field of 1 V/m permits harvesting no more than 0.26 $\mu W/cm^2$, however, a maximum electric field close to the transmitter can generate only few Volts per meter.

2.3.2 Vibrations Energy Harvesting

Electromagnetic Conversion

Electromagnetic energy-harvesting uses a magnetic field to convert mechanical energy to electrical energy [Yil11]. The conversion technique consists of attaching a coil to an oscillating mass and passing them through a magnetic field, which is established by a stationary magnet to produce electric energy. When the coil moves through a varying magnetic flux, a voltage is induced according to Faraday's law. Figure 2.4 illustrates such a system. Furthermore, the induced voltage is inherently small and therefore must be increased to become a viable source of energy [KN04]. Several techniques exist to increase the induced voltage including using a transformer to increase the number of turns of the coil, or increase the permanent magnetic field [TRM10]. However, each of these parameters is limited by size constraints of the microchip as well as its material properties. Most of the applications of electromagnetic transducers are used to harvest vibrations energy to power wireless sensor networks. Some examples of this method can be found in [Mit+04; Chi+00; VB+06; AC98].

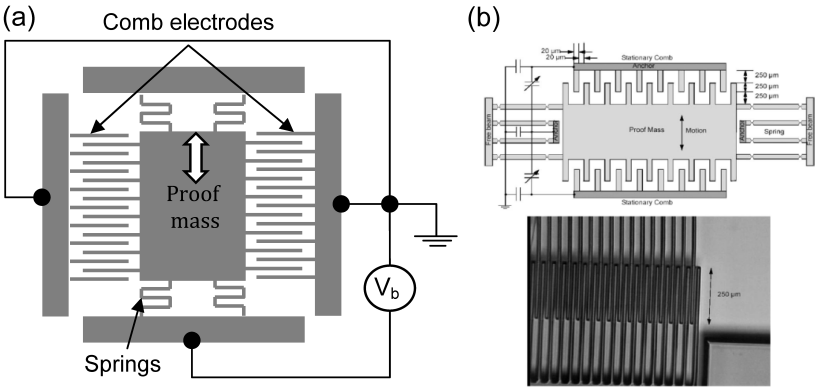


Figure 2.5: Electrostatic energy-harvesting method

Electrostatic Conversion

To exploit this source of energy, an electrostatic machine or generator is required to leverage the static electric effect. The electrostatic machine is called so because it uses the electrostatic laws in contrast of electromagnetic machines. An electrostatic motor is based on the attraction and repulsion of electric charges. Electrostatic machines are usually the dual of conventional coil-based motors. Figure 2.5 illustrates this energy-harvesting method. Although electrostatic motors were developed since the 18th century, their use as macro generator was not very successful because they generate a very high voltage with small current. However, nowadays electrostatic generators are frequently used in **M**icro**e**lectro**m**echanical **S**ystems (MEMS) where their drive voltages are below 10 volts, and where moving, charged plates are far easier to fabricate than coils and iron cores.

Several conversion techniques of electrostatic energy was proposed in literature from friction machines throw nanotuble nanomotors. The energy conversion principal is summarized in [Mit+04] and different conversion techniques are summarized in [Can63; Men+01; Mit+04].

Piezoelectric Conversion

Piezoelectric energy-harvesting devices convert mechanical energy from any type of vibration to electrical energy. It is the most widely used power harvesting techniques for micro-power operations. Strain or deformation of a piezoelectric material causes charge separation across the device, producing an electric field and consequently a voltage drop proportional to the stress applied. Usually, piezoelectric inserts are connected to a mechanical system with a resonance frequency that couples the micro-generator to vibrations source, Figure 2.6 on the facing page illustrates such a configuration. The mechanical part is an oscillating system which is typically a cantilever beam structure with a mass at the unattached end of the lever that provides higher strain for a given

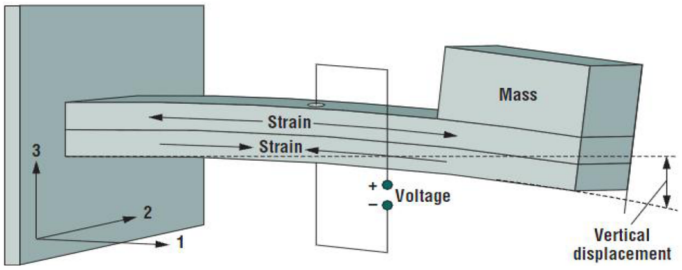


Figure 2.6: Piezoelectric energy-harvesting principle

input force. The voltage produced varies with time and vibrations frequency which produces an irregular current. Furthermore, piezoelectricity has the ability of some elements such as crystals or some types of ceramics to generate an electric potential from a mechanical stress [MVM07]. This process takes the form of separation of electric charge within a crystal lattice. If the piezoelectric material is not short circuited, the applied mechanical stress induces a voltage across the material.

Piezoelectric energy conversion produces relatively higher voltage and power density levels than the electromagnetic system.

Roundy et al [RWR03] performed a comparison between piezoelectric micro-generators and electrostatic ones. They showed that for a vibration frequency of 120Hz, piezoelectric micro-generators outperforms electrostatic ones with a power density of $250\mu W/cm^3$ against $50\mu W/cm^3$. An other comparison between piezoelectric micro-generators and electromagnetic ones was performed by Poulin et al in [PSC04] where they showed the duality of these two technologies and that electromagnetic conversion is suitable for structures with displacement and high speed. Piezoelectric generators work even with small motion amplitudes, their high power density makes them a suitable energy-harvesting technology for small devices. Roundy et al [Rou+03; RW04] realized piezoelectric micro-generator of $1cm^3$ that can supply an RF transmitter.

The main limitation of vibration energy-harvesting techniques is that their efficiency depends highly on the resonance frequency. In fact, most of energy-harvesting devices are designed to work with one or few specific frequencies. If the resonance frequency changes, the conversion is no longer optimal and the output energy can fall brutally. Hopefully, piezoelectric energy-harvesting is currently a very active research area and many solutions have been proposed to solve this problem. Two main approaches have been proposed: linear and non linear. In the linear approach the generation of energy is optimal only for several specific resonance frequencies. The systems use frequency tuning technique to follow the frequency changes [You+11; AS+13] and can support up to 40% of frequency variation [AS+13]. In the opposite side, the non linear approach modifies the classical structure of piezoelectric harvesters by replacing the mass by a magnet and adding a permanent magnet at the limit of the beam. This lead the system to behave in an non linear way and increases the bandwidth of optimal resonance

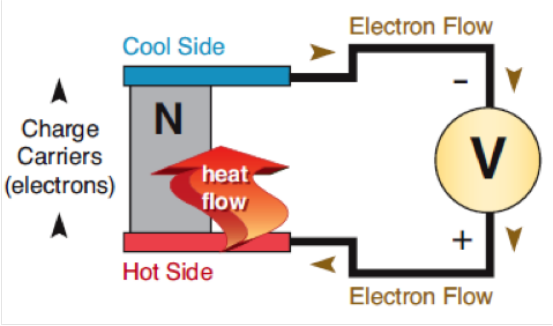


Figure 2.7: Operating principle of a thermoelectric generator

frequencies [Gam11]. This solution allows the harvester to keep a minimum output power and resist to frequency variation.

2.3.3 Thermoelectric Energy Harvesting

The difference of temperature between two points can be used to produce electrical energy by heat transfer. Figure 2.7 illustrates the operating principle of thermoelectric energy-harvesting method. The *Carnot Cycle* is used to compute the maximum theoretical efficiency of such a conversion. It is the ratio of the difference of temperature relative to the highest temperature (see Equation 2.1).

$$efficiency = \frac{The\ maximum\ temperature\ -\ The\ minimum\ temperature}{The\ maximum\ temperature} \tag{2.1}$$

This measure shows the efficiency is very low in the case of a small difference of temperature. For example, between the human body temperature, i.e. 37°C and an ambient temperature of 20°C, the efficiency cannot be more than 5.5%.

Still, some micro-generators were developed and their efficiency was lower than 10% for heat transfer from 200°C to 20°C and lower than 1% from 40°C to 20°C. Toriyama et al realized in [TYS01] a micro-battery that can deliver few μW. Though, some applications with high energy consumption were successfully supplied with thermoelectric generators. Douseki et al [Dou+03] developed an autonomous wireless communication system that harvests its energy from the difference between the ambient temperature and and cold water. For this application, the output power was 1.6 mW. Few industrial applications that use thermoelectric energy-harvesting have been proposed. For example, *Seiko Thermo Wristwatch* generates enough power to run the clock engine by absorbing body heat through the back of the watch. An other commercial application is *Thermo Life* [Sta06]. It is a small thermoelectric generator that can provide $10\mu A$ at 3 V with only 5°C of temperature difference. In [Ven+07], the authors present a new technology of nanoscale thermoelectric generators and claim that it can reach a power density of

Harvesting technology	Power density
Acoustic Noise	0.003 $\mu W/cm^3$ at 100Db
	0.96$\mu W/cm^3$ at 100Db
Radio Frequency	1 $\mu W/cm^2$
Ambient Light	15 mW/cm^2 (outdoor)
	100 $\mu W/cm2$ (indoor)
Vibration (Electrostatic)	4 $\mu W/cm^3$ (human motion in Hz)
	800 $\mu W/cm^3$ (machines in kHz)
Vibrations (Piezoelectric)	1.4-9 mW/cm^3
Thermoelectric	10-100 $\mu W/cm^3$

Table 2.1: Power density comparison of energy-harvesting methods [Yil11]

100 $\mu W/cm^3$ only with a thermal gradient of 1°C.

The above list of exploitable energy sources is not exhaustive, other energy sources can also be exploited, these are the most investigated ones.

2.3.4 Energy Sources Comparison

Recall that the scope of this dissertation is the real-time scheduling for embedded systems working with energy-harvesting. Real-time means predictability, thus, we should carefully select the right energy-harvesting technique to the right application to ensure predictability. In fact, designers should measure the worst energy consumption of the targeted applications and ensure that the selected energy source and its harvesting technology are capable to supply the necessary energy even in the worst case. Solar energy for example cannot be available all the time, then, it is not suitable as main energy source for applications with hard real-time constraints that are expected to run for very long time but can be used as a secondary source. However, piezoelectric energy-harvesting may be a promising harvesting method for small monitoring sensors in manufacturing and petrochemical installations because it has a high power density comparing to the other sources (see Table 2.1) and can resist to vibration frequency variations by capturing a large spectrum of frequencies using tuning and non-linear adaptation approaches.

2.4 Energy Storage Technologies

During the last decades, rechargeable batteries have made only moderate improvements in terms of higher capacity and smaller size, Figure 2.8 on the following page illustrates the progress of technology in different **I**nformation and **C**ommunication **T**echnologies (ICT) comparing to energy storage during the last twenty years. Research

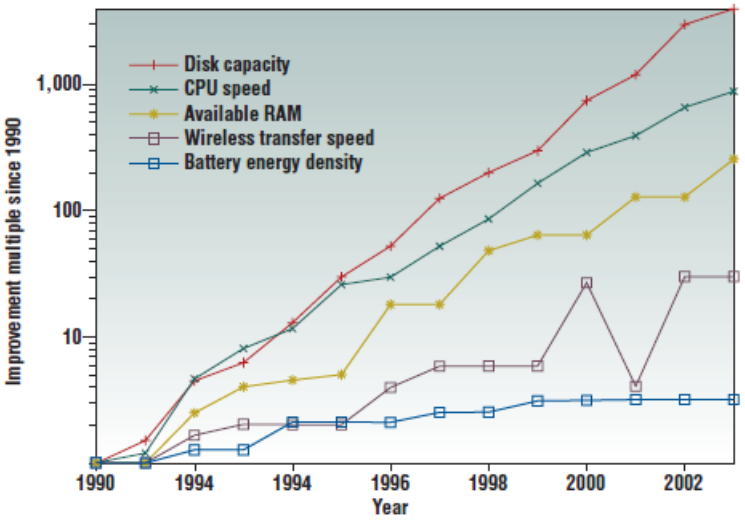


Figure 2.8: Improvement of the battery energy density compared to other significant figures in the ICT panorama in the last twenty years [PS05]

has brought about a variety of chemical energy storage devices, each one offers some advantages but none of them provides a fully satisfactory solution. Therefore, we should select the right energy storage technology for the right application.

Energy storage devices are evaluated by means of several factors and metrics, the following list describes the most important ones.

- *Energy Density*: is the amount of energy stored per unit of mass or volume, it is expressed in Wh/m^3 for *Volumetric Energy density* and Wh/kg for *Gravimetric Energy Density*. The greater energy density is, the better it is.

- *Nominal capacity*: is the amount of charge expressed in Ampere-hour that can be delivered by an energy storage unit

- *Internal resistance*: when the storage device delivers current, the measured voltage output is lower than the theoretical voltage. The difference is the voltage drop caused by the internal resistance. Therefore, the lower internal resistance is, the better it is.

- *Charge rate*: a charge or discharge current of a battery is measured in C-rate. A discharge current of $1C$ draws a current equal to the rated capacity. For example, a battery rated at 1000 mAh provides 1000 mA for one hour if discharged at $1C$ rate.

- *Cycle life*: is the number of charge-discharge cycles that the energy storage device can achieve.

Figure 2.9: Example of Panasonic NiCd batteries

- *Nominal voltage*: also called *average discharge voltage*, is the mid-point voltage of the energy storage unit voltage range during charge or discharge.
- *Self-discharge*: is the percentage of capacity loss when the storage device is not used.
- *Fast charge time*: is the shortest time needed to fully replenish the device

An ideal energy storage device should be small, light and quickly rechargeable with a long lifespan. Different technologies of energy storage have been investigated and commercialized, we can distinguish two main families of energy storage devices: batteries and capacitors (or super capacitors).

2.4.1 Rechargeable Batteries

Chemistry batteries are the most widely used nowadays as energy-storage technology for electronic devices [CC05]. Many materials have been used to develop batteries with different characteristics. In the following we describe briefly the chemistry batteries available in the market:

Nickel Cadmium (NiCd): is the oldest technology, research began on a sealed NiCd battery, which recombined the internal gases generated during charge rather than venting them. These advances led to sealed NiCd battery which is in use today. The NiCd supports better fast charge and performs better under rigorous working conditions. Although it supports a high number of charge-discharge cycles, a periodic full discharge is so important that, if omitted, large crystals will form on the cell plates and the NiCd will gradually loses its performance. This phenomena is also known as *memory effect*. Furthermore, NiCd has relatively a low energy density and has a bad environmental impact due to its toxic metals. Figure 2.9 shows an example of commercial NiCd batteries.

Figure 2.10: Example of Panasonic NiMH batteries

Figure 2.11: Example of Lead Acid batteries

Nickel-Metal Hydrid (NiMH): this technology is gradually replacing NiCd because it offers 30-40% higher capacity and a higher energy density. However, both NiMH and NiCd are affected by high self-discharge. The NiCd loses about 10% of its capacity within the first 24 hours, after which the self-discharge settles to about 10%. Moreover, it needs deep charging cycles and does not support high rate of charge-discharge cycles. Figure 2.10 shows an example of commercial NiMH batteries.

Lead Acid: this kind of batteries work with flooded lead acid. This technology is used today in automobiles batteries where batteries are all the time in vertical position. Unlike the NiCd, lead acid batteries does not like deep charging cycles. A full discharge causes extra strain and each discharge/charge cycle decreases the battery capacity. This loss is very small while the battery is in good operating conditions, but increases dramatically once the performance drops below 80% of its nominal capacity. Even though, this technology has one of the lowest self discharge rates, it offers a low energy density which limits its usage to stationary applications. Figure 2.11 shows an example of car Lead Acid batteries.

Figure 2.12: Example of Panasonic Li-ion batteries

Figure 2.13: Example of Li-polymer batteries

Lithium Ion (Li-ion): Lithium is the lightest of all metals, has the greatest electrochemical potential and provides the largest gravimetric energy density. Rechargeable batteries that uses lithium metal are capable of providing both high voltage and excellent capacity, resulting in an extraordinary high energy density. Moreover, it offers a low self-discharge rate about half less than NiCd and NiMH. Though, it requires protection circuits that reduces voltage and current. Li-ion is widely used to supply portable devices like cameras and phones. Figure 2.12 shows an example of commercial Li-ion batteries.

Lithium-ion Polymer (Li-polymer): Li-polymer batteries use different type of electrolyte, a dry solid polymer electrolyte only. This electrolyte looks like a plastic film that does not conduct electricity but allows an exchange of ions. The polymer electrolyte replaces the traditional porous separator, which is soaked with electrolyte. The dry polymer design offers simplifications with respect to fabrication, ruggedness, safety and thin-profile geometry. There is no danger of flammability because no liquid or gelled electrolyte is used. Unfortunately, the dry Li-polymer suffers from poor conductivity. Internal resistance is too high and cannot deliver the current bursts needed for modern communication devices. Figure 2.13 shows an example of commercial Li-polymer batteries.

Figure 2.14: Example of Maxwell ultracapacitors

Reusable Alkaline : replaces disposable household batteries; suitable for low-power applications. Its limited cycle life is compensated by low self-discharge, making this battery ideal for portable entertainment devices and flashlights.

2.4.2 Supercapacitors

A supercapacitor works like a regular capacitor with the exception that it offers very high capacitance in a small size, resulting in an intermediate energy density between regular capacitors and chemistry batteries. Moreover, the amount of energy a capacitor can hold is measured in microfarads (1 $\mu F = 10^{-6}$ farad). Small capacitors are measured in nanofarads (1000 times smaller than 1 μF) and picofarads (1 million times smaller than 1 μF). Supercapacitors are rated in units of 1 farad and higher. The gravimetric energy density is 1 to 10Wh/kg [Buc11]. This energy density is high in comparison to the electrolytic capacitor but lower than batteries, it is approximately 10% of the one of NiMH battery. A relatively low internal resistance offers good conductivity [Buc11]. Figure 2.14 illustrates some commercial super capacitors.

The major disadvantage of super capacitors is the extremely high self-discharge rate, however, this may not be an obstacle for small devices since a supercapacitor is fully discharged after several days. Some supercapacitors can retain the charged energy longer. Their capacity drops from full charge to 85% in 10 days. In 30 days, the voltage drops to roughly 65% and to 40 % after 60 days [Buc11].

Supercapacitors have also many advantages, we list hear some of them.

- Unlimited cycle life: not subject to the wear and aging experienced by the electrochemical battery,
- Low impedance: enhances pulse current handling by paralleling with an electrochemical battery,

	NiCd	NiMH	Lead Acid	Li-ion	Li-ion polymer	Reusable Alkaline	Super capacitors
Energy Density Gravimetric (Wh/kg)	45-80	60-120	30-50	110-160	100-130	80	1-10
Internal Resistance (mW) (includes peripheral circuits)	100-200	200-300	<100	150-250	200-300	200-2000	$1-30$
Cycle Life to 80% of initial capacity	1500	300-500	200-300	500-1000	300-500	50 (to 50%)	10^6
Fast Charge Time	1h	2-4h	8-16h	2-4h	2-4h	2-3h	1-10s
Self-discharge /Month	20%	30%	5%	10%5	10%	0.3%	40%
Nominal Voltage (V)	1.25V	1.25V	2V	3.6V	3.6V	1.5V	1-3V
Charge rate (C) peak to best	20-1	5-0.5	5-0.2	2-1	2-1	0.5-0.2	
Operating Temperature (°C)	-40-60	-20-60	-20-60	-20-60	0-60	0-65	-40-85

Table 2.2: Energy storage technologies comparison [Buc11]

- Rapid charging: low-impedance supercapacitors charge in seconds,
- Simple charge methods: voltage-limiting circuit compensates for selfdischarge, no full-charge detection circuit needed.

Supercapacitors may be interesting energy storage devices for energy-harvesting systems since they are continuously charging and almost continuously supplying electronics components which fits with their advantages and avoid their drawbacks.

Table 2.2 summarizes the differences between all the presented technologies.

2.5 Energy-Harvesting Applications

Researchers are more and more interested by energy issues in ICT[1]. This interest is due to the disproportional progress between the computing part and energy part in modern embedded systems. Research is leaded by a growing demand of miniaturized and autonomous devices from industries and users. Therefore, several companies which are focusing energy issues of small embedded systems have emerged. Companies like *AdaptiveEnergy, EnOcean, Cymbet and Perpetium*, among others are specialized in energy-harvesting systems. The main applications developed by these companies are mainly concentrated on military fields (e.g. battlefield surveillance, recognition of enemies and drones attacks, etc.), environment issues (e.g. animals movement tracking,

[1]Information and Communication Technologies

Figure 2.15: Energy-Harvesting

forest fire detection, pollution monitoring, etc.), manufacturing industries (engines monitoring, etc.) and health issues (e.g. medical implants, health monitoring, etc.).

2.6 Conclusion

In this chapter, we introduced energy-harvesting systems and their components. After stating the notion of energy-harvesting, we presented a non-exhaustive list of different renewable energy sources that can be exploited and their extraction methods. We observed that energy from solar and vibration sources is the most efficiently extracted thanks to piezoelectric and photovoltaic technologies.

After that, we investigated the different types of energy storage devices and we compared their strengths and weaknesses. We noticed that the market of energy storage is dominated by chemical batteries especially by Lithium-ion batteries whose recent models offer high capacity, high energy density and long lifespan. Nevertheless, the second family of energy storage units, which is supercapacitors, is more and more competitive comparing to batteries. Recent supercapacitors have much higher energy density and capacity than regular capacitors, and offer a huge cycle life with a longer self-discharge time. These advantages allow supercapacitors to be used as a real energy storage device instead of only short time energy buffers, which makes them more suitable for small and autonomous devices.

For autonomous real-time applications, supercapacitors and adaptive piezoelectric energy-harvesting seem to be the most suitable energy storage devices and energy extraction methods since they allow a high and stable extraction of energy and a flexible

and quick storage and use of the extracted energy.

In the next chapter, we focus on the computational part of an energy-harvesting system and we address the real-time scheduling problem that considers the constraints of an energy-harvesting systems.

Chapter 3

Real-time Scheduling for Energy-Harvesting Systems

Contents

3.1 Introduction

The aim of embedded systems that work with environmental energy is to achieve a set of missions autonomously for very long time. The missions of such applications are mainly sensing operations, data processing and data transmission. These operations are managed by the computational part of an energy-harvesting system as mentioned in Chapter 2. Usually, energy-harvesting embedded systems are constrained by time in order to deliver critical results on time, especially for critical embedded systems such as medical implants or nuclear reaction monitoring devices. Therefore, it is mandatory to guarantee the respect of time and energy constraints of these systems. Then, it is worthy to study the real-time scheduling under renewable energy constraints for systems that consider energy.

A naive approach consists of oversizing the capacity of the components of the system: a very large energy harvester (e.g. a huge solar panel), an overestimated energy storage device capacity and an oversized computation unit that guarantees the validity of the real-time system under pessimistic schedulability analysis. However, this may be very costly and may not fit with the desirable size especially for small applications (e.g. medical implants).

This has motivated researchers to focus on this challenging problem. The crucial issue associated to these systems is to find scheduling mechanisms that can manage their performance and their activity according to the available energy along the system lifespan in order to respect both time and energy constraints. Now, the primary concern is that the energy collected from the environment, and that can be stored, should be fully consumed to maximize system's performance and avoid energy waste.

In this chapter we address the real-time scheduling problem for energy-harvesting systems. After setting the scope of this dissertation by specifying the formal model of the targeted real-time energy-harvesting systems in Section 3.2 and the associated scheduling problematic in Section 3.3, we give a brief overview of the available scheduling approaches in Section 3.4. Then, we focus only on energy-aware scheduling approach and we explore the properties of the different classes of scheduling algorithms that are proposed in the literature from Section 3.5 to Section 3.7. Finally, we present the results of our experiments that compares the algorithms, and then, we summarize their differences in Section 3.8.

3.1.1 Task Model

The task model considered here is an extension of Liu and Layland's model, described in Section 1.2 on page 30, that considers task energy consumption in addition to the classical parameters. Then, we consider a real-time task set in a renewable energy environment defined by a set of n periodic/sporadic and independent tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized by its priority P_i, its worst-case execution time C_i, its period T_i, its relative deadline D_i and its *worst-case energy consumption* E_i.

Therefore, a task τ_i releases an infinite number of jobs separated by at least T_i time units and each job is executed during C_i time units and consumes E_i energy units and must finish before D_i time units after being requested. Moreover, the deadlines are constrained or implicit and the task set is priority-ordered such that task τ_n is the task with the lowest priority.

3.2 General Model

An energy-harvesting system being composed of a real-time computational part should be first described as a classical real-time system with a set of recurrent tasks, as explained in Chapter 1, and second as a more complex system by including the energy source and the energy storage constraints described in Chapter 2. In this section we specify the general formal model considered by this dissertation.

3.2.1 Energy Model

We consider an embedded system connected to an energy harvesting device. An energy-harvesting device as described in Section 2.2 on page 48 is a system that collects the ambient energy from the environment using an energy-harvesting technique. The collected energy is then stored in an energy storage unit with fixed capacity (e.g. rechargeable battery or supercapacitor).

Replenishment: We suppose that the amount of energy that arrives into the storage unit is a function of time which is either known or bounded. Recall that the profile of energy arriving from the harvester depends on the energy source and the harvesting technique. Most of energy sources are unpredictable or predictable with difficulty, for example, solar energy harvesting depends on brightness intensity which cannot be predicted accurately. For this reason, since the scope of this dissertation is hard real-time scheduling for energy-harvesting systems, we consider only energy sources and harvesting techniques that can provide a predictable profile of energy or that can be lower bounded. Hopefully, such energy profiles exist especially with vibration energy source and piezoelectric harvesting technique which seem to be suitable for small embedded systems. It provides a nearly stable output of energy even with a deviation of vibration frequency up to 40% form the optimal one as explained in Section 2.3.2 on page 52.

Therefore, as a first step, we can consider a uniform or a bounded replenishment function which means that the storage unit receives a constant amount of energy every time unit. We denote $P_r(t)$ the replenishment function of the battery, then, the energy

replenished during any time interval $[t_1, t_2]$ denoted as $g(t_1, t_2)$ is given by Equation 3.1.

$$g(t_1, t_2) = \int_{t_1}^{t_2} .P_r(t)\ dt \tag{3.1}$$

As mentioned above, we assume that that $P_r(t)$ is a constant function, i.e. $P_r(t) = P_r$. Then, the energy replenished during any time interval $[t_1, t_2]$ is given by Equation 3.2.

$$g(t_1, t_2) = (t_2 - t_1) \times P_r \tag{3.2}$$

In the remainder of this dissertation, we use P_r instead of $P_r(t)$ to denote the replenishment function and we suppose that it is lesser than or equal to the battery capacity.

Storage: The replenishment of the storage unit is performed continuously even during jobs execution and the level of the stored energy fluctuates between two thresholds E_{min} and E_{max} where E_{max} is the maximum capacity of the storage unit and E_{min} is the minimum energy level that keeps the system running. The difference between these two thresholds is the part of the battery capacity dedicated to tasks execution. This capacity is denoted $\mathcal{C}$. We suppose that $\mathcal{C}$ is sufficient to execute at least one time unit of each task. This means that $\mathcal{C}$ must be greater or equal to the maximum instantaneous consumption, i.e. $\mathcal{C} \geq \max_{\forall i}(E_i/C_i)$, otherwise some tasks cannot be executed. We suppose also that the storage device is carefully selected to ensure regular behavior, i.e. regular replenishment and regular discharge in order to avoid charge/discharge speed variations and capacity losses due to numerous charge/discharge cycles. Recall that supercapacitors can offer these requirements as mentioned in Section 2.4.2 on page 60.

For the sake of clarity, we can consider without loss of generality that $E_{min} = 0$ and that $\mathcal{C} = E_{max}$. The battery level at time t is denoted as $E(t)$. Below, we use the word "battery" to refer to the energy storage unit in order to simplify the language.

Consumption: Tasks energy cost should actually include not only dynamic and static processor energy consumption but also the consumption of other devices that a task can use, e.g. sensors and data transmission devices. Moreover, even if we consider only processor consumption, the global consumption depends much more on the kind of circuitry used by the code than on the execution duration [JM06]. For this reason we consider that the execution time C_i and the energy consumption E_i of a task are fully independent. For example, considering two tasks τ_i and τ_j that do not use the same devices, then, we can have $C_i < C_j$ and $E_i > E_j$. Furthermore, we consider that energy consumption is function of time that is in reality not necessarily uniform. Actually, since tasks can use different devices, it is difficult to predict the accurate energy profile of tasks. Moreover, the worst energy consumption profile is not known up to now. This is a serious issue for real-time predictability, however, including this constraint to scheduling decisions makes it a hard problem with many parameters to consider. As a first step and for the sake of simplicity, we consider for the scope of this dissertation

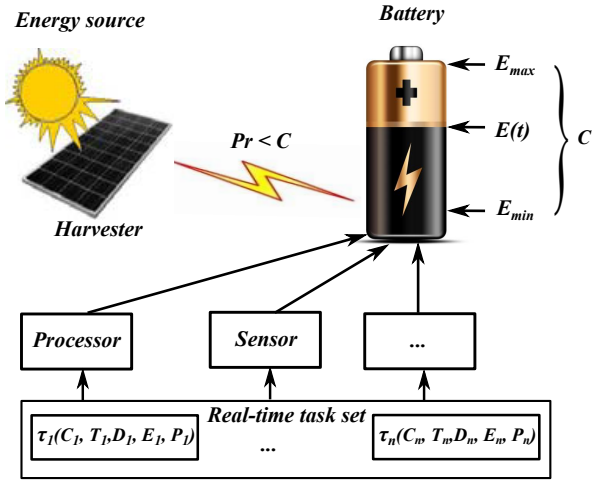


Figure 3.1: Energy-harvesting system model

that the energy consumption function is different from task to task but linear, which means that each task has its own constant instantaneous consumption E_i/C_i.
Figure 3.1 recapitulates these descriptions.

Addressing the limitations of the model

This model seems to be too theoretical and not realistic, especially for energy assumptions, i.e. the replenishment function and the energy consumption profiles. In fact, although we have considered a constant replenishment function, most Lithium batteries have nearly a constant recharging rate in the middle part (0 to 80%), then this rate decreases asymptotically to 0 when the battery is fully charged [Tos01]. Moreover, as shown in Section 2.4.2 on page 60, new generation of supercapacitors are capable of delivering a constant rate of energy all the time. Moreover, there exist some energy extraction technologies that can now provide a nearly constant energy output from specific energy sources, e.g. vibrations and piezoelectric techniques (see Section 2.3.2 on page 52). Note that this may not be possible for other renewable energy sources and harvesting methods.

The second concern is tasks energy consumption profile. If the consumption function is not constant but does not vary much during the execution of the tasks, one can intuitively use the higher boundary as a safe limit. This is still one of the open questions to address.

Although this model is not totally realistic, it is still interesting to study as a first step of this research area because the scheduling problematic for energy-harvesting systems is not easy to handle and contains numerous counter intuitive properties, especially for fixed-priority scheduling.

3.2.2 Definitions

Here we define some new properties and we redefine some properties that we already defined in Section 1.5.1 on page 41 to include energy requirements.

Definition 3.1 (Energy Valid Schedule)**.**
The schedule $\mathcal{S}$ produced by scheduling algorithm $\mathcal{A}$ is considered as valid only and only if all the deadlines are infinitely met and the battery energy level never falls below E_{min}. ■

Definition 3.2 (System Energy Utilization)**.**
The system energy utilization denoted Ue is the ratio of energy to be consumed by the execution of the task set relative to the energy that can be replenished. In the case of constant replenishment function P_r, it can be obtained by Equation 3.3.

$$U_e = \sum_{i=1}^{n} \frac{E_i}{T_i \times P_r} \tag{3.3}$$

■

Definition 3.3 (Energy-Busy-Period)**.**
An energy-busy-period is an interval of time $[t_1, t_2]$ where the system is continuously executing jobs or replenishing energy if the battery level is not sufficient to execute the ready job, i.e. $E(t) < E_i/C_i$, while there are waiting jobs. It starts from time t_1 when the processor is idle for the last time and ends at time t_2 when all waiting jobs are finished. It is possible to have consecutive energy-busy-periods since it depends on the waiting jobs and not only the processor or the replenished energy. ■

Definition 3.4 (Energy-Concrete Systems)**.**
An energy-concrete system is a system whose time and energy parameters are known before run-time. This includes tasks periods, tasks offsets and the initial energy storage level of the storage unit. In the opposite, if all or one of these parameters is known only at run-time, then, the system is said *energy-non-concrete*. ■

Definition 3.5 (Energy-Work Conserving)**.**
A scheduling algorithm is classified as energy-work-conserving policy if whenever there are active tasks requiring execution, the scheduling policy leaves the processor idle only if there is no enough energy available to execute at least one time unit of the active task. In contrast, if the scheduling policy authorizes unnecessary idle times, it is classified as non-energy-work-conserving. ■

Definition 3.6 (Consumming Task)**.**
A task is considered as a consuming one if its energy consumption is greater than the energy replenished during its execution time. ■

Definition 3.7 (Gaining Task)**.**
A task is considered as a gaining one if its energy consumption is lesser than or equal to the energy replenished during its execution time. ■

3.3 Scheduling Problematic

Throughout this dissertation, we will be interested in the monoprocessor fixed-priority real-time scheduling problem for energy-harvesting systems. In the considered model, the system has to ensure the correctness of results and the respect of deadlines before which results must be delivered. It has also to keep the system running by satisfying all energy requirements.

A key consideration that affects power management in energy-harvesting systems is that instead of minimizing the energy consumption and maximizing the achievable lifetime, as n classical battery operated devices, the system must operate in an energy neutral mode by consuming energy only as much as harvested.

Moreover, the classical scheduling algorithms and their associated feasibility conditions are no longer suitable for the energy-harvesting model because they do not consider energy constraints. In fact, respecting energy constraints, namely tasks energy cost and battery capacity, leads the scheduler to add more idle periods to replenish energy, thus, a classical **F**ixed-**T**ask-**P**riority (FTP) scheduling algorithm works only when the battery is overestimated and fully charged. Furthermore, the additional idle periods delay tasks response times which makes the classical response time analysis and feasibility conditions necessary but not sufficient for this model. Figure 3.2 on the next page shows an example where the classical FTP scheduling algorithm leads the system to run out of energy while a feasible schedule is possible by adding greedily replenishment periods. It shows the time chart of the task set described in Table 3.2(a) as well as the battery level chart. We observe that the FTP algorithm executes jobs as soon as possible and consumes energy until the battery runs out of energy (see Figure 3.2(b)). This is because the energy demand of the executed jobs is higher than the available energy, i.e. the replenished and the remaining energy in the battery, during their execution. In contrast, when more idle periods are added, the energy demand is still the same and the available energy is greater, and therefore, a feasible schedule becomes possible (see Figure 3.2(c)). Thus, managing jobs scheduling in energy-harvesting systems is managing the length and the time of replenishment periods.

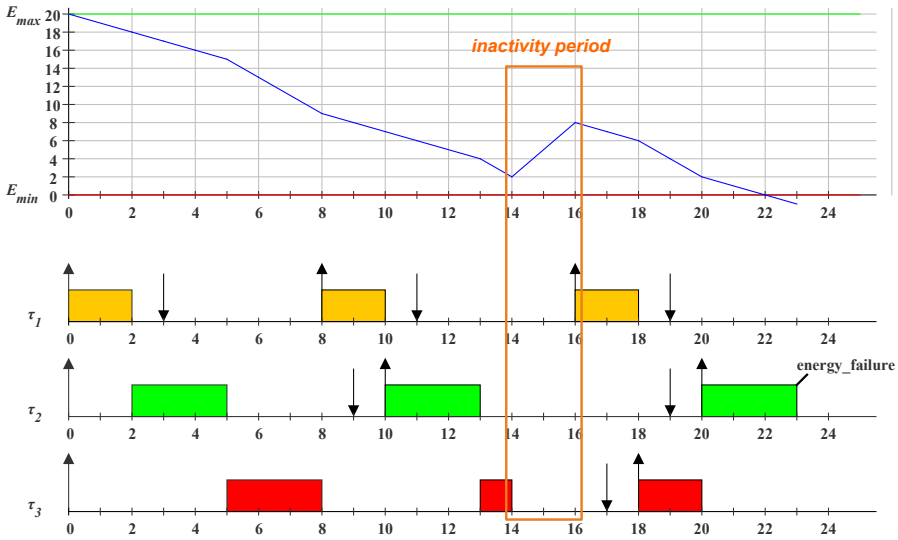
An other key consideration of managing the scheduling of energy-harvesting systems is the battery capacity. The design of a system with an arbitrary battery capacity may lead to an oversized $\mathcal{C}$, which can be very costly in space, weight and money. Finding the lowest battery capacity that preserves the schedulability of a task set is a very important issue.

Given a feasible task set with $\mathcal{C} = \infty$, the minimum battery capacity issue in energy-harvesting systems is to find the smallest capacity denoted $\mathcal{C}_{min}$ that keeps the task set schedulable with the considered algorithm when the system is started with the minimum battery level.

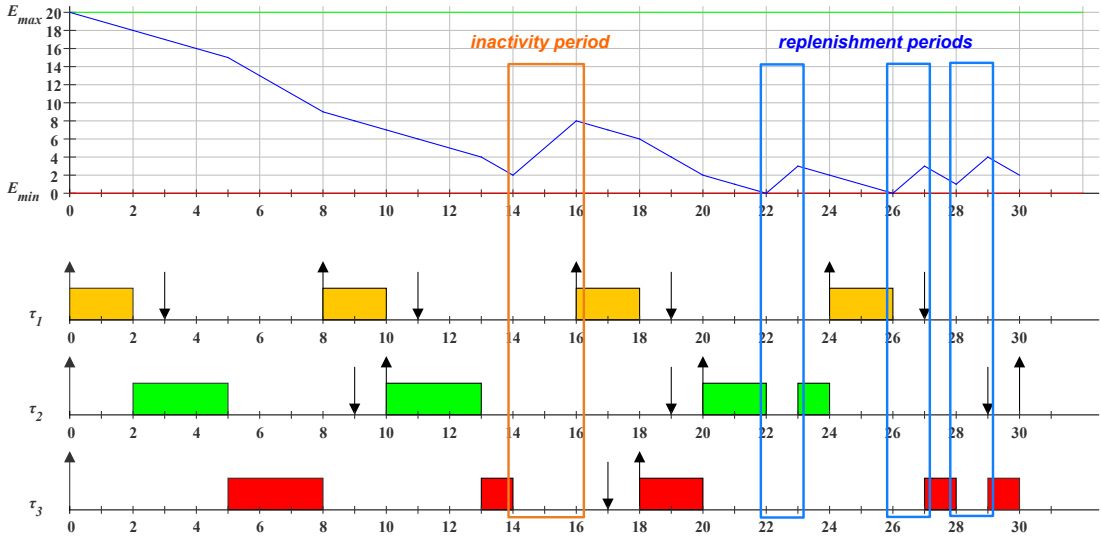
The exact value of $\mathcal{C}_{min}$ is difficult to estimate because it depends on the environmental characteristics and the used scheduling algorithm. One can solve the problem partially by bounding $\mathcal{C}_{min}$.

-	C_i	E_i	T_i	D_i	P_i
τ_1	2	6	8	3	1
τ_2	3	9	10	9	2
τ_3	4	16	18	17	3

(a) Task set Γ with $P_r = 3$



(b) Classical FTP algorithm



(c) Ideling algorithm

Figure 3.2: Classical FTP scheduling algorithms are not suitable

To summarize, the problems we are interested in can be formulated by the following questions.

1. **Scheduling algorithms**: How can we schedule a given fixed-priority real-time task set on one processor so to perpetually respect tasks deadlines and satisfy their energy demand while keeping the battery level higher than E_{min} ?

2. **Feasibility conditions**: Does a necessary and sufficient feasibility condition, that guarantees before run-time the feasibility or the unfeasibility of a given taskset, exist ?

3. **Minimum energy storage capacity:** What is the minimum battery capacity that keeps the feasibility of a given task set ?

Through this dissertation, we will first explore some solutions and some answers to these questions that have been proposed in the literature, and then, we explain the contributions of this thesis.

3.4 Scheduling Approaches

In the last decades, different solutions have been proposed in the literature to deal with energy issues. Most research focused on reducing the global energy consumption of the system. This is due to promising autonomous embedded systems whose lifetime was seriously limited by the capacity of batteries. Later, with the emerging energy-harvesting technologies, researchers realized that they should not focus only on energy consumption optimization but also on the management of renewable energy. This kind of energy is collected from the environment with small quantities but with a continuous supply. From this, we can identify two main families of real-time scheduling research that address energy issues: scheduling for energy consumption optimization, mainly represented by **C**entral **P**rocessing **U**nit (CPU) frequency and voltage scaling, and energy-aware scheduling that manages executions and replenishment to operate perpetually.

3.4.1 Dynamic Voltage and Frequency Scaling

The **D**ynamic **V**oltage and **F**requency **S**caling (DVFS) approach consists of slowing down the processor speed by reducing either the voltage or the processing frequency or both in order to reduce the processor energy consumption [PBB98; Wei+94]. This approach relies on the fact that the energy consumption in small embedded systems comes mainly from processor's dynamic energy consumption [ZMM04]. The research interest for this approach was amplified when microprocessors manufacturers released processors with DVFS capabilities (e.g. Intel's SpeedStep, AMD's Cool'n'Quiet and PowerNow, etc.).

Many solutions propose real-time scheduling schemes that use DVFS capabilities to respect deadlines and to reduce the global energy consumption. When using DVFS, tasks execution times become longer due to slower processing. Then, for real-time scheduling, the slack-time or the available idle time can be used to reduce energy consumption by slowing down the processor and lengthening tasks response times [PBB98; Wei+94]. The challenges of such an approach are to select the right subset of jobs for which are applied frequency scaling and to select the right CPU frequency to apply for each task. Unfortunately, this was proved to be a **N**on-deterministic **P**olynomial-time Hard (NP-Hard) problem in [ZA06], thus, works dealing with this problem focused mainly on proposing adapted heuristics [ZQA07; ZA09]. The main drawback of scaling the voltage and frequency is that it increases the risk of transient errors in **C**omplementary **M**etal–**O**xide–**S**emiconductor (CMOS) logic circuits [JK95; HS00; Zie14]. To cope with this difficulty, fault tolerance techniques have been used by introducing redundancy. Zhu et al. in [ZA06] proposed a heuristic that uses slack-time to perform DVFS and reserve time for rollback tasks as fault tolerance method.

The DVFS is an interesting technology to reduce the energy consumption of the system for embedded systems equipped with processors with several CPU frequencies. However, it increases the risk of transient faults which requires the implementation of fault tolerance techniques and may make the gain of consumed energy not worthy. Furthermore, even though it reduces the global energy consumption, it cannot be used alone in an energy-harvesting system especially when the slower frequency is not sufficient to reduce enough the consumed energy. Some adaptations for energy-harvesting systems have been proposed, however, most of them are still supposing negligible faults rate and a large number of CPU frequencies.

3.4.2 Energy-Aware Scheduling

In contrast to DVFS, energy-aware scheduling aims to dynamically manage the available energy at any time, i.e. the energy collected from the environment and drained from the battery, rather than optimizing or minimizing the global energy consumption. It consists of selecting the right periods of execution and the right periods for replenishment so to ensure a perpetual operation. This scheduling approach considers only mono-frequency processors, and therefore, only two modes: active mode where the system consumes energy and inactive mode where the system is idle and does not consume energy or consume a negligible amount of energy. In this dissertation we will focus only on this scheduling approach. In the remaining part of this chapter, we explore different scheduling algorithms proposed in the literature for this scheduling approach. For each algorithm, we specify the model for which it was designed and we study some of its properties, e.g. optimality, minimum battery capacity, etc. After that we test these algorithms with the model specified in Section 3.2 on page 67 in order to compare them by simulations, and then, we summarize their strengths and weaknesses.

3.4.3 Other Approaches

There exist other approaches to respect energy and temporal constraints. Some of them use the concept of task reward. In this kind of approaches, each task is characterized with a set of versions, each version represents the a different level of precision or quality of service. The response time of a task is composed of a critical or a mandatory part and a set of less critical or optional parts. Thus, the higher the executed version is, the more energy is consumed and the greater the reward is. The aim of reward-based approaches is to find the right version for each task such that the energy budget is enough and the deadlines are met. Rusu et al. proposed in [RMM03a; RMM03b] such a solution for systems that are powered with rechargeable batteries.

However, if we consider tasks with only one version and a processor with only one frequency, the scheduling problem is reduced to the model described in Section 3.2 on page 67.

3.5 Scheduling for Frame-Based Systems

The real-time scheduling for embedded systems with rechargeable batteries was addressed for the first time by Allavena et al. in [AM01]. They studied the problem only for the frame-based task model and they proposed a scheduling algorithm for this model.

Frame-Based Model

The aim of Allavena and Mossé in [AM01] was to propose a solution for monoprocessor embedded systems with voltage and frequency scaling capabilities. In the first part of the paper, they proposed a scheduling algorithm for monoprocessors with only one voltage and one frequency. In this subsection, we discuss only this case to fit with the objectives of this dissertation. Therefore, the considered model is described as follows.

Task Model

They considered a frame-based task set Γ composed of n independent tasks that share the same period T (also called frame) and the same implicit deadline $D = T$. Each task takes C_i time units and consumes E_i energy units at a constant instantaneous rate E_i/C_i.

Energy Model

The system runs with a rechargeable battery, whose energy level at time t denoted $E(t)$ remains between two boundaries E_{max} and E_{min}. The battery has a capacity $\mathcal{C} = E_{max} - E_{min}$ and is replenished with a constant power rate P_r even during executions.

Due to the constant nature of consumption and replenishment rates, we can combine them. Then, we can distinguish two groups of tasks, namely gaining tasks denoted Γ_g, that consume less energy than replenished, i.e. $\Gamma_g = \{\tau_i \in \Gamma,\ P_r - E_i/C_i \geq 0\}$, and dissipating or consuming tasks denoted Γ_c that consume more energy than replenished, i.e. $\Gamma_c = \{\tau_i \in \Gamma,\ P_r - E_i/C_i < 0\}$. Then, the energy gained by the group Γ_g in one frame is $|\Gamma_g| = \sum_{\tau_j \in \Gamma_g} P_r \times C_j - E_j$, and the energy consumed by group Γ_c is $|\Gamma_c| = \sum_{\tau_j \in \Gamma_c} E_j - P_r \times C_j$.

All the tasks are requested at time $t = 0$ with a fully charged battery, i.e. $E(0) = E_{max}$ and preemptions are allowed.

Algorithm Description

For the above model, the scheduling algorithm must execute all the tasks within D time units starting with a fully charged battery and ending the frame with the same battery level. The intuition of the proposed algorithm is to execute tasks until the battery goes to the minimum level by running only consuming tasks, and then, replenish as much as possible by executing gaining tasks and adding idle periods if necessary. Thus, the battery level fluctuates between E_{max} and E_{min} and the same schedule is repeated for each frame.

Algorithm's rules

For the sake of clarity, we call this algorithm **F**rame-**B**ased **A**lgorithm (FBA). It is described by Algorithm 3.1 on the facing page and respects the following rules:

1. if $|\Gamma_c| > |\Gamma_g|$, an idle period of length $idle = \left\lceil \frac{|\Gamma_c| - |\Gamma_g|}{P_r} \right\rceil$ is added before the execution of Γ_c.

2. if $\sum_{i=1}^{n} C_i + idle > D$, the algorithm cannot schedule the task set and the deadline is missed.

3. Schedule tasks of Γ_c until there are no more tasks in Γ_c or the battery is fully discharged, i.e. $E(t) = E_{min}$. The last task being executed can be preempted.

4. Schedule tasks of Γ_g until the battery is fully charged. Analogously, it may be necessary to preempt last task being executed.

5. Repeat step 3 and 4 by scheduling first the preempted task of the right group and so on until the end of the frame workload.

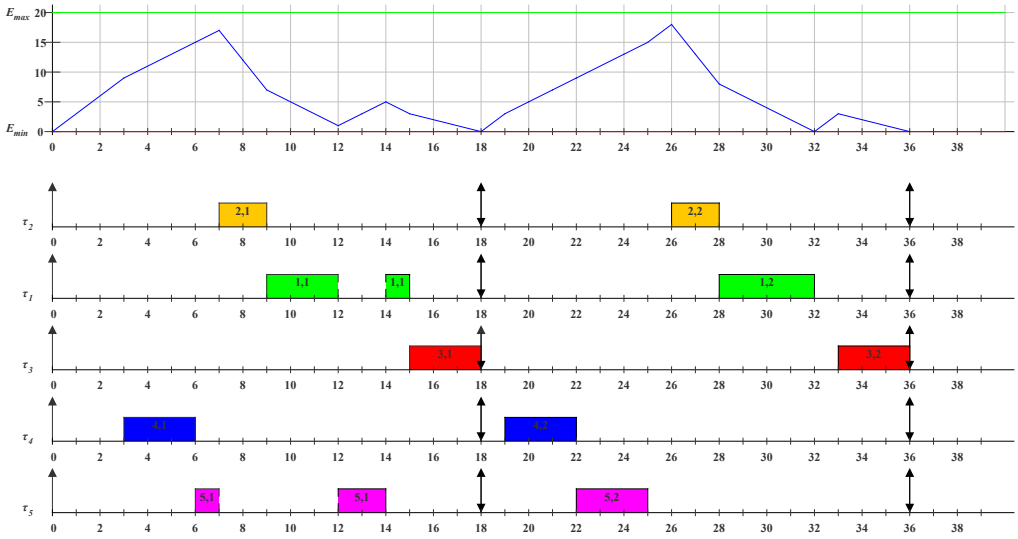
Note that the algorithm can operate even if the system is launched with an empty battery or a different initial battery level. Figure 3.3 on page 78 illustrates an example of FBA algorithm schedule under different battery configurations.

Algorithm 3.1 Scheduling decision of *FBA* Algorithm at time t

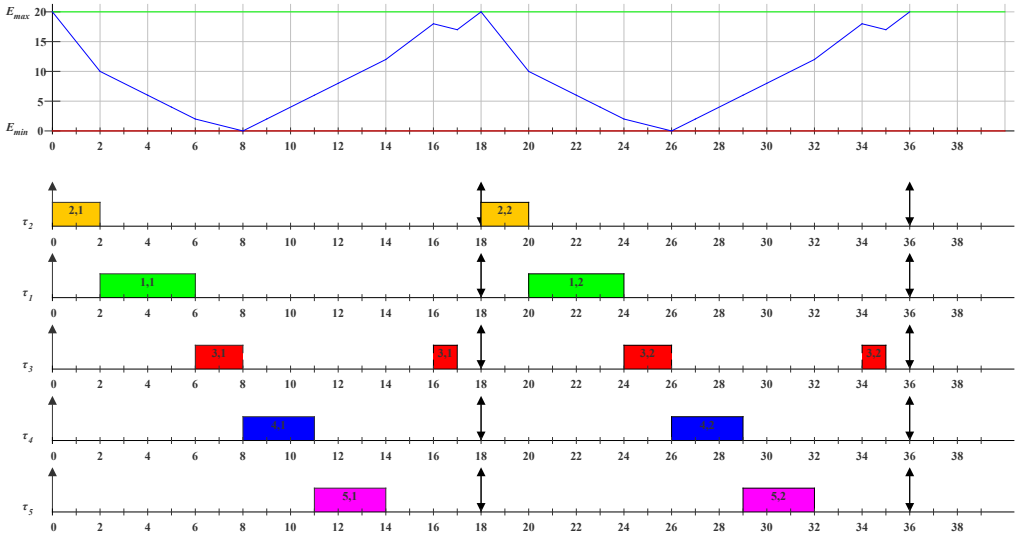
1: $t \leftarrow 0$
2: $exec\Gamma_c = true$
3: $exec\Gamma_g = false$
4: **loop**
5: $\Gamma'_g \leftarrow$ subset of Γ_g of active tasks at time t
6: $\Gamma'_c \leftarrow$ subset of Γ_c of active tasks at time t
7: **if** $\Gamma'_g \neq \emptyset$ And $exec\Gamma_g = true$ **then**
8: $\tau_k \leftarrow$ the first task of Γ'_g
9: execute τ_k
10: **if** $E(t) + P_r > E_{max}$ or $\Gamma'_g = \emptyset$ **then**
11: $exec\Gamma_g = false$
12: $exec\Gamma'_c = true$
13: **end if**
14: **else if** $\mathcal{C}' \neq \emptyset$ And $exec\mathcal{C} = true$ **then**
15: $\tau_k \leftarrow$ the first task of $\mathcal{C}'$
16: **if** $E(t) + P_r >= E_k/C_k$ **then**
17: execute τ_k
18: **else if** $\Gamma'_g = \emptyset$ **then**
19: idle $\left\lceil \frac{|\Gamma_c| - |\Gamma_g|}{P_r} \right\rceil$ time units
20: **end if**
21: **if** $E(t) + P_r - E_k/C_k <= 0$ or $\Gamma'_g = \emptyset$ **then**
22: $exec\Gamma'_c = false$
23: $exec\Gamma_g = true$
24: **end if**
25: **else**
26: idle for one time unit
27: **end if**
28: $t \leftarrow t + 1$
29: **end loop**

-	C_i	E_i	T_i	D_i
τ_1	4	20	18	18
τ_2	2	16	18	18
τ_3	2	16	18	18
τ_4	3	3	18	18
τ_5	3	3	18	18

(a) Task set Γ with $P_r = 3$



(b) $E_{max} = 20$ and $E(0) = E_{max}$



(c) $E_{max} = 20$ and $E(0) = E_{min}$

Figure 3.3: Example of *FBA* scheduling algorithm

Complexity

The *FBA* algorithm is simple, all the instructions are simple and the only operation that can be costly in time is to compute the set of active tasks which can be optimized to a complexity of $\mathcal{O}(n)$ by considering that the tasks are sorted with their consumption rate. Therefore, the global complexity of *FBA* is $\mathcal{O}(n)$.

Minimum Storage Unit Capacity

This property was not addressed in the paper proposing the algorithm but it seems to be obvious due to the simplicity of the model and scheduling schemes. Intuitively, by scheduling all tasks inside one frame with only one request time and one deadline and by following the pattern of gaining-consuming or consuming-gaining tasks, in the worst-case when the battery is empty ($E(t) = E_{min}$), the minimum battery capacity that keeps a schedulable task set schedulable is the maximum instantaneous energy consumption of tasks. Instead of having only one cycle of gaining-consuming tasks, we can keep the same response time by splitting it to many small cycles which lowers the maximum reachable battery level, and thus, the battery capacity. Furthermore, on can also add the energy replenished during inactivity periods in order to ensure the same energy level at the beginning of each frame.

Optimality

This question was not addressed in the paper. Though, the algorithm seems to be optimal for the specified model. Intuitively, when all the tasks share the same period and the same deadline the order of tasks is not important if the deadline is met. This algorithm sorts the tasks so to schedule gaining tasks before consuming ones when the minimum energy level is reached to replenish a maximum of energy before executing the consuming tasks which reduces to the maximum the useless replenishment periods. Then, if a deadline is missed, this means that the available energy in one frame is not sufficient. In this case, no algorithm can avoid this deadline miss.

3.6 EDF-based Scheduling Algorithms

The following algorithms are adaptations of the classical **E**arliest **D**eadline **F**irst (*EDF*) algorithm described in Section 1.4.7 on page 40. These algorithms propose different strategies to cope with the delays caused by energy-harvesting and energy-storage constraints. In this section, we present *EDF* As **L**ate As Possible (*EDL*), *EDF* with **e**nergy **g**uaranty (*EDeg*) and **L**azy **S**cheduling **A**lgorithm (*LSA*) algorithms.

3.6.1 EDF As Late As Possible Scheduling (EDL)

The EDL algorithm was proposed by Silly in [Sil99]. It does not consider energy constraints but schedules tasks by respecting EDF rules and delaying executions as long as possible by anticipating all the available slack-time. It can be used in the context of energy-harvesting systems to maximize replenishment periods.

Model

Task Model

The model associated to EDL is very close to the one described in Section 3.2 on page 67. In order to be able to compute the exact values of slack-time, only periodic tasks with null offsets (i.e. $\forall\ \tau_i,\ O_i = 0$) and constrained or implicit deadlines are considered. Then, the arrival times, energy costs and deadlines are known before run-time. The hyper-period, i.e. the least common multiple of tasks periods, is denoted HP.

Energy Model

The slack-time computation used by EDL algorithm does not include energy constants. Therefore, it can theoretically work with any energy source and storage models. In this dissertation we apply EDL to the general model described in Section 3.2 on page 67 and we use it to compare the effect of delaying execution with other algorithms.

Algorithm Description

The main idea of EDL is to differ executions in order to maximize the length of idle periods and thereby replenishment time. When a job is ready to be executed at time t according to EDF rules, EDL computes first the available slack-time time of the system. Then, the ready job is executed only if the slack-time is 0 and the energy is sufficient to execute. Otherwise, it is delayed until the slack-time is fully consumed. The computation of slack-time with EDF scheduling is described in [Sil99]. The author proposes two approaches: a static offline approach and a dynamic online one. In this section, we give some insights about the static approach. It consists first of identifying in a set denoted $\mathcal{K}$, the time instants when potential idle periods can begin within the hyper-period, and second, computing the length of each of these periods in a set denoted $\mathcal{D}$. Formally:

1. $\mathcal{K}$, the deadlines set is a sorted set of times at which idle times can begin within time interval $[0, HP[$. It is composed of distinct deadlines of all jobs inside $[0, HP[$. Then, $\mathcal{K} = \{\ k_0, k_1, \cdots, k_i, k_{i+1}, \cdots, k_q\ \}$, with $k_i < k_{i+1}$, $k_0 = 0$ and $k_q = HP - \min_{1 \leq i \leq n}(x_i)$, with $x_i = T_i - D_i$. We note that $q \leq N + 1$ where N denotes the number of jobs requested during $[0, HP[$.

2. $\mathcal{D}$, the idle periods lengths set contains the lengths of idle periods. $\mathcal{D} = \{\Delta_0, \Delta_1, \cdots, \Delta_i, \Delta_{i+1}, \cdots, \Delta_q\}$, where Δ_i is the length of the idle period that begins at time k_i. It can be calculated by Formula 3.4:

$$\begin{cases} \Delta_i = \min_{1 \leq i \leq n}(x_i) & if\ i = q \\ \Delta_i = \max(0, F_i) & if\ 0 \leq i \leq q-1 \end{cases} \tag{3.4}$$

with :

$$F_i = (HP - k_i) - \sum_{j=1}^{n} \left\lceil \frac{HP - x_j - k_j}{T_j} \right\rceil \times C_j - \sum_{k=i+1}^{q} \Delta_k$$

Then, *EDL* uses the set $\mathcal{K}$ to decide when to postpone executions and $\mathcal{D}$ to know for how long. Figure 3.4 on the following page shows an example where $\mathcal{K} = \{$ 0, 6, 8, 12, 15, 20, 24, 30, 32, 22 $\}$, and $\mathcal{D} = \{$ 2, 0, 1, 0, 2, 1, 0, 0, 3, 22 $\}$. Then, for example at time 32, *EDL* delays executions for 3 units.

Complexity

We note that the schedule of one hyper-period with *EDL* is achieved in $\mathcal{O}(N \times n)$ operations in the worst case [Sil99]. This complexity depends on the periods and deadlines which make it pseudo-polynomial. It may be relatively high when periods are prime integers because it maximizes the hyper-period length and thus the number of jobs N.

Optimality

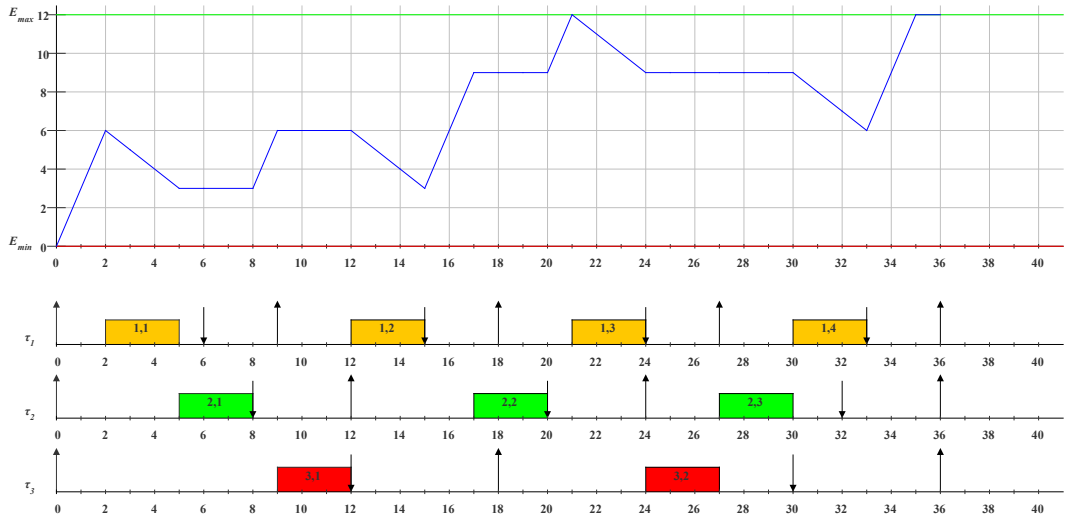
This algorithm is clearly not optimal for energy-harvesting systems because when E_{max} is reached during an idle period, the surplus of energy is wasted. This wasted energy may be necessary for a future job. Figure 3.4(c) on the next page shows a counter example. It illustrates a situation where energy is wasted at time 8 and 15 which leads the system to run out of energy at time 32 while the slack-time was already consumed. However, *EDL* seems to be optimal if we consider an infinite battery capacity or if we resume executions when E_{max} is reached. This intuition is based on the fact that when a deadline is missed at time d, the energy balance, i.e. the difference between the available energy until time d and energy demand of time interval $[0, d[$ is negative. Since *EDF* is driven by deadlines, if energy is not wasted, it is impossible to schedule the task set because there exists at least one interval where an unavoidable negative energy balance happens.

Minimum Storage Unit Capacity

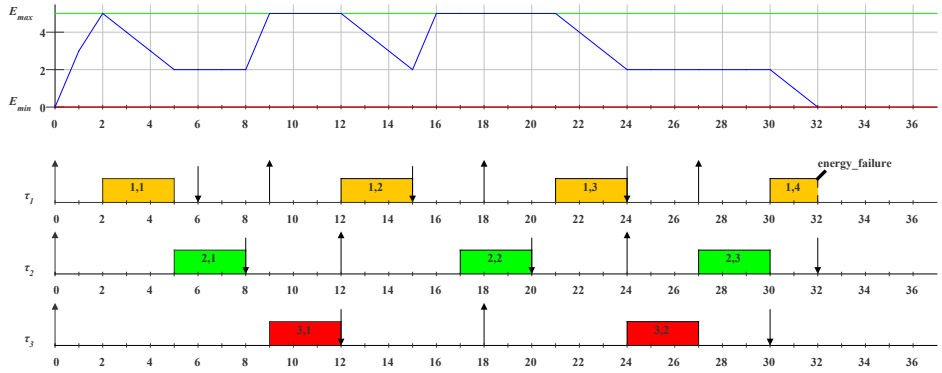
It is not easy to compute the exact battery capacity needed to schedule a feasible task set with *EDL* because we need to know the wort-case scenario that needs the highest capacity. However, one can provide a lower bound of the battery capacity by

-	C_i	E_i	T_i	D_i
τ_1	3	12	9	6
τ_2	3	9	12	8
τ_3	3	9	18	12

(a) Task set Γ with $P_r = 3$



(b) The EDL schedule of Γ with $E_{max} = 12$



(c) The EDL schedule of Γ with $E_{max} = 5$

Figure 3.4: *EDL* scheduling examples

summing the energy collected during all the idle periods of one hyper-period which is the complementary part of the workload of one hyper-period, i.e. $E_{max} \geq (1-U) \times HP \times P_r$. However, it is still an overestimated bound.

