ULB ECOLE POLYTECHNIQUE DE BRUXELLES

beams PARTS

Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems

par

Geoffrey Nelissen

Dissertation originale présentée à l'École polytechnique de Bruxelles
de l'Université libre de Bruxelles (ULB)
en vue de l'obtention du grade de
Docteur en Sciences de l'Ingénieur

Bruxelles, Belgique, 2012

Promoteurs :
Prof. Dragomir Milojevic
Prof. Joël Goossens

Membres du jury :
Prof. Pierre Mathys
Prof. Shelby Funk
Prof. Pascal Richard
Prof. Vandy Berten

Abstract

Real-time Systems are composed of a set of tasks that must respect some deadlines. We find them in applications as diversified as the telecommunications, medical devices, cars, planes, satellites, military applications, *etc*. Missing deadlines in a real-time system may cause various results such as a diminution of the quality of service provided by the system, the complete stop of the application or even the death of people. Being able to prove the correct operation of such systems is therefore primordial. This is the goal of the real-time scheduling theory.

These last years, we have witnessed a paradigm shift in the computing platform architectures. Uniprocessor platforms have given place to *multiprocessor* architectures. While the real-time scheduling theory can be considered as being mature for uniprocessor systems, it is still an evolving research field for multiprocessor architectures. One of the main difficulties with multiprocessor platforms, is to provide an *optimal* scheduling algorithm (i.e., scheduling algorithm that constructs a schedule respecting all the task deadlines for any task set for which a solution exists). Although optimal multiprocessor real-time scheduling algorithms exist, they usually cause an excessive number of task preemptions and migrations during the schedule. These preemptions and migrations cause overheads that must be added to the task execution times. Therefore, task sets that would have been schedulable if preemptions and migrations had no cost, become unschedulable in practice. An *efficient* scheduling algorithm is therefore an algorithm that either minimize the number of preemptions and migrations, or reduce their cost.

In this dissertation, we expose the following results:

- We show that reducing the "fairness" in the schedule, advantageously impacts the number of preemptions and migrations. Hence, all the scheduling algorithms that will be proposed in this thesis, tend to reduce or even suppress the fairness in the computed schedule.

- We propose three new online scheduling algorithms. One of them — namely, BF^2 — is optimal for the scheduling of sporadic tasks in discrete-time environments, and reduces the number of task preemptions and migrations in comparison with the state-of-the-art in discrete-time systems. The second one is optimal for the scheduling of periodic tasks in a continuous-time environment. Because this second algorithm is based on a semi-partitioned scheme, it should favorably impact the preemption overheads. The third algorithm — named U-EDF — is optimal for the scheduling of sporadic and dynamic task sets in a continuous-time environment. It is the first real-time scheduling algorithm which is not based on the notion of "fairness" and nevertheless remains optimal for the scheduling of sporadic (and dynamic) systems. This important result was achieved by extending the uniprocessor algorithm EDF to the multiprocessor scheduling problem.

- Because the coding techniques are also evolving as the degree of parallelism increases in computing platforms, we provide solutions enabling the scheduling of parallel tasks with the currently existing scheduling algorithms, which were initially designed for the scheduling of sequential independent tasks.

Acknowledgements

Writing a Ph.D. thesis is not only a matter of research. Writing a Ph.D. thesis is also a long human and psychological adventure of four years during which you sometimes encounter problems that can seem insuperable. In these difficult moments, it is your family, your friends and colleagues who give you the strength to go forward and not let everything down. It is thanks to them that I am finally able to present this document and I would like to thank them all from the bottom of my heart.

This thesis gave me the opportunity to travel and meet many people that I would have never had the occasion to rub shoulders with in other circumstances. For this, I am grateful to the FNRS and the BEAMS department for having funded my research, travels and visits to other researchers on the other side of the world. However, even though money made all these things possible, it is the human experiences shared with others which made them unforgettable. Hence, I would like to thank every people who played a role in my Ph.D. student life and I already apologize to any of them that I might have forgotten.

Behind any thesis, you have one person who launched the research and shared its experience and knowledge with the student. In my case they were two. The professors Dragomir Milojevic and Joël Goossens believed in me and allowed me to work on this so interesting topic. I thank you for that. Your terrific idea of merging research groups of two different worlds permitted me to discover the real-time scheduling theory. A topic that I would have found rather annoying four years ago but which fascinated me during my whole research. I would like to particularly thank Joël for his thorough review of each and every work I submitted. Even though I must admit that I initially sometimes found them excessive, they certainly are the reason that permitted most of our works to succeed. With you, I learned a lot.

Work is one thing but as I already said, you cannot keep on going without people ready to make you forget everything related to your research. The members of the BEAMS department are just perfect for that particular mission. I will never forget these crazy

nights and days in your company. Special thanks to Alexis, Martin, Gatien, Marc, Michel, Laurent, Kevin and Cédric for all these drinks and other warming houses during my two first years. I really felt as a member of a group since my first days in the department. I cannot cite everyone as so many people were part of the BEAMS during these four years but I cannot forget Yannick for all the french man-power he provided to us, Axel for his constant help, Frédéric Robert for his availability and his will to always improve our life into the department and Pierre Mathys who can answer any question about almost anything.

Thanks to my advisors, I had the chance to work in two different teams as I was also a member of the PARTS research group. Vandy, Irina, Markus, Vincent, Patrick, Anh Vu, I really appreciated to work with all of you. I am especially thankful to Vincent Nélis with whom I spent entire nights working on complex problems. I discovered a new interest in my thesis thanks to you. I will never be grateful enough for that.

During my thesis, I had this great opportunity of traveling all around the world. During one of these travels, I discovered a fantastic group of people in Porto. Your capacity to both work and party is just phenomenal. I spent a really nice stay in Portugal because of you and I wish to see you many times again.

I had a similar experience in Athens — this small town in the USA — when I visited Prof. Shelby Funk. I really enjoyed our interactive work. I would have never had imagine that one day I would spend a whole week-end with a professor in her bureau, just to meet the deadline of a work-in progress session. I was really impressed by your availability and your constant will to help your students. My stay at UGA was enriching from both an intellectual and human perspective.

Although the thesis is a life in itself, I would like to thank my friends who sometimes recalled me that there exists a social life outside the university.

Finally, many thanks to my family which supported me during these four years. I am especially grateful to my parents who permitted me to do these studies and without whom I would not have written this document.

Thank you all!

“À tout le monde,
À tous mes amis,
Je vous aime.
Je dois partir.
These are the last words
I’ll ever speak
And they’ll set me free.”

Megadeth

List of Publications

G. Nelissen and J. Goossens. A counter-example to: Sticky-erfair: a task-processor affinity aware proportional fair scheduler. *Real-Time Systems*, 47:378–381, 2011. ISSN 0922-6443. 10.1007/s11241-011-9128-7.

G. Nelissen, V. Nelis, J. Goossens, and D. Milojevic. High-level simulation for enhanced context switching for real-time scheduling in MPSoCs. In C. Seidner, editor, *Junior Researcher Workshop on Real-Time Computing*, pages 47–50, Paris, France, October 2009.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. An optimal multiprocessor scheduling algorithm without fairness. In *Proceedings of the 31th IEEE Real-Time Systems Symposium (Work in Progress session - RTSS10)*, San Diego, California, USA, December 2010. URL `http://cse.unl.edu/~rtss2008/archive/rtss2010/WIP2010/4.pdf`.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (Work in Progress session - RTSS11-WiP)*, December 2011a.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2011)*, pages 15–24. IEEE Computer Society, August 2011b.

G. Nelissen, S. Funk, J. Goossens, and D. Milojevic. Swapping to reduce preemptions and migrations in EKG. In E. Bini, editor, *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (Work in Progress session - ECRTS11-WiP)*, pages 35–38, Porto, Portugal, July 2011c.

G. Nelissen, S. Funk, J. Goossens, and D. Milojevic. Swapping to reduce preemptions and migrations in EKG. *SIGBED Review*, 8(3):36–39, September 2011d. ISSN 1551-3688.

G. Nelissen, S. Funk, D. Zhu, and J. Goossens. How many boundaries are required to ensure optimality in multiprocessor scheduling? In R. Davis and N. Fisher, editors, *Proceedings of the 2nd International Real-Time Scheduling Open Problems Seminar (RTSOPS 2011)*, pages 3–4, July 2011e.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS12)*, pages 321–330. IEEE Computer Society, July 2012a.

G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic. U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS12)*, pages 13–23. IEEE Computer Society, July 2012b.

G. Nelissen, S. Funk, and J. Goossens. Reducing preemptions and migrations in EKG. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012)*, pages 134–143, Seoul, South Korea, August 2012c. IEEE Computer Society.

Contents

List of Figures

List of Tables

CHAPTER

1 First Few Words

"If you can't explain it simply,
you don't understand it well enough."

Attributed to Albert Einstein

"Don't worry about a thing,
'Cause every little thing is gonna be all right."

Bob Marley

I could have started this document writing something like *"This thesis is dedicated to the study of the real-time system scheduling theory on multiprocessor platforms"*. However, with this first sentence depicting yet rather vaguely the goal of the work presented in this document, I know at least three words which could already discourage many people who are not experts in this field of research to go any further. Nonetheless, if at least one person who is not an expert, and despite the esoteric title of this thesis, decided to open it and start reading it, then I would feel really guilty of discouraging him of pushing forward in his investigations after only a few sentences. Therefore, I will be as didactic as possible in this first chapter to introduce the real-time scheduling theory to the reader. So I hope and I wish that everyone — expert or not in real-time systems scheduling theory — will be able to follow the discussions in the next chapters and have enough knowledge and comprehension of this research field to understand, if not everything, at least the goals, motivations and results presented throughout this document.

If you are an expert of real-time theory, you will perhaps prefer to directly start with the next chapter exposing more comprehensively the context and the contributions of this thesis. Nevertheless, I believe that everyone should have a chance to understand what is explained in the following, or may enjoy to "relearn" what he already knows under a different perspective.

The "real-time system scheduling" may seem to be a really complicated discipline intended to be understood by only a few initiated people. However, everyone is doing it

on an every day basis. We just do not know it.

Example:

Let us for instance take the example of Jack. It is Sunday afternoon and he is considering all the things he has to do during the next week. First, every morning, he has to drive his little daughter to the school before 8 a.m.. Then, as every week, his boss is waiting for a report before Monday 5 p.m.. There are also these two projects which must be finished and approved by his hierarchy by Thursday and Friday, respectively. Still, he has to find some time to pick up his daughter to school between 6 and 7 p.m..

As Jack is someone who likes to be prepared, he always organizes his next week. Hence, in his agenda, he starts every day driving his daughter to school between 7:15 a.m. and 7:45 a.m.. After having dropped his daughter at school on Monday, he should directly start writing the report for his boss until 3:00 p.m.. From that time and every other day of the week, he will work first on the project for Thursday and then on the project which has to be sent to the client by Friday. These tasks must of course be preempted every day at 6 p.m. to come back home with his daughter.

What Jack just did in this example, is "real-time scheduling". Indeed, every thing he has to do during the week is a *task*, and because these tasks all have a deadline, they are called *real-time tasks*. By organizing his next week, he created a *schedule*. Since this schedule holds for real-time tasks, it is a *real-time schedule*. Finally, the method that Jack used to decide when a task had to be done is called an *algorithm*. Hence, Jack used a real-time scheduling algorithm to schedule a system of real-time tasks.

Of course, Jack does probably not know it and he does certainly not care about it. However, it is a fact that whenever we have a set of tasks to do before some given deadlines and we take the decision to execute one of them first, we are actually doing "real-time scheduling". It is nothing more complicated and usual than that. We are all doing "real-time scheduling" every day of our life without even noticing it.

In our scientific jargon, we say that Jack is a *processor*. He indeed processes — i.e., executes — the tasks which are assigned to him. The problem of Jack is therefore a real-time scheduling problem on a *uniprocessor* system. That is, a system composed of only one processor. This kind of systems has already been extensively studied during these last four decades. Even though researchers are still working on some specific problems for uniprocessor systems, effective scheduling solutions already exist since 1969 [Liu 1969b].

This thesis rather considers the so-called *multiprocessor systems* in which multiple processors can execute the set of real-time tasks. To better understand this new problem-

atic, we propose an other example of an everyday situation.

Example:

> *Let us consider the case of Jack's boss. He has a team of three people under his orders. Every day, he has to take decisions for them and manage their work. This week, there are five projects that must be sent to the clients. He already decided to assign two of them to Jack, but he still has three projects but only two remaining people — say Lisa and John — to work on them. Unfortunately, it is impossible for only one person to work alone on two of these three projects and still respect the deadlines. Jack's boss therefore decides to make Lisa work on "project A" during the three first days of the week and then on "project B" during the last two days. John on the other hand, will work on "project B" until Tuesday afternoon and finish his week working on "project C". Hence, all deadlines should be met and accurate results sent in time to the clients.*

In this example, Lisa, Jack and John constitute a multiprocessor system. Their boss who decides when people must work on which task, is a real-time *scheduler*. That is, he takes scheduling decisions so as to respect all the projects (tasks) deadlines. As illustrated through this example, the problem becomes more complex as some tasks may have to be executed by more then one processor to be able to meet their deadlines.

Jack's boss, as Jack in the first example, would certainly like to find a scheduling algorithm — a method — which always produces a schedule respecting all the task deadlines every time that a solution exists. Such an algorithm is said to be *optimal*. There already exist optimal real-time scheduling algorithms for both uniprocessor and multiprocessor systems (e.g., [Liu and Layland 1973] and [Funk et al. 2011]).

One may therefore ask what should still be done since optimal solutions already exist. However, so far, there is one important thing that we did not take into account in our study. When a processor (John for instance) stops executing a task ("Project B" for instance) and starts working on an other task ("Project C"), it takes time to switch of context. John first has to save and classify everything he did for "project B" and then remember everything he did, or somebody else did, on "project C". This situation is called a *task preemption* and the time needed to handle the context switch is called the *preemption overhead*. This overhead must be added to the time initially needed to execute the tasks. The same happens when a task that was executed by one processor must then be executed by an other processor. This task *migration* can in some situations be even worse than a task preemption in terms of overheads. Often, current scheduling algorithms just ignore these overheads in their theoretical studies. Consequently, reducing as much as possible

Processor	=	Employee
Task	=	Project
Scheduler	=	Team manager
Scheduling algorithm	=	Method utilized to dispatch the projects

Table 1.1: Lexicon translating real-time theory vocabulary into equivalent words corresponding to the main example presented in this chapter.

the number of task preemptions and migrations that could occur during the schedule is an important matter for real-time systems if we want to be able to respect all task deadlines not only in theory but in practice too. It is exactly what this thesis intends to do.

For people who would like to read the thesis using the example of Jack's working team as a reference, we provide a short lexicon in Table 1.1 which translates the most important real-time related theoretical words by their equivalence in the situation of Jack.

Note that if we introduced the real-time scheduling theory in the context of a person or a group of persons who must organize their work, it is far from being the sole and certainly further again from being the main domain of application of this discipline. The real-time scheduling theory is used in domains as diversified as the aerospace, automotive, biomedical or telecommunication industries. Actually, any application with real-time constraints in which we want to automatize the process of decision making is a good candidate for the real-time scheduling theory.

Example:

Let us take the example of a car. It contains a large number of real-time tasks; the airbag, the anti-lock braking system (ABS) or the injection controllers for instances. All these tasks are critical for the safety of the users and the good operation of the car. If at least a subset of these tasks must execute on the same processors[1]*, then the utilization of a reliable real-time scheduling algorithm that can ensure the respect of all task deadlines is mandatory.*

The same considerations hold for the autopilot, the altimeter, the pressurization system and many other critical tasks in an airplane. If only one of these tasks fails, then a dramatic accident may happen. Real-time applications are manifold and all of them need scheduling algorithms on which we can rely.

[1] Although they are usually executed in different parts of the car, the automotive industry starts to consider to execute these various tasks on the same processors by reducing the number of processors available in the car.

In this thesis, we propose and describe new *optimal* algorithms solving various real-time scheduling problems but always keeping in mind the negative impact of task preemptions and migrations. Hence, we tackle various problems as different as the scheduling of *discrete-time systems* (i.e., systems in which a processor cannot execute a task for something else than an integer multiple of a fixed amount of time), the scheduling of *dynamic task systems* (i.e., systems where tasks may be added or removed from the scheduling problem at any time) or the scheduling of *multi-threaded parallel tasks* (tasks that may be executed by multiple processors simultaneously).

CHAPTER

2

Introduction to Real-Time Systems

"On fait la science avec des faits, comme on fait une maison avec des pierres : mais une accumulation de faits n'est pas plus une science qu'un tas de pierres n'est une maison."
("Science is built up of facts, as a house is with stones. But a collection of facts is no more a science than a heap of stones is a house.")

Henri Poincaré

"There's a lady who's sure all that glitters is gold
And she's buying a stairway to heaven.
When she gets there she knows, if the stars are all close
With a word she can get what she came for.
And she's buying a stairway to heaven."

Led Zeppelin

Contents

Abstract

Nowadays, many applications have real-time requirements. These systems for which the time at which the results are available is as important as the correctness of these results, are referred to as real-time systems. We can find such systems in domains as diversified as the telecommunications, bioengineering, chemistry, nuclear, spatial or automotive industry. The potential applications of the real-time scheduling theory are therefore large and manifold. However, they all have common determinant properties that can easily be enumerated and modeled.

In this first introduction chapter, we propose a brief description of the real-time systems. Of course, entire books are dedicated to this subject and we do not intend to cover each and every aspect of such a large topic. However, we present some examples of these systems that are more widely used every day. We provide the vocabulary needed to understand the following chapters and explain how these systems can be modeled so as to enable researchers to design efficient real-time scheduling algorithms. Some of the challenges in real-time scheduling theory are also introduced.

2.1 Introduction

In the first chapter, we briefly introduced the concept of real-time scheduling for multiprocessor systems using the example of a manager in a company organizing the work of his team. We however pointed out that this situation is not the main purpose of the real-time scheduling theory. In fact, the results presented in this thesis essentially intend to be utilized in computing systems and more particularly in *embedded systems*. Indeed, the number of such systems is growing every day. We find them in almost any device with automatic features such as cars, airplanes, smartphones, televisions, radio receivers, microwave ovens, ...

But what is exactly an embedded system?
Barr [2007] defines an embedded system as *"a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function."* The keywords here are "dedicated function". Hence, an embedded system is often contrasted with general purpose computers which can be programmed by the end-user to execute any kind of applications [Heath 2003; Barr 2007]. From this definition, we should understand that an embedded system is not limited to small devices such as phones or radio receivers. It can be any system with a well defined purpose such as the temperature controller of a power plant or the air traffic control system in an airport.

Nowadays, many embedded systems have real-time requirements. That is, they have to carry out tasks within fixed amounts of time which will later be defined as being the task deadlines. Such systems that we will refer to as *real-time systems* are divided in three broad categories:

Hard Real-Time Systems. In hard real-time systems, we cannot afford to miss any task deadline. If a task missed its deadline then the security of the system, the environment or the user would be jeopardized. Typical examples of hard real-time systems are the medical implants such as pacemakers which keeps the patient healthy, or the temperature controller of a power plant which may ultimately cause the explosion of the central if the temperature is not regulated at a predefined frequency. Other examples are the anti-lock braking system (ABS) or the airbag controller in a car which both have to react sufficiently fast to prevent the passengers from being hurt in case of an emergency braking or an accident. The results after deadline misses in hard real-time systems are not always so catastrophic. Nevertheless, the total failure of the system may lead to huge money loss due to repair and maintenance

costs.

Soft Real-Time Systems. A soft real-time system is the complete opposite of a hard real-time system. If a task misses its deadline in a soft real-time system then it does not directly impact the system safety and the application can simply keep on running. Even though the deadlines do not have to be mandatorily respected, these systems are real-time as they use real-time scheduling techniques to ensure some quality of service. Hence, application designers can for instance provide some tasks with higher priorities on other tasks and thus favor their completions.

Firm Real-Time Systems. Firm real-time systems are neither soft nor hard real-time systems but are lying somewhere in-between. In firm real-time systems, tasks are allowed to miss some deadlines. However, the number of successive deadline misses is upper bounded in order to keep a certain quality of service. Typically, firm real-time systems are systems that have to keep a given quality of service to please the end-user but will not cause failure or endanger the user safety if some task deadlines are missed. Audio-video decoders such as DVD or Blu-Ray players are typical examples of firm real-time systems. Indeed, ending the decoding of an audio or video frame a little bit later than the predefined deadline will degrade the quality of service experienced by the user but it will not jeopardize its safety neither the correct operation of the system.

In this thesis, we will only consider the scheduling of hard real-time systems. We therefore present scheduling algorithms ensuring that no deadline will ever be missed. Note that scheduling algorithms for hard real-time systems also work for the scheduling of soft or firm real-time systems since real-time constraints are stronger in hard than in firm or soft real-time environments. Thus, the solutions presented in this thesis can be used for the scheduling of any kind of real-time system. Nonetheless, because all task deadlines do not have to be necessarily respected, soft or firm real-time systems may envisage to overload the system, thereby executing more tasks than normally accepted by the platform. For the same reason, they may consider to use techniques improving the response time of non real-time tasks which might be executed on the same platform than the real-time application [Caccamo and Buttazzo 1998; Buttazzo and Caccamo 1999; Bernat et al. 2001; Liu et al. 2006]. However, these practices might cause deadline misses. They are therefore completely prohibited with hard real-time systems.

Real-time systems are typically composed of three different layers:

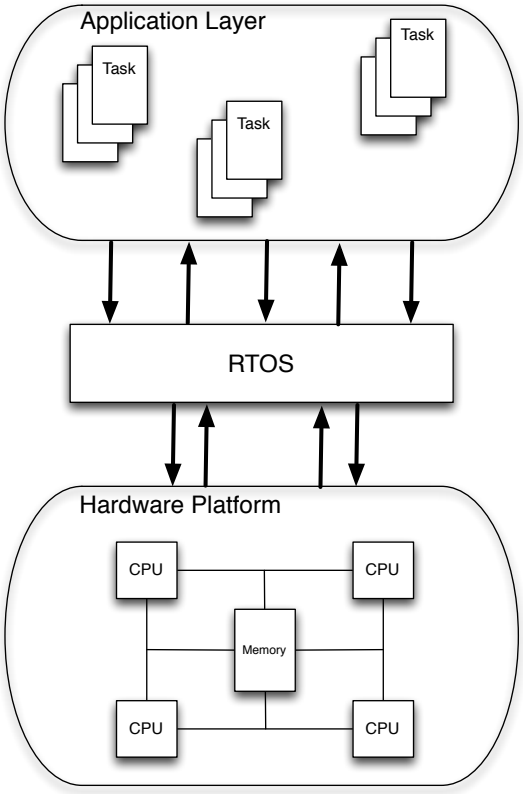


Figure 2.1: Real-time system with an RTOS acting as an interface between the application layer and the hardware platform.

- **An application layer** which is constituted of all the tasks that should be executed.
- **A real-time operating system** (RTOS) which makes the scheduling decisions and provides services to the application layer (e.g., communication tools, timers ...).
- **A hardware platform** which includes the processors (among other things such as memories, communication networks, *etc.*).

As shown on Figure 2.1, the real-time operating system acts as an interface between the hardware platform and the tasks of the application layer. Hence, it is the real-time operating system that decides when a task can be executed on which processor (sometimes called CPU for *central processing unit*). It is also the operating system that configures and manages the platform and the services it may provide. Therefore, the application can be developed independently of the platform on which it will run since the tasks never directly interact with the hardware and only the operating system needs to know the hardware properties. This three-layers model accelerates the product development cycle since

the hardware platform and the application tasks can be designed simultaneously and independently from one another. Moreover, legacy code (i.e., old application code that was written for previous products) can more easily be reused and integrated in new or upgraded products with minor or no modification at all. Similarly, the application maintenance and updates adding new functionalities or correcting bugs in the previous version of the application may be realized for multiple versions of a same product making use of different hardware platforms. For all these reasons, the presence of an operating system in a real-time system reduces the cost and the time needed for the development and the maintenance of a product. Furthermore, efficient scheduling algorithms can be used by the operating system, thereby ensuring that all the real-time task deadlines will be respected and thus providing a reliable solution for the implementation of real-time systems.

Analogy:

> *If we continue the analogy proposed in the first chapter, the application layer corresponds to all the projects that Jack's boss has to manage. The hardware platform is constituted by all the members of his working team (Jack, John and Lisa) and the operating system is represented by the boss himself. As mentioned in the previous paragraph, the boss indeed plays the role of an interface between the projects and the working team.*

We will now present each of these layers in detail and introduce the theoretical models enabling researchers to analyze these systems and design efficient scheduling algorithms which might be used by the real-time operating system to schedule the application tasks on the hardware platform.

2.2 The Application Layer

2.2.1 Characterization of a Real-Time Task

The application layer is composed of all the tasks that the system may have to execute. In an embedded system, these tasks are defined by a software program, i.e., a succession of instructions that must be executed by a processor in order to provide the expected result. Each new execution of a task is called a *job*. Hence, whenever the task — say τ_i — restarts its execution from the first instruction of its software code, then we say that the task τ_i *releases* a new job. Tasks are classified in three broad categories depending on the frequency at which the task releases jobs.

Periodic Task. As its name implies, a periodic task must be periodically executed. This means that the task τ_i is characterized by a *period* T_i and releases a new job *every* T_i time units. A periodic task τ_i is therefore constituted of a potentially infinite sequence of jobs (unless the task is removed from the real-time system at some point of its execution) released every T_i time units apart. Note that, regarding the instant at which the *first* job of each task is released, the systems composed of periodic tasks can still be further categorized as being *synchronous*, *asynchronous*, *non-concrete*, ...

Sporadic Task. Unlike a periodic task which *has to* be executed every T_i time units, a sporadic task τ_i is only characterized by a *minimum inter-arrival time* usually denoted by T_i either. This minimum inter-arrival time means that the amount of time separating two successive executions of τ_i must be *at least* of T_i time units. Hence, a sporadic task τ_i is also constituted of a potentially infinite sequence of jobs. However, the time separating the release of two successive jobs may be greater than T_i time units. Note that a periodic task is a particular case of the sporadic task model. A periodic task is indeed a sporadic task characterized by a minimum inter-arrival time T_i and regularly releasing a job every T_i time units.

Aperiodic Task. An aperiodic task does not have a period or minimum inter-arrival time. It can release a job at any time without any constraint on the amount of time separating two job releases. Aperiodic tasks are therefore the most difficult kind of tasks that could be scheduled in a real-time system. Indeed, we do not have any information on its behavior and we therefore cannot anticipate the arrival of one of its job while taking scheduling decisions. Consequently, in hard real-time systems, aperiodic tasks are usually used to model non real-time tasks that can be executed "when we have time" with what is called a *best-effort* scheduling policy. That is, we provide as much execution time as possible for the execution of the aperiodic task τ_i without jeopardizing the respect of all the real-time periodic and sporadic task deadlines [Buttazzo 2008].

Throughout this entire document, we will denote by τ the set of tasks that must be executed on the platform and we will denote by τ_i the i^{th} task of the set τ. Each real-time task τ_i is characterized by two key parameters: a *deadline* and a *worst-case execution time*.

Deadline. The deadline is certainly the most important parameter of a real-time task. As already mentioned, the deadline of a task τ_i specifies in how much time a job of τ_i

has to be executed in order to provide an appropriate result and not jeopardize the system, the environment or the user safety. It is therefore the deadline that defines the real-time behavior of a task.

The duration of the deadline of task depends on the nature of the application and does not necessarily have to be short. For instance, the deadline associated with a task controlling the opening of an airbag should be less than a second, while the deadline of a task controlling the temperature in a power plant may be defined in minutes. Thus, we should not mix up the notion of real-time with high performance or fast systems. In some cases a real-time task has to be executed "really fast" such as an airbag controller in order to prevent the passenger from being hurt during an accident. In this situation, the real-time system can be associated to a high performance system. However, industrial process controllers for instance may have deadlines defined in minutes like the temperature controller in the power plant. In such cases, we will not try to execute the controller faster than needed because it would consequently ask for more processing time, thereby handicapping other real-time tasks executed on the same platform. To conclude this discussion, real-time systems are more interested by the *predictability* of their behaviors rather than by high performances and fast executions.

The deadline of a task τ_i is noted D_i. Three different types of tasks exist depending on the relation holding between the period or minimum inter-arrival time and the deadline of the task:

Implicit Deadline Task. An implicit deadline task has a deadline equal to its period or minimum inter-arrival time (i.e., $D_i = T_i$)). Hence, the execution of a job of an implicit deadline task τ_i must be completed before the earliest possible release of the next job of τ_i.

Constrained Deadline Task. For a constrained deadline task τ_i, the deadline of τ_i is smaller than or equal to its period or minimum inter-arrival time (i.e., $D_i \leq T_i$). An implicit deadline task is therefore a particular case of a constrained deadline task.

Unconstrained Deadline Task. As its name implies, there is no constraint on the deadline of an unconstrained deadline task τ_i. We can either have $D_i \leq T_i$ or $D_i > T_i$. The unconstrained deadline task can be seen as a general representation of the deadline of a task, while constrained and implicit deadline tasks being two of its particular cases.

It is interesting to note that for both sporadic implicit and constrained deadline tasks, there is always at most one released job which has not yet reached its deadline at any time t. This property is however not necessarily true for unconstrained deadline tasks or aperiodic tasks.

Worst-Case Execution Time. The execution times of successive jobs released by a same task may vary in function of many parameters:

- First, the execution time of a job is highly impacted by the structure of the program describing its behavior. If a program contains many successive tests, the job execution time will probably depend on the test answers as the executed code will not be the same in the different cases.
- Second, the job execution time is influenced by the actual platform on which it is executed. The type of processors, the topology of the platform, the memory architecture, the communication network are various hardware parameters directly impacting the homogeneity of job execution times. For instance, for some memory architectures and configurations, there may be less cache misses during the execution of the second job released by a task than for the first one. Since a cache miss increases the execution time of a job (see Section 2.4.2), the second job will take less time to execute.
- The RTOS and its scheduling algorithm also impact the execution time of jobs as they may impose more or less preemptions and migrations to the jobs. As later explained in Section 2.5, preemptions and migrations cause overheads, which must be added to the initial execution time of a job. Since two successive jobs of a same task may experience different amounts of preemptions and migrations, their execution times may be different.
- Finally, the execution time of a job depends on the state of the system and the data that must be treated. For instance, the task controlling the altitude of an airplane will not need the same time to execute if the actual plane altitude is already correct or if it has to be rectified.

Therefore, still enforcing to be predictable (i.e., always respect some predefined behaviors) and provide proofs that a given scheduling algorithm will find a correct schedule respecting all task deadlines in all situations, each task is characterized by its execution time in the worst-case scenario. That is, we always take an upper bound on the execution time of the jobs released by a task. This so called worst-

case execution time of a task τ_i will always be noted C_i in the following of this document.

To summarize, a periodic or sporadic real-time task τ_i which is characterized by the triplet $\langle C_i, D_i, T_i \rangle$, releases an infinite sequence of jobs separated by at least T_i time units and in the worst-case, each job has to be able to execute C_i time units before its deadline occurring D_i time units after its release. If the task has an implicit deadline, we sometimes forget to specify the deadline D_i as it is equal to its minimum inter-arrival time T_i. In this particular case, we write $\tau_i \stackrel{\text{def}}{=} \langle C_i, T_i \rangle$.

Active Jobs and Tasks. In the remainder of this document, we will often refer to the *active jobs* and *active tasks* at a given instant t of the system execution. These jobs and tasks are defined relatively to their arrivals and deadlines. Hence, we have the following definitions.

Definition 2.1 (*Active Job*)

We say that a job is active at time t if it has been released no later than t and has its deadline after t, i.e., the instant t lies between the arrival time of the job and its deadline.

Definition 2.2 (*Active Task*)

A task τ_i is active at time t if it has an active job at time t.

Note that a job that has not reached its deadline yet, remains active even if it has already completed its execution.

Utilization and Density. Instead of using the worst-case execution time of a task τ_i, some scheduling algorithms sometimes prefer to manipulate its *utilization* or *density*. These two task properties are defined as follows:

Definition 2.3 (*Utilization*)

The utilization of a task τ_i is given by $U_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$. Informally, it represents the percentage of time the task may use a processor by releasing one job every T_i time units and executing each such job for exactly C_i time units.

Definition 2.4 (*Constrained Density*)

The constrained density of a task τ_i is given by $\lambda_i \stackrel{\text{def}}{=} \frac{C_i}{D_i}$. Informally, it represents the percentage of time the task may use a processor by executing a job for exactly C_i time

units between its release and its deadline occurring D_i time units later.

Definition 2.5 (*Generalized Density*)

The generalized density of a task τ_i is given by $\delta_i \stackrel{\text{def}}{=} \frac{C_i}{\min\{D_i,T_i\}}$ and is therefore the maximum between the utilization and the constrained density of τ_i.

Note that the task utilization gives the same value than the two types of task densities in the case of an implicit deadline tasks (i.e., $D_i = T_i$). Similarly, the generalized task density is identical to the constrained density for constrained and implicit deadline tasks (i.e., $D_i \leq T_i$). From this point onward, when we talk about the density of a task τ_i, we always refer to its generalized task density δ_i.

From these definitions, we also define the total utilization and the total density of a task set τ as follows:

Definition 2.6 (*Total Utilization*)

The total utilization of a tasks set τ is defined as $U \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} U_i$. Informally, it gives the minimum computational capacity that must be provided by the platform to meet all the task deadlines assuming that each task τ_i releases a job every T_i time units and each job executes for exactly C_i time units.

Similarly,

Definition 2.7 (*Total Density*)

The total density of a tasks set τ is defined as $\delta \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \delta_i$.

2.2.2 Sequential and Parallel tasks

Over the years, many different models have been proposed for representing the behavior of real-time tasks. They usually differ on the restrictions imposed on jobs. Hence, the *periodic implicit deadline* model (also often called model of Liu and Layland) imposes that a task releases a job periodically and that each job completes its execution before the next job release [Liu and Layland 1973]. The *sporadic* task model on the other hand only specifies a minimum inter-arrival time between two successive jobs [Mok 1983]. To these two models already exposed above, we can add

- the *multiframe* model which defines a sequence of job with different worst-case

execution times that are released cyclically by the same task [Mok and Chen 1997];

- the *generalized multiframe* (GMF) model which imposes different worst-case execution times, deadlines and minimum inter-arrival times for successive jobs of the same task [Baruah et al. 1999];

- and we can continue with the *non-cyclic GMF* [Tchidjo Moyo et al. 2010], the *recurring real-time* (RRT) [Baruah 2003] and the *non-cyclic RRT* models [Baruah 2010].

All the aforementioned works assume that the jobs are executed on only one processor at any time t and that a parallel execution of a same job on two or more processors is strictly forbidden. Tasks complying with this model of execution are commonly named *sequential* tasks as all their instructions are sequentially executed on processors without any possibility of simultaneous execution.

However, there exists another model of tasks in which different parts, usually called *threads*[1], of a same job may execute concurrently on the platform. There are two types of parallel tasks considering the constraints on the threads execution. Hence, with the *gang* model, threads of a same job must always be executed simultaneously. On the other hand, the *independent threads* model is more flexible and different threads of a same job do not have to start their execution synchronously. The problem of real-time parallel tasks scheduling is extensively treated in Chapter 7.

2.2.3 Static and Dynamic Task Systems

The task systems describing an application can be divided in two classes; they are either *static* or *dynamic*.

Definition 2.8 (*Static task system*)

With a static task system, the set of tasks that is executed on the platform is completely *defined* before *to start running the application.*

[1] In some operating systems, there is a difference made between a *process* and a *thread*. Two different processes do not share the same memory space. However, threads are created by processes and two threads of a same process can therefore share the resources that were defined by the process. In the acceptation utilized in this work, a task can either be a process or a thread. Hence, we can model the mechanisms of programming languages or API such as OpenMP that allow not only processes but also threads to be subdivided in a set of smaller threads [OpenMP Architecture Review Board 2011; Chandra et al. 2001].

Most of the scheduling algorithms assume that the task set is static, i.e., it never changes.

Definition 2.9 (*Dynamic task system*)

In a dynamic task system, some tasks may experience a modification *of their properties while other tasks* leave *or* join *the executed task set* at run-time.

The scheduling of dynamic task systems is an old problem studied for more than 20 years [Sha et al. 1989; Tindell et al. 1992]. There are many dynamic systems existing nowadays and their number is growing daily. We can cite the following examples:

- All the applications where the nature of the tasks to execute is only defined during the operation (e.g., multi-task robots sent on a catastrophe site to help victims or intervene on a dangerous situation);
- Systems that would be overloaded if all tasks were executed concurrently. The real-time operating system therefore applies algorithms at run-time deciding which tasks can join a particular cluster of processors.
- Other typical dynamic applications are the multimode systems. In a multimode system, the application is divided in many *application modes* [Nélis et al. 2009]. Each application mode is designed to operate in a different situation. Hence, a different set of tasks must be executed in each mode. However, some tasks may be shared between more than one application mode [Nélis et al. 2011]. A typical example of multimode system is the airplane. There is a different mode and thus a different set of tasks that must be executed for the take-off, the landing or during the flight. The auto-pilot is for instance not needed during the take-off and the landing procedure. On the other hand, the functions helping to correctly land or controlling the undercarriage can be deactivated during the flight. Other tasks may also be needed in case of emergency situations (e.g., depressurization of the cabin or a motor breakdown).

One of the key difficulty of dynamic task systems is the determination of the instants at which tasks can leave or join the system. A typical condition for allowing a task τ_i to leave the real-time system τ at time t is that τ_i cannot be active at time t [Srinivasan and Anderson 2005b]. Indeed, allowing τ_i to be removed from τ at time t if it has a job that has not reached its deadline yet, may sometimes lead to a momentary overload of the system which might cause deadline misses [Srinivasan and Anderson 2005b; Funk et al.

2011]. On the other hand, the condition on which a task may be added to the task set usually depends on the scheduling algorithm and the type of problem studied [Sha et al. 1989; Tindell et al. 1992; Srinivasan and Anderson 2005b; Nélis et al. 2009, 2011].

It has recently been shown that the scheduling of multi-mode systems may be quite difficult on multiprocessor platforms when some tasks are shared by more than one application mode [Nélis et al. 2011]. Therefore, the scheduling of dynamic task systems should not be underestimated and efficient solutions still have to be designed.

2.3 The Real-Time Operating System

The real-time operating system has three main purposes:

- Configure and manage the hardware platform (e.g. manage hardware interrupts, hardware timers, memory accesses, ...);
- Schedule the tasks using a real-time scheduling algorithm;
- Provide tools to the tasks to communicate and synchronize between each others (using timers, mutexes, semaphores, mailboxes, ...).

Note that, excepting the fact that an RTOS might use more complex scheduling algorithms providing better real-time guarantees, the three aforementioned features do not differ from the mission of a typical non real-time operating system such as Microsoft Windows or Mac OS X. There is however a major difference between a non real-time and a real-time operating system; a real-time operating system has to be *reliable* and *predictable*. By predictable, we mean that the real-time operating system must be able to provide an upper bound on the execution of any of its internal routines. Hence, it has to provide the maximum number of *processor ticks* needed to execute the scheduling algorithm, perform an execution context switch or handle an interrupt for instance. A real-time operating system also has to be reliable. Hence, we must be certain that:

1. the real-time operating system will always respect the provided execution upper bounds;
2. it will never initiate a failure (superbly represented by the famous “blue screen” of Windows 95 for instance) that could cause the discontinuation of the execution

of hard real-time tasks, which could ultimately lead to catastrophic results for the application, the environment or the user.

2.3.1 The Scheduler

Since this thesis intends to provide new scheduling algorithms, we will essentially focus on the presentation of the scheduler which takes the real-time scheduling decisions and hence determines in which order the tasks must be executed on the processors so as to respect all the task deadlines. We assume that a scheduler makes use of one particular scheduling algorithm to takes its scheduling decisions. Therefore, the scheduler properties are induced by the chosen scheduling algorithm properties. Thus, we will use the terms scheduler and scheduling algorithm interchangeably in the rest of the thesis.

Note that the real-time scheduling theory and its history will be more extensively studied in the next chapter.

Classification of Schedulers

Many different scheduler classifications exist [Carpenter et al. 2004; Davis and Burns 2011]. One of them is the distinction made between *online* and *offline* scheduling algorithms. An online scheduling algorithm takes its scheduling decisions during the operation of the application. Hence, at any time t, it bases its scheduling choices on the current state of the system. On the other hand, an offline scheduler computes an entire schedule before to start executing the application. An exact knowledge of the task properties and behaviors is therefore needed. These scheduling algorithms are thus usually limited to the scheduling of static and periodic task systems.

Since the distinction between online and offline schedulers is usually not sufficient to completely define a scheduler, we finally decided to adopt the categorization presented in [Nélis 2010] which divides the scheduling algorithms in two broad categories: the *priority driven* and the *time driven* schedulers.

Priority Driven Scheduler. A priority driven scheduling algorithm is a scheduling algorithm assigning priorities to the jobs and executing at any time t the m jobs with the highest priorities on the m processors composing the platform (with m, a natural greater than 0). We assume that the job priorities may vary during the execution. Note that this definition differs from the definition provided in [Ha 1995; Goossens

et al. 2003] that defines a priority driven scheduling algorithm as being an algorithm where a job keeps the same priority between its arrival and its deadline.

There are many algorithms following this scheduling policy. One may cite DM [Leung and Whitehead 1982], EDF [Liu and Layland 1973], LLF [Mok 1983], LLREF [Cho et al. 2006], LRE-TL [Funk 2010], ...

The priority driven scheduling algorithms may be of three different types:

Fixed Task Priorities With fixed task priorities (FTP) schedulers, the priority is statically assigned to the task. Every job released by this task inherits the task priority. Therefore, all jobs released by a same task have the same priority. Furthermore, this priority never changes. Typical examples of FTP algorithms are RM [Liu and Layland 1973] and DM [Leung and Whitehead 1982].

Fixed Job Priorities With fixed job priorities (FJP) schedulers, the priorities are assigned to the jobs rather than to the tasks. Hence, each job released by a same task might have a different priority. However, once assigned, the priority of a particular job cannot vary between its arrival time and its deadline. One may cite EDF [Liu 1969b] and its variants such as EDDP [Kato and Yamasaki 2008b], EDF with C=D [Burns et al. 2010, 2011], EDF$^{(k)}$ [Goossens et al. 2003] or EDF-US$[\zeta]$ [Srinivasan and Baruah 2002] as examples of FJP algorithms.

Dynamic Job Priorities The most general model of priority driven scheduling algorithms is the dynamic job priorities (DJP) scheduler. With this type of schedulers, the priority of a particular job may vary during its execution. Many scheduling algorithms follow this unconstrained scheduling scheme. Hence, we can cite LLF [Mok 1983], LLREF [Cho et al. 2006], LRE-TL [Funk and Nadadur 2009; Funk 2010], PF [Baruah et al. 1993, 1996], PD^2 [Anderson and Srinivasan 1999], ...

It is interesting to note that FTP schedulers are a particular case of FJP schedulers, and FJP schedulers are a particular case of DJP schedulers. Hence, FTP schedulers such as DM could also be categorized as being FJP and DJP but the contrary is not necessarily true.

Time Driven Scheduler A time driven scheduler does not assign priorities to the jobs. It makes use of a table defining the order and the time for which the jobs have to be executed on the platform. The schedule is then computed *a priori* and the scheduler only follows the decisions previously taken.

We should insist on the fact that a time driven scheduler is *not* necessarily synonym of an *offline* scheduling algorithm. That is, it does not imply that the scheduling decisions must be taken at design time before the application startup. Furthermore, a time driven scheduler does not impose to know all the task properties and job arrival times either. The sporadic version of the DP-Wrap scheduling algorithm for instance computes a new schedule that must be saved in a table at each new job arrival [Levin et al. 2010; Funk et al. 2011]. These scheduling decisions are then scrupulously followed by the time driven scheduler until a new job arrives and a new schedule to be computed.

The algorithms BF [Zhu et al. 2003, 2011] and SA [Khemka and Shyamasundar 1997] are two other examples of time driven schedulers. The BF algorithm is an *online* scheduling algorithm (i.e., taking scheduling decisions during the execution of the tasks) while SA is an *offline* algorithm.