Schedulability Conditions

Without energy constraints, *EDL* is equivalent to the regular *EDF* in term of task sets schedulability. When energy constraints are considered, it is difficult to say whether a task set is schedulable with *EDL* or not because we need to check if all deadlines are met even in the worst-case scenario which is not known up to now. The only conditions that we can check are processor and energy utilizations, i.e. $U \leq 1$ and $U_e \leq 1$ which are only necessary conditions.

3.6.2 EDF with Energy Guarantee Scheduling (*EDeg*)

In [EGCC11; Che14], Chetto et al. proposed an enhancement of *EDL*. They proposed an algorithm called *EDeg* for *EDF* with energy guarantee. It consists of mixing soon and late scheduling by using clairvoyance or look-ahead calculations.

Model

Task Model
The same task model as for *EDL* is considered: a periodic task set with constrained or implicit deadlines and null offsets.

Energy Model:
The replenishment function $P_r(t)$ is not necessarily specified but must be known in order to ensure the predictability needed for correct clairvoyance computations. Furthermore, an ideal energy storage unit is considered, i.e. a battery or a capacitor with a linear charging and discharging rates.

Algorithm Description

The intuition behind *EDeg* is to run jobs according to *EDF* rules, but before authorizing a job to execute, *EDeg* uses the notion of *slack-energy* to predict eventual future energy failures. If a future deadline miss due to energy lack is detected, the current jobs are delayed as long as possible by consuming the available *slack-time* to replenish a maximum of energy like *EDL* does. Furthermore, when E_{max} is reached during an idle period, the algorithm resumes executions in order to avoid energy waste. The slack-energy notion is an extension of the notion of slack-time which means that the maximum amount of idle time that can be used to delay executions without violating deadlines. As described in [EGCC11] the slack-energy of a job $J_{i,j}$ at time t denoted $SE_{i,j}(t)$ is the maximum amount of energy that can be used to execute

Algorithm 3.2 *EDeg* Algorithm

```
1: while true do
2:   while there is ready jobs do
3:     while E(t) > E_min and SE(t) > 0 do
4:       execute jobs according to EDF rules
5:     end while
6:     while E(t) < E_max and ST(t) > 0 do
7:       idle the system to replenish energy
8:     end while
9:   end while
10:  while there is no active jobs do
11:    idle the system
12:  end while
13: end while
```

continuously other jobs from time t to the request time of $J_{i,j}$ while still respecting energy requirements and deadlines. It is the difference between the energy available during interval $[t, d_{i,j}]$, i.e. $E(t) + \int_t^{d_{i,j}} P_r(t)\ dt$, and the energy demand of $J_{i,j}$ and higher priority jobs that are requested after $a_{i,j}$ the request time of $J_{i,j}$ and have a deadline earlier than $d_{i,j}$. This energy demand is denoted $A_{i,j}$. Before running the job that is ready to be executed at time t, *EDeg* computes first the system's slack-energy of time t denoted $SE(t)$. The global slack-energy at time t is the minimum of slack-energy of all jobs that are requested after time t and having a deadline earlier than the one of the job being executed at time t. Equation 3.5 summarizes the computation of $SE(t)$.

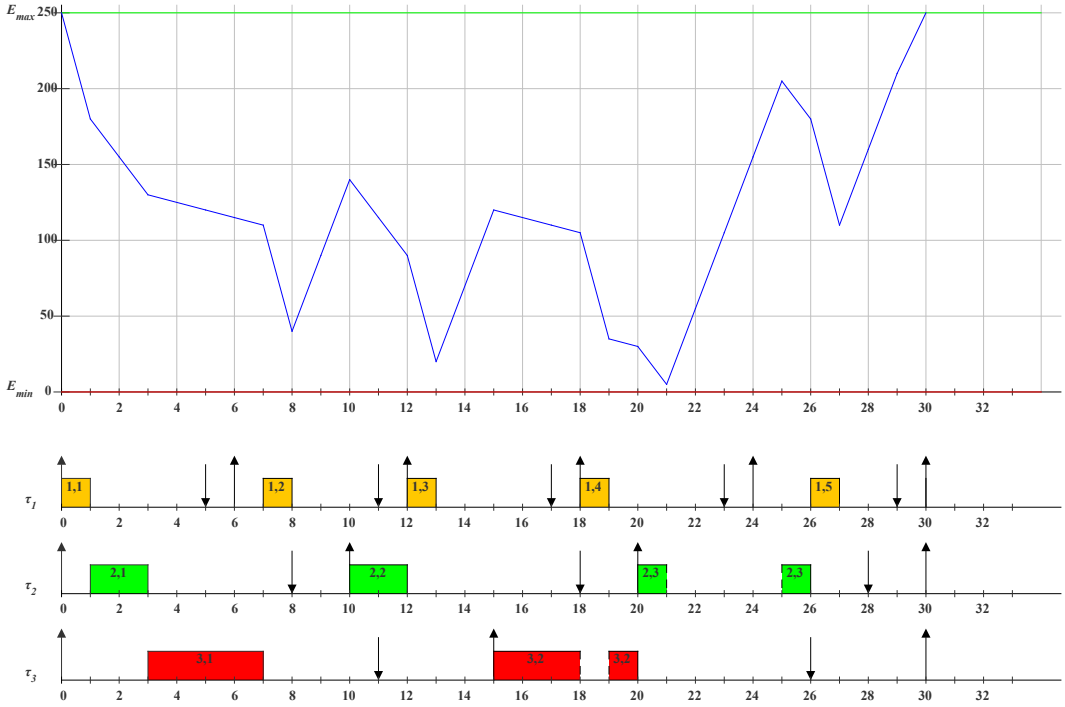
$$\begin{cases} A_{i,j} & = & E_i + \sum_{J_{k,l},\, a_{i,j} < a_{k,l} \leq d_{k,l} \leq d_{i,j}}^{n} E_k \\ SE_{i,j}(t) & = & E(t) + \int_t^{d_{i,j}} P_r(k)\ dk - A_{i,j} \\ SE(t) & = & \min_{J_{k,l},\, t < a_{k,l} \leq d_{k,l} \leq d_{i,j}} (SE_{k,l}) \end{cases} \tag{3.5}$$

Therefore, if $SE(t)$ is positive, the ready job is authorized to execute and consume the energy budget $SE(t)$. Otherwise, it is delayed as much as available slack-time without wasting energy. The system's slack-time denoted $ST(t)$ used by *EDeg* is the same as *EDL*. Algorithm 3.2 summarizes the scheduling schemes of *EDeg*.

Figure 3.5 on the facing page illustrates a typical schedule of *EDeg*. For example at time 15, *EDeg* decides to execute τ_3 because the slack-energy of the system is positive and sufficient to execute few time units. In fact, at time 15, job $J_{3,2}$ has the earliest deadline and is ready to execute. Furthermore, $J_{1,4}$ is the only job that is requested after $J_{3,2}$ and has a deadline earlier than $d_{3,2}$. Then, the slack-energy of the system at time 15 is equal to the one of $J_{1,4}$, i.e. $SE(15) = SE_{1,4}(15) = E(15) + \int_{15}^{d_{1,4}} P_r(t).dt - (E_1) = 120 + 50 \times (23 - 15) - 120 = 400 \geq 0$. With $SE(15) = 400$, the system can execute jobs $J_{3,2}$ and $J_{1,4}$ completely. At time 20, it is no necessary to compute the slack-energy again because there are no jobs that interfere with job $J_{2,3}$. Then, the system executes

-	C_i	E_i	T_i	D_i
τ_1	1	120	6	5
τ_2	2	150	10	8
τ_3	4	220	15	11

(a) Task set Γ with $P_r = 50$



(b) *EDeg* schedule

Figure 3.5: The *EDeg* schedule of Γ with $E_{max} = 250$

$J_{2,3}$ until the battery is not sufficient to execute. At time 21, the battery level is not enough to finish executing, thus, *EDeg* delays executions as long as there is available slack-time, i.e. $ST(20) = 6$. Furthermore, in order to avoid energy losses due to over charging the battery, *EDeg* resumes execution after only 4 units of slack-time. After that, the same behavior is repeated until the end of hyper-period which is 30 time units for this example. We can see that at the end of the hyper-period, the battery is fully charged which means that the task set is schedulable with this configuration of energy.

Complexity

The complexity of *EDeg* comes mainly from slack-time and slack-energy computations. As shown in [Sil99], the slack-time algorithm for *EDF* scheduling is of $\mathcal{O}(N \times n)$ where n is the number of tasks and N is the maximum number of jobs requested

within a hyper-period. Moreover, the complexity of slack-energy algorithm is also of $\mathcal{O}(N \times n)$ as shown in [EGCC11]. Therefore, the complexity of *EDeg* in the worst-case is pseudo-polynomial which might be a serious drawback in practice.

Optimality

EDeg was proved to be optimal for periodic and synchronous task sets in [Che14]. This optimality is inherited from the optimality of *EDF* for Liu and Layland's model. Since *EDeg* predicts future deadline misses due to energy lack and does not waste energy when delaying executions, it takes advantage of the dynamic priority to schedule the earliest deadline first and maximizes the supportable workload. If a deadline is missed with *EDeg*, this means the energy is not sufficient which makes the task set unschedulable by any other algorithm.

Minimum Storage Unit Capacity

Intuitively, one can say that the maximum battery capacity needed for *EDeg* is the same as *EDL* since we can apply the same reasoning on idle periods. Then, the same pessimistic bound can be applied, i.e. the sum of the energy that can be collected during all idle periods of one hyper-period.

Schedulability Conditions

A schedulability test for *EDeg* was proposed in [Che14]. It consists of checking if the energy available during any time interval is sufficient to satisfy the energy demand of the same time interval. Knowing that this can be checked by comparing the energy utilization to 1 for tasks with constrained or implicit deadlines, a task set is said schedulable with *EDeg* if and only if processor and energy utilizations are lesser or equal to one, i.e. $U \leq 1$ and $U_e \leq 1$. Furthermore, knowing that *EDeg* is optimal, this test becomes a necessary and sufficient feasibility condition for synchronous periodic task sets with implicit deadlines, and only necessary for task sets with constrained deadlines.

3.6.3 Lazy Scheduling Algorithm (*LSA*)

The Lazy Scheduling Algorithm (*LSA*) is one of the main EDF-based algorithms proposed in the literature. It was proposed for the first time by Moser et al. in [Mos+06b] and was revisited in [Mos+06a; Mos+07]. It is an energy-driven algorithm in contrast to classical algorithms which are time-driven. This means that scheduling decisions are made only according to the available energy, thus, if energy is sufficient, time will be sufficient too. This comes from the fact that all tasks consume energy with the same rate and that tasks execution times depend on the consumption rate authorized by the system. This algorithm is interesting to study because it has a reasonable overhead and is optimal for the considered model.

Model

Task Model

The task set considered by *LSA* is composed by n independent, preemptive and aperiodic tasks. Each task is characterized by its energy cost E_i and its relative deadline D_i. The tasks drain energy from the storage unit with the same consumption function denoted $P_D(t)$. This function is upper bounded by the maximum consumption rate P_{max}. Moreover, it is up to the scheduling algorithm to decide which consumption rate to apply at time t. Knowing that *LSA* is energy-driven, the execution time C_i of a task depends on its energy cost and the applied consumption rate. Then, when the consumption rate is maximized, C_i is minimized, i.e. $P_D(t) = P_{max} \Rightarrow C_i = E_i/P_{max}$. Therefore, as mentioned earlier, the execution time of a task varies from one job to an other depending on the applied consumption rate. Then, a job $J_{i,j}$ that is requested at time $a_{i,j}$, starts its execution at time $s_{i,j}$ and must finish executing at time $f_{i,j}$ before its deadline $d_{i,j}$ after the energy cost of the job has been consumed, thus, we have $a_{i,j} \leq s_{i,j} < f_{i,j} \leq d_{i,j}$.

Energy Model

In [Mos+06a; Mos+07], the authors considered the solar energy as energy source. Then, to cope with the fluctuations of the output energy that characterizes a photovoltaic cell, they proposed to use the **E**nergy **V**ariability **C**haracterization **C**urves (EVCC) that bound the energy harvested in any time interval Δ: ϵ^l for the lower bound curve. The EVCCs are extracted from actual solar energy traces to provide guarantees on the produced energy. Therefore, the replenishment function denoted $P_r(t)$ is lower bounded by a known function, and the energy harvested in a certain time interval $[t_1, t_2]$ denoted $g(t_1, t_2)$ is given by as:

$$g(t_1, t_2) = \int_{t_1}^{t_2} P_r(t).dt \geq \epsilon^l(t_1, t_2)$$

For the storage unit, they considered an ideal rechargeable battery of capacity $\mathcal{C}$ that can be charged and discharged with a linear function. Its energy level at time t is denoted $E(t)$ and must stay between the two thresholds E_{min} and E_{max} with $\mathcal{C} = E_{max} - E_{min}$.

Algorithm Description

The *LSA* algorithm is EDF-based, thus, it schedules aperiodic jobs according to earliest deadline first rules. The only difference is the time $s_{i,j}$ when a job $J_{i,j}$ begins effectively its execution. This time is computed such that energy constraints are respected, namely the energy cost and the battery capacity limit. Indeed, knowing that all tasks have the same consumption rate, $s_{i,j}$ is the earliest time at which the system can start executing after a replenishment period. This execution is continuous and includes $J_{i,j}$ and a maximum of eventual higher priority jobs. This execution period

Algorithm 3.3 *LSA* Algorithm

```
1: P_D(t) = 0
2: while true do
3:    t ← current time
4:    J_{i,j} the jobs with the earliest deadline at time t
5:    calculate s_{i,j}
6:    if t = a_{i,j} then
7:       P_D(t) = 0
8:    end if
9:    if E(t) = E_max then
10:      P_D(t) = P_r(t)
11:   end if
12:   if t ≥ s_{i,j} then
13:      P_D(t) = P_max
14:   end if
15:   execute J_{i,j} with power P_D(t)
16: end while
```

ends when the pending workload is executed or when the deadline $d_{i,j}$ is reached. The execution must finish without running out of energy and without exceeding the battery capacity $\mathcal{C}$. Then, if there are no interferences within time interval $[a_{i,j}, d_{i,j}]$, starting executing $J_{i,j}$ at time $s^1_{i,j} = d_{i,j} - E_i/P_{max}$ is enough. However, knowing that tasks are aperiodic, higher priority jobs can be requested at any moment. Thus, executing as late as possible may lead to miss deadline $d_{i,j}$ which makes $s^1_{i,j}$ not safe. Then, an earlier $s_{i,j}$ is mandatory, a time from which job $J_{i,j}$ and a maximum of higher priority jobs can be executed before $d_{i,j}$ without running out of energy. We call this time $s^*_{i,j}$, it is the time from which we can execute and consume all the energy available in interval $[a_{i,j}, d_{i,j}]$. Equation 3.6 shows how to compute $s^*_{i,j}$.

$$s^*_{i,j} = d_{i,j} - \frac{E(a_{i,j}) + g(a_{i,j}, d_{i,j})}{P_{max}} \tag{3.6}$$

Moreover, if E_{max} is reached before $s^*_{i,j}$, some energy may be wasted. This energy is included in $s^*_{i,j}$ but actually wasted which may lead to a lack of energy before $d_{i,j}$. Therefore, an earlier time is needed for safe executions. Equation 3.7 shows the relationship between $s'_{i,j}$, the time from which starting executing does not lead to a waste of energy, and the other parameters.

$$g(a_{i,j}, s'_{i,j}) - \mathcal{C} = g(a_{i,j}, d_{i,j}) + (s'_{i,j} - d_{i,j})P_{max} \tag{3.7}$$

Therefore, the optimal starting time $s_{i,j}$ is the maximum between $s'_{i,j}$ and $s^*_{i,j}$, i.e. $s_{i,j} = \max(s'_{i,j}, s^*_{i,j})$.

The pseudo-code of *LSA* algorithm is described by Algorithm 3.3. It is based on the following rules.

- **Rule 1**: *EDF* scheduling is used with $P_D(t) = P_{max}$ when $t \geq s_{i,j}$;
- **Rule 2**: if E_{max} is reached, jobs are run with $P_D(t) = P_r(t)$ in order to avoid energy waste

Complexity

The *LSA* algorithm has the same complexity as *EDF* since it adds only the computation of $s_{i,j}$ of one job and some verifications, which can be done in $\mathcal{O}(1)$. The most costly operation is to select the job to execute. This operation can be done in $\mathcal{O}(n)$ in the worst-case where n is the number of tasks. Therefore, the complexity of *LSA* is $\mathcal{O}(n)$.

Optimality

In [Mos+07], the *LSA* algorithm was proved to be optimal for the specific model described earlier in this section. In fact, the *LSA* algorithm can miss deadlines only in two possible situations: when the time is not sufficient to satisfy the workload or when the available energy is not sufficient to satisfy the demand of energy.

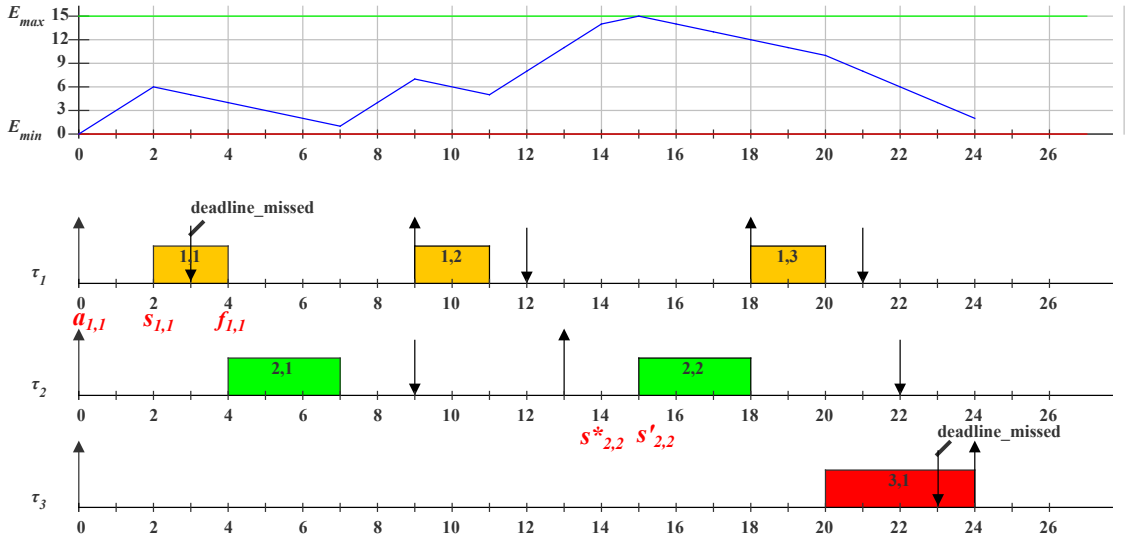
- A deadline cannot be respected if the time is not sufficient to assign the available energy with the maximum consumption rate P_{max}. In this case, unprocessed energy remains in the storage unit and we have $E(d_{i,j}) > E_{min}$.
- A deadline violation occurs also because the required energy is simply not available at the deadline. In this case, the battery is exhausted before deadline, i.e. $E(d_{i,j}) \leq E_{min}$.

These are the insights of *LSA* optimality, the complete and detailed proof is available in [Mos+07].

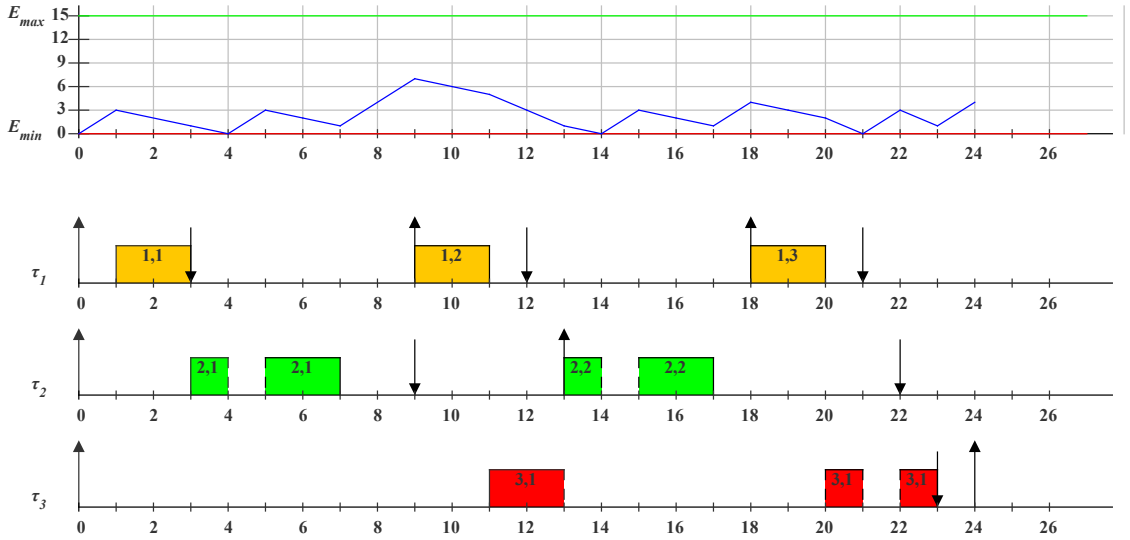
Note that the optimality of *LSA* relies strongly on the assumption of having the same consumption function $P_D(t)$ for all the tasks. Then, if we apply different consumption rates to each task, the computation of the starting time $s_{i,j}$ is no longer exact. In fact, if we suppose that each task has its own consumption rate denoted $P_i(t)$, considering $\max_{1 \leq i \leq n}(P_i(t))$ as P_{max} leads to a later starting time $s_{i,j}$ than the optimal one. Furthermore, considering $\min_{1 \leq i \leq n}(P_i(t))$ as P_{max} leads to an earlier starting time than the optimal one which may lead to avoidable energy failures or deadline misses. Figure 3.6 on the following page shows a *LSA* schedule of a task set with different consumption rates where a deadline is missed while a valid schedule is possible. We can see in Figure 3.6(b) on the next page that $s_{1,1}$ of job $J_{1,1}$ is computed at time 0 with $P_{max} = \max_{1 \leq i \leq n}(P_i(t))$ which gives $s_{1,1} = 3 - \frac{0+3\times(3-0)}{5} = 2$. This value of $s_{1,1}$ leads to a deadline miss because the energy demand is overestimated, and therefore, $s_{1,1}$ is over delayed. We can see that the consumption rate used to compute $s_{i,j}$ is very important because it influences the earliness or the tardiness of execution starting time. *LSA*

-	C_i	E_i	T_i	D_i	$P_i(t)$
τ_1	2	8	9	3	4
τ_2	3	12	13	9	4
τ_3	4	20	24	23	5

(a) Task set Γ with $P_r = 3$



(b) *LSA* schedule with $P_{max} = \max(P_i(t))$



(c) Feasible schedule

Figure 3.6: *LSA* optimality counter example

computes precisely $s_{i,j}$ only if all tasks consume energy with the same rate. Otherwise, it is impossible or at least difficult to find the optimal consumption rate to use when tasks consume energy with different rates.

Minimum Storage Unit Capacity

When a task set is schedulable with *LSA*, it is possible to optimize the battery capacity while keeping the task set schedulable. In [Mos+06b], the authors shows how to compute the minimum battery capacity needed to keep the schedulability of the task set with *LSA*. It consists of the maximum difference between the energy demand and the energy available in a sliding time interval Δ. The energy demand of any interval of time Δ according to *LSA*, which is EDF-based algorithm, can be obtained by applying the demand bound function (see Definition 1.11 on page 42) on periodic tasks. Then, the system energy demand of time interval Δ denoted $A(\Delta)$ can be obtained by Equation 3.8.

$$A(\Delta) = \sum_{i=1}^{n} dbf_i(\Delta) \tag{3.8}$$

Moreover, the energy that can be harvested in any time interval Δ is lower bounded by EVCCs, i.e. $\epsilon^l(\Delta)$. Therefore, the minimum battery capacity needed to schedule a periodic task set according to *LSA* rules denoted $\mathcal{C}_{min}$ is given by Formula 3.9.

$$\mathcal{C}_{min} = \max_{\forall\ \Delta \geq 0}(0, A(\Delta) - \epsilon^l(\Delta)) \tag{3.9}$$

More detailed explanations and proofs are available in [Mos+06b]. Note that in the absence of a worst-case situation, computing $\mathcal{C}_{min}$ in practice requires checking $A(\Delta)$ and ϵ^l of all possible Δ intervals which is a serious limitation.

Schedulability Conditions

A schedulability test of *LSA* was also proposed in [Mos+06b]. Knowing that *LSA* is EDF-based, it is obvious that the accepted workload is maximized. Then, if a task set is schedulable with *EDF* without considering energy constraints, then, it remains only to check whether the energy provided by ϵ^l is sufficient to satisfy the workload energy demand. In [Mos+06b] the authors built an admittance test by comparing the energy demand $A(\Delta)$ of time window Δ to the available energy that can be stored in the same time interval, Formula 3.10 describes formally the condition a task set must satisfy to be schedulable wih *LSA*.

$$A(\Delta) \leq \min_{\forall\ \Delta \geq 0}(\epsilon^l(\Delta) + \mathcal{C}, P_{max} \times \Delta) \tag{3.10}$$

Again, this test requires to check all the possible Δ intervals which is difficult to do in practice.

3.7 Fixed-Priority Scheduling

Most of available research addressing real-time scheduling for energy-harvesting systems focuses mainly on EDF-based solutions because of the optimility of EDF for Liu and Layland's model. Nevertheless, some fixed-task-priority algorithms have been proposed in the literature that we describe in this section.

3.7.1 The PFP_{ALAP} Scheduling Algorithm

Preemptive Fixed-Priority As Late As Possible (PFP_{ALAP}) is a fixed-priority scheduling policy that delays jobs executions as long as possible, i.e. as long as available slack-time, in order to replenish the battery as much as possible before authorizing executions. We studied the properties of this algorithm in [CAM12].

Model

The model applied to this algorithm is the same as for EDL algorithm which is described in Section 3.6.1 on page 80. Since the exact computation of slack-time requires exact knowledge about jobs arrival times and deadlines, only periodic tasks with constrained or implicit deadlines are considered. Moreover, the slack-time computation for fixed-priority does not consider energy constraints, thus, the energy model is not necessarily specified. Therefore, we can adopt the general model described in Section 3.2 on page 67.

Algorithm Description

The **P**reemptive **F**ixed-Task-**P**riority **A**s **L**ate **A**s **P**ossible (PFP_{ALAP}) algorithm is the fixed-task-priority counter-part of EDL. Its main operation is the dynamic computation of slack-time. The slack-time algorithm used for PFP_{ALAP} is the one proposed by Davis et al. in [DTB93]. Each priority level has its own slack-time at time t denoted $ST_i(t)$. For priority level-i, it consists of computing the length of idle periods that can be merged in once within the time window starting from time t and ending at the next deadline of level-i. Algorithm 3.4 on the facing page shows how to compute $ST_i(t)$.

Notations:

- $d_i(t)$: the next absolute deadline of priority level-i after time t,
- $a_i(t)$: the next activation time of priority level-i,
- $c_i(t)$: the remaining execution time at time t of the current job of priority level-i,
- $idle(t_1, t_2)$: the amount of time the processor is idle within an interval of time,
- $W_i^m(t)$: the recurrent workload function.

Algorithm 3.4 Slack-time of priority level-i at time t

$ST_i(t) \leftarrow 0$
$W_i^{m+1}(t) \leftarrow 0$
repeat
  $W_i^m(t) \leftarrow W_i^{m+1}(t)$
  $W_i^{m+1}(t) \leftarrow t + ST_i(t) + \sum_{\forall j \leq i} \left(c_j(t) + \left\lceil \frac{(W_i^m(t) - a_j(t))_0}{T_j} \right\rceil \times C_j \right)$
  if $W_i^m(t) = W_i^{m+1}(t)$ **then**
    $idle(t, d_i(t)) \leftarrow \min((d_i(t) - W_i^m(t))_0, \min_{\forall j \leq i} \left(\left\lceil \frac{W_i^m(t) - a_j(t)}{T_j} \right\rceil \times T_j + a_j(t) - W_i^m(t) \right)$
    $ST_i(t) \leftarrow ST_i(t) + idle(t, d_i(t))$
    $W_i^{m+1}(t) \leftarrow W_i^{m+1}(t) + idle + \epsilon$
  end if

until $W_i^{m+1}(t) \leq d_i(t)$
return $ST_i(t)$

Algorithm 3.5 PFP_{ALAP} Algorithm

$t \leftarrow 0$
loop
  if $ST(t) \geq 0$ **then**
    $\tau_k \leftarrow$ the highest priority active task at time t
    execute τ_k
  else
    idle the system to replenish energy
  end if
  $t \leftarrow t + 1$
end loop

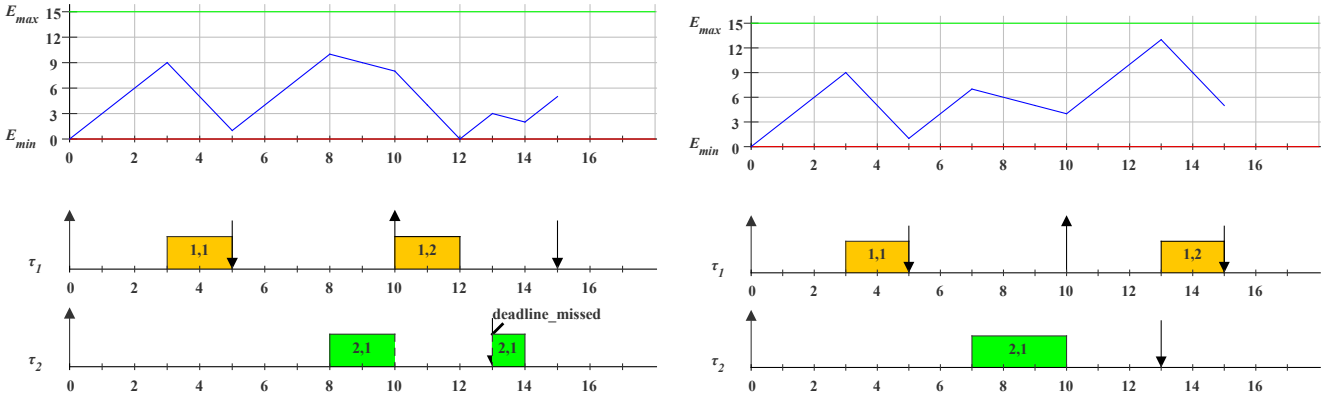
Moreover, the global slack-time of the system at time t denoted $ST(t)$ is the minimum slack-time among all priority levels, i.e. $ST(t) = \min_{1 \leq i \leq n}(ST_i(t))$. Then, before authorizing tasks to be executed according to their priorities, PFP_{ALAP} computes $ST(t)$, and if it is positive, executions are delayed until $ST(t)$ is totally consumed. Otherwise, it executes jobs as the classical FTP scheduler. Algorithm 3.5 shows how does PFP_{ALAP} take scheduling decisions at time t.

Complexity

The complexity of PFP_{ALAP} in the worst-case scenario is the same as the slack-time computation algorithm, which has a complexity of $\mathcal{O}(m \times n)$ where m is the number of iterations and n is the number of tasks. We note that the number of

-	C_i	E_i	T_i	D_i	P_i
τ_1	2	14	10	5	1
τ_2	3	12	18	13	2

(a) Task set Γ with $P_r = 3$



(b) PFP_{ALAP} schedule

(c) Feasible schedule

Figure 3.7: PFP_{ALAP} optimality counter example

iterations m depends on the periods and deadlines of the tasks and is bounded by $\max_{1\leq i\leq n}(T_i)/\max_{1\leq i\leq n}(D_i)$. Thus, the complexity of PFP_{ALAP} is pseudo-polynomial.

Optimality

The idea behind PFP_{ALAP} is that delaying executions as la late as possible maximizes the energy replenished before starting executions which seems to be beneficial for the task set schedulability, especially when the battery capacity is sufficient. Unfortunately, this is one of the counter intuitive ideas in fixed-priority scheduling for energy-harvesting systems. In fact, a counter example can be built by constructing a schedule where a negative energy balance, i.e. difference between available energy and jobs energy demand from time 0 to the absolute deadline of the current job, happens to a lower priority job $J_{i,j}$. When this job suffers many interferences from higher priority jobs that are requested after $J_{i,j}$ and having deadlines later than $J_{i,j}$, the late scheduling may anticipate the executions of all higher priority jobs which may lead to an energy failure or a negative energy balance.

Figure 3.7 shows such a counter example. In Figure 3.7(b), the as late as possible schedule is respected, however, at time 13, the energy is not sufficient to finish executing τ_2 and the algorithm delays executions further which leads to a deadline miss. This is due to the fact that the energy balance at time 12 is negative, i.e. $P_r \times (d_{2,1} - 0) - 2 \times E_1 - E_2 =$

-1, and that the slack-time computation does not take into account tasks energy cost. Furthermore, by executing τ_2 earlier, the task set becomes feasible. Therefore, PFP_{ALAP} cannot be optimal.

Minimum Storage Unit Capacity

For the same reasons as EDL algorithm, it is difficult to estimate the minimum battery capacity needed to keep a schedulable task set schedulable. Indeed, the worst-case scenario that maximizes the needed energy storage capacity is not known up to now.

Schedulability Conditions

Without energy constraints, PFP_{ALAP} has the same schedulability as the classical FTP algorithm. However, when energy constraints are considered, it is difficult to check the schedulability of a task set with PFP_{ALAP} algorithm without simulating the schedule for at least a hyper-period. This is due to the fact that it is difficult to find the worst-case scenario that leads to the worst response times of tasks.

3.7.2 Preemptive Fixed-Priority with Slack-Time Algorithm

Among several heuristics proposed by Chetto et al. in [CMM11], the Preemptive Fixed-Priority with Slack-Time (PFP_{ST}) algorithm was the most pertinent and the one with the highest schedulability rate. It alternates classical FTP schedule with the one of PFP_{ALAP} when battery boundaries are reached.

Model

The model considered in [CMM11] for **P**reemptive **F**ixed-Task-**P**riority with **S**lack-**T**ime (PFP_{ST}) is the same as the one described in Section 3.2 on page 67.

Algorithm Description

The behavior of PFP_{ST} algorithm is simple, it schedules jobs according to their priorities as soon as possible when the battery level is higher than E_{min}. Then, it delays executions as long as available slack-time in order to replenish a maximum of energy before resuming executions. The slack-time algorithm is the same as the one used for PFP_{ALAP}. Moreover, executions are resumed when E_{max} is reached during an idle period in order to avoid energy losses. Algorithm 3.6 on the next page summarizes the scheduling schemes of PFP_{ALAP}.

Algorithm 3.6 PFP_{ST} Algorithm

```
1: while true do
2:   while there is ready jobs do
3:     while E(t) > E_min do
4:       execute jobs according to fixed-priority rules
5:     end while
6:     while E(t) < E_max and ST(t) > 0 do
7:       idle the system to replenish energy
8:     end while
9:   end while
10:  while there is no active jobs do
11:    idle the system
12:  end while
13: end while
```

Complexity

The complexity of PFP_{ST} in the worst-case is the same as PFP_{ALAP} because both use the same slack-time algorithm which is pseudo-polynomial [DTB93]. However, in average, PFP_{ST} behaves better than PFP_{ALAP} because it does not need to compute slack-time every time as PFP_{ALAP} does.

Optimality

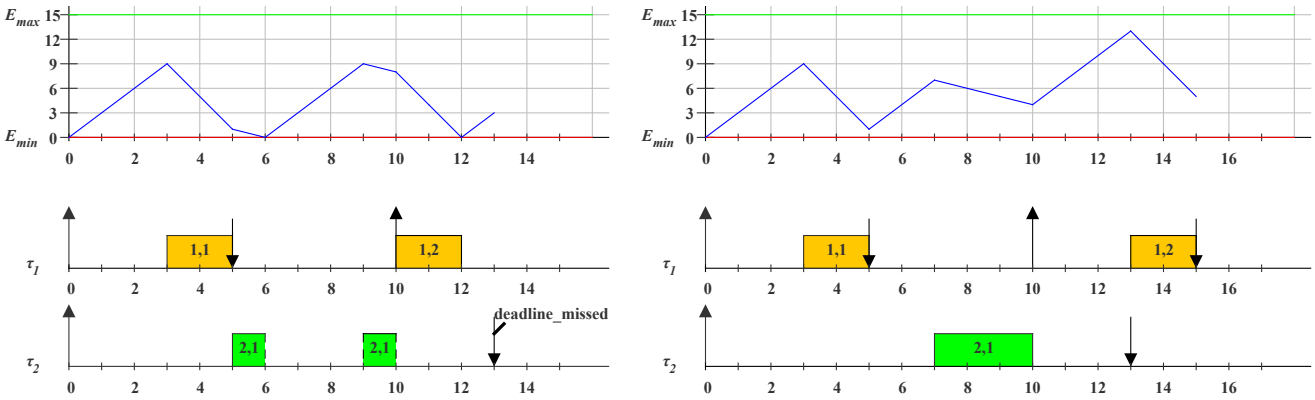
Unfortunately, PFP_{ST} is not optimal because it suffers the same problem as PFP_{ALAP}. Since the computation of slack-time does not consider energy constraints, the same counter example can be used to prove its non optimality. Figure 3.8 on the facing page shows the schedule of the task set described in Figure 3.7(a) on page 94 according to PFP_{ST} rules. We can see that the same negative energy balance occurs at time 13 while it is possible to avoid it as shown in Figure 3.8(b) on the facing page.

Minimum Storage Unit Capacity

Since PFP_{ST} behaves in the worst case like PFP_{ALAP}, the minimum battery capacity problem is the same for both algorithms. Therefore, the exact minimum battery capacity is not known up to now.

Schedulability Conditions

The PFP_{ST} algorithm provides a better schedulability rate than PFP_{ALAP} because it avoids energy losses due to overcharging the battery. Unfortunately, it is not easy to provide a schedulability condition without simulating the schedule for at least one hyper-period because, again, the worst-case scenario is not known.



(a) PFP_{ST} schedule

(b) Feasible schedule

Figure 3.8: PFP_{ST} optimality counter example

3.7.3 Other Scheduling Heuristics

Other scheduling heuristics have been proposed in [CMM11], however, the simulations presented in [CMM11] showed that most of them are not pertinent. In this section we describe some of them briefly.

$EHFP_1$**:** All jobs are executed as soon as possible according to the fixed-priority (FTP) rules until E_{min} is reached or there are no more tasks ready to be executed. If E_{min} is reached when there is at least one ready task, the processor is switched to idle mode for x time units where x is a parameter of the heuristic. During that period, the energy storage unit replenishes. The simulations showed that the longer x is, the worse schedulability is, and that the best values of x are less than 10.

$EHFP_2$**:** All jobs are executed as soon as possible according to FTP. When E_{min} is reached, the processor is switched to idle mode until the battery level reaches a threshold level called E_{th} and given as a parameter of the heuristic. By varying E_{th}, simulations showed that this heuristic is similar to $EHFP_1$.

$EHFP_5$: For this heuristic, there are two threshold parameters, namely E_{th}^{min} and E_{th}^{max}. All tasks execute as soon as possible according to FTP scheduling. When the battery level reaches E_{th}^{min}, executions are suspended as long as available slack-time and the threshold E_{th}^{max} is not exceeded. Simulations showed that this heuristic increases the number of preemptions and does not improve the schedulability rate.

Algorithm family			EDF-based				Fixed-Task-Priority	
Algorithm			*EDL*	*EDeg*	*LSA*	*FBA*	PFP_{ALAP}	PFP_{ST}
Model			periodic	periodic	aperiodic	frame-based	periodic	periodic
Complexity			pseudo-polynomial	pseudo-polynomial	$\mathcal{O}(n)$	$\mathcal{O}(n)$	pseudo-polynomial	pseudo-polynomial
Preemptions rate			high	medium	medium	medium	high	medium
Average energy level			high	medium	medium	medium	high	medium
C_{min}			-	-	known	known	-	unknown
Schedulability condition			-	known	known	known	-	-
Optimality	Frame-Based	$E_i/C_i = Cst$	not optimal	optimal	optimal	optimal	not optimal	not optimal
		$E_i/C_i = f(\tau_i)$	not optimal	optimal	not optimal	optimal	not optimal	not optimal
	Periodic	$E_i/C_i = Cst$	not optimal	optimal	optimal	not optimal	not optimal	not optimal
		$E_i/C_i = f(\tau_i)$	not optimal	optimal	not optimal	not optimal	not optimal	not optimal

Table 3.1: Algorithms comparison summary

3.8 Performance Evaluation and Comparison

We described the main scheduling algorithms available in the state of the art. In this section we evaluate these algorithms through simulations over some metrics in order to show their strengths and weaknesses.

3.8.1 Simulation configuration

In this subsection, we describe the configuration of the experiments, namely the task sets generation, the parameters and the assumptions.

Competitors

- the EDL algorithm described in Section 3.6.1 on page 80,
- the $EDeg$ algorithm described in Section 3.6.2 on page 83,
- the LSA algorithm described in Section 3.6.3 on page 86,
- the FBA algorithm described in Section 3.5 on page 75,
- the PFP_{ALAP} algorithm described in Section 3.7.1 on page 92 with deadline monotonic tasks priority ordering,
- the PFP_{ST} algorithm described in Section 3.7.2 on page 95 with deadline monotonic tasks priority ordering.

Task Models

As mentioned in the above sections, each algorithm is designed for a specific task model. In order to compare them against each other we evaluate them over different task models and assumptions. For this experiment, we select three main models:

- the frame-based model described in Section 3.5 on page 75,
- the periodic task model with implicit deadlines described in Section 3.2 on page 67,
- the periodic task model with constrained deadlines.

For each task model, we run simulations over four energy configurations that cover the main assumptions that make the evaluated algorithms optimal:

- the same consumption rate for all tasks with an infinite battery capacity, i.e. E_i/C_i=$Constant$ and $E_{max} = \infty$,
- the same consumption rate for all tasks with an arbitrary limited battery capacity, i.e. E_i/C_i=$Constant$ and E_{max}=$Constant$,

- variable consumption rates with an infinite battery capacity, i.e. $E_i/C_i = f(\tau_i)$ and $E_{max} = \infty$,
- variable consumption rates with a limited battery capacity, i.e. $E_i/C_i = f(\tau_i)$ and E_{max}=*Constant*.

Assumptions

This experiment considers the following hypothesis and assumptions:

- in order to focus only on the impact of energy on task sets schedulability, we consider only time feasible task sets,
- since actual machines cannot handle continuous values, we consider that time and energy are discretized and that the granularity is small enough to avoid decimal values, i.e. time and energy are integers and all scheduling operations are performed before or after one time unit,
- knowing that most of the evaluated algorithms require constant or at least known replenishment functions, we consider only constant charging functions $P_r(t) = P_r$ in order to make the comparison possible,
- tasks consume energy linearly, i.e. a task consumes E_i/C_i energy units for each execution time unit.

Task set generation

For each task model, we generate randomly a sample of 15000 task sets using an adapted version of the *U-Unifast Discard* algorithm proposed in [BB05]. This algorithm is coupled with the hyper-period limitation technique proposed in [MG01]. In order to cover most of possible task sets, we generate them according to their processor and energy utilizations, i.e. U and U_e. Indeed, we vary U and U_e within interval $[0.2, 1]$ with steps of 0.05. Then, we obtain 50 distinct task sets for each couple (U, U_e).

Tasks are generated by considering a battery capacity of 200 units and a constant energy replenishment rate $P_r = 15$ units.

Metrics

In this experiment we compare the selected algorithms over the following metrics.

- *Failure rate*: is the percentage of non feasible task sets among all the tested ones. Then, the greater the failure rate is, the lower the algorithm performance are. This metric helps to confirm the optimality or the non optimality of some algorithms.
- *Preemption rate*: for one simulation, it is the number of preemption events. A preemption event occurs when a job is stopped before finishing its execution. All

of the events that occur at the same instant are considered as once. Therefore, the number of possible events is bounded by the number of time units composing the simulation, i.e. the simulation duration. For several simulations, this metric is computed only for feasible task sets and represents the ratio of the average number of preemptions relative to all possible events. Then, the greater the number of preemptions is, the greater the context switch is. Note that this leads to increase the overhead cost and decreases the performance. The shown results represent the percentage of the average number of preemption events among the total number of events.

- *Average Energy Level*: represents the average energy level of the battery or capacitor at any instant during the simulation. It is the average of the energy level observed for all scheduling events. The best algorithm relative to this criterion is the one that maximizes the average energy level. This means that the algorithm makes the system less energy constrained.

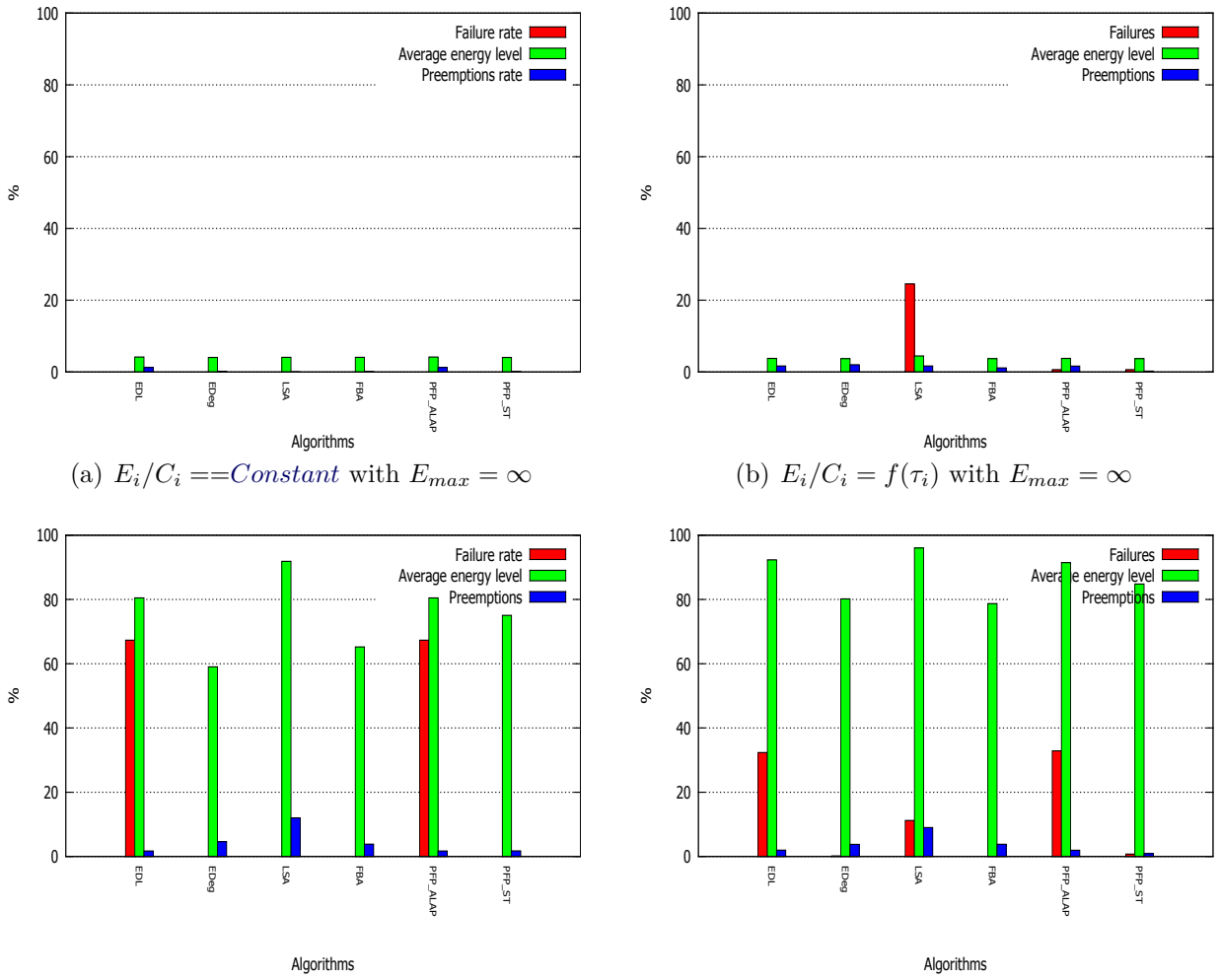
The task sets are launched synchronously with an empty battery, i.e. $\forall\ \tau_i,\ O_i = 0$ and $E(t) = E_{min}$ which seems to be the closest to the actual worst-case scenario of all the evaluated algorithms.

3.8.2 Results analysis

Figure 3.9 on the next page shows the performances of the tested algorithms on the frame-based task model. As expected most of EDF-based algorithms dominates fixed-priority ones. We can see also that FBA and $EDeg$ have the lower failure rates. This was expected because in frame-based model, all tasks share the same period and the same deadline. Thus, ordering tasks according to deadlines or periods does not change the schedule since the used breakdown rule is to give the priority to jobs with lower energy consumption rate. Note that this rule is not respected when considering only deadlines as PFP_{ALAP} and PFP_{ST} do. Furthermore, we observe that LSA is not optimal for this model especially when the energy consumption rate varies from one task to an other. This is due to the fact that LSA considers only the maximum consumption rate to compute the latest time to execute a job.

Figures 3.10 on page 103 and 3.11 on page 104 show the performances of the algorithms on the periodic model in different energy and deadline configurations. Then, we can see that FBA behaves badly when tasks have different periods and different deadlines. Again, when tasks consumption rates can vary, LSA can take wrong decisions by overestimating the latest time form which starting executing a job does not lead to a deadline miss. Note that the constrained deadlines increase the rate of unfeasible task sets for all algorithms which is expected because constrained deadlines let shorter time to jobs to finish their executions than implicit deadlines.

Globally, $EDeg$ is the algorithm with the lower failure rate and preemptions rate over the tested task models with a good average battery level. This is due to its anticipation



(a) E_i/C_i ==*Constant* with $E_{max} = \infty$

(b) $E_i/C_i = f(\tau_i)$ with $E_{max} = \infty$

(c) E_i/C_i=*Constant* with $E_{max} = 200$

(d) $E_i/C_i = f(\tau_i)$ with $E_{max} = 200$

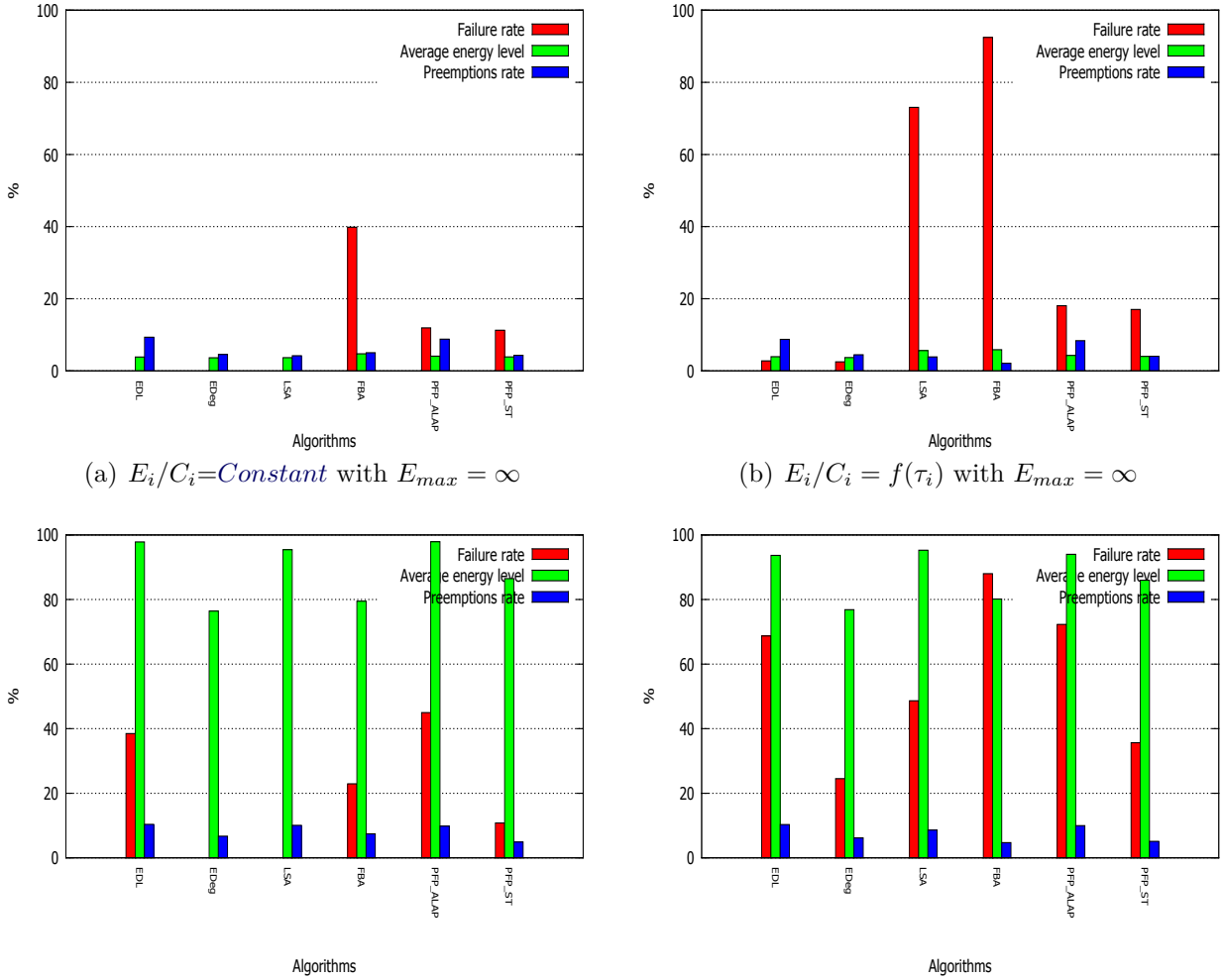
Figure 3.9: Frame-base task model

of future deadline misses with the slack-energy notion. The global performances of the remaining algorithms are summarized in the following points.

- *EDL*: has a high failure rate when the battery capacity is limited but seems to behave better when the stage unit is infinite. This was expected because *EDL* does not take into account energy when computing slack-time. The simulations show that *EDL* has relatively a high preemption rate and average energy level comparing to other algorithms which is due to long idle periods and the interferences within execution windows,

- *LSA*: shows good performances only when all the tasks share the same energy consumption rate,

- *FBA*: shows good performances only for frame-based task sets,



(a) E_i/C_i=*Constant* with $E_{max} = \infty$

(b) $E_i/C_i = f(\tau_i)$ with $E_{max} = \infty$

(c) E_i/C_i=*Constant* with $E_{max} = 200$

(d) $E_i/C_i = f(\tau_i)$ with $E_{max} = 200$

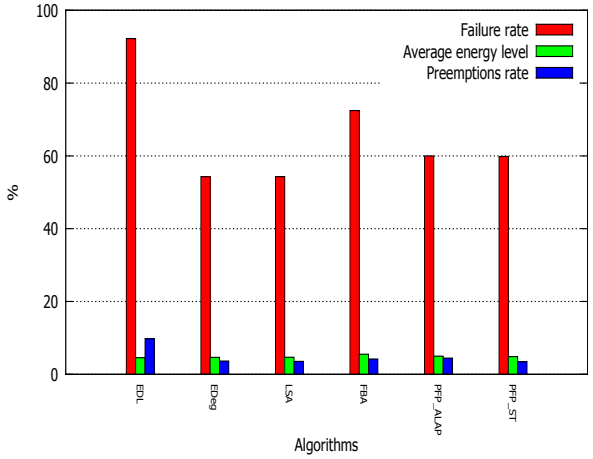
Figure 3.10: Periodic task model with implicit deadlines

- PFP_{ALAP}: behaves like EDL but with a higher level of failure and preemptions because it is a fixed-priority algorithm,

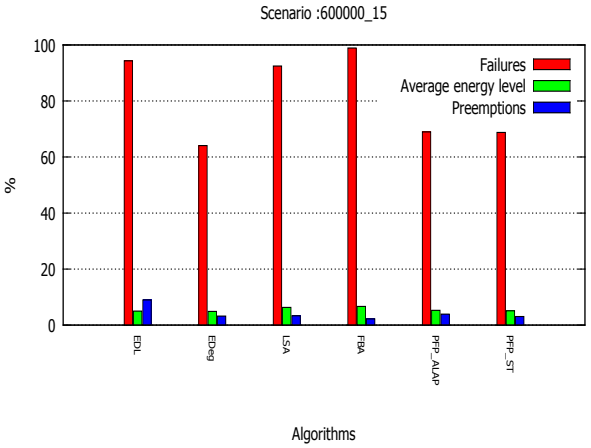
- PFP_{ST}: behaves a little better than PFP_{ALAP} but has the same weaknesses, namely a high failure and preemption rates which are due the slack-time computation that does not consider energy constraints.

3.8.3 Comparison summary

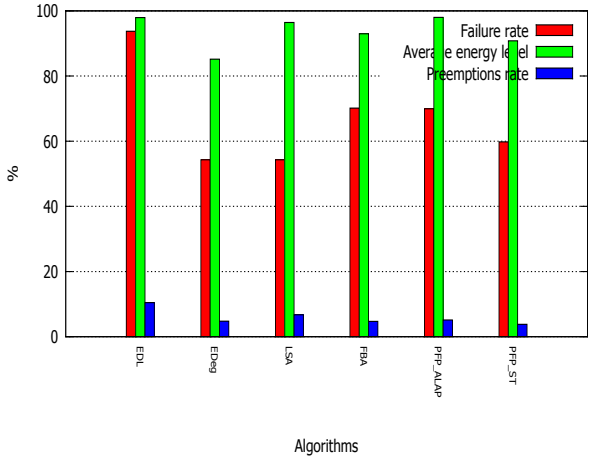
Table 3.1 on page 98 summarizes the properties of the discussed algorithms.



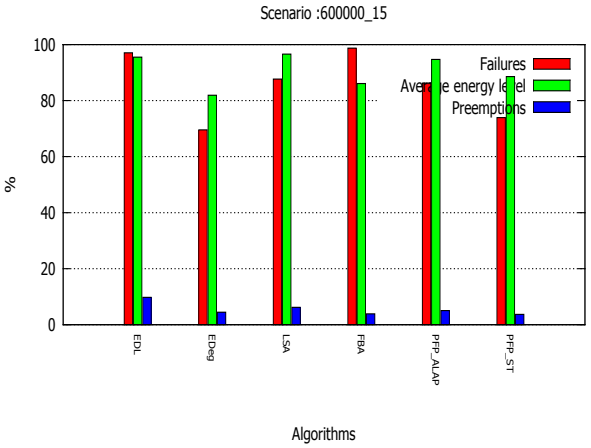
(a) E_i/C_i=*Constant* with $E_{max} = \infty$



(b) $E_i/C_i = f(\tau_i)$ with $E_{max} = \infty$



(c) E_i/C_i=*Constant* with $E_{max} = 200$



(d) $E_i/C_i = f(\tau_i)$ with $E_{max} = 200$

Figure 3.11: Periodic task model with constrained deadlines

3.9 Conclusion

In this chapter we introduced the problematic of real-time scheduling for energy-harvesting systems. We first described the theoretical model we are interested in and we expressed the problematics linked to this model, namely, finding algorithms, feasibility conditions and the minimum energy storage unit capacity. Second, we discussed the two main scheduling approaches for energy-harvesting systems, namely, real-time scheduling with DVFS and energy-aware scheduling. Third, we detailed this later and we enumerated the main scheduling algorithms available in the literature, namely *EDL*, *EDeg*, *LSA*, *FBA*, PFP_{ALAP} and PFP_{ST}. Finally we evaluated and compared these algorithms with simulations that showed that *EDeg* is the best algorithm that keeps its optimality in different task models, i.e. frame-based and periodic, and different energy configurations, i.e. constant and variable tasks energy consumption rates and battery

capacity size.

We note that most of the proposed solutions are EDF-based because of its optimality for systems without energy constraints. Furthermore, the *EDF* scheduling for energy-harvesting systems was the most studied in the past. However, real-time scheduling for energy harvesting systems with fixed-task-priority, which are as important as *EDF* scheduling, was not deeply studied. The algorithms available in the literature are not very efficient and can be widely improved. The aim of this dissertation is to contribute to cover this research area by studying the possibility of adapting some *EDF* solutions to fixed-priority and by proposing new scheduling algorithms and feasibility conditions that can solve the problematics identified in this chapter.

Part II

Contributions

Chapter 4

The PFP_{ASAP} Algorithm

Contents

4.1 Introduction

As noticed in the state of the art, the fixed-priority scheduling for energy-harvesting was not very well studied compared to dynamic-priority or EDF-based scheduling. In fact only few fixed-priority algorithms are available in the literature. In reality, even though the earliest deadline first scheduling has proved its optimality for classical real-time systems, it is still not very popular in industry. However, fixed-priority scheduling is widely used in industry because of its simplicity. Therefore, fixed-priority is as important as EDF-scheduling and it is worthy to study this type of scheduling for energy-harvesting systems.

As mentioned in Section 3.3 on page 71, one of the studied problematics is to find fixed-priority scheduling algorithms that respect tasks deadlines and energy-harvesting systems constraints.

The main fixed-priority algorithms available in the literature, namely PFP_{ALAP} and PFP_{ST} which are described in Section 3.7 on page 92, are not optimal and have a great complexity.

This chapter focuses on optimal fixed-priority solution of the considered problem. We propose the **P**reemptive **F**ixed-Task-**P**riority **A**s **S**oon **A**s **P**ossible (PFP_{ASAP}) algorithm and we prove its optimality for energy-non-concrete task sets that are composed only of tasks that consume more energy than the replenished during their execution. The work presented in this chapter is the result of a collaboration with Dr. Y. Abdeddaïm and was first published in [ACM13a].

The remaining part of the chapter is organized as follows. We recall the model in Section 4.2. In Section 4.3 we introduce the PFP_{ASAP} algorithm, and then, we study some of its properties, namely its worst-case scenario, its optimality and its optimal priority assignment. A feasibility condition based on PFP_{ASAP} is also proposed. In Section 4.4, we evaluate the performances of PFP_{ASAP} by simulations and we compare them with the ones the algorithms presented in the state of the art. Finally, we conclude in Section 4.5.

4.2 Model

The model considered in this chapter is similar to the one described in Section 3.2 on page 67. The difference here is that we consider only energy-non-concrete task sets which means that tasks offsets and the initial energy level of the battery are known only at run time.

Task Model

We consider a non-concrete real-time sporadic task set in a renewable energy environment defined by a set of n periodic and independent tasks. Each task τ_i is characterized by its priority P_i, its worst-case execution time C_i, its minimum inter-arrival time T_i, its relative deadline D_i and its worst-case energy consumption E_i. Since the considered task set is non-concrete, the offsets denoted as O_i are known only at run-time.

Energy Model

The energy model considered in this chapter is exactly the same as the one described in Section 3.2 on page 67. An energy storage with capacity $\mathcal{C}$ and a harvester that replenishes the battery continuously with a constant rate P_r. The energy level of the storage unit at time t denoted $E(t)$ fluctuates between two thresholds E_{max} the maximum level and E_{min} the minimum level such that $\mathcal{C} = E_{max} - E_{min}$. Knowing that the considered systems are energy-non-concrete, we assume that the initial level of the storage unit is E_{min}. Furthermore, we suppose that $P_r \leq \mathcal{C}$ to avoid energy losses. Note that this is not sufficient to avoid energy losses but it is necessary. Moreover, in this chapter we suppose that tasks consume more energy than harvested during a period

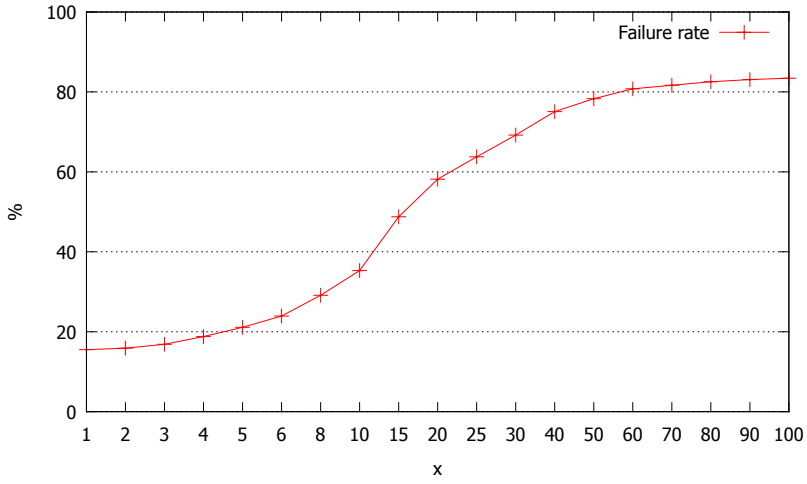


Figure 4.1: The effect of the parameter x on schedulability

of executions, i.e. $\forall i$, $E_i \geq C_i \times P_r$. We first study the scheduling problem with only consuming tasks, then, we consider the gaining ones in the next chapter.

4.3 Theoretical Study of PFP_{ASAP}

4.3.1 As Soon As Possible Preemptive Fixed-Priority

In [CMM11], a set of scheduling heuristics for energy-harvesting systems have been proposed. These heuristics are fixed-priority and take into account tasks energy cost and the battery capacity limit during scheduling operations. One of these heuristics caught our attention. With this heuristic, tasks are executed according to their priority such that whenever there is not enough energy in the battery to execute, jobs executions are suspended to replenish energy for a fixed amount of time x. The authors performed some experiments to evaluate the relevance of this policy by studying the impact of varying the parameter x on the schedulability rate of the policy. They varied x from $x = 4$ to $x = 100$ and the best value for their task sets sample was between 4 and 6. However, they did not evaluate the algorithm for the smallest possible value of x, i.e. $x = 1$. Figure 4.1 shows the results of the same experiment performed in [CMM11] but by varying x starting from 1. We can see that the lowest failure rate is reached when $x = 1$. Adding to this the good performances of this policy comparing to the other ones, this policy becomes very interesting to study deeply, especially its optimality.

In this section we study the case where $x = 1$ that we call *As Soon As Possible Preemptive Fixed-Priority Algorithm* (PFP_{ASAP}).

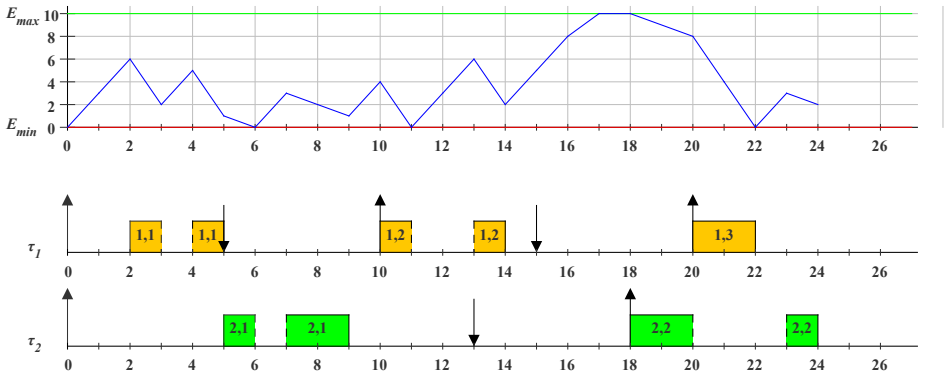
Algorithm 4.1 on the following page shows how PFP_{ASAP} takes decisions at time t. It schedules jobs as soon as possible when there is enough energy to execute at least one time unit, otherwise, it delays jobs executions by adding idle periods in order to

Algorithm 4.1 PFP_{ASAP} Algorithm

```
1: t ← 0
2: loop
3:    A ← set of active tasks at time t
4:    if A ≠ ∅ then
5:       τ_k ← the highest priority task of A
6:       if E(t) + P_r − E_min ≥ E_k/C_k then
7:          execute τ_k for one time unit
8:       end if
9:    end if
10:   t ← t + 1
11: end loop
```

-	C_i	E_i	T_i	D_i	P_i
τ_1	2	14	10	5	1
τ_2	3	12	18	13	2

(a) task set Γ with $P_r = 3$



(b) PFP_{ASAP} time chart for task set Γ with $E_{max} = 10$

Figure 4.2: PFP_{ASAP} schedule

replenish the necessary energy. The replenishment periods are as long as needed to satisfy the energy demand of the execution of at least one time unit.

Figure 4.2(b) illustrates an example of PFP_{ASAP} schedule of the task set described in Table 4.2(a). In this example $E_{max} = 10$, $E_{min} = 0$ and $P_r = 3$. At time $t = 0$ the battery is empty, therefore, task τ_1 cannot be executed. Then, the battery is replenished until time $t = 2$, in other words until there is enough energy to execute one time unit of τ_1, i.e. until $E(t) \geq E_1/C_1$. After that, the algorithm follows the same scheduling schemes for the rest of the schedule.

Below, we first characterize the worst-case scenario of PFP_{ASAP}, then we discuss

its optimality and finally, we build a necessary and sufficient feasibility condition for the considered family of task sets.

4.3.2 Worst-Case Scenario

The aim of this section is to characterize the worst-case scenario that a task set can encounter during its execution with PFP_{ASAP} algorithm. First, we recall the notion of processor demand or workload function for fixed-priority scheduling, then we extend it to include tasks energy consumption.

Definition 4.1 (Processor Demand)**.**
The processor demand of the i^{th} priority level at time t denoted as $Wp_i(t)$, is the amount of time necessary to execute jobs of priority levels $1, \ldots, i-1, i$ requested in the interval of time $[0, t[$. It can be obtained by formula 4.1.

$$Wp_i(t) = \sum_{j \leq i} RBF_j(t) = \sum_{j \leq i} \left\lceil \frac{t - O_j}{T_j} \right\rceil \times C_j \tag{4.1}$$

■

Now we introduce the notion of replenishment demand.

Definition 4.2 (Replenishment Demand)**.**
The replenishment demand of the i^{th} priority level at instant t denoted $We_i(t)$, is the amount of energy to be replenished to execute jobs of priority levels $1, \ldots, i-1, i$ requested in the interval of time $[0, t[$. It can be calculated by Equation 4.2.

$$We_i(t) = \sum_{j \leq i} \left\lceil \frac{t - O_j}{T_j} \right\rceil \times E_j - E(0) \tag{4.2}$$

■

The intuition behind Formula 4.2 is derived from the notion of processor demand. It is the difference between the energy demand of jobs requested inside interval $[0, t[$ and the initial battery level. The initial battery level $E(0)$ is removed to fit with the exact amount of energy to be replenished. Therefore, if $We_i(t)$ is negative, then, the energy stored in the battery is sufficient and more replenishment is not needed.

Definition 4.3 (Time Demand)**.**
The time demand of the i^{th} priority level at time t denoted $W_i(t)$, is the minimum amount of time necessary to satisfy both of the replenishment and processor demands within time interval $[0, t[$. This can be obtained by Equation 4.3.

$$W_i(t) = max\left(\left\lceil \frac{We_i(t)}{P_r} \right\rceil, Wp_i(t)\right) \tag{4.3}$$

■

The fraction $\lceil We_i(t)/P_r \rceil$ gives the number of replenishment time units to charge the replenishment demand $We_i(t)$.

Definition 4.4 (PFP_{ASAP} Response Time)**.**
The response time of the first job of τ_i according to PFP_{ASAP} denoted R_i is the termination time of the execution of the i^{th} priority level minus the first release time of the same priority level, i.e. O_i. The termination time of the first job of τ_i denoted $f_{i,1}$ is the smallest solution of the system of equations 4.4.

$$\begin{cases} W_i(t) & = & t \\ t & \geq & O_i \end{cases} \tag{4.4}$$

■

Now, we can use these definitions to characterize the worst-case scenario which is expected to be the synchronous activation of all the tasks when the battery is at its minimum level. This intuition is justified by the comparison of all possible activation scenarios as shown in Figure 4.3 on the facing page.

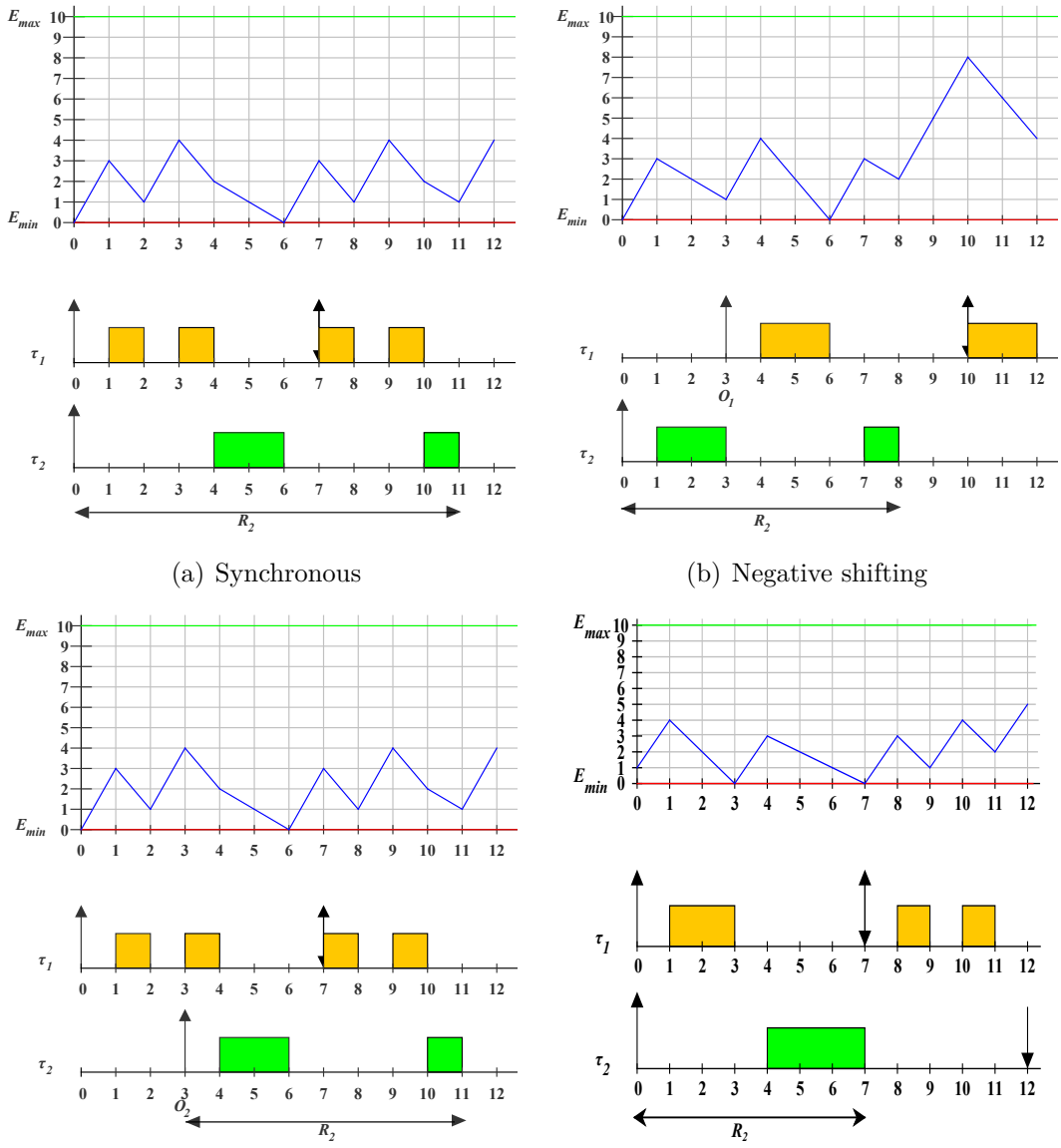
Figure 4.3(a) illustrates the case where all the tasks are requested simultaneously. If at least one higher priority task is requested later, the response time of lower priority tasks decreases as shown in Figure 4.3(b). Then, if higher priority tasks are requested earlier, the response time of lower priority tasks cannot be longer than the one reached in the synchronous scenario as shown in Figures 4.3(c). Moreover, when the initial battery level is higher than E_{min}, the response time of all tasks cannot be longer because the amount of energy available in the battery can directly be used to execute instead of adding more replenishment units that lengthen tasks response times. This case is illustrated by Figure 4.3(d).

Thus, we propose Theorem 4.1.

Theorem 4.1.
Let Γ denote an energy-non-concrete task set composed of n priority-ordered tasks with constrained or implicit deadlines. The PFP_{ASAP} worst-case scenario for any task of Γ occurs whenever this task is requested simultaneously with requests of all higher priority tasks and the battery is at the minimum level E_{min}.

Proof. We compare jobs response times in the scenario described by Theorem 4.1 with all other possible ones. As mentioned in Definition 4.4, the response time of the first job of a task is equal to its termination time minus its offset (first release time). The main key of the proof is to argue with different termination times and offsets by comparing their possible values in different cases of activation scenario and initial battery level.

Let $\{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of n priority-ordered tasks where τ_n is the task with the lowest priority. Let S_i^s denote the scenario where task τ_i and all higher priority tasks are requested simultaneously at the lower battery level E_{min}. The worst-case scenario for a task τ_i is the one which maximizes its response time, i.e. the scenario which delays the termination time of the first job of the i^{th} priority level.



(a) Synchronous

(b) Negative shifting

(c) Positive shifting

(d) $E(0) > E_{min}$

Figure 4.3: PFP_{ASAP} worst-case scenario

If S_i^s is not the worst-case scenario, then, there must be an other scenario that leads to a greater response time for the i^{th} priority level.

Firstly, we consider the scenario where $E(0) > E_{min}$. In this case there is some amount of energy available at time $t = 0$. We denote this scenario as S_i', its replenishment demand as $We_i^{S_i'}$ and its response time $R_i^{S_i'}$. Then, if $E(0) > 0$, we have:

$$
\begin{aligned}
E(0) > 0 \;&\Rightarrow\; \textstyle\sum_{j\le i}\left\lceil \frac{t-O_j}{T_j}\right\rceil \times E_j - E(0) < \sum_{j\le i}\left\lceil \frac{t-O_j}{T_j}\right\rceil \times E_j \\
&\Rightarrow\; We_i^{S_i'}(t) < We_i^{S_i^s}(t) \\
&\Rightarrow\; \frac{We_i^{S_i'}(t)}{P_r} \le \frac{We_i^{S_i^s}(t)}{P_r} \\
&\Rightarrow\; \left\lceil \frac{We_i^{S_i'}(t)}{P_r}\right\rceil \le \left\lceil \frac{We_i^{S_i^s}(t)}{P_r}\right\rceil \\
&\Rightarrow\; \max\left(\left\lceil \frac{We_i^{S_i'}(t)}{P_r}\right\rceil,\, Wp_i(t)\right) \le \max\left(\left\lceil \frac{We_i^{S_i^s}(t)}{P_r}\right\rceil,\, Wp_i^{S_i^s}(t)\right) \\
&\Rightarrow\; W_i^{S_i'}(t) \le W_i^{S_i^s}(t) \\
&\Rightarrow\; R_i^{S_i'} \le R_i^{S_i^s}
\end{aligned}
$$

Therefore, the system needs less replenishment demand than the scenario where $E(0) = 0$, i.e. $E(0) = E_{min}$, and PFP_{ASAP} introduces shorter or equal replenishment periods and leads to shorter or equal response time for all the tasks. This is in contradictory with our hypothesis, thus, such a scenario cannot lead to longer response times.

Secondly, we consider the scenario with different offsets. Let us denote S_i^a as the scenario where $E(0) = E_{min} = 0$ and tasks can have different offsets. Let t_s denote the termination time of the first job of task τ_i in the synchronous scenario S_i^s and let t_a denote the termination time of the same job in the asynchronous scenario S_i^a. Scenario S_i^a is worse than scenario S_i^s implies that $t_a > t_s$.

We know that:

$$
t_s = W_i^s(t_s) = \max\left(\left\lceil \frac{We_i(t_s)}{P_r}\right\rceil, Wp_i(t_s)\right) \tag{4.5}
$$

and $\lceil We_i(t_s)/P_r\rceil \ge Wp_i(t_s)$ because in our model $E(0) = 0$ and $\forall\ i,\ E_i \ge C_i \times P_r$. This reveals the fact that in the considered model, we must have replenishment periods

which increase job response time. Then,

$$t_s = W_i^s(t_s) = \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_s}{T_j} \right\rceil \times E_j}{P_r} \right\rceil \tag{4.6}$$

Similarly,

$$t_a = W_i^a(t_a) = \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_a - O_j}{T_j} \right\rceil \times E_j}{P_r} \right\rceil \tag{4.7}$$

Knowing that $W_i^a(t)$ is strictly increasing in the interval $[0, t_a]$ and $t_a = W_i^a(t_a)$, we obtain:

$$t_s < t_a \Rightarrow t_s < W_i^a(t_s) \tag{4.8}$$

By replacing t_s with $W_i^s(t_s)$ we obtain

$$\left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_s}{T_j} \right\rceil \times E_j}{P_r} \right\rceil < \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_s - O_j}{T_j} \right\rceil \times E_j}{P_r} \right\rceil \tag{4.9}$$

Finally, we have

$$\sum_{j \leq i} \left\lceil \frac{t_s}{T_j} \right\rceil \times E_j < \sum_{j \leq i} \left\lceil \frac{t_s - O_j}{T_j} \right\rceil \times E_j \tag{4.10}$$

We know that $t_s \geq t_s - O_j$ because $O_j \geq 0$. Therefore

$$\sum_{j \leq i} \left\lceil \frac{t_s}{P_r} \right\rceil \times E_j \geq \sum_{j \leq i} \left\lceil \frac{t_s - O_j}{P_r} \right\rceil \times E_j \tag{4.11}$$

Inequality 4.10 is in contradiction with inequality 4.11. Thus, we prove that $t_s \geq t_a$. Knowing that $R_i = f_{i,1} - O_i$, we also have $R_i^s \geq R_i^a$ because $t_s - 0 \geq t_a - O_i$.

Therefore, we prove that the synchronous activation of all tasks when the battery reaches the minimum level is the worst-case scenario that a task set can undergo with PFP_{ASAP} algorithm.

□

4.3.3 Optimality

In this section we discuss the optimality of PFP_{ASAP} for the model described in Section 4.2 on page 110.

Theorem 4.2.
The PFP_{ASAP} algorithm is optimal for the scheduling problem of fixed-priority energy-non-concrete task sets with constrained or implicit deadlines. As defined in Section 1.4.4 on page 37, this optimality means that if PFP_{ASAP} fails to schedule a given task set, then, no other fixed-priority algorithm can.

Proof. Let Γ denote an energy-non-concrete task set. We suppose that Γ is feasible using a fixed-priority assignment, but not schedulable with PFP_{ASAP} using the same priority assignment. This means that there exists at least one task denoted τ_k that misses its first deadline in the worst-case scenario given in theorem 4.1 on page 114. Indeed, it is sufficient to consider only the first job because all tasks have constrained or implicit deadlines, which avoid jobs to overlap (unless not feasible systems), and are requested in the worst-case scenario, which maximizes the interferences and the length of eventual replenishment periods. According to PFP_{ASAP} rules, a deadline miss occurs in the worst-case scenario for the k^{th} priority level only if the energy needed to execute priority levels higher or equal than k is greater than the energy that can be replenished from $t = 0$ to the first deadline of τ_k, Inequality 4.12 summarizes that.

$$D_k \times P_r < \sum_{j \leq k} \left\lceil \frac{D_k}{T_j} \right\rceil \times E_j \tag{4.12}$$

If PFP_{ASAP} is not optimal, then, there must exist an other fixed-priority schedule for Γ that makes it feasible. Let us suppose that such a schedule exists. This implies that there exists at least one task that is executed even if the energy is not sufficient. This is impossible because the system cannot execute without energy, therefore, such a schedule cannot exist. Then we prove that PFP_{ASAP} is optimal for energy-non-concrete fixed-priority task sets with constrained or implicit deadlines. □

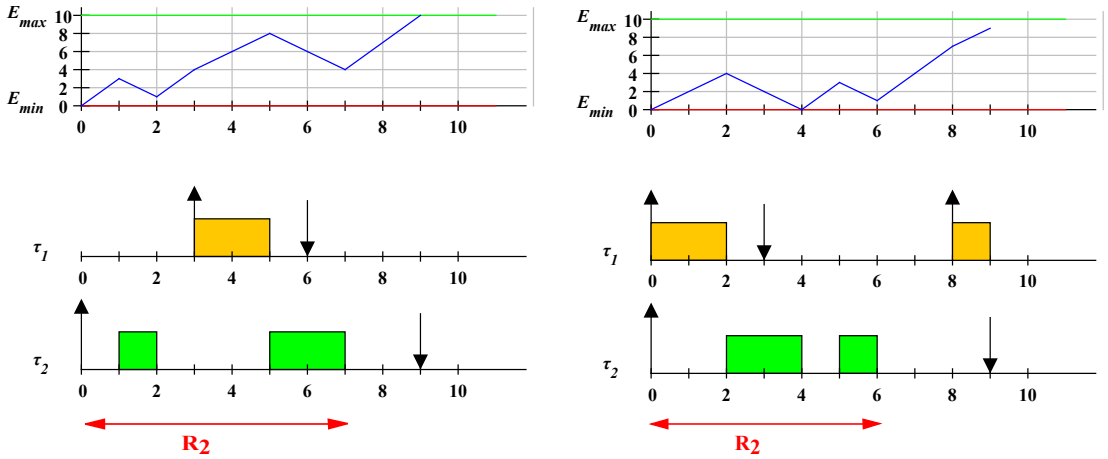
Discussion

Note that the optimality of PFP_{ASAP} relies on the hypotheses set in Section 4.2 on page 110, mainly the assumptions about task consumption and replenishment functions. If we relax some of them PFP_{ASAP} may lose its optimality.

Indeed, if we accept tasks that consume energy less than replenished, i.e. gaining tasks, the worst-case scenario is no longer the same which can lead PFP_{ASAP} to lose its optimality. Figure 4.4(c) on the next page illustrates such a case. We can see that task τ_2, which is a consuming task is delayed three times, the first one at time 0, the second at time 2 to replenish the battery, and the third time at time 3 to execute τ_1 which is a gaining task. This delay leads to a final response time of 7 time units. However, In

Tasks	C_i	E_i	T_i	D_i	P_i
τ_1	2	2	8	3	1
τ_2	3	15	10	9	2

(a) Task set Γ



(b) Asynchronous requests

(c) Synchronous requests

Figure 4.4: PFP_{ASAP} worst-case scenario counter example with gaining tasks

Figure 4.4, we can see that the synchronous request of τ_1 and τ_2 leads to a response time of only 6 units. This proves that when we mix gaining and consuming tasks, the synchronous activation is no longer the worst-case scenario and PFP_{ASAP} is no longer optimal.

Moreover, when we consider the energy cost of mode switching from active to idle or the opposite way, PFP_{ASAP} loses also its optimality and behaves badly as shown in [BA14].

In a more realistic model, the replenishment function is not constant. Therefore Equation 4.12 on the facing page is no longer applicable. Thus, we cannot conclude about PFP_{ASAP} optimality. Finally, we have counter examples that prove the non-optimality of PFP_{ASAP} for concrete task sets. We discuss this point in the next chapter.

4.3.4 Priority Assignment

As mentioned in Section 1.4.6 on page 38, the priorities assigned to tasks are very important in fixed-priority driven scheduling because they are by definition fixed offline which may reduce schedulability if the priorities are not well assigned.

In the classical theory of fixed-priority real-time scheduling, deadline monotonic was proved to be optimal for non-concrete task sets with constrained or implicit deadlines. In this section, we extend this property for energy-non-concrete task sets whose tasks consume more energy than the one replenish during execution time.

Theorem 4.3.
Deadline monotonic is an optimal priority assignment for energy-non-concrete task sets when all the tasks have constrained or implicit deadlines and consume energy more than it is replenished during their execution, i.e. when $\forall\, i,\ D_i \leq T_i$ *and* $E_i > C_i \times P_r$.

The idea behind Theorem 4.3 is that DM can schedule any task set that is schedulable with an other priority ordering. This means that every valid priority assignment can be transformed into deadline monotonic priority ordering.

Proof. When the task set is composed of only consuming tasks, i.e. $\forall\, i,\ E_i > C_i \times P_r$, the time demand of a priority level-i in the worst-case scenario is the time necessary to replenish the energy demand or the replenishment demand because all tasks need more time than their execution time to replenish the needed energy. Then, the maximum in the right hand side of Equation 4.3 on page 113 is $\lceil We_i(t)/P_r \rceil$. Therefore, by considering a synchronous release of all the tasks, i.e. $\forall\, i,\ O_i = 0$, and a minimum initial battery level $E(0) = E_{min} = 0$, the new time demand is obtained by Equation 4.13.

$$W_i(t) = max\left(\left\lceil \frac{We_i(t)}{P_r} \right\rceil, Wp_i(t)\right) = \left\lceil \frac{We_i(t)}{P_r} \right\rceil = \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t}{T_j} \right\rceil E_j}{P_r} \right\rceil \tag{4.13}$$

Now, to prove Theorem 4.3 we assume two tasks at adjacent priorities τ_A and τ_B with $D_A \leq D_B \leq T_B$.

Given that τ_A is schedulable at the lower priority level than τ_B, we swap priorities of τ_A and τ_B and we observe the impact of this change on the schedulability of τ_A and τ_B. Then, if τ_B remains schedulable, this means that deadline priority assignment is optimal as we can keep swapping task priorities from any schedulable priority ordering to get a DM ordering.

1. Assume that τ_A is schedulable at lower priority than τ_B. Then, the response time of the first job of τ_A in the worst-case scenario, denoted R_A, is given by Equation 4.14 where X is the response time of the task with the lower priority task, in this case the one of τ_A.

$$X = R_A = \left\lceil \frac{\left\lceil \frac{R_A}{T_A} \right\rceil E_A + \left\lceil \frac{R_A}{T_B} \right\rceil E_B + \sum_{j<A \,\wedge\, j \neq B} \left\lceil \frac{R_A}{T_j} \right\rceil E_j}{P_r} \right\rceil \tag{4.14}$$

   Knowing that τ_A is schedulable, then, we have $R_A \leq D_A$. Furthermore, in this case, for such value of X, we have respectively $\lceil R_A/T_A \rceil E_A = E_A$ and $\lceil R_A/T_B \rceil E_B = E_B$ because respectively $R_A \leq D_A \leq T_A$, from the fact that we are computing the response time of the first job of τ_A, and $R_A \leq D_A \leq D_B \leq T_B$, from the fact that deadlines are constrained or implicit.

2. Now, we swap priorities of τ_A and τ_B. We consider that the response time of τ_B is given by Equation 4.15.

$$X = R_B = \left\lceil \frac{\left\lceil \frac{R_A}{T_A} \right\rceil E_A + \left\lceil \frac{R_A}{T_B} \right\rceil E_B + \sum_{j<B \wedge j \neq A} \left\lceil \frac{R_A}{T_j} \right\rceil E_j}{P_r} \right\rceil \tag{4.15}$$

   Now, we compare Equations 4.14 on the facing page and 4.15. Thus, we observe that they are the same equation and so the value of X must also be the same when τ_B is at a lower priority than τ_A. Therefore, τ_B is schedulable because $D_B \geq D_A$.

3. It is obvious that τ_A is also schedulable because it is at a higher priority level now.

Then, we show that DM schedules any system that is schedulable with any other priority assignment by swapping adjacent task priorities until we obtain a deadline monotonic priority assignment without loss of schedulability. □

Note that this proof relies on the assumptions set on tasks offsets and energy consumption. If we relax one of them, the proof is no longer correct because the worst-case scenario PFP_{ASAP} is no longer the same and the used formulas are no longer valid.

4.3.5 Schedulability Condition

A simple way to build a necessary and sufficient feasibility condition for energy-non-concrete task sets is to check if the given task set is schedulable with PFP_{ASAP} in the worst-case scenario, in other words, to check if the first job of each task meets its deadline when it is requested simultaneously with the higher priority tasks while the battery is at its minimum level. It consists of computing the worst-case response time according to PFP_{ASAP} rules for each task and to compare it to its deadline. Algorithm 4.2 on the following page explains how to do this. It is an extension of the classical response time algorithm that considers energy replenishment periods.

The complexity of Algorithm 4.2 is $O(m \times n)$ where m is the number of iterations and n is the number of tasks. We note that the number of iterations m depends on the periods and deadlines of the tasks (see line 6 of Algorithm 4.2) and is bounded by $\max_{\forall i}(D_i)/\min_{\forall i}(T_i)$. Thus, the complexity of Algorithm 4.2 is pseudo-polynomial. We can reduce this complexity by computing estimations for response times rather than the exact values. However, the schedulability test we propose will not be necessary but will remain sufficient.

Algorithm 4.2 Feasibility Test

1: **for** $i = 1 \rightarrow n$ **do**
2: $W_i^{m+1} \leftarrow \epsilon$
3: **repeat**
4: $W_i^m \leftarrow W_i^{m+1}$
5: $W_i^{m+1} \leftarrow \max\left(\left\lceil \frac{We_i(W_i^m)}{P_r} \right\rceil, Wp_i(W_i^m)\right)$
6: **if** $W_i^{m+1} > D_i$ **then**
7: **return** $False$
8: **end if**
9: **until** $W_i^{m+1} = W_i^m$
10: **end for**
11: **return** $True$

4.3.6 Battery Capacity

The algorithm PFP_{ASAP} replenishes the minimum amount of energy needed for only one execution unit, in this case the minimum battery capacity needed to keep the task set schedulable with PFP_{ASAP} is the maximum amount of energy that can be consumed during one time unit, i.e. the maximum instantaneous consumption. In our model all the tasks consume energy linearly. Furthermore, the minimum battery capacity must also take into account the eventual energy losses due to the capacity of the battery. Therefore, the minimum battery capacity $\mathcal{C}_{min}$ that keeps the task set schedulable is bounded by $\max(\max_{\forall i}(E_i/C_i),\ P_r)$.

When launching the system with the minimum battery level, this capacity allows PFP_{ASAP} to keep the same schedule as the case where $\mathcal{C} \geq \mathcal{C}_{min}$ and thereby it keeps the same response times and the schedulability of the system. Running the task set with a storage unit with a capacity lower than $\mathcal{C}_{min}$ may either lead to energy losses in the case where $\mathcal{C} < P_r$ which may increase tasks response time and may compromise the schedulability of the task set or may make the executions impossible if $\mathcal{C}$ is lesser than tasks consumption rate.

4.4 Performance Evaluation

We proved that PFP_{ASAP} is optimal for energy-non-concrete task sets. In this section we study empirically the behavior of PFP_{ASAP} and we compare it to other algorithms by simulations.

4.4.1 Competitors

We compare PFP_{ASAP} to other fixed-priority algorithms that are proposed in the literature for energy-harvesting systems. We selected the following competitors.

- PFP_{ALAP}: the fixed-priority scheduling policy described in Section 3.7.1 on page 92. PFP_{ALAP} is not optimal but can be used as a reference for comparison since there is no many algorithms in the state of the art.

- PFP_{ST}: the fixed-priority scheduling policy described in Section 3.7.2 on page 95. We selected PFP_{ST} which is not optimal but has the lowest failure rate according to the experiment presented in Section 3.8 on page 99.

4.4.2 Simulation

In this section, we describe the configuration of the experiments, namely the task sets generation, the parameters and the set assumptions.

To perform such an experiment we used a simulator named YARTISS that we present in Chapter 8.

Task Sets Generation

For these simulations we used an adapted version of the *UUniFast-Discard* algorithm [BB05] coupled with a technique for hyper-period limitation [MG01] to generate task sets. The generated task sets respect the following hypotheses:

- All task sets are time feasible without considering energy parameters and constraints.

- Time and energy are discretized, this means that they are integers and all scheduling operations are performed before or after one time unit,

- The charging rate P_r is constant, i.e. a constant amount of energy is added to the battery level in every time unit,

- Tasks consume energy linearly, i.e. a task consumes E_i/C_i energy units for each execution time unit,

- All task sets are composed of consuming tasks, in other words, all tasks consume energy more than what we can replenish during execution times, i.e. $\forall\, i,\ E_i > C_i \times P_r$.

In order to represent most of the possible task sets, we generate them according to their processor and energy utilizations, i.e. U and U_e. We vary U and U_e in the interval $[0.2, 1]$ to obtain a couple of (U, U_e) for each 0.05 unit of U and U_e. Then, we obtain 50 distinct task sets for each couple (U, U_e).

In this experiment, all task sets are simulated in the worst-case scenario described in Section 4.3.2 on page 113.

Simulation Configuration

In order to evaluate the behavior of the compared algorithms, we vary some parameters, namely the battery capacity $\mathcal{C}$ and the number of tasks per task set. Firstly, we vary $\mathcal{C}$ in six energy scenarios to analyze its effect on the failure rate. Secondly, we vary the number of tasks per task set in several distinct simulations to observe the the scheduling overhead of each algorithm.

We set the remaining parameters to the same values that we used for task sets generation in order to fit with the considered assumptions. For these experiments, we set these parameters as follows, $P_r = 15$ and $E_{min} = 0$. Furthermore, task sets are run for 2600 time units which corresponds to the longest hyper-period. Thus, if a task set does not miss any deadline during the simulation time, then, the task set is said to be empirically feasible. We use **D**eadline **M**onotonic (DM) policy to assign priorities because of its optimality for the considered model as shown in Section 4.3.4 on page 119.

Several statistical metrics are computed during simulations. These metrics give information about algorithms behavior. For our experiments we selected the following metrics: *failure rate*, *preemption count*, *average overhead*, *average idle-period*, *average busy-period* and *average energy level*.

Metrics

Failure Rate: the percentage of unfeasible task sets among all the tested ones. The greater the failure rate is, the lower the algorithm performance is.

Preemption Count: for one simulation, it represents the number of preemption events. A preemption event occurs when a job is suspended while it is still not finished. All the events that occur at the same instant (e.g. job request, a deadline, etc.) are considered as once. Therefore, the number of possible events is bounded by the number of time units composing the simulation, i.e. the simulation duration. For several simulations, this metric is computed only for feasible task sets and represents the ratio of the average number of preemptions relative to the number of all possible events. The greater the number of preemptions is, the greater the context switch is. A large number of preemptions increases the overhead cost and decreases performance, which makes the algorithm unusable in practice.

Average Overhead: it is the amount of time spent while handling a scheduling event, in other words, the spent by the scheduling algorithm to take a decision. For one simulation, this metric represents the average overhead of all of the scheduling events. Its exact value is difficult to compute and must be computed on a real platform. We simply calculate an estimation by distributing the real simulation time (in milliseconds) on the number of scheduling events. The simulation tool that we use is event-based. Therefore, only the events processing consumes processor time. Thus, it gives us an

acceptable estimation of the average overhead. The greater the average overhead is, the greater the timing constraints violation risk is.

Average Idle-Period and Average Busy-Period: represents respectively the average duration of periods when the processor is idle and the average duration of continuous processor activity. For several simulations, we compute the ratio of the average idle-period/average busy-period duration relative to the simulation duration.

The relevance of these two metrics is closely linked to the number of preemptions caused by replenishment periods. The longer the idle/busy periods are, the lesser the number of replenishment preemptions is and the higher the algorithm performance are. Therefore, the longer the idle-periods are, the higher the average energy level is and the lesser energy-constrained the system is. Moreover, grouping idle-periods together is better for batteries that do not support a high rate of charge/discharge cycles.

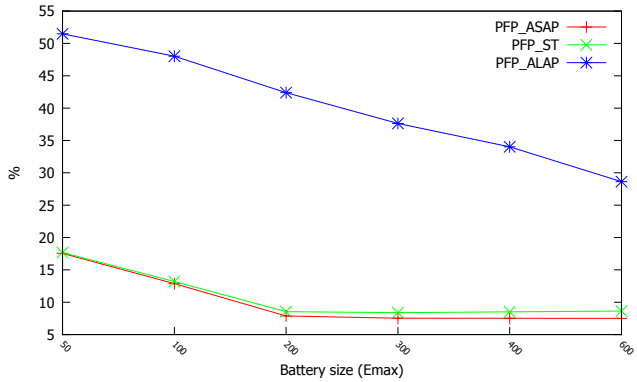
Average Energy Level: represents the average energy level of the battery at any instant during the simulation. It is the average of the energy level of all scheduling events. The best algorithms relative to this criterion are ones that maximize the average energy level. This means that the algorithm makes the system less energy constrained.

4.4.3 Results Analysis

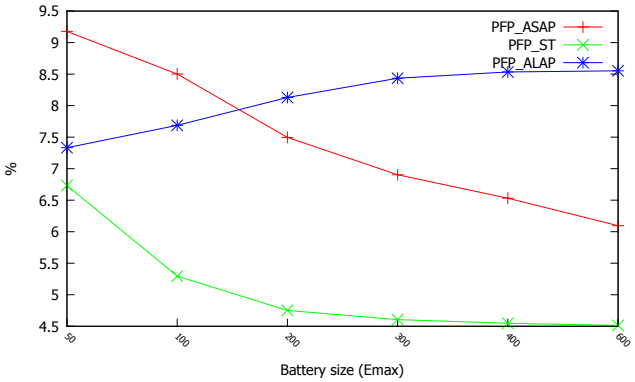
The Variation of E_{max}

Figure 4.5 on the following page presents the results of the comparison of the selected algorithms. In the following part, we analyze the effect of E_{max} variation on the performances of each algorithm for each metric:

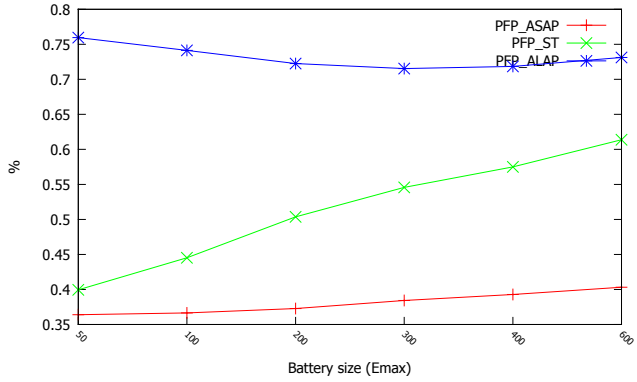
Failure Rate: Increasing E_{max} reduces the failure rate of all the evaluated algorithms. This result was expected because the more E_{max} is increased the less the system is energy-constrained. We also observe that PFP_{ALAP} has the highest failure rate for all values of E_{max}. Both PFP_{ASAP} and PFP_{ST} have a lower failure while PFP_{ASAP} demonstrates the lowest one. To explain why PFP_{ST} fails to schedule some task sets which are schedulable with PFP_{ASAP} we have to examine its behavior closely. When the battery is at its minimum level, PFP_{ST} suspends the system as long as possible before the next execution while PFP_{ASAP} suspends the system only for one time unit. In this case PFP_{ST} may uselessly postpone executions and may accumulate an unbearable energy load for a future time. This can lead the system to replenish more time than the available slack-time and may lead to miss deadlines. PFP_{ALAP} suffers from the same problem than PFP_{ST} because it postpones execution as long as possible regardless to the battery level. When all jobs are postponed to the maximum and the system incurs a long execution period, it is impossible to introduce more replenishment times. Therefore, deadline misses may occur. Figure 3.7 on page 94 illustrates this phenomenon.



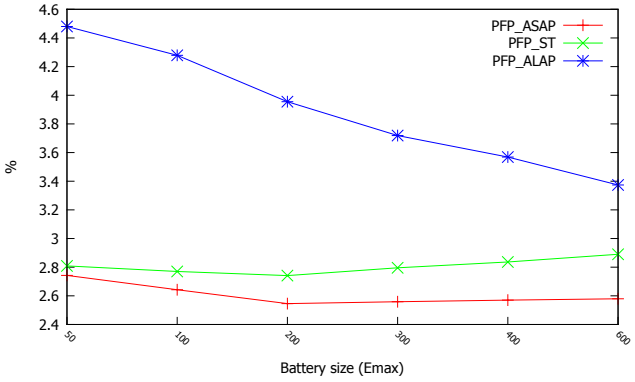
(a) Failure rate



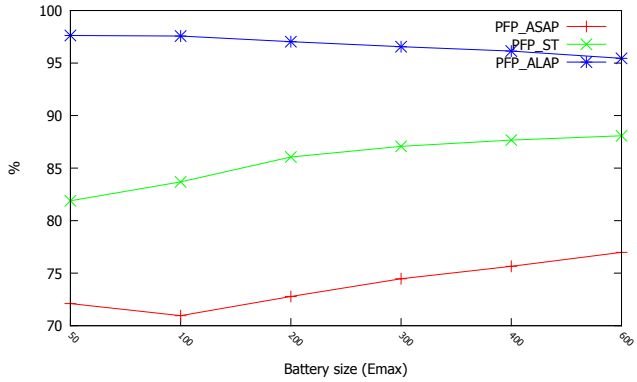
(b) Preemptions



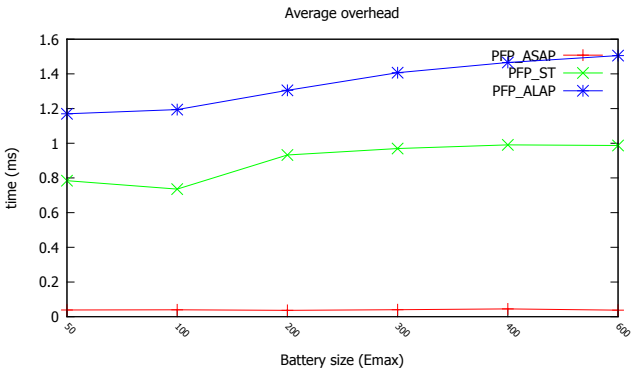
(c) Average idle period length



(d) Average busy period length



(e) Average energy level



(f) Average overhead

Figure 4.5: Comparison between PFP_{ALAP}, PFP_{ST} and PFP_{ASAP}

At time 8 a long busy period begins and the system has already consumed all the slack-time. The energy replenished during the former idle periods is not sufficient and the system runs out of energy. This lack of energy is due to the anticipated execution of job $J_{1,2}$. The PFP_{ASAP} algorithm prevents from this situation by starting executing jobs immediately.

As shown in Section 4.3.3 on page 118, PFP_{ASAP} is optimal for energy-non-concrete task sets, the simulations show that it has the lowest failure rate in all scenarios. All the task sets that are schedulable with PFP_{ALAP} and PFP_{ST} are still schedulable with PFP_{ASAP} but this is not true in the opposite way. However, the difference in the failure rate between PFP_{ST} and PFP_{ASAP} is relatively small, the study of the other metrics might be crucial.

Preemption Rate: The simulations show that increasing E_{max} helps to stabilize the number of preemptions. However PFP_{ASAP} demonstrates a very high number of preemptions regardless to E_{max} values. By construction, PFP_{ASAP} executes for one time unit then preempts tasks to check again if there is enough energy, while PFP_{ST} consumes all the slack-time available to replenish energy and avoid preemptions due to a lack of energy. PFP_{ALAP} does the same for each job activation.

Average Overhead: We observe that for every value of E_{max}, PFP_{ALAP} and PFP_{ST} have much higher average overhead than PFP_{ASAP}. This is due to the pseudo-polynomial complexity of the slack-time algorithm [DTB93]. PFP_{ALAP} computes slack-time whenever a job is requested and PFP_{ST} does the same but only if there is not enough energy while PFP_{ASAP} only needs to order the activated jobs.

Average Idle-Period and Busy-Period: These two metrics are closely linked to the number of preemptions. The longer the idle or busy periods are, the lower the number of preemptions is. We observe that PFP_{ASAP} has a high number of preemptions because it has short idle and busy periods. PFP_{ST} consumes all available slack-time to replenish energy, then, executes tasks while the battery level is sufficient. This maximizes the duration of both of the idle and the busy periods.

Average Energy Level: regardless to E_{max} values, PFP_{ALAP} has the highest average energy level and PFP_{ASAP} has the lowest one. Knowing that this metric is calculated only for feasible task sets, this result was expected because PFP_{ALAP} replenishes energy during long periods (idle periods) which increase the average battery level.

Varying task set Criminality

The aim of this experiment is to study the effect of the task set cardinal on the average overhead. The results are presented by Figure 4.6 on the next page and confirm our previous observations. Both of PFP_{ALAP} and PFP_{ST} have a very large overhead

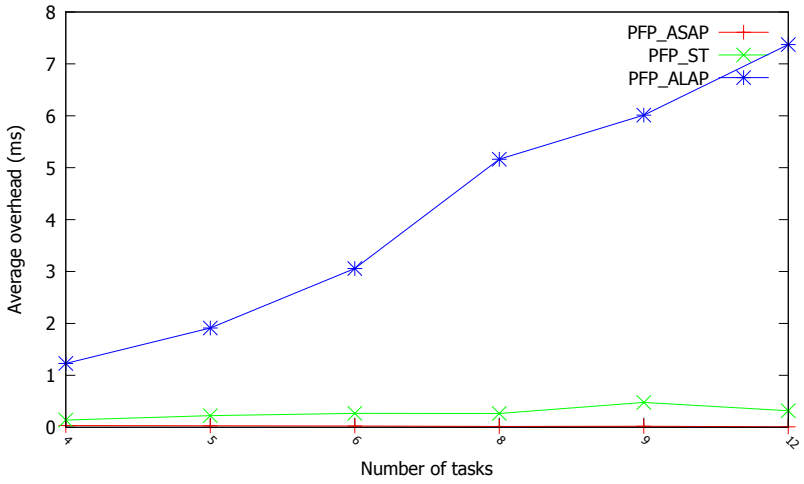


Figure 4.6: Number of tasks variation

-	PFP_{ALAP}	PFP_{ST}	PFP_{ASAP}
worst-case scenario	-	-	synchronous activations
Optimality	bad	bad	good
Failure rate	bad	good	good
Average overhead	bad	bad	good
preemptions	neutral	good	bad
Average idle-period	neutral	good	bad
Average busy-period	neutral	good	bad
Average energy level	neutral	good	bad

Table 4.1: PFP_{ALAP} vs PFP_{ST} vs PFP_{ASAP}

relative to PFP_{ASAP}. This is explained by the complexity of the slack-time computation algorithm.

Table 4.1 summarizes the performance of the evaluated algorithms.

PFP_{ASAP} is optimal for energy-non-concrete task sets and so has the lowest failure rate compared to the other algorithms. Moreover, it needs less battery capacity to operate. However, it increases the number of preemptions and context switches, which could be a serious limitation in practice. PFP_{ST} is not optimal but the simulations show that its failure rate is very close to the one of PFP_{ASAP}. Furthermore, it maximizes the average energy level and reduces the number of preemptions. The pseudo-polynomial complexity of stack-time calculation is the main drawback of PFP_{ST}, then, it cannot be used for systems with a large number of tasks. However, one can imagine using a slack-time approximation algorithm rather than an exact computation [Dav93].

Regarding PFP_{ALAP}, in addition to its non-optimality, simulations demonstrate very bad performance for all metrics.

4.5 Conclusion

In this chapter we studied deeply the PFP_{ASAP} scheduling algorithm. We firstly described in detail its scheduling schemes. Secondly, we presented and proved some of its properties. We proved that the synchronous release of all tasks when the battery is at the minimum level is the worst-case scenario that leads to the longest response time of all tasks. Then, we used this property to prove the optimality of PFP_{ASAP} for energy-non-concrete task sets that consume energy more than the replenishment during their executions. We also proposed a bound of the minimum battery capacity that keeps the schedulability of task set with PFP_{ASAP}. Thirdly, we built a schedulability condition based on PFP_{ASAP} algorithm. This condition is necessary and sufficient for energy-non-concrete task sets. Finally, we presented the results of empirical comparison of PFP_{ASAP} with other algorithms. This experiments showed that PFP_{ASAP} dominates the other algorithms in term of schedulability rate and average overhead. The results showed also that PFP_{ASAP} suffers from high number of preemptions due to its greedy behavior. Then, we conclude that the PFP_{ASAP} is a scheduling algorithm that behaves theoretically well but has a serious limitation in practice. Finally, we can conclude that PFP_{ASAP} is optimal but not applicable for platforms that does not support a high preemption and context switch rates. However, its optimality make it the reference algorithm for fixed-priority energy-harvesting systems.

We will show in the next chapter that PFP_{ASAP} loses its optimility when we relax some energy assumptions, the worst-case scenario and the schedulability test are no longer valid. Removing these assumptions makes the problem more complex and push us to study more scheduling algorithms and to look for more schedulability conditions. This is the topic of the next chapters.

Chapter 5

Approximate Response-Time Analysis for Energy-Harvesting Systems

Contents

5.1 Introduction

Previously, we provided schedulability analysis for PFP_{ASAP} that was restricted to systems where all tasks were consuming tasks (see Section 4.3.5 on page 121). We proved that PFP_{ASAP} is optimal with respect to all fixed-priority algorithms for energy-non-concrete periodic task sets, compliant with that restricted model.

In this chapter, we consider real-time task sets comprising two types of tasks: (i) *consuming tasks* that have a rate of energy consumption that is higher than the

replenishment rate, and (ii) *gaining tasks* that have a rate of energy consumption that is no more than the replenishment rate. Recall that the results presented in Chapter 4 were obtained considering only consuming tasks. We show that for this more general model, the critical instant leading to the worst-case response time of a task does not necessarily correspond to a synchronous release with all higher priority tasks, and so the analysis presented in Section 4.3 on page 111 is not applicable. For the more general model, we derive two response time upper bounds providing sufficient scheduling tests. We also prove that Deadline Monotonic priority assignment [LW82] is an optimal priority assignment policy with respect to these sufficient schedulability tests. The work presented in this chapter is the result of a collaboration with Prof. Rob Davis and mainly contains materials previously published in [Abd+14].

The remainder of the chapter is organized as follows. In Section 5.2 we recapitulate the system model, terminology and notation used in the this chapter. In Section 5.3 we introduce sufficient schedulability analysis for the more general task model with both consuming and gaining tasks. Section 5.4 provides a performance evaluation investigating the effectiveness and the tightness of these schedulability tests. Section 5.5 concludes the chapter.

5.2 Models and Notations

The model considered in this chapter is close to the one described in Section 3.2 on page 67. The deference here is that we relax some assumptions on tasks energy consumption and offsets. The considered model includes also concrete task sets.

Task Model

In addition to the notations presented in Chapter 4, the worst-case power consumption (i.e. energy used per unit of execution time) of a task τ_i is given by Pc_i. Thus the worst-case energy consumption equates to executing for the worst-case execution time, at the maximum rate of power consumption (i.e. $E_i = Pc_i \times C_i$). The set of tasks Γ is separated into two distinct subsets Γ_c and Γ_g. The first one Γ_c contains the *consuming tasks*, the ones that consume more energy than is replenished during their execution, whereas Γ_g is composed by the *gaining tasks*, that consume no more energy than is replenished during their execution. We have $\Gamma_c = \{\tau_i \in \Gamma, E_i > P_r \times C_i\}$ and $\Gamma_g = \{\tau_i \in \Gamma, 0 \leq E_i \leq P_r \times C_i\}$.

Energy Model

The energy model considered in this chapter is exactly the same as the one described in Section 3.2 on page 67.

5.3 Schedulability Analysis

In this section, we provide sufficient schedulability tests for systems with both consuming and gaining tasks. First we show that the critical instant for such task sets does not necessarily correspond to synchronous release. Lack of information about the actual worst-case scenario makes the schedulability analysis problem much more difficult. We address this problem by using the concept of priority level-i energy-busy-period. (see Definition 3.3 on page 70). The worst-case response time of task τ_i must necessarily occur within such a busy-period. We derive two upper bounds on the maximum length of this busy-period, which we then use to obtain upper bounds on the worst-case response time of the task. We use a similar approach to also derive response time lower bounds.

5.3.1 Worst-case scenario

When we consider only consuming tasks or only gaining tasks, the worst-case scenario (critical instant) occurs when all higher priority tasks are released simultaneously and the battery is at its minimum level. For the case when we have only gaining tasks, the response time analysis is the same as the classical formulation, since there are no delays due to energy considerations.

For the case where we have only consuming tasks, launching the tasks simultaneously with the battery at its minimum level maximizes the idle periods needed for energy replenishment. This increases the time required to complete the execution of higher priority tasks, which leads to the longest response time for each task, given by Equation 4.3 on page 113 as proved in Section 4.3.5 on page 121.

In contrast, when we consider a task set composed of both gaining and consuming tasks the worst-case scenario is not the same for all the tasks, it depends on the composition of the subset of higher priority tasks. If that subset contains both gaining and consuming tasks, then the worst-case scenario is not necessarily the synchronous activation of all the tasks with the minimum battery level.

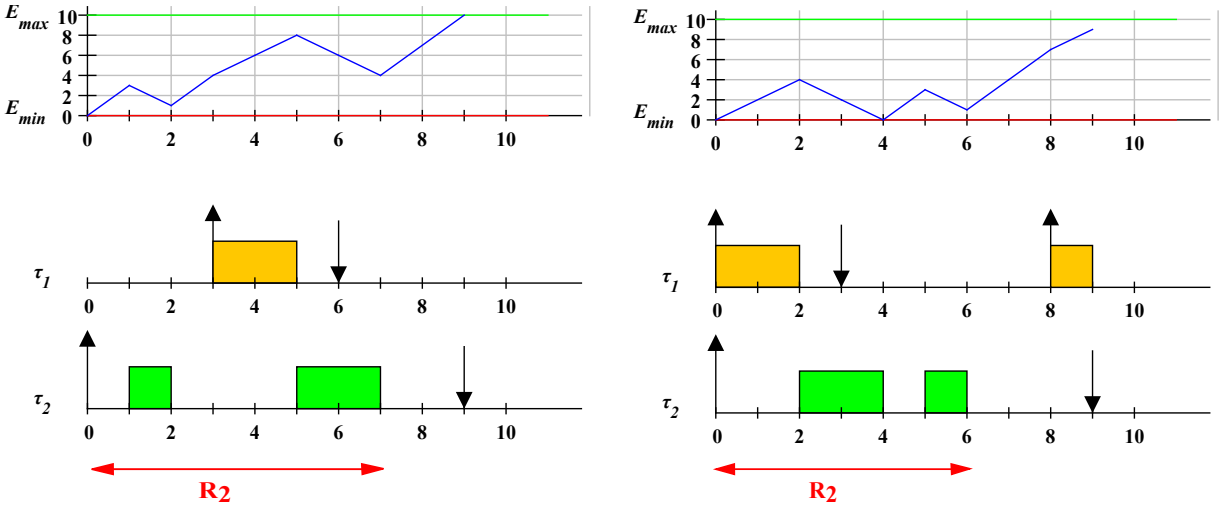
Figure 5.1 on the following page illustrates a situation where the response time of task τ_2 is longer ($R_2 = 7$) when a gaining task of higher priority is requested later, than it is with synchronous release ($R_2 = 6$). This is due to the fact that in the former case, task τ_2 suffers two replenishment delays (at time $t = 0$ and $t = 2$), whereas in the later case it suffers only one replenishment delay (at time $t = 4$). This happens because task τ_1 is a gaining task and there is a net increase in energy as it executes.

5.3.2 Sequences and Energy-busy-periods

We now introduce terminology and concepts that we use in proving key properties about scheduling systems with energy constraints.

Tasks	C_i	E_i	T_i	D_i
τ_1	2	2	8	3
τ_2	3	15	10	9

(a) Task set



(b) Asynchronous requests

(c) Synchronous requests

Figure 5.1: Worst-case scenario counter example

Definition 5.1 (Execution Unit)**.**
We use the term *execution unit* to refer to a non-divisible unit of execution of a job. An execution-unit has the same length as the basic time unit used to describe task execution times, and is of the same length for all tasks. ■

Definition 5.2 (Replenishment Unit)**.**
We use the term *replenishment unit* to refer to the minimum indivisible unit of idling time used to replenish energy. ■

Execution-units and replenishment-units are of the same duration.

Definition 5.3 (Execution Sequence)**.**
An execution *sequence* is a vector X of execution units from 1 to L_X, where L_X is the number of execution units in the sequence. Each element $X[m]$ of the sequence indicates the task that the execution unit belongs to. A sequence does not contain replenishment units, and so L_X does not necessarily represent the number of time units needed to execute the sequence. ■

We denote the energy required by execution unit $X[m]$ by $E_X[m]$. Further we use $E^*_X[m]$ to denote the total energy required by execution units from the start of the

sequence up to and including execution unit $X[m]$. Thus:

$$E^*_X[m] = \sum_{q=1\ldots m} E_X[q] \tag{5.1}$$

The minimum number of replenishment units $I_X[m]$ required to provide sufficient energy to execute $X[m]$ at the end of the subsequence $X[1]$ to $X[m]$ is given by:

$$I_X[m] = \max\left(0, \left\lceil \frac{E^*_X[m] - E(0)}{P_r} \right\rceil - m\right) \tag{5.2}$$

where $E(0)$ is the energy available at the start of the sequence.

We note that an earlier execution unit $X[k]$ may require more prior replenishment units than a later one $X[m]$ due to the presence of execution units of gaining tasks between $X[k]$ and $X[m]$, (i.e. $I_X[k] > I_X[m]$ where $m > k$). We use $I^*_X[m]$ to denote the minimum number of replenishment units required to execute all of the subsequence $X[1]$ to $X[m]$ in order.

$$I^*_X[m] = \max_{k=1\ldots m} (I_X[k]) \tag{5.3}$$

The elapsed time required to execute sequence X is given by $L_X + I^*_X[L_X]$.

Lemma 5.1.
For a fixed sequence X of execution units, the elapsed time for the sequence is maximised when the initial energy available is minimised, i.e. $E(0) = 0$.

Proof. Follows directly from Equation 5.2 and the formula for the elapsed time to execute the sequence: $L_X + I^*_X[L_X]$. □

Lemma 5.2.
Any sequence containing only execution units of consuming tasks requires the same elapsed time to execute irrespective of the order of its execution units provided that the set of execution units and the initial energy are the same. Similarly, any sequence containing only execution units of gaining tasks requires the same elapsed time to execute irrespective of the order of its execution units provided that the set of execution units is the same.

Proof. Case (i) sequence X contains solely execution units of consuming tasks. Since all execution units consume energy, then for every element $X[m]$, we have $E_X[m] > P_r$ and so $E^*_X[m+1] > E^*_X[m] + P_r$ hence the maximum number of prior replenishment units is required by the last element in the sequence and is given by:

$$I^*_X[L_X] = I_X[L_X] = \left\lceil \frac{E^*_X[L_X] - E(0)}{P_r} \right\rceil \tag{5.4}$$

Since the total energy $E^*_X[L_X]$ required by all elements in the sequence is independent of the order of the elements, the elapsed time $I^*_X[L_X] + L_X$ required to execute the sequence is also independent of the order of the elements.

Case (ii) sequence X contains solely execution units of gaining tasks. Since all execution units gain energy, no replenishment units are required and the elapsed time for the sequence equates to its length L_X irrespective of the order of the elements. □

Lemma 5.3.
Under PFP_{ASAP} scheduling, for a schedulable task τ_i, the worst-case response time R_i of the task equates to the longest possible priority level−i energy busy-period. Further, there exists a busy-period of this length that includes a single job of task τ_i, begins at the release of this job and ends with the final execution unit of the job.

Proof. As task τ_i has the lowest priority of any task executing in such a priority level−i energy busy-period, under PFP_{ASAP} scheduling the busy-period necessarily ends with the final execution unit of that task. This is the case because if there were any outstanding higher priority tasks, they would execute in preference to task τ_i.

As task τ_i has a constrained deadline and is schedulable (by the Lemma), it can only have one job starting from the release time in the busy-period, otherwise the completion of the previous job of task τ_i would have to take place after the release of the final job of the task implying (as $D_i \leq T_i$) that the previous job was unschedulable.

Let X be the sequence of execution units representing all execution in the busy-period. If the job of task τ_i was not released at the start of the busy-period, then we can move its release time back to the start of the busy-period. Since task τ_i has the lowest priority of any task in the busy-period, such a change cannot make any difference to the actual order of execution as represented by sequence X and so has no impact on the elapsed time required to execute the sequence. Such a change can therefore only increase the worst-case response time of the job. □

Lemma 5.3 proves that the worst-case response time for a task τ_i occurs in a priority level−i energy busy-period starting with the release of that task. However, as shown in Figure 5.1 on page 134, synchronous release of all higher priority tasks may not result in the worst-case response time for task τ_i. In general, we do not know what scenario, or pattern of releases of higher priority tasks will result in the worst-case response time for task τ_i; however, we can derive further information about possible worst-case scenarios.

Lemma 5.4.
The maximum possible number of jobs of a higher priority task τ_h causing interference in the longest priority level−i busy-period (characterising the worst-case response time of task τ_i) is given by $\lceil w/T_h \rceil$ where w is the length of the busy-period.

Proof. Lemma 5.3 shows that the busy-period starts (at time $t = 0$) with the release of task τ_i, hence at $t = 0$, there can be no jobs of higher priority tasks with outstanding execution, other than those also released at $t = 0$, otherwise the busy-period would have started earlier. It follows that the maximum number of higher priority jobs of task τ_h in the busy-period is given by $\lceil w/T_h \rceil$. □

5.3.3 Response Time Upper Bounds

We do not know up to know the precise pattern of releases of higher priority jobs that leads to the worst-case response time for task τ_i, we cannot determine the exact worst-case response time. Instead, we derive an upper bound R_i^{UB1} and then a tighter upper bound R_i^{UB2} on the exact worst-case response time R_i, where $R_i^{UB1} \geq R_i^{UB2} \geq R_i$. These upper bounds provide sufficient schedulability tests $UB1$ and $UB2$ respectively, where $UB2$ dominates $UB1$.

The process we use to obtain these upper bounds is similar to the classic formulation of response time analysis presented in Section 1.5 on page 40. We aim to find the smallest interval w, for which an upper bound on the response time of task τ_i, considering the maximum possible interference from higher priority tasks released in that interval, equates to the length of the interval. The value of w then provides an upper bound on the worst-case response time of task τ_i.

We require a function $F(w)$ that upper bounds the length of the longest priority level$-i$ energy busy-period formed by a single job of task τ_i and jobs of higher priority tasks released during an interval of length w. Provided that $F(w)$ is a monotonically non-decreasing function of w, then we may obtain an upper bound on the worst-case response time of task τ_i corresponding to the smallest value of $w > 0$ that satisfies:

$$w = F(w) \tag{5.5}$$

Equation 5.5 may be solved using fixed point iteration starting with $w = C_i$ and ending on convergence or when $w > D_i$ in which case the task is deemed unschedulable.

5.3.4 Upper Bound R^{UB1}

We now derive a simple upper bound R_i^{UB1} on the worst-case response time of task τ_i. First we prove a Lemma used in its derivation.

Lemma 5.5.
Let X be some arbitrary sequence of execution units of tasks of priority i or higher, and Y be the equivalent sequence re-ordered such that all execution units of consuming tasks come before all execution units of gaining tasks. The elapsed time required to complete sequence Y is no shorter than that required to complete sequence X.

Proof. We may obtain sequence Y from sequence X by an iterative process of choosing the first execution unit belonging to any gaining task (at position g) and swapping it with that of the last execution unit of any consuming task (at position k) provided that $g < k$. Repeating this process until all consuming execution units come before all gaining execution units transforms sequence X into sequence Y. Let the new sequences produced by this process be $X_1 = X, X_2, X_3 \ldots X_n = Y$. Note at most $L_X/2$ swaps are required. We now show that each swap transforming sequence X_p into sequence X_s where $s = p + 1$, results in an elapsed time for sequence X_s that is no shorter than

that for X_p, and hence by induction that the elapsed time for sequence Y is no shorter than that for sequence X. Let $X_p[g]$ and $X_p[k]$ be the elements being swapped where $g < k$. Since $X_p[g]$ is an execution unit of a gaining task and $X_p[k]$ is an execution unit of a consuming task, the energy required for these execution units has the relationship $E_{X_p}[g] < E_{X_p}[k]$. Recall that $E^*_{X_p}[m]$ is the energy required to execute all execution units in the subsequence from $X_p[1]$ to $X_p[m]$. It follows that:

$$\forall m, 1 \leq m < g \;\; E^*_{X_s}[m] = E^*_{X_p}[m]$$

$$\forall m, g \leq m < k \;\; E^*_{X_s}[m] = E^*_{X_p}[m] + E_{X_s}[k] - E_{X_p}[g]$$

$$\forall m, k \leq m \;\; E^*_{X_s}[m] = E^*_{X_p}[m]$$

$$\Rightarrow \;\; E^*_{X_s}[m] \geq E^*_{X_p}[m]$$

Hence the minimum number of replenishment units required to execute the subsequences from the 1st to the $m-$th element of X_p and X_s have the following relationship: $I^*_{X_s}[m] \geq I^*_{X_p}[m]$ (see Equations 5.2 on page 135 and 5.3 on page 135). Since the number of execution units in each sequence (X_s and X_p) is the same (i.e. $L_{X_p} = L_{X_s}$), we have: $L_{X_s} + I^*_{X_s}[m] \geq L_{X_p} + I^*_{X_p}[m]$. Thus the elapsed time required to execute sequence X_s is no shorter than that required for sequence X_p. Induction over at most $L_X/2$ steps proves that the elapsed time required to complete sequence Y is no shorter than that required for sequence X.