There exist nonetheless scheduling algorithms such as EKG [Andersson and Tovar 2006; Andersson and Bletsas 2008] or NPS-F [Bletsas and Andersson 2009, 2011] that cannot be categorized as being priority driven or time driven. Actually, the two previously cited algorithms group the tasks in what is called *servers* or *supertasks*. A server (or supertask) is a group of tasks that must be considered as a single entity during the schedule. With EKG and NPS-F, the servers are scheduled using a time driven scheduling algorithm. However, whenever a server must be running on the platform, a priority driven scheduling algorithm is used to decide which component task of this server must actually be executed on the platform. This kind of scheduling algorithm is called a *hierarchical scheduler*. Indeed, there is a hierarchy in the schedule. We start using a first scheduling algorithm to schedule the servers and then a second scheduling algorithm (which may be different from the first one) to schedule the component tasks within each server. Note that a hierarchical scheduler might have more than two levels of scheduling. For instance, the algorithm RUN [Regnier et al. 2011] can have an arbitrary number of scheduling levels. In the case of RUN, tasks are grouped in servers that are themselves grouped in servers of servers.

In the following, we will also distinct between *preemptive* and *non-preemptive* schedulers.

Definition 2.10 (*Preemptive scheduler*)

A preemptive scheduler allows the execution of a job to be suspended while it is running on the platform, so as to start executing another job. The execution of the

suspended job may be resumed later.

Definition 2.11 (*Non-preemptive scheduler*)

With a non-preemptive scheduler, a job cannot be interrupted by another job once its execution has started. Therefore, a job always runs until its completion unless it is self-suspended to wait for data, synchronize with an other task or wait an access to a shared resource.

These definitions directly lead to the notion of *preemption* defined as follows.

Definition 2.12 (*Preemption*)

A preemption happens when the execution of a job is suspended on a processor in order to start executing another job on the same processor.

We also define a *migration* and *instantaneous migration* as:

Definition 2.13 (*Migration*)

A migration happens when the execution of a task is suspended on one processor and is resumed on another processor.

Definition 2.14 (*Instantaneous migration*)

An instantaneous migration happens when a task is suspended on one processor and the operating system instantaneously *resumes its execution on another processor.*

Note that even though the migration may be instantaneous from the operating system perspective, it actually takes time to migrate the execution context of a task from one processor to another (see Section 2.4.2). Therefore, a task can never stop running on one processor and instantaneously restart its execution on another processor in a realistic system.

2.3.2 Time Management

To manage the task executions, the scheduler needs tools to measure the time. This requirement is independent of the scheduling algorithm and is certainly not limited in any way to the time driven schedulers. Indeed, let us imagine an application — a controller in a robot on an industrial production line for instance — with tasks that must be executed periodically with a period exactly equal to 2 seconds. In order to actually release a job every 2 seconds, the real-time operating system needs to know how much time elapsed

since the last job release. This consideration is the same whatever the scheduler utilized by the operating system.

Algorithm 2.1 provides the code of a typical task as it would be written in most real-time operating system.

Algorithm 2.1: Typical task code.

```
1 int task1(void*)
2 begin
3     while true do
4         do something;
          ⋮
5         sleep_until_next_period();
6     end
7     return 0;
8 end
```

Each iteration of the infinite loop corresponds to the execution of a new job released by the task. The function *sleep_until_next_period()*, at least partially provided by the operating system, has two goals:

- inform the scheduler that the execution of the job is completed;
- configure a timer (i.e., a counter counting the elapsed time since its last configuration and causing an event when the configured time has been reached) that will wake up and reactivate the task when the next job must be released.

To evaluate the elapsed time, the timer configured by the operating system needs a time reference. This base of time is provided by the hardware platform. Every processor based computing platform includes a *clock*. This clock has a predefined frequency and it periodically emits a *tick*. Then, the operating system can program a *hardware timer* to count the *clock ticks* so as to periodically generate an interrupt. This interrupt, often referred to as the *system tick*, is then used as a reference by the operating system for a precise time management. The time elapsed between two system ticks is called the *system time unit*. Of course, the precision needed for the system time unit is dependent on the application and can therefore be chosen by the application designer. Hence, the operating system configures the hardware timer in function of the application needs. The only constraint lies in the fact that the system time unit must be a natural multiple of

the time elapsed between two consecutive clock ticks [Krten and QNX Software Systems 2012; Wind River Systems, Inc. 2011].

Example:

Let us assume that our computing platform has a clock with a frequency of 50 *MHz. This means that the clock produces a tick every* $\frac{1}{50{,}000{,}000}\ s = 0.02\ \mu s$. *Let us now consider that the application designer wants a system time unit of* 10*ms. The operating system therefore configures the hardware timer so as to divide the clock frequency by* 500000 *thereby generating an interrupt every* $0.02\ \mu s \times 500000 = 10\ ms$. *This interrupt is then treated by the operating system to increment a software counter called* system clock, *representing the time elapsed since the application startup.*

Note that in some cases, the expected value for the system time unit cannot exactly be obtained. For instance, it is explained in [Krten and QNX Software Systems 2012] on page 88, that in some IBM PC hardware, requesting for a 1ms time unit actually leads to 0.999847ms between two system ticks. If this error is not taken into account by the tasks or the operating system, it can lead to dangerous deviations on the long term execution.

2.3.3 Discrete-Time Systems

As already said, a timer is a counter that generates an event when a predefined amount of time elapsed. The timers can either be hardware or software. A hardware timer uses the clock ticks as a time reference while the software timer uses the system tick provided by the operating system.

There exist two types of operating systems regarding the kind of timers that can be accessed and programmed by the application and the RTOS:

- In the most complex real-time operating systems such as the real-time variants of Linux (e.g., Litmus^{RT} [Calandrino et al. 2006; Brandenburg 2011] and RTLinux [Yodaiken and Barabanov 1997; Barabanov 1997]), the application and the operating system can program both software and hardware timers. The hardware timers are referred to as *high-resolution timers* since their resolution equals the clock precision (which can be even smaller than 1 nanosecond). However, the number of high-resolution timers is limited and they must therefore be utilized with parsimony.

- Most real-time operating systems that are not based on Lynux such as RTEMS [RTEMS 2012], VxWorks [Wind River Systems, Inc. 2011], QNX Neutrino [Krten

and QNX Software Systems 2012] or LynxOS [Lynux Works 2005], do not implement high-resolution timers. We should however be careful while reading the documentation of these RTOS since the names of some functions may be misleading. For instance, even though QNX Neutrino provides a function named **nanosleep()** which is supposed to suspend the execution of the task during a time specified in nanoseconds, it is said on page 1553 in [QNX Software Systems Limited 2012] that *"The suspension time may be longer than requested because the argument value is rounded up to be a multiple of the system timer resolution"*. Similarly, for VxWorks, it is said on Section 9.9 of [Wind River Systems, Inc. 2011] that *"The POSIX nanosleep() routine provides specification of sleep or delay time in units of seconds and nanoseconds, in contrast to the ticks used by the VxWorks taskDelay() function. Nevertheless, the precision of both is the same, and is determined by the system clock rate; only the units differ"*. These functions have been implemented for compliancy reasons with the POSIX real-time standard [IEEE 2003] but do not actually provide a better resolution than the software timers based on the system ticks.

Because the number of high-resolution timers is limited with the first presented class of operating systems, we cannot consider managing all periodic task periods with such high resolution timers. We can thus reasonably assume that all job arrivals of periodic tasks are synchronized on the system ticks (whatever the RTOS utilized to manage the tasks). Furthermore, with the second class of operating systems, the operating system cannot program a timer to stop the execution of a task after a time that is not a natural multiple of the system time unit. Hence, such RTOS can only schedule the execution of tasks for natural multiples of the system time unit.

These constraints gave birth to the notion of *discrete-time systems* where task properties (i.e., periods, worst-case execution times and deadlines) and their execution times determined by the scheduling algorithm must be defined as natural multiples of the system time-unit.

The discrete-time systems are contrasted with the *continuous-time systems* where execution times of the tasks may be any real number of system ticks. These continuous-time systems can however only be handled by the most complex RTOS usually based on Linux, and are therefore limited in their choice of implementations.

2.4 The Hardware Platform

The hardware platform includes many different hardware elements such as processors, memories, communication networks, clock, timers, ... All of them have an impact on the performances and the behavior of real-time systems. However, given the breadth of this topic, we will not talk about most of them since they are beyond the scope of this work. Instead, we will essentially focus on those directly impacting the RTOS execution; first clearly defining what we call a processor, and then rapidly explaining the impact of the memory architecture on the worst-case execution time of the tasks executed upon the hardware platform.

2.4.1 Processing Elements

The first thing we should probably do is clarifying all these notions of chip, processors, cores, multiprocessors and multicore architectures that we can encounter in the literature and which are often confusing. The term processor is sometimes utilized to refer to an entire *chip* executing the application. This outdated denomination is a reminiscence of the vocabulary used more than ten years ago when most chips still contained only one processing unit. However, in recent years we have witnessed a major paradigm shift regarding the computing system architectures. Obtaining better performances by increasing the speed of the logic has given way to *multicore* architectures, exploiting parallelism rather than execution speed. Hence, today, multicore platforms are obtained by integrating multiple *cores* on the same chip, thereby exploiting the increasing density of transistors in a single chip. Each core can execute its own task completely independently of what other cores may execute.

From this point onward, when we talk about a *processor*, we actually refer to the smallest element in the hardware platform which is able to process the execution of a job. That is, a processor refers to a single core, even if there are many in the same chip.

Platform Homogeneity

Multiprocessor platforms may be categorized in three different classes depending on the homogeneity of their processors.

Identical Multiprocessor Platform. As its name implies, on an identical multiprocessor

platform all the processors are interchangeable. Hence, the worst-case execution time of a task is not impacted by the particular processor on which it is executed. Also, we assume that all tasks can be executed on all processors without any restriction. This type of platform is sometimes referred to as *symmetric multiprocessor platforms* (SMPs).

Uniform Multiprocessor Platform. Uniform multiprocessor platforms are platforms where processors have the same configuration — i.e., they all have the same functional units, cache sizes and hardware services — but they are running at different frequencies. Hence, all processors can execute all tasks but the speed at which they are executed and therefore their worst-case execution time vary in function of the processor on which they are running.

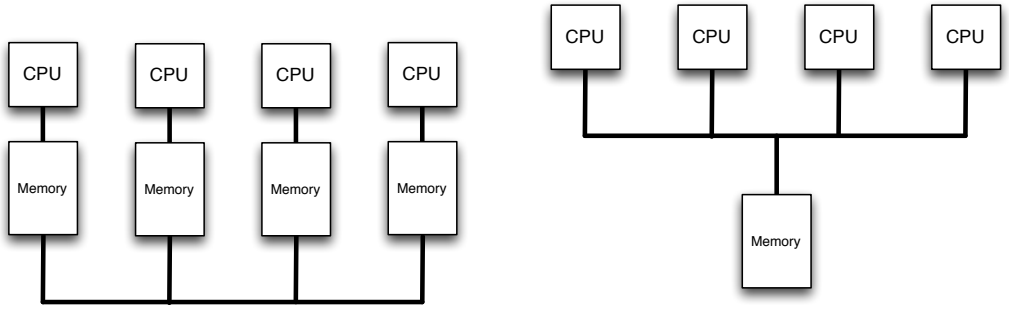
Unrelated Multiprocessor Platform. It is the platform with the highest degree of heterogeneity. All processors may have different configurations, frequencies, cache sizes or instruction sets. Some tasks may therefore not be able to execute on some processors in the platform, while their execution speeds (and their worst-case execution times) may differ on the other processors.

In the following of this document, we only consider *identical* multiprocessor platforms.

2.4.2 Memory Architecture

There exist two main categories of computing platforms when considering the memory architecture, namely the *distributed memory* and *shared memory* architectures. In a distributed memory architecture, each processor has its own local memory (see Figure 2.2(a)). Consequently, a processor cannot directly access the data and instructions stored in the memory of another processor. On the other hand, shared memory architectures have one central memory space which is accessible by all processors (see Figure 2.2(b)).

This difference on the memory access mode has a direct impact on the time needed for a task to migrate from one processor to another. Hence, in shared memory architectures, all tasks are stored in the central memory and are therefore accessible by all processors. However, for distributed memory architectures, the execution context of a task cannot directly be accessed by a processor π_a if this task was previously running on another



(a) (b)

Figure 2.2: (a) Distributed memory architecture. (b) Shared memory architecture.

processor π_b. All the data and the entire task program must first be copied from π_b's memory to π_a's memory, and only then, the task will be able to resume its execution on π_a. This migration of the task context from one processor to another may be extremely time costly (depending on the size of the task execution context). Furthermore, it may also transiently overload the communication infrastructure between processors due to the huge amount of data that must transit between local memories.

Nevertheless, a single central memory cannot afford to serve all the processor reading and writing requests at the same time. Moreover, since the central memory needs to store all the task programs and data, it must have a bigger capacity than local memories in distributed memory architectures. However, bigger memory space usually means slower memory access. Consequently, the central memory may rapidly become a bottleneck in the shared memory architecture.

Because the memory integration and their access performances did not follow the same trends than the processor integration and speed, modern multiprocessor platforms usually use a *hierarchical memory* architecture. With such an architecture, small and fast local memories called *caches* are placed near the processors to alleviate the latencies of a slow central memory. Whenever an instruction or data is accessed by a task running on a processor, this instruction (or data) is saved in the local cache of this processor. Hence, the next time that the task will ask for this instruction (or data), it should be ready in the fast local cache (assuming that the task has not migrated between the two requests). Consequently, the task will not have to make a second request to the slow central memory which must serve all the processors of the platform. However, to be fast, local caches should be relatively small and could therefore not be able to save the entire task code and/or all its data. *Direct mapped*, *fully associative* and *set associative* caches are different

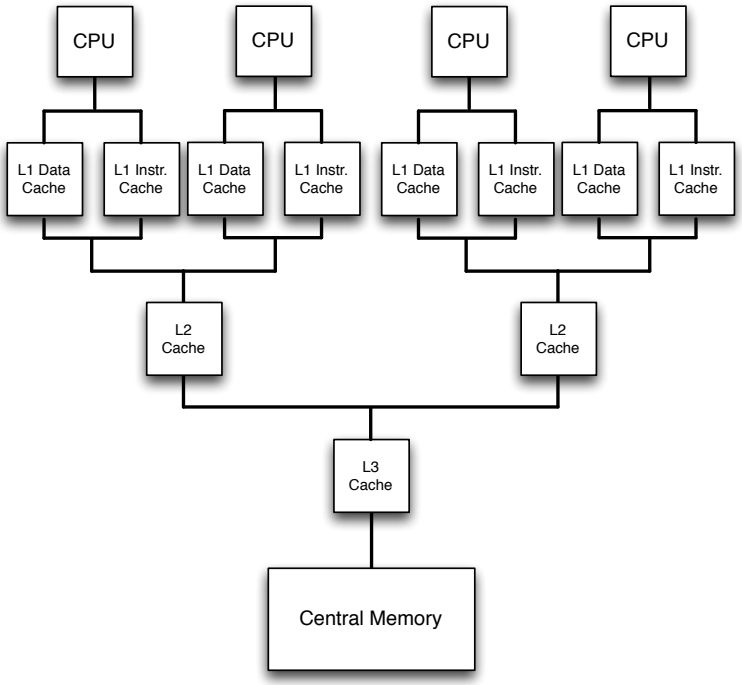


Figure 2.3: Typical memory architecture of modern multiprocessor platforms.

type of caches using different protocols to decide which previously stored data must be replaced by a new data that has just been requested by the running task but is not yet saved in the cache. Whenever a task that is running on a processor tries to read an instruction or data that is not stored in the cache, we say that there is a *cache miss*. A cache miss means that the time to access the requested instruction or data will be significantly higher than the time needed to access a data already stored in the cache (i.e., if there is a *cache hit*).

In practice, the memory architecture may be composed of many *levels* of caches. The smallest and fastest cache which is also the nearest memory from the processor is called the *level 1 cache* (L1). On level 2, we find the L2 and then the L3 on level 3. The L2 cache is usually bigger and therefore save more data than the L1 but may be slower. Both are usually made of SRAM and integrated on-chip. If present, the L3 cache may either be on-chip or off-chip, and made of a combination of SRAM and DRAM. The L3 is usually bigger and slower than the L2. Hence, the higher the requested data is in the memory hierarchy, the longer will be the time needed to bring the data back to the processor. A typical example of memory architecture is provided in Figure 2.3.

Some caches may be exclusively reserved for data or instructions. We then have independent *data caches* and *instruction caches*. If a cache level does not distinct between

data and instructions then we say that this cache is *unified*.

The last problem that has not been addressed so far, is the *cache coherency*. Indeed, because copies of data are saved in local cache memories, we must make sure that a modification of a data in the cache of one processor will be propagated in all the other caches that may contain a copy of this data. However, this difficult problem is far beyond the scope of this work. Also, we always assume in the following of this document that all caches are kept coherent during the application execution. Relevant works on this matter can be found in [Hennessy and Patterson 2006; Baer 2010].

2.5 The Problem of Task Preemptions and Migrations

Let us consider the case of a task τ_a running in *isolation*, i.e., running alone on the platform. In this situation, τ_a has a given worst-case execution time C_a.
Now, let us assume that τ_a is scheduled with other tasks on the platform and another task — say τ_b — preempts τ_a during its execution on a processor denoted by π_j. After the completion of τ_b, the task τ_a can be resumed on processor π_j. However, because τ_b executed on π_j, it is more than likely that the instructions and data of τ_a that were stored in the L1 cache have been replaced by the instructions and data of τ_b. Thus, when resumed, the task τ_a must look for its instructions and data in higher levels of caches or even in the central memory. The worst-case execution time of τ_a is therefore increased in comparison to its execution time in isolation due to the increased number of cache misses.
Furthermore, if τ_a had to migrate on another processor — say π_k — then it would have had to reload all the caches of π_k with its instructions and data. In this case, τ_a cannot take advantage of the instructions and data that were potentially still available in L2 or L3 caches of π_j (unless these caches are shared between π_j and π_k). A task migration is then potentially even more costly than a simple task preemption (as shown on Figures 2(b) and 2(d) in [Bastoni et al. 2010a]).

All these preemption and migration *overheads* must be added to the initial worst-case execution times of the tasks [Petters and Färber 2001]. These overheads may be really high as shown through an empirical study in [Bastoni et al. 2010a] where preemptions and migrations delays may reach up to 10 ms on a platform of 24 cores running at 2.13 GHz and providing three levels of caches. Hence, adding these overheads to the initial worst-case execution times of the tasks, systems that were initially schedulable in theory while respecting all their deadlines, may become unschedulable in practice. This

is confirmed by the study made in [Bastoni et al. 2011], which shows that these overheads may dramatically impact the performances of real-time systems (some scheduling algorithms may not be able to schedule any task set in some situations). This impact however depends on:

- the particular scheduling algorithm utilized by the operating system [Bastoni et al. 2010b, 2011];
- the hardware platform architecture [Bastoni et al. 2010b].

The problem should even worsen with the increasing number of processors available in a processing platform. The communication delays between the processors and the central memory will indeed increase as the number of processors trying to access the memory through the communication network becomes more numerous, thereby increasing the number of contentions on both the communication and memory level.

In conclusion, the scheduling algorithm should minimize the number of preemptions and migrations so as to improve the performances (i.e., increase the number of task sets that are actually schedulable with a given scheduling algorithm) and the predictability of real-time systems.

2.6 Outline and Goals of the Thesis

Throughout this chapter, we exposed the criticality of real-time operating systems — and scheduling algorithms in particular — for the correct operation of many embedded systems.

An optimal scheduling algorithm is an algorithm that can find a schedule respecting all the task deadlines whenever it is possible. This notion which will be formally defined in the next chapter, is therefore one of the best property that an algorithm could have in theory. However, as we pointed out in Section 2.5, if the number of preemptions and migrations is too high, then this property loses all its interest in practice since the preemption and migration overheads will slow down the tasks during their actual executions.

Also, in Section 2.3.3, we introduced the notion of discrete-time systems and we explained the importance of designing discrete-time scheduling algorithms.

Trying to solve some of these problems, this thesis exposes optimal scheduling solutions minimizing the number of preemptions and migrations for both continuous and

discrete-time systems. We also propose methods to extend these scheduling algorithms to more general task models than the "simple" sequential task approach. Hence, we present how existing real-time scheduling algorithms can be utilized to schedule parallel real-time tasks which become widely used to exploit the growing number of processors in computing platforms.

The document is organized as follows:

Chapter 3. In Chapter 3, the multiprocessor real-time scheduling theory is formally introduced. We present a brief history of the real-time scheduling algorithms together with the most important results of the multiprocessor real-time scheduling theory.

Chapter 4. We propose a new optimal scheduling algorithm for the scheduling of discrete-time systems. This algorithm is named BF^2 and is a variation of the BF algorithm [Zhu et al. 2003, 2011]. We prove that BF^2 is optimal for the scheduling of sporadic tasks with constrained deadlines in a discrete-time environment (assuming that the total density is not greater than the number of processors). We finally show that BF^2 outperforms the state-of-the-art optimal algorithm in terms of preemptions and migrations for the scheduling in a discrete-time environment. We indeed reduce by a factor around 2 the average number of preemptions and have up to three times less migrations in average.

Chapter 5. In this chapter, we investigate solutions improving a continuous-time scheduling algorithm which is optimal for the scheduling of periodic tasks. We propose two methods reducing the number of preemptions and migrations caused by this algorithm named EKG [Andersson and Tovar 2006].

Chapter 6. Chapter 6 presents a new scheduling algorithm named U-EDF. This algorithm is optimal for the scheduling of dynamic systems composed of sporadic tasks with constrained deadlines scheduled in a continuous-time environment (still assuming that the total density never exceeds the number of processors in the platform). Furthermore, it is shown that U-EDF causes very few preemptions and migrations in average in comparison to other existing optimal scheduling algorithms (around two preemptions and migrations per job in average when executed on 16 processors with a total utilization of 100%).

Chapter 7. This chapter extends the model of tasks schedulable with currently known optimal scheduling algorithms (including U-EDF). Indeed, it provides techniques transforming parallel tasks in a set of sporadic sequential tasks with constrained

deadlines. These sequential tasks can then be scheduled with optimal (or suboptimal) real-time scheduling algorithms such as U-EDF.

Chapter 8. This last chapter concludes the thesis, summarizes the results and provides future works that could be carried out to improve the results presented throughout this dissertation.

CHAPTER

3

Introduction to the Multiprocessor Real-Time Scheduling Theory

"Everything should be made as simple as possible,
but not simpler."

Albert Einstein

"Hey Momma, look at me.
I'm on my way to the promised land
I'm on the highway to hell."

AC/DC

Contents

Abstract

The number of processors in computing platforms grows every year. Major companies regularly increase the number of processors included in a single chip in order to improve the processing capacity of their products. For instance, Intel designed an 80 cores multiprocessor platform while Tilera currently sells products containing up to 100 processors in a single chip. To efficiently make use of these new hardware platform architectures, the initial uniprocessor scheduling solutions must either be extended to the scheduling on multiprocessor or be replaced by completely innovative scheduling algorithms.

In this chapter, we present some of the major evolutions in the real-time scheduling theory. We define the vocabulary that will be used throughout all this dissertation to characterize task sets and scheduling algorithms and we present some important results of the real-time scheduling theory that will be useful in the remainder of this thesis.

3.1 Introduction

These last years, we have witnessed a major paradigm shift in the computing platform design. Instead of increasing the running frequency of processors to improve the performances of processing platforms, the hardware designers now prefer to increase the number of processors available in a single chip. Parallelization of the task executions is therefore preferred to the execution speed improvement.
The rise of this new paradigm in computing platform architectures is essentially motivated by the increasing power consumption that accompanied the augmentation of the processor clock speed. Indeed, increasing the clock speed of a processor cause an augmentation of its *dynamic power consumption* which can be alleviated by a reduction of the voltage of the power supply of the processing platform. However, the processors are composed of *transistors* and to be able to reduce the voltage and still experience a gain in performances, hardware designers first need to reduce the size of these transistors. Unfortunately, as we reach transistor sizes under 100 nm, the *static power consumption* of transistors due to leakage currents becomes dominant in the total power consumption of the platform [Kao and Chandrakasan 2000; Roy et al. 2003; Yeo and Roy 2005]. Consequently, the processor clock speed cannot be increased anymore without causing extremely high power dissipations. Furthermore, although the commutation delays of transistors can indeed be reduced with their size — i.e., smaller transistors are faster —, we cannot continue to increase the communication speed through the wires interconnecting the transistors. However, because we are still able to reduce the size of the transistors but the size of a chip remains constant, there are more transistors available in one chip, and because it showed up that increasing the complexity of processors has rather small impacts on the overall system performances, we now prefer to multiply the number of processors in a same chip to take advantage of this large amount of transistors.

These succinctly exposed physical problems explain for instance the decision of Intel to cancel the development of the processor Tejas which was supposed to be the successor of the Pentium 4 [Lammers 2004]. Instead, Intel started to work on the development of Core Duo which integrates two processors in the same chip. Since this first step toward multiprocessor platforms, Intel continued its multiprocessor development course and recently designed an 80 processors chip for research purposes [Intel® 2012b].

Similar evolutions happened with the other processing platform vendors such as AMD, IBM, Sun Microsystems, Texas Instruments, *etc*. We can notably cite the Tilera's products proposing up to 100 processors in the same chip [Tilera 2011].

This modification of the processing platform architectures led to a regain of interest for the multiprocessor real-time scheduling theory during the last decade. Indeed, even though the real-time scheduling theory for uniprocessor platforms can be considered as being mature, the real-time multiprocessor scheduling theory is still an evolving research field with many problems remained open due to their intrinsic difficulties.

3.2 Vocabulary

3.2.1 Feasibility, Schedulability and Optimality

Throughout all this document, we will often use various terms such as *feasible*, *schedulable* and *optimal* to describe task sets and algorithms. In this section, we therefore provide formal definitions of these important concepts for the real-time scheduling theory.

Definition 3.1 (*Feasible*)

A task set τ is said to be feasible if there exists a schedule for all the possible sequences of jobs released by the tasks belonging to τ such that they all respect their deadlines.

Definition 3.2 (*Feasibility test*)

A feasibility test for a task set τ executed upon a platform Π, is a test on τ, determining if τ is feasible on Π.

The *feasibility* of a task set τ is therefore independent of the particular scheduling algorithm utilized to schedule the jobs released by the tasks in τ. On the other hand, the *schedulability* of a task set τ is defined relatively to a given scheduling algorithm S.

Definition 3.3 (*Schedulable*)

A task set τ is said to be schedulable by a given scheduling algorithm S if the schedule constructed by S respects all the deadlines of the jobs released by the tasks belonging to τ.

Definition 3.4 (*Schedulability test*)

A schedulability test on a task set τ scheduled with a real-time scheduling algorithm S upon a platform Π, is a test on τ, determining if τ is schedulable with S on Π.

These notions of schedulability and feasibility can be combined to define the *optimality* of a real-time scheduling algorithm S.

Definition 3.5 (*Optimal*)

A real-time scheduling algorithm S is optimal under a set of constraints if all the feasible task sets that respect these constraints are also schedulable by S.

An optimal scheduling algorithm is therefore an algorithm that respects all the job deadlines for any task set for which a scheduling solution exists. However, some constraints can be imposed to the system scheduled by the scheduling algorithm. Hence, the Rate Monotonic (RM) scheduling algorithm is optimal for the scheduling of tasks with implicit deadlines scheduled with fixed task priorities on uniprocessor platforms but is *not* optimal for the scheduling of tasks with constrained deadlines or if dynamic priorities are allowed [Liu and Layland 1973; Devillers and Goossens 2000].

3.2.2 Utilization and Resource Augmentation Bounds

To quantify how far a scheduling algorithm S is from the optimal solution, we sometimes use the notion of *utilization bound* or *resource augmentation bound*. Particularly, we will use the resource augmentation bound that is sometimes referred to as the *speed-up factor*.

We first define the utilization bound of a scheduling algorithm executed on a platform Π.

Definition 3.6 (*Utilization bound*)

The utilization bound UB_S of a real-time scheduling algorithm S provides the minimum value of the total utilization U of the task set τ scheduled upon a platform Π, beyond which we cannot guarantee that all the jobs released by tasks in τ will respect their deadlines.

Hence, the expression $U \leq \mathrm{UB}_S$ is a *sufficient* condition (which is nevertheless not always *necessary)* to respect all the job deadlines when τ is scheduled with S. We can therefore build a schedulability test based on the utilization bound for any task set τ scheduled with S on a platform Π. That is, if $U \leq \mathrm{UB}_S$ then τ is schedulable with S. Otherwise, we may not know.

The utilization bound is based on the properties of the scheduled task set (i.e., its total utilization). On the other hand, the resource augmentation bound is based on the properties of the platform [Phillips et al. 1997]. Indeed, the speed-up factor gives the value by which the processor clock speeds should be multiplied to be able to schedule all the feasible task sets with S. In other words, it says how much faster the processors

should run to be able to respect all the job deadlines when utilizing a scheduling algorithm S instead of an optimal scheduling algorithm.

Definition 3.7 (*Resource augmentation bound on processor speeds (speed-up factor)*)

A scheduling algorithm S has a resource augmentation bound v on the processor speeds if all the task sets that are feasible on processors with a speed 1 are also schedulable by S on the same processors running v times faster.

3.3 The Multiprocessor Real-Time Scheduling Theory

3.3.1 On the Origins of Real-Time Scheduling

Although we can trace the origins of real-time scheduling theory to the early works in the '50s and '60s in *job-shop scheduling* [Conway et al. 1967], it is usually assumed by the real-time research community that the real-time scheduling theory as we know it today is born with the seminal works published by Liu and Layland in the late '60s and early '70s [Liu 1969b; Liu and Layland 1973]. It is indeed in these works that were published for the first time, *optimal online* scheduling algorithms for recurrent tasks in the context of the fixed task priorities and fixed job priorities problems on uniprocessor platforms. Specifically, they proved that[1]:

- any feasible task set composed of periodic tasks with implicit deadlines is schedulable with EDF on a uniprocessor platform;
- any task set which is feasible with a fixed task priority scheduling algorithm on a uniprocessor platform is also schedulable with RM.

Also, they proved a utilization bound for RM which is given by the expression

$$\mathrm{UB}_{\mathrm{RM}} \stackrel{\mathrm{def}}{=} n \times (2^{\frac{1}{n}} - 1)$$

with n the number of tasks scheduled on the platform. Note that this bound tends to be equal to $\ln(2) \approx 0.69$ for a large number of tasks. Since RM is optimal for the scheduling of synchronous periodic tasks with implicit deadlines among the fixed task priorities

[1]The proof of the optimality of RM provided by Liu and Layland [1973] was actually incomplete. However, it has been revisited in [Devillers and Goossens 2000].

scheduling algorithms, it follows from this result that all fixed priorities scheduling algorithms have a utilization bound of $\ln(2)$ when scheduling synchronous periodic tasks with implicit deadlines.

Note that the work of Liu and Layland [1973] is based on the following theorem:

Theorem 3.1

A preemptive task set composed of periodic tasks with implicit deadlines is feasible on a uniprocessor platform if and only if its total utilization U is not greater than 1.

Hence, EDF correctly schedules (i.e., respects all the job deadlines of) any task set composed of periodic tasks with implicit deadlines as long as $U \leq 1$.

Many researches on different problems related to the real-time scheduling on uniprocessor platforms were conducted since these first results which are still the most used algorithms in today's real-time operating systems. We redirect the interested reader to [Audsley et al. 1995] and [Sha et al. 2004] for further documentation on the uniprocessor real-time scheduling theory history.

3.3.2 The Emergence of the Multiprocessor Real-Time Scheduling

The multiprocessor real-time scheduling started to either be studied in the late '60s and early '70s [Liu 1969a; Horn 1974]. Initially, there were two means that were considered to extend uniprocessor real-time scheduling algorithms to the scheduling on multiprocessor platforms:

Global Scheduling. With the global approach, all tasks can migrate between all processors. Hence, at any time t, the m jobs with the highest priorities are running on the m processors of the platform. For instance, with the global version of EDF naturally named global EDF (or sometimes G-EDF), the m active jobs with the earliest deadlines are executed on the m processors of the platform at any time t.

Partitioned Scheduling. Unlike the global scheduling policies, the tasks cannot migrate and each task is therefore executed on only one processor during the whole schedule. Hence, the task set is partitioned (i.e., the tasks are dispatched) amongst the processors of the platform. Then, a uniprocessor scheduling algorithm is used on each processor *independently* to schedule the tasks assigned to this processor. For instance, with partitioned EDF (P-EDF), the EDF algorithm is executed on each

processor independently. Hence, on each processor π_j, the active job with the earliest deadline amongst the jobs released by the tasks assigned to π_j, is scheduled to execute on this processor. The scheduling decision is therefore taken amongst the subset of τ assigned to π_j and not on all the tasks in τ as it is the case with global EDF.
Note that a different uniprocessor scheduling algorithm can be utilized on each processor of the platform once the tasks have been partitioned.

Unfortunately, the only variation of getting more than one processor in the platform, showed up to drastically increase the difficulty of the scheduling problem in comparison with the initial uniprocessor real-time scheduling theory. Hence, Liu stated in [Liu 1969a] that: *"Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors."*

This claim is perfectly illustrated by the result which is referred to as the "Dhall's effect" [Dhall and Liu 1978]. In their paper, Dhall and Liu tried to use RM and EDF as global scheduling algorithms and showed that there exist task sets with total utilizations arbitrarily close to 1 that are not schedulable by these algorithms even if there are more than one processor in the platform. However, on uniprocessor platforms, RM and EDF were optimal for the scheduling of periodic tasks with implicit deadlines under fixed task priorities and dynamic job priorities, respectively [Liu and Layland 1973]. This negative result for global scheduling policies strongly influenced the multiprocessor real-time research community which thereby favored partitioned scheduling algorithms to global scheduling policies during many years.

Unfortunately, the partitioning problem is NP-complete in the strong sense [Johnson 1973, 1974]. Furthermore, the partitioned scheduling algorithms are limited by the performances of the *bin packing* algorithms utilized to partition the tasks between the processors constituting the platform. Indeed, a bin packing algorithm cannot guarantee to successfully partition a task set with a total utilization greater than $\frac{m+1}{2}$ on a platform composed of m processors. Hence, in the worst-case, a partitioned scheduling algorithm can use only a little bit more than 50% of the processing capacity of the platform to actually execute the tasks. In other words, there may be 50% of the platform computing capacity left unused.

Example:

Consider $(m+1)$ periodic tasks with a utilization $0.5+\varepsilon$ (with $\varepsilon > 0$) that must be partitioned on a platform composed of m processors. According to Theorem 3.1, the sum of the task utilizations assigned to the same processor cannot exceed 1 if we want to be able to respect all the job deadlines. Hence, we can assign at most one task on each processor. Otherwise, the sum of the task utilizations assigned to this processor would be at least $(1+2\varepsilon)$ and deadlines would be missed. There is therefore one remaining task which cannot be assigned to any processor without missing a deadline.

However, the total utilization of this task set is

$$U = (m+1) \times (0.5+\varepsilon) = \frac{m+1}{2} + (m+1)\varepsilon$$

Therefore, for arbitrarily small values of ε, the task set is not schedulable with a partitioned scheduling algorithm with a utilization only slightly greater than $\frac{m+1}{2}$.

As mentioned in [Davis and Burns 2011], it is only in 1997 that Phillips et al. [1997] showed that the "Dhall's effect" is more of a problem with high utilization tasks than it is with global scheduling algorithms. This result renewed the interest in global scheduling algorithms. Hence, this property was exploited by Srinivasan and Baruah [2002] and Goossens et al. [2003] to provide the following schedulability test for global EDF:

Theorem 3.2

Any implicit deadline sporadic task system τ satisfying

$$\begin{cases} U \leq m - (m-1) \times \max\limits_{\tau_i \in \tau} \{U_i\} \\ \forall \tau_i \in \tau,\ U_i \leq 1 \end{cases}$$

is schedulable upon a platform that is comprised of m processors by the global EDF scheduling algorithm.

Interesting variations — named EDF-US$[\zeta]$ and EDF$^{(k)}$ — of the global EDF scheduling algorithm were then proposed in [Srinivasan and Baruah 2002] and [Goossens et al. 2003; Baker 2005] to overcome the restrictions of G-EDF caused by high utilization tasks. The scheduling algorithm EDF-US$[\zeta]$ always gives the highest priority to the jobs released by tasks with utilizations greater than a threshold ζ. The other tasks are then scheduled with global EDF. EDF$^{(k)}$ on its side, provides the highest priority to the $(k-1)$ tasks with the highest utilizations. Again, the other tasks are normally scheduled with G-EDF. Thanks to these techniques favoring the execution of tasks with high utilizations,

it was proven by Baker [2005] that EDF-US$[\zeta]$ and EDF$^{(k)}$ both have a utilization bound of $\frac{(m+1)}{2}$ when $\zeta = \frac{1}{2}$ and k is fixed to an optimal value denoted by $k_{\min}$. Hence, the following theorem holds:

Theorem 3.3

Any implicit deadline sporadic task system τ satisfying

$$\begin{cases} U \leq \dfrac{(m+1)}{2} \\ \forall \tau_i \in \tau,\ U_i \leq 1 \end{cases}$$

is schedulable upon a platform that is comprised of m processors by the global scheduling algorithms EDF-US$\left[\frac{1}{2}\right]$ or EDF$^{(k_{\min})}$.

Although they both have the same utilization bound, it was shown that EDF$^{(k_{\min})}$ dominates EDF-US$\left[\frac{1}{2}\right]$ in the sense that EDF$^{(k_{\min})}$ correctly schedules all the task sets that are schedulable with EDF-US$\left[\frac{1}{2}\right]$, but EDF-US$\left[\frac{1}{2}\right]$ cannot successfully schedule all the task sets that are schedulable with EDF$^{(k_{\min})}$.

Theorem 3.2 implies that the two aforementioned global variations of EDF have the same utilization bound than partitioned EDF. Therefore, the first designed global and partitioned extensions of EDF were not able to utilize more than 50% of the platform capacity in the worst-case scenarios. However, partitioned EDF and the various variations of global EDF are *incomparable* as there exist task sets that are schedulable with P-EDF but are not with G-EDF, EDF-US$[\zeta]$ or EDF$^{(k)}$. And inversely, there exist task sets that are schedulable with G-EDF, EDF-US$[\zeta]$ or EDF$^{(k)}$ but are not with P-EDF.

Other global scheduling algorithms such as FPZL, FPCL, FPSL, EDZL or EDCL are also worth to mentioned as they show interesting results in terms of schedulability while remaining quite simple to implement [Lee 1994; Chao et al. 2008; Cirinei and Baker 2007; Kato and Yamasaki 2008a; Davis and Kato 2012].

Note that most of these scheduling algorithms are subject to what is called *scheduling anomalies*. That is, task sets that are schedulable by a given algorithm S on a platform Π become unschedulable by S on Π when they are modified in a sense that may yet seems favorable. Many examples of such modifications exist. We can cite the reduction of the utilization of a task by increasing its period or reducing its worst-case execution time, finishing the execution of a job earlier than initially expected, adding processors to the platform, ...

For further information on global and partitioned scheduling algorithms and their schedulability tests, we redirect the interested reader toward the recent survey written by Davis and Burns in [Davis and Burns 2011].

3.3.3 Hybrid Solutions between Global and Partitioned Policies

Hybrid scheduling algorithms that propose a tradeoff between partitioned and global scheduling policies, have been designed over the years. Three such real-time scheduling algorithm families must be cited:

Semi-partitioned scheduling algorithms. With a semi-partitioned scheduling algorithm, most of the tasks are executed on only one processor as in a partitioned algorithm. However, a few tasks (or jobs) are allowed to migrate between two or more processors. The core idea of this technique is to improve the utilization bound of partitioned scheduling algorithms by globally scheduling the tasks that cannot be assigned to only one processor due to the limitations of the bin-packing heuristics. We can cite as some examples of this policy: EKG [Andersson and Tovar 2006; Andersson and Bletsas 2008], EDDP [Kato and Yamasaki 2008b], EDF-WM [Kato et al. 2009], EDF with C=D [Burns et al. 2010, 2011], ...

Restricted migrations scheduling algorithms. With these scheduling algorithms, all the tasks can migrate between all the processors but each job is assigned to only one processor. The partitionment is therefore applied at the job level instead of the task level. Interesting works on this topic can be found in [Baruah and Carpenter 2005; Funk and Baruah 2005; Funk 2004; Dorin et al. 2010].

Hierarchical scheduling algorithms. As already explained in the previous chapter, with hierarchical scheduling algorithms, the tasks are partitioned amongst servers (or supertasks depending on the particular algorithm). These servers (or supertasks) are then globally scheduled while the component tasks of each server are scheduled using a uniprocessor scheduling algorithm. VC-IDT [Shin et al. 2008; Easwaran et al. 2009], NPS-F [Bletsas and Andersson 2009, 2011] and RUN [Regnier et al. 2011; Regnier 2012] are three noticeable examples of such scheduling algorithms.

3.3.4 The Optimality in Real-Time Multiprocessor Scheduling

Hong and Leung [1988, 1992] proved that no optimal *online* scheduling algorithm can exist for the scheduling of a set of arbitrary real-time jobs with at least two different deadlines on a platform composed of more than one processor. Concurrently to Hong and Leung, Dertouzos and Mok [1989] investigated the minimum amount of parameters that should be known to be able to construct an online schedule respecting all the job deadlines. They concluded that no optimal online scheduling algorithm can exist if the arrival times of the jobs are not known *a priori* even if we know their worst-case execution times. More recently, it was proven in [Fisher et al. 2010] that the optimal online multiprocessor scheduling of sporadic real-time tasks with constrained or unconstrained deadlines is impossible. All these works therefore concluded that the scheduling algorithms would need clairvoyance to estimate the next job arrivals and take the optimal scheduling decision at any time t on a multiprocessor platform.

However, if all job properties are known beforehand, an optimal scheduling algorithm may exist. Hence, Horn proposed an optimal algorithm with a complexity in $O(N^3)$ (where N is the number of jobs) that determines a schedule for any set of *entirely defined* real-time jobs [Horn 1974].

Furthermore, even though no optimal online scheduling algorithm can be designed in the general case, there exist scheduling algorithms such as PD^2, LRE-TL or DP-Wrap, that are optimal for the scheduling of systems respecting the following properties [Srinivasan and Anderson 2002; Funk 2010; Funk et al. 2011]:

Theorem 3.4

Any preemptive periodic or sporadic task set τ that is comprised of implicit deadline tasks, is feasible on a platform composed of m identical processors if and only if $U \leq m$ and $U_i \leq 1$ for every task $\tau_i \in \tau$.

This result can either be extended to the scheduling of sporadic tasks with unconstrained deadlines as long as the following sufficient condition is respected [Funk 2010; Funk et al. 2011]:

Theorem 3.5

Any preemptive periodic or sporadic task set τ is feasible on a platform composed of m identical processors if $\delta \leq m$ and $\delta_i \leq 1$ for every task $\tau_i \in \tau$.

Actually, it is easy to prove that any optimal scheduling algorithm S for the scheduling

of sporadic tasks with implicit deadlines respecting the condition enunciated by Theorem 3.4, can also correctly schedule any task set following the conditions of Theorem 3.5 by converting each unconstrained deadline task in an equivalent implicit deadline task.

Theorem 3.6

Any task set τ composed of n sporadic tasks with unconstrained deadlines $\tau_i \stackrel{\text{def}}{=} \langle C_i, D_i, T_i \rangle$ such that $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$, is schedulable on a platform of m identical processors by any optimal multiprocessor scheduling algorithm for sporadic tasks with implicit deadlines considering the equivalent implicit deadline task set $\tau' \stackrel{\text{def}}{=} \left\{ \tau_i \stackrel{\text{def}}{=} \langle C'_i, T'_i, T'_i \rangle \mid i = 1, \ldots, n \right\}$ where $C'_i \stackrel{\text{def}}{=} C_i$ and $T'_i \stackrel{\text{def}}{=} \min\{D_i, T_i\}$.