□

Theorem 5.1.
An upper bound on the worst-case response time for task τ_i for a set of jobs released in a window of length w can be obtained by assuming that there is one job of task τ_i and $\lceil w/T_h \rceil$ jobs of each higher priority task τ_h. Further, the upper bound is obtained from a sequence Z of the execution units of these jobs where all the consuming execution units are before all the gaining execution units.

Proof. Let X be the sequence of execution units that results in the longest priority level-i energy busy-period under PFP_{ASAP} scheduling, and hence the longest response time for task τ_i), for a set of jobs released in a window of length w. The elapsed time for the sequence is given by $L_X + I^*_X[L_X]$ where L_X is the length of the sequence and $I^*_X[L_X]$ is the total number of replenishment units required. Lemma 5.5 on the previous page shows that the elapsed time required to execute a sequence Y is no shorter than that required to execute sequence X, where sequence Y comprises the execution units in X re-ordered such that all execution units of consuming tasks are placed before execution units of gaining tasks. Note that at this point we do not know how many jobs of higher priority tasks are present in sequence X and therefore also in sequence

Y; however, by Lemma 5.4 on page 136 we know that the maximum number of jobs of a higher priority task τ_h that could be present is $\lceil w/T_h \rceil$. Hence we add execution units to sequence Y as necessary to account for any shortfall in the number of jobs in X below this value, thus forming sequence Z. (Consuming execution units are added at the start of the sequence and gaining execution units at the end). We note that such additional execution units cannot reduce the elapsed time required to execute the sequence since all execution units of both consuming and gaining tasks require a positive amount of energy. Sequence Z (as described in the Theorem) therefore requires an elapsed time to execute that is no smaller than that of sequence X. □

We use Theorem 5.1 on the preceding page to formulate the workload function $W_i^{UB1}(w)$ $W_i^{UB1}(w)$ for upper bound R_i^{UB1}. We assume that the initially available energy is zero, the number of jobs of task τ_i and each higher priority task released in an interval of length w is given by $\lceil w/T_h \rceil$, and that all execution units of consuming jobs are executed before all execution units of gaining tasks. We note that the number of jobs considered equates to synchronous release of all the tasks, with re-release as soon as possible. This is equivalent to the critical instant for classical tasks without energy considerations. Although this is not necessarily the worst-case scenario for tasks that require energy (see Figure 5.1 on page 134 for a counter example), it is the worst-case scenario with respect to how our upper bounds are computed. The workload function for R_i^{UB1} is given by:

$$W_i^{UB1}(w) = \left\lceil \frac{\sum_{j \le i \wedge \tau_j \in \Gamma_c} \left\lceil \frac{w}{T_j} \right\rceil \times E_j}{P_r} \right\rceil + \sum_{j \le i \wedge \tau_j \in \Gamma_g} \left\lceil \frac{w}{T_j} \right\rceil \times C_j \tag{5.6}$$

where the first term represents the total time to complete the execution units of consuming tasks, which are in effect energy-bound, and the second term is the time taken to complete the execution units of gaining tasks, which are processing time bound.

Observe that $W_i^{UB1}(w)$ is a monotonically non-decreasing function of w since all terms are positive and w only appears in the numerator of ceiling functions. Further $W_i^{UB1}(w) \geq C_i$ since $\lceil C_i/T_i \rceil = 1$ and if task τ_i is a consuming task then $E_i/P_r \geq C_i$, hence C_i serves as a valid initial value for fixed point iteration.

We note that in the case where all tasks are gaining tasks and so energy is not a consideration, Equation 5.6 reduces to the exact analysis for classical tasks. Further, in the case where all tasks are consuming tasks Equation 5.6 reduces to the analysis for that case given by Equation 4.3 on page 113.

5.3.5 Upper Bound R^{UB2}

We can refine the first upper bound by considering a more realistic scenario. More precisely, the idea is to take into consideration the fact that some gaining jobs cannot

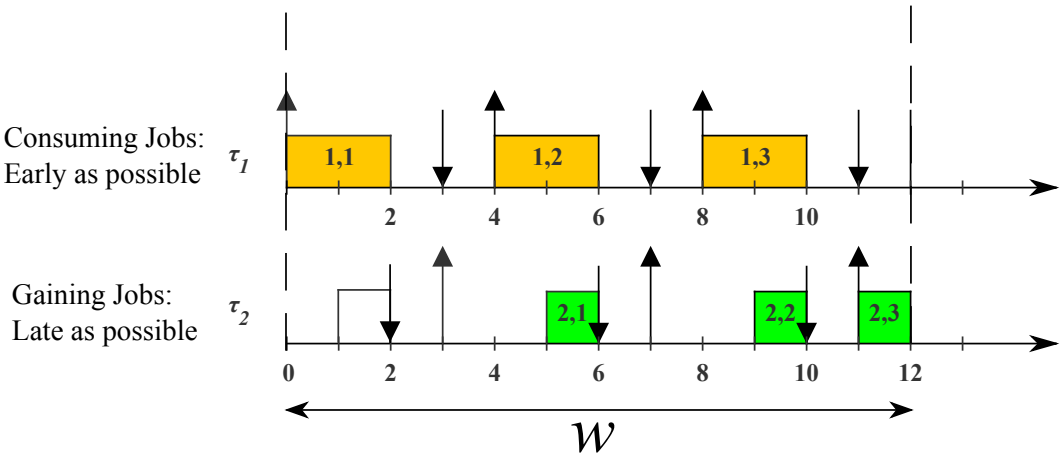


Figure 5.2: Dummy schedule used in the construction of $UB2$

be executed after some consuming ones, because of their respective deadlines and releases, which define sub-intervals in which they are forced to run when the system is schedulable.

This idea is illustrated in Figure 5.2, which shows three jobs of a consuming task and three jobs of a gaining task. We know that *provided the tasks are schedulable*, job 1 of the gaining task must run before job 3 of the consuming task. This information can be used to compute a tighter upper bound on the maximum time needed to complete all of the jobs in the interval.

We now derive our second upper bound R_i^{UB2}. The workload function $W_i^{UB2}(w)$ for R_i^{UB2} is derived from a dummy schedule and the sequence of execution units obtained from it. Construction of the dummy schedule is as illustrated in Figure 5.2. The dummy schedule is measured in time units and covers the interval $[0, w)$. It has a timeline for each task of priority i and higher, with one job of task τ_i and $\lceil w/T_h \rceil$ jobs of each higher priority task τ_h. Jobs of consuming tasks (including task τ_i if it is one) are placed in the dummy schedule starting with a release at $t = 0$, with subsequent releases as soon as possible. These jobs are assumed to execute immediately. For gaining tasks (including task τ_i if it is one) we first align the release of the last job at time $w - C_i$ this job is assumed to execute immediately. Previous jobs of the gaining task are then released as late as possible respecting the release time of the subsequent job, and assumed to execute as late as possible i.e. just prior to their deadlines. Thus jobs of gaining tasks are added from the end of the dummy schedule working backwards in time, and jobs of consuming tasks are added from the start of the dummy schedule working forwards in time. Note that there may be overlaps between the schedules where more than one task appears to execute at the same time. This is shown in Figure 5.2: intervals $[4, 5]$ and $[9, 10]$.

From the dummy schedule, we derive a sequence Z of execution units. This sequence is composed by starting at the beginning of the dummy schedule with an empty sequence

and appending all gaining tasks with execution in that time unit onto the sequence, followed by all consuming tasks with execution in the same time unit. This process is then repeated for all subsequent time units until all execution units have been collected. Note ties between execution units of two or more gaining tasks or two or more consuming tasks may be broken arbitrarily; however, all execution units of gaining tasks associated with some time unit t are placed into the sequence ahead of all execution units of consuming tasks associated with the same time unit. All execution units associated with a later time unit e.g. $t+1$ appear later in the sequence than those associated with an earlier time unit t (We note that clashes may safely be resolved by giving preference to gaining tasks, since those execution units must necessarily take place by that time otherwise a deadline will be missed. Execution units of consuming tasks could and would have been executed earlier in any real schedule that meets all deadlines).

Finally, the workload function $W_i^{UB2}(w)$ is computed giving the elapsed time required to execute sequence Z, assuming that the initial energy is at its minimum. This can be done via simulation, limited to at most a length of time D_i.

Theorem 5.2.
An upper bound on the worst-case response time for task τ_i for a set of jobs released in a window of length w, where no higher priority jobs miss their deadlines, can be obtained by assuming that there is one job of task τ_i and $\lceil w/T_h \rceil$ jobs of each higher priority task τ_h, with the upper bound equating to the maximum time required to execute a sequence Z of the execution units of these jobs constructed according to the rules and dummy schedule construction described previously.

Proof. Let X be the sequence of execution units that results in the longest priority level$-i$ energy busy-period under PFP_{ASAP} scheduling (and hence response time for task τ_i) for a set of jobs released in a window of length w where all higher priority tasks meet their deadlines. For each task τ_h, let N_h be the number of jobs in sequence X. Consider a sequence Y formed by constructing a dummy schedule of length w including N_h jobs of each task τ_h and one job of task τ_i and applying the rules stated above for ordering execution units (Recall from Lemma 5.3 on page 136 that there is only one job of task τ_i in the busy-period, and hence in sequence X). Sequence Y and sequence X contain an identical set of execution units. Since no deadlines are missed when sequence X is executed, and the dummy schedule used to construct sequence Y places execution units of gaining jobs as late as possible without missing a deadline, in relation to the execution units of consuming jobs which are placed as early as possible without invalidating minimum inter-arrival constraints. It follows that sequence Y can be obtained from sequence X by a process of swapping earlier gaining execution units for later consuming execution units (Note that re-ordering of sub-sequences consisting of solely gaining execution units or solely consuming execution units may also be needed to obtain precisely the same sequence; however, Lemma 5.2 on page 135 shows that this re-ordering among execution units of the same type has no effect on the elapsed time required to execute the complete sequence). Finally, we compare sequence Z obtained

as described in the Theorem, and sequence Y. If sequence Y contains the maximum number of jobs $\lceil w/T_h \rceil$ of each task that may be released in a window of length w, then it is identical to sequence Z. Otherwise, sequence Z may be obtained from sequence Y by adding execution units for any missing jobs where $\lceil w/T_h \rceil > N_h$. Since all execution units require energy, addition of execution units into the sequence at any point cannot decrease the elapsed time required to execute the sequence. Hence the elapsed time required to execute sequence Z is no shorter than that required to execute sequence X. □

Theorem 5.2 on the previous page shows that $W_i^{UB2}(w)$ provides a valid upper bound on the worst-case response time for task τ_i considering all jobs released in a window of length w. In order to use $W_i^{UB2}(w)$ in a fixed point iteration to determine an upper bound on the worst-case response time of task τ_i we must also show that $W_i^{UB2}(w)$ is a monotonic non-decreasing function of w, and that $W_i^{UB2}(w) > C_i$, so that we may use C_i as an initial value. The latter is trivially the case since a single job of task τ_i is always included in the workload and takes at least time C_i to execute.

Theorem 5.3.
$W_i^{UB2}(w)$ is a monotonically non-decreasing function of w.

Proof. Consider increasing the length of the window from some arbitrary value w to $w + v$, comparing the dummy schedules used to derive $W_i^{UB2}(w)$ and $W_i^{UB2}(i, w + v)$ there are two effects: (i) all execution units of gaining jobs move to a later time e.g. $t + v$ rather than t, (ii) new execution units of gaining jobs may be added near the start of the schedule and new execution units of consuming jobs may be added near the end of the schedule. Consider sequence X formed in deriving $W_i^{UB2}(w)$ and sequence Y formed in deriving $W_i^{UB2}(i, w + v)$ but omitting all of the execution units of the new jobs from (ii). Sequences X and Y contain the same set of elements. Since all execution units of gaining jobs are v time units later in the dummy schedule used to construct sequence Y, it follows that sequence Y can be formed from sequence X by swapping later gaining execution units in X for earlier consuming execution elements, and as necessary re-ordering sub-sequences containing solely gaining or solely consuming execution units (Lemma 5.2 on page 135). Hence the elapsed time required to execute sequence Y is no shorter than that required for sequence X. Consider a further sequence Z, if there were no additional jobs from (ii) then sequence Z is identical to sequence Y, otherwise it may be obtained from sequence Y by adding execution units for the missing jobs. Since all execution units require energy, addition of execution units into a sequence at any point cannot decrease the elapsed time required to execute the sequence. Hence the elapsed time required to execute sequence Z is no shorter than that required to execute sequence X.

□

We now return to the assumption in Theorem 5.2 on the previous page that all deadlines of higher priority tasks are met. This might seem to imply that task

schedulability must be checked highest priority first; however, this is not necessarily the case. Consider what happens if we test task schedulability lowest priority first. We tentatively test the schedulability of task τ_i on the assumption that all higher priority tasks will later be found to be schedulable. If task τ_i is deemed schedulable (caveat this assumption), then we go on to check higher priority tasks. If some higher priority task τ_h is subsequently found to be unschedulable, then this undermines the validity of our schedulability test for task τ_i; however, this is now of no consequence, since the task set is in any case unschedulable due to task τ_h. If instead, all higher priority tasks are found to be schedulable, then the schedulability test for task τ_i is validated (We note that the schedulability or otherwise of a lower priority task τ_i has no impact on the schedulability of any higher priority task τ_h).

Theorem 5.4.
Schedulability test $UB2$ dominates test $UB1$ i.e $R_i^{UB1} \geq R_i^{UB2}$.

Proof. We prove the theorem by showing that $W_i^{UB2}(w) \leq W_i^{UB1}(w)$. Consider the sequence Y representing $W_i^{UB1}(w)$ and the sequence X representing $W_i^{UB2}(w)$. The sequences contain the same elements; however, in sequence Y all of the consuming execution units are before all of the gaining execution units, hence by Lemma 5.5 on page 137, the elapsed time required to complete sequence Y is no shorter than that required to complete sequence X. □

5.3.6 Battery Capacity

We now return to a consideration of the minimum battery capacity $\mathcal{C}_{min}$ (we note $\mathcal{C}_{min}^{UBi}$ the minimum battery capacity for UBi). For the sufficient test $UB1$ to be valid, we require that. $\mathcal{C}_{min}^{UB1} \geq \max(\max_{\forall i}(P_i) - P_r, P_r)$. This small battery capacity is sufficient, since in computing an upper bound on the worst-case response time, $UB1$ assumes that all consuming execution units come before all gaining execution units. The minimum battery capacity needed to execute this sequence without impinging on the elapsed time required is simply enough to execute the most costly unit of execution in terms of energy, which equates to $\max_{\forall i}(P_i) - P_r$ or $\max_{\forall i}(E_i/C_i) - P_r$. In addition, the battery capacity cannot be less then P_r, the maximum amount of energy replenished during one time unit.

By comparison, for the sufficient schedulability test $UB2$ to be valid, it suffices to have a minimum battery capacity $\mathcal{C}_{min}$ that equates to at least the total *net* energy required to execute all of the consuming jobs in the longest possible priority level-n energy busy period. Such a store of energy upper bounds that which can ever usefully be deployed to execute consuming jobs in any possible busy period. Having a larger battery capacity than this is equivalent in terms of task response times to having infinite battery capacity Note that by the total *net* energy required by consuming jobs, we mean the energy they consume less the energy generated while they actually execute. Since the longest possible level-n energy-busy-period cannot be greater than the longest task deadline,

otherwise the system would be unschedulable, we can upper bound the battery capacity required as follows: $\mathcal{C}_{min}^{UB2} \geq \max\left(\sum_{\forall i} \left\lceil \frac{\max_{\forall j}(D_j)}{T_i} \right\rceil \times \max\left(E_i - C_i \times P_r, 0\right), P_r\right)$.

5.3.7 Response Time Lower Bound

In this section, we derive an analytical lower bound $R_i^{LB1} \leq R_i$ on the worst-case response time of task τ_i. To obtain the lower bound, we analyse a specific scenario that corresponds to the synchronous release of task τ_i along with all higher priority tasks, which are then assumed to be re-released as soon as possible. Further, we assume that the initial energy is a minimum i.e. $E(0) = 0$. Although this is not necessarily the worst-case scenario, it is a valid scenario and hence suffices to provide a valid lower bound on the longest priority level$-i$ energy-busy-period and hence the worst-case response time of task τ_i.

We obtain the lower bound response time R_i^{LB1} via fixed point iteration, using a workload function $W_i^{LB1}(w)$ that is monotonically non-decreasing in w and lower bounds the elapsed time needed to execute all jobs of tasks of priority i or higher released in an interval of length w starting with a synchronous release.

Lemma 5.6.
Let X be some arbitrary sequence of execution units of tasks of priority i or higher, and Y be the equivalent sequence re-ordered such that all execution units of consuming tasks come after all execution units of gaining tasks. The elapsed time required to complete sequence X is no shorter than that required to complete sequence Y.

Proof. Follows by applying similar reasoning to the proof of Lemma 5.5 on page 137. □

Theorem 5.5.
A lower bound on the worst-case response time for task τ_i assuming synchronous release with all higher priority tasks resulting in a priority level$-i$ energy busy-period of at least length w, can be obtained by assuming that there is one job of task τ_i and $\lceil w/T_h \rceil$ jobs of each higher priority task τ_h in the busy-period. Further the lower bound equates to the time required to execute a sequence Z of the execution units of these jobs where all the consuming execution units are after all the gaining execution units, and the initial energy is a minimum.

Proof. By the theorem, the busy-period is at least w long, hence under PFP_{ASAP} scheduling all $\lceil w/T_h \rceil$ jobs of each higher priority task τ_h released during the interval $[0, w)$ must complete before the single job of task τ_i. Let X be the sequence of execution units of all of the jobs under PFP_{ASAP} scheduling. The elapsed time required to execute sequence X lower bounds the worst-case response time of task τ_i. Further, let Z be (as per the theorem) the same set of execution units as sequence X re-ordered such that all the consuming execution units are after all the gaining execution units. By Lemma 5.6, the elapsed time to execute sequence Z is no longer than that required to execute sequence X. □

We use Theorem 5.5 on the preceding page to formulate our lower bound workload function $W_i^{LB1}(w)$. We assume that the initially available energy is zero, the number of jobs of task τ_i and each higher priority task τ_h released in an interval of length w is given by $\lceil w/T_h \rceil$ and that all execution units of consuming jobs are executed after all execution units of gaining tasks.

$$X_i^g = \sum_{h \in hep(i)}^{\tau_h \in \Gamma_g} \left\lceil \frac{w}{T_h} \right\rceil \times C_h \quad , \quad X_i^c = \sum_{h \in hep(i)}^{\tau_h \in \Gamma_c} \left\lceil \frac{w}{T_h} \right\rceil \times C_h$$

$$Y_i^c = \sum_{h \in hep(i)}^{\tau_h \in \Gamma_c} \left\lceil \frac{w}{T_h} \right\rceil \times E_h \quad , \quad Y_i^g = \sum_{h \in hep(i)}^{\tau_h \in \Gamma_g} \left\lceil \frac{w}{T_h} \right\rceil \times E_h$$

$$W_i^{LB1}(w) = X_i^g + \max\left(X_i^c, \left\lceil \frac{Y_i^c - (X_i^g \times P_r - Y_i^g)}{P_r} \right\rceil\right) \tag{5.7}$$

Finally, in order to use the workload function $W_i^{LB1}(w)$ in a fixed point iteration to determine the lower bound R_i^{LB1} on the worst-case response time of task τ_i, we must show that $W_i^{LB1}(w)$ is a monotonically non-decreasing function of w.

Theorem 5.6.
$W_i^{LB1}(w)$ is a monotonically non-decreasing function of w.

Proof. Consider the formula for $W_i^{LB1}(w)$. Since w appears only in the ceiling functions, it follows that X_i^g, X_i^c and Y_i^g are all non-decreasing functions of w. Further, $(X_i^g P_r - Y_i^g)$ represents the net energy increase while all the gaining jobs execute. Since every execution unit of a gaining task is by definition energy positive, this quantity is also a non-decreasing function of w. Thus $Y_i^c - (X_i^g P_r - Y_i^g)$ may decrease with increasing w. The largest possible decrease is obtained when Y_i^c remains at the same value, while $(X_i^g P_r - Y_i^g)$ increases, hence $\lceil (X_i^g P_r - Y_i^g)/P_r \rceil$ decreases. However, such a decrease is always at least compensated for by the increasing value of the first term in Equation 5.7, i.e. X_i^g. This happens because the additional energy made available by each execution unit of an additional gaining job is no more than that available from a replenishment unit. Hence $\lceil (X_i^g P_r - Y_i^g)/P_r \rceil$ cannot decrease by more than X_i^g increases. We note that monotonicity can also easily be seen by considering the sequence Z (in Theorem 5.5 on the preceding page) which can only take a longer elapsed time to execute with the addition of further jobs, since all execution units require a positive amount of energy. □

We note that a tighter lower bound can be obtained via the simple expedient of simulating the actual schedule of execution starting from synchronous release of task τ_i and all higher priority tasks. We return to this point in Section 5.4 on page 147.

5.3.8 Priority Assignment

The *DM* priority assignment is optimal for fixed-priority preemptive scheduling of constrained deadline tasks conforming to the classical task model where energy is not considered. In Section 4.3.4 on page 119, we extended the optimality of *DM* to energy-non-concrete task sets that are composed of only consuming tasks. However, when we relax the assumption on tasks energy consumption, i.e. gaining tasks are accepted, the proof provided in Section 4.3.4 is no longer valid. In this section, we show that *DM* priority assignment is also optimal with respect to our sufficient schedulability tests $UB1$ and $UB2$, for energy-constrained systems with both consuming and gaining tasks. This does not mean that *DM* is optimal for energy-work-conserving scheduling in the general case, but only with respect to the necessary schedulability conditions based on $UB1$ and $UB2$.

Definition 5.4 (Optimal Priority Assignment Policy)**.**
A priority assignment policy P is said to be optimal with respect to a schedulability test S, if for every task set τ where there exists some priority assignment Q such that the task set is schedulable according to test S, then τ is also schedulable according to test S with the priority ordering given by policy P. ■

Theorem 5.7.
Deadline Monotonic (DM) priority assignment is an optimal priority assignment policy with respect to sufficient schedulability test S ($UB1$ or $UB2$) for task sets comprising any arbitrary combination of consuming and gaining tasks.

Proof. To prove the theorem, we show that any task set Γ that is schedulable according to test S ($UB1$ or $UB2$) under some priority ordering Q remains schedulable according to test S under deadline monotonic priority order P. We do this by transforming priority order Q into priority order P by swapping the priorities of tasks that are adjacent to each other in the priority order, but out of *DM* order. We show that on every swap the task set remains schedulable according to test S. Let τ_A and τ_B be two tasks in τ which are at adjacent priorities under the initial, schedulable priority ordering, with $D_A > D_B$ and τ_A at a higher priority k than τ_B, which has a priority $i = k + 1$ (i.e. the tasks are out of *DM* order). Let the upper bound response time of task τ_B according to schedulability test S be R_i^{UB} in the initial priority order. We now swap the priorities of the tasks, so that τ_B has the higher priority. We consider the following groups of tasks:

(i) $hp(k)$: these tasks have higher priorities than either τ_A or τ_B and so their upper bound response times, according to test S ($UB1$ or $UB2$), are unchanged by the swap.

(ii) $lp(i)$: these tasks have lower priorities than either τ_A or τ_B and so their upper bound response times, according to test S ($UB1$ or $UB2$), are unchanged by the swap, since interference from higher priority tasks does not, *according to the test*, depend on the relative priority order of those tasks.

(iii) task τ_B: now has a higher priority than τ_A, and so is only subject to interference from tasks in $hp(k)$, rather than $hp(k) \cup \tau_A$, hence τ_B remains schedulable.

(iv) task τ_A: is now at priority i with τ_B at higher priority. From the previous schedulable priority ordering, we have $R_i^{UB} \leq D_B \leq T_B$ and $D_B < D_A \leq T_A$, hence $w = R_i^{UB}$ was computed by test S by including exactly one job of τ_A, one job of τ_B and $\lceil w/T_h \rceil$ jobs of each higher priority task $\tau_h \in hp(k)$. We observe that the computation of the busy-period length w by test S ($UB1$ or $UB2$) depends only on this set of jobs and not on their relative priorities. We now consider $w = R_i^{UB}$ as a possible value for the response time of task τ_A under the new priority ordering. As $R_i^{UB} \leq T_B$, then there is only one job of task τ_B released in an interval of length w, along with $\lceil w/T_h \rceil$ jobs of each higher priority task $\tau_h \in hp(k)$, and the single job of task τ_A. Therefore, according to test S, R_i^{UB} is *also* the upper bound response time for task τ_A when it is at priority i. Since $R_i^{UB} \leq D_B < D_A$, it follows that task τ_A is schedulable at priority i. □

Recall the exact test for PFP_{ASAP} scheduling with only consuming tasks presented in Equation 4.3 on page 113 which is based on exact response time analysis (see Equation 4.3 on page 113). We note that the $UB1$ and $UB2$ tests reduce to Equation 4.3 on page 113 when there are only consuming tasks, and hence it follows from Theorem 5.7 on the preceding page that DM priority assignment is also optimal in that case. Similarly, if all tasks are gaining tasks, then the $UB1$ and $UB2$ tests reduce to the classical exact test (Equation 1.14 on page 45) for FTP without energy considerations. DM priority assignment is again optimal in that case [LW82].

We note that it remains an open question whether Deadline Monotonic priority assignment is optimal with respect to an exact analysis for constrained deadline task sets with both consuming and gaining tasks scheduled by PFP_{ASAP}.

5.4 Performance Evaluation

In this section, we present the results of an empirical investigation, examining the effectiveness of the sufficient schedulability tests presented in this chapter.

5.4.1 Taskset generation

To perform these experiments, we randomly generated approximately 40000 task sets, varying the processor utilization, the energy utilization, and the percentage of gaining tasks.

We varied U and Ue in the range $[0.05, 1]$ in steps of 0.05. The proportion of gaining tasks was varied from 0% to 100% in steps of 10% for each pair of values (U, Ue), hence we obtained 100 distinct task sets for each pair (U, Ue). Each tasks set comprised 10 tasks.

The task parameters were randomly generated as follows: task processor utilization ($U_i = C_i/T_i$) using the *UUnifast Discard* algorithm [BB05], task energy utilization

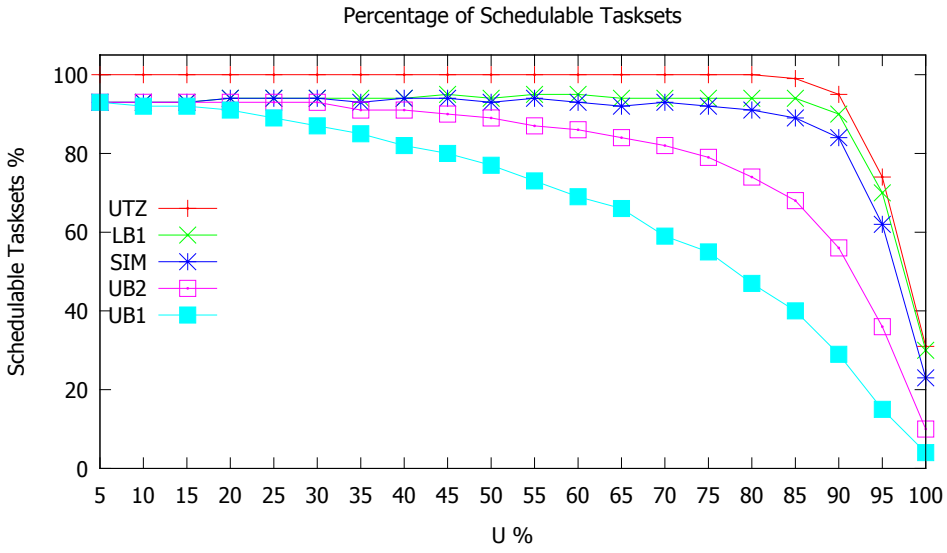


Figure 5.3: Percentage of task sets schedulable

($Ue_i = E_i/T_i \times P_r$) using an adapted version of *UUnifast Discard* to control the type of task generated (gaining or consuming), and periods randomly generated between 2 and 25200 time units with a hyper-period limitation technique [MG01].Task deadlines were implicit and the rate of energy replenishment P_r i was set to 15,

5.4.2 Schedulability tests investigated

We investigated the performance of the following schedulability tests.

- UTZ: the exact test for FTP ignoring energy constraints. This was used to provide a schedulability bound, considering only processing time.
- SIM: is an empirical necessary test based on simulating the schedule of PFP_{ASAP} over more than twice the hyper-period, starting with synchronous release and the minimum energy level. This is not guaranteed to reveal the real worst-case scenario, but can be used as a reference for comparison.
- $UB1$: the sufficient test presented in Section 5.3.4 on page 137.
- $UB2$: the sufficient test presented in Section 5.3.5 on page 139.
- $LB1$: the necessary test presented in Section 5.3.7 on page 144.

Figure 5.3 shows how the percentage of task sets that are deemed schedulable by each of the tests varies with processor utilization. The UTZ test has notionally the highest performance since it ignores energy considerations. When energy is considered, UTZ, $LB1$ and SIM provide necessary tests, upper bounding the number of task sets that could possibly be schedulable. An exact test considering energy would fall

somewhere between SIM and $UB2$. We observe that the results confirm that $UB2$ provides a tighter bound than $UB1$, with a larger improvement at higher utilization levels.

Weighted Schedulability we present a further set of experiments showing how schedulability depends on different parameters, including energy utilization and the proportion of gaining tasks, via the Weighted Schedulability Measure [BBA10]. As well as processor utilization, task set schedulability is dependent on a number of other key parameters, including: energy utilization, and the percentage of gaining tasks. Evaluating all possible combinations of these parameters is not possible, instead, the evaluation in this section varies one parameter at a time, with the results presented in terms of the weighted schedulability measure [BBA10].

The figures in this section show the weighted schedulability measure $W_y(p)$ for each schedulability test y as a function of parameter p. For each value of p, this measure combines results for all of the task sets Γ generated for all of a set of equally spaced utilization levels (5% to 100% in steps of 5%).

Let $S_y(\Gamma, p)$ be the binary result (1 or 0) of schedulability test y for a task set Γ with parameter value p:

$$W_y(p) = \left(\sum_{\forall \Gamma} U_\Gamma \times S_y(\Gamma, p) \right) / \sum_{\forall \Gamma} U_\Gamma \tag{5.8}$$

where U_Γ is the processor utilization of taskset Γ. The weighted schedulability measure reduces what would otherwise be a 3-dimensional plot to 2 dimensions [BBA10]. Weighting the individual schedulability results by task set utilization reflects the higher value placed on being able to schedule higher utilization task sets.

Figure 5.4 on the next page shows how the weighted schedulability measure for each schedulability test depends on task set energy utilization. The UTZ test ignores energy constraints and hence exhibits minimal variation due to the uniform distribution of Ue and U. The tests that consider energy ($LB1$, SIM, $UB2$, $UB1$) all show the same pattern of behaviour as the classical schedulability tests do against processor utilization, i.e. schedulability reduces at high levels of utilization (energy utilization in this case). We note that the performance of the simple sufficient test $UB1$ degrades with increasing energy utilization because in this case the overestimation of worst-case response times is greater.

Figure 5.5 on the following page shows the impact of task set composition. When the task sets comprise only gaining tasks, then all of the tests give precisely the same performance. This is because no energy replenishment is needed, and in this case all of the tests reduce to the exact test for fixed-priority preemptive scheduling with no energy constraints. Similarly, for task sets comprising only consuming tasks, the worst-case scenario is the synchronous release with the battery level set to the minimum [ACM13a].

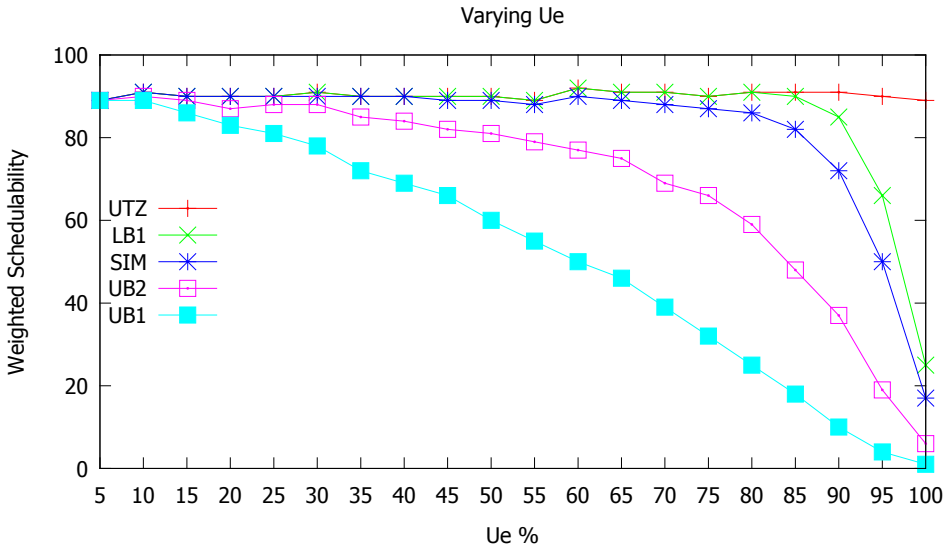


Figure 5.4: Varying the energy utilization

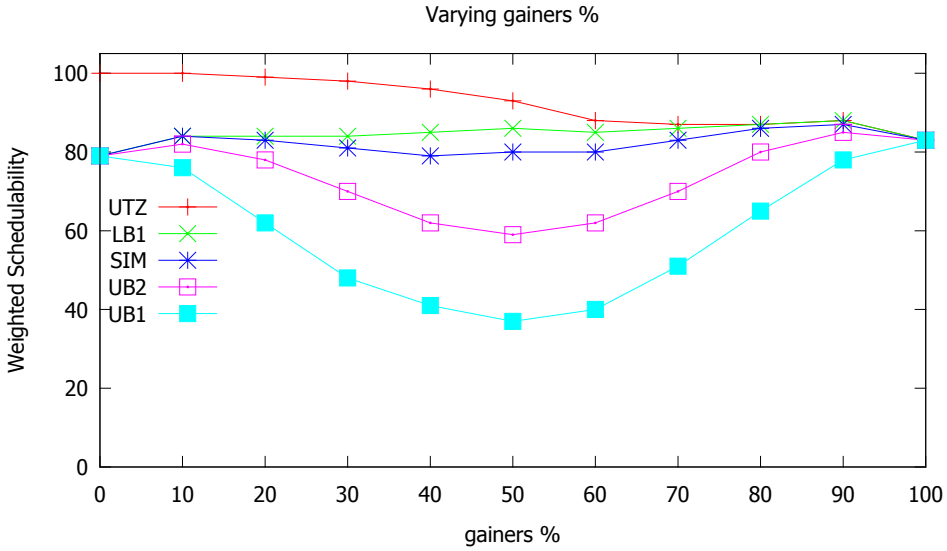


Figure 5.5: Varying the gaining tasks ratio

This is captured by all of the tests that consider energy ($LB1$, SIM, $UB2$, $UB1$), hence they all have the same performance.

Between these two extremes, the closer the task sets are to an equal mix of consuming and gaining tasks, the more opportunity there is for consuming tasks to make use of the net energy gain from gaining tasks, and hence the more $UB1$ and $UB2$ diverge from (SIM) and $LB1$. Here, $UB2$ is less impacted since it takes some account of the net energy gain due to gaining jobs that execute ahead of consuming jobs.

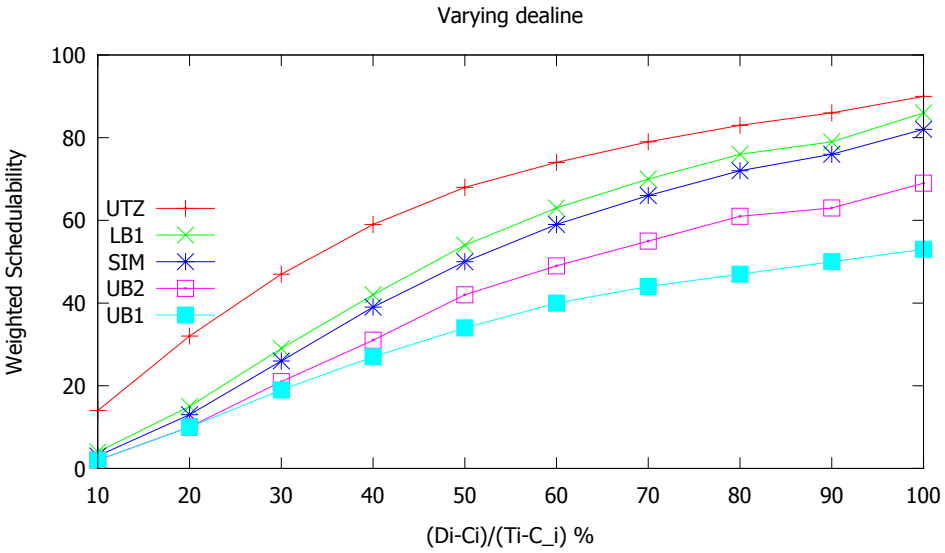


Figure 5.6: Varying relative deadlines in $[C_i, T_i]$

Figure 5.6 shows the impact of constrained deadlines on performance. In this experiment we vary the deadlines from heavily constrained where $D_i - C_i$ is 10% of $T_i - C_i$ to 100% of $T_i - C_i$ (i.e. implicit deadlines). We observe that all of the schedulability tests are influenced by the tightness of deadlines to a similar degree, with heavily constrained deadlines having significant impact on schedulability in all cases.

5.5 Conclusions

In this chapter, we addressed the problem of schedulability conditions in real-time energy-harvesting systems, where the respect of both time and energy constraints have to be guaranteed. In such systems, tasks can be classified as *gaining* or *consuming* tasks depending on whether or not the system has a net gain or loss of energy when the task executes. In Chapter 4 we showed that the energy work-conserving scheduling policy PFP_{ASAP} is optimal among all fixed-priority algorithms for the case where all tasks are consuming tasks.

The major contributions of this chapter are as follows. We showed that under PFP_{ASAP} scheduling algorithm, the critical instant (worst-case scenario) for task sets with both consuming and gaining tasks is not necessarily synchronous release with all other tasks. While we did not identify the specific worst-case scenario for this more general model, we were able to prove a number of properties that it must have. We used these properties to derive two upper bounds on task response times, thus forming two sufficient schedulability tests. In a similar way, we also derived a lower bound response time, and hence a necessary schedulability test. We proved that Deadline Monotonic is the optimal priority assignment policy for PFP_{ASAP} with respect to

our sufficient tests. Finally, we evaluated the performance of the sufficient tests in comparison with a number of necessary tests, including an exact test for fixed-priority preemptive scheduling ignoring energy constraints, and an empirical test based on simulating the schedule for more than a hyperperiod. We found that our tighter upper bound (sufficient schedulability test $UB2$) provides good performance over a wide range of values of different parameters e.g. energy utilization, proportion of gaining tasks etc. (explored using the weighted schedulability measure).

Chapter 6

Optimal Algorithm Investigation

Contents

6.1 Introduction

In the precedent chapters we saw that finding efficient scheduling algorithms for fixed-priority energy-harvesting systems is one of the challenges of this research area. In Chapter 4, we presented PFP_{ASAP} which is an optimal scheduling algorithm. Moreover, the optimality of this algorithm relies on two main assumptions: the considered task sets are energy-non-concrete (see Definition 3.4 on page 70), and all the tasks consume more energy than it is replenished. Unfortunately, removing one of these two assumptions leads PFP_{ASAP} to lose its optimality. This is due to the fact that without these assumptions, the worst-case scenario of PFP_{ASAP} is no longer the synchronous activation with the minimum battery capacity. Moreover, without these assumptions, the worst-case scenario is unknown up to now. There exist some counter examples that prove the non-optimality of PFP_{ASAP} (see Figures 6.2 on page 157 and 5.1 on page 134).

The challenge now is to understand why does PFP_{ASAP} lose its optimality and we try to study deeply the fixed-priority scheduling for energy-harvesting systems by attempting to build an optimal algorithm or otherwise to prove the nonexistence of such an algorithm.

In this chapter, we explore different intuitive ideas of scheduling algorithms and we explain why they are not optimal through counter examples. Then, we show the difficulty of finding an optimal algorithm or proving the nonexistence of such an algorithm with a reasonable complexity.

The remainder of this chapter is organized as follows. In section 6.2 we define and prove some properties of fixed-priority scheduling for energy-harvesting systems. After that, we explore in Section 6.3 different ideas of scheduling algorithms and we discuss the existence of an optimal algorithm. Finally, we conclude the chapter with Section 6.4.

6.2 Definitions and Notations

In order to facilitate the understanding of the next sections, we first redefine and prove some properties.

6.2.1 Model and Notations

In this chapter we consider the model described in Section 3.2 on page 67. Furthermore, in addition to the notations set in Section 1.2 on page 30, we consider the following ones.

- $s_{i,j}$: the starting time of job $J_{i,j}$,
- $a_i(t)$: the next activation time of task τ_i after time t,
- $c_i(t)$: the remaining execution time of the current job of task τ_i at time t. It is equal to 0 if the job is already finished,
- $e_i(t)$: the remaining energy cost of the current job of task τ_i at time t. It is equal to 0 if the job has finished its execution,
- $d_i(t)$: the absolute deadline of the current job of task τ_i at time t, it does not exist if the job is not yet activated,
- $s_i(t)$: the execution starting time of the current job of task τ_i at time t, it is undefined if the job is not yet activated.

6.2.2 Definitions

Definition 6.1 (Energy Demand)**.**
The energy demand of priority level-i of time interval $[t_1, t_2[$ denoted $We_i(t_1, t_2)$ is the amount of energy to be consumed by the execution of the jobs of priority levels $1, \ldots, i-1, i$ that are requested within interval $[t_1, t_2[$ or are pending at time t_1. It can be obtained by Equation 6.1.

$$We_i(t_1, t_2) = \sum_{j \leq i} e_j(t_1) + \left\lceil \frac{t_2 - a_j(t_1)}{T_j} \right\rceil \times E_j \tag{6.1}$$

■

The intuition behind Equation 6.1 on the facing page is derived from the notion of processor demand. It represents the sum of the cost of energy of all the jobs of priority equal or higher than i that are requested during the time interval $[t_1, t_2[$. The energy demand of time interval $[0, t[$ is just noted $We_i(t)$ and can be obtained by Equation 6.2.

$$We_i(t) = We_i(0,t) = \sum_{j \leq i} \left\lceil \frac{t - O_j}{T_j} \right\rceil \times E_j \tag{6.2}$$

Definition 6.2 (Energy Budget)**.**
The energy budget of the system during time interval $[t_1, t_2]$ denoted $Bu(t_1, t_2)$ is the amount of energy available until time t_2, i.e. the battery level at time t_1 plus the energy replenished during time interval $[t_1, t_2]$. It can be computed by Equation 6.3.

$$Bu(t_1, t_2) = E(t_1) + \int_{t_1}^{t_2} P_r(t) \ dt \tag{6.3}$$

■

Definition 6.3 (Energy Balance)**.**
The energy balance of a job $J_{i,j}$ at time t denoted $Ba_i(t)$ is the difference between the energy budget between time t and the deadline of $J_{i,j}$, and the energy demand of the same priority level and the same time interval. It can be obtained by Equation 6.4.

$$Ba_i(t) = Bu(t, d_{i,j}) - We_i(t, d_{i,j}) \tag{6.4}$$

■

We notice that if the time t does not coincide with a request time of task τ_i and the previous job has already finished its execution, we must include the execution of the lower priority jobs between time t and the next request time of priority level-i, i.e. $a_i(t)$, because they consume energy before time $a_i(t)$ and can change the energy balance at time $d_{i,j}$ as illustrated in Figure 6.1 on the following page. However, these lower priority executions units depend on the used scheduling algorithm, i.e. energy-work-conserving or not. This limitation is discussed in Section 6.3 on page 159.

6.2.3 Energy-Work-Conserving

Without taking into account energy constraints, the work-conserving property in real-time scheduling means that the scheduling algorithm does not add idle times when there is at least one job ready to execute. However, when we consider energy-harvesting constraints, scheduling algorithms may add necessary idle periods to replenish energy. Then, this notion is extended to energy-work-conserving in Definition 3.5 on page 70 to include replenishment time.

Furthermore, as described in Definition 3.3 on page 70, in the energy-harvesting context, a scheduling algorithm is considered as *energy-work-conserving* if it schedules

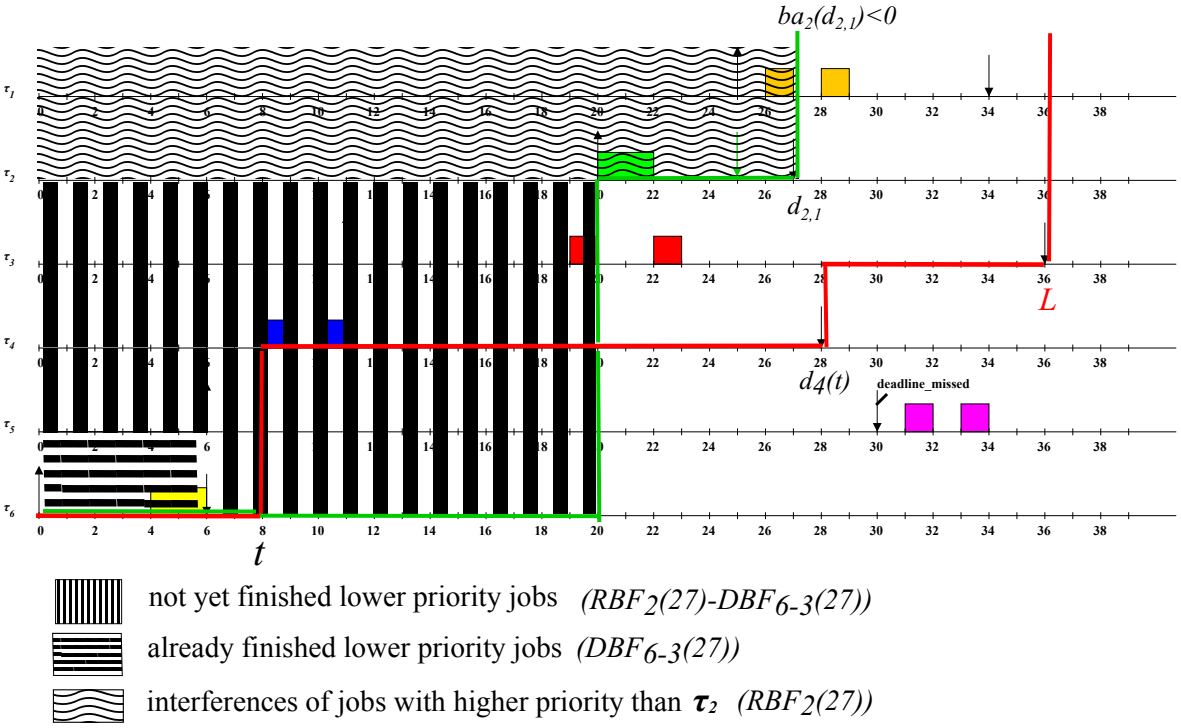


Figure 6.1: Energy balance of $J_{2,1}$ at time t=8

jobs as soon as they are ready to execute and the energy is sufficient to execute at least one time unit.

This definition means that scheduling algorithms do not replenish energy more than needed, otherwise, the algorithm is considered as non-energy-work-conserving.

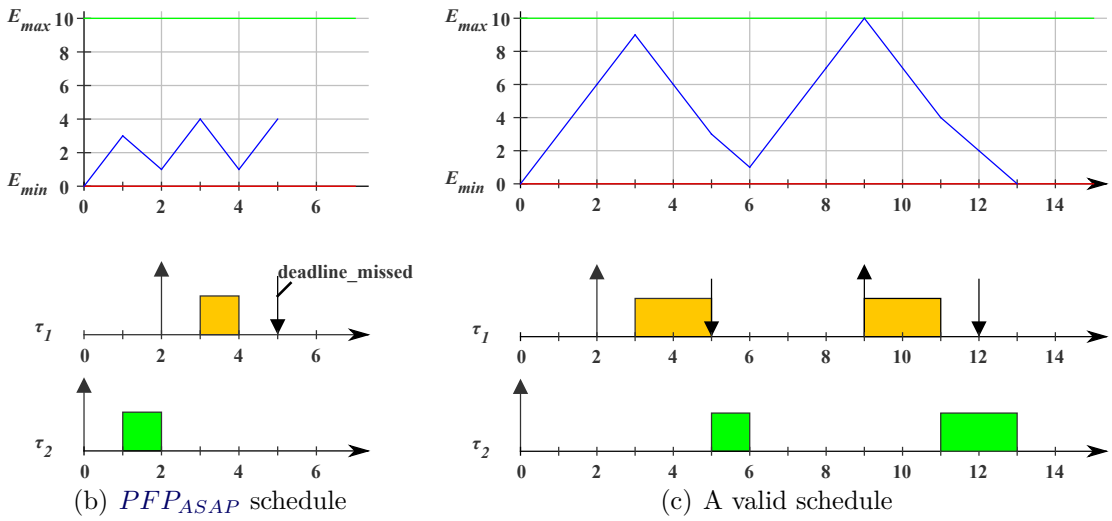
Unfortunately, PFP_{ASAP}, which is an energy-work-conserving algorithm, loses its optimality when we consider tasks with offsets or an initial battery level higher than E_{min}. Figure 6.2 on the facing page illustrates a counter example. We can see in Figure 6.2(b) on the next page that the lower priority task τ_2 is executed before τ_1 following the energy-work-conserving principle, i.e. as soon as the energy is sufficient to execute, it consumes the energy needed for the higher priority task τ_1 which needs more time than its deadline to replenish the required energy. We can see that in such a situation, executing as early as possible can lead to a deadline miss while delaying the execution of lower priority tasks can avoid missing deadlines as shown in Figure 6.2(c) on the facing page. Following this intuition, we propose Lemma 6.1.

Lemma 6.1.
The energy-work-conserving scheduling is not optimal for the scheduling problem of fixed-priority energy-harvesting systems.

Proof. To prove this property we have just to find an example where a valid energy-work-conserving schedule is not possible while a valid schedule exists.

-	O_i	C_i	E_i	T_i	D_i	P_i
τ_1	2	2	12	10	3	1
τ_2	0	3	15	15	15	2

(a) $E_{max} = 10$, $E_{min} = 0$ and $P_r = 3$



(b) PFP_{ASAP} schedule (c) A valid schedule

Figure 6.2: PFP_{ASAP} is not optimal

Let us consider a task set denoted Γ composed of two tasks τ_1 and τ_2 with the following configuration:

$$\begin{aligned}
&D_2 = 2 \times C_2 + C_1 \\
&D_1 = C_1 \\
&O_1 > O_2 \\
&C_1 + C_2 = O_1 + D_1 \\
&E_2 = \int_0^{O_1} P_r(t).dt \\
&E_1 + E_2 > E(0) + \int_0^{O_1+D_1} P_r(t).dt \\
&E_1 + E_2 \leq E(0) + \int_0^{D_2} P_r(t).dt \\
&T_1 = T_2
\end{aligned} \tag{6.5}$$

We can see that it is possible to finish executing τ_1 before O_2 according to an energy-work-conserving scheduling. However, it is not possible to schedule both τ_1 and τ_2 inside time interval $[0, O_1 + D_1]$ because the available energy is not sufficient (see Condition 6.5). Thus, more delay is needed to harvest more energy. Then, executing τ_1 before τ_2 and delaying τ_2 leads to miss the deadline of τ_2. Moreover, delaying τ_1 leads to add idle times while there is an active job and enough energy to execute immediately which violates the property of energy-work-conserving scheduling. In this case, it is impossible to produce a valid schedule with an energy-work-conserving scheduling. Then,

we prove that energy-work-conserving fixed-priority scheduling cannot be optimal. □

6.2.4 Energy-Lookahead Scheduling

In the classical real-time scheduling theory, a lookahead algorithm is an algorithm that is able to predict future job requests, for example in sporadic or periodic task models. However, in energy-harvesting model there is a new parameter subject to fluctuations: the incoming energy or the replenishment function. Therefore, we need to redefine the term of lookahead.

Definition 6.4 (Energy-Lookahead)**.**
A energy-lookahead scheduling algorithm attempts to foresee the effects of a scheduling decision to evaluate the schedulability of future jobs. The aim of lookahead is to chose the best scheduling decision that does not lead to avoidable deadline misses. The clairvoyance includes the battery replenishment function as well as tasks inter arrival times. In the opposite, if the algorithm does not consider the future state of the system, then, it is said non-energy-lookahead. ■

This means that the algorithm has knowledge a priori of the future state of the system, namely jobs activation times and energy replenishment function. The lookahead or clairvoyance consists of computations or scheduling simulation performed over a future interval of time that we call the *lookahead window*.

Definition 6.5 (Lookahead Window)**.**
The lookahead or clairvoyance window for a priority level-i at time t is the shortest time interval $[t, t + L[$ such that the scheduling decision of priority level-i at time t does not impact the scheduling decisions of time interval $[t + L, \infty)$. ■

Using this definition, we propose Conjecture 6.1.

Conjecture 6.1.
No non-energy-lookahead scheduling algorithm is optimal for fixed-priority energy-harvesting systems.

Insight. From Lemma 6.1 on page 156 we know that non-work-conserving scheduling is needed for an optimal algorithm. This means that additional delays are needed to ensure meeting deadlines and energy requirements. It is obvious that the optimal length of these delays depends on the available energy and the potential interferences, which means that the replenishment function and the request times of higher priority jobs are needed to compute the right delay for each job. Therefore, delaying execution without any knowledge about the future incoming energy and the future activation times of higher priority jobs may lead to too short or too long idle periods which can compromise the respect of deadlines.

-	O_i	C_i	E_i	T_i	D_i	P_i
τ_1	28	2	38	80	40	1
τ_2	7	2	32	16	8	2
τ_3	3	2	14	80	70	3
τ_3	0	1	12	68	44	4

Table 6.1: Task set Γ with $P_r = 3$, $E_{max} = 100$, $E_{min} = 0$ and $E_0 = 6$

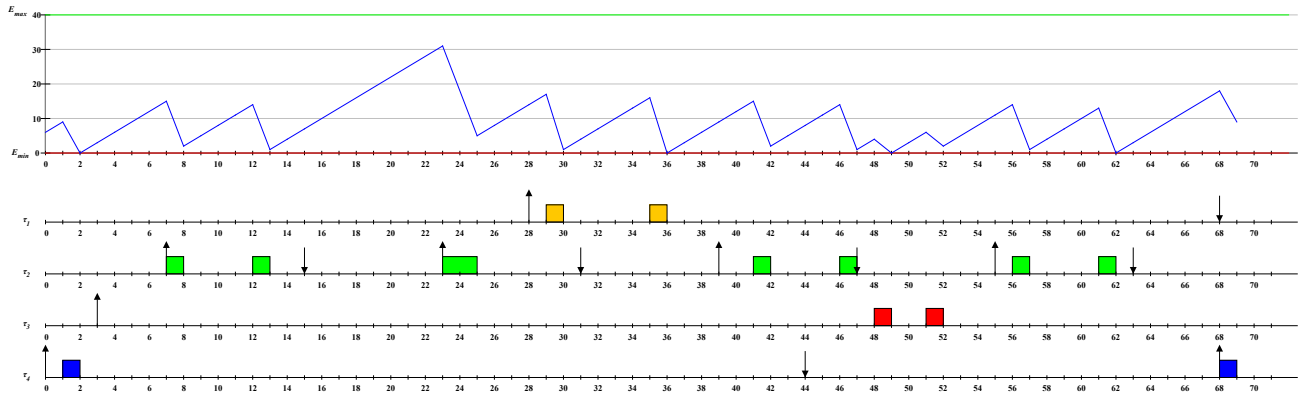


Figure 6.3: A feasible schedule of task set Γ

6.3 Algorithms

From Lemma 6.1 and Conjecture 6.1, we can consider that an optimal algorithm for fixed-priority energy-harvesting systems must be non-work-conserving and energy-lookahead. In this section we explore some scheduling algorithms and heuristics that attempt to be optimal. We start by showing a counter example that is feasible with fixed-priority scheduling but is not schedulable with all the fixed-priority algorithms presented until now in this dissertation. Then, we discuss the possibility of finding or building an optimal algorithm.

The task set described in Table 6.1 shows many situations that make the fixed-priority scheduling for energy-harvesting systems difficult. In the following we explain why each scheduling algorithm fails to schedule this task set in this configuration while a feasible schedule exists: Figure 6.3 shows the beginning of such a schedule, and we know that is feasible because there is no deadline miss within twice the hyper-period.

PFP_{ASAP}: the first intuitive scheduling algorithm for fixed-priority energy-harvesting systems is to use the classical FTP algorithm and add replenishment periods when E_{min} is reached. The PFP_{ASAP} policy behaves so. As described in Chapter 4, it schedules tasks as soon as possible when the energy is sufficient and replenishes otherwise. The replenishment periods are as long as needed to execute one time unit of the higher priority job ready to execute (see Algorithm 4.1 on page 112).

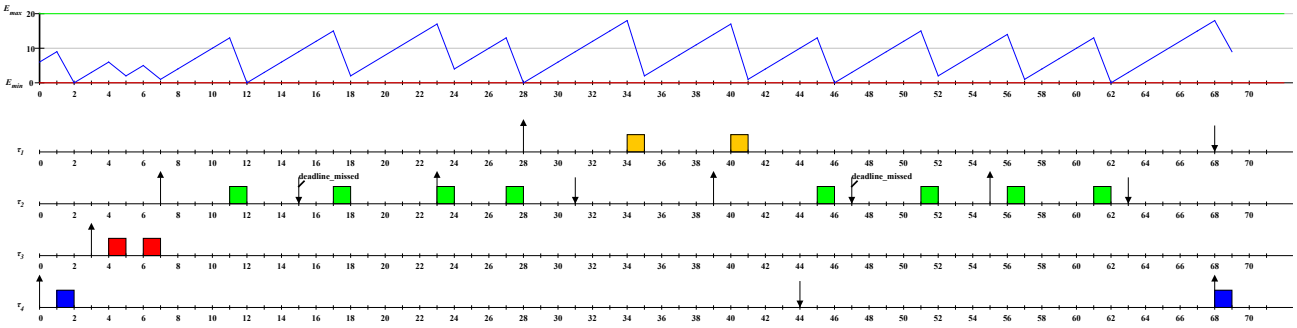


Figure 6.4: PFP_{ASAP} counter example

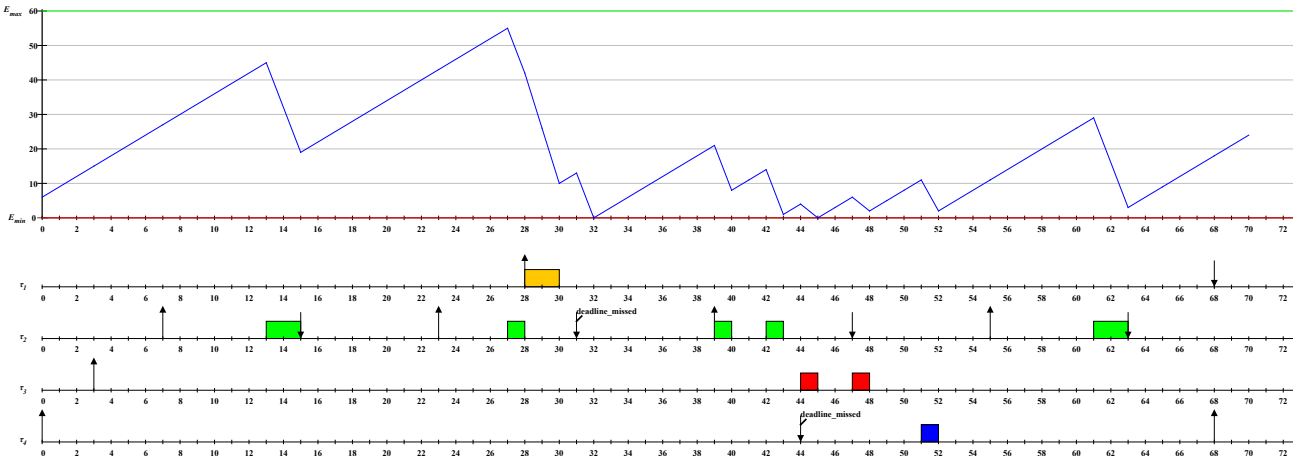


Figure 6.5: PFP_{ALAP} counter example

Again, PFP_{ASAP} is not optimal because executing as soon as possible maximizes the energy demand within a short interval of time which can lead to a lack of energy and then a deadline miss. As we can see in Figure 6.4, jobs of lower priority than $J_{2,1}$, namely $J_{3,1}$ and $J_{4,1}$, are executed at early as possible and consume the energy needed for a higher priority job that is requested few instant later, i.e. $J_{2,1}$. Job $J_{2,1}$ misses its deadline while the lower priority jobs, namely $J_{3,1}$ and $J_{4,1}$, can be delayed to avoid this situation.

PFP_{ALAP}**:** the second intuitive idea is to schedule jobs prior to their deadlines in order to permit a maximum replenishment of energy before executing. The PFP_{ALAP} algorithm was proposed based on this intuition. As described in Section 3.7.1 on page 92, it postpones jobs executions as long as possible all the time. Whenever there is available slack-time, executions are delayed (see Algorithm 3.5 on page 93).

Unfortunately, this is one of the counter intuitive ideas of fixed-priority scheduling for energy-harvesting systems. In fact, the PFP_{ALAP} algorithm is not optimal also

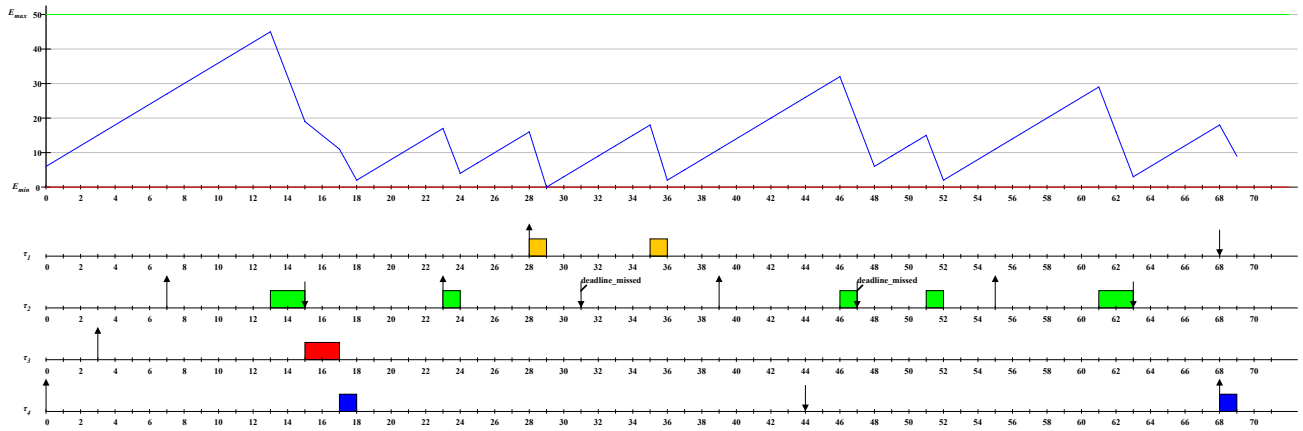


Figure 6.6: PFP_{ST} counter example

even with an unlimited battery capacity because the computation of slack-time does not consider energy constraints. As we can see in Figure 6.5 on the preceding page, a deadline miss can occur while a feasible schedule exists (Figure 6.3 on page 159). A deadline miss occurs at time 31 when the energy balance of time interval $[23, 31]$ is negative even though there is available slack-time. The energy available until the deadline of job $J_{2,2}$ is lesser than the energy demand of the same time interval. This negative energy balance is due to the fact that delaying job too much $J_{2,2}$ leads the system to anticipate the execution of the higher priority job $J_{1,1}$ which increases the energy demand of time interval $[23, 31]$ and leads to an insufficient energy to finish executing $J_{2,2}$ before its deadline. This phenomena is due to the fact that the slack-time computation used for this algorithm does not consider the energy requirements of the system.

PFP_{ST}: after PFP_{ASAP} and PFP_{ALAP} one can propose a hybrid algorithm that can behave sometimes as PFP_{ASAP} and sometimes as PFP_{ALAP}. The PFP_{ST} algorithm was built following this intuition. It executes jobs as soon as possible whenever the energy is sufficient to execute and replenishes otherwise. The replenishment periods are as long as the available slack-time (see Algorithm 3.6 on page 96).

Even though this algorithm improves the schedulability rate comparing to PFP_{ALAP}, it is still not optimal because of the same reasons than PFP_{ALAP} and PFP_{ASAP}. Figure 6.6 shows a counter example.

Fixed-Task-Priority Clairvoyant as soon as possible ($FPCasap$): the computation of slack-time in PFP_{ALAP} and PFP_{ST} algorithms is considered as time clairvoyance or time lookahead because it uses the arrival times of future jobs. However, the energy constraints were not considered, this is why the precedent algorithms fail to schedule some feasible task sets.

Thus, one can add energy clairvoyance to compute the right retardations that lead to

Algorithm 6.1 *FPCasap* Algorithm

```
1: t ← 0
2: loop
3:    A ← set of active jobs at time t
4:    if A ≠ ∅ then
5:       J_{i,j} ← the highest priority job of A
6:       d_i(t) ← the next absolute deadline of J_{i,j}
7:       if ResponseTime_{PFP_ASAP}(t + 1, J_{i,j}, E(t + 1)) > d_i(t) then
8:          execute J_{i,j} for one time unit at time t
9:       else
10:         if Clairvoyance_{PFP_ASAP}(t, J_{i,j}, d_i(t), E(t)) then
11:            execute J_{i,j} for one time unit at time t
12:         else
13:            suspend the system for one time unit
14:         end if
15:      end if
16:   end if
17:   t ← t + 1
18: end loop
```

a valid schedule. Following this idea, the As Soon As Possible Clairvoyant Fixed-Priority Algorithm (*FPCasap*) was proposed in [ACM13b]. Before authorizing a job to execute, it simulates the PFP_{ASAP} schedule of the current and the future jobs in a clairvoyance window or a looakahead window.

The *FPCasap* algorithm inherits the behavior of PFP_{ASAP} and adds clairvoyance capabilities. It schedules jobs as soon as possible whenever the two following conditions are met:

- there is enough energy available in the storage unit to execute at least one time unit,
- the execution of the current job does not lead to a deadline miss of jobs of higher priority which are requested during the clairvoyance window.

If these conditions are not satisfied, then, the algorithm suspends all executions for one time unit and then it tries again.

Algorithm 6.1 shows how *FPCasap* takes scheduling decisions at time t when a job of priority level-i is ready to be executed. The *FPCasap* algorithm checks first if the execution of jobs of priority level higher or equal than i meet theirs deadlines. Then, it checks if it is possible to delay the current job by comparing its response time at time $t + 1$ with its deadline (line 7). After that, it repeats the process for higher priority jobs. This prevent delaying the current job uselessly because in the case where it is impossible to delay, if a deadline miss occurs in a higher priority level in the clairvoyance

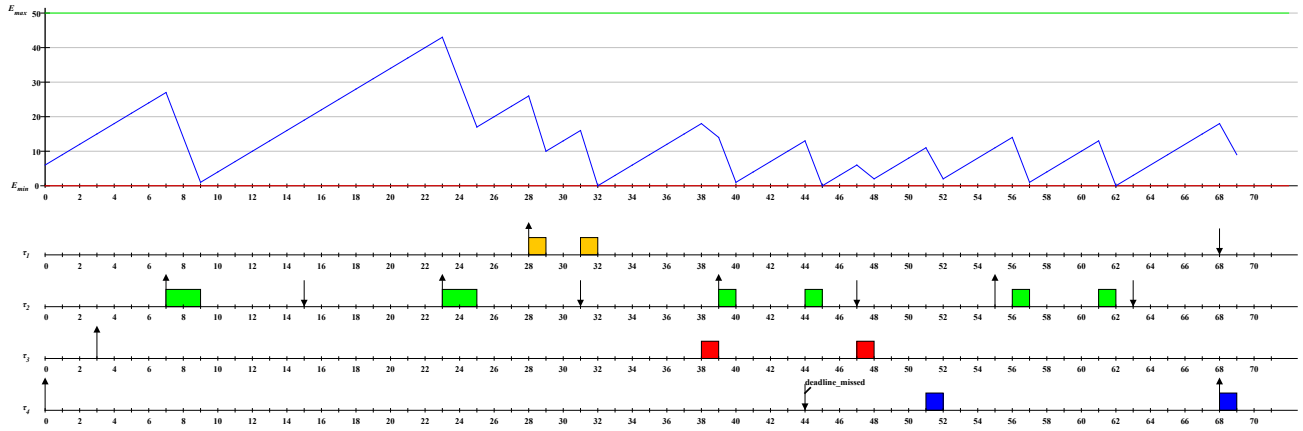


Figure 6.7: *FPCasap* counter example

window, the deadline miss cannot be avoided and the system is not schedulable with *FPCasap*. The length of the clairvoyance window for a job is not proved but can be intuitively defined as the interval of time starting from time t to the absolute deadline $d_i(t)$ because the current job of level-i cannot be delayed more than its deadline.

Unfortunately, *FPCasap* is not optimal because when a future deadline is detected with the clairvoyance algorithm, all the jobs are delayed until the deadline miss disappears and it is too much in certain cases. This delay is from the left to the right following the time increasing axis. When the energy balance is negative at the end of the clairvoyance window, delaying a lower priority job for an unbounded period can lead to a deadline miss when a higher priority job is also delayed to a later time than the deadline of the lower priority job. Computing the response time at time $t+1$ of the ready job according to PFP_{ASAP} algorithm is not sufficient because it does not reflect the real response time of the job. Figure 6.7 illustrates a counter example where such a situation occurs. We can see that at time 0 when job $J_{4,1}$ is ready to execute, the PFP_{ASAP} lookahead schedule detects a deadline miss in a higher priority level in the looakahead window, i.e. the deadline of job $J_{2,1}$ (see Figure 6.4 on page 160), then, it postpones execution for one time unit and then checks again at time 1, 2 and 3 and the same decision is taken. However, from time 3 to time 51, job $J_{4,1}$ cannot be executed because of the higher priority interferences and the required replenishments. This example teaches us that when the energy balance is negative, in this case in time interval $[0, 44]$, it is impossible to schedule all the jobs that are requested inside this interval. Thus, the only way is to delay some jobs out of this interval. Moreover, we can see that all the jobs cannot be pushed out of this interval, only the ones that are requested inside the interval and have deadlines outside the interval. We can see that *FPCasap* takes the wrong decision by delaying the lower priority jobs, in this case $J_{4,1}$, instead of higher priority jobs.

Algorithm 6.2 *FPLSA* Algorithm

```
1: t ← 0
2: while true do
3:    J_{i,j} the ready job with the higher priority at time t
4:    calculate s_{i,j}
5:    if t ≥ s_{i,j} or E(t) + P_r > E_max then
6:       execute job J_{i,j}
7:    else
8:       idle the system
9:    end if
10:   t ← t + 1
11: end while
```

Fixed-Task-Priority Lazy Scheduling Algorithm (*FPLSA*): one of the possible ways to find optimal fixed-priority algorithms is to try to adapt the behavior of some optimal algorithms for *EDF* scheduling. One can use the concept of the *LSA* presented in Section 3.6.3 on page 86. It consists of computing the latest time from which jobs can be executed continuously. This algorithm was proved to be optimal for task sets that consume energy with the same rate. To adapt *LSA* algorithm to fixed-priority scheduling, we assume that all tasks consume energy with the same rate, i.e. $\forall\ \tau_i,\ E_i/C_i = r$. Furthermore, we keep the same scheduling schemes and we use fixed-priority scheduling instead of *EDF* ones. Therefore, the algorithm becomes as described in Algorithm 6.2 and we call it *FPLSA*.

The latest job starting time denoted $s_{i,j}$ is computed by Equation 6.6.

$$\begin{cases} s_{i,j} = \max(s'_{i,j},\ s^*_{i,j}) \\ s^*_{i,j} = d_{i,j} - \dfrac{E(a_{i,j}) + g(a_{i,j},\ d_{i,j})}{r} \\ g(a_{i,j},\ s'_{i,j}) - \mathcal{C} = g(a_{i,j}, d_{i,j}) + (s'_{i,j} - d_{i,j}) \times r \end{cases} \tag{6.6}$$

Unfortunately, this algorithm is not optimal even if we consider the same consumption rate for all tasks. We can see in Figure 6.8 on the facing page that the computation of the starting time $s_{i,j}$ is not adapted to fixed-priority scheduling. In fact, $s_{i,j}$ is the latest time from which we can execute continuously until the deadline $d_{i,j}$. This works for *EDF* scheduling because in *EDF* all higher priority jobs have their request times and deadlines inside interval $[t, d_{i,j}]$. However, this is not the case in fixed-priority scheduling which can lead to too long delays. In Figure 6.8 on the next page, we can see that the delay computed at time 3 for job $J_{3,1}$ is much longer than the delay computed for job $J_{4,4}$ at time 0 which led this later to miss its deadline.

Fixed-Task-Priority As Late As Possible with energy guaranty (*FPLeg*): the counter examples of the precedent algorithms show that even with energy and time clairvoyance the above algorithms are still not optimal. In fact, the priorities of

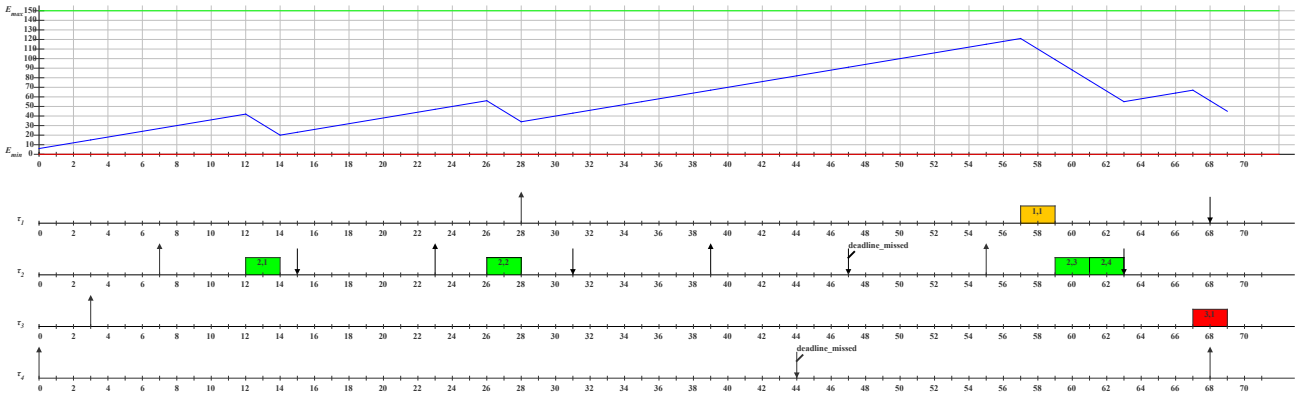


Figure 6.8: *FPLSA* counter example

tasks complicates the computation of the clairvoyance. The precedent examples show that in the case when the energy balance is negative in a certain interval of time, it is necessary to reduce the energy demand by pushing some jobs out of the interval. Then, the difficulty now is to find the subset of jobs to delay and calculate the lengths of their delays. We showed that delaying all the jobs at the maximum without considering energy like FPP_{ALAP}, delaying all the lower priority jobs when a future deadline miss is detected as *FPCasap* and delaying jobs to satisfy only energy constraints like *FPLSA* are not the right decisions to reduce safely the energy demand within a time interval. In fact, the potential jobs to be pushed out of the interval are the ones that are requested inside the interval and have their deadlines outside. These kind of jobs can be delayed by anticipating the execution of lower priority jobs. By doing so, the higher priority jobs to be pushed out need more replenishment time since the energy balance is negative. This leads to push them out of the interval and permits lower priority jobs to have more energy to execute which can help them to meet their deadlines.

The anticipation of lower priority jobs can be done by introducing a kind of virtual deadlines that coincide with request time of the jobs to be pushed out of the considered time interval. Then, once the virtual deadlines are set, we delay all the jobs at the maximum using the new virtual deadlines in the slack-time computation algorithm. Following this intuition we propose the *FPLeg* algorithm for Fixed-Priority as Late as possible with energy guaranty which is inspired from the *EDeg* algorithm presented in Section 3.6.2 on page 83.

The idea behind this algorithm is to use the same scheduling schemes as FPP_{ALAP} but by using virtual deadlines that consider energy constraints.

Definition 6.6 (Virtual deadline)**.**
The virtual deadline of a job $J_{i,j}$ denoted vd_{ij} is the earliest time that makes its energy balance positive. This time can be the effective deadline of the job or the request time of

Algorithm 6.3 *FPLeg* Algorithm

```
1: t ← 0
2: loop
3:    A ← set of active tasks at time t
4:    if A ≠ ∅ then
5:       τ_k ← the highest priority task of A
6:       if E(t) ≥ E_k/C_k and Slack.Time(t) ≤ 0 then
7:          execute τ_k for one time unit
8:       else
9:          replenish until time t + max(1, Slack.Time.with.virtual.deadlines(t))
10:      end if
11:   end if
12:   t ← t + 1
13: end loop
```

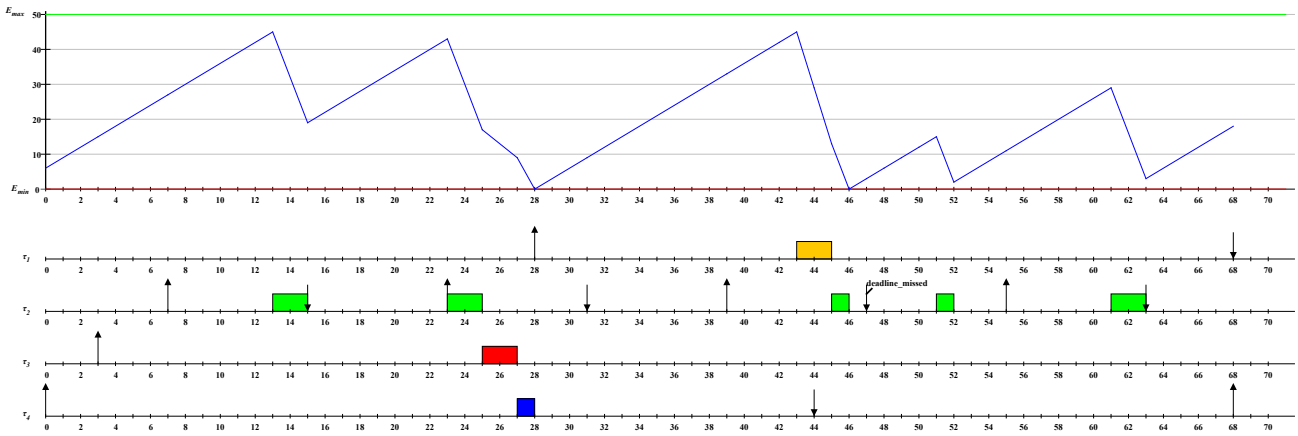


Figure 6.9: *FPLeg* counter example

one of the higher priority jobs described above. It must satisfy the following conditions:

$$\begin{cases} vd_{i,j} \leq d_{i,j} \\ Bu(t, vd_{i,j}) \geq 0 \end{cases}$$

■

By analyzing the counter example of PFP_{ALAP} shown by Figure 6.5 on page 160, we can see that if the energy balance was positive at time 31, the schedule would be valid and the slack-time time would be correctly calculated. Therefore, using virtual deadlines that makes the energy balance positive may be an interesting idea to build an optimal algorithm. Then, *FPLeg* behaves as described in Algorithm 6.3.

Figure 6.9 illustrates the scheduling schemes of *FPLeg* algorithm. We can see that the virtual deadline of job $J_{4,1}$ was shifted from time 44 to time 28 where the energy

balance is positive, i.e. the battery level at time 28 is 0, and then, the as late as possible schedule produced in time interval $[0, 28]$ is valid. However, we notice that this does not solve completely the problem of late scheduling. In fact, using virtual deadlines improves schedulability but there exist some cases where it anticipates the execution of some lower priority jobs that consume the energy needed for other higher priority jobs. This is illustrated by Figure 6.9 on the facing page at time 46. Even though the energy balance was positive at time 28, the missed deadline (time 47) is due to the anticipated execution of job $J_{3,1}$ which can be delayed further. Unfortunately, this proves that *FPLeg* is not optimal and that the fixed-priority scheduling for energy-harvesting systems is subject to many counter intuitive ideas.

Fixed-Task-Priority lookahead (*FPlh*): The use of virtual deadlines in *FPLeg* was a good idea because it helps reducing the energy demand when the energy balance is negative. However, the above counter example shows that having a positive energy balance is not sufficient to check whether the current virtual deadline is the right one or not. Combining the idea of lookahead of *FPCasap* with the idea of virtual deadlines can be an interesting intuition to achieve an optimal algorithm.

In the following we propose a new algorithm called *FPlh* for *Fixed-Priority lookahead* that combines the ideas of virtual deadlines and lookahead. The lookahead computation checks if all future deadlines are met during a bounded window by using the notion of virtual deadlines that guarantees positive energy balances. Therefore, the definition of the virtual deadline should take into account the future higher priority jobs. Moreover, the looakahead computation consists not only of calculating the energy balance of the current job $J_{i,j}$ but also the one of higher priority jobs that are included in the lookahead window. For this reason, we need to generalize the energy balance formula to compute the energy balance of any job at any moment.

Definition 6.7 (Energy Balance)**.**
The energy balance of priority level-i at time interval $[t_1, t_2[$ denoted $Ba_i(t_1, t_2)$ is the difference between the energy budget and the energy demand during $[t_1, t_2[$. It can be computed with Equation 6.7.

$$\begin{cases} A & = & \sum_{j>i} (DBF_j(0, a_i(t_1)) \times E_j \\ B & = & (RBF_j(0, a_i(t_1)) - DBF_j(0, -a_i(t_1))) \times (E_j - e_j(a_i(t_1))) \\ Ba_i(t_1, t_2) & = & Bu(t_1, t_2) - We_i(t_1, t_2) \\ & = & Bu(0, t_2) - We_i(0, t_2) - A + B \end{cases} \tag{6.7}$$

where :

- A: is the energy demand of lower priority jobs within time interval $[0, a_i(t_1)[$, i.e. finished jobs (see Figure 6.1 on page 156),

- B: is the energy demand of lower priority jobs that are probably not yet finished at time $a_i(t_1)$. To compute this value we need to simulate the execution with PFP_{ALAP} algorithm.

■

Now, we redefine the virtual deadline to include the lookahead computation.

Definition 6.8 (Virtual Deadline)**.**
The virtual deadline $vd_{i,j}$ of a job $J_{i,j}$ is an early deadline that makes the energy balance $Ba(t, vd_{i,j})$ positive and does not cause negative energy balances for jobs of higher priority tasks withing the lookahead window that ends at time L. It must respect the following conditions:

- $vd_{i,j} \leq d_{i,j}$
- $Ba_i(t, vd_{i,j}) \geq 0$
- $\forall\ J_{k,l} \in C,\ Ba_k(a_{k,l}, d_{k,l}) \geq 0$ where $C = \{J_{k,l},\ k < i\ and\ a_{k,l} > a_{i,j}\ and\ d_{k,l} \leq L\}$

■

The $FPlh$ algorithm is an extension of $FPLeg$ algorithm described above. It postpones executions whenever there is available slack-time like PFP_{ALAP} but the slack-time computation is done using the virtual deadlines. Note that the virtual deadline $vd_{i,j}$ coincides with a request time of a higher priority job that is activated before the deadline of $J_{i,j}$ and has an absolute deadline later than the one of $J_{i,j}$.

For a job $J_{i,j}$ at time t, we know that the higher priority jobs that have an absolute deadline earlier than $d_{i,j}$ cannot be delayed outside the time interval $[t, d_i(t)]$. Thus, the only jobs that can be pushed totally or partially out of this interval are those that have a deadline later than $d_{i,j}$. The virtual deadline of $J_{i,j}$ can be the request time of one of these jobs. To find the right virtual deadline, we test all of them starting with the earliest one. This test consists of checking two conditions:

- whether the energy balance of the considered job at this time is positive or not,
- whether there is future deadline misses within the lookahead window.

The length of the lookahead window is one of the problems of lookahead scheduling, it is at least bounded but should be specified carefully. For this algorithm, we consider that the lookahead window begins at time t and ends at time L which is the latest deadline of higher priority jobs that are requested before time $d_{i,j}$ and have their deadlines after $d_{i,j}$. We choose this length because the jobs to delay cannot be delayed longer than the length of the lookahead window.

The lookahead function consists of checking the energy balance of all the jobs that are requested within the lookahead window as described in Algorithm 6.5 on the next page.

Algorithm 6.4 *FPlh* Algorithm

```
1: t ← 0
2: loop
3:    A ← set of active tasks at time t
4:    if A ≠ ∅ then
5:       τ_k ← the highest priority task of A
6:       if E(t) − E_min ≥ E_k/C_k and Slack.Time.with.virtual.deadlines(t) ≤ 0 then
7:          execute τ_k for one time unit
8:          t ← t + 1
9:       else
10:         replenish until slack-time time unit
11:         t ← t + max(1, Slack.Time.with.virtual.deadlines(t))
12:      end if
13:   end if
14: end loop
```

Algorithm 6.5 *Lookahead*$(Ba_i(t), t, \tau_i, L)$ Algorithm

```
1: C = {J_{k,l}, k < i and a_{k,l} > t and d_{k,l} ≤ L}
2: for J_{k,l} ∈ C do
3:    if Ba_k(t, d_{k,l}) < 0 then
4:       return False
5:    end if
6: end for
7: return True
```

These rules allow us to be sure that the selected virtual deadlines prevent negative energy balances and future deadline misses, and lead to a correct energy-aware slack-time computation that can give the correct retardations. Figure 6.3 on page 159 shows the correct schedule of the task set that was not feasible with all the precedent algorithms. We can see that the virtual deadline of job $J_{4,1}$ was shifted to time 3 which makes the energy balance in time interval $[0, 3]$ positive and ensure that all higher priority jobs included in the lookahead window $[0, 70]$ meet their deadlines in contrast to *FPLeg*.

After all the precedent intuitions and counter intuitive ideas, we think that this algorithm is optimal or at least dominates than the other algorithms. However, its main issue is its complexity. In fact, the length of the lookahead window and the relationship between jobs deadlines are the main factors that increase the complexity of *FPlh*. Therefore, the following questions arise:

1. Does the use of virtual deadlines change the set of jobs to check with lookahead computations ?

   - Using earlier deadlines can decrease the number of jobs to check because the interval to study is shortened. For example, when a valid virtual deadline is

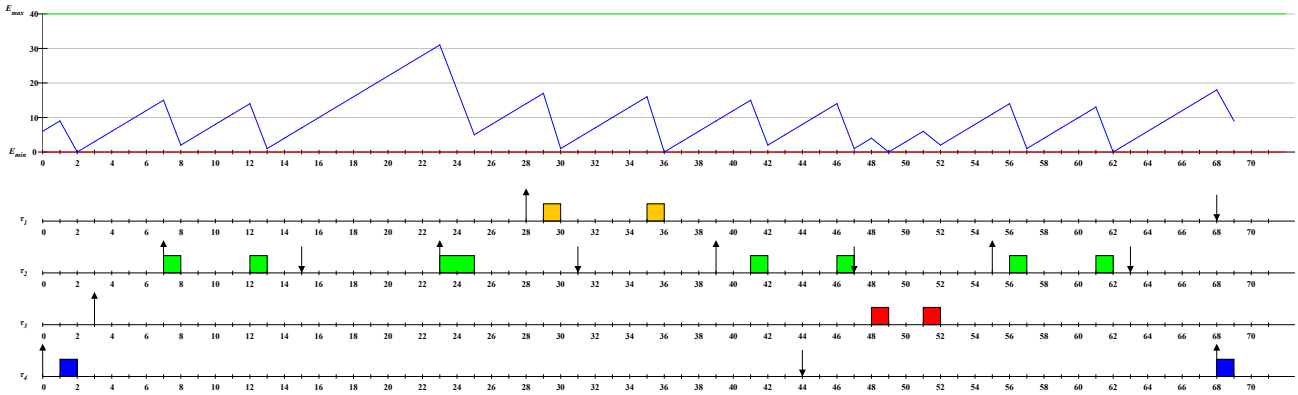


Figure 6.10: A *FPlh* schedule

found for the higher priority job with latest deadline, the lookahead window becomes shorter.

- Using real deadlines may lead to a negative energy balance for a higher priority job, as shown in the counter example of PFP_{ALAP}. Furthermore, computing virtual deadlines by only checking the energy balance of the current job can lead to a wrong virtual deadline as shown in the counter example of *FPLeg*.

- Then, the virtual deadline computation for a job of priority level-i needs to compute the energy balance $Ba_i(t, d_i(t))$ and the energy balances of higher priority jobs within the lookahead window as shown in Definition 6.8 on page 168. Moreover, since a part of the energy demand of lower priority jobs that have already began their execution before the request time of the job whose virtual deadline is being computed is necessary as shown with vertical lines pattern in Figure 6.1 on page 156. Their virtual deadlines are also needed to compute the energy balance of a higher priority job. This means that to compute the virtual deadline of one job, we need both virtual deadlines of higher and lower priority jobs which leads to a kind of cross dependency between the virtual deadlines of different priority levels. Figure 6.11 on the next page illustrates such a situation. We can see that to compute the virtual deadline of job $J_{4,1}$ at time 8, we need the one of $J_{2,1}$ because it is inside the lookahead window. Moreover, to compute the virtual deadline of job $J_{2,1}$ at time 20, we need the real schedule of jobs $J_{4,1}$ and $J_{5,1}$ which depend on their virtual deadlines. Therefore, there is an interdependence between virtual deadlines computation which makes calculating them one by one impossible. A combination of virtual deadlines is the set of virtual deadlines of the jobs that are included in the lookahead window. Since each job may have several potential virtual deadlines, finding the right one means finding the right virtual deadlines of all the other jobs.

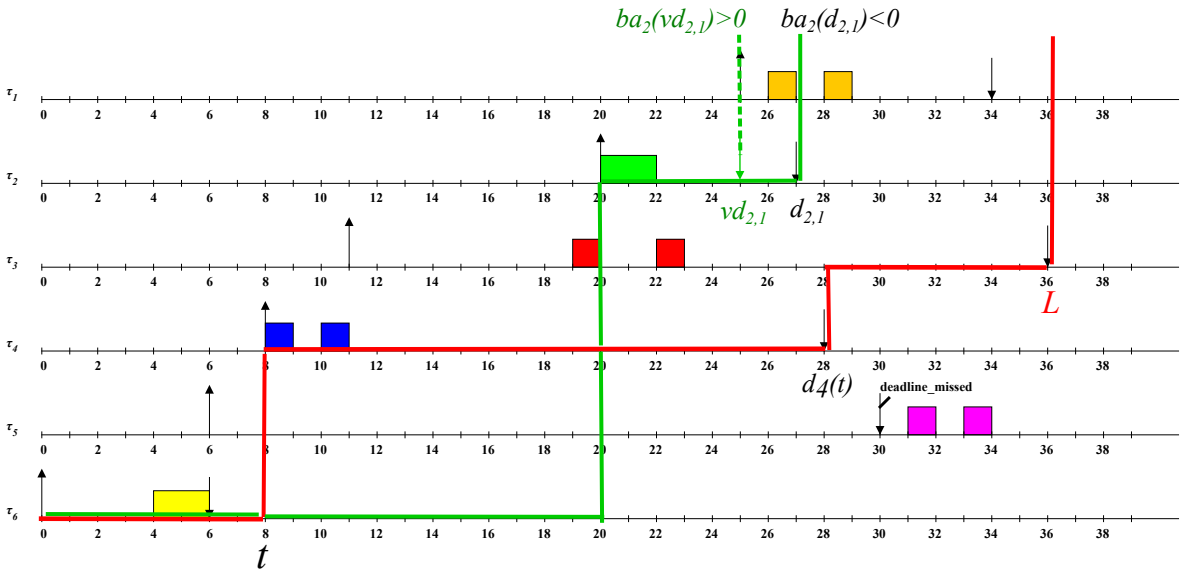


Figure 6.11: Interdependency between virtual deadlines

Then, to find such correct virtual deadlines, we have to pick one combination and test if it works. We do this until we find a correct combination. In the worst-case we have M^N combinations to test which is an exponential complexity, where M is the maximum number of potential virtual deadlines of a job and N is the maximum number of jobs that we can have inside a lookahead window.

2. Are the virtual deadlines necessary to compute the energy balance ?

   - The energy demand of higher priority levels is calculated only with requests times, i.e. with request bound function, thus, virtual deadlines of higher priority levels cannot change the final energy demand.
   - The energy demand of lower priority jobs that have already finished their execution is calculated only with deadlines, i.e. with demand bound function denoted A in Equation 6.4 on page 155. Then, the use of virtual deadlines of higher priority levels cannot change the final energy demand because they are earlier than the real deadlines.
   - The energy demand of lower priority jobs that have not yet finished, denoted B in Equation 6.7 on page 167 depends on the executions done before time $a_i(t_1)$ with PFP_{ALAP} policy which is based on deadlines. Using virtual deadlines for these jobs can change the final energy demand.

   Therefore, virtual deadlines of lower priority jobs are needed for the energy balance computation.

3. Assuming that computing virtual deadlines is an NP-hard problem, are they necessary for an optimal algorithm ?

    - If yes, the scheduling problem for fixed-priority energy-harvesting systems is also NP-Hard also,
    - Otherwise, it is an open problem.

These insights confirm that the fixed-priority scheduling for energy-harvesting systems is not easy to handle. Moreover, the only algorithm we have up to now that could be optimal has an exponential complexity. Therefore, finding an optimal algorithm with a reasonable complexity or proving that the fixed-priority problem scheduling for energy-harvesting systems is an NP-Hard problem is still an open problem.

6.4 Conclusion

In this chapter we explored the possibility of the existence of an optimal scheduling algorithm for energy-harvesting systems. We started by proving that such an algorithm must be both non-energy-work-conserving and lookahead (or clairvoyant). Then, we listed some ideas of scheduling algorithms that seem intuitively optimal and we showed with counter examples why they are not. We showed that the computation of the needed delays, i.e. replenishment periods, must consider jobs deadlines and energy cost as well as the replenished energy. Moreover, we showed also that deciding which job to delay is the main problem of building an optimal algorithm. We know now that considering a late or a lazy scheduling and having a positive energy balance at the deadline of a job is not sufficient. The maximum delay for a job must ensure not only a positive energy balance but also the respect of all deadlines in a bounded lookahead window. This idea is very interesting, however, the exact lookahead computation seems to be complicated and have an exponential complexity. This complexity seems to be difficult to reduce because the computation of the maximum delay of one job, i.e. the computation of the virtual deadline, depends on the maximum delay of the other jobs inside the lookahead window. There is a kind of cross dependency between jobs which complicates the computations. The only investigated solution here is to perform a brute force search for the right combination of jobs delays or virtual deadlines which has an exponential complexity. The conclusion of this chapter is that the correct late scheduling that respects the energy constraints needs earlier deadlines and that the computation of these deadlines has an exponential complexity. Moreover, if an optimal algorithm requires these deadlines, then, the scheduling problem of fixed-priority energy-harvesting systems is a hard problem. Otherwise, we should find an other way to compute the right delays, this is still an open problem.

Chapter 7

Response-Time Analysis for Thermal-Aware Scheduling

Contents

7.1 Introduction

The main purpose of real-time systems is to guarantee predictable timing behavior for controlled devices. Therefore, the correctness of the results provided by such systems depends not only on the logical correctness of the output but also on the time at which it is yielded. Several formal models of real-time behavior have been proposed up to now (e.g. task models such as sporadic, periodic, aperiodic, Directed Acyclic Graphs (DAG), etc.) as mentioned in Section 1.2 on page 30 and 3.2 on page 67. Prior research in real-time systems have also addressed a wide array of hardware architectures (e.g.

monoprocessor, multiprocessors, memory caches, etc). However, for a new generation of real-time systems applications, e.g. medical implants, the physical environment poses additional design challenges.

One such new challenge is the necessity of managing the energy and the thermal behavior of systems. As technology scales, chips power consumption and power density are increasing rapidly. Indeed, the miniaturization of small embedded systems has allowed new real-time applications. **I**mplantable **M**edical **D**evices (IMDs) are an example of these new embedded systems where managing the thermal aspect is essential. IMDs are increasingly being used in medical treatments (e.g. pacemakers for heart diseases or neural implants to restore hearing/vision). However, recent studies [Kim+07; Laz05] have shown that the heat generated by IMDs due to the processor activity is non-negligible. Thus, designing thermal aware IMDs becomes critical as medical research has shown that a temperature increase of even $1°C$ can damage tissues [LaM+89] and may cause death in extreme cases [Rug+03].

Therefore, thermal-aware real-time systems must respect not only timing constraints, expressed with deadlines, but also thermal constraints which are expressed as a maximum temperature not to be exceeded. For fixed-priority real-time scheduling on monoprocessor platforms, considering this constraint requires the schedulers to add cooling periods. This additional idle times must be taken into account by scheduling algorithms and included in schedulability analysis.

Thermal-aware system design presents challenges similar to the design of energy-harvesting systems. The later collects the environmental energy to store it and use it to supply real-time systems. The similarities with thermal-aware systems come from the fact that the scheduling for energy-harvesting system has to consider battery replenishment times which are analogous to the cooling periods required in thermal-aware systems. Intuitively, the real-time scheduling problem of energy-harvesting systems seems to be close to the one of thermal-aware systems. Therefore, it is interesting to explore the possible similarities between these two models.

As a first step, we apply some results of energy-harvesting systems to the thermal-aware model. In this chapter we use the **P**reemptive **F**ixed-Task-**P**riority **A**s **S**oon **A**s **P**ossible (PFP_{ASAP}) scheduling algorithm presented in Section 4.3 on page 111, which was proved to be optimal for energy-non-concrete fixed-priority energy-harvesting systems, to build a thermal-aware scheduling approach and a schedulability analysis based on upper and lower bounds of tasks worst-case response-time.

The work presented in this chapter is the result of a collaboration with Prof. Nathan Fisher from Wayne State University and mainly contains materials published in [CFM15].

The remainder of this chapter is organized as follows. Section 7.2 gives a brief state of the art about thermal-aware real-time systems. Section 7.3 specifies and describes the model and the scope of this chapter. Section 7.4 describes the thermal-aware version of PFP_{ASAP} scheduling algorithm and proves its optimality for thermal-non-concrete systems. Section 7.5 details an approximate response-time analysis based on upper and

lower bounds of tasks worst-case response time. Section 7.6 shows some simulation results to evaluate the effectiveness of the proposed schedulability analysis. Finally, Section 7.7 concludes this chapter.

7.2 Related Work

In this section we give a brief overview of prior research related to thermal-aware real-time scheduling. As for the energy-aware real-time systems, most works addressing this problem consider **D**ynamic **V**oltage and **F**requency **S**caling (DVFS) strategies. DVFS consists of scaling down the DVFS speed, and thereby lengthening task execution times as described in Section 3.4 on page 73, to reduce energy consumption and reduce the peak temperature [CWT09; WAB10; WB06; WB08].

Among existing works, the proposed techniques can be divided into *reactive* and *proactive* approaches. The difference between these two approaches is that reactive schemes adapt to the temperature of the system when it reaches the maximum temperature or a specific trigger by switching the **C**entral **P**rocessing **U**nit (CPU) speed or by changing scheduling decisions. In this scope, Wang et al. proposed a schedulability analysis for speed scaling scheme for frame-based task model in [WB08], and they completed this with a worst-case response time analysis for **F**irst **I**n **F**irst **O**ut (FIFO) and fixed-priority scheduling in [WB06; WB08]. In contrast, proactive approaches set the configuration of the system judiciously beforehand (CPU speed and scheduling decisions) so that the maximum temperature is never reached [CWT09; QZ09; Qua+08]. In this scope, Chen et al. proposed in [CHK07; CWT09] a proactive EDF-based scheduling approach that changes the processor speed proactively by requests issued by the scheduler.

There exists also some works that address this scheduling problem without DVFS schemes by considering processors with only one frequency. In this scope, Ahmed et al. proposed in [Ahm+11] a technique that computes proactively the length of execution and cooling intervals so that a certain temperature is never reached. This idea was extended in [Het+12] to support unpredictable ambient temperature fluctuations. Rehan et al. proposed in [RPK13; RPK14] a kind of thermal utilization of the system (using a fluid schedule) and leveraged it to obtain a necessary and sufficient conditions for systems thermal feasibility.

All the mentioned work have the following limitations:

1) Except for works cited in the previous paragraph [Ahm+11; RPK13; RPK14; Het+12], all the proposed solutions rely on speed scaling to manage energy and temperature. These approaches cannot be applied to systems without DVFS capabilities.

2) Most of the scheduling solutions proposed in the literature are EDF-based. Knowing that static fixed-priority scheduling is highly used in industry, it deserves more attention and effort.

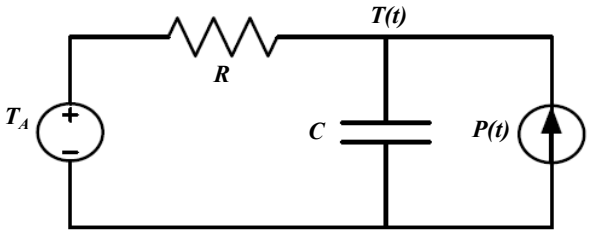


Figure 7.1: Thermal model

7.3 Models

7.3.1 Task Model

We consider a classical non-concrete real-time task set defined by a set of n sporadic and independent tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized by its priority P_i, its worst-case execution time C_i, its minimum inter-arrival time T_i, its deadline D_i and its first release time O_i. Deadlines are constrained or implicit, i.e. $\forall\, i,\ D_i \leq T_i$.

7.3.2 Thermal Model

In our model, the temperature of the system fluctuates due to heat dissipation when real-time tasks are executed on the CPU. The temperature must stay between two thresholds T_A and T_{max} where T_{max} is the maximum tolerated temperature and T_A is the ambient temperature. The temperature of the system at time t is denoted as $T(t)$. The only way to cool down the system is to temporarily suspend task execution. Furthermore, we consider that the system may be in one of two states at any given time: active (i.e. heating) during which tasks may execute, or inactive (i.e. cooling) during which tasks are not permitted to execute.

Heating Model

The thermal behavior of a processor can be modeled using the **R**esistance **C**apacitance circuit (RC) [Ska+04] shown in Figure 7.1. In this model, the heating is characterized by the electrical current denoted $P(t)$ passing through a thermal resistance R. The thermal capacitance is denoted $\mathcal{C}$. Using this model, the derivative of the system temperature function with respect to time can be calculated with *Fourier's law* [WAB10] given by Equation 7.1.

$$T'(t) = \frac{P(t)}{C} - \frac{T(t) - T_A}{R \times C} \tag{7.1}$$

The current passing through the resistance can be separated into two parts: the dynamic part $P_D(t)$ that evolves linearly with the processor frequency, denoted s,

and the part corresponding to the energy leakage $P_L(t)$ which is a function of the temperature.

$$P(t) = P_D(t) + P_L(t) \tag{7.2}$$
$$P_D(t) = \beta_0 s^\alpha \tag{7.3}$$
$$P_L(t) = \beta_1 T(t) + \beta_2 \tag{7.4}$$

Equations 7.2 to 7.4 give the formula to compute $P(t)$, where α, β_0, β_1 and β_2 are system specific constants [WAB10]. We consider only a monoprocessor with active/inactive modes; thus, during active periods, $P_D(t)$ is constant.

Let us denote $a = \frac{\beta_0 s^\alpha}{C}$, $b = \frac{1}{R \times C} - \frac{\beta_1}{C}$ and scale $T(t)$ to be $T(t) - \frac{R\beta_2 - T_A}{R\beta_1 - 1}$ to shift T_A to 0.

We can now recognize in Equation 7.1 on the facing page a classical linear differential equation:

$$T'(t) = a - b \times T(t) \tag{7.5}$$

Then, the solution is given by:

$$T(t) = \frac{a}{b} + \left(T(t_0) - \frac{a}{b}\right) \times e^{-b(t-t_0)} \tag{7.6}$$

The heating function only depends on time and constants and is not task specific. Recall that the parameters a and b are processor specific constants. Typical settings for these two variables are $b \approx 0.228$, and $a > 1$ with $\alpha \approx 3$ (see [Ahm+11]).

Cooling Model

During the cooling phases, the processor is inactive. We assume for simplicity that the frequency s is 0. However, this can easily be generalized to allow some fixed power dissipation during inactive phases. Then, $a = 0$ and the formula 7.6 becomes:

$$T(t) = T(t_0) \times e^{-b(t-t_0)} \tag{7.7}$$

Again, the cooling function only depends on time and is not task specific, i.e. all tasks consume energy and heat the system following the same pattern. Figure 7.2 on the following page shows the curves of cooling and heating functions. We can see that the cooling function slows down rapidly because of its reverse exponential part. This means that cooling for several short intervals is better for temperature and thereby for tasks response time than few and long ones.

Moreover, we assume that heating is greater than cooling. In other words, temperature cannot decrease while executing.

As we consider non-concrete task sets, we extend the definition to include the thermal aspect of the model.

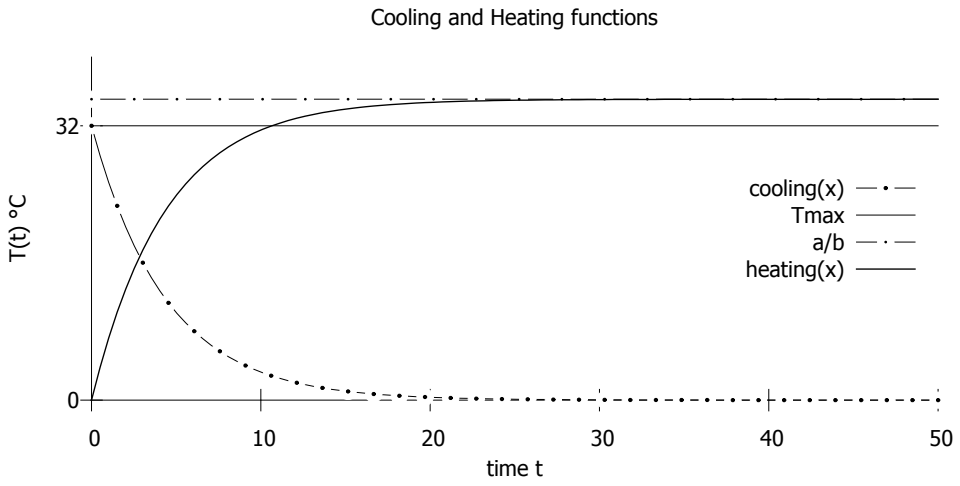


Figure 7.2: Cooling and heating functions

Definition 7.1 (Thermal-Concrete Systems)**.**
A thermal-concrete system is a system whose time and thermal parameters are known before run-time. This includes tasks periods, tasks offsets and the initial temperature of the system. In the opposite, if one or all of these parameters are known only at run-time, then, the system is said *thermal-non-concrete*. ■

Therefore, we consider a thermal-non-concrete system where the initial temperature of the system, the offsets and the exact inter-arrivals times are known only at run-time. In the considered model, the system has to respect all deadlines and thermal constraints. The temperature must never exceed its threshold T_{max}. Thus, a task set is feasible if and only if there is a schedule where all the deadlines are met and the maximum temperature T_{max} is never exceeded.

7.4 The PFP_{ASAP} algorithm

In Chapter 4, we presented the energy-harvesting version of PFP_{ASAP} scheduling algorithm. This algorithm is a fixed-priority one which takes into account tasks energy cost and the battery capacity during scheduling operations for energy-harvesting systems. Tasks are executed according to their priority whenever the available energy is enough to execute. Otherwise, the algorithm suspends executions to replenish the battery. These replenishment periods are as long as needed to execute at least one time unit. We proved that this algorithm is optimal for energy-non-concrete fixed-priority energy-harvesting systems. In this section we adapt this algorithm to thermal-aware systems and we explore its optimality for the model described in Section 7.3 on page 176.

With the thermal constraints, the behavior of PFP_{ASAP} becomes as follows: it executes jobs whenever the temperature is below T_{max} enough to execute at least one time unit without exceeding it, then, it idles the system to cool down as long as needed to resume executions.

Below, we will first address the PFP_{ASAP} worst-case scenario, then we will discuss its optimality.

7.4.1 Worst-case scenario

The aim of this section is to prove that the worst-case scenario for thermal-non-concrete fixed-priority thermal-aware systems is still the synchronous activation of tasks but with $T(0) = T_{max}$.

Figure 7.3(a) on the following page illustrates the case where all the tasks are requested simultaneously. If at least one higher priority task is requested later, the response time of lower priority tasks decreases as illustrated in Figure 7.3(c) on the next page. Then, if higher priority tasks are requested earlier, the response time of lower priority tasks cannot be larger than the one in the synchronous scenario as shown in Figures 7.3(d) on the following page. Furthermore, if the initial temperature of the system is less than T_{max}, then, less cooling time is needed which leads to a shorter response time for all tasks.

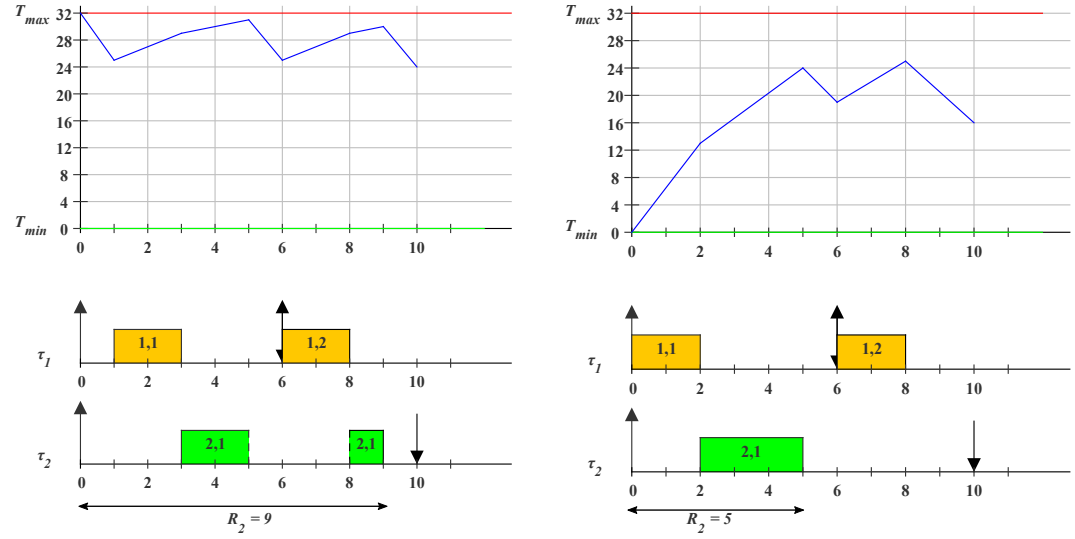
Theorem 7.1.
Let Γ denote a non-concrete task set composed of n priority-ordered tasks with constraint or implicit deadlines. The PFP_{ASAP} worst-case scenario for any task of Γ occurs whenever this task is requested simultaneously with requests of all higher priority tasks and the system temperature is at the maximum level T_{max}.

To prove this theorem we compare jobs response times in the scenario proposed by the theorem with all other possible ones. The response time of a job is the difference between its termination time and its offset or request time.

Proof. Let $\{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of n priority-ordered tasks where τ_n is the task with the lowest priority. Let S_i^s denote the scenario where task τ_i and all higher priority tasks are requested simultaneously at the highest temperature level T_{max}. The worst-case scenario for a task τ_i is the one that maximizes its response time, i.e. the scenario that maximizes the termination time of the first job of the i^{th} priority level.

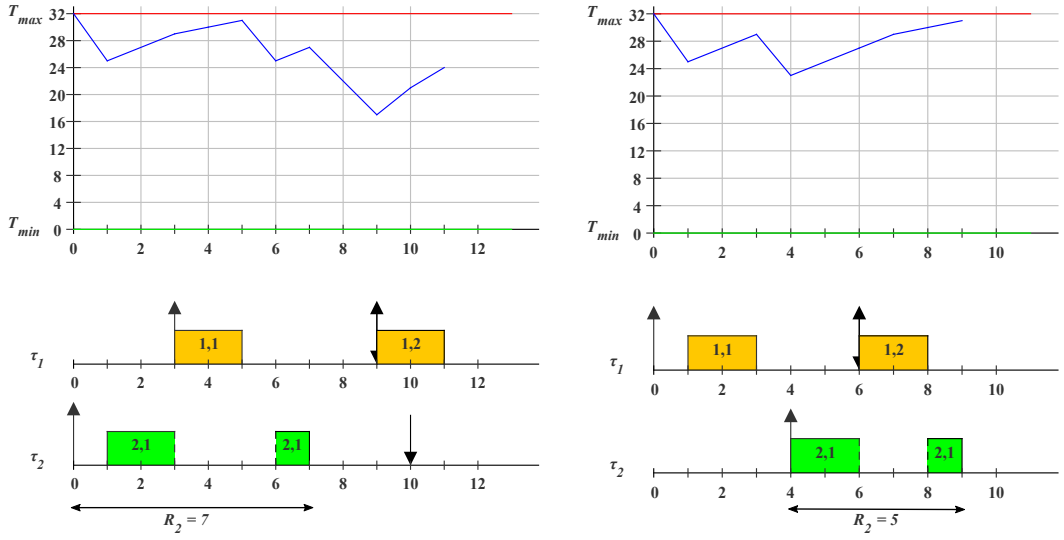
If S_i^s is not the worst scenario, there must exist an other one leading to a greater response time for the i^{th} priority level.

Firstly, we consider the scenario where $T(0) < T_{max}$. In this case the system is not heated at the maximum. Therefore, the system may need less cooling periods than the scenario where $T(0) = T_{max}$, and PFP_{ASAP} introduces shorter or equal cooling periods and leads to a shorter response time for all the tasks. This is in contradiction with our hypothesis, thus, such a scenario cannot lead to longer response times.



(a) Synchronous

(b) $T(0) < T_{max}$



(c) Negative shifting

(d) Positive shifting

Figure 7.3: Worst-case scenario

Secondly, we consider the scenario with different offsets. Let us denote S_i^a as the scenario where $T(0) = T_{max}$ and all tasks have different offsets. In this case we distinguish two possibilities:

1. At least one task of higher priority level than τ_i is requested later: knowing that all the tasks consume energy and heat the system following the same pattern, i.e. by considering that the heating comes only from processor energy consumption, and that heating is greater than cooling, task τ_i will undergo less higher priority interferences, and then, it may need less cooling to finish executing. Therefore, the final response time of τ_i is lesser than or equal to the one given by scenario S_i^s which is a contradiction. Thus, such a scenario cannot lead to longer response times.

2. At least one task of higher priority than τ_i is requested earlier: when τ_i is requested later than a higher priority task, it undergoes less interference from this task because, first, a part of it was executed before τ_i request time, and second, the increase of temperature due to the higher priority task execution cannot be higher than T_{max}, and finally, if τ_i is requested much later than the higher priority tasks, we just shift the landscape and will have case 1. Then, this scenario cannot be worse than S_i^s.

Therefore, in all possible situations, the response-time of a task τ_i is lesser or equal to the one led by a synchronous activation of all higher priority tasks when the temperature is at the maximum level.

□

7.4.2 The optimality of PFP_{ASAP}

We proved in Section 4.3.3 on page 118 that the PFP_{ASAP} algorithm is optimal for the fixed-priority scheduling problem of energy-non-concrete energy-harvesting systems which is close to the same scheduling problem of thermal-aware systems. In this subsection we extend the optimility of PFP_{ASAP} to thermal-non-concrete systems.

Theorem 7.2.
The PFP_{ASAP} scheduling algorithm is optimal for fixed-priority thermal-non-concrete task sets with constrained or implicit deadlines. As defined in Section 1.4.4 on page 37, this optimality means that if PFP_{ASAP} fails to schedule a given task set, then, no other fixed-priority algorithm can.

Proof. Let Γ denote a thermal-non-concrete task set. We suppose that Γ is feasible using a fixed-priority assignment, but not schedulable with PFP_{ASAP} using the same priority assignment. This means that at least one task denoted as τ_k misses its deadline in the worst-case scenario (see Theorem 7.1 on page 179). Indeed, it is sufficient to consider only the first job and an initial temperature $T(0) = T_{max}$ because we are

dealing only with constrained or implicit deadlines. The jobs of the same task cannot overlap unless for unfeasible tasks.

According to PFP_{ASAP} schemes, a deadline miss can occur in the worst-case scenario only in two cases: 1) the workload is greater than the available time, 2) the workload plus the accumulated cooling time is greater than the available time.

1) If the workload from the critical instant (time 0) to the first deadline of τ_k is alone greater than time interval $[0, D_i]$, then, it is obvious that it is impossible to schedule the first job of τ_k and higher priority jobs without missing D_k, this is not possible even without thermal constrains because the available time is not sufficient to schedule all the workload within $[0, D_k]$. Then, in this case the task set cannot be feasible with any algorithm and the supposed algorithm cannot exist.

2) If a deadline is missed with PFP_{ASAP} even though the workload is lesser than the available time, then, this means that the sum of workload of time interval $[0, D_k]$ and the needed cooling time is greater than the available time, i.e D_k time units. We know that the cooling periods produced by PFP_{ASAP} are as long as needed to execute one time unit which means that they are as short as possible. Furthermore, we know that the cooling function is exponentially decreasing (see Equation 7.7 on page 177), then, the shorter cooling periods are, the shorter the total needed cooling time is. This is true because the longer cooling is, the less efficient it is, as we mentioned in Section 7.3 on page 176. More formally, cooling down the system x times for one time unit is greater than the one of only one cooling period of length x time units because $e^{-bx} \leq xe^{-b}$ for all $x \geq 1$. Thus, Equation 7.7 on page 177) implies Inequality 7.8.

$$T(0) - T(0) \times e^{-b \times x} \geq T(0) - x \times T(0) \times e^{-b(1-0)} \tag{7.8}$$

Therefore, any other schedule than the one of PFP_{ASAP} has necessarily cooling periods of same length or longer, then, the response time of τ_k produced by the supposed algorithm is necessarily greater than D_k. Thus, in this case, no other algorithm can schedule this task set.

Then we prove that PFP_{ASAP} is optimal for non-concrete fixed-priority thermal-constrained task sets with constrained or implicit deadlines. □

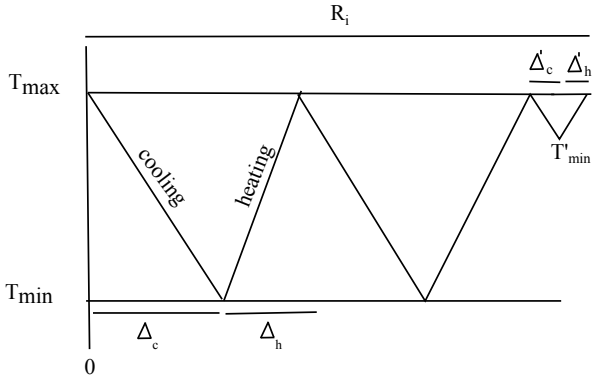


Figure 7.4: First upper bound ($UB_{T_{min}}$)

7.5 Response-Time Analysis

This section provides a response-time analysis for the schedule produced by the optimal algorithm PFP_{ASAP} in the worst-case scenario, i.e. the synchronous release of all the tasks when $T(0) = T_{max}$. We discuss the difficulty of an exact analysis and then we propose an approximate one.

7.5.1 Exact Analysis

The exact analysis provides the exact response time of all tasks. Thus, it must estimate accurately the length of all cooling and heating periods.

However, this cannot be easily achievable with a generic equation. Due to the discrete time, all cooling periods are not of the same length in the actual schedule. Furthermore, without the effective values of parameters, it is hard to estimate the order and the number of long and short cooling periods which have a significant impact on the response time value. Therefore, the only way to get an exact analysis is to simulate the schedule of PFP_{ASAP} in the worst-case scenario and compute the response time of the first job of each task.

7.5.2 Approximate Analysis

The aim of this work is to propose a schedulability analysis for thermal-aware real-time systems. Such an analysis must consider not only the processor workload but also the additional cooling time needed to respect the thermal constraints. To cope with the difficulty of providing an exact analysis, one can propose an approximate one that can be only sufficient instead of necessary and sufficient. This can be achieved by upper bounding tasks worst-case response time produced by PFP_{ASAP} algorithm.

First Upper bound ($UB_{T_{min}}$):

Knowing that PFP_{ASAP} cools down the system enough to execute at least one time unit, cooling periods are as short as possible. Furthermore, we know also that the cooling function is exponentially decreasing and the heating one is asymptotically increasing. Then, one can lengthen jobs response times by putting the cooling units together and the heating ones together such that T_{max} is never exceeded. By doing so, the cooling slows down after a while and the system needs more time to cool down. Similarly, the heating accelerates which heats up the system in a shorter amount of time.

Description

The upper bound of task τ_i worst-case response time according to $UB_{T_{min}}$ is described by Figure 7.4 on the previous page. It consists of:

- cooling down the system from T_{max} to T_{min}, where $T_{min} > T_A$,[1]
- a ceiling function applied to the time from T_{max} to T_{min} to ensure an integer number of time units (this is safe since it only overestimates the time required to reach T_{min}),
- executing jobs and heating up the system until T_{max} is reached or there is no pending workload,
- repeating cooling-heating cycles until there is no pending workload,
- the last cycle may be shorter because of the remaining workload which can be shorter than a full cycle. The corresponding cooling time is adjusted.

The upper bound of the worst-case response time of task τ_i of priority level-i that is requested simultaneously with higher priority tasks with $T(0) = T_{max}$, denoted $R_i^{T_{min}}$, is given by Equation 7.9

$$\begin{cases} w_i^{n+1} & = & N(w_i^n) \times (\Delta_c + \Delta_h) + \Delta'_c + \Delta'_h \\ R_i^{UB_{T_{min}}} & = & \{min_{t>0} \backslash\ w_i^{n+1} = w_i^n\} \end{cases} \tag{7.9}$$

where:

- $N(w)$ is the number of full cooling-heating cycles needed to execute the workload w without exceeding T_{max}.

$$N(w) = \left\lfloor \frac{w}{\Delta_h} \right\rfloor \tag{7.10}$$

[1] Observe that a low T_{min} value may result in an extremely pessimistic upper bound due to the nature of the cooling function; It decreases asymptotically to T_A, so waiting until T_{min} is too pessimistic because of the nature of the cooling function.

- w is the workload of time interval $[0, w_i^n[$.

$$w = \sum_{j \leq i} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times C_j \tag{7.11}$$

- Δ_h is the time to execute jobs and heat up the system from T_{min} to T_{max}. (Obtained by solving Equation 7.6 on page 177).

$$\Delta_h = \left\lfloor \frac{\ln\left(\frac{b \times T_{min} - a}{b \times T_{max} - a}\right)}{b} \right\rfloor \tag{7.12}$$

- Δ_c is the time to cool down the system from T_{max} to T_{min}. (Obtained by solving Equation 7.7 on page 177).

$$\Delta_c = \left\lceil \frac{\ln(T_{max}) - \ln(T_{min})}{b} \right\rceil \tag{7.13}$$

- Δ'_h is the remaining execution time of the busy period.

$$\Delta'_h = w - N(w_i^n) \times \Delta_h \tag{7.14}$$

- Δ'_c is the cooling time needed for Δ'_h.

$$\Delta'_c = \left\lceil \frac{\ln(T_{max}) - \ln(T'_{min})}{b} \right\rceil \tag{7.15}$$

- T'_{min} is the maximum temperature needed to execute the remaining part of the workload Δ'_h without exceeding T_{max}.

$$T'_{min} = (T_{max} - a/b)e^{b\Delta'_h} + a/b \tag{7.16}$$

Theorem 7.3.
An upper bound on the worst-case response time for task τ_i in the worst-case scenario described in Section 7.4.1 on page 179 can be obtained from a sequence of execution units of higher priority jobs and the necessary cooling time units where the cooling periods are as long as needed to cool down the system from T_{max} to T_{min} ($T_{min} > 0$) and the heating periods are as long as needed to reach T_{max} starting from T_{min}, as described by Formula 7.9 on the facing page.

Proof. In the computation of $UB_{T_{min}}$, the cooling periods are as long as needed to cool down the system from T_{max} to T_{min} and the execution periods are as long as needed to heat up the system from T_{min} to T_{max}. Furthermore, we know that in the actual schedule produced by PFP_{ASAP} algorithm for task τ_i in the critical instant, a cooling

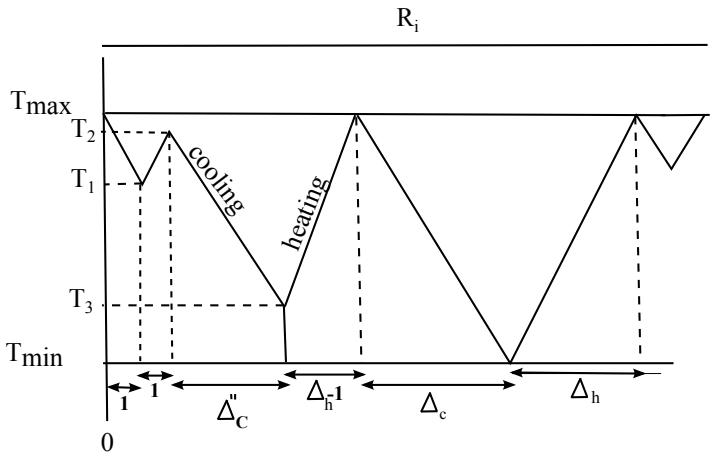


Figure 7.5: $UB_{T_{min}}$ proof insight

period is as long as needed to execute at least one time unit. The length of this period is shorter or equal than a cooling period of $UB_{T_{min}}$, and thus, by repeating these short cooling/heating cycles, it cools down the system faster than fewer and longer cooling periods. Then, to prove Theorem 7.3 on the previous page, we suppose that the actual schedule has at least one short cycle and we compare the two response times by using Fourier's law described by Equation 7.1 on page 176.