Proof:

Let $U'_i \stackrel{\text{def}}{=} \frac{C'_i}{T'_i}$ be the utilization of a task $\tau'_i \in \tau'$ and let $U' \stackrel{\text{def}}{=} \sum_{\tau'_i \in \tau'} U'_i$ be the total utilization of τ'. Because it is assumed that for each task $\tau_i \in \tau$ there is a corresponding task $\tau'_i \in \tau'$ such that $C'_i = C_i$ and $T'_i = \min\{D_i, T_i\}$, it holds that

$$\forall \tau'_i \in \tau' \ : \ U'_i = \frac{C'_i}{T'_i} = \frac{C_i}{\min\{D_i, T_i\}} = \delta_i$$

and

$$U' = \sum_{\tau'_i \in \tau'} U'_i = \sum_{\tau_i \in \tau} \delta_i = \delta$$

Therefore, because by assumption we have $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$, it results that $U' \leq m$ and $U'_i \leq 1, \forall \tau'_i \in \tau$. Hence, if S is an optimal scheduling algorithm for sporadic tasks with implicit deadlines, then according to Theorem 3.4 the system τ' is schedulable by S. Furthermore, we have the three following properties:

- each task τ'_i has the same worst-case execution time than its corresponding task $\tau_i \in \tau$ (i.e., $C'_i \stackrel{\text{def}}{=} C_i$). Consequently, if S schedules the jobs of τ'_i for their worst-case execution time, then the jobs of τ_i have been executed for their worst-case execution time either;

- the deadline of each task τ'_i is always smaller than or equal to the deadline of τ_i (i.e., $T'_i \stackrel{\text{def}}{=} \min\{D_i, T_i\}$ implying that $T'_i \leq D_i$). Therefore, if S respects the deadlines of the jobs released by any task $\tau'_i \in \tau$ then it also respects the deadlines of the jobs of the corresponding task $\tau_i \in \tau$;

- each task τ'_i has a minimum inter-arrival time smaller than the corresponding task τ_i (i.e., $T'_i \stackrel{\text{def}}{=} \min\{D_i, T_i\}$ implying that $T'_i \leq T_i$). The task τ_i therefore never releases

jobs more often than what can be handled by the scheduling algorithm S;

For these three reasons, all the jobs released by tasks belonging to τ respect their deadlines. ■

Unfortunately, only a handful of *online* scheduling algorithms are actually optimal for the scheduling of *sporadic* tasks (assuming that $\delta \leq m$ and $\delta_i \leq 1$ for every task τ_i in τ). One may cite PD^2 [Srinivasan and Anderson 2002], VC-IDT [Shin et al. 2008; Easwaran et al. 2009], LRE-TL [Funk and Nadadur 2009; Funk 2010] and DP-Wrap [Levin et al. 2010; Funk et al. 2011]. However, there exist a few more algorithms that are optimal for the scheduling of *periodic* tasks with *implicit deadlines* (e.g., PF [Baruah et al. 1993, 1996], BF [Zhu et al. 2003, 2011], LLREF [Cho et al. 2006], EKG [Andersson and Tovar 2006], RUN [Regnier et al. 2011; Regnier 2012]).

Nevertheless, at the exception of RUN and, to some extent, of EKG, we should note that all these optimal scheduling algorithms suffer from a large number of preemptions and migrations.

In the next chapters, we will therefore propose optimal multiprocessor scheduling algorithms with few preemptions and migrations. Furthermore, the most advanced of these newly proposed scheduling algorithms (presented in Chapter 6), is optimal for the scheduling of *dynamic* systems composed of *sporadic* tasks with *constrained deadlines* in a *continuous-time* environment. This is a far more general model than what can handle the best performing algorithms in terms of preemptions and migrations that currently exist in the state-of-the-art. Indeed, RUN and EKG are "only" optimal for the scheduling of *static* systems composed of *periodic* tasks with *implicit deadlines* in a *continuous-time* environment. Furthermore, we believe that U-EDF can be extended to the scheduling in discrete-time environment while keeping its optimality.

CHAPTER

4

Discrete-Time Systems

An Optimal Boundary Fair Scheduling Algorithm for Sporadic Tasks

"Le temps est un grand maître, dit-on;
le malheur est qu'il soit un maître inhumain qui tue ses élèves"
("Time is a great teacher, but unfortunately it kills all its pupils")

Hector Berlioz

"Welcome to the jungle! We take it day by day.
If you want it you're gonna bleed, but it's the price you pay."

Guns N' Roses

Contents

Abstract

Nowadays, many real-time operating systems discretize the time relying on a system time unit (based on a system tick). To take this behavior into account, real-time scheduling algorithms must adopt a *discrete-time* model in which both timing requirements of tasks and their time allocations have to be integer multiples of the system time unit. That is, tasks cannot be executed for less than one system time unit, which implies that they always have to achieve a minimum amount of work before they can be preempted. In such systems, the scheduling overheads (i.e., number of scheduling points, preemptions and task migrations) can be reduced by intelligently dimensioning the system time unit. This in turn reduces the impact of such overheads on the application schedulability.

Assuming such a discrete-time model, the authors of [Zhu et al. 2003, 2011] proposed an efficient "boundary fair" algorithm (named BF) and proved its optimality for the scheduling of periodic tasks while achieving full system utilization. However, BF cannot handle *sporadic tasks* due to the inherent irregular and *a priori* unknown job release patterns. In this chapter, we propose an *optimal* boundary-fair scheduler for *sporadic* tasks (named BF^2), which follows the same principle as BF by making scheduling decisions only at the job arrival times and (expected) task deadlines. We show through simulations that BF^2 outperforms the state-of-the-art discrete-time based optimal scheduler (PD^2), benefiting from much less scheduling overheads.

Note: This work was carried out in collaboration with the research team of Prof. Dakai Zhu (University of Texas at San Antonio, Texas, USA) and with Dr Vincent Nélis (Cister Research Centre, Porto, Portugal).

4.1 Introduction

Many optimal scheduling algorithms for multiprocessor platforms have been designed over the years [Zhu et al. 2011; Baruah et al. 1996, 1995; Srinivasan and Anderson 2002; Andersson and Tovar 2006; Funk 2010; Regnier et al. 2011; Nelissen et al. 2012b]. Most of them base their scheduling decisions on a continuous-time model. That is, a task can be scheduled for any amount of time, thereby authorizing arbitrarily short task executions. However, this model is not in accordance with many today's real-time operating systems which take all their timing decisions relying on a system time unit [Wind River Systems, Inc. 2011; Krten and QNX Software Systems 2012]. For examples, delays imposed on tasks or timer initialization values must be defined as an integer multiple of the system time unit. Since you cannot delay a task or program the end of its execution before at least one time unit, it is quite unrealistic to schedule the execution of a task for less than one system time unit as it is nonetheless the case with continuous-time algorithms [Andersson and Tovar 2006; Funk 2010; Regnier et al. 2011; Nelissen et al. 2012b].

A solution to this problem consists in building the scheduling algorithm on a discrete-time model. In this case, both timing requirements of tasks and their time allocations have to be integer multiples of the system time unit. Note that, even though some real-time operating systems make use of high resolution timers which are not based on the system tick, the scheduling overheads can be reduced by intelligently dimensioning the system time unit. Indeed, the discrete-time approach ensures that a task always achieves a minimum amount of work before it can be preempted. This in turn reduces the impact of such overheads on the application schedulability.

The first optimal multiprocessor scheduling algorithm for periodic real-time tasks with *discrete* timing requirements was proposed in [Baruah et al. 1993, 1996]. This algorithm named PF is based on the notion of *proportionate fairness* (PFairness). The core idea of the Pfairness is to enforce proportional progress for all tasks by ensuring that the deviation from an ideal *fluid schedule* (see Definition 4.1 presented in Section 4.3 for a formal definition) *never* exceeds one system time unit. Several proportionate fair (PFair) algorithms have been proposed over the years [Baruah et al. 1993, 1996, 1995; Anderson and Srinivasan 2000a; Srinivasan and Anderson 2002]. However, by making scheduling decisions at every time unit, PFair schedulers can incur high scheduling overheads as they generally produce an excessive amount of task preemptions and migrations.

Observing the fact that a periodic real-time task with implicit deadline can only miss

its deadline at its period boundary (because the deadline of a task also corresponds to the end of its period when we consider periodic tasks with implicit deadlines), an optimal discrete-time based *boundary fair* scheduling algorithm (named BF) had previously been studied in [Zhu et al. 2003, 2011]. BF makes scheduling decisions *only* when a task reaches the end of its period (which corresponds also to its current deadline and the release of its next job). Specifically, at every such event henceforth called *boundary*, BF takes a scheduling decision for the whole time interval extending from the current boundary to the next one (the earliest next task deadline). Similar to PFair schedulers, BF ensures *fairness* for tasks at the period boundaries to avoid deadline misses. That is, at each period boundary, the deviation of any task from the theoretical fluid schedule is less than one time unit. It has been shown that BF can achieve full system utilization while guaranteeing all tasks to meet their deadlines [Zhu et al. 2003, 2011]. Moreover, compared to PFair schedulers, BF may substantially reduces the number of preemptions, migrations and scheduling points (up to more than 90%) [Zhu et al. 2003, 2011]. However, BF assumes that all the boundary instants (i.e., the deadlines and arrivals of jobs) are known *beforehand* and thus cannot handle sporadic tasks due to their irregular and *a priori* unknown arrival patterns.

More recent works aimed at reducing the number of task preemptions and migrations for periodic task systems [Andersson and Tovar 2006; Funk 2010; Funk et al. 2011; Regnier et al. 2011; Nelissen et al. 2012b]. Most of these algorithms are also optimal for *sporadic* tasks [Andersson and Bletsas 2008; Funk 2010; Funk et al. 2011; Nelissen et al. 2012b]. However, in spite of their ability to handle sporadic tasks, they adhere to a continuous-time model whose drawbacks have already been discussed earlier in the introduction. Hence, algorithms such as EKG [Andersson and Bletsas 2008], NPSF [Bletsas and Andersson 2009, 2011], LRE-TL [Funk 2010], DP-Wrap [Levin et al. 2010; Funk et al. 2011] or RUN [Regnier et al. 2011] are *not* directly comparable to the discrete-time solutions such as PF [Baruah et al. 1996], PD^2 [Srinivasan and Anderson 2002] or BF [Zhu et al. 2003, 2011]. Note that most of these continuous-time algorithms are presented in Chapter 5.

Contribution: In this chapter, we focus on *discrete-time* based systems and propose an optimal multiprocessor *boundary-fair* scheduling algorithm for *sporadic* tasks named BF^2. BF^2 extends the principles and ideas of BF. Specifically, BF^2 makes scheduling decisions only at the arrival time and (expected) deadlines of tasks. However, unlike periodic tasks for which all boundaries are known at system design-time and coincide

with the deadlines of periodic tasks, the irregular and *a priori* unknown job arrival patterns of sporadic tasks cause non-trivial challenges. The arrival time of a sporadic task may indeed not coincide with a task deadline and such timing disparities lead to unexpected complexity for the scheduler.

As the main contribution of this work, we present BF^2 and prove its optimality for the scheduling of sporadic tasks with implicit deadlines.

As exposed in Section 4.9, our simulation results show that BF^2 can substantially reduce the scheduling overheads such as the number of scheduling points, task preemptions and migrations, compared to PD^2 (i.e., the state-of-the-art scheduler for discrete-time systems).

Organization of this chapter: System models are presented in Section 4.2, while Section 4.3 reviews many fair schedulers for discrete-time systems. The basic steps of BF^2 are presented in Section 4.4 and Section 4.5 addresses the particularities inherent to the scheduling of sporadic tasks. The optimality of BF^2 is analyzed in Sections 4.6 and 4.7. Implementation considerations and improvement techniques are discussed in Section 4.8. Finally, Section 4.9 presents our simulation results and Section 4.10 concludes the chapter.

4.2 System Model

We address the problem of scheduling a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of n independent sporadic tasks with constrained deadlines on a platform composed of m identical processors. Each task $\tau_i \stackrel{\text{def}}{=} \langle C_i, D_i, T_i \rangle$ is characterized by a worst-case execution time C_i, a relative deadline D_i, and a *minimum* inter-arrival time T_i. Hence, a task τ_i releases a (potentially infinite) sequence of jobs. Each job $J_{i,q}$ of τ_i that arrives at time $a_{i,q}$ must execute for exactly C_i time units before its deadline occurring at time $a_{i,q} + D_i$ and the earliest possible arrival time of the next job of τ_i is at time $a_{i,q} + T_i$.

Since we are considering a discrete-time model, C_i, D_i and T_i are assumed to be natural multiples of the system time unit.

Recall that we say that a job is active at time t if the instant t lies between the arrival time of the job and its deadline. If a task τ_i has an active job at time t then we say that τ_i is active and we define $a_i(t)$ and $d_i(t)$ as the arrival time and absolute deadline of the

currently active job of τ_i at time t. Since we consider tasks with implicit or constrained deadlines, at most one job of each task can be active at any time t. Therefore, without causing any ambiguity, we use the terms "tasks" and "jobs" interchangeably in the remainder of this chapter.

4.3 Previous Works: Optimal Schedulers for Discrete-Time Systems

The notion of fairness has been introduced by Baruah *et al.* in [Baruah et al. 1996]. As its name implies, the main idea is to fairly distribute the computational capacity of the platform between tasks. At any time t, each task τ_i is therefore executed on the processing platform for a time proportional to its utilization. This led to the concept of *fluid schedule* defined as follows:

Definition 4.1 (*Fluid schedule*)

A schedule is said to be fluid if and only if at any time $t \geq 0$, the active job (if any) of every task τ_i arrived at time $a_i(t)$ has been executed for **exactly** *$U_i \times (t - a_i(t))$ time units.*

4.3.1 Proportionate and Early-Release Fairness

In discrete-time systems, tasks are always executed for an integer number of system time units. Consequently, task executions might deviate from the fluid schedule during the system lifespan. Indeed, consider a task τ_i with a utilization $U_i = 0.5$ and releasing a job a time 0. At time $t = 3$, τ_i should have been executed for $0.5 \times (3 - 0) = 1.5$ time units according to Definition 4.1. However, since τ_i can only be executed for integer multiples of the system time unit, it can only achieve 1 or 2 but certainly not 1.5 time units of execution. To measure this deviation from the fluid schedule, the *allocation error* (or *lag*) of a task is defined as follows [Baruah et al. 1993]:

Definition 4.2 (*Allocation Error (lag)*)

The lag of a task τ_i at time t is the difference between the amount of work $\text{exec}_i(a_i(t), t)$ executed by the active job of τ_i until time t in the actual schedule, and the amount of work that it would have executed in the fluid schedule by the same instant t. That is,

$$\text{lag}_i(t) \stackrel{\text{def}}{=} U_i \times (t - a_i(t)) - \text{exec}_i(a_i(t), t)$$

with $a_i(t)$ being the arrival time of the active job of τ_i.

Fair schedulers impose constraints on the lag of every task in order to bound the deviation from the fluid schedule. For instance, with a *Proportionate Fair* (PFair) scheduler the allocation errors of the tasks are always kept smaller than one system time unit [Baruah et al. 1993]. That is,

Definition 4.3 (*Proportionate fair schedule*)

A schedule is said to be proportionate fair (or PFair) if and only if

$$\forall \tau_i \in \tau,\ \forall t \geq 0\ :\ \mid \mathrm{lag}_i(t) \mid < 1$$

On the other hand, with an *Early-Release Fair* (ERFair) scheduler, tasks can be ahead by more than one time unit, but never be late by more than one time unit on the fluid schedule [Anderson and Srinivasan 2001]. Formally,

Definition 4.4 (*Early-Release fair schedule*)

A schedule is said to be Early-Release fair (or ERFair) if and only if

$$\forall \tau_i \in \tau,\ \forall t \geq 0\ :\ \mathrm{lag}_i(t) < 1$$

The intuition lying behind the PFair (or ERFair) approach is easily understandable; in discrete-time systems, each task must have been running for an integer number of time units before its deadline. Since the worst-case execution time of any task τ_i is assumed to be an integer either, if a task misses its deadline at time t, then it must have an integer number of remaining time units to execute. That is, there is at least a difference of one time unit between the fluid and the real schedule, i.e., there is $\mathrm{lag}_i(t) \geq 1$. As a result, enforcing $\mathrm{lag}_i(t) < 1$ at every time t and therefore, by extension, at every task deadline, ensures that no task will ever miss its deadline.

In the remainder of this work, we will use the terms *behind*, *punctual* and *ahead* to qualify the state of a task at time t. Formally,

Definition 4.5 (*Task behind at time t*)

A task τ_i is said to be behind at time t, if it has been executed for less time in the actual schedule than in the corresponding fluid schedule until time t. That is, $\mathrm{lag}_i(t) > 0$.

Definition 4.6 (*Task punctual at time t*)

A task τ_i is said to be punctual at time t, if it has been executed for exactly the same amount of time in the actual schedule than in the corresponding fluid schedule until

time t. That is, $\text{lag}_i(t) = 0$.

Definition 4.7 (*Task ahead at time t*)

A task τ_i is said to be ahead at time t, if it has been executed for more time in the actual schedule than in the corresponding fluid schedule until time t. That is, $\text{lag}_i(t) < 0$.

A task should therefore always be punctual at each of its deadlines if the schedule is correct.

The PF Algorithm

PF is the first optimal algorithm which was designed for the scheduling of periodic task sets with implicit deadlines. It was proposed by Baruah et al. in [Baruah et al. 1996].

Initially, PF took its scheduling decisions at any time t relying on *"characteristic strings"* expressing the future load request of each task in τ so as to respect the PFairness. This procedure has however been simplified, first in [Baruah et al. 1995] and then in [Anderson and Srinivasan 1999], introducing the notion of *pseudo-deadline*. We therefore present this refined version of PF in this document.

Under a PFair scheduling, each task τ_i is divided in an infinite sequence of time units named *subtasks*. Each subtask has an execution time of one time unit and the j^{th} subtask of a task τ_i is denoted $\tau_{i,j}$ with $j \geq 1$. Notice that a job $J_{i,q}$ is composed of C_i successive subtasks $\tau_{i,j}$.

To keep the lag of a task τ_i smaller than 1 and greater than -1, each subtask $\tau_{i,j}$ belonging to the job $J_{i,q}$ has to be executed in an associated *window*. This window extends from a *pseudo-release* $pr(\tau_{i,j})$ to a *pseudo-deadline* $pd(\tau_{i,j})$. In [Anderson and Srinivasan 1999], it was shown for *periodic* tasks released at time 0 that[1]

$$pr(\tau_{i,j}) \stackrel{\text{def}}{=} \left\lfloor \frac{j-1}{U_i} \right\rfloor$$

and

$$pd(\tau_{i,j}) \stackrel{\text{def}}{=} \left\lceil \frac{j}{U_i} \right\rceil$$

[1] In [Anderson and Srinivasan 1999, 2000a; Srinivasan and Anderson 2002] the pseudo-release and pseudo-deadlines were defined on a "slot" basis. In this document as in [Anderson et al. 2005; Srinivasan and Anderson 2005b], the pseudo-release and pseudo-deadline refer to time-instants. This explain why the formula given here for $pd(\tau_{i,j})$ is slightly different to the one presented in [Anderson and Srinivasan 1999, 2000a; Srinivasan and Anderson 2002] but identical to those of [Anderson et al. 2005; Srinivasan and Anderson 2005b].

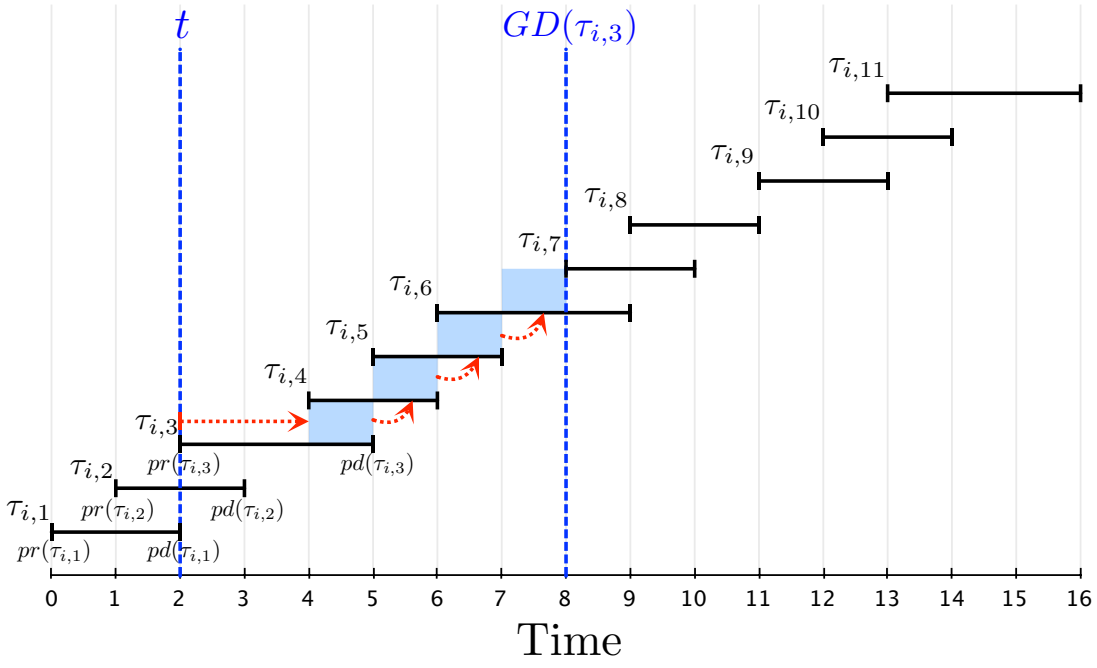


Figure 4.1: Windows of the 11 first subtasks of a periodic task τ_i with $U_i = \frac{8}{11}$ and illustration of the group deadline of the subtask $\tau_{i,3}$.

Figure 4.1 shows, as an example, the repartition of the windows for a periodic task τ_i with $U_i = \frac{8}{11}$ releasing its first job at time 0.

More generally, for either *periodic* or *sporadic* tasks, the pseudo deadline of a subtask $\tau_{i,j}$ can simply be defined in function of the pseudo release of $\tau_{i,j}$ [Srinivasan and Anderson 2002]. Hence,

$$pd\left(\tau_{i,j}\right) = pr\left(\tau_{i,j}\right) + \left\lceil \frac{j}{U_i} \right\rceil - \left\lfloor \frac{j-1}{U_i} \right\rfloor$$

and the pseudo-deadline of a subtask $\tau_{i,j}$ which is the p^{th} subtask to execute in a job $J_{i,q}$ is given by Equation 4.1.[2]

$$pd\left(\tau_{i,j}\right) \stackrel{\text{def}}{=} a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil \tag{4.1}$$

where $a_{i,q}$ is the arrival time of job $J_{i,q}$.

Note that because there are C_i consecutive subtasks in any job $J_{i,q}$, it holds that $q =$

[2]It was shown in [Anderson and Srinivasan 1999] that the window pattern is identical for each job of τ_i. Since the pseudo-deadlines of subtasks $\tau_{i,j}$ belonging to the first job of a task τ_i starting its execution at $t = 0$ is given by $pd\left(\tau_{i,j}\right) = \left\lceil \frac{j}{U_i} \right\rceil$ (see [Anderson et al. 2005]), the pseudo-deadlines of any other job $J_{i,q}$ is just translated by $a_{i,q}$ time units.

$\left\lceil \frac{j}{C_i} \right\rceil$ and $p = j - (q-1) \times C_i$.

At each time t, the PF algorithm determines which subtasks are *eligible* to be scheduled. A subtask $\tau_{i,j}$ of τ_i is said to be eligible at time t under PF if it respects the following definition:

Definition 4.8 (*Eligible Subtask under PF*)

A subtask $\tau_{i,j}$ of a task τ_i is eligible to be scheduled at time t if the subtask $\tau_{i,j-1}$ has already been executed prior to t and $pr\left(\tau_{i,j}\right) \leq t < pd\left(\tau_{i,j}\right)$, i.e., t lies within the execution window of $\tau_{i,j}$.

PF gives the highest priority to the active subtasks with the earliest pseudo-deadlines.

If there is a tie between two subtasks with the same pseudo-deadline, an additional parameter named *successor bit* is used. Informally, the successor bit $b\left(\tau_{i,j}\right)$ of a subtask $\tau_{i,j}$ is equal to 1 if and only if $\tau_{i,j}$'s window overlaps $\tau_{i,(j+1)}$'s window. $b\left(\tau_{i,j}\right)$ is equal to 0 otherwise. For instance, in Figure 4.1, $b(\tau_{i,2}) = 1$ while $b(\tau_{i,8}) = 0$. Using the definitions of the pseudo-deadline and pseudo-release, it was proven that

$$b\left(\tau_{i,j}\right) \stackrel{\text{def}}{=} \left\lceil \frac{j}{U_i} \right\rceil - \left\lfloor \frac{j}{U_i} \right\rfloor = \left\lceil \frac{p}{U_i} \right\rceil - \left\lfloor \frac{p}{U_i} \right\rfloor \tag{4.2}$$

Hence, PF orders the eligible subtasks at time t by their priorities using the following rules:

Prioritization Rules 4.1 (*Prioritization Rules of PF*)

With PF, a subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,\ell}$ (denoted $\tau_{i,j} \succ \tau_{k,\ell}$) iff:

(i) $pd\left(\tau_{i,j}\right) < pd\left(\tau_{k,\ell}\right)$ **or**

(ii) $pd\left(\tau_{i,j}\right) = pd\left(\tau_{k,\ell}\right) \ \wedge \ b\left(\tau_{i,j}\right) > b\left(\tau_{k,\ell}\right)$ **or**

(iii) $pd\left(\tau_{i,j}\right) = pd\left(\tau_{k,\ell}\right) \ \wedge \ b\left(\tau_{i,j}\right) = b\left(\tau_{k,\ell}\right) = 1 \ \wedge \ \tau_{i,j+1} \succ \tau_{k,\ell+1}$

Then, at each time t, the m eligible subtasks with the highest priorities according to Prioritization Rules 4.1 are chosen to be executed on the m processors of the platform. If $\tau_{i,j}$ and $\tau_{k,\ell}$ both have the same priority (i.e., we neither have $\tau_{i,j} \succ \tau_{k,\ell}$ nor $\tau_{k,\ell} \succ \tau_{i,j}$) then the tie can be broken arbitrarily by the scheduler.

Note that the recursion of the third rule of Prioritization Rules 4.1 always ends. Indeed, it was proven in [Anderson and Srinivasan 1999] that $b\left(\tau_{i,j}\right) = 0$ at least at the deadline

of a job.

The PD^2 algorithm

The PF algorithm performs very poorly due to the third recursive rule in Prioritization Rules 4.1. This problem has been addressed by Baruah et al. in [Baruah et al. 1995]. They proposed PD, a new PFair algorithm which replaces the third rule of PF by the calculation of three new tie breaking parameters. Hence, PD makes use of five different rules, each being computed in a constant time.

In [Anderson and Srinivasan 1999, 2000a], Anderson and Srinivasan proposed PD^2, an improvement of PD. They proved that the three additional rules of PD could be replaced by the computation of one quantity. Since PD^2 is a simplified version of PD, we only present PD^2 in this work.

Comparing with PF, PD^2 introduces a third parameter to compute the priority of a subtask $\tau_{i,j}$. This quantity is called the *group deadline* $GD\left(\tau_{i,j}\right)$ of a subtask $\tau_{i,j}$. On the one hand, for light tasks (i.e., tasks such that $U_i < 0.5$), the group deadline is always equal to 0. On the other hand, for heavy tasks (i.e., tasks with $U_i \in [0.5, 1]$), the group deadline depends on the future load request of the task. Indeed, imagine a sequence of subtasks $\tau_{i,j}$ to $\tau_{i,k}$ belonging to the task τ_i such that $\left(pd\left(\tau_{i,\ell+1}\right) - pd\left(\tau_{i,\ell}\right)\right) = 1$ for all $j \leq \ell < k$ (see subtasks $\tau_{i,3}$ to $\tau_{i,5}$ in Figure 4.1 for an example). If the subtask $\tau_{i,j}$ is scheduled in the last slot of its window, all the next subtasks of the sequence are forced to be scheduled in the last slot of their own windows. In Figure 4.1 for instance, if the subtask $\tau_{i,3}$ is scheduled at the fourth time unit then subtasks $\tau_{i,4}$ and $\tau_{i,5}$ have respectively to be scheduled at the fifth and sixth time unit. The group deadline is defined as the earliest time-instant so that such a sequence ends. Hence,

Definition 4.9 (*Group Deadline*)

The group deadline of any subtask $\tau_{i,j}$ belonging to a task τ_i such that $U_i < 0.5$ (i.e., a light task), is $GD\left(\tau_{i,j}\right) \stackrel{\text{def}}{=} 0$.
The group deadline $GD\left(\tau_{i,j}\right)$ of a subtask $\tau_{i,j}$ belonging to a heavy task τ_i (i.e., $U_i \geq 0.5$), is the earliest time t, where $t \geq pd\left(\tau_{i,j}\right)$, such that either $(t = pd\left(\tau_{i,k}\right) \wedge b\left(\tau_{i,k}\right) = 0)$ or $(t = pd\left(\tau_{i,k}\right) + 1 \wedge \left(pd\left(\tau_{i,k+1}\right) - pd\left(\tau_{i,k}\right)\right) \geq 2)$ for some subtask $\tau_{i,k}$ of τ_i such that $k \geq j$.

Informally, for heavy tasks, $GD\left(\tau_{i,j}\right)$ is the earliest time instant greater than or equal to $pd\left(\tau_{i,j}\right)$ that either finish a succession of pseudo-deadlines separated by only one time

unit or where the task τ_i becomes punctual.

In Figure 4.1, the group deadline of $\tau_{i,3}$ is thereby $GD(\tau_{i,3}) = 8$. The interested reader may consult [Anderson and Srinivasan 1999] for further information on the computation of the group deadlines in a constant time (see Equation (32) in [Anderson and Srinivasan 1999]).

With the definition of this new quantity, we can compare the priorities of two eligible subtasks at time t using the following set of three rules:

Prioritization Rules 4.2 (*Prioritization Rules of PD^2*)

A subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,\ell}$ under PD^2 (denoted $\tau_{i,j} \succ \tau_{k,\ell}$) iff:

(i) $pd(\tau_{i,j}) < pd(\tau_{k,\ell})$ ***or***

(ii) $pd(\tau_{i,j}) = pd(\tau_{k,\ell}) \ \wedge \ b(\tau_{i,j}) > b(\tau_{k,\ell})$ ***or***

(iii) $pd(\tau_{i,j}) = pd(\tau_{k,\ell}) \ \wedge \ b(\tau_{i,j}) = b(\tau_{k,\ell}) = 1 \ \wedge \ GD(\tau_{i,j}) > GD(\tau_{k,\ell})$

Again, if $\tau_{i,j}$ and $\tau_{k,\ell}$ both have the same priority (i.e., neither $\tau_{i,j} \succ \tau_{k,\ell}$ nor $\tau_{k,\ell} \succ \tau_{i,j}$ holds) then the tie can be broken arbitrarily by the scheduler.

PD^2 has first been proven to be optimal for the scheduling of periodic tasks with implicit deadlines following a PFair scheduling policy [Anderson and Srinivasan 1999]. That is, a subtask $\tau_{i,j}$ of a task τ_i is eligible to be scheduled only between its pseudo-release and its pseudo-deadline, thereby keeping the lag of τ_i within $(-1,1)$. However, PD^2 was further extended over the years; first, for the scheduling of tasks under an ER-Fair scheduling policy [Anderson and Srinivasan 2000a], and then for the scheduling of more complete and complex task models [Anderson and Srinivasan 2000b; Srinivasan and Anderson 2002, 2005b].

Early-Release Fairness The *Early Release Fair* (ERFair) approach consists in relaxing the PFairness property. In an ERFair algorithm, a subtask can be scheduled earlier than its pseudo-release. The lag of a task τ_i can therefore be smaller than -1 and the fairness constraint becomes:

$$\text{lag}_i(t) < 1, \forall t \tag{4.3}$$

rather than $|\ \text{lag}_i(t)\ | < 1, \forall t$.

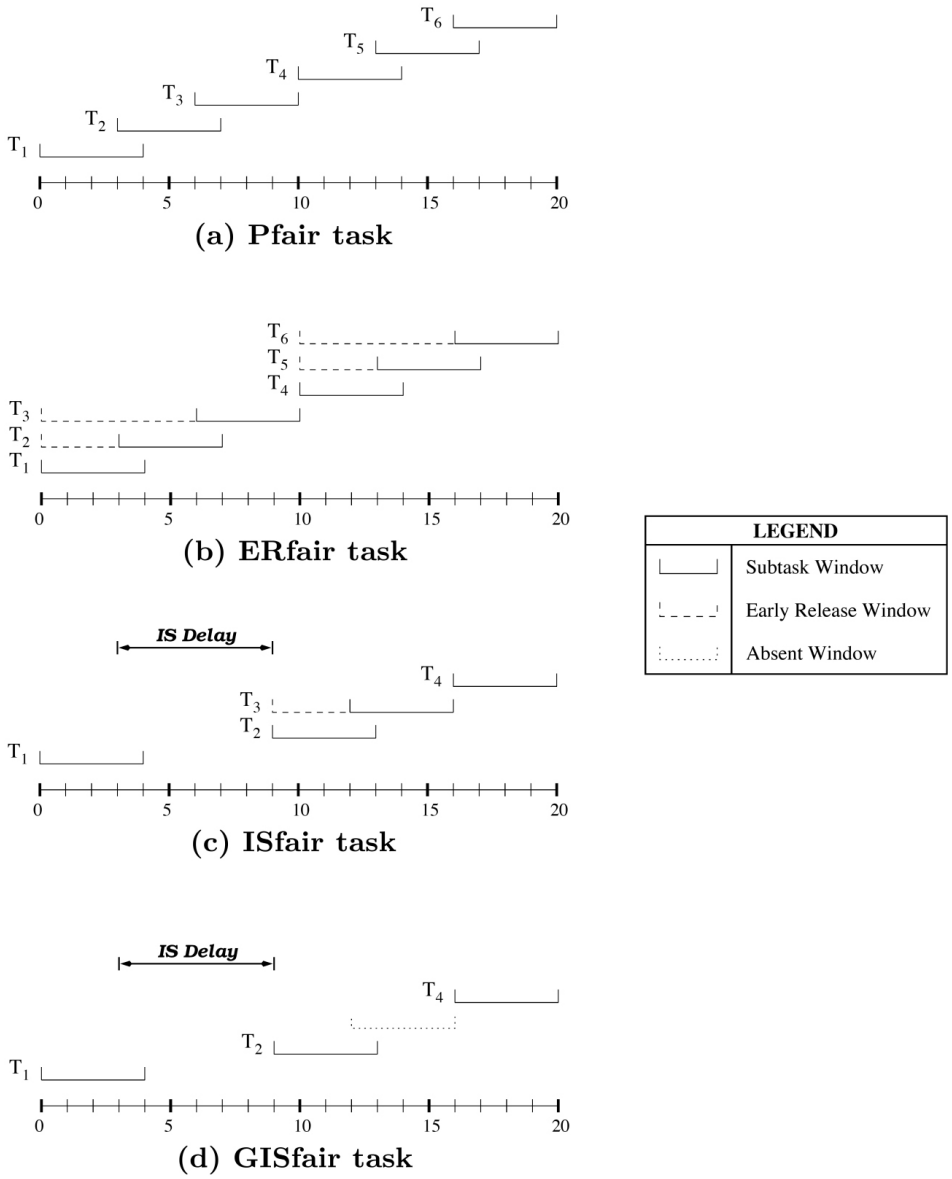


Figure 4.2: (From [Holman and Anderson 2003a]) The windows for a task τ_i with a utilization $U_i = \frac{3}{10}$ is shown under a variety of circumstances. (a) Normal windowing used by a Pfair task. (b) Early releases have been added to the windowing in (a) so that each grouping of three subtasks becomes eligible simultaneously. (c) Windows appear as in (b) except that $\tau_{i,2}$ release is now preceded by an intra-sporadic delay of six slots. ($\tau_{i,5}$ and $\tau_{i,6}$ are not shown.) (d) Windows appear as in (c) except that $\tau_{i,3}$ is now absent.

Hence, subtasks can take some advance compared to a strictly PFair schedule, thereby increasing the flexibility of the algorithm. Note that if an ERFair algorithm permits to subtasks to be released earlier than their pseudo-releases, the subtasks do not actually have to be eligible prior to their pseudo-releases. Hence, for each subtask $\tau_{i,j}$, there is a

third parameter $e\left(\tau_{i,j}\right)$ which corresponds to the first time-instant at which $\tau_{i,j}$ becomes eligible to be scheduled. It is assumed that this eligible time $e\left(\tau_{i,j}\right)$ is earlier than or concurrent to the the pseudo-release $pr\left(\tau_{i,j}\right)$, i.e.,

$$e\left(\tau_{i,j}\right) \leq pr\left(\tau_{i,j}\right)$$

Therefore, any PFair scheduling algorithm can be considered as a particular case of an ERFair algorithm but not *vice et versa*. Figures 4.2 (a) and (b) illustrate the difference between a PFair and an ERFair algorithm. In the ERFair algorithm of Figure 4.2 (b), each subtask becomes eligible to be scheduled as soon as its predecessor has been executed. In [Anderson and Srinivasan 2000a], the authors proved that PD^2 is optimal for an ERFair scheme. They also showed that an ERFair scheduler is less time consuming than a PFair one, once executed in a real operating system.

Sporadic task sets Until now, we were assuming that the task set was composed of strictly periodic tasks. In [Anderson and Srinivasan 2000b; Srinivasan and Anderson 2002], the *intra-sporadic task* model was introduced. In this kind of system, any subtask of any task τ_i could be released late. Furthermore, if a subtask $\tau_{i,j}$ is released with an offset of $\theta_{i,j}$ time units then all the next subtasks of τ_i have their pseudo-releases delayed by at least $\theta_{i,j}$ time units (see Figure 4.2 (c)). Hence, the pseudo-release and the pseudo-deadline of a subtask $\tau_{i,j}$ is redefined as

$$pr\left(\tau_{i,j}\right) \stackrel{\text{def}}{=} \theta_{i,j} + \left\lfloor \frac{j-1}{U_i} \right\rfloor$$

and

$$pd\left(\tau_{i,j}\right) \stackrel{\text{def}}{=} \theta_{i,j} + \left\lceil \frac{j}{U_i} \right\rceil$$

This model can be seen as a generalization of a sporadic task model where the C_i successive subtasks of a same job share the same offset $\theta_{i,j}$. In this case, Expression 4.1 still holds. It was proven in [Srinivasan and Anderson 2002] that PD^2 is optimal for the scheduling of intra-sporadic tasks under a PFair or ERFair scheduling policy.

That is, Srinivasan and Anderson [2002] proved that

Theorem 4.1

For any set τ of sporadic tasks with implicit deadlines executed on m identical processors, PD^2 respects all task deadlines provided that $\sum_{\tau_i \in \tau} U_i \leq m$, $\forall \tau_i \in \tau : U_i \leq 1$ and $\forall \tau_{i,j} \in \tau_i : e\left(\tau_{i,j}\right) \leq pr\left(\tau_{i,j}\right)$.

And using Theorem 3.6,

Theorem 4.2

For any set τ of sporadic tasks with unconstrained deadlines executed on m identical processors, PD^2 respects all task deadlines provided that $\sum_{\tau_i \in \tau} \delta_i \leq m$, $\forall \tau_i \in \tau : \delta_i \leq 1$ and $\forall \tau_{i,j} \in \tau_i : e\left(\tau_{i,j}\right) \leq pr\left(\tau_{i,j}\right)$.

The authors further generalized their model permitting to some subtasks to be absent [Srinivasan and Anderson 2005b]. This new kind of task is named *generalized intra-sporadic task* and is illustrated on Figure 4.2 (d). Even though this model of task is not of a real practical interest in its general version, it proves as a particular case that PD^2 is still optimal for the scheduling of tasks with jobs that do not necessarily execute for their worst-case execution times (i.e., when the last subtasks of some jobs are absent) [Srinivasan and Anderson 2005b].

Dynamic task sets In a *dynamic system*, tasks can dynamically leave and join the system. Some conditions have however to be respected to make sure that the system is still correctly schedulable after this load modification. It was proven in [Srinivasan and Anderson 2005b] that PD^2 correctly schedules any generalized intra-sporadic task set that satisfies the following conditions:

- A task can join the system at time t if and only if $\sum_{\tau_i \in \tau} \delta_i \leq m$ still holds after joining.
- A task τ_i can leave at time t if and only if $t \geq \max\left(GD\left(\tau_{i,j}\right), pd\left(\tau_{i,j}\right) + b\left(\tau_{i,j}\right)\right)$, where $\tau_{i,j}$ is the last scheduled subtask of τ_i.

The EPDF Algorithm

Every known PFair algorithm is based on a *Earliest Pseudo Deadline First* (EPDF) mechanism (i.e. rule (i) of Prioritization Rules 4.1 and 4.2). It was proven in [Anderson and Srinivasan 1999] that no tie-breaking parameters are needed to get an optimal scheduler

in case of a platform composed of two processors. The rule (i) of Prioritization Rules 4.2 is therefore sufficient to optimally schedule any set of tasks on a platform composed of one or two processors.

This last claim is however not true for a platform composed of more than two processors. Nevertheless, Srinivasan and Anderson proposed to use EPDF to schedule *soft real-time systems* on multiprocessor platforms [Srinivasan and Anderson 2005a; Devi and Anderson 2009]. In soft real-time applications, the tasks may miss their deadlines and finish their executions with some tardiness. A less costly algorithm like EPDF can therefore be used to schedule the task set at the price of occasional deadline misses. This may be acceptable if the *tardiness bound* (i.e. a bound on the termination delay in case of a deadline miss) stays reasonable. In [Srinivasan and Anderson 2005a] and [Devi and Anderson 2009], the authors calculated such a tardiness bound for EPDF and experimentally concluded that deadline misses rarely occurred and no tardiness greater than one time unit ever occurred with their randomly generated task sets.

4.3.2 Boundary Fairness

The authors of [Zhu et al. 2003, 2011] showed that all deadlines can be respected by ensuring the fairness property only at task deadlines, rather than making scheduling decisions at every system tick. They proposed an optimal scheduling algorithm called BF for the scheduling of *periodic* tasks with implicit deadlines. This algorithm divides the time in slices bounded by two successive task deadlines. Then, the BF scheduler is invoked at every such boundary to make scheduling decisions for the next time slice. This algorithm is said to be *boundary fair* (BFair).

Formally, let the boundary b_k denote the k^{th} time-instant in the schedule at which the scheduler is invoked. We say that b_k and b_{k+1} are the boundaries of the k^{th} time slice denoted by TS^k. If $B \stackrel{\text{def}}{=} \{b_0, b_1, b_2, ...\}$ (with $b_k < b_{k+1}$ and $b_0 = 0$) denotes the set of boundaries encountered in the schedule, then a boundary fair schedule is defined as follows:

Definition 4.10 (*Boundary fair schedule*)

A schedule is said to be boundary fair if and only if, at any boundary $b_k \in B$, it holds for every $\tau_i \in \tau$ that $\text{lag}_i(b_k) < 1$.

Note that PFair and ERFair algorithms are also boundary fair (i.e., they also respect the fairness at all boundaries as they are fair at every time unit). However, the counter

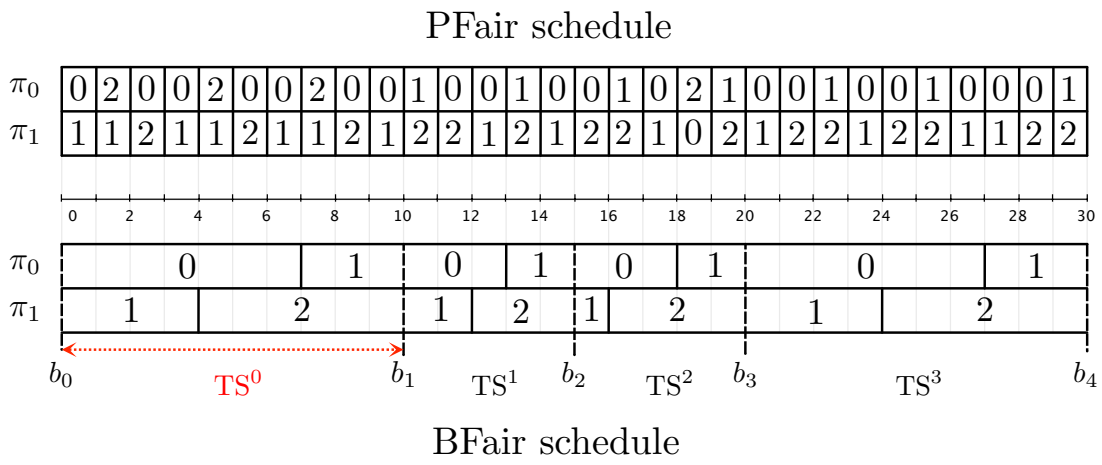


Figure 4.3: Proportionate Fair and Boundary Fair schedules of three tasks τ_0, τ_1 and τ_2 on two processors. The periods and worst case execution times are defined as follows: $T_0 = 15$, $T_1 = 10$, $T_2 = 30$, $C_0 = 10$, $C_1 = 7$ and $C_2 = 19$.

part is not necessarily true: a boundary fair algorithm may not ensure the fairness at every time unit.

The Figure 4.3 shows the correspondence between a PFair and a Boundary Fair schedule. By simply regrouping all the time units of a same task executed within a time slice TS^k, it is possible to drastically reduce the amount of preemptions and migrations of this task between the two boundaries. This property is illustrated in Figure 4.3 where, for instance, the task τ_0 is subject to 4 preemptions in the boundary fair schedule instead of 11 in the PFair schedule.

As previously mentioned, a boundary fair scheduler is invoked at every boundary b_k and makes scheduling decisions for the whole time slice extending from b_k to the next boundary b_{k+1}. Any Bfair scheduler invoked at boundary b_k can be decomposed in three consecutive steps:

1. Determine the next boundary b_{k+1}, compute and allocate the minimum amount of time units each task τ_i must *mandatorily* execute in order to satisfy the condition $\text{lag}_i(b_{k+1}) < 1$ at the next boundary b_{k+1};

2. If all the available time units within the interval $[b_k, b_{k+1})$ have not been allotted to tasks during step 1, distribute the remaining time units amongst the tasks as *optional* time units;

Algorithm 4.1: Boundary fair scheduler.

Input:
$t :=$ current time;
begin
  $b_{k+1} :=$ ComputeNextBoundary(τ);
  forall the $\tau_i \in \tau$ **do**
    /* Allocate mandatory units for τ_i */
    $\text{mand}_i(t, b_{k+1}) :=$ComputeMandatoryUnits($t$, b_{k+1});
  end
  $\text{RU}(t, b_{k+1}) := m \times (b_{k+1} - t) - \sum_{\tau_i \in \tau} \text{mand}_i(t, b_{k+1})$;
  AllocateOptionalUnits(RU(t, b_{k+1}), τ);
  GenerateSchedule(t, b_k, $\text{mand}_i(t, b_{k+1})$, $\text{opt}_i(t, b_{k+1})$);
end

3. Generate a schedule avoiding intra-job parallelism for the interval $[b_k, b_{k+1})$ according to the number of mandatory and optional time units allotted to each task.

Algorithm 4.1 exposes the skeleton of any boundary fair scheduler following these three steps.

The BF Algorithm

The steps 1 to 3 of boundary fair algorithms are now detailed for BF, the optimal BFair algorithm for the scheduling of *periodic* tasks with implicit deadlines, which was proposed in [Zhu et al. 2003, 2011].

For periodic tasks with implicit deadlines the next boundary b_{k+1} is always defined as the earliest task deadline after the current boundary b_k. That is,

$$b_{k+1} \stackrel{\text{def}}{=} \min_{\tau_i \in \tau} \{d_i(b_k)\}$$

Once b_{k+1} has been determined, the three steps previously cited consist in the following:

Step 1: Allocation of the Mandatory Time Units. The BF scheduler first computes the minimum number of time units that each task τ_i has to execute within the interval $[b_k, b_{k+1})$, in order to respect the fairness property at the next boundary b_{k+1} (i.e., $\text{lag}_i(b_{k+1}) < 1$). These time units are henceforth called *mandatory time units* and their

number is denoted by $\text{mand}_i(b_k, b_{k+1})$. It was shown in [Zhu et al. 2003, 2011], that $\text{mand}_i(b_k, b_{k+1})$ can be computed using the following equation:

$$\text{mand}_i(b_k, b_{k+1}) \stackrel{\text{def}}{=} \max\left\{0, \left\lfloor \text{lag}_i(b_k) + (b_{k+1} - b_k) \times U_i \right\rfloor\right\} \tag{4.4}$$

That is, within $[b_k, b_{k+1})$, each task τ_i must at least execute for the floor value of what it would have been executed in the fluid schedule, taking into account the allocation error $\text{lag}_i(b_k)$ of τ_i at boundary b_k. Here, the floor operator is used because in a discrete time system the execution time of a task must be an integer multiple of the system time unit.

Step 2: Allocation of the Optional Time Units. After receiving its mandatory time units, the lag of each task τ_i at the next boundary b_{k+1} is upper-bounded by 1. That is, for each individual task τ_i the fairness is ensured at time b_{k+1}. However, to avoid deadline misses in future time slices, the whole task set needs to make an appropriate progress as well.

Example:

Consider three periodic tasks $\tau_1 = \langle 1,2,2\rangle$, $\tau_2 = \langle 1,4,4\rangle$ and $\tau_3 = \langle 1,4,4\rangle$. The total utilization U is equal to 1 and the task set is therefore feasible on one processor. The two first time slices TS^0 and TS^1 extend from time 0 to 2 and from time 2 to 4, respectively. Using Expression 4.4 to calculate the number of mandatory time units that must be allocated to each task within TS^0, we get $\text{mand}_1(0,2) = 1$, $\text{mand}_2(0,2) = 0$ and $\text{mand}_3(0,2) = 0$. If we do not allocate a time unit either to τ_2 or τ_3, then all three tasks will need one mandatory time unit within TS^1. Since TS^1 extends only on two time units, this leads to a deadline miss.

Consequently, if the sum of all the mandatory time units allocated to all tasks differs from the total processing capacity of the platform in the time interval $[b_k, b_{k+1})$, then the spare processing time should be distributed to tasks.

The total number of available time units on m processors in the time interval $[b_k, b_{k+1})$ is given by $m \times (b_{k+1} - b_k)$. Hence, after all the tasks received their mandatory time units in Step 1, the number of *remaining time units* $\text{RU}(b_k, b_{k+1})$ within this interval is given by:

$$\text{RU}(b_k, b_{k+1}) = m \times (b_{k+1} - b_k) - \sum_{\tau_i \in \tau} \text{mand}_i(b_k, b_{k+1}) \tag{4.5}$$

Tasks compete for these $\mathrm{RU}(b_k, b_{k+1})$ time units, and each task τ_i possibly receives *one* of these remaining time units as an *optional time unit*. The number of optional time units allotted to τ_i is denoted by $\mathrm{opt}_i(b_k, b_{k+1})$. It is important to note that, even though these time units are optional for tasks (i.e., τ_i does not need to execute for this extra time to respect the fairness at boundary b_{k+1}), they actually have to be distributed amongst the tasks in order to guaranty an appropriate progress of the whole task set and avoid future deadline misses.

However, not all the tasks can receive an optional time unit. Specifically,

Definition 4.11 (*Eligible Task with BF*)

A task τ_i is said to be eligible *for an optional unit if*
(i) its allocation error (i.e., $\mathrm{lag}_i(t)$) is greater than 0 ***and***
(ii) $\mathrm{mand}_i(b_k, b_{k+1}) < \big(b_{k+1} - b_k\big)$, i.e., the number of time units already allocated to τ_i is strictly less than the length of the time slice. This second condition prevents a task from being executed concurrently on two (or more) processors.

Every eligible task competes for one optional time unit with an associated priority representing the future load requirement of the task. Similarly to PF in [Baruah et al. 1996], the BF algorithm computes the priority of a task τ_i in a time slice TS^k relying on a characteristic string $\alpha(\tau_i, k)$ and a corresponding urgency factor UF_i^k. The characteristic string is defined as a succession of characters over $\{-, 0, +\}$ (where $- < 0 < +$). Hence,

$$\alpha(\tau_i, k) \stackrel{\text{def}}{=} \{\alpha_{k+1}(\tau_i), \alpha_{k+2}(\tau_i), ..., \alpha_{k+s}(\tau_i)\}$$

where $\alpha_{k+\ell}(\tau_i) = \mathrm{sign}\Big(b_{k+\ell+1} \times U_i - \lfloor b_{k+\ell} \times U_i \rfloor - (b_{k+\ell+1} - b_{k+\ell})\Big)$ and s is the minimal natural greater than 0 such that $\alpha_{k+s}(\tau_i) \neq +$. Then, if the last character $\alpha_{k+s}(\tau_i) = -$, the urgency factor is defined as

$$\mathrm{UF}_i^k \stackrel{\text{def}}{=} \frac{1 - (b_{k+s} \times U_i - \lfloor b_{k+s} \times U_i \rfloor)}{U_i}$$

The characteristic string is based on the same idea followed in [Baruah et al. 1996]. Each character $\alpha_{k+\ell}(\tau_i)$ takes one of the values '$-$', '0' or '$+$' and gives an information on the future load request of τ_i in the time slice $\mathrm{TS}^{k+\ell}$:

- If $\alpha_{k+\ell}(\tau_i) = -$ then there will be $\mathrm{mand}_i(b_{k+\ell}, b_{k+\ell+1}) < (b_{k+\ell+1} - b_{k+\ell})$ assuming that τ_i does not receive an optional time unit in TS^k. That is, τ_i does not need to execute for the entire time slice length.

Algorithm 4.2: Compare(τ_i, τ_j) which compares the priorities of two tasks τ_i and τ_j in the time slice TS^k [Zhu et al. 2003, 2011]

```
   /* For task τ_i and τ_j, we assume that i < j                    */
 1 s := 1;
 2 while α_{k+s}(τ_i) = α_{k+s}(τ_j) = + do
 3 |   s := s + 1;
 4 end
 5 if α_{k+s}(τ_i) > α_{k+s}(τ_j) then return(τ_i ≻ τ_j);
 6 else if α_{k+s}(τ_i) < α_{k+s}(τ_j) then return(τ_i ≺ τ_j);
 7 else if α_{k+s}(τ_i) = α_{k+s}(τ_j) = 0 then return(τ_i ≻ τ_j);
 8 else if UF_i^{k+s} > UF_j^{k+s} then return(τ_i ≺ τ_j);
 9 else
10 |   return(τ_i ≻ τ_j);
11 end
```

- If $\alpha_{k+\ell}(\tau_i) = 0$ then there will be $\text{mand}_i(b_{k+\ell}, b_{k+\ell+1}) = (b_{k+\ell+1} - b_{k+\ell})$ and τ_i will be *punctual* at $b_{k+\ell+1}$, assuming that τ_i does not receive an optional time unit in TS^k. That is, τ_i does need to execute for the entire time slice length but will catch up its lateness on the fluid schedule at the end of the time slice $TS^{k+\ell}$.

- If $\alpha_{k+\ell}(\tau_i) = +$ then there will be $\text{mand}_i(b_{k+\ell}, b_{k+\ell+1}) = (b_{k+\ell+1} - b_{k+\ell})$ and τ_i will be *behind* at $b_{k+\ell+1}$, assuming that τ_i does not receive an optional time unit in TS^k. That is, τ_i does need to execute for the entire time slice length and will still be late compared to the fluid schedule at the end of the time slice $TS^{k+\ell}$.

Hence, a task τ_i with a character '$+$' constrains the schedule (i.e., it mandatorily has to be executed) during a larger time interval than a task with a character equal to '0' or '$-$'.

The urgency factor UF_i^k on the other hand, expresses in how much time the allocation error $\text{lag}_i(t)$ of the task τ_i will reach 1 assuming that we stop to execute τ_i from the time instant b_{k+s}. It is therefore more urgent to execute tasks with smallest urgency factors.

Hence, BF uses the function Compare(τ_i, τ_j) presented in Algorithm 4.2 to compare the priority of two tasks and select the $\text{RU}(b_k, b_{k+1})$ highest priority tasks.

The algorithm used by BF to distribute the optional time units at time-instant t ($b_k \leq t < b_{k+1}$) is shown in Algorithm 4.3. First, the algorithm identifies all the active tasks eligible for an optional time unit (line 1). Then, the $\text{RU}(t, b_{k+1})$ unallocated time units are distributed amongst the eligible tasks with the help of the function GetHighestPriorityTask($\mathcal{E}(t)$) (lines 6 to 12). Whenever a task τ_i is selected, it gets one

Algorithm 4.3: AllocateOptionalUnits(RU(t, b_{k+1}), τ) which dispatches the remaining time units amongst the eligible tasks. General algorithm for BFair schedulers. The function GetHighestPriorityTask($\mathcal{E}(t)$) must be particularized for each particular scheduler.

```
1  E(t) := set of eligible tasks;
2  forall the τ_i ∈ E(t) do
3  |   Compute τ_i's priority at the next boundary;
4  end

   // Select RU(t, b_{k+1}) optional units
5  RU(t, b_{k+1}) := m × (b_{k+1} − t) − Σ_{τ_i ∈ τ} mand_i(t, b_{k+1});
6  while RU(t, b_{k+1}) > 0 and E(t) ≠ ∅ do
7  |   τ_i :=GetHighestPriorityTask(E(t)) ;        // calls Compare(τ_i, τ_j)
   |   function
8  |   opt_i(t, b_{k+1}) := 1;
9  |   lag_i(b_{k+1}) := lag_i(b_{k+1}) − 1;
10 |   E(t) := E(t) \ τ_i;
11 |   RU(t, b_{k+1}) := RU(t, b_{k+1}) − 1;
12 end
```

optional unit (line 8) and its lag at boundary b_{k+1} is updated accordingly (line 9). The task τ_i is then removed from the eligible task set (line 10).

Step 3: Generation of the Schedule. Once we have determined the execution time of each task for the next time slice $[b_k, b_{k+1})$ (i.e., we have calculated $\text{mand}_i(b_k, b_{k+1})$ and $\text{opt}_i(b_k, b_{k+1}), \forall \tau_i \in \tau$), we still have to generate the schedule that will be executed in the time interval $[b_k, b_{k+1})$. It was shown in [Zhu et al. 2003, 2011] that McNaughton's wrap around algorithm proposed in [McNaughton 1959], can be used to schedule periodic tasks with implicit deadlines.

Figure 4.4 depicts how the wrap around algorithm generates a schedule of five tasks τ_1 to τ_5 on three processors. The length of each box corresponds to the execution time $(\text{mand}_i(t, b_{k+1}) + \text{opt}_i(t, b_{k+1}))$ that each task τ_i must execute within the current time slice TS^k. First, all the tasks are packed on the first processor π_1. Then, all boxes (or part of boxes) that overflow from TS^k are packed on the next processor and this process continues until all the tasks are allocated.

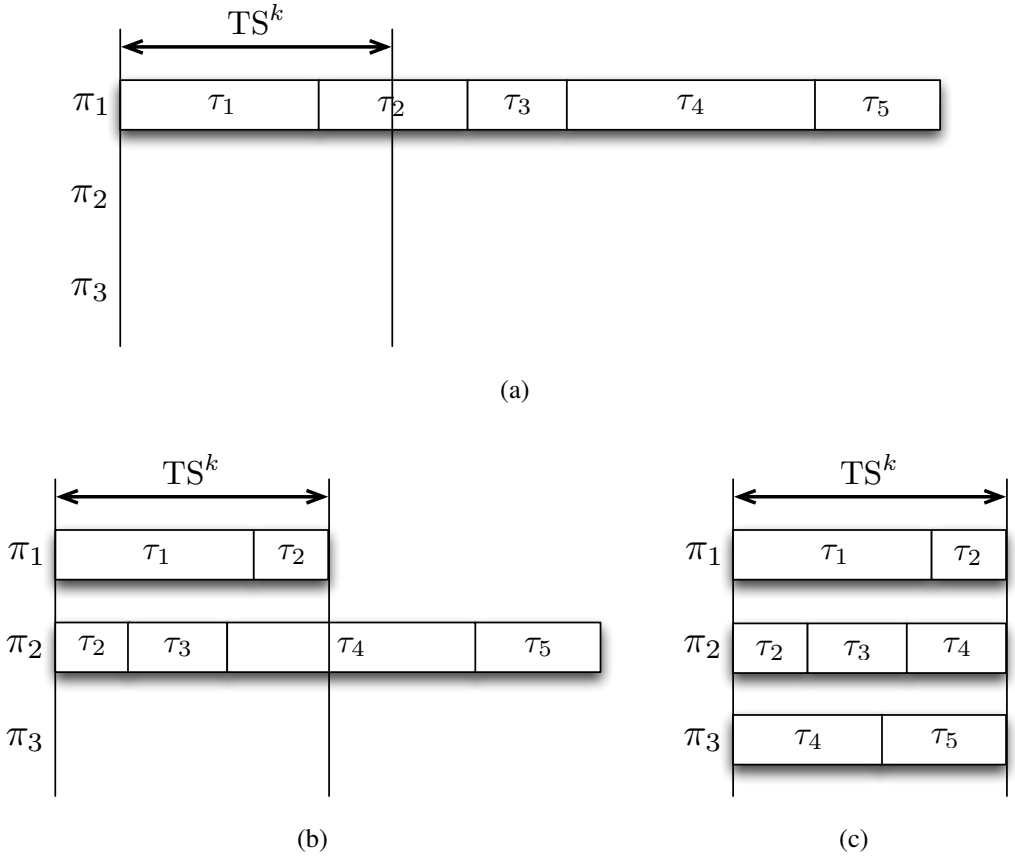


Figure 4.4: Illustration of the wrap around algorithm.

Improved version of BF

Similarly to the set of prioritization rules of PF (Prioritization Rules 4.1), Algorithm 4.2 is inefficient due to the potential need to compare all the characters of the characteristic strings of τ_i and τ_j (i.e., the *while* loop of Algorithm 4.2). Note that the characteristic string of a task τ_i may contain up to T_i characters. Moreover, Algorithm 4.2 must be called n times to find the highest priority tasks. Hence, the run-time complexity of BF is $O(n \times T_{\max})$ with $T_{\max} \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} \{T_i\}$. This complexity is way too high for an algorithm that must be executed online. The authors of [Zhu et al. 2003, 2011] therefore proposed an alternative solution to compare the priority of two tasks in a constant time.

This improved version of BF is based on the notion of *dual schedule* and makes use of the properties of a *dual* task (also called *counter-task*) τ_i' to derive properties on the *primal* task τ_i. The dual task τ_i' of the primal task τ_i is a task with an utilization $U_i' = 1 - U_i$. The idea behind the notion of dual schedule, is that a schedule can be obtained for τ_i by executing τ_i when the dual task τ_i' is not running.

Algorithm 4.4: ConstantTimeCompare(τ_i, τ_j) which compares the priorities of two tasks τ_i and τ_j in the time slice TS^k in a constant time [Zhu et al. 2003, 2011]

```
   /* For task τ_i and τ_j, we assume that i < j                    */
 1 if α_{k+1}(τ_i) > α_{k+1}(τ_j) then return(τ_i ≻ τ_j);
 2 else if α_{k+1}(τ_i) < α_{k+1}(τ_j) then return(τ_i ≺ τ_j);
 3 else if α_{k+1}(τ_i) = α_{k+1}(τ_j) = 0 then return(τ_i ≻ τ_j);
 4 else if α_{k+1}(τ_i) = α_{k+1}(τ_j) = − then
 5     if NUF_i^{k+1} > NUF_j^{k+1} then
 6         return(τ_i ≺ τ_j);
 7     else
 8         return(τ_i ≻ τ_j);
 9     end
10 end
11 else if α_{k+1}(τ_i) = α_{k+1}(τ_j) = + then
12     if NUF_i^{k+1} ≥ NUF_j^{k+1} then
13         return(τ_i ≻ τ_j);
14     else
15         return(τ_i ≺ τ_j);
16     end
17 end
```

It was proven in [Zhu et al. 2003, 2011] that $\alpha_{k+1}(\tau_i') = -$ when $\alpha_{k+1}(\tau_i) = +$. Since the urgency factor of the dual task of τ_i computed at b_{k+1} when $\alpha_{k+1}(\tau_i') = -$ expresses in how much time τ_i' will stop running ahead if it received an optional time unit in TS^k, it also indicates the minimum amount of time during which the primal task τ_i will run behind. Therefore, if the dual task τ_i' of τ_i has a urgency factor greater than the urgency factor of the dual task τ_j' of τ_j, then the task τ_i must have a higher priority than τ_j. Hence, the priority of a task τ_i is now computed with the two parameters $\alpha_{k+1}(\tau_i)$ and NUF_i^{k+1} where $\text{NUF}_i^{k+1} \stackrel{\text{def}}{=} \text{UF}_i^{k+1}$ if $\alpha_{k+1}(\tau_i) = -$; otherwise, if $\alpha_{k+1}(\tau_i) = +$, then $\text{NUF}_i^{k+1} \stackrel{\text{def}}{=} \frac{1-(b_{k+1}\times(1-U_i)-\lfloor b_{k+s}\times(1-U_i)\rfloor)}{1-U_i}$ which is the urgency factor of the dual task of τ_i. The new constant time compare function is presented in Algorithm 4.4.

Because $\alpha_{k+1}(\tau_i)$ and NUF_i^{k+1} can be computed in constant time, the BF algorithm has now a linear complexity of $O(n)$ using this new function to select the highest priority tasks.

The PL Algorithm

A new boundary fair scheduling algorithm for a set of synchronous periodic tasks was recently proposed in [Kim and Cho 2011]. This new algorithm named PL (which stands for *Pseudo Laxity*) makes use of the prioritization rules of PD^2 (Prioritization Rules 4.2) to distribute the optional time units. Furthermore, relying on the fact that the *least laxity first* (LLF) algorithm correctly schedules any feasible set of *jobs* with the same arrival time (see [Dertouzos and Mok 1989]), PL utilizes the LLF algorithm to generate the schedule within each time slice instead of McNaughton's wrap around algorithm.

PL was proven to be optimal for the scheduling of synchronous periodic tasks with implicit deadlines [Kim and Cho 2011].

4.4 BF^2: A New BFair Algorithm to Schedule Periodic Tasks

Unlike PL which defines a new way to schedule the tasks in each time slice, our newly proposed BFair scheduling algorithm named BF^2 is built on the same structure than BF (see Algorithm 4.1). It first determines the next boundary, then assigns mandatory and optional time units to tasks and finally generates a schedule for the time slice TS^k according to the number of time units allotted to each task.

For the scheduling of periodic tasks, the only difference between BF and BF^2 lies in the prioritization rules used to dispatch the RU(b_k, b_{k+1}) remaining time units as optional time units amongst the eligible tasks. The determination of the next boundary b_{k+1}, the computation of the mandatory time units allotted to each task and the generation of the schedule remain unchanged. On the other hand, the scheduling of sporadic tasks necessitates deeper modifications which will be treated in Section 4.5.

4.4.1 BF^2 Prioritizations Rules

To prioritize the eligible tasks during the optional time units dispatching, BF^2 uses two parameters reflecting the *future* execution requirement of each task. These parameters will further be shown to be a simple variation of the prioritization rules of PD^2.[3]

[3]Note that PL uses the exact PD^2's prioritization rules [Kim and Cho 2011] while we propose a simplified version of these rules making use of only to parameters instead of three.

According to Definition 4.2, if a running task τ_i interrupts its execution at time t then its lag starts increasing gradually by an amount proportional to its utilization U_i. Specifically, after x time units without being executed, its lag becomes

$$\text{lag}_i(t+x) = \text{lag}_i(t) + x \times U_i \tag{4.6}$$

To respect the fairness property, the allocation error of τ_i can never exceed 1. Hence, the first parameter used by BF^2 is the smallest number of time units x such that, if τ_i does not execute from the current time t to time $t+x$, then $\text{lag}_i(t+x)$ will exceed 1. This quantity can be computed by solving the following inequality

$$\text{lag}_i(t) + x \times U_i \geq 1 \tag{4.7}$$

We call this value of x the *urgency factor* of τ_i at time t (denoted by $\text{UF}_i(t)$)[4] which can be formally defined as follows (solving Expression 4.7)

Definition 4.12 (*Urgency Factor*)

The urgency factor $\text{UF}_i(t)$ *of a task* τ_i *at time* t *is the minimum number of time units such that, if* τ_i *is not executed from time* t *to time* $t + \text{UF}_i(t)$ *then its allocation error* $\text{lag}_i(t + \text{UF}_i(t))$ *at time* $t + \text{UF}_i(t)$ *is greater than or equal to* 1. *That is,*

$$\text{UF}_i(t) \stackrel{\text{def}}{=} \left\lceil \frac{1 - \text{lag}_i(t)}{U_i} \right\rceil$$

Informally, $\text{UF}_i(t)$ can be seen as the relative deadline from time t for executing τ_i for one time unit and still have a lag smaller than 1.

From now on, we say that task τ_i is more urgent than task τ_j at time t if $\text{UF}_i(t) < \text{UF}_j(t)$, i.e., τ_i's lag would reach 1 before τ_j's lag if both tasks τ_i and τ_j were not executed anymore from time t.

Now, suppose that at time t, we stop executing a task τ_i and we wait the "very last instant" before resuming its execution, i.e., just before its lag equalizes 1. That is, we do

[4]Note that the notation and the name of this parameter can lead to confusion with the urgency factor defined by the algorithm BF. There are two differences between the urgency factor defined in BF and BF^2. First, unlike the urgency factor UF_i^{k+1} defined in BF, the urgency factor of BF^2 (i.e., $\text{UF}_i(t)$) is an integer. Second, $\text{UF}_i(t)$ is defined according to the state of τ_i at time t, while UF_i^{k+1} is always defined for the boundary b_{k+1} irrespective of the actual state of τ_i. From this point onward, when we talk about the urgency factor of a task τ_i, we always refer to the urgency factor defined in BF^2 (i.e., $\text{UF}_i(t)$).

not execute τ_i from time t to time $t + \mathrm{UF}_i(t) - 1$. According to Definition 4.2, its lag at this instant $t + \mathrm{UF}_i(t) - 1$ is given by

$$\mathrm{lag}_i(t + \mathrm{UF}_i(t) - 1) = \mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1) \times U_i$$

Then, if we resume the execution of τ_i at time $t + \mathrm{UF}_i(t) - 1$ and we execute τ_i for y consecutive time units, its lag at time $t + \mathrm{UF}_i(t) - 1 + y$ becomes (using Definition 4.2)

$$\mathrm{lag}_i(t + \mathrm{UF}_i(t) - 1 + y) = \mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1 + y) \times U_i - y \tag{4.8}$$

The second parameter used by BF^2 can be expressed as the exact amount of time y for which τ_i needs to execute to catch up its deviation from the fluid schedule at time $t + \mathrm{UF}_i(t) - 1$, i.e., to get its lag equal to 0. According to Equation 4.8, this amount of time is the value of y for which

$$\mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1 + y) \times U_i - y = 0 \tag{4.9}$$

The value of y which satisfies the above equation is called the *recovery time* of τ_i at time t (denoted $\rho_i(t)$) and is formally defined as follows (using Expression 4.9):

Definition 4.13 (*Recovery Time*)

The recovery time $\rho_i(t)$ of the task τ_i at time-instant t is the minimum execution time needed by τ_i to become punctual, assuming that τ_i was not executed during $\mathrm{UF}_i(t) - 1$ time units from time t. That is,

$$\rho_i(t) \stackrel{\text{def}}{=} \frac{\mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1) \times U_i}{1 - U_i}$$

If two eligible tasks τ_i and τ_j have the same urgency factor, BF^2 favors the task with the largest recovery time. Indeed, $\rho_i(t) > \rho_j(t)$ means that τ_i will need more execution time than τ_j to catch up its lateness on the fluid schedule, thereby constraining the future scheduling decisions during a longer period of time.

With these two parameters, a priority order between eligible tasks can now be defined:

Prioritization Rules 4.3 (*Prioritization Rules of BF^2*)

We say that an eligible task τ_i has a higher priority than a task τ_j in time slice TS^k if

and only if

(i) $\mathrm{UF}_i(b_{k+1}) < \mathrm{UF}_j(b_{k+1})$ ***or***

(ii) $\mathrm{UF}_i(b_{k+1}) = \mathrm{UF}_j(b_{k+1}) \wedge \rho_i(b_{k+1}) > \rho_j(b_{k+1})$

If τ_i and τ_j have the same priority, then the scheduler can break the tie arbitrarily.

Note that, the priority of an eligible task does not depend on the current time t within the current time slice $[b_k, b_{k+1})$ at which the scheduler is invoked. Rather, it depends only on the next boundary b_{k+1}.

4.5 BF^2: A New BFair Algorithm to Schedule Sporadic Tasks

We start this section by illustrating the challenges in scheduling sporadic tasks through a motivational example.

4.5.1 Challenges in Scheduling Sporadic Tasks

Let us consider a task set composed of three sporadic tasks: $\tau_1 = \langle 14, 20, 20\rangle$, $\tau_2 = \langle 5, 10, 10\rangle$ and $\tau_3 = \langle 4, 5, 5\rangle$; scheduled on two identical processors ($U = \sum_{i=1}^{3} U_i = 2$). The first job of τ_1 arrives at time 0 while τ_2 and τ_3 both release their first job at time $t = 2$.

At time $t = 0$, there is only one active job (the first job of τ_1) which has its deadline at time 20. However, determining the appropriate *next boundary* comes to be our **first challenge**.

It has been shown in [Zhu et al. 2003, 2011] and illustrated on Figure 4.3 that considering a longer scheduling interval can help aggregate task execution time and then reduce the number of preemptions and task migrations in the resulting schedule. On the other hand, to simplify the scheduling algorithm, no task deadline should occur between two successive boundaries. As a trade-off, BF^2 computes the earliest *expected* deadline. In this example, if we assume that τ_2 and τ_3 release a job as soon as they can (i.e., at $t = 1$), then the deadlines of their jobs will occur at time $t = 11$ and $t = 6$, respectively. Hence, the earliest expected deadline is at time $b_1 = 6$.

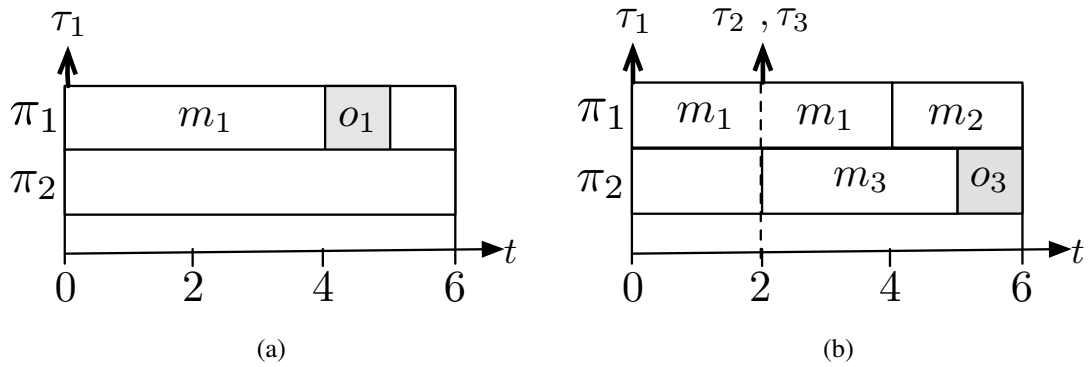


Figure 4.5: Example of a schedule for a first time slice extending from 0 to 6.

Note that, even if task τ_3 does not arrive as expected at time $t = 1$, it is still safe to have the next boundary at time 6 since there will never be another deadline before that instant. Now that we have our first boundary b_1, BF^2 can use the algorithm presented in the previous section to select the execution time allocated to the active tasks (see Figure 4.5(a) for the produced schedule). In Figure 4.5, mandatory units are represented by a clear rectangle labeled with the task number and optional time units are depicted by the shaded rectangles. The blank rectangles indicate the idle time of the corresponding processors.

As long as no new job is released, task τ_1 executes as shown in the schedule of Figure 4.5(a). However, if other jobs arrive during the interval $[0,6)$, the schedule needs to be adjusted accordingly to ensure fairness to the newly activated tasks at the end of the interval (i.e., at boundary $b_1 = 6$). Therefore, after tasks τ_2 and τ_3 released their first job at time $t = 2$, the allocation of mandatory and optional units have to be recomputed for all active tasks and the schedule has to be updated accordingly.

Here comes the **second challenge**: how to schedule the allocated mandatory and optional time units, knowing that new jobs can arrive at any time. Note that, if there were only periodic tasks, the allocation of optional time units would be *final* (since no job arrives before the next boundary) and the optional unit of a task can be scheduled right after its mandatory units following McNaughton's algorithm [Zhu et al. 2003, 2011]. However, for sporadic tasks, the arrivals of new jobs before the next boundary may require to *revoke* the optional time unit allocated to the tasks as the newly arrived jobs may have higher priorities. Such an adjustment is crucial to ensure that higher priority tasks always get the optional time unit that they need to meet their deadlines (see Section 4.7). Therefore, the optional time units of the tasks have to be scheduled *separately* from the mandatory time units. Optional time units are executed at the end of the time slice, which enables them to

be revoked if needed.

The adjusted schedule for the interval extending from $t = 2$ to $b_1 = 6$ is presented in Figure 4.5(b). As you can see, the optional time unit allocated to τ_1 has been revoked and been allocated to task τ_3 instead.

From the above example, we can see that, the arrival time of sporadic tasks can be different from boundaries. This is the major difference with periodic task sets. Therefore, the BF^2 scheduler needs to be invoked at two different occasions: (i) at the expected task deadlines; and (ii) at the arrival instants of sporadic tasks.

In the remainder of this section, we first discuss how to determine the time slice boundaries with sporadic tasks. Then, we present the detailed algorithm that must be used to schedule tasks in each time slice TS^k according to the execution time granted to each individual task. Finally, we explain how the schedule must be adjusted if a new job arrives during TS^k.

4.5.2 Determining the Next Boundary

Boundaries for periodic tasks can be readily determined as tasks arrive regularly [Zhu et al. 2003, 2011]. However, for sporadic tasks, jobs can be released at *any* instant provided that they are separated by (at least) their minimum inter-arrival times. Therefore, at a time-instant t, finding the *next boundary* b_{k+1} is not straightforward.

Based on the running state of sporadic tasks, we first define three disjoint task sets at time t:

- the *ready task set* $\Phi(t)$ contains active tasks (i.e., $t \in [a_i(t), d_i(t))$) whose execution has not been completed yet (i.e., $\text{ret}_i(t) > 0$);
- the *early-completion task set* $\Psi(t)$ contains tasks whose current jobs have deadlines later than t (i.e. $d_i(t) > t$) but have already finished their executions;
- the *delayed task set* $\Omega(t)$ contains tasks that do not have an active job at time t.

Note that, there is $\tau = \Phi(t) \cup \Psi(t) \cup \Omega(t)$ and the intersection between any two of these sets is empty.

From the previous work proposed in [Zhu et al. 2003, 2011], we know that longer scheduling intervals can help aggregate task allocations and thus reduce the number of

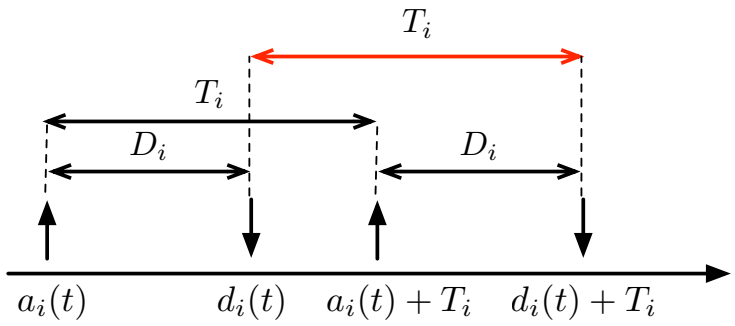


Figure 4.6: Arrival times and deadlines of two consecutive jobs of τ_i separated by exactly T_i time units.

resulting preemptions and task migrations. Therefore, it is preferred to have the next boundary to be as late as possible. However, to simplify the scheduling algorithm, no other task deadline should ever occur before the next boundary. Following these principles, we compute the *earliest expected deadline* $d_i^e(t)$ for any task τ_i at time t as follows:

- For any task τ_i belonging to the ready task set $\Phi(t)$, the deadline of its current active job is $d_i(t)$. Thus, $d_i^e(t) \stackrel{\text{def}}{=} d_i(t)$;
- For an early-completion task τ_i in $\Psi(t)$, its current job has finished its executions and therefore cannot miss its deadline. Hence, we must consider its next job that can arrive no earlier than $a_i(t)+T_i$. Thus, the deadline of this next job cannot occur before $d_i(t)+T_i$ (see Figure 4.6). Therefore, the earliest expected deadline of this new instance is $d_i^e(t) \stackrel{\text{def}}{=} d_i(t)+T_i$;
- For a delayed task τ_i in the delayed task set $\Omega(t)$, its next job can arrive as early as $t+1$, which gives $d_i^e(t) \stackrel{\text{def}}{=} t+1+D_i$;

Formally, at any time t, the next boundary b_{k+1} can be determined as follows:

$$b_{k+1} = \min\{d_i^e(t) \mid \tau_i \in \tau\} \tag{4.10}$$

where

$$d_i^e(t) \stackrel{\text{def}}{=} \begin{cases} d_i(t) & \text{if } \tau_i \in \Phi(t) \\ d_i(t)+T_i & \text{if } \tau_i \in \Psi(t) \\ (t+1)+D_i & \text{if } \tau_i \in \Omega(t) \end{cases} \tag{4.11}$$

Remember that $\tau = \Phi(t) \cup \Psi(t) \cup \Omega(t)$.

4.5.3 Generation of a Schedule

As explained in Section 4.3.2, there are two kinds of execution time units allocated to tasks: the mandatory and optional time units. Mandatory time units have to be executed before the next boundary b_{k+1}, and as their name implies, optional time units are optional (even though they have to be distributed to highest priority tasks to ensure appropriate progress of the whole task set). Therefore, if we need to allocate time for the execution of a new task arriving during the time interval $[b_k, b_{k+1})$, optional time units that were already distributed could be revoked and reallocated to other tasks, whereas mandatory time units cannot be unassigned. Hence, as another fundamental difference from the periodic case, the mandatory and optional time units of a same task cannot be scheduled consecutively in BF^2. Instead, mandatory time units must be executed as early as possible, while optional time units must be scheduled at the end of the time slice, i.e, we start executing what is mandatory before considering running the optional part. Furthermore, if a delayed task τ_i released within $[b_k, b_{k+1})$, has a higher priority than an other active task τ_j which already received an optional time unit, then τ_j's optional time unit must be reallocated to τ_i's execution. Consequently, optional time units must be scheduled in a decreasing priority order, thereby ensuring that a delayed task with higher priority (regardless of its arrival time) can always obtain an optional time unit before a low priority task does. Such a property is crucial to ensure the correctness of BF^2 as shown in Section 4.7.

Algorithm 4.5 summarizes the steps to schedule the allocated mandatory and optional time units when BF^2 is invoked at any time t such that $b_k \leq t < b_{k+1}$. Let Γ be the set of tasks that need to execute mandatory units within the interval $[t, b_{k+1})$ (i.e., $\text{mand}_i(t, b_{k+1}) > 0$). We first schedule all mandatory units of tasks in Γ as soon as possible in the interval $[t, b_{k+1})$. We therefore use the approach proposed by McNaughton minimizing the completion time of a set of jobs [McNaughton 1959]. Hence, we first compute the earliest completion time $\text{ct}(\Gamma, p)$ for the execution of all the mandatory time units in Γ on a number p of processors. This quantity is obtained by dividing the total load to execute, by the number of available processors. That is,

$$\text{ct}(\Gamma, p) \stackrel{\text{def}}{=} \frac{\sum_{\tau_i \in \Gamma} \text{mand}_i(t, b_{k+1})}{p}$$

If a task τ_i in Γ requires $\text{ct}(\Gamma, p)$ time units[5] or more for its mandatory part, we dedicate one processor for the execution of τ_i's mandatory units from time t to $t + \text{mand}_x(t, b_{k+1})$

[5]Remember that the time unit is simply a measure of the time and does not impose to $\text{ct}(\Gamma, p)$ to be an integer.

Algorithm 4.5: Generation of the schedule in a time slice extending from t to b_{k+1}.

```
1  Γ := tasks with allocated mandatory units;
2  p := number of processors;
3  ct(Γ, p) := earliest completion time of tasks in Γ on p processors;
   // Schedule the tasks that need a dedicated processor
4  while ∃τ_x ∈ Γ | mand_x(t, b_{k+1}) ≥ ct(Γ, p) do
5      Schedule mandatory units of τ_x on processor π_p from time t to
       t + mand_x(t, b_{k+1});
6      p := p − 1;
7      Γ := Γ \ {τ_x};
8      Recompute earliest completion time ct(Γ, p) of tasks in Γ;
9  end
10 if (Γ ≠ ∅) then
11     Schedule mandatory units of tasks in Γ according to McNaughton's wrap
       around algorithm in a decreasing priority order;
12 end
13 Schedule optional units in a decreasing priority order at the earliest available time
   slot without parallelism with their mandatory parts;
```

(line 5). The task τ_i is then removed from Γ and the earliest completion time of mandatory units that are not allocated yet, is updated taking into account that one processor is not available anymore (lines 6 to 8). Once all remaining tasks in Γ have $\text{mand}_i(t, b_{k+1}) < \text{ct}(\Gamma, p)$, McNaughton's wrap around algorithm is used to schedule the mandatory units of the remaining tasks in Γ within the time slice of length $\text{ct}(\Gamma, p)$. That is, mandatory time units are assigned in a non-increasing priority order, and whenever the number of time units assigned to a processor π_j would exceed $\text{ct}(\Gamma, m)$, then the task is split between π_j and π_{j+1} (see Figure 4.4). However, in a discrete time environment, every execution time must be an integer, which is probably not the case of $\text{ct}(\Gamma, p)$. We therefore have two possibilities: split the task when the load assigned to a processor exceeds either $\lfloor \text{ct}(\Gamma, p) \rfloor$ or $\lceil \text{ct}(\Gamma, p) \rceil$. Let first assume that we split when we reach $\lceil \text{ct}(\Gamma, p) \rceil$. In this case, there are *smt* less time units assigned to the last processor where *smt* is given by

$$smt \stackrel{\text{def}}{=} p \times \lceil \text{ct}(\Gamma, p) \rceil - \sum_{\tau_i \in \Gamma} \text{mand}_i(t, b_{k+1})$$

Hence, we instead assign $\lfloor \text{ct}(\Gamma, p) \rfloor$ time units to the *smt* first processors and $\lceil \text{ct}(\Gamma, p) \rceil$ to the $(p - smt)$ last processors (see the following example for a detailed illustration).

Finally, all optional units are scheduled in a decreasing priority order at the earliest available time slot without intra-job parallelism (line 13).

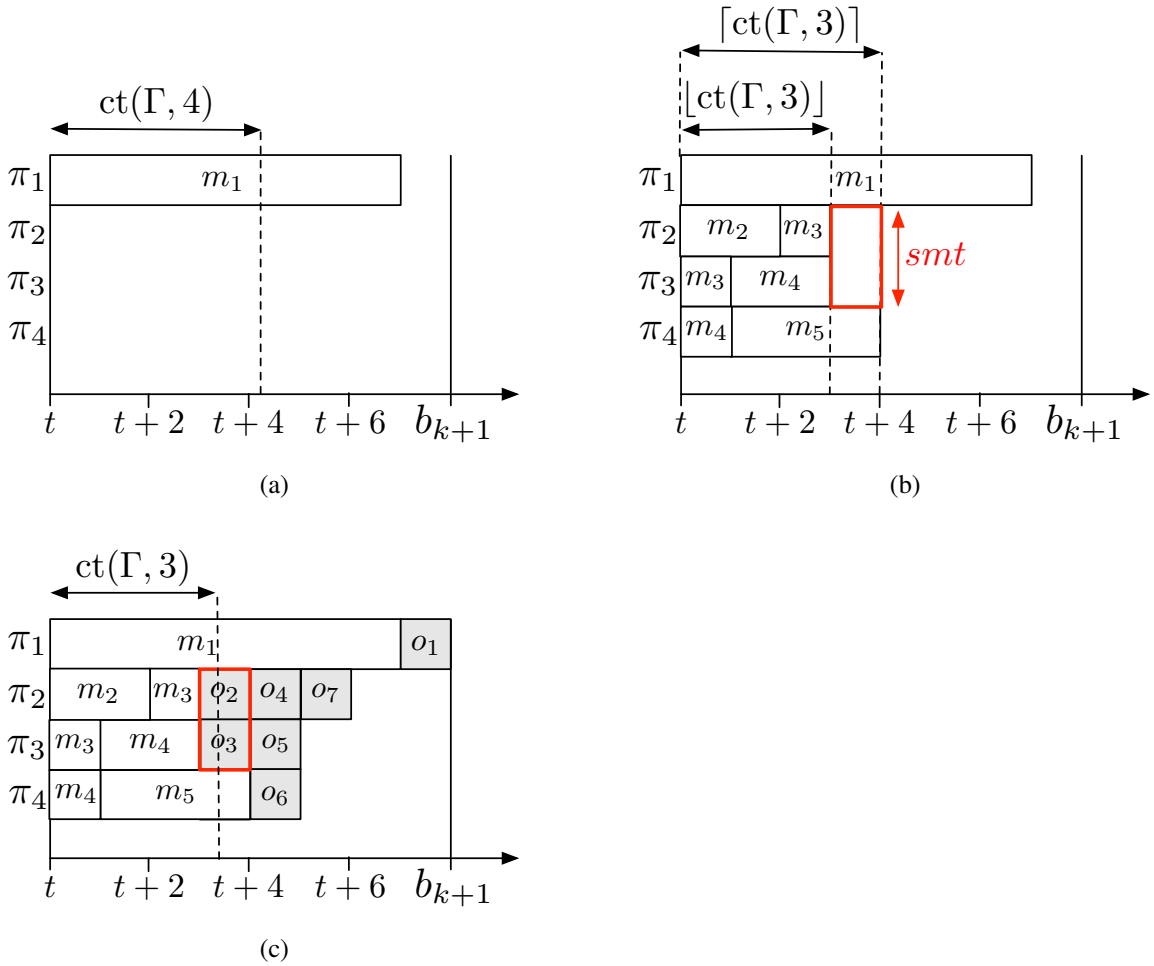


Figure 4.7: Example of a schedule slice generation in BF^2.