As described in Figure 7.5, we suppose that the system cools down for one time unit from T_{max} to T_1 (see Equation 7.17), and then heats up for one time unit reaching temperature T_2 (see Equation 7.17). After that, we follow the same schedule as $UB_{T_{min}}$ by finishing the cooling period needed to finish the workload $\Delta_h - 1$ and reach T_{max} (see Equation 7.18). The temperature of the system after this cooling period is denoted T_3 (see Equation 7.17).

$$\begin{cases} T_1 = T_{max} \times e^{-b} \\ T_2 = \dfrac{a}{b} + \left(T_1 - \dfrac{a}{b}\right) e^{-b} \\ T_{max} = \dfrac{a}{b} + \left(T_3 - \dfrac{a}{b}\right) e^{-b \cdot (\Delta_h - 1)} \\ T_3 = \dfrac{a + (b \times T_{min} - a) \times e^{-b}}{b} \\ T_3 = T_2 \times e^{-b \times \Delta''_c} \\ \Delta''_c = \left\lceil \dfrac{\ln(T_2) - \ln(T_3)}{b} \right\rceil \end{cases} \tag{7.17}$$

$$\Delta''_c = \left\lceil \frac{\ln\left(\dfrac{a + \left(b \times T_{max} \times e^{-b} - a\right) \times e^{-b}}{a + (b \times T_{min} - a) \times e^{-b}}\right)}{b} \right\rceil \tag{7.18}$$

Let us now compare the response time given by this assumption and the one given by $UB_{T_{min}}$. Knowing that the first cooling period, i.e. the first period of of time elapsed between T_{max} and T_1, is the same for both cases, and supposing that after reaching T_{max} at the end of the first heating cycle, i.e. from temperature T_3 to T_{max}, the actual schedule follows the same pattern as $UB_{T_{min}}$ cycles (see Figure 7.5 on the facing page). If $UB_{T_{min}}$ does not upper bound the actual response time, then the total cooling time needed for Δ_h workload in the actual schedule is greater than the one of $UB_{T_{min}}$, i.e. $\Delta_c'' + 1 > \Delta_c$. Knowing that Δ_c can be written as Equation 7.19

$$\begin{aligned} \Delta_c &= 1 + \left\lceil \frac{\ln(T_1) - \ln(T_{min})}{b} \right\rceil \\ &= 1 + \left\lceil \frac{\ln\left(\frac{T_{max} \times e^{-b}}{T_{min}}\right)}{b} \right\rceil \end{aligned} \tag{7.19}$$

Therefore, if $\Delta_c'' + 1 > \Delta_c$, then,

$$\Delta_c'' + 1 > \Delta_c \Rightarrow \left\lceil \frac{\ln\left(\frac{a + \left(b \times T_{max} \times e^{-b} - a\right) \times e^{-b}}{a + (b \times T_{min} - a) \times e^{-b}}\right)}{b} \right\rceil > \left\lceil \frac{\ln\left(\frac{T_{max} \times e^{-b}}{T_{min}}\right)}{b} \right\rceil$$

$$\Rightarrow \ln\left(\frac{a + \left(b \times T_{max} \times e^{-b} - a\right) \times e^{-b}}{a + (b \times T_{min} - a) \times e^{-b}}\right) > \ln\left(\frac{T_{max} \times e^{-b}}{T_{min}}\right)$$

$$\Rightarrow \frac{a + \left(b \times T_{max} \times e^{-b} - a\right) \times e^{-b}}{a + (b \times T_{min} - a) \times e^{-b}} > \frac{T_{max} \times e^{-b}}{T_{min}}$$

$$\Rightarrow \frac{a + (b \times T_{min} - a) \times e^{-b}}{a + \left(b \times T_{max} \times e^{-b} - a\right) \times e^{-b}} < \frac{T_{min}}{T_{max} \times e^{-b}}$$

$$\Rightarrow 1 - \frac{b \times e^{-b}(T_{max}\dot{e}^{-b} - T_{min})}{a + \left(b \times T_{max} \times e^{-b} - a\right) \times e^{-b}} < 1 - \frac{(T_{max} \times e^{-b} - T_{min})}{T_{max} \times e^{-b}}$$

$$\Rightarrow b \times e^{-2b} \times T_{max} > a + (b \times T_{max} \times e^{-b} - a) \times e^{-b}$$

$$\Rightarrow b \times e^{-2b} \times T_{max} - b \times e^{-2b} \times T_{max} > a(1 - e^{-b})$$

$$\Rightarrow e^{-b} > 1 \textit{ which is impossible because } b > 0 \text{ !}$$

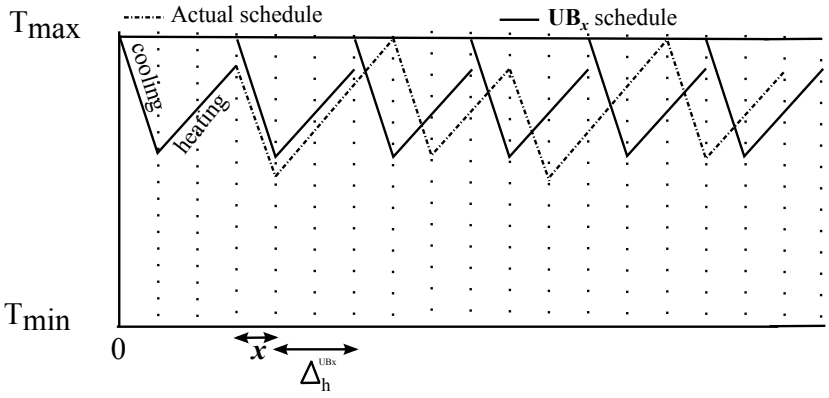


Figure 7.6: Parametric upper bound UB_x

Therefore, we prove by contradiction that $\Delta_c'' + 1 \leq \Delta_c$, and then $UB_{T_{min}}$ upper bounds the actual worst-case response time. □

Parametric Upper bound (UB_x):

Description

The idea of this upper bound is to keep the same behavior of the PFP_{ASAP} algorithm by cooling down for some time units and then executing jobs until reaching T_{max}. The approximation comes from the fact that time is discrete and that cooling periods are of a fixed length x (where $x \in \mathbb{N}^*$) instead of the minimum length needed to execute at least one time unit. Then, executing or the heating periods may not reach T_{max} in an integer number of time units. Thus, we consider only the integer part of heating periods (with floor function) and that T_{max} is exactly or nearly reached at the end of each heating period which adds additional cooling time than the actual schedule. Figure 7.6 describes the scenario used to obtain UB_x. It consists of:

- Cooling down the system for x time units, where x is a positive integer that must be greater or equal to $\Delta_c^{UB_x}$, the minimum time needed to decrease the temperature such that the system can execute at least one time unit, i.e. $x \geq \Delta_c^{UB_x}$. Equation 7.20 computes $\Delta_c^{UB_x}$; the ceiling function is used to respect the discrete time and to ensure that the system is cold enough to execute at least one time unit.

$$\Delta_c^{UB_x} = \left\lceil \frac{\ln\left(\frac{bT_{max}}{(bT_{max} - a)e^b + a}\right)}{b} \right\rceil \tag{7.20}$$

- Then, executing jobs and heating up the system until T_{max} is reached (without exceeding) or there is no pending workload. The length of this period is an integer.

The floor function is used to ensure not exceeding T_{max}.

- Repeating the cooling-heating cycles until there is no pending workload.
- The length of the last cooling period is still the same even if the remaining workload is smaller.

The upper bound according to UB_x of task τ_i of priority level-i that is requested simultaneously with higher priority tasks with $T(0) = T_{max}$ is given by Equation 7.21.

$$\begin{cases} w_i^{n+1} &= N(w_i^n) \times x + w \\ R_i^{UB_x} &= w_i^{n+1} = w_i^n \end{cases} \tag{7.21}$$

where :

- $N(w)$ is the number of cooling periods needed to execute the workload w without exceeding T_{max}:

$$N(w) = \left\lceil \frac{w}{\Delta_h^{UB_x}} \right\rceil$$

- w is the workload of time interval $[0, w_i^n[$:

$$w = \sum_{j \leq i} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times C_j$$

- $\Delta_h^{UB_x}$ is the time to execute jobs and heat up the system from the temperature reached after x time units of cooling to T_{max}:

$$\Delta_h^{UB_x} = \left\lfloor \frac{\ln\left(\frac{b \times T_{max} \times e^{-b \times x} - a}{b \times T_{max} - a}\right)}{b} \right\rfloor \tag{7.22}$$

We choose cooling periods longer or equal to $\Delta_c^{UB_x}$, i.e. $x \geq \Delta_c^{UB_x}$, because it is sufficient to execute at least one time unit without exceeding T_{max} which is close the behavior of PFP_{ASAP} algorithm.

To prove that UB_x upper bounds the actual response time, we first check the case where $x = \Delta_c^{UB_x}$. We know that in the actual schedule, the accumulation of the temperature gained at the end of each heating period due to the discrete time, is lesser than T_{max}. We denote this difference of temperature δ.Thus, this accumulated temperature can be used at least by one heating period which is supposed to be longer as shown in Figure 7.7 on the next page. Then, we compare the length of the new heating period Δ'_h (see Equation 7.23 on the following page) to the one of UB_x, i.e. $\Delta_h^{UB_x}$, and the total number of cooling/heating cycles produced by UB_x, i.e. $N^{UB_x}(w)$, and the one produced by the actual schedule denoted $N'(w)$.

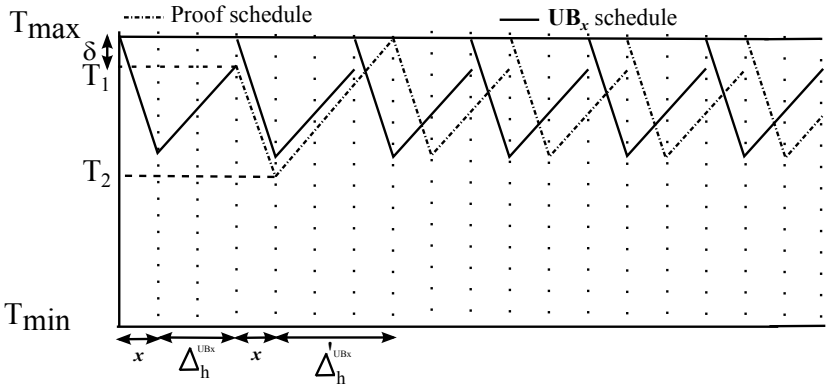


Figure 7.7: UB_x proof insight

$$\begin{cases} T_{max} &= (a + (b \times T_2 - a)e^{-b \times \Delta'_h})/b \\ T_2 &= (T_{max} - \delta)e^{-b \times x} \end{cases}$$

$$\Delta'_h = \left\lfloor \frac{\ln\left(\frac{b \times (T_{max} - \delta) \times e^{-b \times x} - a}{b \times T_{max} - a}\right)}{b} \right\rfloor \tag{7.23}$$

Lemma 7.1.
For $x = \Delta_c^{UB_x}$, each heating interval of the actual schedule Δ'_h given by Equation 7.23 is greater than or equal to UB_x's ones, i.e. $\Delta_h^{UB_x} \leq \Delta'_h$.

Proof. Let us suppose that $\Delta_h^{UB_x} > \Delta'_h$, then:

$$\Delta_h^{UB_x} > \Delta'_h \Rightarrow \left\lfloor \frac{\ln\left(\frac{b \times T_{max} \times e^{-bx} - a}{bT_{max} - a}\right)}{b} \right\rfloor > \left\lfloor \frac{\ln\left(\frac{b(T_{max} - \delta)e^{-bx} - a}{bT_{max} - a}\right)}{b} \right\rfloor$$

$$\Rightarrow \frac{\ln\left(\frac{bT_{max}e^{-bx} - a}{bT_{max} - a}\right)}{b} > \frac{\ln\left(\frac{b(T_{max} - \delta)e^{-bx} - a}{bT_{max} - a}\right)}{b}$$

$$\Rightarrow \frac{bT_{max}e^{-bx} - a}{bT_{max} - a} > \frac{b(T_{max} - \delta)e^{-bx} - a}{bT_{max} - a}$$

Knowing that $bT_{max} < a$, then:

$$\begin{aligned}
\Delta_h^{UB_x} > \Delta'_h \quad &\Rightarrow \quad bT_{max}e^{-bx} - a < b(T_{max} - \delta)e^{-bx} - a \\
&\Rightarrow \quad bT_{max}e^{-bx} < b(T_{max} - \delta)e^{-bx} \\
&\Rightarrow \quad bT_{max}e^{-bx} < b(T_{max} - \delta)e^{-bx} \\
&\Rightarrow \quad T_{max} < T_{max} - \delta \\
&\Rightarrow \quad \delta < 0
\end{aligned}$$

Contradiction because $0 < b < 1$, $\delta \geq 0$ and $b \times T_{max} < a$.
Therefore, we prove by contradiction that $\Delta_h^{UB_x} \leq \Delta'_h$ □

Lemma 7.2.
For $x = \Delta_c^{UB_x}$, the number of cooling periods produced by a PFP_{ASAP} actual schedule denoted $N'(w)$ is lesser than or equal to the ones produced by UB_x, i.e. $N'(w) \leq N^{UB_x}(w)$.

Proof. Let us suppose that $N'(w) > N^{UB_x}(w)$. From Lemma 7.1 on the facing page we know that $\Delta_h^{UB_x} \leq \Delta'_h$ at least for one time. Then:

$$N'(w) = \left\lceil \frac{w - \Delta'_h}{\Delta_h^{UB_x}} \right\rceil + 1 = \left\lceil \frac{w - (\Delta_h^{UB_x} + \delta)}{\Delta_h^{UB_x}} \right\rceil + 1$$

where $\delta \geq 0$. Then, $N'(w)$ can be written as follows:

$$N'(w) = \left\lceil \frac{w - \delta}{\Delta_h^{UB_x}} \right\rceil$$

Therefore, if $N'(w) > N^{UB_x}(w)$, then:

$$\begin{aligned}
N'(w) > N^{UB_x}(w) \quad &\Rightarrow \quad \left\lceil \frac{w - \delta}{\Delta_h^{UB_x}} \right\rceil > \left\lceil \frac{w}{\Delta_h^{UB_x}} \right\rceil \\
&\Rightarrow \quad \frac{w - \delta}{\Delta_h^{UB_x}} > \frac{w}{\Delta_h^{UB_x}} \\
&\Rightarrow \quad \delta < 0
\end{aligned}$$

Contradiction, because $\Delta'_h \geq \Delta_h^{UB_x}$ and $\delta \geq 0$. Therefore, we prove that $N'(w) \leq N^{UB_x}(w)$ □

Theorem 7.4.
An upper bound on the worst-case response time for task τ_i in the worst-case scenario

described in Section 7.4.1 on page 179 can be obtained from a sequence of execution units of τ_i and those of higher priority jobs and the necessary cooling time units where the cooling periods are of x time units and the heating periods are integers and as long as needed to reach T_{max} (without exceeding) after x time units of cooling, as described by Formula 7.21 on page 189.

Proof. To prove this theorem, we have to study the case where $x = \Delta_c^{UB_x}$ and the one where $x > \Delta_c^{UB_x}$.

Case where $x = \Delta_c^{UB_x}$: From Lemma 7.1 on page 190 and Lemma 7.2 on the preceding page we know that $N'(w) \leq N^{UB_x}(w)$, then:

$$\begin{aligned} N'(w) \leq N^{UB_x}(w) &\Rightarrow N'(w_i^n)x + w_i^n \leq N^{UB_x}(w_i^n)x + w_i^n \\ &\Rightarrow w_i' \leq w_i^{UB_x} \Rightarrow R_i' \leq R_i^{UB_x} \\ &\Rightarrow R_i' \leq R_i^{UB_x} \end{aligned}$$

Hence, when $x = \Delta_c^{UB_x}$, UB_x is an upper bound of tasks worst-case response time according to PFP_{ASAP} algorithm.

Case where $x > \Delta_c^{UB_x}$: Lengthening cooling periods by increasing the x parameter is expected to increase the pessimism of UB_x by increasing tasks response time over estimation given by Equation 7.21 on page 189. To prove that, one can check if the UB_x's response time computation function is increasing. From Equation 7.21 on page 189, the response time function can be written as follows:

$$w_i^{n+1} = \left\lceil \frac{w}{\left\lfloor \frac{\ln\left(\frac{b \times T_{max} \times e^{-b \times x} - a}{b \times T_{max} - a}\right)}{b} \right\rfloor} \right\rceil \times x + w$$

Recall that the ceil function is used to ensure a non null integer length for cooling period, and that floor function is used ensure never exceeding T_{max} after a heating period. Even though, these two functions contribute to increase the pessimism of UB_x, removing them does not change the response time function monotonicity. Then, we can study the monotonicity of this new function that we call $f(x)$ by computing its derivative function as follows:

$$f(x) = \frac{w \times b \times x}{\ln\left(\frac{b \times T_{max} \times e^{-b \times x} - a}{b \times T_{max} - a}\right)} + w$$

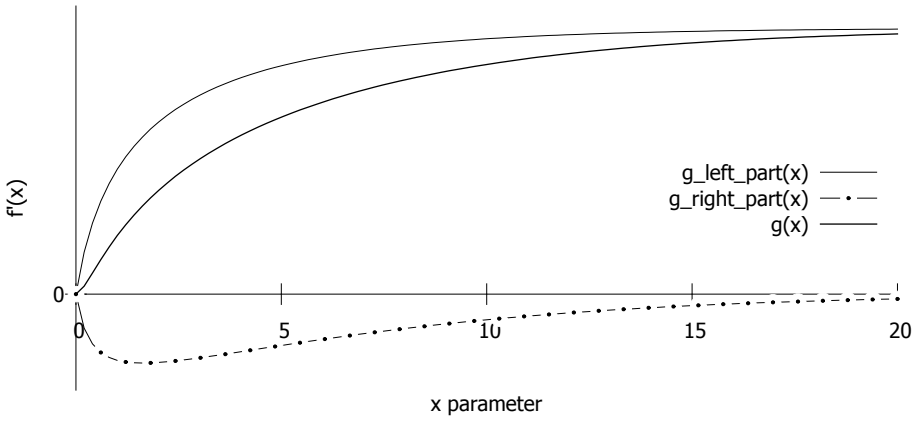


Figure 7.8: $f'(x)$ sign

$$f'(x) = \frac{b \times \ln\left(\frac{b \times T_{max} \times e^{-b\times x} - a}{b \times T_{max} - a}\right) + \frac{b^3 \times T_{max} \times x \times e^{-b\times x}}{b \times T_{max} \times e^{-b\times x} - a}}{\left(\ln\left(\frac{b \times T_{max} \times e^{-b\times x} - a}{b \times T_{max} - a}\right)\right)^2}$$

We do not show the whole study of $f(x)$'s sign, we do this only with deductions. We can see that the sign of $f'(x)$ depends only on the numerator part of the fraction, we denote this part $g(x)$. Then, knowing that $g(0) = \ln(a/(a - bT_{max})) > 0$, we can say that $g(x)$ is positive in interval $[1, +\infty[$ because of the following arguments. Firstly, the left part (i.e the logarithm part) is positively increasing because the combination of the logarithm and the reverse exponential functions leads to decrease the numerator part of the fraction which is inside the logarithm. Knowing that $bT_{max} - a$ is negative, then, $bT_{max}e^{-bx} - a$ is negatively increasing which makes the sign of the fraction necessarily positive and increasing. Secondly, the right part is negative because $b \times T_{max} < a$. It is slightly decreasing from 0 for a while and then it increases asymptotically to 0 as shown in Figure 7.8. Therefore, the sum of these two parts leads $g(x)$ to be positive because the left part dominates the right one as shown in Figure 7.8, and thereby it leads $f'(x)$ to be also positive which means that $f(x)$ is increasing in interval $[1, +\infty[$. Then tasks response time according to UB_x increases when x is increasing.

Therefore, we prove that UB_x upper bounds the actual PFP_{ASAP} worst-case response time. □

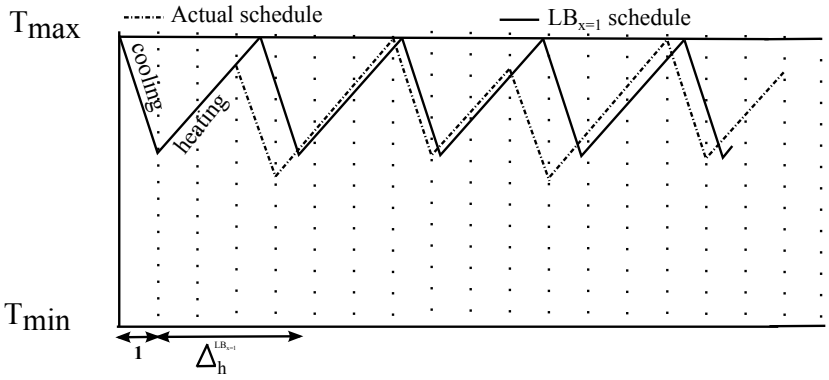


Figure 7.9: Lower bound ($LB_{x=1}$)

Lower bound ($LB_{x=1}$):

Knowing that the actual schedule respects the discrete time constraint, we can compute a lower bound of the actual tasks worst-case response time by violating this constraint, i.e. allowing non-discrete execution times (see 7.9). The following points summarize the behavior of $LB_{x=1}$:

- Cooling down the system for one time unit. This is sufficient because continous time allows executing less than one time unit.
- Then, executing jobs and heating up the system until T_{max} is reached or there is no pending workload; the length of this period is not necessarily an integer.
- Repeat cooling-heating cycles until there is no pending workload.

The lower bound of task τ_i of priority level-i that is requested simultaneously with higher priority tasks with $T(0) = T_{max}$ is given by Equation 7.24

$$\begin{cases} w_i^{n+1} &= N(w_i^n) + w \\ R_i &= w_i^{n+1} = w_i^n \end{cases} \tag{7.24}$$

where :

- $N(w)$ is the number of cooling periods needed to execute the workload w without exceeding T_{max}:

$$N(w) = \left\lceil \frac{w}{\Delta_h^{LB_{x=1}}} \right\rceil \tag{7.25}$$

- w is the workload of time interval $[0, w_i^n[$.
- $\Delta_h^{LB_{x=1}}$ is the time to execute jobs and heat up the system from the temperature reached after one time unit of cooling to exactly T_{max}:

$$\Delta_h^{LB_{x=1}} = \frac{\ln\left(\frac{b \times T_{max} \times e^{-b} - a}{b \times T_{max} - a}\right)}{b} \tag{7.26}$$

Conjecture 7.1.
A lower bound on the worst-case response time for task τ_i in the worst-case scenario described in Section 7.4.1 on page 179 can be obtained from a sequence of execution units of τ_i and higher priority jobs and the necessary cooling time units where the cooling periods are of one time unit and the heating periods are continuous (not necessarily integers) and as long as needed to reach exactly T_{max} after one time unit of cooling, as described by Formula 7.24 on the facing page.

Proof. Let's denote Δ_H the longest execution period produced by the actual PFP_{ASAP} schedule. Knowing that $\Delta_h^{LB2} \geq \Delta_H$ because Δ_H is an integer and Δ_h^{LB2} is a real number, then:

$$\begin{aligned} \Delta_h^{LB2} \geq \Delta_H \quad &\Rightarrow \quad \left\lceil \frac{w}{\Delta_h^{LB2}} \right\rceil \leq \left\lceil \frac{w}{\Delta_H} \right\rceil \\ &\Rightarrow \quad \left\lceil \frac{w}{\Delta_h^{LB2}} \right\rceil + w \leq \left\lceil \frac{w}{\Delta_H} \right\rceil + w \\ &\Rightarrow \quad R^{LB2} \leq R_H \leq R \end{aligned}$$

Therefore, we prove that $LB2$ lower bounds the actual worst-case response time according to PFP_{ASAP}. □

$UB_{T_{min}}$ vs. UB_x

The tightness of the upper bound UB_x relative to $UB_{T_{min}}$ depends on the parameter x. In fact, the greater x is, the more pessimistic UB_x is because of the nature of the cooling function which is asymptotically decreasing to T_A. Then, for small values of x, UB_x is tighter and for great values $UB_{T_{min}}$ is tighter. The experiments presented in Section 7.6 on page 197 demonstrates the differences in practice between UB_x and $UB_{T_{min}}$ in term of tightness and complexity.

Utilization bound

Under thermal constraints, cooling periods are needed to prevent the system exceeding T_{max}. This means that for a certain processor utilization, more time is needed for cooling which means that the processor cannot be used at 100%. One can use this idea to propose a new maximum processor utilization that can respect the thermal constraints. In the following we discuss utilization bounds that consider cooling time.

Maximum utilization

Without considering thermal constraints, task sets cannot be feasible with a processor utilization greater than 100% for monoprocessor platforms. Furthermore, knowing that respecting the thermal constraints needs to add some cooling time, then, it is obvious

that a task sets with a processor utilization of 100% cannot be feasible with PFP_{ASAP}, especially when T_{max} is limited. To compute the maximum supportable processor utilization that take into account cooling time, one can use the idea of overestimating response times, by overestimating the cooling time needed to execute the workload of one hyper-period. We can use for instance the idea of UB_x to compute an upper bound for the maximum supportable processor utilization.

Lemma 7.3.
An upper bound of the processor utilization $U = \sum_{1 \leq i \leq n} C_i/T_i$ for thermal-aware fixed-priority task sets can be obtained by Equation 7.27.

$$U \leq \frac{\Delta_h^{UB_x}}{\Delta_h^{UB_x} + x} \tag{7.27}$$

where $\Delta_h^{UB_x}$ is given by Equation 7.26 on page 194.

Proof. We first upper bound the workload of one hyper period with UB_x then we compute the corresponding processor utilization, and finally we compute the maximum achievable utilization. The workload of a hyper-period HP can be obtained by multiplying HP by the processor utilization U. Then, we can replace w by $U \times HP$ in Equation 7.21 on page 189 to compute the time needed (cooling and workload) to satisfy the workload $U \times HP$. Finally, we can compute the new utilization U^*, that considers cooling time, by dividing the time demand (cooling and workload) by the available time HP, Equation 7.28 shows how to compute U^*.

$$U^* = \frac{\left\lceil \frac{U \times HP}{\Delta_h^{UB_x}} \right\rceil \times x + U \times HP}{HP} \tag{7.28}$$

For a task set to be feasible, the new utilization U^* must be lesser than 1 because the available time must be greater or equal to the time demand. Then,

$$\begin{aligned}
U^* \leq 1 \quad &\Rightarrow \quad \frac{\left\lceil \frac{U \times HP}{\Delta_h^{UB_x}} \right\rceil \times x + U \times HP}{HP} \leq 1 \\
&\Rightarrow \quad \frac{\frac{U \times HP \times x}{\Delta_h^{UB_x}} + U \times HP}{HP} \leq 1 \\
&\Rightarrow \quad U \times \left(\frac{x}{\Delta_h^{UB_x}} + 1 \right) \leq 1 \\
&\Rightarrow \quad U \leq \frac{\Delta_h^{UB_x}}{\Delta_h^{UB_x} + x}
\end{aligned}$$

□

Liu and Layland bound We can use the same reasoning as the above utilization upper bound to propose a sufficient and pessimistic feasibility test based on Liu and Layland bound. In fact, we can compare the total time utilization (processor and cooling) to Liu and Layland bound.

Lemma 7.4.
An upper bound of the processor utilization $U = \sum_{1 \leq i \leq n} C_i/T_i$ for thermal-aware fixed-priority task sets with implicit deadlines can be obtained by Equation 7.29.

$$U \leq \frac{\Delta_h^{UB_x} \times n(\sqrt[n]{2} - 1)}{\Delta_h^{UB_x} + x} \tag{7.29}$$

Proof. To prove this Lemma we just have to compare the over estimated utilization U^* given by Equation 7.28 on the preceding page to Liu and Layland bound.

$$\begin{aligned}
U^* \leq n(2^{1/n} - 1) \quad &\Rightarrow \quad \frac{\left\lceil \frac{U \times HP}{\Delta_h^{UB_x}} \right\rceil \times x + U \times HP}{HP} \leq n(\sqrt[n]{2} - 1) \\
&\Rightarrow \quad \frac{\frac{U \times HP \times x}{\Delta_h^{UB_x}} + U \times HP}{HP} \leq n(\sqrt[n]{2} - 1) \\
&\Rightarrow \quad U \times \left(\frac{x}{\Delta_h^{UB_x}} + 1 \right) \leq n(\sqrt[n]{2} - 1) \\
&\Rightarrow \quad U \leq \frac{\Delta_h^{UB_x} \times n(\sqrt[n]{2} - 1)}{\Delta_h^{UB_x} + x}
\end{aligned}$$

□

7.6 Performance Evaluation

In this section, we present the results of an empirical investigation, examining the effectiveness of our sufficient schedulability tests.

7.6.1 Task set generation

To perform these experiments, we randomly generated 100000 task sets, varying the processor utilization. We varied U in the range $[0.05, 1]$ in steps of 0.05. Hence we obtained 5000 distinct task sets for each U step. Each task set is composed of 10 tasks. The thermal parameters was set as, $T_{max} = 32\ ^\circ C$, $b = 0.228$, and $a = \beta_0 \times S^3 = 8$. These parameters are the ones of the whole system (including an eventual cooling device) and correspond to a classical Intel Pentium processor parameters [Het+12]. The task parameters were randomly generated as follows: task processor utilization

($U_i = C_i/T_i$) using the *U-Unifast Discard* algorithm [BB05], and periods randomly generated between 2 and 25200 time units with a hyper-period limitation technique [MG01]. Task deadlines were implicit.

We used YARTISS as a simulation environment (see Chapter 8) which respects the following hypotheses: discrete time (all scheduling operations are performed before or after one time unit), the heating behavior follows the Fourier's law (see Equation 7.6 on page 177) and temperature values are real numbers.

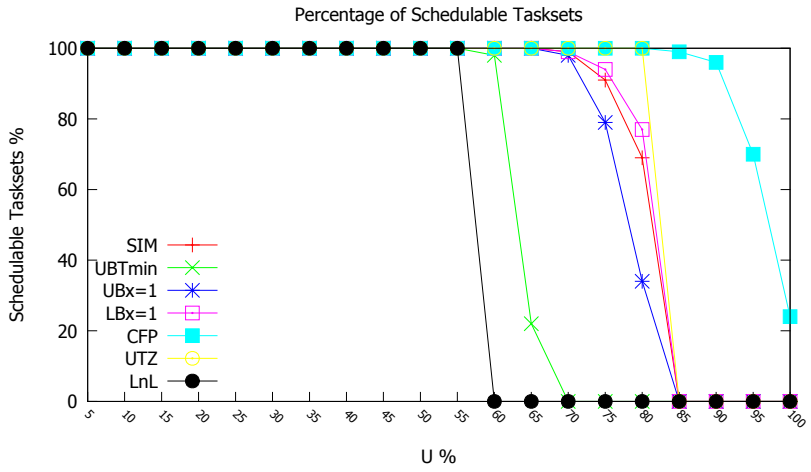
7.6.2 Schedulability tests investigated

We investigated the performance of the following schedulability tests.

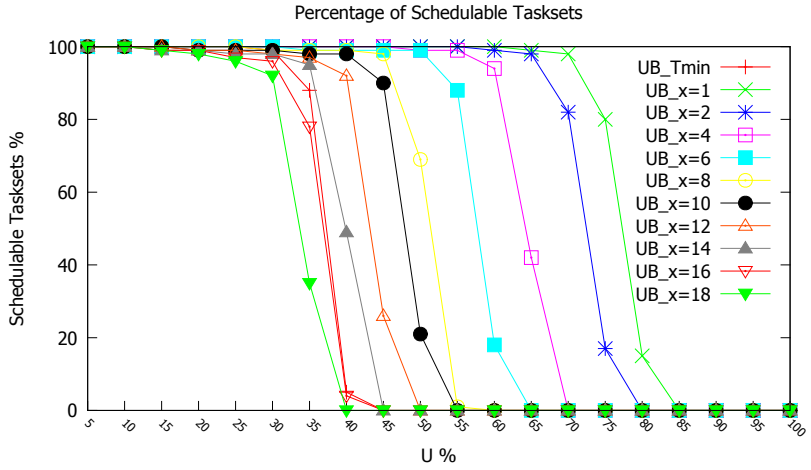
- SIM: is an empirical necessary and sufficient test based on simulating the schedule of PFP_{ASAP} over more than one hyper-period, starting with synchronous release and the maximum temperature level which corresponds to the worst-case scenario discussed in Section 7.4.1 on page 179.
- $UB_{T_{min}}$: the sufficient test presented in Section 7.5.2 on page 184, we consider that $T_{min} = 1\ °C$.
- UB_x: the sufficient test presented in Section 7.5.2 on page 188, the parameter x is varied from 1 to 18.
- $LB_{x=1}$: the necessary test presented in Section 7.5.2 on page 194.
- CFP: the exact test for fixed-priority ignoring thermal and energy constraints. This was used to provide a schedulability bound, considering only processing time.
- UTZ: the necessary condition described in Section 7.5.2 on page 195.
- LnL: the sufficient condition described in Section 7.5.2 on the preceding page.

7.6.3 Experiments

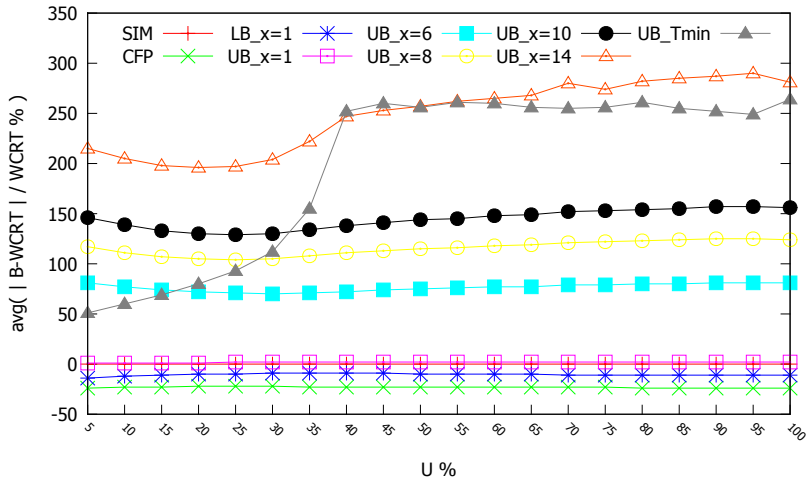
Figure 7.10(a) on the next page shows how the percentage of task sets that are deemed schedulable by each of the tests varies with processor utilization. The CFP test has notionally the highest performance since it is widely optimistic and ignores thermal considerations. When temperature is considered, UTZ, $LB_{x=1}$ provide necessary tests, upper bounding the number of task sets that are proved to be schedulable by the exact empirical test SIM. We observe that the results confirm that $UB_{T_{min}}$ and UB_x provide sufficient schedulability tests and that for $x = 1$, UB_x is a tighter bound than $UB_{T_{min}}$ with a larger improvement at higher utilization levels. Furthermore, this experiment confirms also the validity of the utilization bounds given in Section 7.5.2 on page 195. We can see that the maximum achievable utilization for 10 tasks is 80% for UTZ and the adapted Liu and Layland bound is 57%.



(a) Percentage of Task sets schedulable over U variation

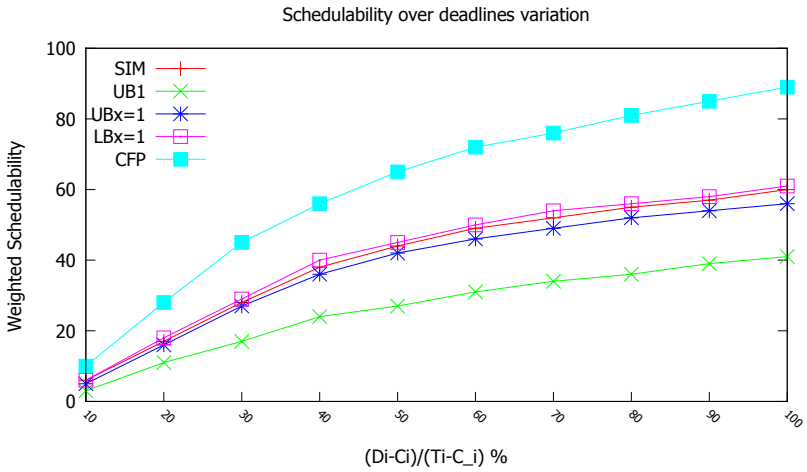


(b) Schedulability by Varying x

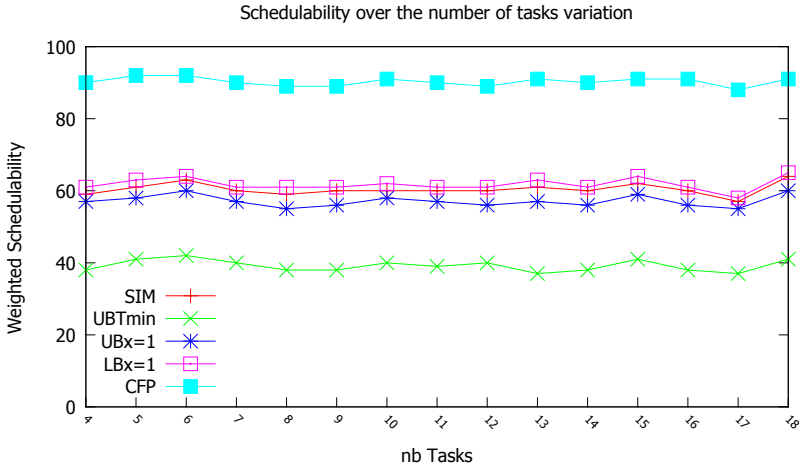


(c) Bounds tightness over U variation

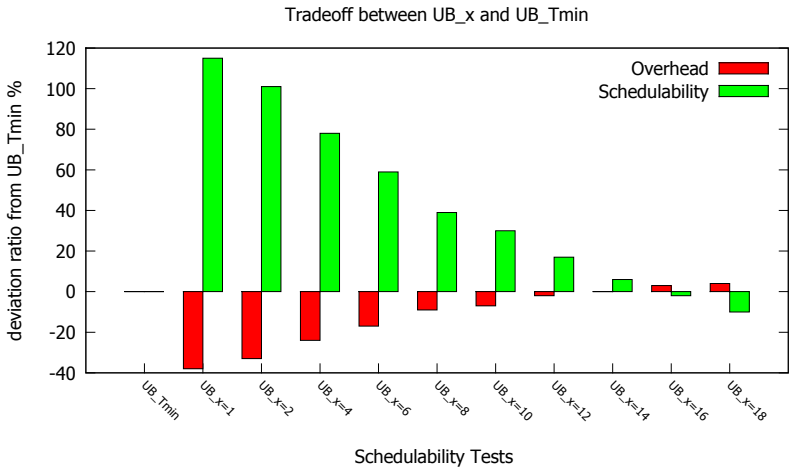
Figure 7.10: Simulations results



(a) Varying relative deadlines



(b) Varying the number of tasks



(c) Trade-off of schedulability and overhead between UB_x and $UB_{T_{min}}$

Figure 7.11: Simulations results

Figure 7.10(b) on page 199 compares the pessimism of $UB_{T_{min}}$ based shcedulability test to UB_x's one by varying the x parameter. We observe that UB_x stays less pessimistic than $UB_{T_{min}}$ for small values ($1 \leq x \leq 14$ in this experiment), however, it becomes more pessimistic starting from $x = 14$. This result is as expected because the longer cooling periods are, the slower the temperature decreases and the longer the response times are.

Figure 7.10(c) on page 199 shows average deviation of bounds from the exact response time given by simulations over processor utilization. The upper bounds have positive values and lower bounds have negative values (the deviation of SIM is 0 because it gives the exact response time). We can see that deviation of UB_x, $LB_{x=1}$ and CFP are still stable over utilization variation in contrast of $UB_{T_{min}}$ which behaves badly when utilization goes high. We notice also that when $x = 1$, UB_x and $LB_{x=1}$ are very close to the actual response time which makes them very interesting tools for approximate schedulability analysis. However, increasing x leads UB_x to be less precise, when $x > 14$, UB_x behaves as bad as $UB_{T_{min}}$ or worse.

Weighted Schedulability As well as processor utilization, task set schedulability is dependent on a number of other key parameters, including: tasks deadlines model and the number of tasks. Evaluating all possible combinations of these parameters is not possible. Instead, the evaluation in this section varies one parameter at a time, with the results presented in terms of the weighted schedulability measure [BBA10].

The figures in this section show the weighted schedulability measure described in Section 5.4.2 on page 149.

Figure 7.11(a) on the facing page shows the impact of constrained deadlines on performance. Here we vary the deadlines from heavily constrained where $D_i - C_i$ is 10% of $T_i - C_i$ to 100% of $T_i - C_i$ (i.e. implicit deadlines). We observe that all of the schedulability tests are influenced by the tightness of deadlines to a similar degree, with heavily constrained deadlines having significant impact on schedulability in all cases.

Figure 7.11(b) on the preceding page shows the impact of the number of tasks on the effectiveness of the feasibility tests. We can see that the number of tasks does not alter the feasibility rate of all the bounds.

Figure 7.11(c) on the facing page shows the trade-off of schedulability and overhead between UB_x and $UB_{T_{min}}$ over parameter x variation. The shown percentages represent the gain or the loss of schedulability and overhead comparing to $UB_{T_{min}}$. For example, for $x = 1$, The schedulability test based on $UB_{x=1}$ has more than 100% of schedulability than $UB_{T_{min}}$ and almost 40% more overhead than $UB_{T_{min}}$. Positive values mean better performances than $UB_{T_{min}}$ and negative value mean worse performances. We observe that increasing parameter x decreases the percentage of schedulable task

sets, however, it decreases the test complexity or overhead. We observe also that the loss of schedulablity is higher than the gain of overhead.

7.7 Conclusion

In this chapter, we addressed the problem of fixed-priority real-time scheduling for thermal-aware systems, where both time and thermal constraints have to be met. In a previous chapter, we showed that the scheduling policy PFP_{ASAP} is optimal among all fixed-priority scheduling algorithms for energy-non-concrete energy-harvesting systems. The main contributions of this chapter are as follows. we adapted PFP_{ASAP} algorithm to the thermal-aware model, we proved its optimality and we proposed two schedulability tests based on response-time upper bounds $UB_{T_{min}}$ and UB_x which is a parametric bound. Finally we performed simulations to validate the theoretical results. The scheduling problems of energy-harvesting and thermal-aware systems seem to be close to each other. This opens the door to use and adapt scheduling solutions from one side to an other. This work is a first step in this way. It is interesting to continue studying deeply the possible similarities between energy-harvesting model and the thermal-aware's one and to explore more adaptable and extensible results of each model.

Chapter 8

Simulation Tool: YARTISS

Contents

8.1 Introduction

The real-time scheduling theory has been studied by many researchers since decades, and many solutions and approaches have been proposed in the past to optimize the scheduling of real-time tasks on single and multiple processor systems. In order to check if a task set respects its temporal constraints according to a specific scheduling policy or to evaluate the efficiency of a new approach against other algorithms, using simulation software is considered as a valid comparison technique and it is commonly used in the evaluation of real-time systems.

Unfortunately, there is no standard simulation tool approved by the real-time community and the existing tools are usually hard to be extended due to various reasons such as code complexity, software copyrights or poor documentation. As a result, most of the researchers tend to create their own simulation tools. This situation raises some concerns. On one hand, results are hard to be validated without careful examination of the used simulation tool. So these results might be biased toward the proposed approach either by adapting the generation of tasks or by biased implementation against the compared algorithms. On another hand, the use of out-of-date simulation tools or the lack of good documentation pushes researchers to not use the existing tools and to create their own ones, which leads to repetitive implementations especially the common ones e.g. **E**arliest **D**eadline **F**irst (*EDF*), **R**ate **M**onotonic (*RM*) and **D**eadline **M**onotonic (*DM*).

Besides the time and the effort spent by researchers in developing new simulation tools, the process of implementing comparable algorithms can be sometimes complicated and time-consuming. If a standard simulation platform succeeds to emerge, one can re-use already-implemented policies from literature and compare them with new policies without the need to understand their specifications and particularities. Finally, the simulation protocols can be standardized, and easily describable by the use of such a reference tool.

In this chapter, we introduce ***Y**et **A**n Other **R**eal-**Ti**me **S**ystems **S**imulator (YARTISS)*, a new simulation tool for real-time multiprocessor systems, which provides various functions to simulate the scheduling process of real-time task sets and their temporal behavior.

The main particularity of *YARTISS* is its genericity, by which we aim to overcome the previously mentioned problems. Its architecture is designed in a way to allow new users to add their own policies and algorithms and extend the simulator easily with no need to modify the core of the simulator or even understand how it is built. *YARTISS* is an extension of previous simulator projects [Mas; Fau] designed earlier by our research team. We learned a lot from these attempts and we included this experience in the development of *YARTISS*.

YARTISS is written in Java programming language, which is a popular object-oriented language that offers valuable attributes regarding code portability. In order to ensure independence between the different features of the simulator and to control the development process, we used modern programming paradigms like module-oriented programming and agile programming methods. We tried to develop *YARTISS* keeping in mind that for a simulator in order to become a reference tool, it should have the following properties: 1) the software must be available under an open source license which gives any researcher the freedom to analyze, verify and modify its implementation without the permission of the copyright holder; 2) the **A**pplication **P**rogramming **I**nterface (API) of the software must be well-documented and the developer who wants to add or modify a part should be able to do it easily with no need to read the entire source code in order to understand its behavior; 3) each part of the simulator must be

independent from the other parts (or at least acyclic dependences must be guaranteed , and easily replaceable by an external module; and 4) the simulator has to be easy to use in a way that a non-developer researcher can be able to use it easily. Due to its genericity and modularity, we hope that *YARTISS* makes a valuable contribution to the long process of developing a standard simulation tool recognized by the real-time scheduling research community. *YARTISS* was developed in our Lab in collaboration with Manar Qamhieh and Frédéric Fauberteau. The work presented in this chapter is also available in [Cha+12].

The structure of this chapter is as follows. We review related works and examples of real-time simulators in Section 8.2. Section 8.3 on the following page contains our motivation and shows the history of *YARTISS*. Then, in Section 8.4 on page 207 we present the various features of our simulation tool. The architecture design of the simulator is described in Section 8.5 on page 213. We present a case study in Section 8.6 on page 221 in order to demonstrate the extensibility of the tool. Information about the available versions of *YARTISS* and its download and install instructions are provided in Section 8.7 on page 223. Finally, Section 8.8 on page 223 concludes this chapter.

8.2 Related Works

In this section we explore the existing works related to simulation tools and we identify their respective strengths and weaknesses. There exist many tools to test and visualize the execution of real-time systems. These tools are divided mainly into two categories: the execution analyzer frameworks and the simulation software. Regarding the execution analyzers, one can refer to RESCH [KRI09] which is a loadable real-time scheduler framework for Linux. Also, Grasp [Hol+10] which is a set of tools designed for tracing and measuring the scheduling of real-time tasks. Furthermore, LitmusRT [Cal+06] is a real-time extension to Linux kernel for multiprocessor systems that monitors the scheduling behavior of a real task set on a real platform.

Among open-source simulation tools, we start by referring to MAST [GH+01] which is a modeling and analysis suite for real-time applications that was developed in 2000. MAST is an event-driven scheduling simulator that permits modeling distributed real-time systems and offers a set of tools to test their feasibility or to compute their sensitivity analysis. Another known simulator is Cheddar [Sin+09; Sin+04] which was developed in 2004, it handles the scheduling of real-time tasks on multiprocessor systems. It provides many implementations of basic scheduling, partitioning and analysis of algorithms. Unfortunately, no API documentation is available to help implementing new algorithms and to facilitate its extensibility. Moreover, Cheddar is written in Ada programming language [MSH11] which is used mainly in embedded systems and has strong features such as modularity mechanisms and parallel processing. Ada is often the language of choice for large systems that require real-time processing, but in general, it is not a common language for desktop or web applications. We believe that the choice

of Ada reduces the potential contributions to the software from average developers and researchers.

Finally, STORM [UDT09] and FORTAS [CG11] are simulation tools which, as *YARTISS*, are written in Java. The first one, STORM, was released in 2009 and was presented as a simulation tool for real-time multiprocessor scheduling. It has a modular architectures and can simulate the scheduling of task sets on multiprocessor systems according to several scheduling policies. The specifications of the simulation parameters and the scheduling policies are modifiable using an **Ex**tensible **M**arkup **L**anguage (XML) file. However, the simulator tool lacks a detailed documentation and description of the available scheduling methods and the features of the software. The second one, FORTAS, is a real-time simulation and analysis framework which targets uniprocessor and multiprocessor systems. It was developed mainly to facilitate the comparison process between the different scheduling algorithms, and it includes features such as task generators and computation of results of each tested and compared scheduling algorithm. FORTAS represents a valuable contribution in the effort towards providing an open and modular tool. Unfortunately, it seems to suffer from the following issues: its development is not open to other developers up to now (we can only download *.class* files), no documentation is yet provided and no new version has been released to public since its presentation in [CG11].

During the development of *YARTISS*, we learned from those existing tools and we included some of their features in addition to others of our own. Our aim is to provide a simulation tool for multiprocessor systems that is easily extendable by fellow researchers and developers, and it can be used to compare, simulate and visualize the scheduling of real-time tasks on multiprocessor systems.

8.3 The History of *YARTISS*

YARTISS is the fourth simulation tool developed by our team during the last few years. In this section we present each one of these tools, their contributions, challenges and limitations. Each one of these tools was built for a specific purpose and consumed relatively a long period of time to be developed. We used the experience from the earlier simulators in the design of *YARTISS*. This is done by summing and optimizing their functionalities while avoiding their limitations.

Our first try in writing a real-time system simulator was called RTSS [Mas] and it was developed between 2005 and 2008. RTSS was initially developed to test some scheduling algorithms on uniprocessor systems in order to handle temporal fault tolerance, such as preemptive fixed priority, *EDF* and D^{OVER} algorithms [KS95]. Later, it was extended in order to test aperiodic tasks with task servers such as *Polling and Deferrable Servers* and their handling algorithms [MM08]. RTSS suffered from some problems. For example, many modifications have been made in a hurry with certain assumptions on the behavior of the existing code without a proper documentation. Also, RTSS

was not easily extendable, a single modification in one part of the code could cause non-understandable errors in another part of the software. Moreover, although the tool was initially written in Java, it began to rely more and more on bash scripts which are used mainly to launch the simulation processes and to transform outputs into human readable files.

Based on RTSS, a second simulation tool was developed between 2008 and 2011. It was called RTMSim [Fau] and it targeted the scheduling simulation of multiprocessor platforms [FMG11]. In the meanwhile, RTSS had become such complicated and not maintainable that we had to start over rather than to extend its functionalities to multiprocessor systems. The general key ideas of RTSS were kept for RTMSim. Unfortunately, the validated parts of RTSS, which were of no interest at that time, were not re-implemented in RTMSim and thus they were lost.

A third try was made in early 2011. RTSS v2 [Mas] was developed as a rebuild of RTSS in the aim of including energy consuming tasks. Unfortunately, even if RTSS v2 is more usable today than the first version, it still suffers from the same problems of the original tool, namely the poor documentation and the lack of modularity. Moreover, it seemed difficult to extend it to simulate multiprocessor platforms.

So we came to the development of a new software: *YARTISS*. From the beginning, we aimed to produce a tool where the task models, the number of processors and scheduling behavior such as energy consumption models can be easily added and modified. Another important point is the usability of the user interface to produce human readable traces of the simulation process of scheduling algorithms. Our goal was to develop a tool that is able to perform evaluations as well as to debugging various algorithms including the energy-related algorithms. When we wanted to use *YARTISS* for another purpose, for example, the simulation of scheduling dependent real-time tasks of the acyclic graph model (see [Qam+12]), this was done without any problems which validated the extensibility of our simulator. This point is explained in more detail in Section 8.6 on page 221.

8.4 Functionalities

There are two main functionalities in our simulation tool, the first is the simulation of tasks execution and the temporal behavior of a task set according to a specific scheduling policy. The second functionality is the large-scale comparison of several scheduling policies in different scenarios. Both functionalities require a third feature which is the random task set generation.

In this section we explain all functionalities of *YARTISS* in details while showing their specifications and various characteristics regarding the scheduling problem of real-time systems.

8.4.1 Single Task Set Simulation

This feature deals with the simulation of the execution of a task set on mono/-multiprocessor platform with a specific scheduling policy. The used task sets can be loaded into the simulator either through the graphic user interface by loading an input file or entering the parameters manually, or by using a task set generator. We can parameterize the desired simulation and run it easily. Furthermore, several views are proposed. The simulation parameters are the task set, the number of processors, the scheduling algorithm and the energy profile.

Task Models

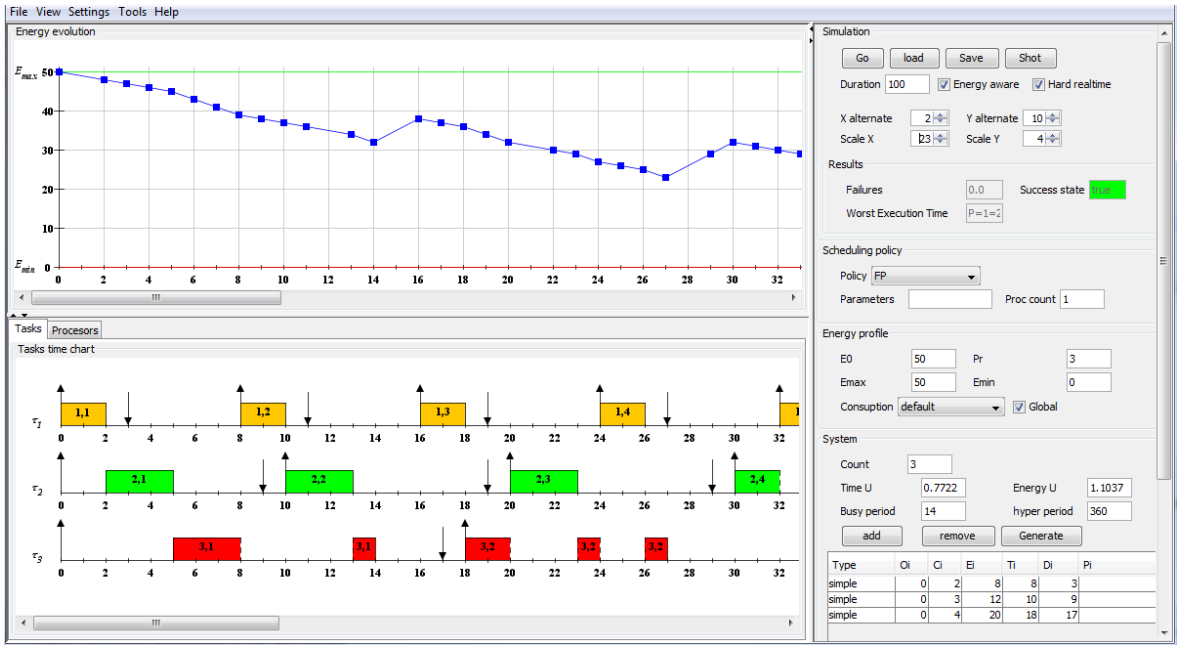
YARTISS offers an open architecture that greatly facilitates the integration of different task models. The current version proposes two models, the first one is the Liu and Layland task model augmented with energy related parameters. All tasks are considered independent and each task τ_i is characterized by its **W**orst **C**ase **E**xecution **T**ime (WCET) C_i, its worst case energy consumption E_i, its period T_i and its deadline D_i. The second model is the **D**irected **A**cyclic **G**raph (DAG) task model which is a common real-time dependent task model for multiprocessor systems. It is used to implement systems consisting of number of subtasks with dependencies to control their execution flow. In this model, a graph task G_i is a collection of real-time subtasks $\{\tau_{i,1}, \tau_{i,2}, ...\tau_{i,q}\}$ that share the same deadline D_i and period T_i of the DAG, and they differ in their own WCET $C_{i,j}$. The directed edges between the tasks of the graph determine their precedence constraints, and since each task in the graph might have more than one successor and predecessor, concurrent execution can be generated.

These two models are the common models of independent and dependent real-time tasks, and they can have different characteristics regarding deadlines (implicit, constrained), and regarding periods (periodic, sporadic). We show in Section 8.6 on page 221 a case study to demonstrate how to easily integrate new task models into *YARTISS*.

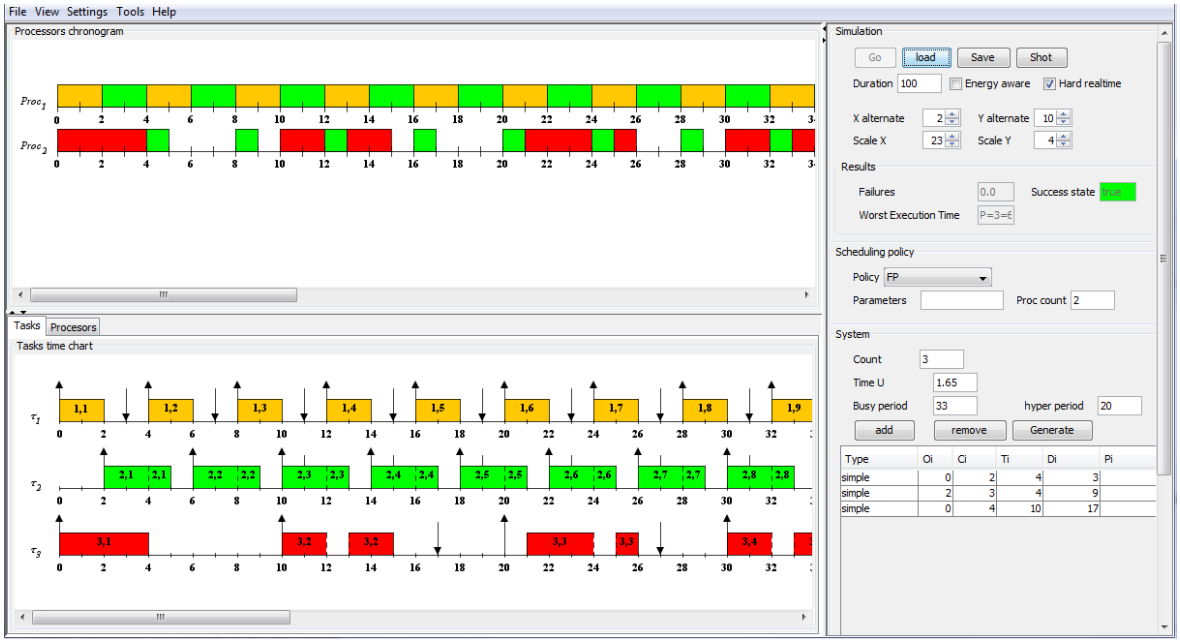
Mono / Multiprocessor

Our simulation tool supports multiprocessor scheduling. According to this, the number and the type of processors is passed as a parameter of the simulation. Initially we considered a single type of platforms which consists of identical monospeed processors. This model is extended in *YARTISS* to include heterogeneous systems in which the processors can have different characteristics.

By using the simulator, one can implement and test his own multiprocessor algorithms and partitioning policies. Also, we considered a global scheduling of task sets on multiprocessor systems. It is defined as the scheduling in which the execution of a job can be interrupted by higher priority jobs and the execution can be resumed on a different processor. By default, some multiprocessor scheduling algorithms were

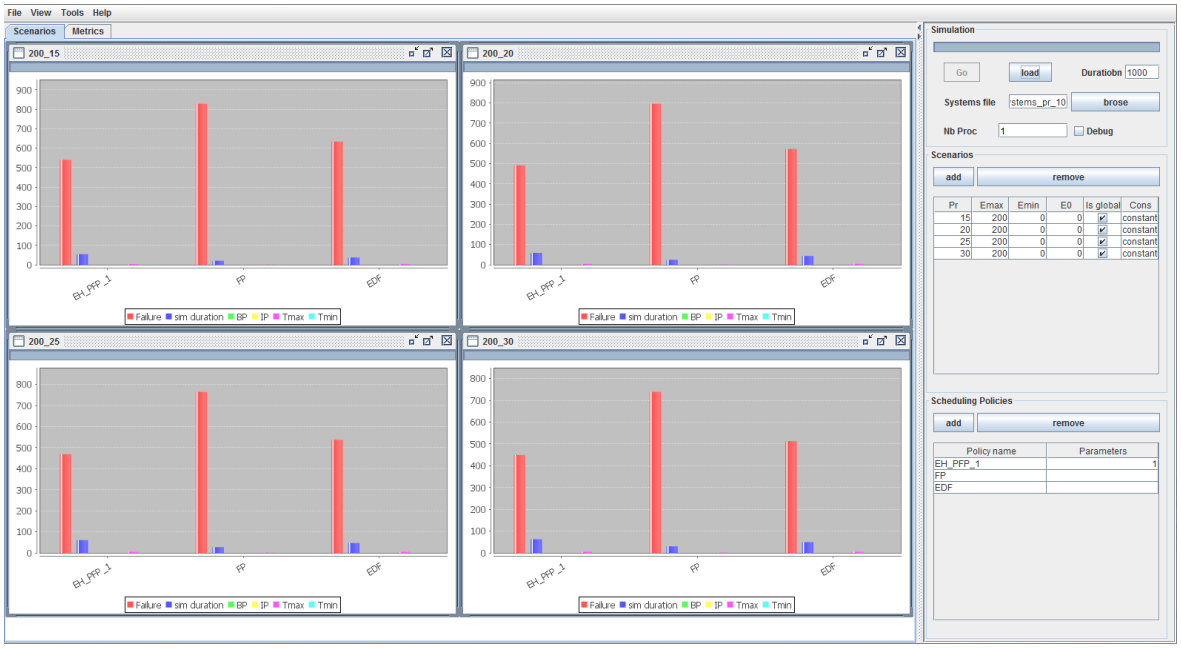


(a) Time-line view

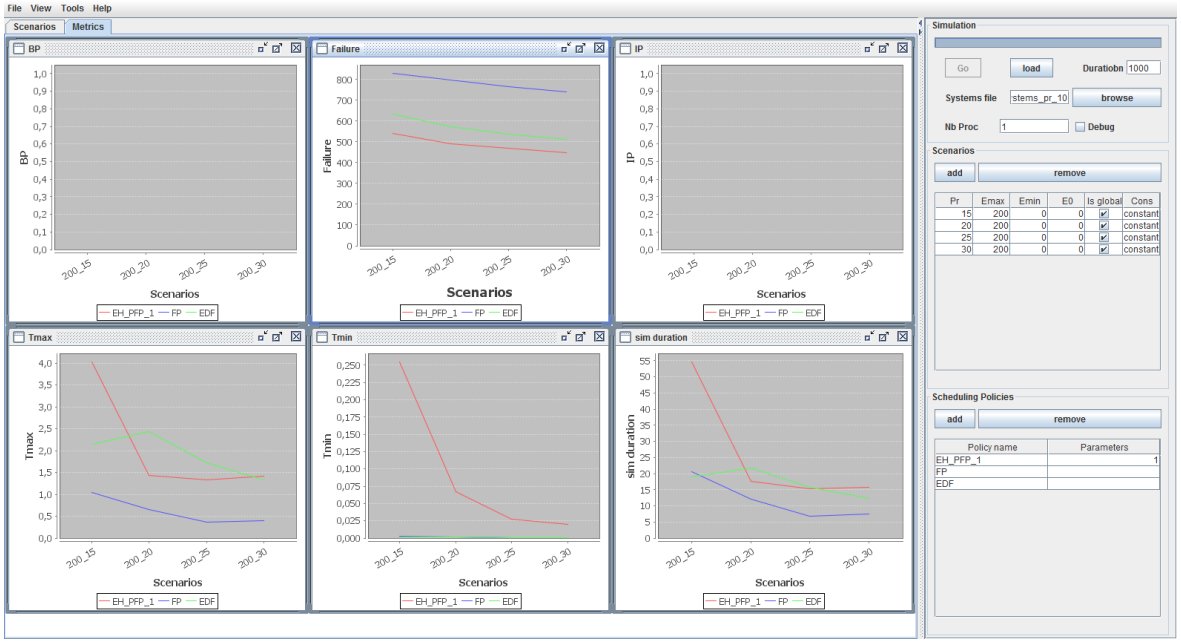


(b) Multiprocessor view

Figure 8.1: Energy and multiprocessor views



(a) Scenarios view



(b) Metrics view

Figure 8.2: Large scale simulation view

implemented in *YARTISS* on multiprocessor systems like *EDF* and **F**ixed-**T**ask-**P**riority (FTP).