Example:

Suppose that, when BF^2 is invoked at time t, the next boundary is at time $b_{k+1} = t+7$. We assume seven active tasks τ_1 to τ_7 and four processors. The mandatory units allocated to the seven tasks are 7, 2, 2, 3, 3, 0 and 0, respectively. Moreover, every task receives one optional unit and we assume that optional units have a priority inversely proportional to their associated task index (i.e., optional unit o_1 of task τ_1 has the highest priority and o_7 has the lowest).

Hence, we initially have a total of 17 mandatory units to schedule. Since we have 4 processors in the platform, the earliest completion time of all mandatory units is $\mathrm{ct}(\Gamma,4) = \frac{17}{4} = 4.25$. *However, since τ_1 needs to execute for 7 mandatory units (which is greater than* $\mathrm{ct}(\Gamma,4)$*), we dedicate one processor for the schedule of τ_i's*

mandatory units (see Figure 4.7(a)). With one processor and 7 mandatory units less, we must recompute the earliest completion time of tasks in $\Gamma = \{\tau_2, \tau_3, \tau_4, \tau_5\}$ *which is now equal to* $\mathrm{ct}(\Gamma, 3) = \frac{3+3+2+2}{3} = 3.33$. *Since all remaining tasks in* Γ *have a number of mandatory units smaller than* 3.33, *we can use McNaughton's wrap around algorithm proposed in [McNaughton 1959]. Note that, for the remaining tasks in* Γ, *there are only a total of* 10 *mandatory units to schedule. Hence,* $smt = p \times \lceil \mathrm{ct}(\Gamma, p) \rceil - \sum_{\tau_i \in \Gamma} \mathrm{mand}_i(t, b_{k+1}) = 3 \times 4 - 10 = 2$. *Therefore, as shown on Figure 4.7(b), mandatory units of tasks* τ_2 *to* τ_5 *are scheduled using McNaughton's wrap around algorithm, reserving* $\lfloor \mathrm{ct}(\Gamma, 3) \rfloor = 3$ *time units on the two first processors (i.e.,* $smt = 2$*) and* $\lceil \mathrm{ct}(\Gamma, 3) \rceil = 4$ *time units on the last processor.*
Finally, the optional time units o_1 *to* o_7 *are scheduled at the earliest time without intra-job parallelism in a decreasing priority order (see Figure 4.7(c)).*

4.5.4 BF^2 at Arrival Times of Delayed Tasks

As explained earlier, when new jobs arrive at time t ($b_k < t < b_{k+1}$), the BF^2 scheduler needs to be invoked to adjust the schedule for the remaining interval $[t, b_{k+1})$. In particular, the optional units of existing tasks that have not been executed should be revoked. Then, together with the newly arrived tasks, all eligible tasks will re-compete for optional units based on their priorities.

Suppose that the previous invocation of the scheduler was at time t' ($b_k \leq t' < t < b_{k+1}$). For an active task τ_i at time t', its mandatory and optional units allocation for the interval $[t', b_{k+1})$ were given by $\mathrm{mand}_i(t', b_{k+1})$ and $\mathrm{opt}_i(t', b_{k+1})$, respectively. Moreover, the number of executed units for task τ_i during the interval $[t', t)$ is denoted $\mathrm{exec}_i(t', t)$. For every task τ_i, the number of mandatory units that τ_i still has to execute within the interval $[t, b_{k+1})$ is given by

$$\mathrm{mand}_i(t, b_{k+1}) = \max\left\{0, \mathrm{mand}_i(t', b_{k+1}) - \mathrm{exec}_i(t', t)\right\}$$

Moreover, if the task τ_i was granted an optional time unit at time t' (i.e., $\mathrm{opt}_i(t', b_{k+1}) = 1$), we must consider two different situations:

- The optional unit has already been executed within $[t', t)$ (that is, $\mathrm{exec}_i(t', t) = \mathrm{mand}_i(t', b_{k+1}) + \mathrm{opt}_i(t', b_{k+1})$). This leads to

$$\mathrm{mand}_i(t, b_{k+1}) = \mathrm{opt}_i(t, b_{k+1}) = 0$$

- The optional unit has not been executed yet (i.e., $\text{exec}_i(t',t) < \text{mand}_i(t',b_{k+1}) + \text{opt}_i(t',b_{k+1})$). Then we must revoke the optional unit allocated to τ_i (that is, $\text{opt}_i(t,b_{k+1}) = 0$). Since the number of time units allocated to τ_i is decreased by 1, the allocation error of τ_i at the next boundary b_{k+1} must be updated as follows

$$\text{lag}_i(b_{k+1}) \leftarrow \text{lag}_i(b_{k+1}) + 1$$

  Task τ_i will then compete against other eligible tasks to regain its optional time unit in the interval extending from t to b_{k+1}.

For any newly arrived task τ_x, its number of mandatory time units within $[t,b_{k+1})$ can be computed as $\text{mand}_x(t,b_{k+1}) = \lfloor (b_{k+1}-t) \times U_x \rfloor$ (from Equation 4.4). Moreover, there is $\text{lag}_x(b_{k+1}) = (b_{k+1}-t) \times U_x - \text{mand}_x(t,b_{k+1})$ (from Definition 4.2).

Finally, the distribution of optional units to eligible tasks and the generation of the schedule for $[t,b_{k+1})$ is carried out using Algorithms 4.3 and 4.5, respectively.

Due to this optional time unit redistribution at each new job arrival, and because Algorithm 4.5 schedules the execution of the allocated optional time units in a decreasing priority order, the following lemma holds:

Lemma 4.1

Let τ_i be any task that is (at least partially) active in time slice TS^k. That is, τ_i either released a job within the interval $[b_k,b_{k+1})$ or was already active at boundary b_k. Let $a_i(t)$ be the arrival time of τ_i and let $\text{act}_{i,k} \stackrel{\text{def}}{=} \max\{a_i(t),b_k\}$. At any time $t > b_k$ at which a task τ_x executes an optional time unit, for every task τ_i such that $\text{act}_{i,k} \leq t$, if τ_i has a higher priority than τ_x according to Prioritization Rules 4.3, then either τ_i also received an optional time unit no later than t or $\text{mand}_i(\text{act}_{i,k},t) \geq (t - \text{act}_{i,k})$.

4.6 BF^2: A Generalization of PD^2

As stated in Section 4.3.2, the PFair theory is a particular case of the BFair theory. In particular, BF^2 is the generalization of PD^2, the simplest known PFair algorithm. We will show that the two rules of BF^2 (Prioritization Rules 4.3) are equivalent to the three rules of a slight variation of PD^2 presented in Section 4.6.1. That is, for a given state of the system, the rules of BF^2 and the slight variation of PD^2 named PD^{2*} provide the same

task priority order (at the exception of the ties of PD^{2*} that could be broken differently in BF^2).

There are two major differences between BF^2 and PD^2:

- Unlike PD^2, BF^2 does not make any distinction between "light" and "heavy" tasks (i.e., tasks with $U_i < 0.5$ and tasks with $U_i \geq 0.5$, respectively). Indeed, the parameters $UF_i(t)$ and $\rho_i(t)$ used in the computation of the task priorities are defined for *all* tasks, irrespective to their utilization.
- Only two parameters are needed to prioritize tasks with BF^2, while there are three parameters in PD^2.

In spite of these differences, we show in this section that the definition of the group deadline proposed in PD^2 (Definition 4.9) can be slightly modified so that rule 4.3.(ii) of Prioritization Rules 4.3 of BF^2 can replace both rules 4.2.(ii) and 4.2.(iii) of Prioritization Rules 4.2 of PD^2, while rule 4.3.(i) of BF^2 provides identical results than rule 4.2.(i) of PD^2. Note that it can easily be shown that the number of basic operations (i.e., addition, subtraction, multiplication, division) needed for the computation of PD^2 and BF^2 parameters are almost identical. Hence, similar to PD^2 which simplified PD suppressing two tie breaking parameters, BF^2 can be seen as a simplification of PD^2 in the sense that it suppresses one tie breaking parameter.

4.6.1 PD^{2*}: A New Slight Variation of PD^2

One particularity of PD^2 is that the group deadline $GD(\tau_{i,j})$ is not similarly defined for light and heavy tasks (i.e., tasks with $U_i < 0.5$ and $U_i \geq 0.5$, respectively). Indeed, for light tasks $GD(\tau_{i,j})$ is always equal to 0 but for heavy tasks $GD(\tau_{i,j})$ is the earliest time instant after or at the pseudo-deadline $pd(\tau_{i,j})$ following a succession of pseudo-deadlines separated by only one time unit.

We propose a slight variation of PD^2 where the group deadline is defined identically for all tasks (whatever their utilization). Hence, the group deadline of a light task τ_i is not systematically equal to 0. This *generalized group deadline* denoted by $GD^*(\tau_{i,j})$ is defined as follows:

Definition 4.14 (*Generalized Group Deadline*)

The generalized group deadline $GD^*(\tau_{i,j})$ of any subtask $\tau_{i,j}$ belonging to a task τ_i,

is the earliest time t, where $t \geq pd\left(\tau_{i,j}\right)$, such that either $(t = pd\left(\tau_{i,k}\right) \wedge b\left(\tau_{i,k}\right) = 0)$ or $(t = pd\left(\tau_{i,k}\right) + 1 \ \wedge \ \left(pd\left(\tau_{i,k+1}\right) - pd\left(\tau_{i,k}\right)\right) \geq 2)$ for some subtask $\tau_{i,k}$ of τ_i such that $k \geq j$.

This slight variation of PD^2 is named PD^{2*} and the new set of rules ordering the subtasks becomes:

Prioritization Rules 4.4 (*Prioritization Rules of PD^{2*}*)

With PD^{2*}, a subtask $\tau_{i,j}$ has a higher priority than a subtask $\tau_{k,\ell}$ iff:

(i) $pd\left(\tau_{i,j}\right) < pd\left(\tau_{k,\ell}\right)$

(ii) $pd\left(\tau_{i,j}\right) = pd\left(\tau_{k,\ell}\right) \ \wedge \ b\left(\tau_{i,j}\right) > b\left(\tau_{k,\ell}\right)$

(iii) $pd\left(\tau_{i,j}\right) = pd\left(\tau_{k,\ell}\right) \ \wedge \ b\left(\tau_{i,j}\right) = b\left(\tau_{k,\ell}\right) = 1 \ \wedge \ GD^*\left(\tau_{i,j}\right) > GD^*\left(\tau_{k,\ell}\right)$

As for PD^2, if both $\tau_{i,j}$ and $\tau_{k,\ell}$ have the same priority, then the tie can be broken arbitrarily by the scheduler.

Note that, with this new definition of the group deadline, the proof of optimality of PD^2 given in [Srinivasan and Anderson 2002] has only one lemma which is impacted by the modification of the rule (iii). The updated proof of this lemma is provided in Appendix A, thereby proving that the proposed variation of PD^2 is still optimal for the scheduling of sporadic tasks (as well as for intra-sporadic and dynamic tasks) under a PFair or ERFair policy. Hence, we can write as a consequence of Theorem 4.2

Theorem 4.3

For any set τ of sporadic tasks with unconstrained deadlines executed on m identical processors, PD^{2*} respects all task deadlines provided that $\sum_{\tau_i \in \tau} \delta_i \leq m$ and $\forall \tau_i \in \tau$: $\delta_i \leq 1$ and $\forall \tau_{i,j} \in \tau_i : e\left(\tau_{i,j}\right) \leq pr\left(\tau_{i,j}\right)$.

4.6.2 Equivalence between $pd\left(\tau_{i,j}\right)$ and $\mathrm{UF}_i(t)$

Let $\tau_{i,j}$ denote the next subtask that must be executed by the task τ_i at time t. As proven below in Lemma 4.2, $\mathrm{UF}_i(t)$ is a measure of the *relative* pseudo-deadline of the subtask $\tau_{i,j}$ from time t, while $pd\left(\tau_{i,j}\right)$ denotes the *absolute* pseudo-deadline of $\tau_{i,j}$. That is,

$$\mathrm{UF}_i(t) = pd\left(\tau_{i,j}\right) - t \tag{4.12}$$

Lemma 4.2

Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. If $\tau_{i,j}$ is the subtask of τ_i ready at time t, then $\mathrm{UF}_i(t) = pd\left(\tau_{i,j}\right) - t$.

Proof:

Let $\tau_{i,j}$ be the p^{th} subtask of the current active job $J_{i,\ell}$ of τ_i released at time $a_{i,\ell}$. By definition of the pseudo-deadline (Equation 4.1), $pd\left(\tau_{i,j}\right) = a_{i,\ell} + \left\lceil \frac{p}{U_i} \right\rceil$.

Since t and $a_{i,\ell}$ are both integers (i.e. the time is discrete in a PFair schedule), it holds that:

$$\begin{aligned} pd\left(\tau_{i,j}\right) - t &= a_{i,\ell} + \left\lceil \frac{p}{U_i} \right\rceil - t \\ &= \left\lceil \frac{p}{U_i} - t + a_{i,\ell} \right\rceil \\ &= \left\lceil \frac{p - U_i \times (t - a_{i,\ell})}{U_i} \right\rceil \end{aligned} \tag{4.13}$$

Moreover, because $\tau_{i,j}$ is the p^{th} subtask which must be scheduled, it means that $p-1$ subtasks of $J_{i,\ell}$ have already been executed. By definition of the allocation error (Definition 4.2), there is

$$\mathrm{lag}_i(t) = U_i \times (t - a_{i,\ell}) - (p-1)$$

and rearranging the terms

$$p - U_i \times (t - a_{i,\ell}) = 1 - \mathrm{lag}_i(t) \tag{4.14}$$

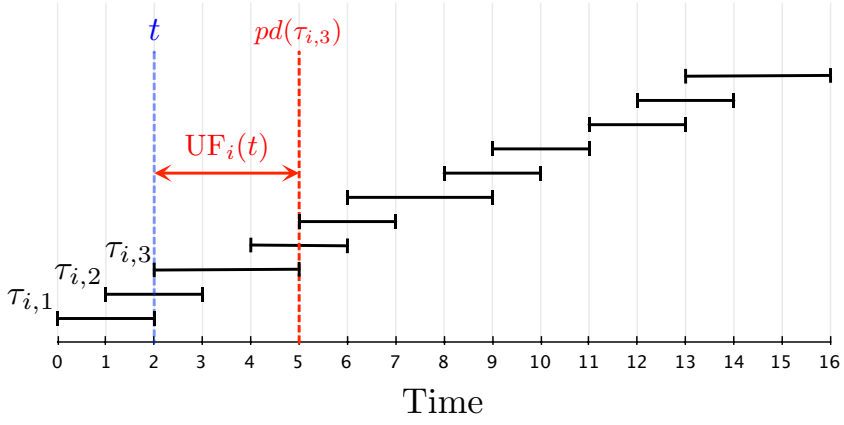
Then, using Equation 4.14 on the right-hand side of Equation 4.13, we get

$$pd\left(\tau_{i,j}\right) - t = \left\lceil \frac{1 - \mathrm{lag}_i(t)}{U_i} \right\rceil$$

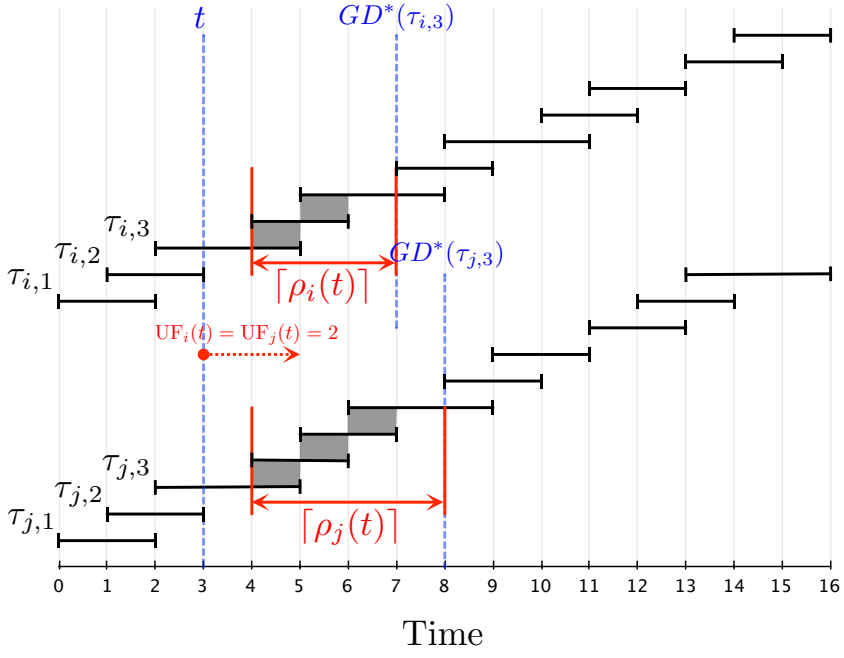
Since, by Definition 4.12, $\mathrm{UF}_i(t) = \left\lceil \frac{1-\mathrm{lag}_i(t)}{U_i} \right\rceil$, we finally obtain that

$$\mathrm{UF}_i(t) = pd\left(\tau_{i,j}\right) - t$$

which states the Lemma. ■



(a)



(b)

Figure 4.8: Comparison between: (a) the pseudo-deadline $pd(\tau_{i,3})$ of the third subtask of τ_i and its urgency factor $\mathrm{UF}_i(t)$ at time t. (b) the generalized group deadlines of $\tau_{i,3}$ and $\tau_{j,3}$, and their recovery times at time t.

Hence, Lemma 4.2 proves that Prioritization Rules 4.2.(i) and 4.3.(i) are equivalent. Indeed, if we have $pd(\tau_{i,j}) < pd(\tau_{p,\ell})$ (i.e., Rule 4.2.(i)), then $pd(\tau_{i,j}) - t < pd(\tau_{p,\ell}) - t$, thereby leading to $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$ (i.e., Rule 4.3.(i)).

The Figure 4.8(a) illustrates, as an example, the urgency factor of a task τ_i with $U_i = \frac{8}{11}$ at time $t = 2$ compared to the third subtask pseudo-deadline.

4.6.3 Equivalence between $b\left(\tau_{i,j}\right)$ and $\rho_i(t)$

We now prove through Lemma 4.3 (see below) that $\rho_i(t) = 1$ if $b\left(\tau_{i,j}\right) = 0$ and $\rho_i(t) > 1$ if $b\left(\tau_{i,j}\right) = 1$. Note that from Definition 4.13, the recovery time $\rho_i(t)$ is a real number while $b\left(\tau_{i,j}\right)$ can only be equal to 0 or 1.

Lemma 4.3

Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. Let $\tau_{i,j}$ be the subtask of τ_i ready at time t. If $b\left(\tau_{i,j}\right) = 0$ ($b\left(\tau_{i,j}\right) = 1$, respectively) then $\rho_i(t) = 1$ ($\rho_i(t) > 1$, respectively).

Proof:

Let $\tau_{i,j}$ be the p^{th} subtask of the current active job $J_{i,\ell}$ of τ_i released at time $a_{i,\ell}$. By definition of the successor bit (Equation 4.2), we have

$$b\left(\tau_{i,j}\right) = \left\lceil \frac{p}{U_i} \right\rceil - \left\lfloor \frac{p}{U_i} \right\rfloor \tag{4.15}$$

Now, suppose that the successor bit $b\left(\tau_{i,j}\right)$ equals 0. Then, according to Equation 4.15, it holds that $\frac{p}{U_i}$ is an integer. Hence, Equation 4.1 implies that $pd\left(\tau_{i,j}\right) = a_{i,\ell} + \frac{p}{U_i}$ and applying Lemma 4.2 we get that

$$\begin{aligned} \mathrm{UF}_i(t) &= pd\left(\tau_{i,j}\right) - t \\ &= a_{i,\ell} + \frac{p}{U_i} - t \end{aligned} \tag{4.16}$$

Similarly to the reasoning proposed in Lemma 4.13, because $\tau_{i,j}$ is the p^{th} subtask which must be scheduled, $p-1$ subtasks of $J_{i,\ell}$ have already been executed. By definition of the allocation error (Definition 4.2), there is

$$\mathrm{lag}_i(t) = U_i \times (t - a_{i,\ell}) - (p-1)$$

and rearranging the terms

$$p - U_i \times (t - a_{i,\ell}) = 1 - \mathrm{lag}_i(t) \tag{4.17}$$

Then, using Equation 4.17 with Equation 4.16, we get

$$\mathrm{UF}_i(t) = \frac{\mathrm{lag}_i(t) + 1}{U_i}$$

Replacing $\mathrm{UF}_i(t)$ in the expression of $\rho_i(t)$ given by Definition 4.13, we obtain

$$\begin{aligned} \rho_i(t) &= \frac{\mathrm{lag}_i(t) + (\mathrm{UF}_i(t) - 1)\, U_i}{1 - U_i} \\ &= \frac{\mathrm{lag}_i(t) + \left(\frac{1-\mathrm{lag}_i(t)}{U_i} - 1\right) U_i}{1 - U_i} \end{aligned} \tag{4.18}$$

and after simplification it holds that $\rho_i(t) = 1$ if $b\left(\tau_{i,j}\right) = 0$.

Similarly if $b\left(\tau_{i,j}\right) = 1$ then, by Equation 4.15, $\frac{p}{U_i}$ is not an integer and therefore $\frac{p}{U_i} - t + a_{i,\ell}$ neither. Then, using Equation 4.17 with this last expression, we get that $\frac{1-\mathrm{lag}_i(t)}{U_i}$ is not an integer either and because $\mathrm{UF}_i(t) = \left\lceil \frac{1-\mathrm{lag}_i(t)}{U_i} \right\rceil$ (Definition 4.12), it holds by the properties of the ceil operator that $\mathrm{UF}_i(t) > \frac{\mathrm{lag}_i(t)+1}{U_i}$. Hence, using Equation 4.18, we get

$$\rho_i(t) > \frac{\mathrm{lag}_i(t) + \left(\frac{1-\mathrm{lag}_i(t)}{U_i} - 1\right) U_i}{1 - U_i}$$

thereby leading to $\rho_i(t) > 1$ when simplifying. ■

4.6.4 Equivalence between $GD^*\left(\tau_{i,j}\right)$ and $\rho_i(t)$

As shown in Section 4.4, the floor value of the recovery time gives the number of successive subtasks that a task τ_i will mandatorily have to execute contiguously if the current subtask is executed in the last slot of its window, i.e., at time $t + \mathrm{UF}_i(t) - 1$ (see Figure 4.8(b) for an illustration). This claim is proven in Lemma 4.4. Then, in Lemma 4.5, we prove that $GD^*\left(\tau_{i,j}\right)$ and $\lceil \rho_i(t) \rceil$ are equivalent when we compare the priorities of two tasks τ_i and τ_p such that $\mathrm{UF}_i(t) = \mathrm{UF}_p(t)$. Specifically, we prove that

$$GD^*\left(\tau_{i,j}\right) = t + \mathrm{UF}_i(t) + \lceil \rho_i(t) \rceil - 1$$

Hence, if $\tau_{i,j}$ and $\tau_{p,q}$ are the two subtasks eligible at time t, then, assuming that $\mathrm{UF}_i(t) = \mathrm{UF}_p(t)$, it results that $\lceil \rho_i(t) \rceil > \lceil \rho_p(t) \rceil$ when $GD^*\left(\tau_{i,j}\right) > GD^*\left(\tau_{p,q}\right)$.

Lemma 4.4

Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. Then, τ_i

has exactly $\lfloor \rho_i(t) \rfloor$ successive pseudo-deadlines separated by one time unit following t. Formally, if $\ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$ and $\tau_{i,j}$ is the subtask of τ_i ready at time t, then

$$pd\left(\tau_{i,j+k+1}\right) - pd\left(\tau_{i,j+k}\right) = 1, \quad 0 \leq k < \ell - 1 \tag{4.19}$$

$$pd\left(\tau_{i,j+\ell}\right) - pd\left(\tau_{i,j+\ell-1}\right) \geq 2 \tag{4.20}$$

Proof:

Let $\tau_{i,j}$ be the subtask of τ_i ready at time t. Hence, the first pseudo-deadline of τ_i after t is $pd\left(\tau_{i,j}\right)$.

Let us assume that we are at time t_k ($> t$), that we have executed the subtasks $\tau_{i,j}$ to $\tau_{i,j+k-1}$ within $[t, t_k)$ and that the active subtask of τ_i at time t_k is $\tau_{i,j+k}$. That is, k time units have been executed between t and t_k and assuming that $\tau_{i,j}$ is a subtask of the job $J_{i,q}$ released at time $a_{i,q}$, it holds that

$$\text{exec}_i\left(a_{i,q}, t_k\right) - \text{exec}_i\left(a_{i,q}, t\right) = k \tag{4.21}$$

Using Definition 4.2, the difference between the lag of τ_i at time t_k and t is given by

$$\text{lag}_i(t_k) - \text{lag}_i(t) = U_i \times (t_k - t) - \left(\text{exec}_i(a_{i,q}, t_k) - \text{exec}_i(a_{i,q}, t)\right)$$

Hence, using Expression 4.21

$$\text{lag}_i\left(t_k\right) = \text{lag}_i(t) + U_i \times (t_k - t) - k \tag{4.22}$$

There are two cases that must be studied regarding the value of the recovery time $\rho_i(t)$ at time t.

(a) If $\rho_i(t) < k+1$ (i.e., $\lfloor \rho_i(t) \rfloor \leq k$), then by Definition 4.13,

$$\rho_i(t) = \frac{\text{lag}_i(t) + (\text{UF}_i(t) - 1) \times U_i}{1 - U_i} < k+1$$

leading to

$$\text{lag}_i(t) < k + 1 - (k+1) \times U_i - (\text{UF}_i(t) - 1) \times U_i$$

Using this last expression to replace $\text{lag}_i(t)$ in Expression 4.22, we get

$$\begin{aligned}\text{lag}_i(t_k) &= \text{lag}_i(t) + U_i \times (t_k - t) - k \\ &< k + 1 - (k+1) \times U_i - (\text{UF}_i(t) - 1) \times U_i + U_i \times (t_k - t) - k \\ &< 1 - U_i \times \big(\text{UF}_i(t) + k - (t_k - t)\big)\end{aligned}$$

Rearranging the terms, we obtain

$$\frac{1 - \text{lag}_i(t_k)}{U_i} > \text{UF}_i(t) + k - (t_k - t)$$

Because by definition of the urgency factor (Definition 4.12), $\text{UF}_i(t_k) \stackrel{\text{def}}{=} \left\lceil \frac{1-\text{lag}_i(t_k)}{U_i} \right\rceil$, this leads to

$$\text{UF}_i(t_k) > \text{UF}_i(t) + k - (t_k - t)$$

Finally, applying Lemma 4.2 to $\text{UF}_i(t_k)$ and $\text{UF}_i(t)$, we get

$$pd\left(\tau_{i,j+k}\right) - t_k > pd\left(\tau_{i,j}\right) - t + k - (t_k - t)$$

and simplifying

$$pd\left(\tau_{i,j+k}\right) > pd\left(\tau_{i,j}\right) + k \tag{4.23}$$

That is, there are more than k time units between the first and the $(k+1)^{\text{th}}$ pseudo-deadline.

(b) If $\rho_i(t) \geq k+1$ (i.e., $\lfloor \rho_i(t) \rfloor > k$) then by Definition 4.13,

$$\rho_i(t) = \frac{\text{lag}_i(t) + (\text{UF}_i(t) - 1) \times U_i}{1 - U_i} \geq k + 1$$

leading to

$$\text{lag}_i(t) \geq (k+1) - (k+1) \times U_i - (\text{UF}_i(t) - 1) \times U_i$$

Using this last expression to replace $\text{lag}_i(t)$ in Expression 4.22, we get

$$\begin{aligned}\text{lag}_i(t_k) &= \text{lag}_i(t) + U_i \times (t_k - t) - k \\ &\geq k + 1 - (k+1) \times U_i - (\text{UF}_i(t) - 1) \times U_i + U_i \times (t_k - t) - k \\ &\geq 1 - U_i \times \big(\text{UF}_i(t) + k - (t_k - t)\big)\end{aligned}$$

Consequently,

$$\frac{1-\text{lag}_i(t_k)}{U_i} \leq \text{UF}_i(t)+k-(t_k-t)$$

Since $\text{UF}_i(t_k) = \left\lceil \frac{1-\text{lag}_i(t_k)}{U_i} \right\rceil$ (Definition 4.12) and $\text{UF}_i(t)$, k, t_k and t are natural numbers (remember that the time is discrete), by the ceil operator property, it holds that

$$\text{UF}_i(t_k) \leq \text{UF}_i(t)+k-(t_k-t)$$

Therefore, applying Lemma 4.2 to $\text{UF}_i(t_k)$ and $\text{UF}_i(t)$, we get

$$pd\left(\tau_{i,j+k}\right) \leq pd\left(\tau_{i,j}\right)+k \tag{4.24}$$

However, since $pd\left(\tau_{i,j}\right) \stackrel{\text{def}}{=} a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil$ if $\tau_{i,j}$ is the p^{th} subtask a job $J_{i,q}$ released at time $a_{i,q}$ (Expression 4.1), the pseudo-deadlines of two different subtasks $\tau_{i,j}$ and $\tau_{i,j+r}$ of τ_i ($r>0$) are give by $pd\left(\tau_{i,j}\right) = a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil$ and $pd\left(\tau_{i,j+r}\right) = a_{i,q} + \left\lceil \frac{p+r}{U_i} \right\rceil$, respectively. Because, p and r are natural numbers and U_i is assumed to be smaller than or equal to 1, it holds that

$$\begin{aligned} pd\left(\tau_{i,j+r}\right) &\geq a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil + \left\lfloor \frac{r}{U_i} \right\rfloor \\ &\geq a_{i,q} + \left\lceil \frac{p}{U_i} \right\rceil + r \\ &\geq pd\left(\tau_{i,j}\right) + r \end{aligned}$$

Using this last expression in conjunction with Expression 4.24, we obtain

$$pd\left(\tau_{i,j+k}\right) = pd\left(\tau_{i,j}\right)+k \tag{4.25}$$

That is, there are exactly k time units separating the first and the $(k+1)^{\text{th}}$ pseudo-deadline.

Now assuming that $0 \leq k < \ell - 1$, then both k and $k+1$ are smaller or equal to $\ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$. We are therefore in case (b) for both subtasks $\tau_{i,j+k}$ and $\tau_{i,j+k+1}$. Hence,

Expression 4.25 yields

$$\begin{aligned} pd\left(\tau_{i,j+k+1}\right) - pd\left(\tau_{i,j+k}\right) &= \left(pd\left(\tau_{i,j}\right) + k + 1\right) - \left(pd\left(\tau_{i,j}\right) + k\right) \\ &= 1 \end{aligned}$$

thereby proving Expression 4.19.

If $k = \ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$, the corresponding subtask $\tau_{i,j+\ell}$ is in case (a). Therefore, $pd\left(\tau_{i,j+\ell}\right) > pd\left(\tau_{i,j}\right) + \ell$ (Expression 4.23). On the other hand, $\tau_{i,j+\ell-1}$ is in case (b) leading to $pd\left(\tau_{i,j+\ell-1}\right) = pd\left(\tau_{i,j}\right) + \ell - 1$ (Expression 4.25). Hence,

$$\begin{aligned} pd\left(\tau_{i,j+\ell}\right) - pd\left(\tau_{i,j+\ell-1}\right) &> \left(pd\left(\tau_{i,j}\right) + \ell\right) - \left(pd\left(\tau_{i,j}\right) + \ell - 1\right) \\ &> 1 \end{aligned}$$

and because $pd\left(\tau_{i,j+\ell}\right)$ and $pd\left(\tau_{i,j+\ell-1}\right)$ are both integers, it proves Expression 4.20. ■

This property can now be used to provide an expression of the generalized group deadline of a subtask $\tau_{i,j}$ in function of the urgency factor and the recovery time of the task τ_i at time t.

Lemma 4.5

Let t be the current time in a PFair schedule. Let τ_i be a task such that $U_i \leq 1$. If $\tau_{i,j}$ is the subtask of τ_i ready at time t, then

$$GD^*\left(\tau_{i,j}\right) = t + \mathrm{UF}_i(t) + \lceil \rho_i(t) \rceil - 1$$

Proof:

From Definition 4.14, the generalized group deadline $GD^*\left(\tau_{i,j}\right)$ of a subtask $\tau_{i,j}$ is defined as the earliest time t_G, where $t_G \geq pd\left(\tau_{i,j}\right)$, such that either

$$t_G = pd\left(\tau_{i,j+k}\right) \ \wedge \ b\left(\tau_{i,j+k}\right) = 0 \tag{4.26}$$

or

$$t_G = pd\left(\tau_{i,j+k}\right) + 1 \ \wedge \ \left(pd\left(\tau_{i,j+k+1}\right) - pd\left(\tau_{i,j+k}\right)\right) \geq 2 \tag{4.27}$$

for some $k \geq 0$.

As stated in Lemma 4.2, $pd\left(\tau_{i,j}\right) = t + \mathrm{UF}_i(t)$. Since from Definition 4.14, $GD^*\left(\tau_{i,j}\right) \geq pd\left(\tau_{i,j}\right)$, it holds that

$$GD^*\left(\tau_{i,j}\right) \geq t + \mathrm{UF}_i(t)$$

Moreover, from Lemma 4.4, the instant t is followed by exactly $\lfloor \rho_i(t) \rfloor$ pseudo-deadlines of τ_i separated by one time unit[6]. Formally, if $\ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$, then

$$pd\left(\tau_{i,j+k+1}\right) - pd\left(\tau_{i,j+k}\right) = 1, \qquad 0 \leq k < \ell - 1 \tag{4.28}$$

$$pd\left(\tau_{i,j+\ell}\right) - pd\left(\tau_{i,j+\ell-1}\right) \geq 2 \tag{4.29}$$

Therefore,

(A) *The condition expressed by* (4.27) *to have a generalized group deadline at the time instant* $pd\left(\tau_{i,j+k}\right) + 1$ *is not respected for* $0 \leq k < \ell - 1$.

Furthermore, since $pd\left(\tau_{i,j+k+1}\right) - pd\left(\tau_{i,j+k}\right) = 1$ for $0 \leq k < \ell - 1$ (Expression 4.28), the pseudo-deadline of $\tau_{i,j+k}$ is given by

$$pd\left(\tau_{i,j+k}\right) = pd\left(\tau_{i,j}\right) + k \tag{4.30}$$

Now, let us assume that we are at time t_k ($\geq t$), that we have executed the subtasks $\tau_{i,j}$ to $\tau_{i,j+k-1}$ within $[t, t_k)$ and that the active subtask of τ_i at time t_k is $\tau_{i,j+k}$. That is, k time units have been executed between t and t_k and assuming that $\tau_{i,j}$ is a subtask of the job $J_{i,q}$ released at time $a_{i,q}$, it holds that

$$\mathrm{exec}_i\left(a_{i,q}, t_k\right) - \mathrm{exec}_i\left(a_{i,q}, t\right) = k \tag{4.31}$$

Using Definition 4.2, the difference between the lag of τ_i at time t_k and t is given by

$$\mathrm{lag}_i(t_k) - \mathrm{lag}_i(t) = U_i \times (t_k - t) - \left(\mathrm{exec}_i(a_{i,q}, t_k) - \mathrm{exec}_i(a_{i,q}, t)\right)$$

[6]Note that if $\lfloor \rho_i(t) \rfloor = 1$ then Lemma 4.4 says that there is exactly one pseudo-deadline "separated" by one time unit which means that the two pseudo-deadlines $pd\left(\tau_{i,j}\right)$ and $pd\left(\tau_{i,j+1}\right)$ are separated by more than one time unit, i.e., $pd\left(\tau_{i,j+1}\right) - pd\left(\tau_{i,j}\right) \geq 2$.

Hence, using Expression 4.31

$$\mathrm{lag}_i(t_k) = \mathrm{lag}_i(t) + U_i \times (t_k - t) - k \tag{4.32}$$

Moreover, since $\tau_{i,j+k}$ is the subtask active at time t_k, applying Lemma 4.2, we have that

$$\mathrm{UF}_i(t_k) = pd\left(\tau_{i,j+k}\right) - t_k$$

and using Expression 4.30 and Lemma 4.2

$$\begin{aligned} \mathrm{UF}_i(t_k) &= pd\left(\tau_{i,j}\right) + k - t_k \\ &= \mathrm{UF}_i(t) + t + k - t_k \end{aligned} \tag{4.33}$$

Applying Expression 4.33 to Definition 4.13, we get that

$$\begin{aligned} \rho_i(t_k) &= \frac{\mathrm{lag}_i(t_k) + U_i \times (\mathrm{UF}_i(t_k) - 1)}{1 - U_i} \\ &= \frac{\mathrm{lag}_i(t_k) + U_i \times (\mathrm{UF}_i(t) - t + k - t_k - 1)}{1 - U_i} \end{aligned}$$

and Expression 4.32 leads to

$$\begin{aligned} \rho_i(t_k) &= \frac{\mathrm{lag}_i(t) + U_i \times (t_k - t) - k + U_i \times (\mathrm{UF}_i(t) + t + k - t_k - 1)}{1 - U_i} \\ &= \frac{\mathrm{lag}_i(t) + U_i \times (\mathrm{UF}_i(t) - 1) + U_i \times k - k}{1 - U_i} \end{aligned}$$

Finally, Definition 4.13 yields

$$\begin{aligned} \rho_i(t_k) &= \rho_i(t) + \frac{U_i \times k - k}{1 - U_i} \\ &= \rho_i(t) - k \end{aligned} \tag{4.34}$$

Therefore, since $\ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$, we have that $\rho_i(t_k) > 1$ for every k such that $0 \leq k < \ell - 1$. Hence, using Lemma 4.3, it holds that $b\left(\tau_{i,j+k}\right) = 1$ for every subtask $\tau_{i,j+k}$ such that $0 \leq k < \ell - 1$ (remember that $\tau_{i,j+k}$ is the subtask active at time t_k). It therefore results that

(B) *The condition expressed by* (4.26) *to have a generalized group deadline of* τ_i *at time* $pd\left(\tau_{i,j+k}\right)$ *is not respected for* $0 \leq k < \ell - 1$.

Hence, by (A) and (B), none of the conditions to have a generalized group deadline is encountered before $pd\left(\tau_{i,j+\ell-1}\right)$. That is,

$$GD^*\left(\tau_{i,j}\right) \geq pd\left(\tau_{i,j+\ell-1}\right)$$

Since $\ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$, Expression 4.34 implies that two different situations can hold at time $t_{\ell-1}$ (i.e., when $\tau_{i,\ell-1}$ is the active subtask); either $\rho_i(t_{\ell-1}) = 1$ or $1 < \rho_i(t_{\ell-1}) < 2$.

- If $\rho_i(t_{\ell-1}) = 1$ (i.e., $\rho_i(t) = \ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$) then because $\tau_{i,j+\ell-1}$ is the active subtask at time $t_{\ell-1}$ we get from Lemma 4.3 that

  $$b\left(\tau_{i,j+\ell-1}\right) = 0$$

  Therefore, from (4.26), $GD^*\left(\tau_{i,j}\right) = pd\left(\tau_{i,j+\ell-1}\right)$. Consequently, using Expression 4.30 for $k = \ell - 1$ and applying Lemma 4.2,

  $$\rho_i(t) = \lfloor \rho_i(t) \rfloor \Rightarrow GD^*\left(\tau_{i,j}\right) = t + \mathrm{UF}_i(t) + \lfloor \rho_i(t) \rfloor - 1 \tag{4.35}$$

- If $\rho_i(t_{\ell-1}) > 1$ (i.e. $\rho_i(t) > \ell \stackrel{\text{def}}{=} \lfloor \rho_i(t) \rfloor$) then we get from Lemma 4.3 that

  $$b\left(\tau_{i,j+\ell-1}\right) = 1$$

  Moreover, we know from Expression 4.29 that $pd\left(\tau_{i,j+\ell}\right) - pd\left(\tau_{i,j+\ell-1}\right) \geq 2$. Therefore, from Expression (4.27), $GD^*\left(\tau_{i,j}\right) = pd\left(\tau_{i,j+\ell-1}\right) + 1$. Consequently, using Expression 4.30 for $k = \ell - 1$ and applying Lemma 4.2,

  $$\rho_i(t) > \lfloor \rho_i(t) \rfloor \Rightarrow GD^*\left(\tau_{i,j}\right) = t + \mathrm{UF}_i(t) + \lfloor \rho_i(t) \rfloor - 1 + 1 \tag{4.36}$$

From the properties of the ceil and floor operators, Expressions 4.35 and 4.36 can be simplified in

$$GD^*\left(\tau_{i,j}\right) = t + \mathrm{UF}_i(t) + \lceil \rho_i(t) \rceil - 1$$

which states the Lemma. ■

Assuming that $\tau_{i,j}$ and $\tau_{p,q}$ are two subtasks eligible at time t, if $\mathrm{UF}_i(t) = \mathrm{UF}_p(t)$ and $\lceil \rho_i(t) \rceil > \lceil \rho_p(t) \rceil$ then using Lemma 4.5, there is $GD^*\left(\tau_{i,j}\right) > GD^*\left(\tau_{p,q}\right)$. Since $\mathrm{UF}_i(t)$

gives the distance of $\tau_{i,j}$'s pseudo-deadline from time t, the ceil value of the recovery time $\rho_i(t)$ can be seen as a measure of the generalized group deadline of $\tau_{i,j}$ relatively to $\tau_{i,j}$'s pseudo-deadline (see Figure 4.8(b)).

4.6.5 Equivalence between PD^{2*} and BF^2 Prioritization Rules

By literally translating Prioritization Rules 4.4.(i), (ii) and (iii) using Lemmas 4.2, 4.3 and 4.5, we obtain:

(i) $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$

(ii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \ \wedge \ \rho_i(t) > 1 \ \wedge \ \rho_p(t) = 1$

(iii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \ \wedge \ \rho_i(t) > 1 \ \wedge \ \rho_p(t) > 1 \ \wedge$
$\mathrm{UF}_i(t) + \lceil \rho_i(t) \rceil - 1 > \mathrm{UF}_p(t) + \lceil \rho_p(t) \rceil - 1$

which can be rewritten as

(i) $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$

(ii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \ \wedge \ \rho_i(t) > 1 \ \wedge \ \rho_p(t) = 1$

(iii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \ \wedge \ \rho_i(t) > 1 \ \wedge \ \rho_p(t) > 1 \ \wedge \ \lceil \rho_i(t) \rceil > \lceil \rho_p(t) \rceil$

Because the ties can be broken arbitrarily, this can be simplified in:

(i) $\mathrm{UF}_i(t) < \mathrm{UF}_p(t)$

(ii) $\mathrm{UF}_i(t) = \mathrm{UF}_p(t) \ \wedge \ \rho_i(t) > \rho_p(t)$

proving that Prioritization Rules 4.3 of BF^2 are equivalent to Prioritization Rules 4.4 of the slight variation of PD^2 presented in Section 4.6.1.

Hence, we proved that the rules to prioritize the tasks under BF^2 are equivalent to the rules of the slight variation of PD^2 named PD^{2*}. Since we proved that PD^{2*} respects all the task deadlines for the scheduling of sporadic task sets with unconstrained deadlines provided that $\delta_{\text{sum}} \leq m$ and $\delta_i \leq 1$ for every task τ_i (Theorem 4.3), BF^2 also meets all the deadlines for the scheduling of the same class of systems when the scheduler is invoked at each quantum of time.

Lemma 4.6

If the prioritization rules of BF^2 (Prioritization Rules 4.3) are used at every time unit to decide which eligible tasks must be scheduled in the next time slot, then there is $\text{lag}_i(t) < 1$ *for all* τ_i *and every time* t*, ensuring that all deadlines are respected for sporadic (and intra-sporadic, and dynamic) tasks with unconstrained deadlines, provided that* $\delta \leq m$ *and* $\delta_i \leq 1$ *for every* τ_i *in* τ*.*

In particular, for implicit deadline tasks:

Lemma 4.7

If the prioritization rules of BF^2 (Prioritization Rules 4.3) are used at every time unit to decide which eligible tasks must be scheduled in the next time slot, then there is $\text{lag}_i(t) < 1$ *for all* τ_i *and every time* t*, ensuring that all deadlines are respected for sporadic tasks with implicit deadlines, provided that* $U \leq m$ *and* $U_i \leq 1$ *for every* τ_i *in* τ*.*

4.7 Optimality of BF^2

Lemma 4.7 states that BF^2 is optimal if it is invoked at each and every time unit. However, as explained in the previous sections, in order to reduce the number of preemptions and migrations, BF^2 should be invoked only at boundaries and job arrivals. Let S_{BFair} denote the schedule produced by BF^2 when invoked at boundaries and job arrivals, and let S_{ERfair} be the schedule produced by BF^2 when invoked at every time units. To prove the optimality of BF^2 when invoked only at boundaries and job arrivals, we show in the remainder of this section that, even though tasks are not scheduled in the same order in S_{BFair} and S_{ERfair}, the same tasks are executed for the same amount of time between two instants corresponding to two boundaries. Therefore, since all deadlines are respected in S_{ERfair} (Lemma 4.7) and because deadlines occur only at boundaries (see Section 4.5.2), BF^2 must also meet all deadlines in S_{BFair}. Hence, BF^2 is optimal for the scheduling of sporadic tasks with implicit deadlines when it is invoked only at boundaries and job arrivals.

Let us first provide some precisions on the schedules S_{BFair} and S_{ERfair}. According to Section 4.4, a task τ_i may receive a time unit in S_{BFair} at time $t \in [b_k, b_{k+1})$

- as a mandatory unit. In this case, according to Step 1 in Section 4.4, we have $\text{lag}_i(b_{k+1}) \geq 1$ without executing this time unit;

- as an optional unit. Then, according to the definition of an eligible task for an optional time unit given at Step 2 in Section 4.4, we have $\text{lag}_i(b_{k+1}) > 0$ if τ_i does not receive this time unit.

Therefore, a task τ_i is eligible for a time unit (either mandatory or optional) in S_{BFair} whenever $\text{lag}_i(b_{k+1}) > 0$. Hence, we assume from this point onward, that, in S_{ERfair}, a task τ_i is also eligible for a time unit at any time $t \in [b_k, b_{k+1})$ if $\text{lag}_i(b_{k+1}) > 0$.[7] The goal of this definition of the eligibility of a task in S_{ERfair} is the following: if τ_i is eligible for a time unit in S_{BFair} then it is also eligible for a time unit in S_{ERfair}.

The proof of the optimality is made by induction on the boundaries. Assuming that every task in τ has been executed for the same amount of time in both S_{BFair} and S_{ERfair} until boundary b_k, then we prove that every task in τ is scheduled for the same amount of time between b_k and the next boundary b_{k+1} in S_{BFair} and S_{ERfair}. Notice that the induction hypothesis must be true at boundary b_0 (i.e., at the start of the schedule), since nothing has been executed yet. Hence, we only have to prove the induction step.

We start by proving an interesting property on the urgency factor and the recovery time of a task τ_i (Lemmas 4.8 and 4.9). Then, the induction step is proven in Lemma 4.10 and the optimality is stated in Theorem 4.4.

Lemma 4.8

Let $\text{exec}_i(t)$ *and* $\text{exec}_i(t')$ *be the amount of time the task* τ_i *has been executed until time* t *and* t' $(t' > t)$*, respectively. If* $\text{exec}_i(t) = \text{exec}_i(t')$ *then* $\text{UF}_i(t') = \text{UF}_i(t) - (t' - t)$ *and* $\rho_i(t') = \rho_i(t)$.

Proof:

Let us first consider the urgency factor of τ_i. From Definition 4.12, $\text{UF}_i(t') = \left\lceil \frac{1-\text{lag}_i(t')}{U_i} \right\rceil$.
Furthermore, from Definition 4.2

$$\text{lag}_i(t') = \text{lag}_i(t) + U_i \times (t' - t) \tag{4.37}$$

[7] Since $b_{k+1} > t$, we have $\text{lag}_i(b_{k+1}) > \text{lag}_i(t)$ if τ_i is not executed (Expression 4.6), thereby implying that $\text{lag}_i(b_{k+1}) > 0$ if $\text{lag}_i(t) > 0$. Therefore, τ_i is always eligible for a time unit when $\text{lag}_i(t) > 0$ which is a sufficient condition to have a correct schedule with an optimal ER-Fair scheduler such as BF^2[Srinivasan and Anderson 2002].

Hence, using both expressions together, it holds that

$$\begin{aligned} \mathrm{UF}_i(t') &= \left\lceil \frac{1-\mathrm{lag}_i(t)-U_i\times(t'-t)}{U_i} \right\rceil \\ &= \left\lceil \frac{1-\mathrm{lag}_i(t)}{U_i} - (t'-t) \right\rceil \\ &= \left\lceil \frac{1-\mathrm{lag}_i(t)}{U_i} \right\rceil - (t'-t) \end{aligned}$$

and from Definition 4.12

$$\mathrm{UF}_i(t') = \mathrm{UF}_i(t) - (t'-t) \tag{4.38}$$

Regarding the recovery time of τ_i, we have from Definition 4.13 that $\rho_i(t') = \frac{\mathrm{lag}_i(t')+(\mathrm{UF}_i(t')-1)\times U_i}{1-U_i}$. Using Equations 4.37 and 4.38, and re-applying Definition 4.12 afterward, we get

$$\begin{aligned} \rho_i(t') &= \frac{\mathrm{lag}_i(t)+U_i\times(t'-t)+(\mathrm{UF}_i(t)-(t'-t)-1)\times U_i}{1-U_i} \\ &= \frac{\mathrm{lag}_i(t)+(\mathrm{UF}_i(t)-1)\times U_i}{1-U_i} \\ &= \rho_i(t) \end{aligned}$$

Hence, the lemma. ■

Lemma 4.9

Let $\mathrm{exec}_i(t)$ *and* $\mathrm{exec}_i(t')$ *be the amount of time the task* τ_i *has been executed until time* t *and* t' $(t' > t)$*, respectively. If* $\mathrm{exec}_i(t) = \mathrm{exec}_i(t')$ *for all* $\tau_i \in \tau$ *then the priority order between all tasks at time* t' *is identical to the priority order computed at time* t*.*

Proof:

Let τ_k and τ_ℓ be any two distinct tasks in τ and let assume that τ_k has a higher priority than τ_ℓ at time t. This means that either $\mathrm{UF}_k(t) < \mathrm{UF}_\ell(t)$ or $\mathrm{UF}_k(t) = \mathrm{UF}_\ell(t) \wedge \rho_k(t) \geq \rho_\ell(t)$ (see Prioritization Rules 4.3). Since, by assumption, $\mathrm{exec}_k(t) = \mathrm{exec}_k(t')$ and $\mathrm{exec}_\ell(t) = \mathrm{exec}_\ell(t')$, we can use Lemma 4.8. This leads to $\mathrm{UF}_k(t') = \mathrm{UF}_k(t) - (t'-t)$, $\mathrm{UF}_\ell(t') = \mathrm{UF}_\ell(t) - (t'-t)$, $\rho_k(t') = \rho_k(t)$ and $\rho_\ell(t') = \rho_\ell(t)$. Hence,

either $\mathrm{UF}_k(t') < \mathrm{UF}_\ell(t')$ or $\mathrm{UF}_k(t') = \mathrm{UF}_\ell(t') \wedge \rho_k(t') \geq \rho_\ell(t')$, thereby implying that τ_k has a higher priority than τ_ℓ at time t' (see Prioritization Rules 4.3). Applying this argument to every pair of tasks in τ states the lemma. ■

Lemma 4.10

If every task in τ was executed for the same amount of time in both S_{BFair} and S_{ERfair} until boundary b_k, then every task in τ executes for the same amount of time between b_k and the next boundary b_{k+1} in S_{BFair} and S_{ERfair}.

Proof:

In this proof, we build the schedules S_{ERfair} and S_{BFair} in parallel, and, for each decision taken in S_{ERfair} we verify that the same decision is taken in S_{BFair}.

Let us assume that for each time unit allocated to a task τ_i before time t ($b_k \leq t < b_{k+1}$) in S_{ERfair}, a time unit is also allocated to τ_i in S_{BFair}. Note that this claim is true at boundary b_k by the lemma assumption.

At time t, in S_{ERfair}, the m highest priority tasks are selected to execute. Let τ_i be the task with the *highest* priority executed in S_{ERfair} which is not executed in S_{BFair} yet. That is, we assume that the execution time allocated to every task in S_{ERfair} until time $t+1$ is identical to the execution time allocated to the same tasks in S_{BFair}, except for τ_i which received one more time unit in S_{ERfair}. There are two cases considering the urgency factor of τ_i:

1. If $\mathrm{UF}_i(t) \leq (b_{k+1} - t)$ (i.e., the urgency factor is not greater than the time separating b_{k+1} from t), then, according to Definition 4.12, it holds that the allocation error of τ_i would be at least equal to 1 at the next boundary b_{k+1} if we do not execute this time unit of τ_i before b_{k+1} (i.e., $\mathrm{lag}_i(b_{k+1}) \geq 1$). Therefore, according to Step 1 in Section 4.4, this time unit is allocated as a mandatory time unit to τ_i in the interval $[t, b_{k+1})$ in S_{BFair}.

2. $\mathrm{UF}_i(t) > (b_{k+1} - t)$: Since τ_i is the highest priority task at time t, applying Lemma 4.9, we get that τ_i also has the highest priority at time b_{k+1}. Hence, according to Algorithm 4.3, τ_i is chosen for an optional time unit in S_{BFair}. Note that, whatever the arrival time of τ_i, BF^2 ensures that the optional time units are always allocated to tasks with the highest priority in S_{BFair} (Lemma 4.1). Hence, we can be sure that S_{BFair} actually executes this optional time unit.

In conclusion, for each time unit allocated to a task τ_i in S_{ERfair}, a time unit is also granted to τ_i in S_{BFair} in the interval $[b_k, b_{k+1})$. ■

Theorem 4.4

For any set τ of sporadic tasks with implicit deadlines executed on m identical processors, S_{BFair} respects all task deadlines provided that $U = \sum_{\tau_i \in \tau} U_i \leq m$ and $U_i \leq 1$, $\forall \tau_i \in \tau$.

Proof:

By iteratively applying Lemma 4.10 to the intervals $[0, b_1)$, $[b_1, b_2)$, $[b_2, b_3)$, *etc.*, we prove that every task $\tau_i \in \tau$ is executed for the same number of time units in S_{ERfair} and S_{BFair} in any interval bounded by two boundaries. Since Lemma 4.7 implies that S_{ERfair} respects all task deadlines, and because Expression 4.10 imposes that task deadlines coincide with boundaries, it must hold that all task deadlines are also met in S_{BFair}. ■

This last lemma proves the optimality of BF^2 for the scheduling of sporadic tasks with implicit deadlines such that $U = \sum_{\tau_i \in \tau} U_i \leq m$ and $U_i \leq 1$, $\forall \tau_i \in \tau$. Furthermore, applying Theorem 3.6, we can state that any set of sporadic tasks with *unconstrained* deadlines is schedulable by BF^2 provided that $\delta = \sum_{\tau_i \in \tau} \delta_i \leq m$ and $\delta_i \leq 1$, $\forall \tau_i \in \tau$. Hence,

Theorem 4.5

For any set τ of sporadic tasks with unconstrained deadlines executed on m identical processors, BF^2 respects all task deadlines provided that $\delta = \sum_{\tau_i \in \tau} \delta_i \leq m$ and $\delta_i \leq 1$, $\forall \tau_i \in \tau$.

4.8 Implementation Considerations

In this section, we discuss three improvements of BF^2 that can help to reduce its scheduling overheads.

4.8.1 Work Conservation

After having allocated the optional time units for active tasks, some processors can still have idle time units since the platform may not be necessarily fully utilized as well as not all sporadic tasks are always active during the schedule. To efficiently exploit these processor idle times, a work conserving technique can be added to BF^2. That is, no processor should be idle if at least one non-executed task has some remaining work. As an example of such work conserving technique; whenever a processor π_j is idle, the non-running task with the earliest deadline (if any) is executed on processor π_j.

Note that this improvement of BF^2 does not impact its optimality. Indeed, it was proven in [Srinivasan and Anderson 2005b] that PD^2 is optimal for the scheduling of *generalized intra-sporadic tasks*. This model of tasks assumes that some subtasks $\tau_{i,j}$ of some tasks τ_i can be absent during the scheduling of the tasks set τ. As stated in Appendix A, PD^{2*} — i.e., the slight variation of PD^2 we introduced in Section 4.6.1 — is also optimal for the scheduling of such tasks and the proofs of Sections 4.6 and 4.7 can be used to prove the optimality of BF^2 for the scheduling of generalized intra-sporadic tasks.

Now, let us assume that the task set τ has a total density smaller than m. There exists a correct schedule as stated by Theorem 4.4. However, the platform is not fully utilized and it must therefore exist instants where processors remain idle in the schedule produced by BF^2. Let us execute a subtask $\tau_{i,j}$ of the task τ_i in one of these idle times. This subtask $\tau_{i,j}$ will not be present anymore in the system when it should be executed in the original non work conserving BF^2 schedule. Hence, BF^2 will have to take other scheduling decisions relying on the actual state of the system. This new system state is identical to the state that the system would have had if τ_i was a generalized intra-sporadic task where the subtask $\tau_{i,j}$ would have been absent.

Because BF^2 is optimal for the scheduling of generalized intra-sporadic tasks, adding a work conservation technique which executes subtasks earlier than initially expected, does not jeopardize the optimality of BF^2.

4.8.2 Clustering

Clustering techniques as discussed in [Qi et al. 2011; Andersson and Tovar 2006] can also help reducing the number of task migrations and preemptions when the system is not fully utilized. They divide the platform into clusters of size k (i.e., subsets of k processors).

Then, a bin-packing algorithm can be applied to dispatch the tasks among the clusters such that the total utilization in each cluster does not exceed k. An optimal scheduling algorithm is then executed on each cluster independently.

Additionally, in order to minimize the amount of preemptions and migrations, every task with a utilization greater than or equal to $\frac{k}{k+1}$ receives its own processor.

Using this approach, an optimal scheduling algorithm such as BF^2 can correctly schedule any task set with a total utilization $U \leq \left(\frac{k}{k+1} \times m\right)$ [Andersson and Tovar 2006].

Hence, we can compute the smallest value for k such that a given task set τ with a total utilization U remains schedulable. By minimizing the number of processors k in each cluster, we also reduce the number of tasks which interact with each other, thereby decreasing the number of preemptions and migrations.

Note that this clustering technique reduces to a fully partitioned scheduling algorithm when the total utilization of the platform is smaller than 50%.

4.8.3 Semi-Partitioning and Supertasking

Nowadays, most continuous-time scheduling algorithms are based on a semi-partitioned scheme in which most tasks are statically assigned to one processor and only a few tasks can migrate.

In the case of discrete-time systems, *supertasking* techniques [Moir and Ramamurthy 1999; Holman and Anderson 2001, 2003b] can be used to create a semi-partitioned algorithm. In such approaches, tasks are grouped in servers scheduled with a proportionate fair or boundary fair algorithm. Then, the component tasks of each server are scheduled using their own *uniprocessor* scheduling algorithm (EDF for instance). That is, servers are first scheduled with BF^2 or PD^2 and whenever a server S_p should be running on a processor π_j, the component task $\tau_i \in S_p$ with the highest priority according to the scheduling algorithm adopted by S_p, is executed on π_j.

Finally, to produce a semi-partitioned algorithm, one server is statically assigned to each processor and the remaining servers migrate between all processors.

This technique has three main interests:

- It reduces the number of migrating tasks;
- It enables to force the execution of some specific tasks on specific processors (for

instance, if a task has to be executed on a processor with an access to a given resource to produce a result);

- It provides a solution to the synchronization problem of *inter-dependent* tasks. Indeed, since each server can be considered as an independent processor executing its own scheduling algorithm, by grouping inter-dependent tasks in a same server, we can apply well performing synchronization algorithms for the scheduling of inter-dependent tasks on uniprocessor platforms (see [Buttazzo 2008] for examples).

In [Moir and Ramamurthy 1999], the authors proposed to compute the *utilization* U_{S_p} of a server S_p as follows

$$U_{S_p} = \sum_{\tau_i \in S_p} U_i \tag{4.39}$$

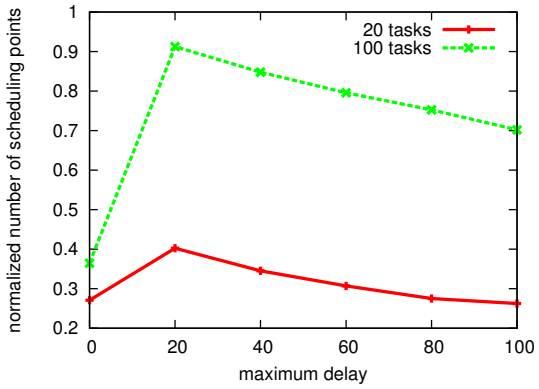
Unfortunately, even if they proved that a feasible schedule always exists, they also showed that there are feasible task sets which are not correctly schedulable with the today's known PFair algorithms when using Equation 4.39 to compute the server utilization [Moir and Ramamurthy 1999]. Consequently, supertasking techniques do usually not maintain the optimality. Nevertheless, several algorithms were proposed [Holman and Anderson 2001, 2003b] to compute server utilizations allowing to respect all job deadlines while minimizing the platform utilization loss. With the best performing method, this utilization loss remains smaller than 1% in average for PFair algorithms (assuming that EDF is used to schedule the component tasks of each server) [Holman and Anderson 2003b]. However, a generalization of these methods to BFair algorithms must still be investigated.

4.9 Experimental Results

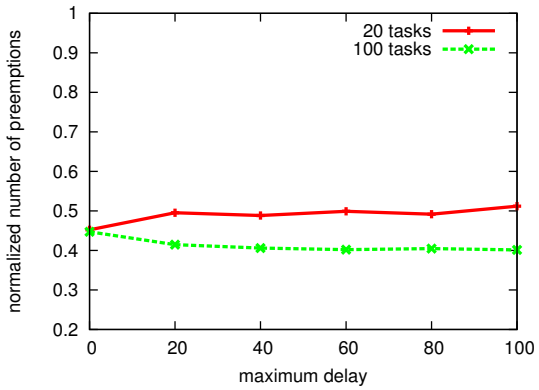
The decisions taken by BF^2 with the work conserving technique presented in Section 4.8.1 were simulated to evaluate the performances of this new algorithm. We then compared these results to the state-of-the-art scheduler for discrete time systems — namely, PD^2.[8]

The parameter used for the simulator are as follows. Each simulated task set contains 20 or 100 sporadic tasks. The periods of tasks are randomly generated within $[10, 100]$ under a uniform distribution. The worst-case execution time of a task is randomly generated between 1 and its period. For each job, its arrival delay is randomly chosen between 0

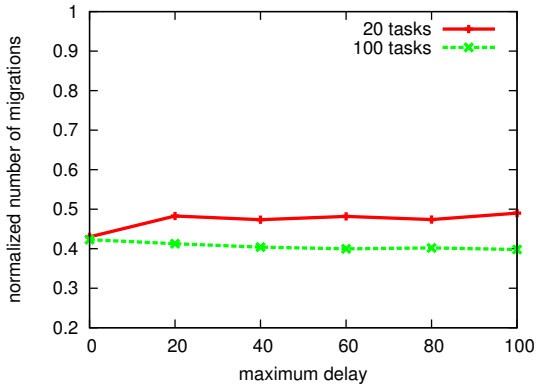
[8]The simulator including a description of BF^2 and PD^2 was developed by Prof. Dakai Zhu and his team following the algorithms presented in this chapter. The presented results are all produced with their simulator.



(a)



(b)



(c)

Figure 4.9: Normalized number of scheduling points, preemptions and task migrations vs. maximum delay.

and a maximum delay (which is a variable in Figures 4.9(a) to (c) presenting the results). We execute each task set from time 0 to time 100,000. Each point in Figures 4.9(a) to (c), is the average of the results of 100 randomly generated task sets.

By varying the maximum delay, Figures 4.9(a), (b) and (c) shows the normalized (i.e., the results of BF^2 are divided by those of PD^2) number of scheduling points, preemptions and migrations generated by BF^2, respectively.

We can see on Figure 4.9(a) that BF^2 can reduce the number of scheduling points by more than 70% (i.e., BF^2 needs less than 30% of the number of scheduling points generated by PD^2) when each task set contains 20 tasks. However, as the number of tasks grows, more time instants are likely to be a boundary or the arrival time of a new job. Hence, the number of scheduling points increases. For task sets containing 100 tasks, BF^2 may need up to 94% of the scheduling points of PD^2 in average.

As shown in Figures 4.9(b) and (c), the number of preemptions and task migrations generated by BF^2 remains around 40% and 50%.

Note that the bounds on the maximum number of preemptions and migrations are identical for PD^2 and BF^2. Indeed, in the worst-case, there is a boundary after each system time unit and BF^2 must therefore take scheduling decisions at every system tick exactly like PD^2. In this worst-case scenario, there are at most m preemptions and m migrations at each system tick. Thus, the maximum number of preemptions and migrations that the jobs may incur with BF^2 cannot be worse than with PD^2.

All these results are similar to those previously obtained with BF — a boundary fair algorithm for *periodic* tasks with implicit deadlines (see Section 9) — presented in [Zhu et al. 2003, 2011].

4.10 Conclusions

In this chapter, we addressed the problem of scheduling sporadic tasks in *discrete-time* systems. We proposed a new optimal boundary-fair scheduling algorithm for sporadic tasks (named BF^2) and proved its optimality. BF^2 makes scheduling decisions only at expected task deadlines and new job releases. It is the first boundary fair scheduling algorithm that remains optimal for the scheduling of sporadic tasks with implicit deadlines in a discrete-time environment and meets all the deadlines of unconstrained deadline tasks as long as $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$.

Our simulation results show that BF^2 outperforms the state-of-the-art optimal discrete-time system scheduling algorithm (i.e., PD^2) with respect to the preemption and migration overheads. However, even though the number of preemptions and migrations has been

reduced in comparison with PD^2, it remains significant and we believe that it can still be improved. This latter consideration is the main purpose of the two next chapters.

Finally, it must still be shown that the total overheads caused by BF^2 are actually smaller than those of PD^2 when implemented in a real operating system. Hence, Prof. Dakai Zhu and his team are currently working on the implementation of PD^2 and BF^2 in Linux.

CHAPTER

5

Continuous-Time Systems

Reducing Task Preemptions and Migrations in a DP-Fair Algorithm

"Le problème avec notre époque
est que le futur n'est plus ce qu'il était."
('The trouble with our times
is that the future is not what it used to be.")

Paul Valéry

"Every day for us, something new.
Open mind for a different view
And nothing else matters"

Metallica

Contents

Abstract

Continuous-time algorithms have been extensively studied during the last decade. This interest for continuous-time systems can partially be explained by the simplicity of the solutions producing optimal scheduling algorithms when comparing to those for discrete-time systems. More general and better performing algorithms (in terms of run-time complexity, preemptions and migrations) can therefore be considered. These solutions eventually help to understand the mechanisms improving the performances of scheduling algorithms and can then be transferred to the discrete-time environment.

This chapter considers the case of EKG [Andersson and Tovar 2006]. EKG is a multiprocessor scheduling algorithm which is optimal for the scheduling of real-time periodic tasks with implicit deadlines. It is built upon a continuous-time model and consists in a semi-partitioned algorithm which adheres to the deadline partitioning fair (DP-Fair) theory. However, it was shown in recent studies that the division of the time in slices bounded by two successive deadlines and the systematic execution of migratory tasks in each time slice inherent in DP-Fair algorithms, significantly reduce the practicality of EKG. Nevertheless, its semi-partitioned approach increases the locality of the tasks in memories, thereby potentially lowering the time overheads imposed by task preemptions. Hence, we propose two techniques with the aim of reducing the amount of preemptions and migrations incurred by the system when scheduled with EKG, while maintaining the advantages of its semi-partitioned approach. The first improvement consists in a swapping algorithm which exchanges execution time between tasks and time slices. The second one aims at decreasing the number of time slices needed to ensure that all job deadlines are respected. Both have a strong impact on the number of preemptions and migrations while keeping the optimality of EKG.

Note: This work was carried out in collaboration with Prof. Shelby Funk (University of Georgia, USA). Some parts of this research have been published at the work-in-progress session of ECRTS 2011 [Nelissen et al. 2011c], by the ACM Special Interest Group on Embedded Systems Review [Nelissen et al. 2011d] and as a full paper at RTCSA 2012 [Nelissen et al. 2012c].

5.1 Introduction

The scheduling of periodic real-time tasks on uniprocessor platforms has been extensively studied over the years. Nowadays, well mastered and high performing algorithms exist. For instance, the earliest deadline first (EDF) algorithm is a simple *optimal* scheduling algorithm on uniprocessor platforms [Liu and Layland 1973]. That is, EDF respects all task deadlines for any feasible task set. Unfortunately, when straightforwardly extended to the multiprocessor real-time scheduling, EDF cannot guarantee that all task deadlines will still be respected. Nevertheless, the simplicity of EDF on the one hand, and its performances in terms of preemptions on the other hand, encourage the real-time community to keep EDF as a model for the design of new scheduling algorithms [Kato and Yamasaki 2008b; Kato et al. 2009; Burns et al. 2010; Regnier et al. 2011; Nelissen et al. 2012b].

Over the last two decades, numerous optimal multiprocessor scheduling algorithms for periodic tasks have been proposed [Baruah et al. 1996, 1995; Anderson and Srinivasan 2000a; Zhu et al. 2011; Funk et al. 2011; Cho et al. 2006; Funk and Nadadur 2009; Andersson and Tovar 2006; Megel et al. 2010]. Each new result attempts to outperform previous algorithms in terms of preemptions, migrations, number of scheduling points or complexity. Indeed, many studies state that the run-time overheads caused by these various factors impact the practicality of optimal multiprocessor scheduling algorithms [Brandenburg and Anderson 2009; Bastoni et al. 2010b, 2011]. Hence, an algorithm which was proven optimal in theory may perform really poorly in practice.

As already introduced in the previous chapter, the first optimal scheduling algorithms for multiprocessor platforms were based on the notion of *proportionate fairness* (e.g., PF [Baruah et al. 1993, 1996] and PD [Baruah et al. 1995]) and *early-release fairness* (e.g., ER-PD [Anderson and Srinivasan 2000a]). With these solutions each task is scheduled at a rate proportional to its utilization. These algorithms that are presented in detail in Chapter 4 are built on a discrete-time model. Hence, a task is never executed for less than one *system time unit* (which is based on a *system tick* as already mentioned in Section 2.3.2). This particularity intrinsic to discrete-time algorithms facilitates their integration into real-time operating systems. Indeed, many real-time operating systems take their scheduling decisions relying on this system tick. It is therefore quite unrealistic to schedule the execution of a task for less than one system time unit. However, in spite of this indisputable advantage, PF, PD and PD^2 also cause an extensive amount of task preemptions and migrations by taking scheduling decision at each and every time unit. This excessive number of context switches on the processors increases the run-time overheads

affecting the system schedulability.

Zhu et al. [2003, 2011] therefore proposed the *boundary fairness* solution to overcome the drawbacks of the proportionate fairness approach. With boundary fair algorithms, the scheduling decisions are taken only at the job deadlines rather than at every system time unit. Unfortunately, until our very recent works presented in Chapter 4, the boundary fairness theory never led to an optimal scheduling algorithm for *sporadic* tasks. Indeed, imposing to schedule tasks only for integer multiples of the system time unit highly constrains and complicates the scheduling decisions of the scheduling algorithm. Consequently, during the past few years, researchers focalized on the study of *continuous-time* systems instead of their discrete-time equivalents. In a continuous-time environment, task executions do not have to be synchronized on the system tick and tasks can therefore be executed for any amount of time. This constraint relaxation drastically eases the design of optimal scheduling algorithms for multiprocessor platforms by increasing the flexibility on the scheduling decisions.

The continuous-time counter-part for the boundary fairness theory is named *deadline partitioning fairness* (DP-Fairness). Many algorithms follow this technique to reach the optimality [Levin et al. 2010; Funk et al. 2011; Cho et al. 2006; Funaoka et al. 2008; Funk and Nadadur 2009; Funk 2010]. Levin et al. [2010] and Funk et al. [2011] formalized the DP-Fair theory explaining how the optimality could be reached on multiprocessor platforms by ensuring the *fairness* for all tasks at the deadlines of jobs released in the system. In this approach, the time is divided in *slices*. All tasks are assigned a *local execution time* in each time slice which is determined so as to ensure that all deadlines will be met.

In 2006, Andersson et al. developed the *semi-partitioned* algorithm EKG which categorizes tasks as *migratory* or *non-migratory* [Andersson and Tovar 2006]. Migratory tasks are shared between two processors and scheduled according to the DP-Fair approach. On the other hand, non-migratory tasks are statically assigned to one processor and scheduled under an EDF scheduling policy. The EKG algorithm seems very promising in terms of reducing preemptions and migrations. However, recent studies showed that the systematic execution of migratory tasks in each time slice significantly increases the number of preemptions and migrations compared to non-optimal scheduling algorithms and hence, has a negative impact on the schedulability of task systems on real computing platforms [Bastoni et al. 2011].

Some of the most recent scheduling algorithms are not based on the notion of fairness

but rather extend the mechanisms lying behind the uniprocessor algorithm EDF [Kato and Yamasaki 2008b; Kato et al. 2009; Burns et al. 2010; Regnier et al. 2011; Nelissen et al. 2012b]. These algorithms perform better than EKG in terms of average preemptions and migrations per job and can even compete with fully partitioned algorithms [Kato and Yamasaki 2008b; Kato et al. 2009; Regnier et al. 2011; Nelissen et al. 2012b]. However, each of these "unfair" algorithms suffers from at least one of the two following disadvantages: either the algorithm is not optimal, or, unlike EKG for which there is a limited number of tasks that can migrate between two different processors, all tasks can migrate between all processors. This last particularity increases the preemption and migration overheads by lowering the task code and data locality in memories (as shown in [Bastoni et al. 2010a] on Figures 2(b) and 2(d)).
Hence, the **goal of the work** presented in this chapter is to improve EKG so that its average number of preemptions and migrations gets close to the results of "unfair" algorithms while keeping its optimality and the advantages of its semi-partitioned approach.

Contribution: In this chapter, we propose two techniques to address the preemption and migration overheads in EKG. We first present a swapping algorithm which increases (or decreases) the time reserved for migratory tasks in each time slice, thereby suppressing migratory tasks (and their associated run-time overheads) from time slices where their execution is not required to keep a correct schedule. Then, we propose a set of rules determining if the length of a time slice can be extended without any risk of missing a task deadline. Hence, by reducing the number of time slices, we are able to decrease the associated preemptions, migrations and scheduling points. Finally, we propose simulation results illustrating the performances of our two improvement techniques of the EKG algorithm.

Organization of this chapter: Section 5.2 reviews the EKG algorithm and other major optimal schedulers for continuous-time systems. System models are presented in Section 5.3. Section 5.4 discusses the means to improve the performances of EKG and proposes two such solutions, namely the swapping and the skipping algorithm. Various considerations about the implementation and modeling of these algorithms are presented in Section 5.5. Finally, Section 5.6 presents our simulation results and Section 5.7 concludes the chapter.

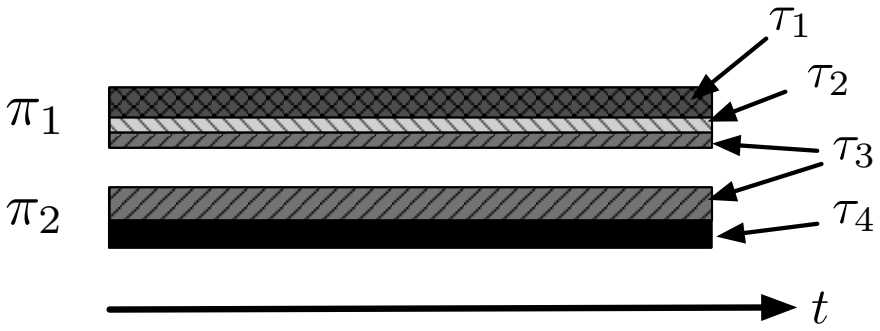


Figure 5.1: Fluid schedule of 4 tasks τ_1, τ_2, τ_3 and τ_4 with utilizations 0.25, 0.5, 0.8 and 0.45 respectively, on 2 identical processors π_1 and π_2.

5.2 Previous Works: Optimal Schedulers for Continuous-Time Systems

The first optimal algorithm for periodic tasks with implicit deadlines on multiprocessor platforms was proposed in [Baruah et al. 1993]. This algorithm was derived from the notion of *fluid schedule* already defined in Chapter 4 and repeated hereafter.

Definition 5.1 (*Fluid schedule*)

A schedule is said to be fluid if and only if at any time $t \geq 0$, the active job (if any) of every task τ_i arrived at time $a_i(t)$ has been executed for **exactly** *$U_i \times (t - a_i(t))$ time units until time t.*

The fluid schedule therefore implies that all tasks are permanently executed on the processing platform with an execution speed equal to their respective utilization. An illustration of this behavior is presented in Figure 5.1.

Unfortunately, a processor cannot execute more than one task at a time. Thus, even though the fluid schedule ensures that all the tasks always meet their deadlines, it is impossible to implement such a scheduler for systems composed of more tasks than processors because of the limitations of the actual computing platforms. Nevertheless, it was shown in [Levin et al. 2010; Funk et al. 2011] that respecting the fluid schedule only at the job deadlines rather than at any instant t, is sufficient to meet all the jobs deadlines. Hence, the authors of [Levin et al. 2010; Funk et al. 2011] formalized the notion of *Deadline Partitioning Fairness* (DP-Fairness). The deadline partitioning approach consists in dividing the time in *slices* bounded by two successive job deadlines. Formally, as in Chapter 4, if we let b_k denote the k^{th} time instant from the beginning of the schedule where at least one

job has its deadline, then we define the k^{th} time slice TS^k as the time interval extending from the instant b_k to b_{k+1} (with $b_0 \stackrel{\text{def}}{=} 0$). We say that b_k and b_{k+1} are the *boundaries* of TS^k and the length of TS^k is given by $L^k \stackrel{\text{def}}{=} b_{k+1} - b_k$. If $B \stackrel{\text{def}}{=} \{b_0, b_1, b_2, ...\}$ (with $b_k < b_{k+1}$ and $b_0 \stackrel{\text{def}}{=} 0$) denotes the set of boundaries encountered during the schedule, then a DP-Fair schedule is defined as follows:

Definition 5.2 (*Deadline partitioning fair schedule*)

A schedule is said to be deadline partitioning fair (or DP-Fair) if and only if for all $\tau_i \in \tau$ and for all $b_k \in B$: the active job at time b_k (if any) of every task τ_i arrived at time $a_i(b_k)$ has been executed for ***exactly*** *$U_i \times (b_k - a_i(b_k))$ time units until b_k.*

Therefore, DP-Fair algorithms enforce the *fairness* between tasks in each time slice by allocating to each task a time share whose duration is proportionate to its utilization. That is, within each time slice of length L^k, each task τ_i executes for $\ell_i^k = U_i \times L^k$ time units. We say that ℓ_i^k is the *local execution time* of the task τ_i in time slice TS^k.

To illustrate the DP-Fair principles we now present two related algorithms. We then expose in detail two other optimal scheduling algorithms for periodic tasks with implicit deadlines, namely RUN and EKG.

5.2.1 The LLREF Algorithm

The *largest local remaining execution time first* (LLREF) algorithm is a DP-Fair algorithm which is optimal for the scheduling of periodic tasks with implicit deadlines [Cho et al. 2006, 2007]. It relies on the *time and local remaining execution-time plane* (T-L plane) abstraction. A T-L plane represents the *local* remaining execution time $r_i^k(t)$ of each task τ_i in function of the time spent into the time slice TS^k (see Figure 5.2). Scheduling decisions within TS^k are then taken according to the remaining local execution times of all tasks and the remaining time until the end of the time slice.

As with any DP-Fair scheduler, in every time slice TS^k, each task τ_i must execute for its local execution time ℓ_i^k equals to its utilization U_i multiplied by the length L^k of TS^k. When the time slice begins (i.e., at the boundary b_k), the m processors of the processing platform are allocated to the m tasks with the largest local execution time. Then, during the execution of these tasks, two kinds of events can happen and impose to reschedule the system.

B event: a B event occurs when one of the running tasks τ_i has executed all its local

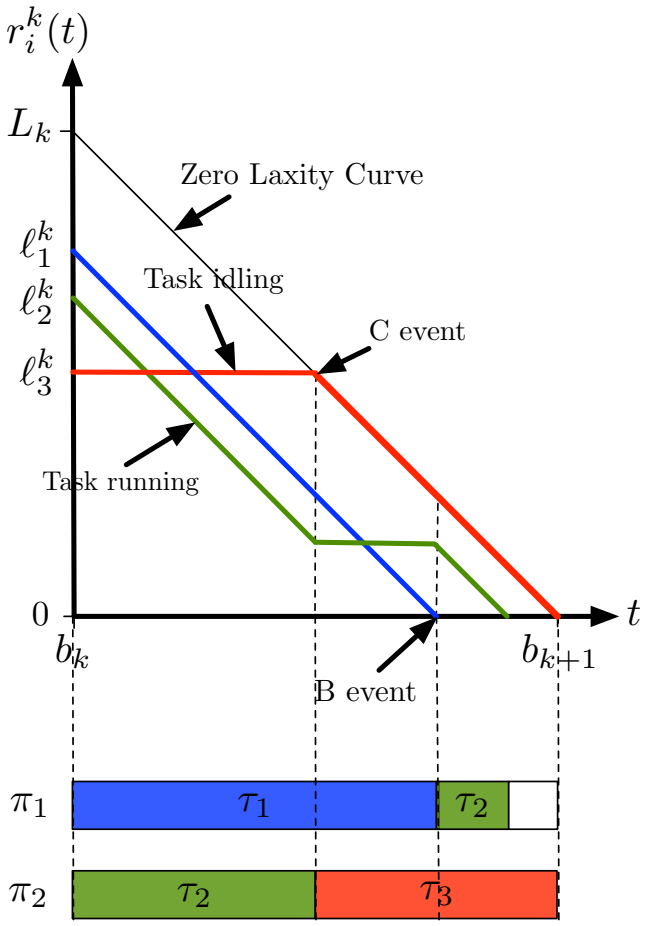


Figure 5.2: T-L plane of three tasks τ_1 (blue), τ_2 (red) and τ_3 (green) in a time slice TS^k representing a scenario of execution.

execution time (see Figure 5.2). A new ready (and not currently running) task with the largest *local remaining execution time* is therefore chosen to be executed instead of τ_i.

C event: a C event happens at time t ($b_k \leq t < b_{k+1}$) when the *laxity* of a non-running task τ_i reaches zero (see Figure 5.2). That is, the local remaining execution time $r_i^k(t)$ of τ_i at time t in TS^k is equal to the remaining time before the end of the time slice, i.e., $r_i^k(t) = b_{k+1} - t$. Hence, if 100% of the capacity of a processor is not reserved for the execution of τ_i until the end of the time slice TS^k, then τ_i will not be able to complete its local execution time by b_{k+1}. Whenever a C event occurs, the running task with the smallest local remaining execution time is preempted to execute τ_i instead.

LLREF was further improved using the notion of *extended T-L plane abstraction* [Funaoka et al. 2008]. The goal of this refined approach was to design work-conserving algorithms with lower scheduling overheads. This led to the proposition of a new algorithm named NVNLF (No Virtual Nodal Laxity First).

More recently, a variation of LLREF named LRE-TL was published in [Funk and Nadadur 2009; Funk 2010]. LRE-TL has three main advantages over LLREF:

1. it improves the run-time complexity of LLREF;
2. it reduces the number of preemptions and migrations per time slice by a factor of n (i.e. the number of tasks in the system);
3. it is optimal for the scheduling of sporadic tasks with unconstrained deadlines on both identical and uniform multiprocessor platforms (assuming that $\delta \leq m$).

5.2.2 The DP-Wrap Algorithm

DP-Wrap is another optimal scheduling algorithm for the scheduling of periodic and sporadic tasks with constrained deadlines [Levin et al. 2010; Funk et al. 2011]. It differs from LLREF in regard of the algorithm utilized to schedule the task within each time slice.

As with LLREF, each task τ_i is executed for a local execution time $\ell_i^k \stackrel{\text{def}}{=} U_i \times L^k$ in each time slice TS^k. These local execution times are then assigned to the processors following a slight variation of the *next fit* heuristic which was inspired by McNaughton's wrap-around algorithm [McNaughton 1959]. Tasks are assigned to a processor π_j as long as (i) the total execution time allocated to this processor is not greater than the length L^k of the time slice TS^k and (ii) the total execution time allocated to each processor π_1 to π_{j-1} is equal to L^k. Also, whenever the assignment of a task to π_j would cause the amount of execution time allocated to π_j to exceed the length L^k of TS^k, the task is "split" between π_j and the next processor π_{j+1} so that the total execution time assigned to π_j is exactly equal to L^k.

Example:

> *Figure 5.3 shows an example of the task local execution time assignment. It considers five tasks* τ_1 *to* τ_5 *with local execution times of* 8, 5, 4, 9 *and* 4 *time units respectively to schedule in the time slice* TS^0 *of length* 10. *We first schedule the execution of* τ_1 *from time* 0 *to* 8 *on processor* π_1. *Since there are only two time units left on* π_1 *before*

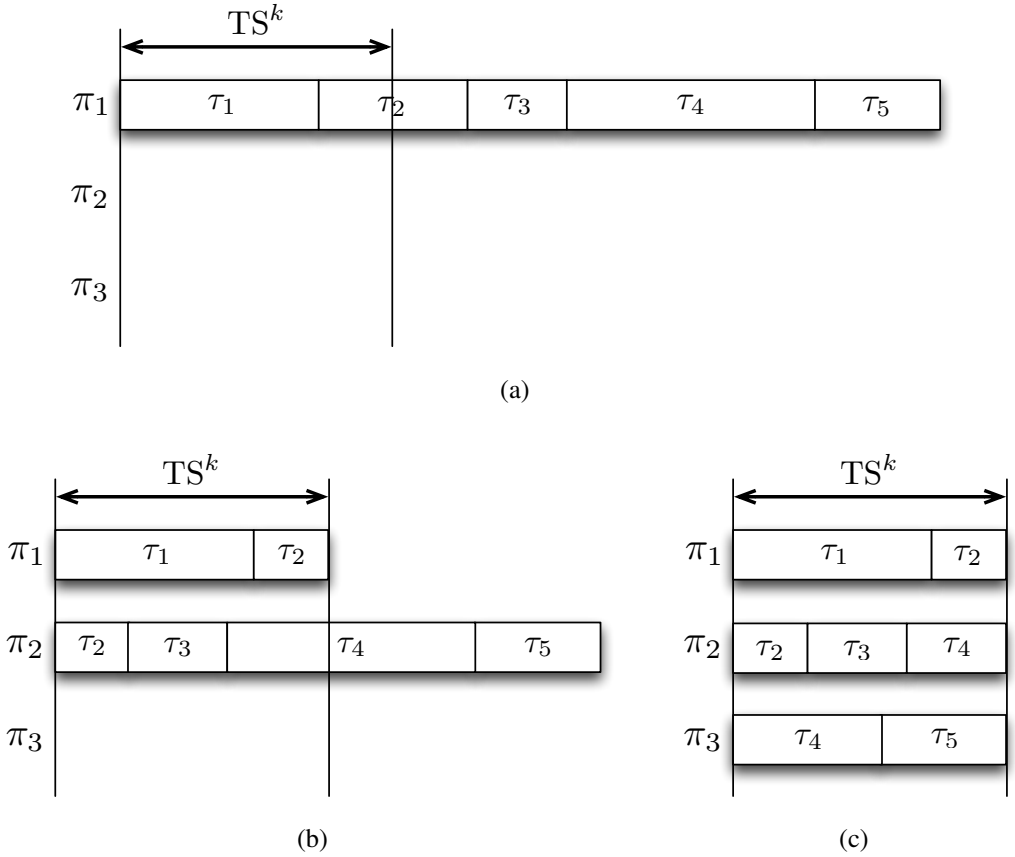
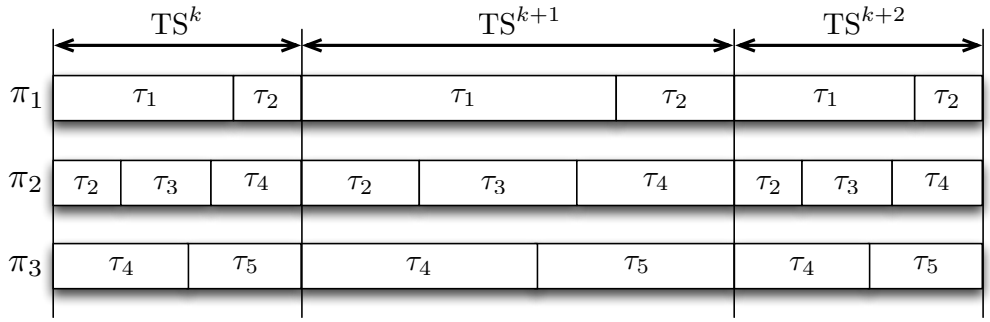


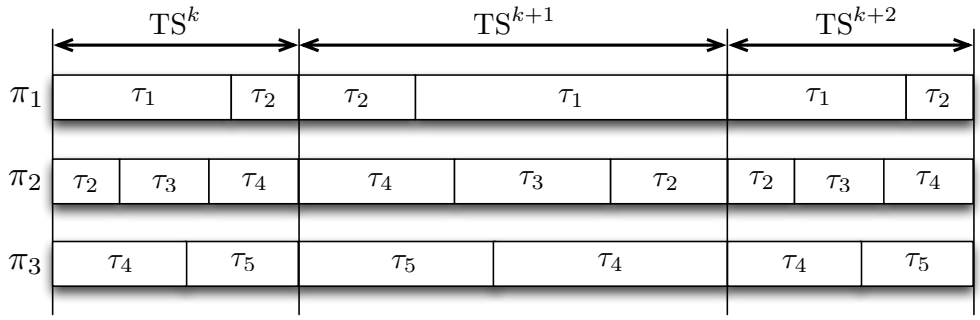
Figure 5.3: Scheduling of the local execution times of 5 tasks τ_1 to τ_5 on 3 processors in a time slice TS^k.

the end of the time slice, we split τ_2 between the processors π_1 and π_2 and we assign two time units of τ_2 on π_1 and the 3 other time units on π_2. Then, τ_3 is scheduled on π_2 and the three remaining time units of π_2 are allocated to τ_4. Since τ_4 has a local execution time of 9 time units, 6 time units are also reserved for τ_4 on processor π_3. Finally, the last 4 time units available on π_3 are allotted to the task τ_5.