Energy Profile

Unlike many other simulators, *YARTISS* permits us to simulate the production and the consumption of energy in real-time systems. It permits the user to model an energy-harvester and an energy storage unit such as a battery or a capacitor with a certain capacity. A renewable energy source can be modeled also using a charging function. It is possible as well for the user to implement and use customized energy profiles. Figure 8.2 on the facing page shows screen-shots of the user interface of *YARTISS*. We can see in Figure 8.1(a) on page 209 the time chart of the storage unit energy level where we can see the consumption during the scheduling process. The energy view, which is used to print the energy level chart is among many views provided by *YARTISS* to show other metrics.

Energy Source Model: We have implemented an energy source profile that models a renewable energy source represented by an energy storage unit with limited capacity. Other models can be added by the user by implementing the profile interface and injecting it into the engine module of the simulator. This process is not complicated and it can be done without modifying the other modules of the simulator.

Consumption Model: It is important to note that the energy consumption of a task must be modeled independently from its WCET as in the case of [JM06]. Therefore, our simulator provides the ability to define a consumption profile for each task of the system or to choose one global profile to apply to all tasks. A consumption model is represented by a function and must be able to provide the amount of energy consumed within a time interval during the tasks execution i.e. the integral of the consumption function. The implemented models so far are: *Linear consumption* (not realistic but permits to establish some interesting conclusions) and *Early instantaneous consumption* where all the energy cost of a task is consumed as soon as a task is scheduled. This latter model seems to be the worst-case scenario.

8.4.1.1 Scheduling Policy

The main purpose of the simulator is to test scheduling algorithms by comparing them to each other in order to measure their performances and efficiency. Much attention has been focused on the design of this part of *YARTISS* to make it as generic as possible so that users can add, override and inject their own scheduling policies easily. There are currently many algorithms implemented in *YARTISS* with different priority assignment policies such as fixed-job, fixed-task and dynamic priority. It includes also classical algorithms such as *RM*, *DM*, *EDF*, **L**east **L**axity **F**irst (*LLF*) for both mono and multiprocessor platforms. Users can add new scheduling policies easily, and Section 8.6 on page 221, we demonstrate in details how to integrate a new policy independently from the rest of the modules of the simulator.

8.4.2 Run Large Scale Simulations

One of the major features of *YARTISS* is the large scale comparison of several algorithms or scheduling policies. It is similar to the single task set simulation but it is performed on a large set of task sets in different configurations and scenarios. The comparison results are based on statistics such as the number of schedulability failures or deadline misses, the system's lifetime, the amount of time spent at maximum and minimum energy levels or the average duration of idle and busy periods. Multiple simulations can run simultaneously due to the use of the multi-threading concept supported by Java. As a result, the duration of simulations is greatly reduced through the parallelism of the used hardware.

8.4.3 Task Set Generation

Performing large scale simulations requires a large set of task sets. For the simulation results to be credible, the used task sets should be randomly generated and varied sufficiently. The simulator provides the ability to choose a generator according to the desired task models and algorithms. The current version includes by default a generator based on the *UUniFast-Discard algorithm* [BB05] coupled with a hyper-period limitation technique [MG01] adapted to energy constraints. This algorithm generates task sets by distributing the processor utilization and the energy utilization on tasks randomly. Since the original version of the algorithm does not include energy, we had to adapt it to produce feasible systems regarding time and energy. The idea behind the algorithm is to distribute the system's utilization on the tasks of the system. When we consider tasks energy cost, we end up with two parameters to vary and two conditions to satisfy. The algorithm in its current version distributes U and U_e uniformly on the tasks in such a way that when we generate n task sets, we get approximately the same number of task sets for each utilization value. Then, the algorithm finds the couples (C_i, E_i) which satisfies all the conditions namely U_i, U_e and energy consumption constraints.

The operation is repeated several times until the desired couples satisfy the imposed conditions.

8.4.4 Graphical User Interface (GUI)

To facilitate the use of the simulator by a large number of users, we provide our tool with a **G**raphic **U**ser **I**nterface (GUI) to support the above-mentioned features in an interactive and intuitive way. After the simulation of a specific system with a specific energy profile and a specific scheduling policy, the user can follow and analyze the scheduling process using three different views: a time chart (Gant chart), a processor view and the energy storage level chart which monitors the energy levels of the system. In order to run simulations and get the results of comparison of scheduling policies, the simulator offers a control panel that allows the user to select the scheduling policies, to choose the energy scenarios and to start the simulations. Thus, the user can see the simulation results as one graph per scenario or per comparison criterion or metric. Then, the simulator can obtain results on a large number of randomly generated task sets in order to evaluate the performance of the tested scheduling policies, and easily explore their properties. This view offers also a debugging window in which the user can analyze the results of comparison and can optionally display the time chart of a task set over each scheduling policy. This helps the user to analyze and debug scheduling policies. It can be also useful to find counter examples and to isolate degenerate cases. For example, in the case of energy-aware scheduling, no optimal algorithm exists yet. In order to test empirically whether a new algorithm is potentially optimal or not, a simple approach consists in simulating the scheduling of all possibilities of task sets and ask the simulator to filter cases where the new scheduling algorithm fails whereas other heuristics succeed.

8.5 Architecture

Usually, the common concern of real-time simulation tools is the possibility of using the simulator in different contexts. In many cases, extending a simulator to include customized modules needs a lot of modifications and refactoring of the code. Careless modifications can lead to incompatibility with the original modules of the tool, or worst, it may lead to a different or wrong simulation results. Furthermore, most of the time, researchers do not like to spend much time or effort to understand and preserve an existing tool, and finally, the tool becomes difficult to maintain. In our team, we experienced this issue previously, and we noticed the importance of design and software architecture. The more generic and flexible the software is, the less time and effort we spend to integrate new customized modules. For this reason and in order to meet the requirements mentioned in Section 8.3 on page 206, we decided to merge the old versions of our simulator and to provide a new tool with an enhanced architecture and

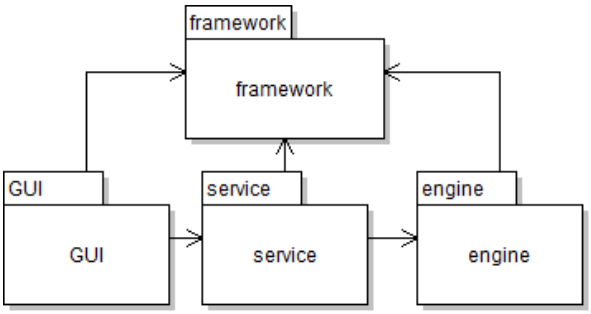


Figure 8.3: Modules UML diagram

design. We have ensured that the design is as generic and open as possible by applying the appropriate design patterns and programming paradigms.

As shown in Figure 8.3, the simulation tool is divided into four main modules each with a specific responsibility:

- the *engine* (core) module of the simulator which contains all the classes necessary to simulate the scheduling of real-time systems,
- the service module responsible of handling the transparent interactions between the engine and the presentation,
- the *GUI* module which contains the graphical components and classes,
- the *framework* module which contains useful tools and classes necessary for the application.

This module separation follows the classical **M**odel-**V**iew-**C**ontroller (MVC) design pattern (see Figure 8.3) that allows a secure isolation of the business part of the application from its presentation, and thus, it allows the engine module to be more generic and easily usable by other applications as an API. Furthermore, splitting the code into several modules limits the communication between the different modules to specific classes or interfaces by respecting the **I**nversion **o**f **C**ontrol (IoC) design pattern. The main advantage of IoC is to prevent inter-dependency between modules and to facilitate their maintenance.

8.5.1 Engine Module

The aim of a real-time simulator is to imitate the execution behavior of systems while respecting the hypotheses. Real-time systems are usually composed of a platform (i.e. processors, memories and caches), an energy profile (i.e., supply, consumption, heating, etc), a real-time task set and a scheduling policy. Many types of these components are studied in research works and are simulated and compared. The architecture of the

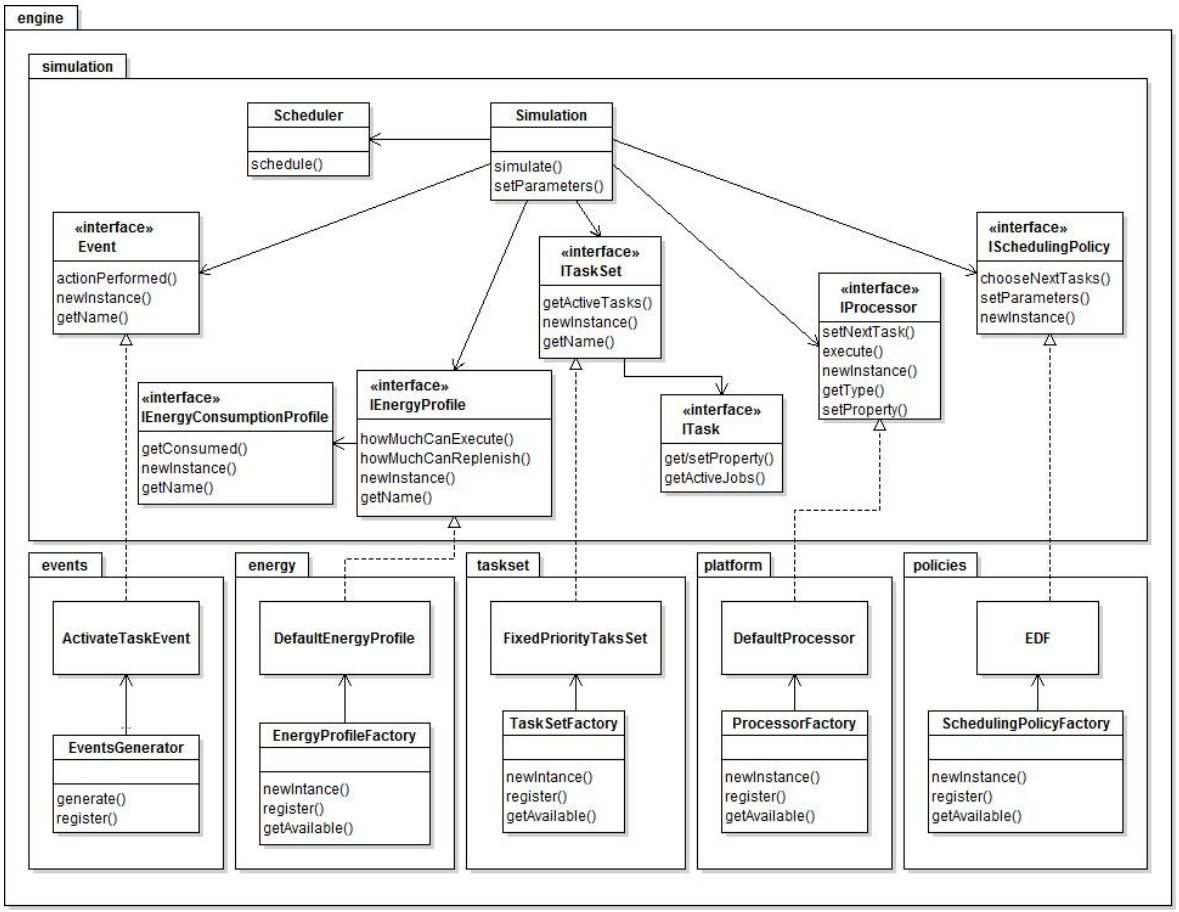


Figure 8.4: The engine module UML diagram

simulator must take into account the need to integrate new customized components, and then be used for the comparison process using the same simulator.

In *YARTISS*, the concept of each component is defined with a java interface that describes its responsibility, its input and its output as detailed in Figure 8.4.

Task set: Firstly, a task is defined mainly by the interface `ITask` that describes, with getters and setters, the most common parameters of real-time tasks. A task is generally characterized by a deadline, a period, a WCET and energy consumption. Other parameters can be added based on the desired model by inheriting classes from the basic interface. As a result, several task models can be implemented from the same interface with different parameters, such as periodic and sporadic.

Secondly, a task set is defined in *YARTISS* as a collection of tasks sorted according to a static or dynamic priority assignment following the adopted model. This aims to provide a sorted subset of active tasks at any time to be used by scheduling policies. It can be seen as a container of tasks that manages the dependencies between the tasks (as

in the case of DAG tasks) and provides a task ordering according to the specifications of the user (fixed job or task priority and dynamic priority). The combination of the inherited classes of tasks and task sets defines the task set model to be simulated. The user can implement a customized behavior of tasks and task sets by extending these interfaces and adding his own properties by using the generic available getters and setters.

The current release of *YARTISS* contains by default the following task set models: Liu and Layland model (i.e., periodic fixed-priority task sets), the dependent tasks of the DAG model and the energy-harvesting task model.

Scheduling policy: The responsibility of a scheduling policy is to decide at any time t which task or job to execute on which processor. *YARTISS* defines this behavior with the interface `ISchedulingPolicy`. It has a main method called `chooseNextTasks()` that selects the highest priority tasks at time t, to execute on the processors of the system. This method receives the necessary components for simulation such as the state of the processors, the task set and the energy profile. The scheduling policy can use external and generic parameters to calibrate its behavior with the method `setParameters()`. A scheduling algorithm is usually adapted to a specific task set model. For this reason, we give to the scheduling policies the responsibility of creating new instances of the adapted type of task model with the method `createTaskset()`. By outsourcing the scheduling decisions from the core of the `Simulation` class, we are able to implement several scheduling algorithms for different task models, which simplifies the comparison process. We have already implemented many scheduling policies in *YARTISS* for different task models, including fixed-priority *RM*, *DM*, *EDF*, *LLF*, etc. Also, there are algorithms for energy-harvesting systems and DAG task model.

Energy profile: The energy profile of a system includes the profiles of energy, production and consumption, and heating models. The interface `IEnergyProfile` describes how the system is supplied with energy and how it manages its consumption. It offers a set of methods that demonstrates the energy source (battery for instance), its limits and its energy levels within a time interval. The method `howLongCanExecute()` calculates the maximum time at which a task can execute w.r.t. the available energy and the used consumption profile. This can be used to simulate different energy sources, different rechargeable batteries with different environments and energy sources. The second part of the energy profile is the energy consumption model. *YARTISS* proposes two modes, the global mode in which all the tasks have the same energy consumption model, and the non-global mode which allows the tasks to consume energy based on their own consumption model. According to this, the simulator can be general and specific at the same time. The interface `IEnergyConsumptionProfile` describes how tasks consume energy with the method `getConsumed()`. This method gives the amount of energy to consume for a given execution time. The thermal behavior is also defined by this

interface. It consists of providing the heating and the cooling functions that can be general or task or processor specific.

Execution platform: A system platform consists mainly of a number of processors on which the tasks execute their code. The interface `IProcessor` defines the common behavior of processors. A task is allocated to a processor within a time interval and the method `execute()` performs the execution of the tasks within this interval by updating the state of the tasks. The current release of *YARTISS* proposes only one default type of multiprocessor which is the homogeneous mono-speed processors. However, it is possible to add new types of processors with different speeds and, consequently, the method `execute()` has to be modified according to the new characteristics. An example of the different type of processors is the processors with **D**ynamic **V**oltage and **F**requency **S**caling (DVFS) capabilities.

The Simulation Process

The aim of this tool is to simulate the scheduling of a system according to the parameters and the assumptions set by the user, namely the task set, the platform, the energy profile and the scheduling policy. According to the scheduler, the scheduling decisions are taken based on specific events, such as activation and finish times of a job, its deadline and preemption events.

A scheduling decision consists of assigning priorities to the active jobs statically or dynamically according to the scheduling policy, and allocating them to processors by considering the state of the system (energy state for example). The simulator uses a basic set of events that guarantee a correct execution of the system such as job request events, finish executing events, deadline events, etc. These events are natively implemented in *YARTISS*. The simulation starts generating the events responsible of requesting the first job of each task. Then other events are generated dynamically during the simulation process. For example, when a job meets its deadline, the corresponding deadline event is generated in order to check if the job has met its deadline and to request an other event for the next job as well. The events are generated sorted according to their temporal order and priority ordering. For each event, the scheduling policy is called to select the highest priority jobs to execute on the available processors. Then, the selected jobs execute while respecting the type of processors and the energy profile. This mechanism is implemented in class `Simulation`.

The interface `Event` describes the role of an event. The current release of *YARTISS* provides the necessary events to schedule a real-time system. However, the user is allowed to add his own events by implementing their customized behavior. To do so, the user should extend the `Event` interface and register the new event with the method `register()` from class `EventGenerator`. Furthermore, to generate an event, the user should use the method `generate()` of class `EventGenerator`. Then the event will be processed with the desired behavior which is implemented by the user.

Metrics and Statistics: The aim of a simulation is not restricted to checking the feasibility of a given system. It is important also to compute some metrics and statistics of the performed simulation in order to analyze the performance of the tested solution. We considered this functionality in *YARTISS* and we integrated a generic mechanism dedicated to compute statistics. It is possible to compute a metric at the beginning or the end of the simulation, or even during events processing. Moreover, it can be aggregated with other values when several simulations are performed. In the simulator, a metric is represented by the interface `IStatisticCriterion` which has a method for each possible calculation time (i.e., beginning, end, event processing). Several metrics are implemented natively and, as for the other parts, the user can add new customized metrics to the simulator by implementing the interface and injecting an instance of the new metric to the metrics manager class. At the beginning of each simulation, an instance of each active metric is created. Then, at each metric calculation time, the corresponding method of the metric is called. At the end of the simulation we can get the final value of each metric using the method `getValue()`. In the case of several simulations, we can also aggregate the values of the same metric on all of the simulations in order to compute the maximum/minimum values, the average value, etc. The aggregation function is implemented by the metric class and can provide several values: maximum, minimum, average values, percentage value, etc. By doing so, the simulation class computes several statistics independently from their implementation. Among the implemented metrics available in *YARTISS*, we can cite: deadline miss counter, average busy and idle periods, average overheads.

Inversion of Control and Dependency Injection

Based on the above described components, we notice that the simulator deals only with interfaces and never directly with implemented classes. This allows the simulator to be generic regarding the parts that can be replaced, such as the scheduling policy, the task set model, the platform, the energy profile and the statistics metrics. This implantation respects the IoC design pattern and the Dependency Injection that stipulates that the objects composing the application must be weakly coupled and dependencies should be linked only at run-time.

The IoC design pattern is a design pattern of software construction where reusable code controls the execution of problem-specific code. Its main advantage is that the reusable code is developed independently from the problem-specific code, which often results in a single integrated application. The IoC design guideline is useful for the following purposes:

- the execution of a certain task is independent from its implementation,
- every part of the system focuses on its design purpose,
- all parts have no assumptions about the constraints and the limitations of the other parts,

- replacing or modifying a part in the system does not affect the rest of the parts.

Sometimes, IoC is referred to as the *Hollywood Principle*: *Don't call us, we'll call you*, because program logic runs against abstractions such as callbacks.

In the case of *YARTISS*, the simulation code is a reusable part and the different implementations of scheduling policies, processors, task sets or metrics are done on specific parts that can be replaced and developed independently. This is what makes *YARTISS* a generic real-time simulator that can be used easily by other researchers.

8.5.2 Service Module

The engine module contains all the code necessary to build and run a simulation. Furthermore, it can be used alone or imported as an API to be integrated into another software. However, we were interested in providing a complete simulation tool and not only an API. So we implemented more classes which allow an intermediate user to build and run simulations easily. This is done using the *Service module* which contains the functionalities described above in Section 8.4 on page 207. From a design point of view, the service module represents the Controller and the Model parts of the famous MVC design pattern or the View-Model part of the **M**odel **V**iew **V**iew-**M**odel (MVVM) design pattern. It is a software brick of higher level that uses the engine module. The service module provides the classes that set the replaceable parts of a simulation (e.g. the scheduling policy, the task set, the platform, metrics, etc), run the simulation and save the results. It contains mainly three public classes, each represents a certain functionality:

- The `TimeLineViewModel` class provides the necessary parameters to simulate a task and allows the user to run the simulation and visualize the results.

- The `BenchmarkViewModel` class allows the user to set quickly the description of a large scale simulation where several scheduling policies are compared using the chosen statistic metrics in different scenarios of energy profiles. This class helps the user to build such a simulation without knowing the implementation of any parts. The description is passed using a `String` object and all created generic objects are done using the *Dependency Injection* paradigm. This class saves results which are readable by the chart plotter tool *Gnuplot*.

- The `task GenViewModel` class builds a custom task set generator in the same way.

These classes are packed in an independent module so as to be built easily using a higher level software layer with a friendly user interface (such as a web server or a graphical or textual interface).

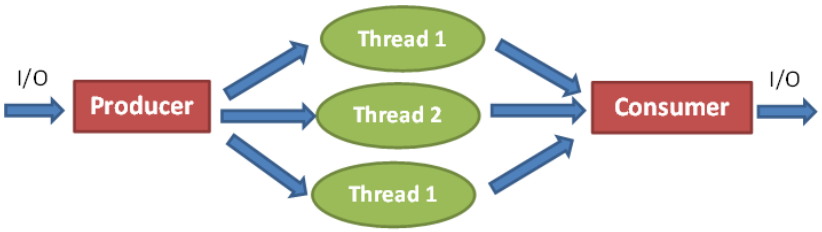


Figure 8.5: Producer/Consumer design pattern

8.5.3 Framework Module

This is a toolbox module that contains generic classes and functions that facilitate building simulations. It is completely independent from the other modules and can be reused by other projects easily. This module is considered as a small API that provides a simple and small abstraction of the Java concurrency API and some generic classes for the MVC design pattern :

Concurrency: This feature allows the user to run several Java `Runnables` objects simultaneously with the possibility of collecting results. It implements the *producer/-consumer* design pattern to get the final results. It aims to accelerate simulations by leveraging the concurrency API of Java. Furthermore, running several computations in parallel can be done directly by using Java threads. However, getting and computing the final results of simulation is difficult due to threads concurrency. For this reason, we adopted the *producer/consumer* design pattern such that the threads perform the required computations and produce the results in parallel. On the other hand, a single consumer is allowed to collect and aggregate the results with a *thread safe mode* by using a blocking queue. The service module uses this functionality to run large scale simulations in parallel to collect aggregated results in order to present them through the user interface. It can also use the task set generator. Figure 8.5 illustrates the producer/consumer design pattern.

Model-View-Controller (MVC): This part provides a simple framework for building MVC applications. It is not as sophisticated as commercial frameworks, but it is sufficient for the simulator needs. Mainly, it contains classes that use the Java reflection capabilities to handle getter and setter properties. This enhances the generic aspect of the architecture and keeps a weak coupling between objects and modules.

8.5.4 View-Module or GUI

This module is responsible of building and running a graphical user interface that facilitates the use of the different features and functionalities of the simulator that are described in section 8.4 on page 207. It represents the highest level of the application

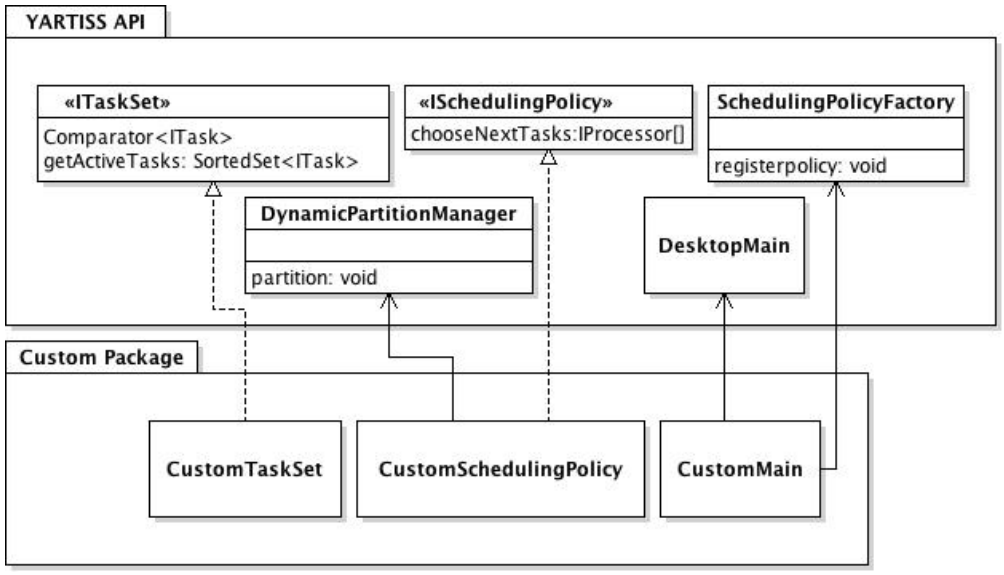


Figure 8.6: UML diagram describing the addition of a new scheduling policy

and it proposes some independent graphical components that can be used to build a customized graphical interface (components to visualize the time chart of a simulation, to visualize the energy or the battery evolution chart). The views provided by this module are linked to the other models and controllers of the service model only at run-time.

8.6 Case Study

As described above, the particularity of *YARTISS* is its genericity. This means that its parts can be extended easily by users. They can develop new customized parts and use them in the simulating tool without need to understand the implementation of simulator core.

In this section we demonstrate the generality of *YARTISS* by adding a new scheduling policy, The **P**reemptive **F**ixed-Task-**P**riority **A**s **S**oon **A**s **P**ossible (PFP_{ASAP}) scheduling algorithm presented in Chapter 4 on page 109.

8.6.1 Adding a new Scheduling Policy

As mentioned in Section 8.5.1 on page 216, to add a new scheduling policy, the user have to import the modules of *YARTISS* and to implement the interface `ISchedulingPolicy` that defines the concept of a scheduling algorithm.

The implementation of the interface `ISchedulingPolicy` means the implementation of all methods. The most important method is `chooseNextTasks( Platform platform, ITaskSet taskSet, IEnergyProfile energyProfile, long t, EventGenerator gen`

`)` that assigns tasks to the available processors at any time t. The parameter `platform` represents the hardware part, i.e. processors and memories, etc. The parameter `taskSet` is the real-time task set to simulate, `energyProfile` is the energy profile chosen by the user and `evGen` is the event generator that can be used to generate additional events. We notice that except the task set, all of the parameters are typed with interfaces which means that the scheduling policy deals with generic parameters as illustrated by the UML diagram of Figure 8.6 on the previous page. The real types are instantiated at run-time following the configuration set by the user. For this reason the implementation of a new scheduling algorithm is easy and the new class contains only the code of the scheduling algorithm.

Therefore, knowing that PFP_{ASAP} schedules tasks as soon as possible by consuming greedily the available energy, the method `chooseNextTasks(...)` can be implemented by writing in Java the following instructions:

1. check if there is at least one active task at time t,
2. get the task with the highest priority among the active tasks,
3. ask the energy profile how long this task can be executed continuously,
4. do not execute the task if the energy is not sufficient to execute by assigning `null` to the processor and generating a new event at time $t + 1$ to retry again,
5. execute the remaining time of the highest priority active task by assigning the task to the processor with `platform.getProcessor().setNextTask(task)`, and generating an other event at the time when the energy become insufficient to execute all the remaining execution time of the active task.

The second important method is `createTaskSet()`. It creates an instance of the task set model that is compatible with the current policy. In the case of PFP_{ASAP}, the suitable task set is the classical fixed-priority one, i.e. Liu and Layland model, where the tasks are sorted according to their static priorities. The task set models follows the same design pattern as scheduling policies but in this case we use an already existing model (with `TaskSetFactory.getNewInstance( "priority")`) instead of implementing a new one.

The third important method is `newInstance()` which is a factory method that creates new instances of the current scheduling policy, name PFP_{ASAP} in this case. This method is used to create new objects of the scheduling policy class without requiring to do it manually with `new` constructors. This design patter allows us to centralize the creation of objects of classes that implement the same interface which allows a generic injection of new implementations, e.g. new scheduling algorithms in this case.

The remaining methods are used to identify the policy among others and to provide parameters if necessary (which is not the case of PFP_{ASAP}).

Then, the newly-created scheduling policy should be registered in the simulator by the use of the `register()` method from the `SchedulingPolicyFactory` class that centralize the creation of classes implementing `ISchedulingPolicy` interface. This method injects a sample object of the policy we want to register, then using the methods `newInstance()` and `getPolicyName()`, the new scheduling policy can appear in the list of the available scheduling policies and can be instantiated generically. The call of the method `register()` should be done in the customized main class created by the user or by using the GUI of *YARTISS* as shown in Listing 1 on page 251 in the Appendix.

The adopted design allows us to implement other scheduling policy easily, we did the same with all the policies discussed in this dissertation.

8.6.2 Adding an Energy Profile

The same methodology can be applied if we want to add a new energy consumption profile to *YARTISS*. Listing 2 on page 252 in the Appendix shows the Java code needed to use a logarithmic energy consumption profile as an external module.

8.7 Distribution

The project is available on the collaborative development platform hosted at `https://svnigm.univ-mlv.fr/projects/YARTISS/`. This environment provides a subversion repository allowing anonymous checkouts, documentation hosting, RSS feeds subscriptions, and public forums. A web page dedicated to *YARTISS* is also available at `http://YARTISS.univ-mlv.fr`. In addition to a general presentation of the tool, it proposes a demo applet version which allows interested readers to try *YARTISS* directly from their web browser and an application form to allow anybody to share external modules.

8.8 Conclusion

In this paper we presented *YARTISS*, a real-time multiprocessor scheduling simulator. A consequent effort has been made to make it as extensible as possible. To justify the need for an open and generic tool, we presented the history of *YARTISS* development. Then we briefly presented existing simulation tools. We have described the three main functionalities of *YARTISS*: 1) simulate a task set on one or several processors while monitoring the system energy consumption, 2) concurrently simulate a large number of task sets and present the results in a user friendly way that permits us to isolate interesting cases, and 3) randomly generate a large number of task sets. Then, in order to demonstrate the modularity and extensibility of our tool, we presented its architecture and a case study that shows how to add new parts, in most cases without

having to open or modify the core modules. Finally, we gave the instructions to get and test *YARTISS*. We hope that this software can contribute to build a standard simulation tool that can be adopted through the real-time scheduling community.

General Conclusion

Scope

In this thesis we addressed the real-time scheduling problem of fixed-priority energy-harvesting systems. A real-time energy harvesting system must respect not only the deadlines of all tasks but also their energy constraints. This means that the energy cost of all jobs must be satisfied by the energy stored in the storage unit and the energy collected from the environment. To satisfy tasks energy cost, additional idle times may be necessary to collect the required energy. These necessary delays create a new scheduling problematic. Now, the challenge for a fixed-priority scheduling algorithm is to execute tasks according to their deadlines and to delay executions at the right moments and for the right durations in order to meet all deadlines and satisfy the energy demand.

The problems are: first to find an optimal scheduling for such a model; second to provide schedulability conditions that help designers to check if a given task set is schedulablable or not; third, to find the minimum battery capacity that ensure the feasibility of a given task set.

In this dissertation, we started by presenting the state of the art of this research area. We first reminded briefly the classical real-time scheduling theory including the different task models and the main scheduling algorithm families. Second, we explored the state of the art of energy-harvesting hardware, namely the different technologies of energy-harvesting and energy storage and the applications for which they are suitable. Third, we presented the main scheduling algorithms proposed in the literature and their properties. We showed also that the fixed-priority scheduling problematic for energy-harvesting systems is not very well studied by the community.

After that, we presented in detail the contributions of this thesis, summarized in the next section.

Main Results

The PFP_{ASAP} Algorithm

The first result of this thesis is the PFP_{ASAP} algorithm which is an adaptation of the classical FTP Algorithm. It schedules tasks as soon as possible by executing whenever the energy available in the storage unit is sufficient to execute at least one time unit, and by replenishing otherwise but only as long as needed to replenish the energy necessary to execute one time unit of the job with the highest priority. In other words, it is a fixed-priority energy-work-conserving algorithm. We proved that this algorithm is optimal for energy-non-concrete task sets where the initial battery

capacity and task offsets are known only at run-time. This optimality relies first on the fact that all tasks consume more energy than what it is possible to replenish during their execution time, and second, on the fact that in this case the schedulability of the task set can be studied only for the first instance of each task in the worst-case scenario which is the simultaneous release of all the tasks with the minimum battery level. A necessary and sufficient schedulability test was also proposed for this family of task sets. Furthermore, we proposed a bound of the minimum battery capacity that preserves the schedulability of a task set with PFP_{ASAP} algorithm. This capacity is equal to the maximum instantaneous energy consumption of task. Moreover, we compared PFP_{ASAP} to the algorithms presented in the state of the art with simulations. These later confirmed the theoretical results and showed in one hand the advantages of PFP_{ASAP} which are its optimility and its low overhead, and in an other hand its drawbacks which are mainly the high rate of preemptions and context switches.

Schedulability Analysis for PFP_{ASAP} Algorithm

The second result is a schedulability analysis of PFP_{ASAP} that removes some restrictive assumptions. As mentioned above, the optimality of PFP_{ASAP} relies on the fact that the task sets are energy-non-concrete and that the consumption rate of all the tasks is greater than the replenishment rate. If we remove one of these assumptions, PFP_{ASAP} is no longer optimal, the simultaneous request of all the tasks with the minimum battery level is no longer the worst-case scenario and the proposed schedulability test becomes only necessary.

To cope with this problem, we proposed a schedulability analysis using tasks worst-case response time upper bounds. For this analysis we relaxed the assumption of tasks consumption, offsets and initial battery level. Then, we can check with a sufficient test the schedulability of task sets that can include tasks with a consumption rate lesser than the replenishment, i.e. gainer tasks, for any scenario. The idea behind the first upper bound is to maximize the number of interferences and to assume that all consuming jobs are scheduled before gaining jobs, which maximizes the delays due to replenishments and the delays due to interferences. The second upper bound keeps the same idea but by respecting jobs request time and deadlines which tighten the upper bound. The schedulability test consists of computing an upper bound of the longest response time of each task and to compare it to its deadline. We proved also that the proposed schedulability tests are compatible with the optimal priority assignment algorithm of Audsley. We performed some simulations to evaluate the validity of the theoretical results and to measure the tightness of the upper bounds comparing to the empirical worst-case response time. The experiments showed that the second upper bound is tighter and that its deviation is reasonable.

Schedulability Analysis for Thermal-aware Systems

We showed in Chapter 6 the difficulty of finding an optimal scheduling algorithm and exact feasibility conditions for the general model of fixed-priority energy-harvesting systems. To cope with this problem, we decided to address the problem from an other point of view by studying other real-time scheduling problems that might present some similarities especially the ones where some delays are needed to produce a valid schedule. Then, we started studying the real-time scheduling for thermal-aware systems where the temperature of the system must be managed such that it never exceeds a certain maximum threshold. To respect this constraint, it may be necessary to delay executions in order to cool down the system reactively when reaching the maximum temperature or proactively before reaching it.

As a first step, we started by studying the behavior of the equivalent of PFP_{ASAP} algorithm for thermal-aware systems. We proved that PFP_{ASAP} is also optimal for thermal-non-concrete task sets where tasks offsets and the initial temperature are known only at run-time. In this model, PFP_{ASAP} schedules jobs whenever the system is cold enough to execute at least one time unit, and otherwise, it delays executions in order to cool down enough to execute one time unit. In this model the worst-case scenario occurs for a task occurs when all the higher priority tasks are requested simultaneously while the temperature is at its maximum level. However, computing the exact response times is difficult to express with a formula or an algorithm. For this reason, we proposed two sufficient schedulability conditions that uses tasks worst-case response time approximation. The idea of the first bound is to maximize the cooling periods in the worst-case scenario by leveraging the fact that the cooling function is exponentially decreasing which leads to long cooling periods. The second bound is parametric, it considers that the cooling periods are of the same length x and the corresponding execution periods are integers (floor value). The performed simulations showed that the first upper bound is very pessimistic, however, the second one is very tight to the empirical worst-case response time when $x = 1$. This result is the first step of the exploration of similarities between energy-harvesting and thermal-aware systems. More work is planned to try to adapt some results from the thermal-aware side to the energy-harvesting one.

Simulation tool

The fourth main result of this thesis is the development of YARTISS the simulation tool used to perform all the experiments of this thesis and the ongoing work of our team. YARTISS is a modular and Java-based real-time simulator that provides a generic simulation framework and a friendly graphical user interface. This simulator was designed such that the major aspects of real-time scheduling can be substituted. The concept of each part, e.g. scheduling algorithm, task, task set, energy consumption, etc, is modeled by a java interface which facilitates the substitution of implementations. YARTISS provides also a graphic user interface that allows the visualization of the

schedule of a given task set according to a given scheduling algorithm in a given environment of energy. Moreover, it can be also used to run large scale simulations which is very useful to evaluate and validate theoretical results.

Future Research

The work done along this thesis helped us to better understand the problematic of fixed-priority real-time scheduling for energy-harvesting systems. Moreover, the results of this thesis contribute to cover a part of the not yet studied research areas of this field.

As any research work, the achievement of a project is the starting point of an other one, and each answer arises more questions. In the case of this thesis, the following points and open questions deserve more investigations.

Optimal Algorithm

The main question that is still without answer is the existence of an optimal algorithm for the general model of energy-harvesting systems that includes all energy consumption types, i.e. gaining and consuming tasks, concrete task sets, i.e. without any assumption on tasks offsets and the initial battery level. We showed in Chapter 6 that many intuitive ideas do not work to build an optimal algorithm. The only idea that could lead to an optimal algorithm is to combine virtual deadlines with lookahead computation to compute the optimal delay at any moment. However, the complexity of this algorithm is exponential. The question now is: does an optimal algorithm for fixed-priority energy-harvesting systems exist ? or is the fixed-priority scheduling problem of energy-harvesting systems an NP-hard problem ?

We tried to answer both of these two questions but we was not able neither to find an optimal algorithm nor to prove the NP-hardness of the problem. Therefore, this is still an open question that deserves more interest and effort to answer.

Energy Assumptions

One of the interesting future axes is to study the problem in more realistic assumptions even with non optimal solutions. In fact, the results presented in this thesis assume that the replenishment and tasks consumption are constant functions. Even though there exist some applications where these assumptions are not far to be realistic, they are still marginal. For example, with energy sources like solar energy, the harvested energy varies over time which might make the proposed solutions very optimistic and not realistic. It might be useful to provide energy-source-specific scheduling algorithms and schedulability analysis in order to build real efficient applications. Moreover, the energy consumption of tasks may also vary over time due to the use of different circuitry or due to the thermal state of the system. This problem should be addressed by either

taking into account the real consumption profile or to find the worst consumption profile of each task.

Similarities With Thermal-Aware Systems

This point is as important as the precedent ones because more work has been done for thermal-aware systems than energy-harvesting systems. Many ideas that have been proposed for the thermal model might be applicable for energy-harvesting systems especially for more realistic assumptions of energy replenishment and consumption. The results presented in Chapter 7 represent the first step in the exploration of such ideas.

Simulation Tool

YARTISS is still under development and requires more time and effort to improve its architecture and to add implementations of more algorithms task models, etc. The simulator is used by our team and few students around the world. We aim to form a dedicated development team that can maintain its code and do more communications about it in order to attract more interest and more contributions.

Part III

Résumé en français

Chapter 9

Ordonnancement temps réel avec gestion de l'énergie renouvelable

9.1 Introduction

L'informatique est l'une des sciences qui ont révolutionné le monde durant le dernier siècle. L'homme a inventé les ordinateurs électroniques durant la deuxième guerre mondiale. Ces derniers étaient capables d'effectuer des opérations de cryptographie en utilisant le modèle de la Machine de Turing. Depuis, ces ordinateurs n'ont cessé d'évoluer.

Pendant les dernières décennies, l'utilisation des ordinateurs dans la vie moderne a fortement augmenté et a fait exploser la demande en termes de puissance de calcul et d'efficacité. Le progrès de la technologie a permis aux ordinateurs modernes d'être beaucoup plus petits et plus performant que les premières machines. Les ordinateur simples sont devenus tellement petits qu'il est aujourd'hui possible d'avoir des petits appareils mobiles qui peuvent fonctionner avec très peu d'énergie, ce qui a permis à de nouvelles applications d'émerger. De nos jours, les ordinateurs sont aussi utilisés pour contrôler des opérations critiques telles que les réactions nucléaires ou les commandes de vol d'un avion. Ce type d'ordinateurs est souvent utilisé dans les systèmes embarqués autonomes qui vont du simple lecteur MP3 aux très complexes navettes spatiales et des jouets d'enfants aux robots industriels.

Dans ce genre d'applications, le temps que les tâches du système prennent pour délivrer les résultats est aussi important que leur exactitude. Les systèmes qui sont soumis à des contraintes temporelles sont les systèmes dits temps réel.

De plus, avec l'augmentation de la vitesse des processeurs et la miniaturisation des appareils électroniques, la gestion de l'énergie est devenue l'une de problématiques importantes à étudier. La mission des petits appareils est de fournir un service de manière autonome et cela pour une longue durée de vie comme par exemple dans les réseaux de capteurs ou les implants médicaux. Le défi ici est de pouvoir utiliser l'énergie ambiante pour permettre une longue durée de vie en éliminant les opérations de maintenance. Plusieurs sources d'énergie peuvent être exploitées en fonction des besoins de l'application, par exemple, les énergies solaires, vibratoires ou thermiques.

Dans cette thèse, nous nous intéressons à la problématique de l'ordonnancement temps réel de ce type de systèmes.

Durant la dernière décennie, le comportement temps réel de ce genre de systèmes a attiré plus d'attention. Le défi ici est de trouver le bon ordonnancement des tâches tout en respectant les contraintes énergétiques et temporelles. Quelques algorithmes d'ordonnancement ont été proposés dans la littérature mais la majorité des travaux sont concentrés sur l'ordonnancement à priorité dynamique du type EDF. L'ordonnancement à priorité fixe a quant à lui été très peu traité alors qu'il est largement utilisé en industrie. Cette thèse traite la problématique d'ordonnancement pour cette famille d'algorithmes.

Nous proposons d'abord une solution partielle qui consiste en un algorithme optimal pour une sous catégorie des systèmes étudiés, ainsi que l'analyse d'ordonnançabilité associée, puis nous expliquons la difficulté de la généralisation.

Le reste de ce résumé est organisé comme suit. Un bref aperçu de l'état de l'art de l'ordonnancement temps réel des systèmes collecteurs d'énergie est exposé dans les sections 9.2, 9.3, et 9.4. La section 9.5 introduit l'algorithme PFP_{ASAP} qui est la première contribution de cette thèse. La section 9.6 présente une analyse approximative de temps réponse des tâches avec PFP_{ASAP} en relâchant quelques hypothèses. La section 9.7 explique la difficulté de trouver un algorithme optimal pour notre modèle. Dans la section 9.8 nous discutons la similitude entre le modèle des systèmes collecteurs d'énergie et celui des systèmes contraints par la chaleur. La section 9.9 présente brièvement l'outil de simulation YARTISS développé pendant cette thèse. Pour terminer, la section 9.10 conclut ce résumé.

9.2 Ordonnancement temps réel

En informatique, un système est considéré comme étant temps réel si le respect de ses contraintes temporelles est aussi important que l'exactitude des résultats [But11]. Le terme temps réel n'est pas forcément associé à la rapidité. L'objectif d'un système temps réel correct est de respecter perpétuellement les échéances individuelles des tâches dans le pire cas même si l'exécution est lente. Généralement, les systèmes temps réel sont classifiés en fonction de la gravité des conséquences des dépassements d'échéances. Nous distinguons alors deux familles : *les systèmes temps réel durs* et *les systèmes réel souples*. Dans la première famille, aucun dépassement n'est toléré car les dommages occasionnés sont catastrophiques, par exemple, la perte des commandes de vol d'un avion. Par contre, les dépassements d'échéances dans la deuxième famille ne font que dégrader la qualité du service, par exemple, la perte de quelques images lors du décodage d'un flux vidéo en direct.

9.2.1 Modèle de tâches

Un système temps réel est généralement composé d'un ensemble de tâches temps réel récurrentes noté Γ. Chaque tâche τ_i est caractérisée par son pire temps de réponse C_i, par son échéance temporelle relative D_i, par sa période T_i et sa première date d'activation O_i. Chaque tâche se réactive au moins toutes les T_i unités de temps pour

exécuter ce que l'on appelle un *travail* ou un *job*. Un travail $J_{i,j}$ est caractérisé par sa date d'activation $a_{i,j}$ et son échéance absolue $d_{i,j}$.

Ces paramètres peuvent êtres spécifiés avant le lancement du système ou uniquement au moment de l'exécution. De ce fait, nous distinguons deux familles de systèmes : les systèmes complètement spécifiés et les systèmes partiellement spécifiés. Dans la première catégorie, tous les paramètres sont connus à l'avance, ainsi le système est dit concret. Dans cette famille, nous trouvons les systèmes de tâches périodiques qui se réactivent à un intervalle régulier de temps. Dans la deuxième famille, au moins un paramètres n'est connu qu'au moment de l'exécution, par exemple les systèmes de tâches sporadiques où seul le temps minimal d'inter-arrivée est connue, ou encore les systèmes apériodiques où modèle de récurrence n'est pas connue du tout.

9.2.2 Algorithmes d'ordonnancement

Les tâches du système sont exécutées de manière concurrente sur le processeur. De ce fait, il est nécessaire de spécifier une politique pour choisir quelle tâche exécuter à tout moment. C'est le rôle de *l'algorithme d'ordonnancement*. Dans la théorie de l'ordonnancement temps réel, nous distinguons distingue trois familles d'algorithmes : les algorithmes à priorité de tâche fixe FTP, ceux à priorité de job fixe FJP et ceux à priorité dynamique DP. Dans la première famille, la tâche se voit attribuer une priorité fixe tout au long de son cycle de vie. Ainsi, l'algorithme exécute en premier la tâche active avec la priorité la plus élevée. Dans la deuxième famille, les priorités sont données aux jobs et non aux tâches, par exemple, l'ordonnancement EDF qui attribue les priorités aux jobs en fonction de leur échéance temporelle (le job avec l'échéance la plus proche est exécuté en premier). Dans la troisième famille, les priorités sont attribuées dynamiquement et peuvent changer d'un unité de temps à une autre, l'exemple typique de cette famille est l'algorithme LLF.

9.2.3 Analyses d'ordonnançabilité

Pour s'assurer qu'un système de tâches donné est faisable ou ordonnançable avec un algorithme spécifique, nous avons besoin d'outils d'analyse qui nous permettent de statuer sur le respect des échéances de toutes les tâches y compris dans le pire cas. Dans la théorie classique des systèmes temps réel, nous trouvons des conditions d'ordonnançabilité qui sont basées principalement sur la charge processeur du système ($U = \sum C_i/T_i$) ou sur une analyse du temps réponse des tâches dans le pire scénario. Pour la priorité de tâche fixe, la borne de Liu et Layland constitue une condition d'ordonnançabilité suffisante ($U \leq n(2^{1/n} - 1)$) dans le cas des échéances sur requête ($\forall_i T_i = D_i$) tandis que la condition de charge maximale ($U \leq 1$) est une condition nécessaire. Pour EDF, la condition sur la charge maximale est nécessaire et suffisante dans le cas des échéances sur requête ($\forall_i T_i = D_i$). D'autre conditions et analyses sont détaillées dans le chapitre 1.

Dans cette thèse, nous nous focalisons sur l'ordonnancement à priorité de tâche fixe sur des plateformes monoprocesseur.

9.3 Les systèmes collecteurs d'énergie

Un système collecteur d'énergie est un système qui récupère l'énergie ambiante et la stocke dans un réservoir d'énergie afin d'alimenter un appareil électronique.

Généralement, ce genre de système est composé de trois composants principaux : un collecteur d'énergie, un réservoir d'énergie et une unité de calcul.

Le collecteur d'énergie est responsable de la transformation de l'énergie ambiante en courant électrique.

Le réservoir d'énergie stocke temporairement l'énergie collectée par le collecteur. Généralement, les batteries rechargeables ou les super-condensateurs sont utilisés pour cet effet en fonction des besoins de l'application.

L'unité de calcul est généralement composée de capteurs de données, de processeurs et de composants de transmission. C'est ce composant héberge le système temps réel que nous étudions dans cette thèse.

9.3.1 Les technologies d'extraction de l'énergie ambiante

Plusieurs sources d'énergie peuvent être exploitées. Toutes les sources ne sont pas adaptées à toutes les applications. La source d'énergie et la méthode d'extraction doivent être soigneusement choisies pour répondre aux besoins de l'application.

Parmi les méthodes de conversion on trouve les suivantes :

- *Conversion de l'énergie solaire :* Cela passe par l'utilisation de cellules photovoltaïques. Cette technique présente un rendement assez faible (de l'ordre de 13%) et la quantité d'énergie produite est très fluctuante dans la journée et dans l'année.

- *Conversion de l'énergie des ondes radio :* Cette méthode présente un rendement faible mais très utile pour un usage ponctuel, par exemple, les puces **R**adio **F**requency **Id**entification (RFID).

- *Conversion électromagnétique de l'énergie vibratoire :* Cette technique utilise le champ magnétique pour transformer l'énergie mécanique des vibrations en courant électrique. Cette technique présente un rendement faible par rapport aux précédentes.

- *Conversion électrostatique de l'énergie vibratoire :* Cette technique utilise cette fois le champ électrostatique pour transformer l'énergie mécanique des vibrations. Le rendement de cette dernière est meilleur et peut être intéressant pour les applications qui subissent une fréquence de vibration importante.

- *Conversion piézoélectrique de l'énergie vibratoire :* Cette méthode utilise un matériau piézoélectrique qui crée une différence potentiel lors des mouvements produits par les vibrations. Cette technique est très prometteuse et peut être facilement utilisée dans des applications de surveillance de machines industrielles où le taux de génération d'énergie s'approche du constant.

- *Conversion de l'énergie thermoélectrique :* Cette technique utilise le différentiel de température pour faire circuler les électrons et par conséquent un courant électrique. Cela peut être efficace uniquement dans le cas où la différence de température est assez élevée.

Cette liste de méthode de conversion n'est pas exhaustive, d'autres techniques moins connues existent dans la littérature.

9.3.2 Les technologies de stockage d'énergie

Le stockage d'énergie est le domaine qui a le moins évolué durant les dernières décennies par rapport aux autres technologies. Cela n'a pas empêché l'utilisation de supports de stockage rechargeables dans les petits systèmes embarqués.

Différentes technologies de stockage ont été explorées et commercialisées. Parmi ces dernières, nous distinguons deux familles : les batteries rechargeables et les super-condensateurs.

Les batteries chimiques sont les plus répandues dans le marché. Cela est dû non seulement aux facilités de fabrication mais aussi à leurs caractéristiques de durée de vie, de capacité, de courant et de taux d'auto-déchargement. Les batteries chimiques les plus utilisées de nos jours sur des petits appareils électroniques sont : les batterie **N**ickel **C**admium (NiCd), **Ni**ckel-**M**etal **H**ydrid (NiMH), et **Li**thium **Ion** (Li-ion).

Les inconvénients des batteries chimiques sont leur taille et leur perte de capacité (effet mémoire) au fils des cycles de charge et décharge.

Ces problèmes peuvent être résolus par l'usage des super-condensateurs qui sont capables de stocker une quantité d'énergie suffisante pour un petit appareil en offrant un taux de chargement et de déchargement quasiment constant avec un nombre de cycles de charge/décharge très important.

Dans cette thèse, nous supposons que le collecteur d'énergie recharge le réservoir avec un taux constant et que les tâches peuvent consommer l'énergie avec un taux constant également. Cela peut être réalisé avec un collecteur du type piézoélectrique et un super-condensateur pour des applications de surveillance des machines industrielles.

9.4 L'ordonnancement temps réel dans les systèmes collecteurs d'Énergie

Dans cette section, nous nous concentrons sur la partie applicative en spécifiant le modèle général des systèmes temps réel collecteurs d'énergie visé par cette thèse. Nous présentons par la suite les différents algorithmes et approches d'ordonnancement proposés dans la littérature.

9.4.1 Modèle

Le modèle considéré ici est une extension du modèle de Liu et Layland décrit dans la section 9.2. L'extension consiste à intégrer les paramètres énergétiques au modèle, soit principalement la pire consommation d'énergie de la tâche notée E_i.

Le rechargement : la quantité d'énergie qui arrive dans le réservoir est une fonction du temps notée $P_r(t)$. Dans un premier temps et afin de simplifier le modèle, nous considérons que cette fonction est constante. Ainsi, nous utiliserons par la suite P_r à la place de $P_r(t)$ pour designer la fonction de rechargement constante.

Même si l'utilisation d'une fonction de rechargement constante parait irréaliste, il existe tout de même des sources d'énergie et des techniques d'extraction comme l'énergie vibratoire et la technique piézoélectrique qui permettent d'avoir un flux d'énergie proche du constant dans certaines applications industrielles (voir le chapitre 2).

Le stockage d'énergie : nous considérons un support de stockage idéal qui peut être chargé et déchargé avec des taux constants. Le niveau d'énergie dans le réservoir fluctue entre deux seuils, le niveau minimal E_{min} qui assure la continuité de fonctionnement du système et le niveau maximal E_{max} qui ne peut être dépassé. La capacité effective du réservoir pour notre système temps réel notée $\mathcal{C}$ est la différence entre E_{min} et E_{min}. Ces caractéristiques peuvent être satisfaites par les super-condensateurs qui permettent des taux de chargement et de déchargement constants et qui sont capables de supporter des cycles fréquents sans pour autant dégrader la capacité.

On utilisera par la suite le terme batterie pour désigner le réservoir d'énergie décrit dans cette section.

La consommation : pour ce travail nous considérons que les tâches consomment l'énergie de façon uniforme tout au long de leurs temps d'exécution. Nous considérons aussi que chaque tâche a son propre taux de consommation instantanée $Pc_i = E_i/C_i$ qui diffère d'une tâche à une autre, c'est à dire qu'il est possible d'avoir deux tâches τ_i et τ_j avec $C_i < C_j$ et $E_i > E_j$.

9.4.2 Problématique

Rappelons que cette thèse se focalise sur l'ordonnancement monoprocesseur à priorité de tâche fixe des systèmes temps réel. Les problèmes soulevés par cette famille d'ordonnancement peuvent être résumés par les questions suivantes.

1. **Les algorithmes** : Comment peut-on ordonnancer un ensemble de tâches à priorité fixe sur un seul processeur de telle sorte que les échéances de toutes les tâches soient respectées et que leur demande d'énergie soit satisfaite tout en gardant le niveau du réservoir entre E_{min} et E_{max} ?

2. **Les conditions d'ordonnançabilité** : Existe-t-il une condition nécessaire et suffisante pour vérifier l'ordonnançabilité d'un système de tâches donné ?

3. **La capacité minimale du réservoir** : Quelle est la capacité minimale de la batterie qui préserve l'ordonnançabilité d'un système donné ?

Nous apportons quelques éléments de réponse à ces questions au travers de cette thèse.

9.4.3 Approches d'ordonnancement

Dans la littérature on identifie principalement deux approches pour répondre au besoin temps réel des systèmes collecteurs d'énergie. La première consiste à minimiser la consommation d'énergie en réduisant la fréquence et/ou la tension du processeur [PBB98; Wei+94]. La deuxième consiste à gérer l'énergie disponible sans avoir à baisser la consommation. Cela passe par une sélection judicieuse des moments d'exécution et ceux de rechargement ou des périodes d'activité et celles d'inactivité afin de ne jamais tomber à court d'énergie et faire au mieux pour respecter toutes les échéances. Dans la suite de ce manuscrit, nous nous intéresseront uniquement à la deuxième approche en explorant les différents algorithmes proposés dans la littérature pour cet effet.

9.4.4 Les algorithmes existants

Cette section présente brièvement les principaux algorithmes disponibles dans l'état de l'art des systèmes temps réel collecteurs d'énergie.

Frame-Based Algorithm (FBA) : Il s'agit du premier algorithme proposé pour les systèmes collecteurs d'énergie. Il est spécifique au modèle de tâches dit *Frame-Based* où toutes les tâches partagent la même période et la même échéance. L'idée de l'algorithme est de séparer les tâches en deux groupes : les tâches dites *consommatrices* qui ont un taux de consommation supérieur à celui du rechargement, et les tâches dites *génératrices* qui ont un taux de consommation inférieur ou égal. L'algorithme ordonnance alors en continue les tâches consommatrice pour vider la batterie puis les génératrices pour la recharger. Si l'énergie générée par les tâches génératrices n'est pas suffisante, une

période de rechargement suffisamment longue est ajoutée au début du cycle. La même séquence d'ordonnancement est répétée à chaque période.

EDF As Late As Possible (EDL) : Cet algorithme s'applique à notre modèle décrit plus haut. Il s'agit d'une adaptation de l'algorithme EDF qui exécute les tâches au plus tard tout en respectant les échéances temporelles des tâches [Sil99]. Quand une tâche est prête pour être exécutée selon les règles de EDF, EDL calcule d'abord le retardement maximal à appliquer, et si ce dernier n'est pas nul, alors la tâche est retardée jusqu'à la consommation de la totalité des temps creux disponibles.

Malheureusement EDL n'est pas optimal car même si les retardements servent à recharger un maximum d'énergie avant les exécutions, le calcul des retardements ne prend pas en compte les contraintes énergétiques, ce qui peut aboutir à les surestimer.

EDF with energy guaranty ($EDeg$) : L'intuition derrière cet algorithme est d'ordonnancer les tâches selon EDF, mais avant d'autoriser un job à s'exécuter, $EDeg$ utilise la notion du *surplus d'énergie* [Che14; EGCC11] pour prédire d'éventuels futurs dépassements d'échéances dûs à une insuffisance d'énergie. Cette notion est l'équivalent "énergie" de celle du *temps creux* utilisée par EDL.

Cet algorithme est optimal (voir [Che14]) mais seulement pour les systèmes de tâches périodiques et synchrones.

La condition de charge maximale de l'énergie ($U_e = \sum_{i=1}^{n} E_i/T_i \times P_r \leq 1$) représente d'après [Che14] un test d'ordonnançabilité nécessaire et suffisant dans le cas où $\forall_i D_i = T_i$.

Lazy Scheduling Algorithm (LSA) : Cet algorithme a été proposé par Moser et al dans [Mos+06a]. Il s'agit d'un algorithme basé sur EDF mais qui retarde les exécutions à une date à partir de laquelle la tâche en cours peut s'exécuter continuellement et finir avant son échéance et cela en prévoyant les éventuelles interférences et en évitant de perdre de l'énergie si E_{max} est atteint.

Cet algorithme a été présenté comme étant optimal mais uniquement dans le cas où les tâches consomment l'énergie avec le même taux et que le temps d'exécution des tâches varie en fonction du taux de consommation appliqué.

Preemptive Fixed-Task-Priority As Late As Possible (PFP_{ALAP}) : Cet algorithme est l'équivalent de EDL pour la priorité fixe. Il retarde les exécutions au plus tard pour laisser le système recharger un maximum d'énergie. Malheureusement PFP_{ALAP} n'est pas optimal non plus parce que l'algorithme utilisé pour calculer les temps creux ne prend pas en considération les contraintes d'énergie.

Preemptive Fixed-Task-Priority with Slack-Time (PFP_{ST}) : Cet algorithme exécute les tâches au plus tôt en respectant leurs priorités statiques quand l'énergie est suffisante et retarde au plus tard sinon. L'algorithme de calcul des retardements est le

même que celui de PFP_{ALAP}. Cette approche améliore le taux d'ordonnançabilité par rapport à PFP_{ALAP} mais n'est pas optimal car l'algorithme de retardement n'intègre pas les paramètres énergétiques du système.

9.5 L'algorithme PFP_{ASAP}

La première contribution de cette thèse est l'étude de l'algorithme PFP_{ASAP}. Il s'agit d'une adaptation de l'ordonnancement classique à priorité fixe au contexte des systèmes collecteurs d'énergie.

C'est un algorithme à priorité de tâche fixe qui exécute les tâches au plut tôt dès que l'énergie est suffisante pour exécuter au moins une unité de temps et qui laisse le réservoir d'énergie se recharger le cas échéant. Les périodes de rechargement sont aussi longues que nécessaire pour exécuter une seule unité de temps. PFP_{ASAP}, malgré sa simplicité, présente des propriétés intéressantes telles qu'un pire scénario et une capacité minimale de batterie caractérisés et celle d'être un algorithme optimal dans sa classe.

9.5.1 Le pire scénario

La considération de la consommation d'énergie des tâches et les deux seuils du réservoir d'énergie impose des changements dans l'ordonnancement à priorité fixe et par conséquent la théorie classique d'analyse d'ordonnançabilité n'est plus applicable en l'état à commencer par la caractérisation du pire scénario.

Le pire scénario ou l'instant critique est la situation d'énergie et d'activation de tâches qui engendre les plus long temps de réponse des tâches. Cela se traduit par la maximisation des interférences de tâche plus prioritaires et la maximisation des temps de rechargement.

Dans le cas où toutes les tâches présentent un taux de consommation d'énergie plus élevé que le taux de rechargement, le pire scénario qu'un système peut subir avec un ordonnancement PFP_{ASAP} est l'activation synchrone de toutes les tâches quand le niveau de la batterie est à son seuil minimal (voir le chapitre 4). Cela découle du fait que quand on a que des tâches consommatrices d'énergie, le système a un comportement très proche de l'ordonnancement classique.

9.5.2 L'optimalité

Dans le cas des systèmes temps réel non-concrets, c'est à dire les systèmes où les premières dates d'activation des tâches et le niveau initial d'énergie du réservoir ne sont connus qu'au moment de l'exécution, l'étude d'ordonnançabilité du système se fait dans le pire cas.

Sachant que ce scénario est connu pour l'algorithme PFP_{ASAP}, si un système rate une échéance dans cette configuration, cela signifie que ce dernier n'est pas ordonnançable avec PFP_{ASAP}. De plus, nous savons qu'une échéance ne peut être violée que dans

deux cas avec PFP_{ASAP}. Le premier cas arrive quand la somme du temps d'exécution de la tâche et des interférences de tâches plus prioritaires est supérieure à l'échéance relative de la tâche. Dans ce cas le système n'est pas ordonanançable même sans les contraintes énergétiques. Le deuxième cas se produit quand la somme des interférences et des temps de rechargement dépassent l'échéance. Sachant que le système est composé uniquement de tâches qui nécessitent des rechargements, le système ne peut pas être ordonnnançable avec un autre algorithme à priorité fixe parce que les interférences ne peuvent pas être réduite et les temps de rechargement sont aussi court que possible. De ce fait, PFP_{ASAP} est optimal pour les systèmes non-concrets composés uniquement de tâches consommatrices.

9.5.3 Condition d'ordonnançabilité

Connaissant un algorithme optimal et son pire scénario, il est possible de construire un test de faisabilité nécessaire et suffisant en vérifiant que le temps réponse de chaque tâche dans le pire scénario ne dépasse pas son échéance. C'est ce que nous avons fait en proposant la formule 4.13 on page 120 pour calculer le pire temps réponse des tâches en adoptant l'algorithme itératif classique.

9.5.4 La capacité minimale de la batterie

La capacité minimale de la batterie recherchée ici est la plus petite capacité qui permet à un système de tâches ordonnançable avec PFP_{ASAP} et une capacité non bornée de le rester. Sachant que PFP_{ASAP} introduit des périodes de rechargement aussi courtes que possible pour exécuter une seule unité de temps, la capacité nécessaire pour stocker cette quantité d'énergie doit être supérieure ou égale au plus grand taux de consommation des tâches.

9.5.5 Avantages et inconvénients

Le principal inconvénient de PFP_{ASAP} est son taux très élevé de préemptions qui peut être bloquant en pratique car le coût des transitions entre les modes actif/inactif n'est pas négligeable dans certains cas. Par contre, PFP_{ASAP} peut être considéré comme un algorithme de référence dans l'ordonnancement à priorité fixe des systèmes collecteurs d'énergie de par sa simplicité et son optimalité pour les systèmes non-concrets.

9.6 Analyse de temps de réponse avec approximation

Malheureusement, quand on relâche les hypothèse sur la consommation des tâches, c'est à dire lorsque l'on considère également les tâches qui ont un taux de consommation

inférieur au taux de rechargement (tâches génératrices d'énergie), le pire scénario de PFP_{ASAP} n'est plus l'activation synchrone de toutes les tâches, et il n'existe à ce jour aucune caractérisation de ce pire scénario. Sans ce scénario, le calcul du pire temps de réponse des tâches n'est plus possible et par conséquent le test d'ordonnançabilité précédemment mentionné non plus.