Note that if all the tasks are periodic with implicit deadlines, then the pattern of the schedule computed for each time slice is identical (see Figure 5.4(a)). Only the length (i.e., the time between two consecutive boundaries) of this schedule may vary. Consequently, we always have the same $(m-1)$ tasks that migrate. Exploiting this property, the amount of preemptions and migrations can be reduced utilizing a *mirroring mechanism* that keeps the tasks that were executing before the boundary b_{k+1}, running on the same processors after b_{k+1} (see Figure 5.4(b)). Hence, if as on Figure 5.4(b), the schedule of odd-numbered time slices are the mirror image of the schedule of even-numbered



(a)



(b)

Figure 5.4: Scheduling with DP-Wrap of 5 periodic tasks with implicit deadlines in three consecutive time slices (a) without using the mirroring mechanism and (b) with the mirroring of TS^{k+1}.

time slices, then the number of preemptions in each time slice is decreased by m and the number of migrations is divided by two.

As a result, for a set of periodic tasks with implicit deadlines, there are $n+m-1$ preemptions and $2 \times (m-1)$ migrations per time slice when the mirroring technique is not implemented, but $n-1$ preemptions and $m-1$ migrations when odd-numbered time slices are mirrored [Levin et al. 2010; Funk et al. 2011].

It is worth to notice that the number of preemptions and migrations is a linear function of the number of time slices encountered during the schedule. Hence, the authors of [Funk et al. 2011] investigated a mechanism avoiding some time slice boundaries in order to extend the time slice length and eventually reduce their number. Since some tasks could have their deadlines within an extended time slice TS^k, their priority over the other active tasks is increased within TS^k. We encourage the reader to consult [Funk et al. 2011] for further information on this boundary skip mechanism for DP-Wrap.

Note that DP-Wrap can easily be adapted to the scheduling of sporadic task sets as

explained in [Levin et al. 2010; Funk et al. 2011]. However, in this case, the mirroring technique does not improve the number of preemptions and migrations anymore.

5.2.3 The RUN Algorithm

RUN is one of the most recent optimal algorithms for the scheduling of periodic tasks with implicit deadlines [Regnier et al. 2011; Regnier 2012]. Its name stands for ***R**eduction to **UN**iprocessor*. Indeed, RUN solves the multiprocessor scheduling problem by reducing its resolution to the study of an equivalent uniprocessor system. A well performing uniprocessor scheduling algorithm such as EDF can therefore be used to produce the tasks schedule.

All the theory developed around RUN is based on a simple idea: it is equivalent to schedule the task idle times than the task execution times. This approach named *dual scheduling* had already been investigated in a few previous works [Baruah et al. 1995; Levin et al. 2009; Funk et al. 2011; Zhu et al. 2011]. The dual schedule of a set of tasks τ consists in the schedule produced for the dual set τ^* defined as follows:

Definition 5.3 (*Dual task*)

τ_i^* is the dual task of τ_i if τ_i^* has the same period and deadline then τ_i and has a utilization $U_i^* \stackrel{\text{def}}{=} 1 - U_i$.

Definition 5.4 (*Dual task set*)

τ^* is the dual task set of τ if (i) for each task $\tau_i \in \tau$ then τ^* contains the dual task τ_i^*, and (ii) for each $\tau_i^* \in \tau^*$, there is $\tau_i \in \tau$.

The schedule of τ^* therefore represents the schedule of the idle times of the tasks in τ implying that τ's schedule can be derived from τ^*'s schedule. Indeed, if τ_i^* is running at time t in the dual schedule then τ_i must stay idle in the actual schedule of τ (also called *primal* schedule). Inversely, if τ_i^* is idle in the dual schedule then τ_i must execute in the primal schedule. In the following, we say that a DUAL operation is applied to the task set τ when the dual task set τ^* is built from τ.

Example:

Figure 5.5 shows an example of the correspondence between the dual and the primal schedule of a set τ composed of three tasks executed on two processors. In this example the three tasks τ_1 to τ_3 have a utilization of $\frac{2}{3}$ implying that their dual tasks τ_1^* to τ_3^* have utilizations of $\frac{1}{3}$. The dual task set is therefore schedulable on one proces-

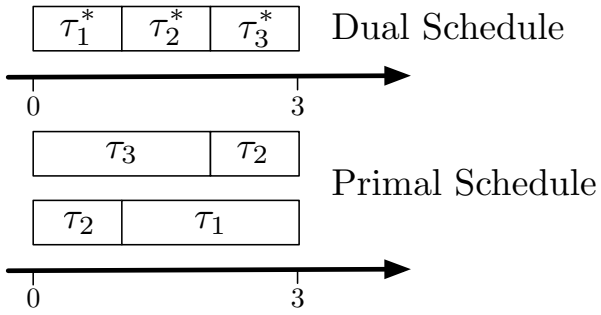


Figure 5.5: Correspondence between the primal and the dual schedule on the three first time units for three tasks τ_1 to τ_3 with utilizations of $\frac{2}{3}$ and deadlines equal to 3.

sor while the primal tasks τ_1 to τ_3 need two processors. Whenever a dual task τ_i^* is running in the dual schedule, the primal task τ_i remains idle in the primal schedule. And inversely, when τ_i^* is idling in the dual schedule then τ_i is running in the primal schedule. Note that the number of processors does not always diminish in the dual schedule. This actually depends on the utilization of the tasks in the scheduled task set.

The cornerstone of RUN is enunciated in Lemma 5.1. This lemma was proven in [Regnier et al. 2011] and can be rephrased as follows:

Lemma 5.1

Let τ be a set of n periodic tasks. If the total utilization of τ is U, then the utilization of τ^* is $U^* \stackrel{\text{def}}{=} n - U$.

Hence, if U is an integer then, according to Theorem 3.4 presented on page 48, τ and τ^* are feasible on $m = U$ and $m^* = n - U$ processors, respectively. Consequently, in systems where n is small enough to get

$$(n - U) < m \tag{5.1}$$

we can reduce the number of processors that must be taken into account and hence reduce the complexity of the scheduling problem if we schedule the dual task set τ^* instead of τ. For instance, in the last example illustrated on Figure 5.5, we have $U = 2$ thereby implying that τ is feasible on two processors. However, we have $n = 3$, which leads to $U^* = n - U = 1$. Hence, the dual task set τ^* is feasible on one processor.

To enforce Expression 5.1 being true, RUN first uses a bin-packing algorithm in order to PACK the tasks in as few bins as possible (each with a size smaller than or equal

to 1). These bins are called *servers* and can be assimilated to the supertasking approach discussed in Sections 4.8.3 and 5.2.4. Each server S contains a set of tasks with an accumulated utilization $U_S = \sum_{\tau_i \in S} U_i \leq 1$. S inherits the deadlines of all its component tasks and releases a job at every such deadline. After this packing phase, RUN applies the DUAL operation on the packed servers.

By executing these two operations (PACK and DUAL) recursively, Regnier et al. [2011] proved that the number of processors eventually reaches one. Hence, the initial multiprocessor scheduling problem is reduced to the scheduling of a uniprocessor system.

The reduction of the multiprocessor problem to its uniprocessor equivalent is made *offline* and each application of the PACK and DUAL operations corresponds to a reduction level.

Example:

Let τ be composed of 6 tasks τ_1 to τ_6 such that $U_1 = U_2 = U_3 = U_4 = 0.6$ and $U_5 = U_6 = 0.3$. The total utilization of τ is 3 and τ is therefore feasible of 3 processors. If we directly apply the DUAL operation, we would have $U_1^* = U_2^* = U_3^* = U_4^* = 0.4$ and $U_5^* = U_6^* = 0.7$. Hence, U^* would be equal to 3 and we would gain nothing by scheduling the dual task set τ^* instead of τ.

Therefore, as shown on Figure 5.6, RUN first applies the PACK operation which creates 5 servers S_1^1 to S_5^1 such that $S_1^1 = \{\tau_1\}$, $S_2^1 = \{\tau_2\}$, $S_3^1 = \{\tau_3\}$, $S_4^1 = \{\tau_4\}$ and $S_5^1 = \{\tau_5, \tau_6\}$. Hence, $U_{S_1^1} = U_{S_2^1} = U_{S_3^1} = U_{S_4^1} = U_{S_5^1} = 0.6$ and $U_{S_1^{1*}} = U_{S_2^{1*}} = U_{S_3^{1*}} = U_{S_4^{1*}} = U_{S_5^{1*}} = 0.4$ (where $U_{S_i^{r*}}$ is the utilization of S_i^{r*}, the i^{th} dual server of reduction level r). The set of dual servers now has a total utilization $U_{S^1}^* = 2$ implying that this set of dual servers in feasible on 2 processors.

If we apply once more the PACK and DUAL operation, we have three new servers $S_1^2 = \{S_1^{1*}, S_2^{1*}\}$, $S_2^2 = \{S_3^{1*}, S_4^{1*}\}$ and $S_3^2 = \{S_5^{1*}\}$ with utilizations $U_{S_1^2} = U_{S_2^2} = 0.8$ and $U_{S_3^2} = 0.4$. The dual servers of this second reduction level therefore have utilizations $U_{S_1^{2*}} = U_{S_2^{2*}} = 0.2$ and $U_{S_3^{2*}} = 0.6$. Consequently, the total utilization of the dual set of servers of this second reduction level is equal to 1, implying that they are feasible on a uniprocessor platform.

The *online* work of RUN consists in deriving the schedule of τ from the schedule constructed with EDF at the final reduction level (i.e., the equivalent uniprocessor system). This is achieved by applying the two following rules at each reduction level:

1. each server of reduction level r which is running in the primal schedule at time t ex-

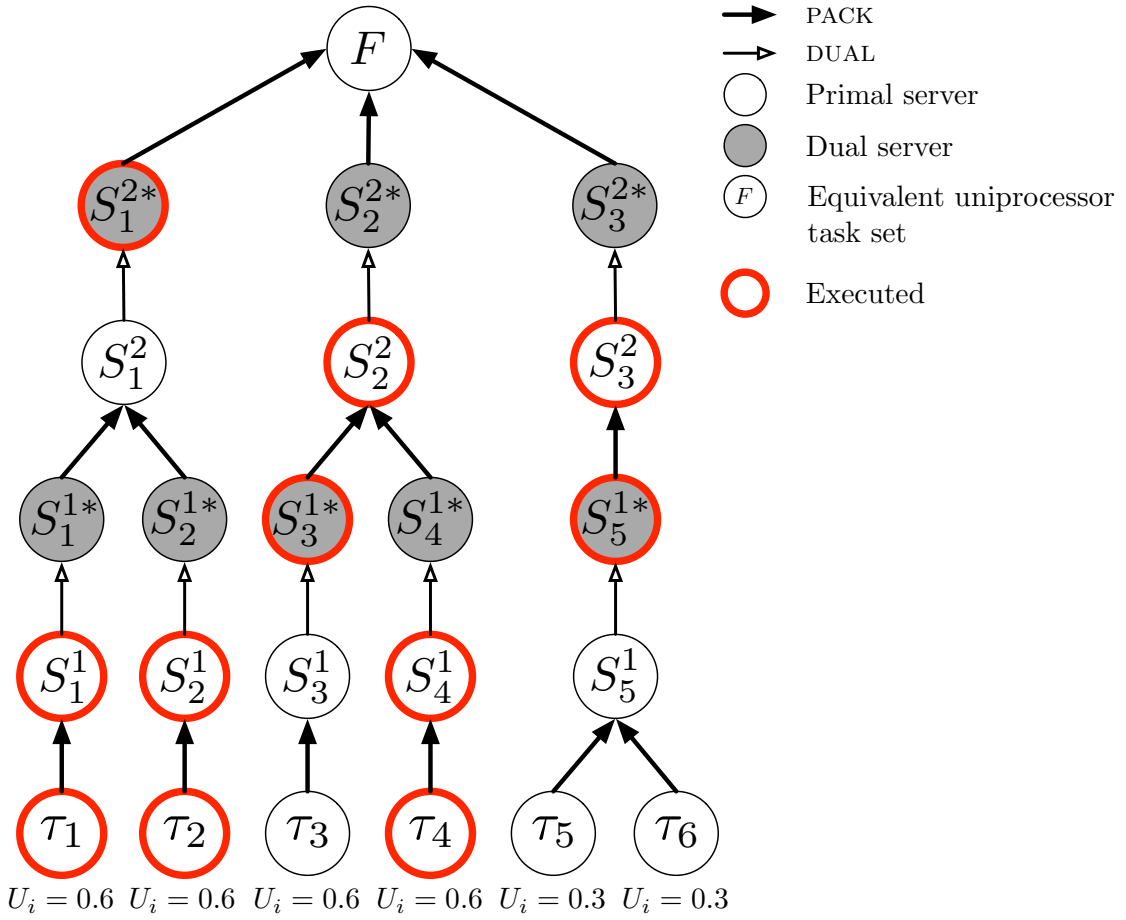


Figure 5.6: Example of the the execution of the algorithm RUN.

ecutes its component server (or task) with the earliest deadline in the dual schedule of reduction level $r-1$;

2. each server of reduction level $r-1$ which is not running in the dual schedule is executed in the primal schedule and inversely, each server which is running in the dual schedule is kept idle in the primal schedule.

Example:

Let us take the same task set τ than in the previous example and let us assume that each of the six tasks $\tau_i \in \tau$ has an active job at time t such that $d_1(t) < d_2(t) < d_3(t) < d_4(t) < d_5(t) < d_6(t)$. Since a server inherits the deadlines of its component tasks, each server S_i^1 (with $1 \leq i \leq 5$) at the first reduction level inherits the deadline $d_i(t)$ of the corresponding task τ_i. At the second reduction level, the deadline of S_1^2 is equal to $d_1(t)$ while the deadline of S_2^2 is $d_3(t)$ and S_3^2 has the same deadline than τ_5. Because a dual server has the same deadline than the corresponding primal server, if we execute

> *EDF on the set of dual server at the last reduction level, the dual server S_1^{2*} is chosen to be executed at time t (see Figure 5.6). It therefore means that both S_2^2 and S_3^2 should be running in the primal schedule of the second reduction level. Applying EDF in each of these servers, we get that S_3^{1*} and S_5^{1*} must be running in the dual schedule of the first reduction level. Therefore, S_3^1 and S_5^1 must stay idle while S_1^1, S_2^1, S_4^1 must be executed in the primal schedule of the first reduction level. Consequently, applying EDF in each of these servers, it results that τ_1, τ_2 and τ_4 must execute on the platform at time t.*

It was proven in [Regnier et al. 2011; Regnier 2012] that the online part of RUN has a complexity of $O(jn\log(m))$ when there are a total of j jobs released by n tasks on m processors within any interval of time.

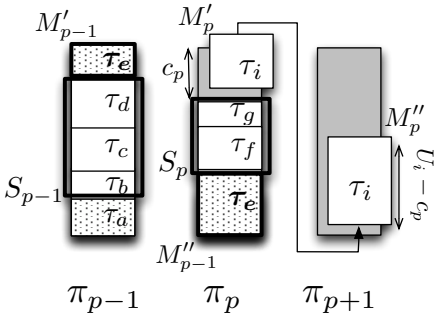
Note that RUN is one of the two only algorithms that do not make use of the fairness approach to reach the optimality. The only other "unfair" optimal algorithm being U-EDF; an algorithm we proposed in [Nelissen et al. 2011a, 2012b] and which will be presented in detail in Chapter 6. However, unlike U-EDF, we recall that RUN is optimal only for the scheduling of periodic tasks with implicit deadlines.

5.2.4 The EKG Algorithm

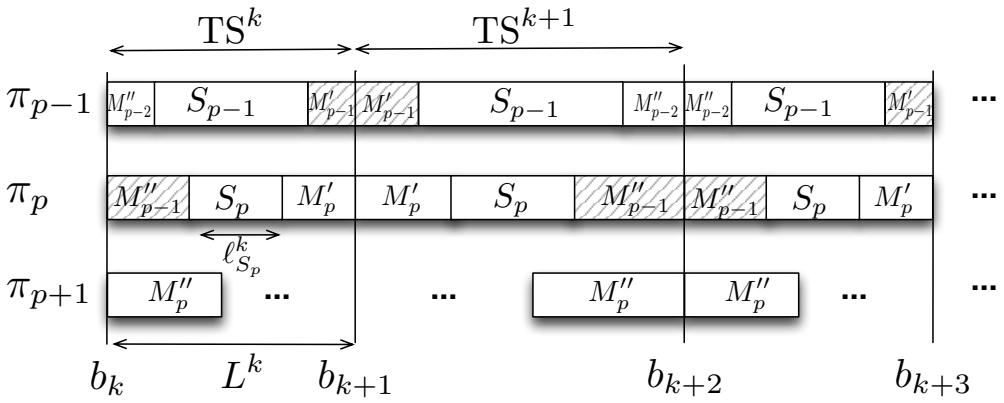
EKG is an optimal algorithm for the scheduling of periodic tasks with implicit deadlines. It was proposed by Andersson and Tovar in [Andersson and Tovar 2006]. EKG is the short-hand notation for ***E**DF with task splitting and **k** processors in a **G**roup*. As its name implies, it splits the computing platform into several clusters through the definition of a parameter k (with $k \leq m$). The platform is divided in $\left\lfloor \frac{m}{k} \right\rfloor$ clusters containing k processors and one cluster composed of $\left(m - \left\lfloor \frac{m}{k} \right\rfloor \times k\right)$ processors. A bin-packing algorithm[1] is then used to partition the tasks amongst the clusters so that the total utilization on each cluster is not greater than its number of constituting processors. After the partitioning of τ amongst the clusters, EKG works in two different phases:

- First, the tasks of each cluster are assigned to the processors of this cluster.
- Then, the tasks are scheduled in accordance with this assignment, using a hierarchical scheduling algorithm based on both the DP-Fairness and EDF.

[1] In the original version of EKG, a *next fit* heuristic was used to partition the task set [Andersson and Tovar 2006]. However, any *reasonable* bin packing algorithm such as *first fit*, *worst fit* or *best fit*, does work [Qi et al. 2010, 2011].



(a) Assignment



(b) Schedule with mirroring technique

Figure 5.7: Assignment and schedule produced by EKG.

It was proven in [Andersson and Tovar 2006] that EKG ensures a utilization bound of $(\frac{k}{k+1} \times m)$ when $k < m$, and is optimal for the scheduling of periodic tasks with implicit deadlines when $k = m$.

Also, in order to minimize the number of preemptions and migrations, if $k < m$ then every task τ_i with a utilization U_i not smaller than $\frac{k}{k+1}$ is scheduled independently of the other tasks on its own processor.

In the remainder of this chapter, we will assume that there is only one cluster in the system, i.e., $k = m$. However, if there should be multiple clusters (i.e., $k < m$), the same reasoning could be applied on each individual cluster without any variation.

We now present the assignment and scheduling phases that EKG utilizes in each individual cluster.

Phase 1. The assignment follows the same variation of the *next fit* heuristic inspired by McNaughton's algorithm which is utilized in DP-Wrap (see Section 5.2.2). That is, tasks are assigned to a processor π_j as long as the utilization capacity on this processor is not exhausted. It then starts assigning tasks on the next processor π_{j+1}. Also, whenever the assignment of a task on π_j would cause the utilization of π_j to exceed 1, the task is "split" between π_j and the next processor π_{j+1}.

Formally, let c_j denote the remaining capacity on processor π_j. This capacity represents the proportion of time during which the processor π_j will remain idle in average during the schedule. Hence, if we assign a task τ_i with an utilization U_i to the processor π_j, its capacity c_j will decrease by an amount U_i.
Let $u_{i,j}$ denote the part of τ_i's utilization assigned on processor π_j. We define S_j as the set of tasks entirely assigned on π_j. That is,

$$S_j \stackrel{\text{def}}{=} \left\{ \tau_i \in \tau \mid u_{i,j} = U_i \right\}.$$

If we assume that τ_i is the next task to assign, and π_p is the first processor with some remaining capacity (i.e., $c_p > 0$ and $c_j = 0$ for all $j < p$), if $c_p \geq U_i$ then $S_p = S_p \cup \{\tau_i\}$. Otherwise, τ_i becomes a *migratory task*, denoted M_p, which is split into two subtasks M'_p with a utilization $U_{M'_p} = c_p$, and M''_p with a utilization $U_{M''_p} = U_i - c_p$ (see Figure 5.7(a)). M'_p is assigned to the processor π_p and M''_p is assigned to π_{p+1}. Hence, the utilization of π_p is exactly 1.

For instance, in Figure 5.7(a), $S_{p-1} = \{\tau_b, \tau_c, \tau_d\}$ and τ_e is split in M'_{p-1} and M''_{p-1}.

Note that there is at most one migratory task assigned to each couple of processors π_j and π_{j+1}, and therefore only two different migratory tasks are executed on each processor.

The set of tasks S_j is henceforth called the *supertask* of processor π_j and the tasks in S_j are referred to as the *component tasks* of S_j or the *non-migratory tasks* of π_j. The utilization of S_j is denoted $U_{S_j} \stackrel{\text{def}}{=} \sum_{\tau_i \in S_j} U_i$.

Phase 2. After the assignment, EKG schedules the migratory tasks and supertasks using a *Deadline Partitioning Fair* (DP-Fair) technique. That is, the time is divided in slices bounded by two successive job deadlines and each migratory task and supertask is executed for a time proportional to its utilization within each time slice. Hence, in each time slice of length L^k and on each processor π_j, the supertask S_j and the subtasks M'_j and

τ_i	Task belonging to the task set τ.
S_j	Set of non-migrating tasks assigned to processor π_j.
M_j	Task migrating between processor π_j and π_{j+1}.
M'_j	Portion of the migratory task M_j assigned to processor π_j.
M''_j	Portion of the migratory task M_j assigned to processor π_{j+1}.
$M^{\text{in}}_{j,k}$	Migrate-in task of processor π_j in TS^k, i.e., *first* task (or portion of task) executed on π_j in TS^k. In EKG, the migrate-in task $M^{\text{in}}_{j,k}$ is either M''_{j-1} or M'_j.
$M^{\text{out}}_{j,k}$	Migrate-out task of processor π_j in TS^k, i.e., *last* task (or portion of task) executed on π_j in TS^k. In EKG, the migrate-out task $M^{\text{out}}_{j,k}$ is either M''_{j-1} or M'_j.

Table 5.1: Notations used in in Chapter 5 to refer to tasks in EKG.

M''_{j-1} execute for $\ell^k_{S_j} = U_{S_j} \times L^k$, $\ell^k_{M'_j} = U_{M'_j} \times L^k$ and $\ell^k_{M''_{j-1}} = U_{M''_{j-1}} \times L^k$ time units, respectively. We say that $\ell^k_{S_j}$ is the *local execution time* of the supertask S_j in time slice TS^k (and similarly for the migratory subtasks M'_j and M''_{j-1}). Figure 5.7 shows an example of the schedule produced by EKG.

Whenever, a supertask S_j is scheduled on processor π_j, its component task with the earliest deadline is actually executed on π_j. EKG can therefore be seen as a hierarchical scheduling algorithm. On the one hand, it uses a DP-Fair approach to schedule the supertasks and, on the other hand, it uses EDF to decide which component task to execute on the processors.

Notice that on Figure 5.7, in each time slice, the schedule produced by EKG is the "mirror" version of the schedule in the previous time slice. This *mirroring* technique proposed together with EKG in [Andersson and Tovar 2006] and already introduced in Section 5.2.2, reduces the number of preemptions and migrations by keeping running the same task on each processor after a boundary.

Due to this mirroring mechanism, the first task executing on each processor π_j in each time slice TS^k is not always the same. It alternates between the migratory subtasks M''_{j-1} and M'_j (see Figure 5.7). In the remainder of this paper, we will refer to the first task scheduled to execute in the time slice TS^k on processor π_j as the *migrate-in task* (denoted by $M^{\text{in}}_{j,k}$). Hence, the migrate-in task $M^{\text{in}}_{j,k}$ of processor π_j refers to the subtask M''_{j-1} when we consider odd numbered time slices and refer to M'_j otherwise. Similarly, the last task executed on processor π_j in time slice TS^k is referred to as the *migrate-out task* (denoted by $M^{\text{out}}_{j,k}$) and alternates between M'_j and M''_{j-1} either.

A summary of all these notations is provided in Table 5.1.

5.3 System Model

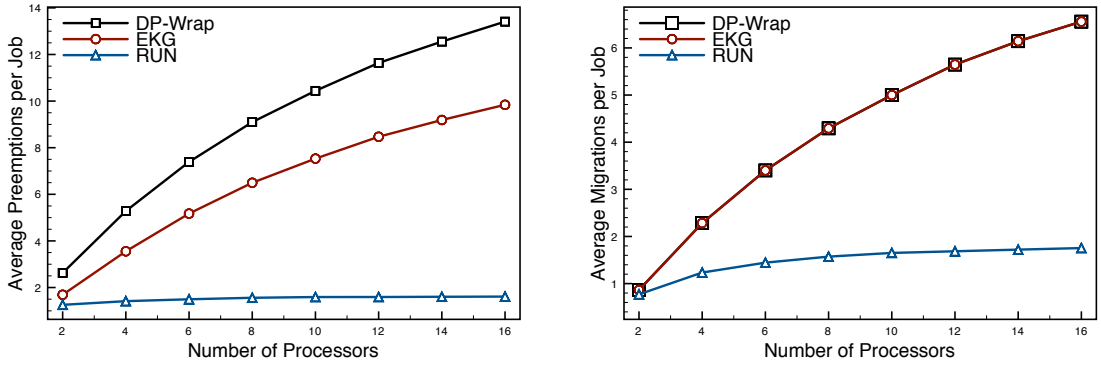
Differently from the previous chapter, in this chapter, we consider the scheduling of a set τ composed of *n periodic* tasks with *implicit deadlines* on *m identical* processors. Each task τ_i in the set τ is characterized by a period T_i and a worst case execution time C_i.

Since we are working with periodic tasks with implicit deadlines, each task has one (and only one) active job at any time t. We can therefore write without ambiguity that the deadline $d_i(t)$ of the task τ_i at time t is the deadline of the current active job of τ_i at time t. Similarly, we denote the arrival time of the current active job of τ_i at time t as $a_i(t)$ and its remaining execution time at time t is given by $\text{ret}_i(t)$.

In the remainder of this chapter, we will assume that, at any instant t, the set τ is ordered according to the deadlines of the tasks at time t. That is, for any two tasks τ_x and τ_y, if $x < y$ then $d_x(t) \leq d_y(t)$. Furthermore, for theoretical reasons, we assume that the platform is fully utilized, i.e., we have a total utilization $U = m$. However, if this last assumption should not be respected in the actual application, we can at least assume that $\sum_{\tau_i \in \tau} U_i \geq m - 1$ (otherwise some processors are useless and can be removed from the platform, or the processing platform can be divided into clusters as presented in Section 5.2.4). Then, we simply model the processors idle time adding an "idle" task τ_{idle} such that $U_{idle} = m - U$ and $T_{idle} = +\infty$. That is, we never miss τ_{idle}'s deadline and it always has the smallest priority regarding to the EDF algorithm. Since this additional task τ_{idle} represents the idle time of the processors, whenever the task τ_{idle} is scheduled to be executed on a processor in the theoretical schedule, this means that the processor actually remains idle in the real schedule. Again, τ_{idle} is only introduced for theoretical reasons but does not impact the resulting schedule produced by EKG.

5.4 Reducing Task Preemptions and Migrations in EKG

Figure 5.8 provides a comparison between the average number of preemptions and migrations per job (i.e., total number of preemptions/migrations divided by the number of jobs released during the schedule) incurred by DP-Wrap, EKG and RUN (LLREF has been omitted because of its really poor performances when comparing to the three other algorithms). These results were computed with the simulation environment described in Section 5.6. We can see that EKG performs better than DP-Wrap in terms of preemptions but has the exact same number of migrations. On the other hand, RUN drastically im-



(a) (b)

Figure 5.8: Average number of preemptions and migrations per released job for DP-Wrap, EKG and RUN on a varying number of processors when the platform is fully utilized (i.e., $U = m$).

proves both the number of preemptions and migrations. Our goal is therefore to identify and address the causes of the excess of preemptions and migrations in EKG in order to bring its performances closer to, or even outperform, those of RUN.

As shown on Figure 5.7, for each time slice in EKG, there is one migration between the processors π_j and π_{j+1} for each migratory task M_j. Furthermore, each migratory task M_j causes up to two preemptions (one for M'_j and another for M''_j) per time slice. On the other hand, because EDF is used to schedule the component tasks of any supertask S_j, the tasks included in S_j do not cause any preemption between two successive job arrivals (which correspond to the time slices boundaries) [Andersson and Tovar 2006]. Additionally, a supertask S_j never migrates. Hence, the number of preemptions and migrations is proportional to the number of time slices. Moreover, most of the preemptions and all the migrations are caused by migratory tasks. Therefore, the number of preemptions and migrations can be reduced by two means:

1. reduce the number of time slices by skipping some boundaries;

2. remove the execution of the migratory tasks (or subtasks) from as many time slices as possible. This can be done by exchanging execution time between tasks and time slices.

The second objective is formalized in Section 5.4.1 and the first one in Section 5.4.2. Then, a combination of both solutions is investigated in Section 5.4.3. Note that we will skip a deadline or perform a swap of execution time between tasks only if it does not

jeopardize the correctness of the schedule previously built by EKG (i.e., all job deadlines are still respected). Hence, our boundary skipping and execution swapping algorithms do not affect the optimality of EKG.

5.4.1 Swapping Execution Times between Tasks

The swapping algorithm is applied at each time t corresponding to a boundary b_k (i.e., at the beginning of each time slice TS^k). We consider the current time slice TS^c as the time interval extending from the current boundary $b_c \stackrel{\text{def}}{=} t$ to the next boundary $b_{c+1} \stackrel{\text{def}}{=} d_1(t)$ (remember that the tasks are sorted in an increasing deadline order).

The algorithm follows four different rules:

Rule 5.1

Maximize the execution of M''_{j-1} on processor π_j in the current time slice TS^c.

Rule 5.2

Minimize the execution of M'_j on processor π_j in the current time slice TS^c.

Rule 5.3

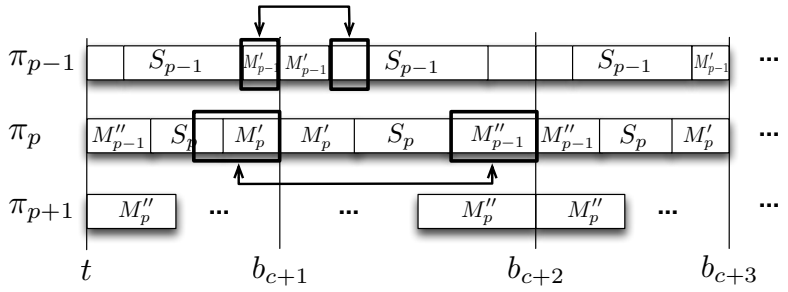
Avoid intra-job parallelism.

Rule 5.4

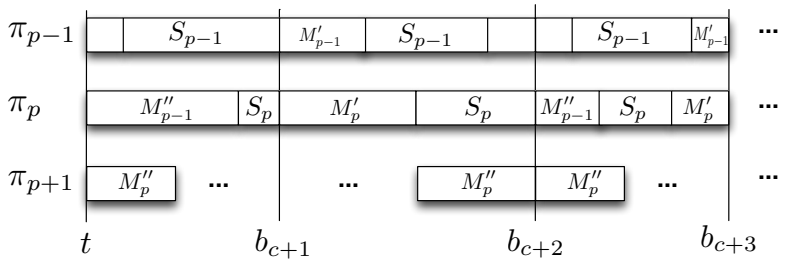
Ensure that every task will be able to complete its execution before its deadline.

The main idea of Rule 5.1 is to complete the execution of M''_{j-1} as early as possible, while on the other hand, Rule 5.2 aims at delaying the execution of M'_j as much as possible. The goal is that M''_{j-1} only appears in the earlier time slices after each of its new job arrival, and to schedule M'_j in the later time slices just prior to each of its job deadlines. Ideally, their executions can be eliminated altogether in some time slices (thereby eliminating the associated preemptions and migrations). Furthermore, reducing M'_j's execution in the current time slice increases the interval in which M''_{j-1} can execute in TS^c giving the opportunity to M''_{j-1} to complete early (and therefore not appear in next time slices anymore).

Assuming that a task set τ is schedulable under EKG, Rules 5.3 and 5.4 make sure that the schedule remains correct after executing the swapping algorithm (i.e., all deadlines



(a) Schedule before swaps



(b) Resulting schedule after swaps

Figure 5.9: Two swaps of execution time between time slices TS^c and TS^{c+1}.

are respected without intra-job parallelism). That is, we do not perform a swap if it could impact the correctness of τ's schedule.

Figure 5.9 shows how the schedule might change after swapping. In this example, both the number of preemptions and migrations have been reduced by 2 in the first time slice. By reapplying the swapping algorithm at each boundary b_k, the number of preemptions and migrations can be drastically reduced on the entire schedule.

Principles Ensuring Correctness

Before to formally describe our swapping algorithm in the next section, we first present some principles that we must follow to ensure that we never violate Rule 5.4.

Principle 5.1

For any migratory task M_j, we cannot swap execution time between TS^c *and any time slice* subsequent *to the deadline of the current active job of the migratory task (i.e., a time slice* TS^k *such that $b_k \geq d_{M_j}(t)$). Indeed, the time reserved for M_j after $d_{M_j}(t)$ is dedicated to the execution of "future" jobs of M_j that are not yet active in* TS^c.

Principle 5.2

If we increase (respectively decrease) the local execution time ℓ_i^c of a task τ_i by Δ_i time units in the current time slice TS^c, we have to decrease (respectively increase) the local execution time ℓ_i^k by the same quantity Δ_i in another time slice TS^k such that $k > c$ and TS^k is before $d_i(t)$ (according to Principle 5.1). That is, the time reserved for the execution of a task must remain constant for the whole schedule.

Principles 5.1 and 5.2 give us a straightforward approach for enforcing all migratory tasks adherence to Rule 5.4. However, supertasks require more thought. Indeed, a supertask S_j is a collection of tasks. Hence, each component task τ_i has its own deadline $d_i(t)$ and its own remaining execution time $\mathrm{ret}_i(t)$. We must therefore ensure that S_j has enough time reserved on π_j to fulfill the execution of *all* its component tasks before their respective deadlines. Moreover, if $d_i(t)$ is the deadline of the current job of a component task τ_i of S_j, then a part of the time reserved for S_j in the time slices *subsequent* to $d_i(t)$ is dedicated to the execution of "future" jobs of τ_i. These jobs are not yet active within the interval $[t, d_i(t))$. Therefore, we must be sure that we keep enough time available after $d_i(t)$ to execute these future jobs.

Let $F_j^k(t)$ be the tasks in S_j with a deadline no later than b_k, i.e.,

$$F_j^k(t) \stackrel{\text{def}}{=} \left\{ \tau_i \in S_j \mid d_i(t) \leq b_k \right\}$$

The time reserved in TS^k (i.e., the time slice extending from b_k to b_{k+1}) for any task τ_i in $F_j^k(t)$, is dedicated to future jobs of τ_i (because τ_i's current job has its deadline no later than b_k). Hence, the local execution time $\ell_{S_j}^k$ allocated to the supertask S_j within the time slice TS^k can be divided into two parts – namely, the time reserved for the future jobs (i.e., jobs of tasks in $F_j^k(t)$), denoted $f_{S_j}^k(t)$, and the time allotted to the execution of active jobs at time t (i.e., jobs of tasks in $S_j \setminus F_j^k(t)$), denoted $a_{S_j}^k(t)$. That is, we have $\ell_{S_j}^k = a_{S_j}^k(t) + f_{S_j}^k(t)$. Since, by definition, future jobs have not yet arrived at time t, we cannot swap execution time with these jobs. Therefore, the value of $f_{S_j}^k(t)$ can never be modified. When using a DP-Fair approach such as EKG, $f_{S_j}^k(t)$ is equal to the utilizations of the tasks in $F_j^k(t)$ multiplied by the length of the time slice [Funk et al. 2011; Andersson and Tovar 2006]. That is,

$$f_{S_j}^k(t) \stackrel{\text{def}}{=} L^k \times \sum_{\tau_i \in F_j^k(t)} U_i$$

We must therefore respect the following two principles:

Principle 5.3

In any time slice TS^k*, we cannot decrease the local execution time of a supertask* S_j *by more than* $a^k_{S_j}(t) \stackrel{\text{def}}{=} \ell^k_{S_j} - f^k_{S_j}(t)$.

Principle 5.4

Assuming that b_{q+1} *is a boundary subsequent to* t *(i.e.* $b_{q+1} \geq b_{c+1}$*), then, to respect Rule 5.4 for a supertask* S_j*, we must ensure that the remaining execution at time* t *of the active jobs in* S_j *with a deadline before or at* b_{q+1}*, never exceeds the time reserved for those jobs in the interval* $[t, b_{q+1})$*. Specifically, for each* $q \geq c$*,*

$$\sum_{v=c}^{q} a^v_{S_j}(t) \geq \sum_{\substack{\tau_i \in S_j \ \wedge \\ d_i(t) \leq b_{q+1}}} \mathrm{ret}_i(t). \tag{5.2}$$

Note that Expression 5.2 can be rewritten as

$$a^c_{S_j}(t) \geq \sum_{\substack{\tau_i \in S_j \ \wedge \\ d_i(t) \leq b_{q+1}}} \mathrm{ret}_i(t) - \sum_{v=c+1}^{q} a^v_{S_j}(t)$$

Hence, after a swap between TS^c and TS^k, Principle 5.4 requires that the local execution time $\ell^c_{S_j}$ of S_j in TS^c to be large enough to ensure, for each TS^q $(q < k)$, that there is enough time allocated during TS^c to TS^q to meet the demand. That is,

$$\ell^c_{S_j} \geq \bar{a}^c_{S_j}(t) \stackrel{\text{def}}{=} \max_{c \leq q < k} \Big(\sum_{\substack{\tau_i \in S_j \ \wedge \\ d_i(t) \leq b_{q+1}}} r_i(t) - \sum_{v=c+1}^{q} a^v_{S_j}(t) \Big) \tag{5.3}$$

where $\bar{a}^c_{S_j}(t)$ is the minimum time S_j must execute in TS^c to respect Expression 5.2 in each time slice TS^q with $c \leq q < k$.

Algorithm 5.1: Swapping algorithm.

```
1  forall the TS^k : k := c+1 → c+η do
2      for j := m to 1 do
3          Swap between ℓ^c_{M'_j} and ℓ^k_{M'_j} using Lemma 5.2;
4          Swap between ℓ^c_{M''_{j-1}} and ℓ^k_{M''_{j-1}} using Lemma 5.3;
5      end
6      for j := 1 to m do
7          if There is parallelism between M'_j and M''_j then
8              Correct ℓ^c_{M'_j} and ℓ^c_{M''_j} values using (5.4);
9          end
10         Update ℓ^c_{S_j} and ℓ^k_{S_j} using (5.5);
11     end
12 end
```

Algorithm Description

The swapping algorithm works as follows; initially, EKG is used to assign the tasks amongst the processors and compute the local execution time of every migratory task and supertask in each time slice bounded by the job deadlines. Then, at $t = 0$ and at any time t corresponding to a time slice boundary, our swapping algorithm is executed on a depth of η time slices as shown in Algorithm 5.1. That is, Algorithm 5.1 browses the η time slices TS^k subsequent to b_{c+1} and updates the local execution times by performing swaps (if possible) between TS^k and the current time slice TS^c in accordance with Rules 5.1 to 5.4. The value of η is chosen at design time and must be greater than 0. The impact of this parameter on the number of preemptions and migrations will further be discussed in Section 5.6.

The swapped quantities are computed starting with processor π_m and ending with processor π_1. On each processor π_j ($m \geq j \geq 1$), we first perform the swap for the subtask M'_j and then for the subtask M''_{j-1} using Lemmas 5.2 and 5.3 presented later (lines 2 to 5 in Algorithm 5.1). However, since the swapping decision on processor π_j is taken without a complete knowledge of the swaps that will be performed on processors π_1 to π_{j-1}, Rule 5.3 is partially ignored during this first swapping pass and some intra-job parallelism could arise. Therefore, with a second pass, we must correct the values of the local execution times on each processor to remove the parallelism and hence respect Rule 5.3 (lines 7 to 9). Finally, the local execution time of the supertask S_j is updated (line 10).

We now derive the maximum execution time that can be swapped between TS^c and TS^k for the migratory tasks on processor π_j. According to Rules 5.1 and 5.2, we will increase the local execution time $\ell^c_{M''_{j-1}}$ of M''_{j-1} by $\Delta^k_{M''_{j-1}}$ in TS^c and decrease $\ell^c_{M'_j}$ by $\Delta^k_{M'_j}$. Moreover, as stated in Principle 5.2, in order to respect Rule 5.4, we must correspondingly decrease $\ell^k_{M''_{j-1}}$ by $\Delta^k_{M''_{j-1}}$ and increase $\ell^k_{M'_j}$ by $\Delta^k_{M'_j}$ in TS^k.