Pour répondre à cette limitation, nous proposons de borner le pire temps réponse des tâches afin de construire des conditions d'ordonnançabilité suffisantes.

Pour ce faire, nous avons besoin d'une fonction qui borne par le haut le temps de réponse d'une tâche dans un intervalle de temps fini. Cela passe par une borne des interférences et des temps de rechargement.

Pour cela, nous construisons un scénario virtuel où nous maximisons indépendamment le nombre de tâches qui interfèrent avec la tâche dont nous bornons le pire temps de réponse, et le nombre d'unités de temps de rechargement nécessaires pour satisfaire la demande d'énergie de la tâche et les interférences.

Pour maximiser le nombre de jobs activés dans un intervalle donné, le seul moyen est d'activer le premier job immédiatement au début de l'intervalle ce qui anticipe l'activation des jobs suivants de la même tâche et permet d'avoir le plus d'activation possible. En faisant cela pour toutes les tâches, le scénario qui maximise les interférences est l'activation synchrone de toutes les tâches.

Une fois le nombre d'interférence maximisé, nous construisons nos scénarios virtuels qui maximisent le temps de rechargement.

La première borne supérieure $UB1$

Pour la première borne supérieure notée $UB1$, nous construisons une séquence d'exécution virtuelle qui suppose que tous les jobs consommateurs sont exécutés avant les jobs générateurs. Cela force le système à rajouter des unités de temps de rechargement à tous les jobs consommateurs et les prive de l'énergie générée par tâches génératrices. Ce scénario ne peut pas se produire dans l'ordonnancement réel mais peut être utilisé pour borner le pire temps réponse des tâches. Ainsi, la fonction de calcul de pire temps de réponse est construite. Cette fonction est ensuite utilisée dans l'algorithme itératif classique pour calculer les temps de réponse en priorité fixe.

Il est aussi possible de construire un scénario symétrique pour calculer une borne inférieure du pire temps de réponse afin de construire un test de faisabilité suffisant.

La deuxième borne supérieure $UB2$

Le borne $UB1$ est très pessimiste car elle considère un scénario irréaliste. Pour réduire ce pessimisme, nous proposons une deuxième borne notée $UB2$ qui reprend le même principe que $UB1$ mais en respectant cette fois les dates d'activation des jobs et leurs échéances. Nous construisons un scénario virtuel où les jobs consommateurs sont exécutés au plut tôt immédiatement après leurs activations et les jobs générateurs sont

exécutés au plus tard juste avant leurs échéances. De plus, les activations des tâches génératrices sont décalées de telle sorte que le dernier job soit exécuté immédiatement après son activation et juste avant la fin de l'intervalle. Un fois le scénario construit, toutes les unités de temps de l'intervalle sont parcourues pour calculer le temps réponse dans ce scénario. Quand une unité d'un job consommateur est rencontrée, cette dernière est ajoutée au temps réponse avec les unités de rechargement nécessaires, et quand une unité d'un job générateur est rencontrée, celle-ci est ajoutée au temps réponse à son tour. En cas de chevauchement entre consommateurs et générateurs, les générateurs sont plus prioritaires car ils sont contraints par leurs échéances et cela permet de réduire le pessimisme. Ainsi, la fonction de bornage du pire temps de réponse est construite et peut être utilisée itérativement pour un test de faisabilité.

Les simulations montrent qu'effectivement $UB2$ est moins pessimiste que $UB1$ en particulier quand l'utilisation du système U est très élevée. Nous constatons aussi que les conditions d'ordonnançabilité basées sur ces bornes deviennent des tests exacts quand le système est composé uniquement de tâches consommatrices ou uniquement de tâches génératrices.

La capacité minimale de la batterie

Pour que ces bornes soient valides, la capacité de la batterie doit être suffisamment grande pour stocker l'énergie collectée pendant les périodes de rechargement et pendant l'exécution des tâches génératrices. Pour $UB1$ la batterie doit pouvoir stocker au moins l'énergie nécessaire pour satisfaire le plus grand taux de consommation d'énergie des tâches. Cela découle des exécutions gloutonnes produites par PFP_{ASAP}.

Pour $UB2$, on borne la capacité minimale par la plus grande demande d'énergie de la plus longue période d'exécution.

9.7 Recherche d'algorithme optimal

Nous avons vu dans les sections précédentes que ni l'ordonnancement au plut tôt avec PFP_{ASAP}, ni l'ordonnancement au plus tard avec PFP_{ALAP}, ni le mixte des deux avec PFP_{ST} ne sont optimaux. Pour PFP_{ASAP}, cela est dû au fait que l'ordonnancement au plus tôt peut mener à la situation où une tâche moins prioritaire qui s'exécute au plus tôt peut consommer l'énergie nécessaire pour une autre tâche plus prioritaire qui s'active juste après alors que la tâche la moins prioritaire peut être retardée pour permettre la tâche la plus prioritaire d'utiliser l'énergie rechargée pendant le retardement. Pour PFP_{ASAP} et PFP_{ST}, le problème vient du fait que les retardements sont calculés sans prendre en compte les contraintes énergétiques.

L'algorithme optimal doit alors calculer les retardements exacts qui prennent en compte la demande d'énergie des tâches et l'énergie disponible. Cela veut dire que

cet algorithme ne peut être non-oisif énergiquement (comme PFP_{ASAP}) et doit être clairvoyant pour éviter les futurs dépassements d'échéances.

Pour aller dans ce sens, nous avons exploré les idées intuitives suivantes :

- **F**ixed-Task-**P**riority **C**lairvoyant **a**s **s**oon **a**s **p**ossible ($FPCasap$) : Cet algorithme utilise une fonction de clairvoyance pour prédire de futurs dépassements d'échéances avant l'autorisation de l'exécution du plus prioritaires des jobs actifs. Cette fonction consiste à simuler l'ordonnancement du système selon PFP_{ASAP} dans l'intervalle de temps qui va de l'instant courant jusqu'à l'échéance du job en cours. Ainsi, si un dépassement d'échéance est détecté, l'exécution est retardée jusqu'à ce que toutes les échéances de l'intervalle soient respectées.

  Malheureusement, même si $FPCasap$ est clairvoyant et oisif, il n'est pas optimal car il retarde toutes les tâches dans l'ordre chronologique de leur activations. La figure 6.7 on page 163 montre un contre-exemple où l'ordonnancement optimal ne respecte pas cet ordre.

- **F**ixed-Task-**P**riority **L**azy **S**cheduling **A**lgorithm ($FPLSA$) : Cet algorithme est l'équivalent "priorité de tâche fixe" de LSA qui est basé sur EDF (voir section 9.4). Il n'est pas optimal non plus pour la priorité fixe même en supposant un taux de consommation d'énergie unique pour toutes les tâches car le même problème de retardement persiste.

- **F**ixed-Task-**P**riority As **L**ate As Possible with **e**nergy **g**uaranty ($FPLeg$) : Cet algorithme reprend l'ordonnancement produit par PFP_{ALAP} mais cette fois en prenant en compte les contraintes énergétiques. Cela passe par l'utilisation des *échéances virtuelles* à la place des vraies pour calculer les retardements. Ici, une échéance virtuelle est une date chronologiquement inférieure à l'échéance réelle mais qui garantit un bilan d'énergie positif de l'intervalle de temps qui va de l'instant courant à cette échéance virtuelle. Cela suppose que si un ordonnancement faisable existe, le bilan énergétique à la fin de chaque job doit être positif.

  Cette idée est une piste très intéressante mais n'est pas suffisante car si un job moins prioritaire trouve une telle échéance, cela force toutes les interférences plus prioritaires à s'exécuter plus tôt que nécessaire ce qui les expose au même problème que celui de PFP_{ASAP}. La figure 6.9 on page 166 illustre une telle situation.

- **F**ixed-Task-**P**riority **l**ooka**h**ead ($FPlh$) : L'idée de cet algorithme est d'utiliser des échéances virtuelles exactes qui garantissent non seulement un bilan positif mais aussi une vérification des bilans énergétiques des jobs activés dans une fenêtre de clairvoyance. Cela est supposé calculer le retardement optimal pour chaque job. Malheureusement, la complexité de cet algorithme est exponentielle du fait que le calcul des échéances virtuelles est interdépendant entre les jobs plus prioritaires

et les moins prioritaires et que la taille de la fenêtre de clairvoyance est très importante dans le pire cas.

Cela montre la difficulté de trouver un algorithme optimal pour la priorité fixe et mène à la conclusion suivante :

- Si le calcul des échéances virtuelles est nécessaire pour un algorithme optimal, alors, il aura une complexité pseudo-polynomiale multipliée par la complexité de l'algorithme de calcul de ces échéances virtuelles.

- Dans ce cas, si la complexité de calcul des échéances virtuelles est NP-difficile alors la problématique d'ordonnancement à priorité de tâche fixe des systèmes collecteurs d'énergie l'est également.

- Sinon, le problème reste ouvert.

9.8 Analyse de temps de réponse des systèmes à contraintes thermiques

Dans le but de mieux comprendre la problématique d'ordonnancement à priorité fixe des systèmes collecteurs d'énergie, nous proposons d'explorer les solutions proposées pour des problématiques d'ordonnancement similaires, en particulier celles où le retardement des exécutions est parfois nécessaire pour respecter certaines contraintes.

L'ordonnancement avec contraintes thermiques est l'une de ces problématiques. Cette dernière consiste à exécuter les tâches de telle sorte qu'une certaine température maximale n'est jamais atteinte ou dépassée. Cette contrainte pousse le système à suspendre les exécutions de temps en temps pour insérer des temps de refroidissement afin d'éviter que la température maximale ne soit atteinte.

Dans ce domaine, il existe deux approches. La première dite *réactive* consiste à suspendre l'exécution en réaction à l'atteinte de la température maximale. La seconde dite *proactive* consiste à suspendre l'exécution ou à réduire la vitesse du processeur avant d'atteindre la température maximale. Différents travaux ont été menés sur ce sujet mais la majorité des résultats sont basés sur la technique de réduction de la fréquence et/ou de la tension du processeur ou sur l'ordonnancement EDF.

Comme première étape de ce travail, nous proposons dans cette thèse d'adapter les solutions proposées pour les systèmes à énergie renouvelable aux systèmes à contraintes thermiques pour l'ordonnancement à priorité fixe. Ainsi, nous adaptons l'algorithme PFP_{ASAP} pour que la contrainte thermique soit respectée ce qui constitue une solution réactive. Nous proposons également une analyse d'ordonnançabilité basée sur des bornes du pire temps de réponse des tâches.

Modèle

Nous considérons un système temps réel non-concret composé d'un ensemble de tâches sporadiques et indépendantes avec des échancres contraintes ($\forall_i D_i \leq T_i$).

Le modèle Thermique : nous considérons que la température du système fluctue en fonction de la dissipation de chaleur générée par l'exécution des tâches sur le processeur.

La température à l'instant t notée $T(t)$ est toujours supérieure ou égale à la température ambiante T_A et ne doit jamais dépasser la température maximale T_{max}. Le seul moyen pour refroidir le système est de suspendre temporairement l'exécution des tâches et de mettre le processeur en mode inactif.

Le comportement thermique du processeur peut être modélisé par un filtre Résistance Capacitance (RC) [TYS01] illustré par la Figure 7.1 on page 176. Nous utilisons alors l'équation différentielle de Fourrier pour modéliser le comportement thermique de notre système (voir l'équation 7.1 on page 176). La solution à cette equation donne la fonction qui exprime la température en fonction du temps et des paramètres du processeur (voir l'équation 7.6 on page 177). Quand le processeur est inactif, sa vitesse devient nulle, et par conséquent la dissipation de chaleur est nulle, le système commence alors à se refroidir jusqu'à atteindre la température ambiante (voir l'équation 7.7 on page 177).

L'adaptation de l'algorithme PFP_{ASAP}

L'équivalent de l'algorithme PFP_{ASAP} pour le modèle avec contraintes thermiques est l'ordonnancement au plus tôt qui choisit la plus prioritaire des tâches actives à tout instant dès que le système est suffisamment froid pour pouvoir exécuter au moins une unité de temps, et qui laisse seulement refroidir dans le cas échéant. Les périodes de refroidissement sont aussi longues que nécessaire pour pouvoir exécuter une seule unité de temps. Il s'agit de l'ordonnancement thermiquement non-oisif, c'est à dire que des périodes de refroidissement ne sont ajoutées que lorsque c'est nécessaire.

Pour le modèle considéré, nous prouvons que le pire scénario de PFP_{ASAP} est l'activation synchrone de toutes les tâches quand la température est à son niveau maximal. Cela découle du fait que toutes les tâches chauffent le système de la même façon ce qui est très proche du cas des systèmes collecteurs d'énergie composés uniquement de tâches consommatrices d'énergie.

Nous prouvons également que PFP_{ASAP} est optimal pour les systèmes non-concrets dont la température initiale et la date de la première activation des tâches ne sont connus qu'au moment de l'exécution. L'idée de la preuve consiste à analyser tous les cas où PFP_{ASAP} rate une échéance dans le pire cas et voir s'il est possible de l'éviter.

9.8.1 Analyse de temps de réponse

Notre objectif maintenant est de construire des conditions d'ordonnançabilité de PFP_{ASAP} pour le modèle considéré. Le temps de réponse d'une tâche étant difficile à

écrire sous forme d'une formule générique à cause du temps discret, nous proposons une analyse de temps de réponse basée sur des bornes du pire temps de réponse des tâches. Pour cela, nous proposons deux bornes supérieures qui maximisent le temps de refroidissent nécessaire pour l'exécution selon PFP_{ASAP}.

La première borne $UB_{T_{min}}$: Cette borne consiste à supposer que toutes les périodes de refroidissement sont les plus longues possible, c'est à dire le temps nécessaire pour faire baisser la température de T_{max} à la température ambiante. L'approximation vient du fait de la nature exponentielle inversée de la fonction de refroidissement, c'est à dire que plus le refroidissent est long plus il est lent. On utilisera T_{min} à la place de T_A car quand le processeur est éteint, la température baisse asymptotiquement à T_A ce qui peut rendre la borne très pessimiste.

La borne paramétrée UB_x : L'idée de cette borne est de reproduire l'ordonnancement de PFP_{ASAP} en considérant que toutes les périodes de refroidissent ont la même longueur et que cette dernière est toujours plus longue que celle de l'ordonnancement réel. En variant celle-ci, nous augmentons le pessimisme de la borne mais nous baissons sa complexité en moyenne.

Ces bornes sont utilisées dans l'algorithme itératif classique pour calculer le pire temps réponse final des tâches. Ainsi, nous obtenons deux tests d'ordonnançabilité suffisants.

Les simulations montrent que UB_x domine de loin $UB_{T_{min}}$ et est très proche du pire temps de réponse empirique pour des petites valeurs du paramètre x mais devient très pessimiste quand x devient très grand. Nous observons aussi qu'en variant x, le gain en complexité est plus faible que la perte au niveau de l'ordonnançabilité.

9.9 L'outil de simulation YARTISS

Pour terminer, nous présentons brièvement *YARTISS* : l'outil de simulation développé pendant cette thèse pour en évaluer les résultats théoriques.

YARTISS est un logiciel qui permet de simuler l'ordonnancement des systèmes de tâches temps réel. Il permet de tester et comparer différents algorithmes et politiques d'ordonnancement dans différents environnements, par exemple, multi/monoprocesseur, contraintes d'énergie, etc.

Un effort conséquent a été fait pour rendre son architecture la plus générique et extensible possible afin de pouvoir y intégrer et tester différents modèles de tâches, d'algorithmes d'ordonnancement, de plateformes, etc.

YARTISS offre deux fonctionnalités principales : la première est la visualisation de l'ordonnancement temps réel d'un système donné selon un algorithme donné et dans un environnement donné. La deuxième est la simulation à grande échelle d'un ensemble de

systèmes de tâches afin de comparer les performances d'algorithmes d'ordonnancement ou de tests de faisabilité/ordonnançabilité donnés.

Nous espérons que cet apport pourra contribuer au développement d'un outil de simulation qui pourra être adopté par une grande partie de la communauté de l'ordonnancement temps réel.

9.10 Conclusion

Dans cette thèse nous avons traité le problème de l'ordonnancement temps réel à priorité fixe des systèmes collecteurs d'énergie. Un tel système doit respecter non seulement les échéances des tâches mais aussi leurs contraintes énergétiques. Cela veut dire que le coût énergétique des tâches doit être satisfait par l'énergie stockée dans la batterie ainsi que par l'énergie collectée depuis l'environnement. Pour satisfaire la demande énergétique des tâches, des périodes d'inactivité du processeur additionnelles peuvent être nécessaires pour collecter l'énergie demandée. Le défi pour un algorithme d'ordonnancement à priorité fixe est alors d'exécuter les tâches en respectant leurs échéances et de retarder les exécutions au bon moment et pendant la bonne durée pour satisfaire au mieux les contraintes énergétiques et les échéances.

La problématique ici est de premièrement trouver un algorithme optimal pour le modèle considéré ; deuxièmement, de fournir des outils d'analyse d'ordonnançabilité ; troisièmement, de trouver la capacité minimale de la batterie qui préserve l'ordonnançabilité d'un système donné.

Dans ce résumé, nous avons commencé par un bref état de l'art de ce domaine de recherche incluant un aperçu de la théorie classique de l'ordonnancement temps réel, les différentes technologies de stockage et d'extraction de l'énergie environnementale et les principaux algorithmes d'ordonnancement proposés dans la littérature pour les systèmes collecteurs d'énergie.

Nous avons présenté ensuite les résultats de cette thèse. Le premier résultat est l'étude de l'algorithme PFP_{ASAP} qui est optimal pour les systèmes non-concrets composés uniquement de tâches qui consomment plus d'énergie que le rechargement. Ce travail a été effectué en collaboration avec Dr. Y. Abdeddaïm et a été publié dans les actes de la conférence ECRTS 2013 [ACM13a]. Le deuxième résultat est une analyse d'ordonnançabilité basée sur des bornes du pire temps de réponse des tâches qui prend en compte tous les types de taux de consommation des tâches. Ce travail est le fruit d'une collaboration avec Prof. R. Davis et a été publié dans les actes de la conférence RTNS 2014 [Abd+14]. Ce papier a obtenu le prix du meilleur article de cette conférence. Le troisième résultat est une exploration des différentes idées intuitives pour construire un algorithme optimal où nous avons soulevé la difficulté de trouver un tel algorithme. Ce travail est publié dans un rapport de recherche [AC13]. Le quatrième résultat est la transposition des résultats obtenus pour les systèmes collecteurs sur les systèmes contraints par la température qui présentent des similitudes au niveau de la problé-

matique de l'ordonnancement à priorité fixe. Cette idée a été étudiée dans le cadre d'une collaboration avec Prof. N. Fisher et est en cours de soumission. Finalement, le cinquième résultat est le développement d'un outil de simulation de l'ordonnancement des systèmes temps réels. YARTISS a fait l'objet d'une publication dans le workshop WATERS 2012 [Cha+12].

Appendix

YARTISS Case Study

Listing 1: How to add a scheduling policy

```
1  public class MainDemoSP {
2    public static void main(String[] args) {
3      SchedulingPolicyFactory.registerPolicy(new PFPASAP());
4      DesktopMain main = new DesktopMain();
5      main.setVisible(true);
6    }
7  }
8
9  /**
10  * Preemptive Fixed Priority. When there is not enough energy, the system is
11  * paused for x time unit (even if it results in missing a deadline)
12  */
13 class PFPASAP extends ISchedulingPolicy {
14
15   protected final String tasksetType;
16   protected final String policyName;
17   protected List<Integer> parameters;
18   private long sleepUntil = -1;
19   private final int x = 1;
20
21   public PFPASAP(String tasksetType, String policyName) {
22     this.tasksetType = tasksetType;
23     this.policyName = policyName;
24   }
25
26   public PFPASAP(){
27     this("priority", "PFP_ASAP");
28   }
29
30   @Override
31   public void chooseNextTasks(Platform platform, ITaskSet taskSet, IEnergyProfile energyProfile,
         long t, EventGenerator evGen) {
32
33     SortedSet<ITask> activeTasks = taskSet.getActiveTasks(t);
34
35     if (!activeTasks.isEmpty()) {
36       ITask first = activeTasks.first();
37       long hlcet = energyProfile.howLongCanExecute(first);
38
39       if (hlcet <= 0) {
40         evGen.generateEvent("check_energy_state", activeTasks.first(), t + x, null);
41         platform.getProcessor().setNextTask(null);
```

```
42          return;
43        }
44
45        if (hlcet < activeTasks.first().getRemainingCost())
46          evGen.generateEvent("check_energy_state", activeTasks.first(), t + hlcet, null);
47
48        platform.getProcessor().setNextTask(first);
49        return;
50      }
51      platform.getProcessor().setNextTask(null);
52    }
53
54    @Override
55    public void setParameters(List<Integer> parameters) {
56      parameters = parameters!= null ? parameters: new ArrayList<Integer>();
57      this.parameters = new ArrayList<Integer>(parameters);
58    }
59
60    @Override
61    public List<Integer> getParameters() {
62      return parameters;
63    }
64
65    @Override
66    public String getPolicyName() {
67      return policyName;
68    }
69
70    @Override
71    public ITaskSet createTaskSet() {
72      return TaskSetFactory.getNewInstance(tasksetType);
73    }
74
75    @Override
76    public ISchedulingPolicy newInstance() {
77      return new PFPASAP(tasksetType, policyName);
78    }
79  }
```

Listing 2: How to add a new energy profile

```
1  public class MainDemo {
2    public static void main(String[] args) {
3      SchedulingPolicyFactory.registerPolicy(new LLF());
4      ConsumptionProfileFactory.registerConsumptionProfile(new LogConsumption());
5      DesktopMain main = new DesktopMain();
6      main.setVisible(true);
7    }
8  }
9
10 class LogConsumption implements
```

```
    IEnergyConsumptionProfile {
    @Override
    public String getName() {return "log";}
    @Override
    public List<Double> getParameters() {return null;}
    @Override
    public void setParameters(List<Double> params) {}

    @Override
    public long getConsumed(long wcet, long wcee,
            long remainingTimeCost, long duration) {
      double a = wcet - romainingTimeCost;
      double b = a + duration;
      if(b > wcet) b = wcet;
      if( (b-a) <= 0 )return 0;
      long result = (long) Math.log(b/a);
      if(result > wcee )result = wcee;
      return result;
    }

    @Override
    public IEnergyConsumptionProfile cloneProfile() {
      return new LogConsumption();
    }
}
```

Glossaries

Symbols

$Ba_i(t)$

The energy balance of a job $J_{i,j}$ at time t. 155

$Bu(t_1, t_2)$

The energy budget of time interval $[t_1, t_2]$. 155

C_i

The worst case execution time of task τ_i. 31

$Constant$

A constant. 99, 100, 102–104

$DBF_i(t)$

The demand bound function of task τ_i in time interval $[0, t]$. 42

D_i

The relative deadline of task τ_i. 31

$E(t)$

The level of the energy storage unit at time t. 68

$E^*_X[m]$

The total energy required by execution units from the start of the sequence up to and including execution unit $X[m]$. 134

$E_X[m]$

The amount of energy required by the execution unit $X[m]$ of sequence X. 134

E_i

The worst case energy consumption of task τ_i. 67

E_{max}

The highest reachable level of the energy storage unit. 68, 238

E_{min}

The lowest authorized level of the energy storage unit. 68, 238

$I_X^*[m]$

The the minimum number of replenishment units required to execute all of the subsequence $X[1]$ to $X[m]$ in order. 135

$I_X[m]$

The minimum number of replenishment units required to provide sufficient energy to execute $X[m]$ at the end of the subsequence $X[1]$ to $X[m]$. 135

L_X

The number of execution units in sequence X. 134

N_h

The number of jobs in sequence X. 141

O_i

Offset or first release time of task τ_i. 30

$P(t)$

The global power consumption function of the system. 176

$P_D(t)$

The dynamic power consumption function of the system. 176

$P_L(t)$

The power leakage or the static power consumption function of the system. 177

P_i

The priority of task τ_i. 38

P_r

Constant replenishment function. 68

$P_r(t)$

The power rate or the replenishment function. 67, 238

$P_{i,j}$

The priority of job $J_{i,j}$. 38

Pc_i

The worst-case power consumption of task τ_i during one time unit. 132

R

The electrical resistance. 176

$RBF_i(t)$

The request bound function of task τ_i in time interval $[0, t]$. 42

R_i

The **W**orst **C**ase **R**esponse **T**ime (WCRT) of task τ_i. 32

R_i^{UB1}

The WCRT of task τ_i according to response time upper bound UBx. 137

$R_{i,j}$

The response time of job $J_{i,j}$. 32

$SE(t)$

The available slack-energy of the system at time t. 84

$ST(t)$

The available slack-time of the system at time t. 84

$ST_i(t)$

The available slack-time of priority level-i at time t. 92

$T(t)$

The temperature of the system at time t. 176, 247

T_i

The period or the inter-arrival time of task τ_i. 31

T_A

The ambient temperature. 176

T_{max}

The the maximum tolerated temperature. 176, 247

U

The processor utilization of the system. 41

U_i

The processor utilization of task τ_i. 41

Ue

The energy utilization of the system. 70

V

Volt. 54

W

Watt. 50

$W(t_1, t_2)$

The workload of the system in time interval $[t_1, t_2]$. 42

W/cm^2

Watt per square centimeter. 50

$W_i(t_1, t_2)$

The workload of task τ_i in time interval $[t_1, t_2]$. 42

$W_i^m(t)$

The recurrent workload function. 92

$W_i^{UB1}(w)$

The workload function of priority level-i according to upper bound $UB1$. 139

$We_i(t)$

The energy workload or demand of priority level-i in time interval $[0, t]$. 113

Wh/kg

Watt-hour per kilogram. 56

Wh/m^3

Watt-hour per cubic centimeter. 56

$Wp_i(t)$

The processor workload or demand of priority level-i in time interval $[0, t]$. 113

X

A vector of execution units from 1 to L_X. 134, 255, 256

$X[m]$

The m-th element of sequence X. 134, 255, 256

Δ

A certain interval of time. 87

Γ

A task set. 38, 132, 259

Γ_c

The subset of Γ that contains consuming tasks. 132

Γ_g

The subset of Γ that contains gaining tasks. 132

Λ

The density of the system. 41

Λ_i

The density of task τ_i. 41

$^{\circ}\mathbf{C}$

Celsius degree. 54

$\mathcal{C}$

The capacity of the energy storage unit. 68, 176

$\mathcal{C}_{min}^{UBi}$

The minimum energy storage unit capacity needed for response time upper bound UBi. 143

$\mathcal{C}_{min}$

The minimum capacity of the energy storage unit that keeps a task set schedulable. 91

μA

Microampere. 54

μF

Microfarad. 60

μW

Microwatt. 54

$\mu W/cm^2$

Milliwatt per square meter. 51

$\mu W/cm^3$

Milliwatt per cubic centimeter. 53

τ_i

Task number i or of priority level i. 30

$a_i(t)$

The next job request time of priority level-i. 92, 154

$a_{i,j}$

The arrival time of job $J_{i,j}$. 32

$c_i(t)$

The remaining execution time at time t of the current job of task τ_i. 92, 154

cm^2

Square centimeter. 50

cm^3

Cubic centimeter. 53

$d_i(t)$

The next absolute deadline of priority level-i after time t. 92, 154

$d_{i,j}$

The absolute deadline of job $J_{i,j}$. 32

$e_i(t)$

The remaining energy cost of the current job of task τ_i at time t. 154

$f_{i,j}$

The termination time of job $J_{i,j}$. 32

$g(t_1, t_2)$

The energy replenished during time interval $[t_1, t_2]$. 68

$hp(k)$

Subset of tasks of priority higher or than τ_k. 146

$idle(t_1, t_2)$

The amount of time where the processor is idle within interval of time $[t_1, t_2]$. 92

$lp(i)$

Subset of tasks of priority lower or equal than τ_i. 146

mA

Milliampere. 56

mAh

Milliampere-hour=$10^{-3}Ah$. 56

mW

Milliwatt. 54

n

The task set cardinal. 38

$s_i(t)$

The effective execution starting time of the current job of task τ_i at time t. 154

$s_{i,j}$

The effective execution starting time of job $J_{i,j}$. 87, 154

t

Time instant t. 36

Acronyms

DM

***D**eadline **M**onotonic.* 38, 124, 146, 147, 204, 212, 216,
— Glossary: *DM*

EDF

***E**arliest **D**eadline **F**irst.* 22, 40, 79–81, 83–86, 89, 91, 92, 105, 164, 204, 206, 211, 212, 216, 234, 235, 240, 262, 268, 270,
— Glossary: *EDF*

EDL

*EDF As **L**ate As Possible.* 79–84, 86, 92, 95, 98, 99, 102–104, 240, 273,
— Glossary: *EDL*

EDeg

*EDF with **e**nergy **g**uaranty.* 79, 83–86, 98, 99, 101, 104, 240, 273, 275,
— Glossary: *EDeg*

FBA

***F**rame-**B**ased **A**lgorithm.* 76–79, 98, 99, 101, 102, 104, 239, 273, 275,
— Glossary: *FBA*

FPCasap

***F**ixed-Task-**P**riority **C**lairvoyant **a**s **s**oon **a**s **p**ossible.* 161–163, 165, 167, 245, 274, 275,
— Glossary: *FPCasap*

FPLSA

***F**ixed-Task-**P**riority **L**azy **S**cheduling **A**lgorithm.* 164, 165, 245, 274, 275,
— Glossary: *FPLSA*

FPLeg

***F**ixed-Task-**P**riority As **L**ate As Possible with **e**nergy **g**uaranty.* 164–170, 245, 274, 275,
— Glossary: *FPLeg*

FPlh

***F**ixed-Task-**P**riority **l**ooka**h**ead.* 167–170, 245, 274, 275,
— Glossary: *FPlh*

LLF

***L**east **L**axity **F**irst*. 40, 212, 216, 235, 269,
— Glossary: *LLF*

LSA

***L**azy **S**cheduling **A**lgorithm*. 79, 86–91, 98, 99, 101, 102, 104, 164, 240, 245, 273, 275,
— Glossary: *LSA*

PFP_{ALAP}

***P**reemptive **F**ixed-Task-**P**riority **A**s **L**ate **A**s **P**ossible*. 92–96, 98, 99, 101, 103, 104, 110, 123, 125–129, 160, 161, 165, 166, 168, 170, 240, 241, 273–275,
— Glossary: PFP_{ALAP}

PFP_{ASAP}

***P**reemptive **F**ixed-Task-**P**riority **A**s **S**oon **A**s **P**ossible*. 23, 110–114, 116–119, 121, 122, 125–129, 131, 136, 138, 141, 144, 147, 148, 151, 153, 156, 157, 159–163, 174, 178, 179, 181–185, 188, 189, 191–193, 196, 198, 202, 221, 222, 225–227, 241, 273–275,
— Glossary: PFP_{ASAP}

PFP_{ST}

***P**reemptive **F**ixed-Task-**P**riority with **S**lack-**T**ime*. 95–99, 101, 103, 104, 110, 123, 125–128, 161, 240, 273–275,
— Glossary: PFP_{ST}

RM

***R**ate **M**onotonic*. 38, 204, 212, 216,
— Glossary: *RM*

API

***A**pplication **P**rogramming **I**nterface*. 204, 205, 214, 219, 220

CMOS

***C**omplementary **M**etal–**O**xide–**S**emiconductor*. 74

CPU

***C**entral **P**rocessing **U**nit*. 21, 29, 35, 73, 74, 175, 176,
— Glossary: CPU

DAG

***D**irected **A**cyclic **G**raph*. 35, 173, 208, 216,
— Glossary: DAG

DBF

***D**emand **B**ound **F**unction*. 42,
— Glossary: DBF

DP

***D**inamic **P**riority*. 38,
— Glossary: DP

DVFS

***D**ynamic **V**oltage and **F**requency **S**caling*. 73, 74, 175, 217

EH

***E**nergy-**H**arvesting*. 47,
— Glossary: EH

EHS

***E**nergy-**H**arvesting **S**ystem*. 22, 48

EVCC

***E**nergy **V**ariability **C**haracterization **C**urves*. 87,
— Glossary: EVCC

FIFO

***F**irst **I**n **F**irst **O**ut*. 175

FJP

***F**ixed-**J**ob-**P**riority*. 38, 40,
— Glossary: FJP

FTP

***F**ixed-**T**ask-**P**riority*. 38, 40, 71, 95, 97, 147, 148, 211, 225, 268, 269,
— Glossary: FTP

GUI

***G**raphic **U**ser **I**nterface*. 213, 214, 223

HP

***H**yper-**P**eriod*. 42,
— Glossary: HP

ICT

***I**nformation and **C**ommunication **T**echnologies*. 55, 56, 61, 273

IMD

***I**mplantable **M**edical **D**evice*. 174

IoC

***I**nversion **o**f **C**ontrol*. 214, 218, 219,
— Glossary: IoC

LCM

***L**east **C**ommon **M**ultiple*. 42, 270

Li-ion

Li**thium **Ion. 59, 237, 273

Li-polymer

Li**thium-ion **Polymer. 59, 273

MEMS

***M**icro**e**lectro**m**echanical **S**ystems*. 52

MP3

MPEG Audio Layer 3. 21,
— Glossary: MP3

MVC

***M**odel-**V**iew-**C**ontroller*. 214, 219, 220,
— Glossary: MVC

MVVM

***M**odel **V**iew **V**iew-**M**odel*. 219,
— Glossary: MVVM

NiCd

***N**ickel **C**admium*. 57–59, 237, 273

NiMH

***Ni**ckel-**M**etal **H**ydrid*. 58, 59, 237, 273

NP-Hard

***N**on-deterministic **P**olynomial-time Hard*. 74

OPA-compatible

***O**ptimal **P**riority **A**ssignment (OPA)-compatible*.
— Glossary: OPA-compatible

OPA

***O**ptimal **P**riority **A**ssignment*. 38, 39, 265, 270,
— Glossary: OPA

PC

***P**ersonal **C**omputer*. 21

QoS

***Q**uality **o**f **S**ervice*. 29, 35

RBF

***R**equest **B**ound **F**unction*. 42,
— Glossary: RBF

RC

***R**esistance **C**apacitance circuit*. 176, 271,
— Glossary: RC

RF

***R**adio **F**requency*. 50, 51, 53

RFID

***R**adio **F**requency **Id**entification*. 51, 236

RTEHS

***R**eal-**T**ime **E**nergy-**H**arvesting **S**ystems*. 22

RTS

***R**eal-**T**ime **S**ystems*. 21, 28

TA

***T**hermal-**A**ware.*

— Glossary: TA

WCET

***W**orst **C**ase **E**xecution **T**ime.* 31, 34, 35, 208, 211, 215

WCRT

***W**orst **C**ase **R**esponse **T**ime.* 45, 257, 271,

— Glossary: WCRT

XML

*E**x**tensible **M**arkup **L**anguage.* 206

YARTISS

***Y**et **A**n Other **R**eal-**Ti**me **S**ystems **S**imulator.* 23, 204–208, 211, 212, 215–219, 221, 223, 224

Glossary

DM

The priority assignment policy that grants the highest priority to the task with the smallest deadline. 262

EDF

The scheduling policy that executes first the jobs with the nearest deadlines. 262

EDL

The scheduling algorithm that schedules jobs as late as possible according to *EDF* rules. 262

EDeg

The scheduling algorithm that checks if the execution of the current job does not lead to future deadline miss before authorizing it to execute according to *EDF* rules. 262

FBA

The scheduling algorithm that schedules continuously the group of gaining tasks when the minimum battery level is reached and executes the group of consuming tasks when the maximum battery level is reached. 262

FPCasap

The FTP algorithm that uses lookahead computations to detect future deadline misses in order to adapt the scheduling decisions. 262

FPLSA

The FTP counter-part of Lazy Scheduling Algorithm. 262

FPLeg

The FTP algorithm that uses only virtual deadlines to compute the maximum delay to apply for each job. 262

FPlh

The FTP algorithm that uses virtual deadlines and lookahead verifications to compute the maximum delay to apply for each job. 262

LLF

The scheduling policy that executes first the jobs with the smallest laxity. 262

LSA

The Lazy Scheduling Algorithm is an EDF-based algorithm that delays execution such that the current job and the eventual higher priority jobs can be executed continuously without running out of energy and without exceeding the maximum capacity of the battery. 263

PFP_{ALAP}

The scheduling algorithm that schedules jobs as late as possible according to FTP rules. 263

PFP_{ASAP}

An energy-work-conserving FTP scheduling policy. 263

PFP_{ST}

The scheduling algorithm that executes jobs according to FTP rules when some energy is available and replenishes as long as possible otherwise. 263

RM

The priority assignment policy that grants the highest priority to the task with the smallest period. 263

CPU

The processor. 263

DAG

The task model that considers dependencies between the jobs of the same task. 263

DBF

The demand bound function of recurrent is the worst cumulative workload generated jobs that are *requested and finished* during time interval $[0, t]$. 263

DP

According to dynamic priority scheduling, a job may be scheduled with different priorities, e.g. *LLF*. 264

EH

Energy-Harvesting systems collect the energy from the environment and use it to supply the operations of the system. 264

EVCC

The **E**nergy **V**ariability **C**haracterization **C**urves is technique used to bound the amount of energy that comes from a solar energy harvester. 264

FJP

Scheduling algorithms driven by jobs priorities, e.g. EDF. 264

FTP

Scheduling algorithms driven by the static priorities of tasks. 264

HP

The hyper-period is the smallest interval of time after which the global periodic pattern of all the tasks are repeated. It is typically defined as the **L**east **C**ommon **M**ultiple (LCM) of the periods of all the tasks of the system.. 264

IoC

The **I**nversion **o**f **C**ontrol design pattern. 265

MP3

An audio codec. 265

MVC

Model-**V**iew-**C**ontroller design pattern. 265

MVVM

The **M**odel **V**iew **V**iew-**M**odel design pattern. 265

OPA

The algorithm of Audsley that calculate a valid tasks priority assignment whenever there exists one. 266

OPA-compatible

A schedulability test is compatible with the OPA algorithm if the priority ordering of higher and lower priority tasks does not alter the schedulability of the current level. 265

RBF

The request bound function computes the worst cumulative workload generated by jobs that are requested during time interval $[0, t[$. 266

RC

The **R**esistance **C**apacitance circuit (RC) circuit is used to model the thermal behavior of the system. 266

TA

Thermal-Aware systems adapt the performances in order to keep the temperature of the system below a maximum threshold. 266

WCRT

The WCRT of a task is the maximum duration between the activation of the task and the moment it finishes its execution. 266

List of Figures

List of Algorithms

Bibliography

[AC13] Yasmina Abdeddaïm and Younès Chandarli. *Optimal Real-Time Scheduling Algorithm for Fixed-Priority Energy-Harvesting Systems*. technical report. Paris, France, 2013. URL: http://hal.archives-ouvertes.fr/hal-01076021.

(Cited on pages 2, 249).

[ACM13a] Yasmina Abdeddaïm, Younès Chandarli, and Damien Masson. "The Optimality of PFP_{ASAP} Algorithm for Fixed-Priority Energy-Harvesting Real-Time Systems". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 2013.

(Cited on pages 1, 110, 149, 249).

[ACM13b] Yasmina Abdeddaïm, Younès Chandarli, and Damien Masson. "Toward an Optimal Fixed-Priority Algorithm for Energy-Harvesting Real-Time Systems". In: *Proceedings of the Work in progress session of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pages 45–48.

(Cited on pages 1, 162).

[Abd+14] Yasmina Abdeddaïm, Younès Chandarli, R.I Davis, and Damien Masson. "Schedulability Analysis for Fixed Priority Real-Time Systems with Energy-Harvesting". In: *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*. 2014, pages 67–76.

(Cited on pages 1, 132, 249).

[AS+13] B. Ahmed-Seddik, G. Despesse, S. Boisseau, and E. Defay. "Self-powered resonant frequency tuning for Piezoelectric Vibration Energy Harvesters". In: *Journal of Physics: Conference Series* 476.1 (2013).

(Cited on page 53).

[Ahm+11] Masud Ahmed, Nathan Fisher, Shengquan Wang, and Pradeep Hettiarachchi. "Minimizing Peak Temperature in Embedded Real-Time Systems Via Thermal-aware Periodic Resources". In: *Sustainable Computing: Informatics and Systems*. Elsevier, 2011.

(Cited on pages 175, 177).

[AM01] André Allavena and Daniel Mossé. "Scheduling of Frame-based Embedded Systems with Rechargeable Batteries". In: *Proceedings of the International Workshop on Real-Time and Embedded Systeme (in conjunction with RTAS)*. IEEE, 2001.

(Cited on page 75).

[AC98] R. Amirtharajah and A.P. Chandrakasan. "Self-powered Signal Processing Using Vibration-based Power Generation". In: *Solid-State Circuits* 33.5 (1998), pages 687–695.

(Cited on page 51).

[Aud01] N. C. Audsley. "On Priority Asignment in Fixed Priority Scheduling". In: *Information Processing Letters* 79.1 (2001), pages 39–44.

(Cited on page 39).

[Aud+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. "Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling". In: *Software Engineering Journal* 8 (1993), pages 284–292.

(Cited on page 45).

[Aud91] N.C. Audsley. *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. 1991.

(Cited on page 39).

[BA14] Mario Bambagini and Hakan Aydin. "Periodic charging scheme for fixed-priority real-time systems with renewable energy". In: *Proceedings of the IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2014.

(Cited on page 119).

[BHR90a] Sanjoy K. Baruah, Rodney R. Howell, and Louis Rosier. "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor". In: *Real-Time Systems* 2 (1990), pages 301–324.

(Cited on page 42).

[BHR90b] Sanjoy K. Baruah, Rodney R. Howell, and Louis Rosier. "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor". In: *Real-Time Systems* 2 (1990), pages 301–324.

(Cited on page 46).

[Bar+12] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. "A Generalized Parallel Task Model for Recurrent Real-time Processes". In: *Proceedings of the IEEE Conference on Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2012, pages 63–72.
(Cited on page 35).

[BBA10] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. "Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability". In: *Proceedings of the Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (in conjunction with ECRTS)*. 2010.
(Cited on pages 149, 201).

[BD13] Adam Betts and Alastair Donaldson. "Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 2013.
(Cited on page 31).

[BB05] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30.1-2 (2005), pages 129–154.
(Cited on pages 100, 123, 147, 198, 212).

[BBB03] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. "Rate Monotonic Analysis: The Hyperbolic Bound". In: *IEEE Transactions on Computers Publication* 52.7 (2003), pages 933–942.
(Cited on page 44).

[Buc11] Isidor Buchmann. *Batteries in a Portable World*. Cadex Electronics Inc, 2011.
(Cited on pages 60, 61).

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer, 2011.
(Cited on pages 28, 35, 37, 234).

[Cal+06] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. "LITMUS-RT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers". In: *27th IEEE International Real-Time Systems Symposium*. 2006, pages 111–126.
(Cited on page 205).

[Can63] Edward T. Canby. *Histoire de l'électricité*. Rencontre, 1963.
(Cited on page 52).

[CC05] Roberto Casas and Oscar Casas. "Battery Sensing for Energy-Aware System Design". In: *IEEE Transactions on Computers Publication* 38.11 (2005), pages 48–54.
(Cited on page 57).

[CAM12] Younès Chandarli, Yasmina Abdeddaïm, and Damien Masson. "The Fixed Priority Scheduling Problem for Energy Harvesting Real-Time Systems". In: *Proceedings of the work in progress session of the the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2012.
(Cited on pages 1, 92).

[CFM15] Younès Chandarli, Nathan Fisher, and Damien Masson. "Response Time Analysis for Thermal-Aware Real-Time Systems Under Fixed-Priority Scheduling". In: *Proceedings of the IEEE Symposium On Real-time Computing (ISORC)*. 2015.
(Cited on pages 1, 174).

[CQF13] Younès Chandarli, Manar Qamhieh, and Frédérique Fauberteau. *YARTISS: A Generic, Modular and Energy-Aware Scheduling Simulator for Real-Time Multiprocessor Systems*. technical report. Paris, France, 2013. URL: http://hal.archives-ouvertes.fr/hal-01076022.
(Cited on page 2).

[Cha+12] Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms". In: *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2012, pages 21–26.
(Cited on pages 1, 205, 250).

[CHK07] Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. "On the Minimization Fo the Instantaneous Temperature for Periodic Real-Time Tasks". In: *Proceedings of the International Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2007, pages 236–248.
(Cited on page 175).

[CWT09] Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. "Proactive Speed Scheduling for Real-Time Tasks Under Thermal Constraints". In: *Proceedings of the International Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2009, pages 141–150.
(Cited on page 175).

[Che14] Maryline Chetto. "Optimal Scheduling for Real-Time Jobs in Energy Harvesting Computing Systems". In: *IEEE Transactions on Emerging Topics in Computing* (2014).
(Cited on pages 83, 86, 240).

[CMM11] Maryline Chetto, Damien Masson, and Serge Midonnet. "Fixed Priority Scheduling Strategies for Ambient Energy-Harvesting Embedded systems". In: *The IEEE/ACM International Conference on Green Computing an Communications*. 2011, pages 50–55.
(Cited on pages 95, 97, 111).

[Chi+00] Neil N. H. Ching, Gordon M. H. Chan, Wen J. Li, Hiu Yung Wong, and Philip H. W. Leong. "PCB Integrated Micro Generator for Wireless Systems". In: *Proceedings of the International Symposium on Smart Structures and Microsystems*. 2000.
(Cited on page 51).

[CG11] Pierre Courbin and Laurent George. "FORTAS: Framework fOr Real-Time Analysis and Simulation". In: *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2011, pages 21–26.
(Cited on page 206).

[DB09] Robert I. Dais and Alan Burns. "Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems". In: *Proceedings of the IEEE Conference on Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2009, pages 398–409.
(Cited on page 39).

[Dav93] R.I. Davis. *Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems*. 1993.
(Cited on page 128).

[DTB93] R.I. Davis, K.W. Tindell, and A. Burns. "Scheduling slack-time in fixed priority pre-emptive systems". In: *Proceedings of the International Real-Time Systems Symposium (RTSS)*. 1993.
(Cited on pages 92, 96, 127).

[DB11] Robert I. Davis and Alan Burns. "A Survey of Hard Real-time Scheduling for Multiprocessor Systems". In: *ACM Computing Surveys* 43.4 (2011), 35:1–35:44.
(Cited on page 43).

[Dav+13] Robert I. Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. "Schedulability Analysis for Controller Area Network (CAN) with FIFO Queues Priority Queues and Gateways". In: *Real-Time Systems* 49.1 (2013), pages 73–116.
(Cited on page 38).

[Dou+03] T. Douseki, Y. Yoshida, F. Utsunomiya, and N. more authors Itoh. "A batteryless wireless system uses ambient heat with a reversible-power-source compatible CMOS/SOI DC-DC converter". In: *Proceedings of the IEEE International Solid-State Circuits Conference, Digest of Technical Papers*. 2003, pages 388–501.
(Cited on page 54).

[EGCC11] Hussein EL Ghor, Maryline Chetto, and Rafic Hage Chehade. "A Real-time Scheduling Framework for Embedded Systems with Environmental Energy Harvesting". In: *Computers & Electrical Engineering* 37.4 (2011), pages 498–510.
(Cited on pages 83, 86, 240).

[ECLEZ11] L. El Chaara, L.A. Lamon, and N. El Zeinb. "Review of photovoltaic technologies". In: *Renewable and Sustainable Energy Reviews* 6 (2011), pages 242–247.
(Cited on page 50).

[Fau] Frédéric Fauberteau. "RTMSIM". `http://rtmsim.triaxx.org/`.
(Cited on pages 204, 207).

[FMG11] Frédéric Fauberteau, Serge Midonnet, and Laurent George. "Laxity-Based Restricted-Migration Scheduling". In: *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE Computer Society, 2011, pages 1–8.
(Cited on page 207).

[FBB06] Nathan Fisher, Theodore P. Baker, and Sanjoy Baruah. "Algorithms for Determining the Demand-Based Load of a Sporadic Task System". In: *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE Computer Society, 2006, pages 135–146.
(Cited on page 38).

[Gam11] Luca Gammaitoni. "Sustainable ICT: Micro and Nanoscale Energy Management". In: *Procedia Computer Science* 7 (2011), pages 103–105.
(Cited on page 54).

[GH+01] M. Gonzalez Harbour, J. J. Gutierrez Garcia, J. C. Palencia Gutierrez, and J. M. Drake Moyano. "MAST: Modeling and analysis suite for real time applications". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 2001.
(Cited on page 205).

[HP13] Damien Hardy and Isabelle Puaut. "Static Probabilistic Worst Case Execution Time Estimation for Architectures with Faulty Instruction Caches". In: *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2013, pages 35–44.
(Cited on page 31).

[HS00] P. Hazucha and C. Svensson. "Impact of CMOS technology scaling on the atmospheric neutron soft error rate". In: *IEEE Transactions on Nuclear Science* 47 (2000), pages 2586–2594.
(Cited on page 74).

[Het+12] Pradeep M. Hettiarachchi et al. "The Design and Analysis of Thermal-Resilient Hard-Real-Time Systems". In: *Proceedings of the International Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2012, pages 67–76. isbn: 978-0-7695-4667-4.
(Cited on pages 175, 197).

[Hol+10] M.J. Holenderski, M.M.H.P. van den Heuvel, R.J. Bril, and J.J. Lukkien. "Grasp: Tracing, visualizing and measuring the behavior of real-time systems". In: *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2010.
(Cited on page 205).

[JM06] R. Jayaseelan and T. Mitra. "Estimating the Worst-Case Energy Consumption of Embedded Software". In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2006.
(Cited on pages 68, 211).

[JLY89] Lehoczky John, Sha Lui, and Ding Ye. "The rate monotonic scheduling algorithm: exact characterization and average case behavior". In: *Proceedings of the IEEE Conference Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 1989.
(Cited on page 42).

[JK95] T. Juhnke and H. Klar. "Calculation of the soft error rate of submicron CMOS logic circuits". In: *IEEE Journal of Solid-State Circuits* 30 (1995), pages 830 –834.
(Cited on page 74).

[KRI09] Shinpei Kato, Ragunathan Raj Rajkumar, and Yutaka Ishikawa. *A Loadable Real-Time Scheduler Suite for Multicore Platforms*. Technical report. Graduate School of Information Science, Nagoya University, 2009.
(Cited on page 205).

[Kim+07] Sohee Kim, P. Tathireddy, R.A. Normann, and F. Solzbacher. "Thermal Impact of an Active 3-D Microelectrode Array Implanted in the Brain". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. IEEE Computer Society, 2007, pages 493–501.
(Cited on page 174).

[KS95] Gilad Koren and Dennis Shasha. "D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems". In: *Society for Industrial and Applied Mathematics Journal* 24.2 (1995), pages 318–339.
(Cited on page 206).

[KN04] H. Kulah and K. Najafi. "An Electromagnetic Micro Power Generator for Low-Frequency Environmental Vibrations". In: *Proceedings of the IEEE International Conference on Micro Electro Mechanical Systems (MEMS)* (2004).
(Cited on page 51).

[LaM+89] Joseph C. LaManna, Kimberly A. McCracken, Madhavi Patil, and Otto J. Prohaska. "Stimulus-activated changes in brain tissue temperature in the anesthetized rat". In: *Metabolic Brain Disease*. 1989, pages 225–237.

(Cited on page 174).

[Laz05] G. Lazzi. "Thermal effects of bioimplants". In: *IEEE Engineering in Medicine and Biology Magazine*. IEEE Computer Society, 2005, pages 75–81.

(Cited on page 174).

[LW82] J. Leung and J. Whitehead. "On the complexity of fixed-priority scheduling of periodic real-time tasks". In: *Performance Evaluation* 2 (1982), pages 237–250.

(Cited on pages 132, 147).

[LL73] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (1973), pages 46–61.

(Cited on pages 30, 33, 34, 38, 39, 40, 44, 46).

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

(Cited on page 46).

[MG01] C. Macq and J. Goossens. "Limitation of the hyper-period in real-time periodic task set generation". In: *Proceedings of the International Conference on Real-Time Systems*. 2001.

(Cited on pages 100, 123, 148, 198, 212).

[Mas] Damien Masson. "RTSS v1 and v2". https://svnigm.univ-mlv.fr/projects/rtsimulator/.

(Cited on pages 204, 206, 207).

[MM08] Damien Masson and Serge Midonnet. "Userland Approximate Slack Stealer with Low Time Complexity". In: *Proceedings of the International Conference on Real-Time and Network Systems (RTNS)*. 2008, pages 29–38.

(Cited on page 206).

[MVM07] L. Mateu, C. Villavieja, and F. Moll. "Physics-Based Time-Domain Model of a Magnetic Induction Microgenerator". In: *IEEE Transactions on Magnetics* 43.3 (2007), pages 992–1001.
(Cited on page 53).

[MPK86] Joseph Mathai and Pandya Paritosh K. "Finding Response Times in a Real-Time System". In: *The Computer Journal* (1986), pages 390–395.
(Cited on page 45).

[Mat+14] Simon Mathew et al. "Dye-sensitized solar cells with 13% efficiency achieved through the molecular engineering of porphyrin sensitizers". In: *Nature Chemistry* 6 (2014), pages 242–247.
(Cited on page 50).

[MSH11] John W. McCormick, Frank Singhoff, and Jérome Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, 2011.
(Cited on page 205).

[Men+01] Scott Meninger, Rajeevan Amirtharajah, Anantha P. Chandrakasan, Jeffrey H. Lang, and Jose Oscar Mur-Miranda. "Vibration-to-electric Energy Conversion". In: *IEEE Transsactions on Very Large Scale Integration Systems* 9.1 (2001), pages 64–77.
(Cited on page 52).

[Mit+04] P.D. Mitcheson, T.C. Green, E.M. Yeatman, and A.S. Holmes. "Architectures for vibration-driven micropower generators". In: *Microelectromechanical Systems* 13.3 (2004), pages 429–440.
(Cited on pages 51, 52).

[Mos+06a] C. Moser, D. Brunelli, L. Thiele, and L. Benini. "Lazy Scheduling for Energy Harvesting Sensor Nodes". In: *Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems*. 2006.
(Cited on pages 86, 87, 240).

[Mos+06b] C. Moser, L. Thiele, L. Benini, and D. Brunelli. "Real-Time Scheduling with Regenerative Energy". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. ECRTS '06. IEEE Computer Society, 2006, pages 261–270.
(Cited on pages 86, 91).

[Mos+07] C. Moser, D. Brunelli, L. Thiele, and L. Benini. "Real-time scheduling for energy harvesting sensor nodes". In: *Real-Time Systems* (2007). (Cited on pages 86, 87, 89).

[MW98] Frank Mueller and Joachim Wegener. "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints". In: *Real-Time Systems*. 1998, pages 144–154. (Cited on page 31).

[Now+14] Jan Nowotsch et al. *Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement*. Technical report. 2014. (Cited on page 31).

[PS05] Joseph A. Paradiso and Thad Starner. "Energy Scavenging for Mobile and Wireless Electronics". In: *IEEE Pervasive Computing* 4.1 (2005), pages 18–27. (Cited on page 56).

[PBB98] Trevor Pering, Tom Burd, and Robert Brodersen. "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms". In: *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 1998, pages 76–81. (Cited on pages 73, 74, 239).

[PSC04] G. Poulin, E. Sarraute, and F. Costa. "Generation of Electrical Energy for Portable Devices: Comparative Study of an Electromagnetic and a Piezoelectric System". In: *Sensors and Actuators A: Physical* 116.3 (2004), pages 461–471. (Cited on page 53).

[Qam+12] Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. "Global EDF scheduling of directed acyclic graphs on multiprocessor systems". In: *Proceedings of the Work-in-Progress Session of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2012. (Cited on page 207).

[QZ09] Gang Quan and Yan Zhang. "Leakage Aware Feasibility Analysis for Temperature-Constrained Hard Real-Time Periodic Tasks". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, 2009, pages 207–216.
(Cited on page 175).

[Qua+08] Gang Quan, Yan Zhang, William Wiles, and Pei Pei. "Guaranteed Scheduling for Repetitive Hard Real-time Tasks Under the Maximal Temperature Constraint". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2008, pages 267–272.
(Cited on page 175).

[RPK13] Ahmed Rehan, Ramanathan Parameswaran, and K.Saluja Kewal. "On Thermal Utilization of Periodic Task Sets in Uni-Processor Systems". In: *Proceedings of the work in progress session of the the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE Computer Society, 2013.
(Cited on page 175).

[RPK14] Ahmed Rehan, Ramanathan Parameswaran, and K.Saluja Kewal. "On Thermal Utilization of Periodic Task Sets in Uni-Processor Systems". In: *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, 2014.
(Cited on page 175).

[RW04] S. Roundy and P.K. Wright. "A piezoelectric vibration based generator for wireless electronics". In: *Smart Materials and Structures* 13.5 (2004).
(Cited on page 53).

[RWR03] Shad Roundy, Paul K. Wright, and Jan Rabaey. "A Study of Low Level Vibrations As a Power Source for Wireless Sensor Nodes". In: *Computer Communications* 26.11 (2003), pages 1131–1144.
(Cited on page 53).

[Rou+03] Shad Roundy, Brian P. Otis, Yuen-hui Chee, and Paul Rabaey Jan M.and Wright. "A 1.9GHz RF Transmit Beacon using Environmentally Scavenged Energy". In: *Proceedings of the International symposium on Low Power Electronics and Devices*. 2003.
(Cited on page 53).

[Rug+03] P.S. Ruggera, D.M. Witters, G. von Maltzahn, and H.I. Bassen. "In vitro Assessment of Tissue Heating Near Metallic Medical Implants by Exposure to Pulsed Radio Frequency Diathermy". In: *Physics in Medicine and Biology*. 2003, pages 2919–2928.
(Cited on page 174).

[RMM03a] Cosmin Rusu, Rami Melhem, and Daniel Mossé. "Maximizing Rewards for Real-time Applications with Energy Constraints". In: *ACM Transactions on Embedded Computing Systems (TECS)* 2 (2003), pages 537–559.
(Cited on page 75).

[RMM03b] Cosmin Rusu, Rami Melhem, and Daniel Mossé. "Multiversion scheduling in rechargeable energy-aware real-time systems". In: 2003, pages 95–104.
(Cited on page 75).

[SJ03] Roundy Shadrach Joseph. "Energy Scavenging for Wireless Sensor Nodes with a Focus on Vibration to Electricity Conversion". PhD thesis. University of California, Berclay, 2003.
(Cited on page 51).

[Sil99] Maryline Silly. "The EDL Server for Scheduling Periodic and Soft AperiodicTasks with Resource Constraints". In: *Real-Time Systems* 17.1 (1999), pages 87–111.
(Cited on pages 80, 81, 85, 240).

[Sin+04] F. Singhoff, J. Legrand, L. Nana, and L. Marc. "Cheddar: a Flexible Real Time Scheduling Framework". In: *The Special Interest Group on Ada*. 2004.
(Cited on page 205).

[Sin+09] Frank Singhoff, Alain Plantec, Pierre Dissaux, and Jérôme Legrand. "Investigating the usability of real-time scheduling theory with the Cheddar project". In: *Real-Time Systems* 43.3 (2009), pages 259–295.
(Cited on page 205).

[Ska+04] Kevin Skadron et al. "Temperature-aware microarchitecture: Modeling and implementation". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 1.1 (2004), pages 94–125.
(Cited on page 176).

[Sta06] Ingo Stark. "Invited Talk: Thermal Energy Harvesting with Thermo Life". In: *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*. IEEE Computer Society, 2006, pages 19–22.
(Cited on page 54).

[TYS01] T. Toriyama, M. Yajima, and S. Sugiyama. "Thermoelectric Micro Power Generator Utilizing Self-standing Polysilicon-metal Thermopile". In: *Proceedings of the IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*. 2001, pages 562–565.
(Cited on pages 54, 247).

[TRM10] Erick O. Torres and Gabriel A. Rincon-Mora. "Energy-harvesting chips and the quest for everlasting life". In: *IEEE Georgia Tech Analog and Power IC Design Lab* (2010).
(Cited on page 51).

[Tos01] Toshiba. *Lithium Ion Rechargeable Batteries specifications*. technical report. 2001. URL: www.tbcl.co.jp/TB_e/liion.htm.
(Cited on page 69).

[UDT09] Richard Urunuela, Anne-Marie Déplanche, and Yvon Trinquet. "STORM: a Simulation Tool for Real-time Multiprocessor Scheduling Evaluation". In: *Proceedings of the Workshop of GDR SOC SIP* (2009), page 1.
(Cited on page 206).

[Ven+07] Rama Venkatasubramanian, C. Watkins, D. Stokes, J. Posthill, and C. Caylor. "Energy Harvesting for Electronics with Thermoelectric Devices using Nanoscale Materials". In: *Electron Devices Meeting (IEDM)*. IEEE Computer Society, 2007, pages 367–370.
(Cited on page 54).

[VB+06] T. Von Buren et al. "Optimization of Inertial Micropower Generators for Human Walking Motion". In: *Sensors Journal* 6.1 (2006), pages 28–38.
(Cited on page 51).

[WAB10] Shengquan Wang, Youngwoo Ahn, and Riccardo Bettati. "Schedulability Analysis in Hard Real-time Systems Under Thermal Constraints". In: *Real-Time Systems* 46.2 (2010), pages 160–188.
(Cited on pages 175, 176, 177).

[WB06] Shengquan Wang and Riccardo Bettati. "Delay Analysis in Temperature-Constrained Hard Real-Time Systems with General Task Arrivals". In: *Proceedings of the International Real-Time Systems Simposium (RTSS)*. IEEE Computer Society, 2006, pages 323–334.
(Cited on page 175).

[WB08] Shengquan Wang and Riccardo Bettati. "Reactive Speed Control in Temperature-constrained Real-time Systems". In: *Real-Time Systems* 39.1-3 (2008), pages 73–95.
(Cited on page 175).

[Wei+94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. "Scheduling for Reduced CPU Energy". In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 1994.
(Cited on pages 73, 74, 239).

[Yea04] E. M. Yeatman. "Advances in Power Sources for Wireless Sensor Nodes". In: *proceedings of the International conference on Body Sensor Networks (BSN)*. 2004, pages 6–7.
(Cited on page 51).

[Yil11] Faruk Yildiz. "Potential Ambient Energy-Harvesting Sources and Techniques". In: *The Journal of Technology Studies* (2011).
(Cited on pages 51, 55).

[You+11] Hu Youfan et al. "Self-Powered System with Wireless Data Transmission". In: *Nano letters* (2011).
(Cited on page 53).

[ZA06] Dakai Zhu and Hakan Aydin. "Energy Management for Real-time Embedded Systems with Reliability Requirements". In: *Proceedings of the IEEE/ACM International Conference on Computer-aided Design*. ACM, 2006, pages 528–534.
(Cited on page 74).

[ZA09] Dakai Zhu and Hakan Aydin. "Reliability-Aware Energy Management for Periodic Real-Time Tasks". In: *IEEE Transactions on Computers* (2009).
(Cited on page 74).

[ZMM04] Dakai Zhu, R. Melhem, and D. Mosse. "The Effects of Energy Management on Reliability in Real-time Embedded Systems". In: *Proceedings of the IEEE/ACM International Conference on Computer-aided Design*. IEEE Computer Society, 2004, pages 35–40.
(Cited on page 73).

[ZQA07] Dakai Zhu, Xuan Qi, and Hakan Aydin. "Priority-Monotonic Energy Management for Real-Time Systems with Reliability Requirements". In: *Proceedings of IEEE International Conference on Computer Design*. 2007, pages 629–635.
(Cited on page 74).

[Zie14] J. F. Ziegler. *Trends in Electronic Reliability: Effects of Terrestrial Cosmic Rays*. http://www.srim.org/SER/SERTrends.htm. technical report. 2014. URL: http://www.srim.org/SER/SERTrends.htm.
(Cited on page 74).