Lemma 5.2

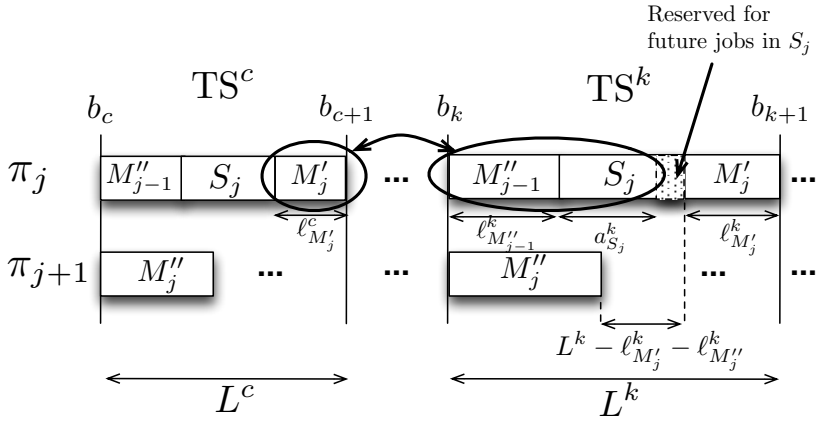
The maximum execution time of the migratory task M'_j that can be swapped from time slice TS^c to time slice TS^k is given by

$$\Delta^k_{M'_j} = \begin{cases} \min\left\{\ell^c_{M'_j},\ a^k_{S_j}(t) + \ell^k_{M''_{j-1}},\ L^k - \ell^k_{M'_j} - \ell^k_{M''_j}\right\} & \text{if } d_{M_{j-1}}(t) \geq b_{k+1} \wedge \\ & \quad d_{M_j}(t) \geq b_{k+1} \\ \min\left\{\ell^c_{M'_j},\ a^k_{S_j}(t),\ L^k - \ell^k_{M'_j} - \ell^k_{M''_j}\right\} & \text{if } d_{M_{j-1}}(t) < b_{k+1} \wedge \\ & \quad d_{M_j}(t) \geq b_{k+1} \\ 0 & \text{otherwise} \end{cases}$$

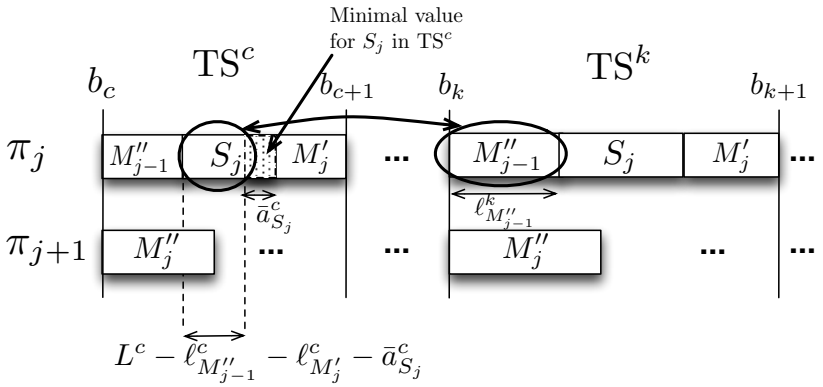
Proof:

We cannot swap more from TS^c than what is allocated to M'_j. Hence, $\Delta^k_{M'_j} \leq \ell^c_{M'_j}$. Similarly, increasing M'_j's allocation in TS^k will decrease the time allocated to M''_{j-1} and S_j. Therefore, we cannot swap more into TS^k than what is allocated to active jobs of M''_{j-1} and S_j (see Principles 5.1 and 5.3). That is, $\Delta^k_{M'_j} \leq a^k_{S_j}(t) + \ell^k_{M''_{j-1}}$ if $d_{M_{j-1}}(t) \geq b_{k+1}$ (see Figure 5.10 (a)) and $\Delta^k_{M'_j} \leq a^k_{S_j}(t)$ if the time reserved for M_{j-1} in TS^k is for a job which is not yet active at time $t = b_c$ (i.e., if $d_{M_{j-1}}(t) < b_{k+1}$). Furthermore, from Rule 5.3, we forbid any intra-job parallelism. Therefore, because $\ell^k_{M''_j}$ has already been computed and will remain fixed (because swaps are performed from π_m to π_1 according to Algorithm 5.1), we must have $\Delta^k_{M'_j} \leq L^k - \ell^k_{M'_j} - \ell^k_{M''_j}$ (recall that M'_j and M''_j belong to the same migratory task M_j).

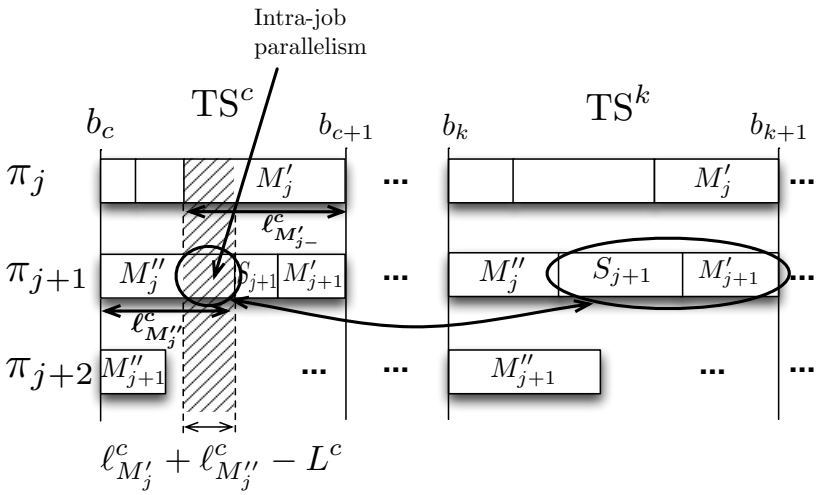
Finally, we stated in Principle 5.1 that we cannot swap any time from M_j between TS^c and a time slice subsequent to $d_{M_j}(t)$. That is, $\Delta^k_{M'_j} = 0$ for such time slices. ■



(a)



(b)



(c)

Figure 5.10: Task swapping according to Algorithm 5.1.

Lemma 5.3

The maximum execution time of the migratory task M''_{j-1} that can be swapped from time slice TS^k to time slice TS^c is given by

$$\Delta^k_{M''_{j-1}} = \begin{cases} \min\left\{ \ell^k_{M''_{j-1}} \ , \ \left(L^c - \ell^c_{M''_{j-1}} - \ell^c_{M'_j} - \bar{a}^c_{S_j}(t) \right) \right\} & \text{if } d_{M_{j-1}}(t) \geq b_{k+1} \\ 0 & \text{otherwise} \end{cases}$$

Proof:

We cannot swap more from TS^k than what is allocated to M''_{j-1}. Therefore, $\Delta^k_{M''_{j-1}} \leq \ell^k_{M''_{j-1}}$. Similarly, we cannot swap more from TS^c than what must not mandatorily be executed in TS^c. Since M'_j and S_j must execute for $\ell^c_{M'_j}$ (determined at line 3 of Algorithm 5.1) and $\bar{a}^c_{S_j}$ (from Expression 5.3) time units in time slice TS^c respectively, and because M''_{j-1} already has $\ell^c_{M''_{j-1}}$ time units allocated in TS^c, that leaves $\Delta^k_{M''_{j-1}} \leq \left(L^c - \ell^c_{M''_{j-1}} - \ell^c_{M'_j} - \bar{a}^c_{S_j}(t) \right)$ (see Figure 5.10 (b)).

Finally, we stated in Principle 5.1 that we cannot swap any time from M_{j-1} between TS^c and a time slice subsequent to $d_{M_{j-1}}(t)$. That is, $\Delta^k_{M''_{j-1}} = 0$ for such time slices. ■

Note that Lemma 5.3 uses the updated values of the local execution times of S_j, M''_{j-1} and M'_j in TS^c and TS^k after the application of Lemma 5.2 at line 3 of Algorithm 5.1.

In case of intra-job parallelism between M'_j and M''_j (recall that they both belong to the migratory task M_j), we can correct the values of the local execution times by adjusting the $\Delta^k_{M''_j}$ and $\Delta^k_{M'_{j+1}}$ quantities (see Figure 5.10 (c)). In this situation, we decrease $\Delta^k_{M''_j}$ so that task M'_j starts to execute at the exact moment when M''_j finishes on processor π_j (i.e., M_j executes for exactly L^c time units during TS^c). This frees up some time which can be spent either executing S_{j+1} or M'_{j+1}. By Rule 5.2, our preference would be to avoid increasing M'_{j+1}. However, Principle 3 states that we cannot decrease the execution of S_{j+1} in TS^k for more than $a^k_{S_{j+1}}$ time units. Therefore, if $a^k_{S_{j+1}}$ is too small then we may need to increase M'_{j+1} in TS^c. This gives the following adjustments to $\Delta^k_{M''_j}$ and $\Delta^k_{M'_{j+1}}$.

$$\begin{cases} \Delta^k_{M''_j} \leftarrow \Delta^k_{M''_j} - \left(\ell^c_{M'_j} + \ell^c_{M''_j} - L^c \right) \\ \Delta^k_{M'_{j+1}} \leftarrow \Delta^k_{M'_{j+1}} - \max\left\{ 0 \ , \ \ell^c_{M'_j} + \ell^c_{M''_j} - L^c - a^k_{S_{j+1}}(t) \right\} \end{cases} \tag{5.4}$$

Whenever we have computed new values for the local execution times of tasks M''_{j-1} and M'_j on processor π_j, we must update $\ell^c_{S_j}$ and $\ell^k_{S_j}$ so that the total execution time remains constant in each time slice. We therefore get

$$\begin{cases} \ell^c_{S_j} \leftarrow \ell^c_{S_j} + \Delta_{M'_j} - \Delta_{M''_{j-1}} \\ \ell^k_{S_j} \leftarrow \ell^k_{S_j} - \Delta_{M'_j} + \Delta_{M''_{j-1}} \end{cases} \tag{5.5}$$

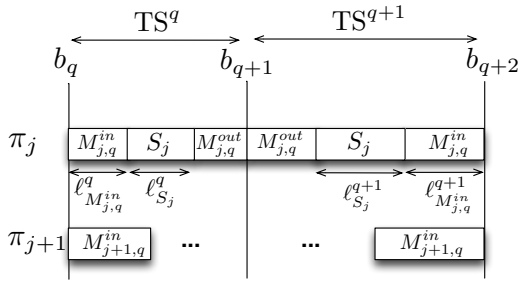
5.4.2 Skipping Boundaries

As previously explained, the EKG algorithm causes up to two preemptions and one migration per processor in each time slice. Therefore, a straightforward solution improving the number of preemptions and migrations during the schedule would be to reduce the number of time slices by suppressing some time slice boundaries. Clearly, we cannot remove all such boundaries – it is the imposition of these boundaries that makes EKG an optimal algorithm[2]. In this section, we explore how to determine which boundaries can be skipped without jeopardizing the schedulability.

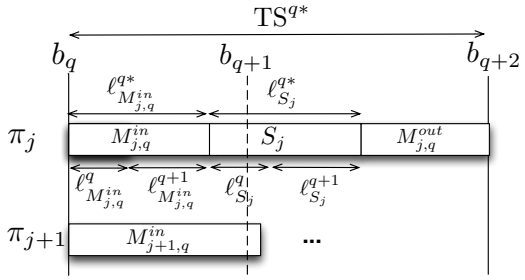
A Set of Rules to Safely Suppress a Boundary

Suppose that we are at time $t = b_q$ and we remove the boundary b_{q+1}. Hence, we are scheduling in the interval $[b_q, b_{q+2})$ rather than $[b_q, b_{q+1})$ (see Figure 5.11). We call this extended time slice TS^{q*}. If we are to maintain a correct schedule, we cannot reduce the amount of work the migratory tasks complete during this extended interval. Additionally, we still want to execute S_j between $M^{\text{in}}_{j,q}$ and $M^{\text{out}}_{j,q}$ on each processor π_j. Otherwise, suppressing the boundary will not have the benefit of reducing the number of preemptions. Hence, we can view the boundary suppression as the merging of two time slices. While we would normally have $M^{\text{out}}_{j,q}$ executing before b_{q+1} and $M^{\text{in}}_{j,q}$ executing after b_{q+1}, these execution time shares are shifted so that the migratory tasks execute contiguously within the merged time slice as illustrated in Figures 5.11(a) and (b). We note that because of this shifting, the merged time slice will not have any intra-job parallelism.

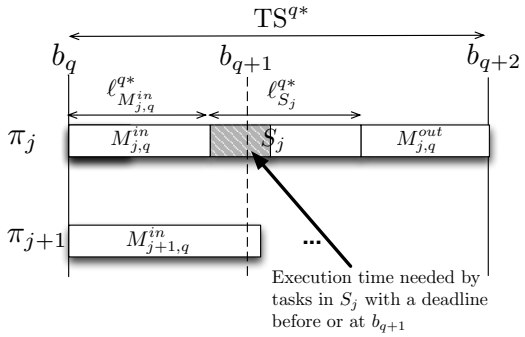
[2]In fact, at the exception of RUN [Regnier et al. 2011; Regnier 2012], all the optimal multiprocessor scheduling algorithms use some sort of time slice mechanism to achieve optimality.



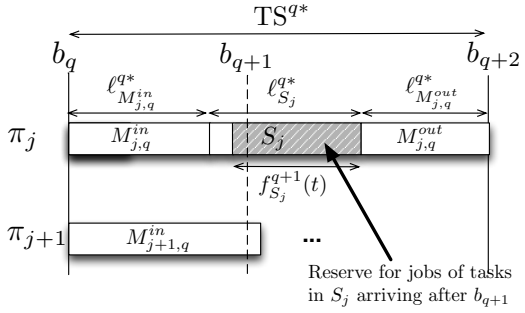
(a) Before the boundary suppression



(b) After the boundary suppression



(c) Invalid boundary suppression according to Lemma 5.5



(d) Invalid boundary suppression according to Lemma 5.6

Figure 5.11: Suppressing a boundary and merging the time slices.

Whenever we consider to skip a boundary, we must be concerned by the impact that the shifting of the execution time shares could have on the correctness of the produced schedule. Hence, we must verify that we still respect the two following rules after the boundary suppression:

Rule 5.5

Never relegate the execution of a job after its deadline.

Rule 5.6

Never schedule the execution of a job before its release time.

We first consider what would happen if either $M_{j,q}^{\text{in}}$ or $M_{j,q}^{\text{out}}$ had a deadline at time b_{q+1}. Since $\ell_{M_{j,q}^{\text{in}}}^{q+1}$ is initially scheduled to be executed after b_{q+1}, it means that this time is reserved for the execution of the next job of $M_{j,q}^{\text{in}}$, and because we shift $\ell_{M_{j,q}^{\text{in}}}^{q+1}$ backward in time when we merge TS^q and TS^{q+1} (see Figure 5.11(a) and (b)), we would end up trying to execute the newly arrived job before it has arrived. Similarly, since we shift $\ell_{M_{j,q}^{\text{out}}}^{q}$ forward in time, this would result in $M_{j,q}^{\text{out}}$ missing its deadline. Therefore, both situations break one of the two Rules 5.5 and 5.6. Hence, this gives us our first principle regarding the suppression of time slice boundaries:

Lemma 5.4

A time slice boundary b_k cannot be skipped if a job of any migratory task M_j has its deadline at time b_k.

Even for non-migratory tasks, we need to be concerned about the same two issues, i.e., we need to ensure that the boundary suppression will not result in a missed deadline or in the attempt to execute a job before it has arrived.

Let $F_j^{q+1}(t)$ be the set of non-migratory tasks that have a deadline no later than b_{q+1} on processor π_j. Formally,

$$F_j^{q+1}(t) \stackrel{\text{def}}{=} \left\{ \tau_i \in S_j \mid d_i(t) \leq b_{q+1} \right\}$$

As for migratory tasks with Lemma 5.4, to ensure that the tasks in $F_j^{q+1}(t)$ still finish

their executions before their deadlines after the suppression of b_{q+1}, we could simply impose $\text{ret}_i(t)$ to be equal to 0 for all tasks $\tau_i \in F_j^{q+1}(t)$ on every processor π_j. In other words, to be authorized to skip a boundary b_{q+1}, all tasks with a deadline at or before b_{q+1} must have finished their execution. This, however, is more restrictive than necessary. Tasks in $F_j^{q+1}(t)$ can execute within the merged time slice TS^{q*} provided that each task $\tau_i \in F_j^{q+1}(t)$ executes for exactly $\text{ret}_i(t)$ time units during the interval $[b_q, b_{q+1})$. Hence, after the boundary suppression, the local execution time $\ell^{q*}_{M^{\text{in}}_{j,q}}$ allotted to $M^{\text{in}}_{j,q}$ cannot be so large that the tasks in $F_j^{q+1}(t)$ are unable to finish the execution of their current jobs by b_{q+1} (see Figure 5.11(c)). This gives the following constraint:

$$\ell^{q*}_{M^{\text{in}}_{j,q}} + \sum_{\tau_i \in F_j^{q+1}(t)} \text{ret}_i(t) \leq b_{q+1} - b_q$$

where $\ell^{q*}_{M^{\text{in}}_{j,q}} \stackrel{\text{def}}{=} \ell^{q}_{M^{\text{in}}_{j,q}} + \ell^{q+1}_{M^{\text{in}}_{j,q}}$. That is, there is enough time to execute $M^{\text{in}}_{j,q}$ and all tasks in $F_j^{q+1}(t)$ within $[b_q, b_{q+1})$.

Now, let us assume that we want to skip s successive boundaries. Applying the same reasoning, we obtain the following lemma:

Lemma 5.5

Assume that we are at time $t = b_c$. Let $k \stackrel{\text{def}}{=} c + s$. We can skip the s successive boundaries $b_{c+1}, b_{c+2}, ..., b_k$ if and only if for all q such that $c \leq q < c + s$ and all processors π_j, the following inequality is respected

$$\sum_{v=0}^{s} \ell^{c+v}_{M^{\text{in}}_{j,c}} + \sum_{\tau_i \in F_j^{q+1}(t)} \text{ret}_i(t) \leq b_{q+1} - b_c$$

Lemma 5.5 makes certain that Rule 5.5 is respected. Additionally, a second constraint must ensure that no job arriving after b_{q+1} will be scheduled to execute before its arrival time (Rule 5.6).

Let us assume that a task τ_i that has its deadline at time b_{q+1} (i.e., $d_i(t) = b_{q+1}$) is a non-migratory task of processor π_j. That is, τ_i is a component task of S_j. As explained in Section 5.4.1, EKG reserved a time equal to $f_j^{q+1}(t) = U_i \times (b_{q+2} - b_{q+1})$ in the interval

$[b_{q+1}, b_{q+2})$ for the execution of the next jobs of τ_i. Because the local execution time $\ell^q_{M^{\text{out}}_{j,q}}$ is shifted forward in the merged time slice TS^{q*} (see Figure 5.11), it is possible that

$$\ell^q_{M^{\text{out}}_{j,q}} + \ell^{q+1}_{M^{\text{out}}_{j,q}} + f_i^{q+1}(t) > (b_{q+2} - b_{q+1})$$

thereby implying that the future job of τ_i should at least partially execute in $[b_q, b_{q+1})$ (see Figure 5.11(d)). However, this execution time is not available before b_{q+1} (i.e., the arrival time of the new job), and because we assume $U = m$, there can be no idle time in the schedule. Hence, we must respect the following constraint when suppressing the boundary b_{q+1}:

$$\forall \pi_j \ : \ \ell^{q*}_{M^{\text{out}}_{j,q}} + f_{S_j}^{q+1}(t) \leq (b_{q+2} - b_{q+1})$$

where $\ell^{q*}_{M^{\text{out}}_{j,q}} \stackrel{\text{def}}{=} \ell^q_{M^{\text{out}}_{j,q}} + \ell^{q+1}_{M^{\text{out}}_{j,q}}$ and $f_{S_j}^{q+1}(t)$ is the time reserved in TS^{q+1} for future jobs of tasks in S_j with a deadline at or before b_{q+1}, i.e., $f_{S_j}^{q+1}(t) \stackrel{\text{def}}{=} \sum_{\tau_i \in F_j^{q+1}(t)} U_i \times (b_{q+2} - b_{q+1})$.

Now, if we want to skip s successive boundaries, then applying the same argument for each boundary, we get the following lemma:

Lemma 5.6

Assume that we are at time $t = b_c$. Let $k \stackrel{\text{def}}{=} c + s$. We can skip the s successive boundaries $b_{c+1}, b_{c+2}, ..., b_k$ if for all q such that $c \leq q < k$ and all processors π_j, the following inequality is respected

$$\sum_{v=c}^{q+1} \ell^v_{M^{\text{out}}_{j,c}} + f_{S_j}^{q+1} \leq (b_{q+2} - b_c)$$

Algorithm Description

In order to reduce the number of time slices, we try to increase the length of the current time slice TS^c as much as possible. Therefore, we execute Algorithm 5.2 at each time t corresponding to the end of a time slice. Algorithm 5.2 selects the next boundary b_k we must imperatively take into account to meet all job deadlines occurring within the interval $[t, b_k]$, and still respect Rules 5.5 and 5.6. Hence, all boundaries $b_q \in (t, b_k)$ can be ignored

Algorithm 5.2: Selection of the next boundary that must be taken into account.

```
1 forall the b_k > t do
2 |   if Lemma 5.4 or Lemma 5.5 or Lemma 5.6 is not respected then
3 |   |   return b_k;
4 |   end
5 end
```

and the current time slice TS^c therefore extends from t to b_k.

5.4.3 Skipping and Swapping Altogether

The skipping and swapping algorithms are both designed to reduce the number of preemptions and migrations. It is therefore worthwhile to think about adding their effects by using Algorithms 5.1 and 5.2 altogether. Two possibilities must be considered: (i) swapping execution times before skipping boundaries; or (ii) skipping before swapping.

Swapping and then Skipping

The execution of the skipping algorithm (Algorithm 5.2) just after the swapping algorithm (Algorithm 5.1) at each boundary does not impose to take any particular precaution. However, as it will be shown in Section 5.6, it has a negative impact on the resulting number of preemptions and migrations (see Figure 5.13(a) and (d)). This can be explained by the fact that skipping boundaries will cancel the swapping effects by merging the current time slice TS^c with time slices with which some execution time had just been swapped.

To illustrate this last claim, assume that we execute Algorithm 5.1 with a depth $\eta = 1$. This means that we swap execution times only between the current time slice TS^c and the next time slice TS^{c+1}. Now, suppose that we can skip the boundary b_{c+1}. Hence, time slices TS^c and TS^{c+1} can be merged in a single time slice TS^{c*}. Consequently, the local execution times allocated to the tasks in TS^c and TS^{c+1} are executed altogether and the swaps that were performed by Algorithm 5.1 between TS^c and TS^{c+1} have no impact on the total execution time allocated to each task in TS^{c*}. In this case, executing Algorithm 5.2 after Algorithm 5.1 therefore produces the same schedule as if Algorithm 5.1 was not executed at all.

Skipping and then Swapping

Unlike suppressing boundaries after swapping execution times, swapping execution times (Algorithm 5.1) after the execution of the skipping algorithm (Algorithm 5.2) should improve the results obtained with either technique utilized independently. However, we cannot swap execution time after having merged many time slices without taking some precautions. Indeed, through Lemmas 5.5 and 5.6, the boundary suppression algorithm imposes constraints on the repartition of the task execution times. These constraints should be added to those already present in Lemmas 5.2 and 5.3 which are used by the swapping algorithm. Note that Lemmas 5.5 and 5.6 impose their constraints on the migrate-in and migrate-out tasks (i.e., $M_{j,c}^{in}$ and $M_{j,c}^{out}$), while Lemmas 5.2 and 5.3 refer to the subtasks M'_j and M''_{j-1}. Therefore, it must be taken into account that the constraints on $M_{j,c}^{in}$ imposed by the skipping algorithm refer alternatively to the subtasks M'_j and M''_{j-1} (and similarly for $M_{j,c}^{out}$).

Since we first skip boundaries before to swap, we start by executing Algorithm 5.2 as it is presented in Section 5.4.2. Hence, we compute the next boundary $b_k > b_c$ that must be taken into account for the merged time slice TS^{c*} in order to respect all job deadlines. When we skip s successive boundaries, Lemma 5.5 imposes that the local execution time $\ell^{c*}_{M^{\text{in}}_{j,c}} \stackrel{\text{def}}{=} \sum_{v=0}^{s} \ell^{c+v}_{M^{\text{in}}_{j,c}}$ of $M^{\text{in}}_{j,c}$ in the resulting time slice TS^{c*} respects the following expression

$$\ell^{c*}_{M^{\text{in}}_{j,c}} \leq \min_{c \leq q < c+s} \left\{ \left(b_{q+1} - b_c\right) - \sum_{\tau_i \in F_j^{q+1}(t)} \text{ret}_i(t) \right\} \tag{5.6}$$

Similarly, Lemma 5.6 imposes for $\ell^{c*}_{M^{\text{out}}_{j,c}} \stackrel{\text{def}}{=} \sum_{v=0}^{s} \ell^{c+v}_{M^{\text{out}}_{j,c}}$ that

$$\ell^{c*}_{M^{\text{out}}_{j,c}} \leq (b_{c+s+1} - b_c) - f_{S_j}^{c+s} \tag{5.7}$$

Then, when executing the swapping algorithm, two cases must be analyzed:

- if $M^{\text{in}}_{j,c}$ corresponds to M''_{j-1}, then because we intend to increase the local execution time of M''_{j-1} within TS^{c*} (Rule 5.1) and $M''_{j-1} \equiv M^{\text{in}}_{j,c}$, Expression 5.6 adds the following constraint to those already stated in Lemma 5.3

$$\ell^{c*}_{M''_{j-1}} \leq \min_{c \leq q < c+s} \left\{ \left(b_{q+1} - b_c\right) - \sum_{\tau_i \in F_j^{q+1}(t)} \text{ret}_i(t) \right\}$$

- if $M^{\text{in}}_{j,c}$ corresponds to M'_j, then $M^{\text{out}}_{j,c}$ represents M''_{j-1}. For the same reasons than those expressed in the previous case, Expression 5.7 imposes that

$$\ell^{c*}_{M''_{j-1}} \leq L^{c*} - f^{c+s}_{S_j}$$

with $L^{c*} \stackrel{\text{def}}{=} b_{c+s+1} - b_c$

Lemma 5.3 therefore becomes

Lemma 5.7

The maximum execution time of the migratory task M''_{j-1} that can be swapped from time slice TS^k to the merged time slice TS^{c*} is given by

$$\Delta^k_{M''_{j-1}} = \begin{cases} \min\left\{ \ell^k_{M''_{j-1}},\ \left(L^{c*} - \ell^{c*}_{M''_{j-1}} - \ell^{c*}_{M'_j} - \bar{a}^{c*}_{S_j}(t) \right),\ \Delta^{c*}_{\max} \right\} & \text{if } d_{M_{j-1}}(t) \geq b_{k+1} \\ 0 & \text{otherwise} \end{cases}$$

where

$$\Delta^{c*}_{\max} = \begin{cases} \min\limits_{c \leq q < c+s} \left\{ \left(b_{q+1} - b_c\right) - \sum_{\tau_i \in F^{q+1}_j(t)} \text{ret}_i(t) \right\} & \text{if } M''_{j-1} \equiv M^{\text{in}}_{j,c} \\ L^{c*} - f^{c+s}_{S_j} & \text{if } M''_{j-1} \equiv M^{\text{out}}_{j,c} \end{cases}$$

Moreover, because the swapping algorithm tries to reduce the local execution time of M'_j in TS^{c*} (Rule 5.2) and because Expressions 5.6 and 5.7 do not impose a lower bound on $\ell^{c*}_{M^{\text{out}}_{j,c}}$, Lemma 5.2 is not impacted by the boundaries suppression.

As a result, when we first skip boundaries using Algorithm 5.2, the swapping algorithm must be updated and is now given by Algorithm 5.3.

5.4.4 Run-Time and Memory Complexity

By analyzing Algorithms 5.1 and 5.2, we can easily show that their run-time complexities are $O(n)$. Indeed, at each step considering a boundary b_k, Algorithms 5.1 and 5.2 divide the task set τ in two parts – the set of tasks which have a deadline no later then b_k, and the set of tasks with a deadline after b_k. Hence, in the worst case, the deadlines of all tasks must be checked, thereby implying a run-time complexity of $O(n)$. Note that, even though swaps can be carried out with multiple time slices (Algorithm 5.1) and many consecutive

Algorithm 5.3: Swapping algorithm executed after Algorithm 5.2.

```
1  forall the TS^k : k := c* + 1 → c* + η do
2      for j := m to 1 do
3          Swap between ℓ^{c*}_{M'_j} and ℓ^k_{M'_j} using Lemma 5.2;
4          Swap between ℓ^{c*}_{M''_{j-1}} and ℓ^k_{M''_{j-1}} using Lemma 5.7;
5      end
6      for j := 1 to m do
7          if There is parallelism between M'_j and M''_j then
8              Correct ℓ^{c*}_{M'_j} and ℓ^{c*}_{M''_j} values using (5.4);
9          end
10         Update ℓ^{c*}_{S_j} and ℓ^k_{S_j} using (5.5);
11     end
12 end
```

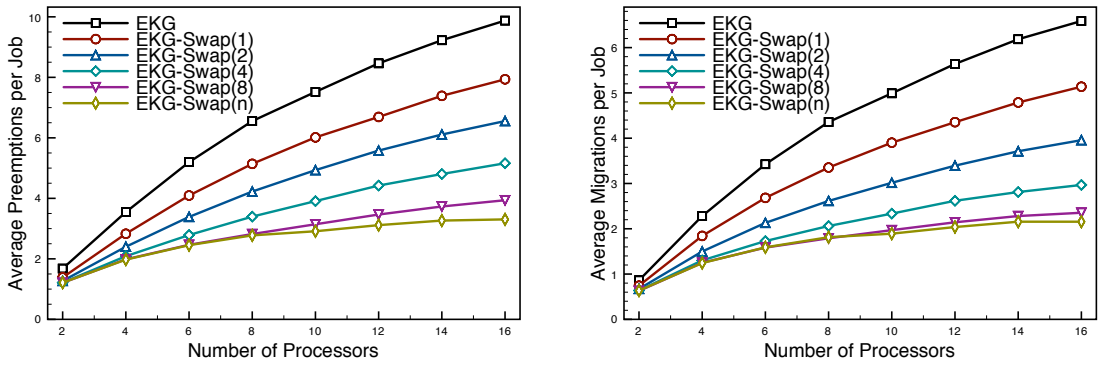
boundaries can be skipped (Algorithm 5.2), if the tasks are sorted in a non-decreasing deadline order, then each task must be checked only once as we consider boundaries (and associated time slices) in an increasing time order.

Moreover, while EKG basically plans the schedule of one time slice at each boundary b_k, the swapping algorithm needs to save the schedule for $\eta + 1$ consecutive time slices to swap execution time between them. Planning the schedule of one time slice TS^k necessitates to save the local execution times of the two migratory tasks and the supertask of each processor for this time slice TS^k. Therefore, $3 \times m$ values must be saved. Consequently, the swapping algorithm makes use of a memory space proportional to $O(m \times \eta)$ whereas the schedule produced by the initial EKG algorithm had a memory complexity of $O(m)$.

5.5 Various Considerations

5.5.1 Instantaneous Migrations

In addition to a reduction of the number of preemptions and migrations, the mirroring technique introduced in Section 5.2.4 also helps avoiding instantaneous migrations of a task from one processor to another, i.e., it avoids that a task stops executing on a processor π_j and directly restarts running on an other processor π_k. Unfortunately, the swapping algorithm re-introduces such instantaneous migrations. Indeed, we assumed in the argumentation of Section 5.4.1 that a migratory task can execute for up to the length L^k of the time slice TS^k in TS^k. However, by simply modifying this assumption to allow a task to



(a) (b)

Figure 5.12: Comparison of the number of preemptions and migrations for various depths of swap η (with $U = m$).

execute for up to $L^k - \varepsilon$ (with $\varepsilon > 0$) in TS^k, we overcome the instantaneous migration problem. Expressions 5.4 and 5.5 and Lemmas 5.2 and 5.3 must be updated accordingly but it does not increase the algorithm complexity.

5.5.2 Modelization and Implementation

Since for periodic tasks the job deadlines are known beforehand, we can execute Algorithms 5.1 and 5.2 as an extra-task τ_s. This task τ_s is executed in each time slice TS^c to compute the swaps and boundaries removals for the next time slice TS^{c+1}. Hence, when we reach the boundary of the current time slice, we can directly start executing the next time slice TS^{c+1} because all the task local execution times have already been computed by τ_s. This modelization of Algorithms 5.1 and 5.2 should facilitate the analysis of the schedulability of the system on the real platform executing a real-time operating system.

Note that implementation techniques such as those presented in [Sousa et al. 2011a,b,c] can easily be used with the algorithms proposed in this paper.

5.6 Simulation Results

We evaluated the performances of the swapping and skipping algorithms through an extensive number of simulations. In each experiment, we simulated the scheduling of 1,000 task sets from time 0 to time 100,000. Each task had a period randomly chosen within $[5, 100]$ using a uniform integer distribution. For the experiments where the number of

tasks is not imposed, task utilizations were randomly generated between 0.01 and 0.99 until the targeted system utilization was reached. On the other hand, if the number of tasks was imposed, the task utilizations were generated using the procedure proposed in [Emberson et al. 2010].

The first experiment aims at comparing the improvement on the number of preemptions and migrations when the depth of swapping (i.e., the number of successive time slices with which the current time slice TS^c exchanges execution time) increases. The results can be consulted on Figure 5.12. We see that for the presented range on the number of processors, it does not benefit to the system to increase the depth of swapping beyond 8. Indeed, the difference in terms of preemptions and migrations per job between a depth of 8 and n (i.e., when we swap with all the time slices after TS^c) remains quite small. Hence, for all the following experiments we arbitrarily decided to always use the swapping algorithm with a depth of 8. Furthermore, when EKG could have a standard deviation as high as 1.25 on the preemptions on 16 processors, it tends to diminish with the depth of swapping. Hence, the standard deviation remains around 0.5 for both preemptions and migrations when we use the swapping algorithm with a depth of 8. The skipping algorithm, however, does not improve the standard deviation of EKG.

The next experiments compare the results of the swapping and skipping algorithms used in conjunction with EKG, with the today's best performing online optimal scheduling algorithm, namely RUN.

On Figures 5.13(a) and (b), we show the average number of preemptions and migrations per job released in the system for various number of processors when the platform is fully utilized (i.e., $U = m$). We can note that EKG used with the swapping algorithm (with $\eta = 8$) performs better than EKG used with the skipping algorithm. Furthermore, skipping boundaries after having swapped execution time has a negative impact on the average number of preemptions and migrations.

Figures 5.13(c) and (d) present the average number of preemptions and migrations on a fully utilized platform of 8 processors when the number of tasks varies. We see that there is almost no difference between the swapping and the skipping algorithm in terms of preemptions for a number of tasks greater than 50. Furthermore, EKG upgraded with the swapping algorithm causes only one more preemption per job than RUN in average, while EKG initially caused around four more preemptions than RUN. Moreover, the skipping algorithm used with EKG has almost the same average number of migrations than RUN for 30 tasks and more. The swapping algorithm ($\eta = 8$) even outperforms RUN when

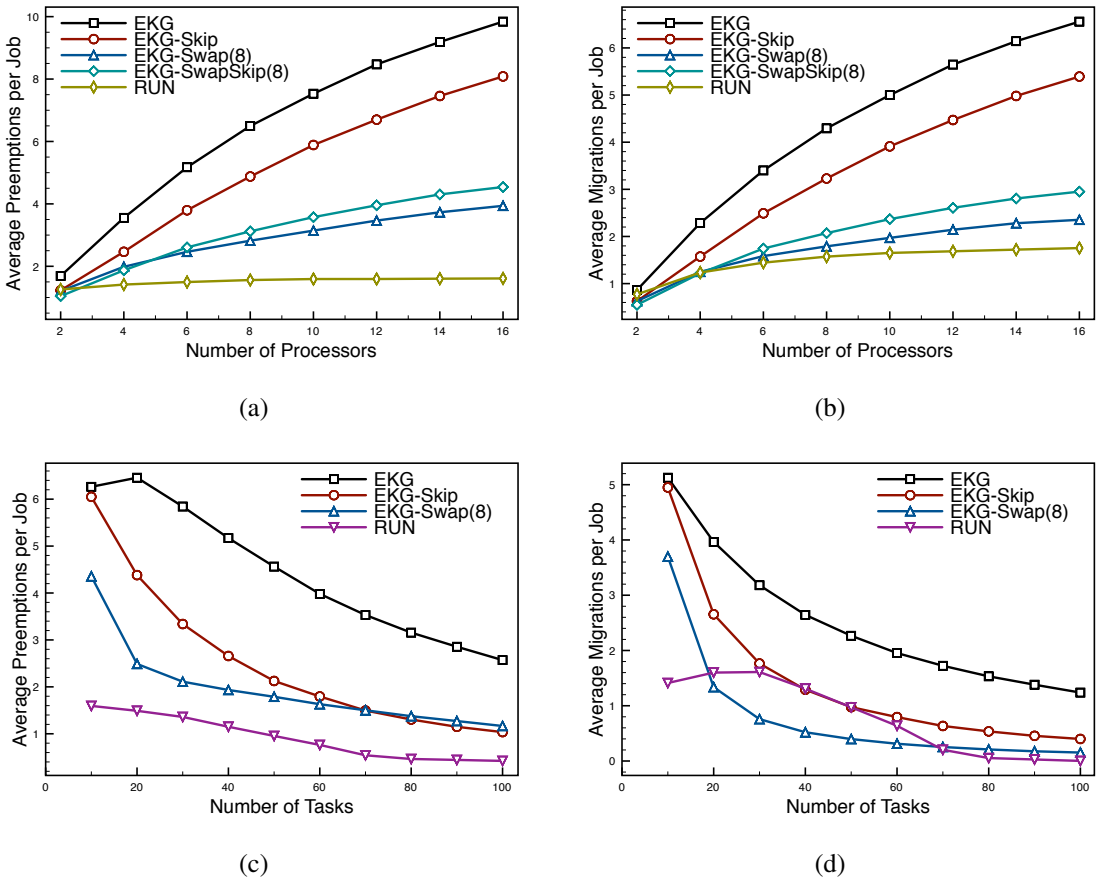


Figure 5.13: Simulation Results: for fully utilized processors varying between 2 and 16 (a) and (b); for 8 fully utilized processors with a varying number of tasks (c) and (d).

there are 20 (or more) tasks in the system.

5.7 Conclusion

In this chapter, we presented two techniques to improve the number of preemptions and migrations incurred by the tasks when they are scheduled with EKG. The simulation results showed a drastic improvement in regard of these two considerations (dividing the average number of preemptions and migrations by more than five in some situations). Hence, EKG used in conjunction with these new algorithms can compete with RUN (i.e., the best performing online optimal algorithm for the scheduling of *periodic* tasks) in terms of preemptions and migrations. However, This improved version of EKG maintains its semi-partitioned approach which allows to upper-bound the number of migrating tasks and improves the code and data locality in memories, thereby reducing the time over-

heads caused by task preemptions. Furthermore, the execution of both the skipping and the swapping algorithm should not be more time consuming than the execution of an algorithm such as RUN. Overall, our new solution should therefore well perform in real applications.

At the light of this chapter, we can now observe that the lowest is the fairness between tasks, the lowest is the number of preemptions and migrations during the schedule. It was true when we favored BFair or DPFair algorithms (such as BF^2 or DP-Wrap) to PFair scheduling policies (such as PF or PD^2). It was true again when EKG grouped tasks in "supertasks" and ensured the fairness for supertasks instead of tasks. It is still true with RUN which does not use any notion of fairness anymore. And it is once again the case with the work presented in this chapter. The improvement in terms of preemptions and migrations has been achieved through a reduction of the fairness quality during the schedule initially produced with EKG. Indeed, by modifying the execution time reserved for each supertask or migratory task in each time slice with the swapping algorithm, supertasks and migratory tasks are not executed proportionally to their utilization anymore. Similarly, because the fairness is initially ensured at the time slice boundaries, by removing some of these boundaries with the skipping algorithm, the computing capacity of the platform is less often fairly distributed between the supertasks and migratory tasks.

It is therefore in this direction (i.e., a reduction of the fairness) that we should continue to investigate to improve the performances of our optimal multiprocessor scheduling algorithms. The next chapter is dedicated to this matter and propose a new *optimal* real-time scheduling algorithm which is not based on the notion of fairness but rather extends EDF — which is optimal and efficient on uniprocessor platform — to the multiprocessor scheduling problem.

CHAPTER

6 An Unfair but Optimal Scheduling Algorithm

"Pour examiner la vérité, il est besoin, une fois dans sa vie,
de mettre toutes choses en doute autant qu'il se peut."
("If you would be a real seeker after truth, it is necessary that at least once in your life you doubt, as far as possible, all things.")

René Descartes

"So much for the golden future, I can't even start.
I've had every promise broken, there's anger in my heart.
You don't know what it's like, you don't have a clue.
If you did you'd find yourselves doing the same thing too.
Breaking the law! Breaking the law!"

Judas Priest

Contents

Abstract

In Chapter 4, we talked about the importance of designing scheduling algorithms based on discrete-time models and proposed a new optimal algorithm named BF^2. Unfortunately, even though the performances of BF^2 outperform those of the state-of-the-art discrete-time optimal scheduling algorithm, the number of preemptions and migrations remains high when comparing with the results of continuous-time solutions.

In Chapter 5, we showed that drastic improvements on the number of preemptions and migrations can be realized when the quality of the fairness is reduced and EDF policies are preferred to the strict fairness appliance.

In this chapter, we propose an algorithm named U-EDF for the scheduling of *sporadic* tasks with implicit deadlines, and we prove its *optimality*. This result is then extended for systems composed of sporadic tasks with unconstrained deadlines when the total density remains smaller than the number of processors in the processing platform. Following the same idea than in Chapter 5, unlike the other existing optimal multiprocessor scheduling algorithms for sporadic tasks, U-EDF is not based on the notion of fairness. Instead, it extends the main principles of EDF so that it achieves optimality while benefiting from a substantial reduction in the number of preemptions and migrations.

Furthermore, we provide a generalization of U-EDF for the scheduling of dynamic task systems and we prove its optimality.

Finally, applying the recommendation made in Chapter 4, we propose a modification of U-EDF for the scheduling of discrete-time systems. However, even though we strongly believe that this discrete-time variation of U-EDF is optimal, we do not provide a proof of its optimality.

Note: Parts of this work were published at the work-in-progress session of RTSS '10 [Nelissen et al. 2010] and as full papers at RTCA'11 [Nelissen et al. 2011a] and ECRTS '12 [Nelissen et al. 2012b].

6.1 Introduction

As presented in Chapter 4, the first optimal scheduling algorithm on multiprocessor platforms considered periodic task sets. This solution named Proportionate Fairness (PFair) was, as its name implies, based on a fair distribution of the processing capacity between tasks (i.e., each task being executed proportionally to its utilization factor) [Baruah et al. 1993, 1996]. Many PFair algorithms were designed over the years (e.g., PD [Baruah et al. 1995], PD^2 [Srinivasan and Anderson 2002], ER-PD [Anderson and Srinivasan 2000a]). However, although the fairness property ensures the optimality, it generates an extensive amount of preemptions and migrations, thereby limiting its applicability.

More recently, it has been noted that imposing a fair progress in task executions at each and every time instant t was too restrictive; respecting the fairness constraint only at task deadlines suffices to reach the optimality [Zhu et al. 2003, 2011; Levin et al. 2010; Funk et al. 2011]. From this observation discussed in detail in Chapters 4 and 5, a new family of schedulers called DP-Fair [Levin et al. 2010; Funk et al. 2011] or Boundary fair [Zhu et al. 2003, 2011] is born. One may cite DP-Wrap [Levin et al. 2010; Funk et al. 2011], LLREF [Cho et al. 2006], BF [Zhu et al. 2003, 2011] or BF^2 (Chapter 4) as some algorithms following this refined concept.

As shown in Chapter 5, Andersson and Tovar went even further with the EKG algorithm, grouping tasks into servers [Andersson and Tovar 2006]. Servers are scheduled in a DP-Fair manner but component tasks of each server are executed according to the EDF algorithm. It has been proven that EKG is optimal for the scheduling of periodic tasks.

Very recently, Regnier et al. presented the RUN algorithm which reduces the multiprocessor scheduling problem to a uniprocessor schedule using a "dualization" technique [Regnier et al. 2011; Regnier 2012]. This algorithm is, to the best of our knowledge, the only online scheduling solution different from the algorithm presented in this chapter, that does not use any fairness assumption but is nevertheless optimal for the scheduling of periodic tasks on multiprocessor. However, this approach does not handle sporadic tasks yet, although intuitions about the possible extension of RUN were proposed in [Regnier 2012].

An interesting observation issued from the study of all the aforementioned algorithms, is the fact that the number of preemptions and migrations decreases as the fairness constraint is relaxed. The impact is even amplified when EDF scheduling rules are incorporated into the scheduler. That is, PFair algorithms cause much more overheads than

Boundary Fair (or DP-Fair) approaches which in turn are more costly in terms of preemptions than EKG. Finally, it was shown that RUN dominates all these algorithms [Regnier et al. 2011; Regnier 2012].

Amongst the previously cited algorithms, some of them were successfully extended to the scheduling of sporadic tasks while keeping their optimality (e.g., PD^2 and ER-PD [Srinivasan and Anderson 2002], DP-Wrap [Funk et al. 2011] and LRE-TL [Funk and Nadadur 2009]). On the other hand, RUN applies only to periodic task sets, and the sporadic generalizations of EKG [Andersson and Bletsas 2008; Bletsas and Andersson 2011] are optimal only in some particular configurations which often cause an extensive amount of preemptions and migrations [Bastoni et al. 2011].

Hence, we can observe that, so far, the only scheduling algorithms that are optimal for the scheduling of periodic and *sporadic* tasks are based on the notion of fairness and usually cause a large amount of preemptions and migrations.

Contribution: In this chapter, we propose a generalization of EDF for the scheduling on multiprocessor platforms. This generalization — named U-EDF — is *optimal* for the scheduling of *sporadic* and *dynamic* task systems in *continuous-time* environments. In contrast with other existing algorithms, U-EDF is not based on the notion of fairness, but instead uses the main principles of EDF so that it achieves optimality while benefiting from a substantial reduction in the number of preemptions and migrations compared to other optimal algorithms (see Section 6.10 for precise results). A modification of U-EDF for the scheduling of discrete time systems is also provided. However, we are not able yet of proving the optimality of U-EDF under the discrete-time constraints.

Organization of this chapter: We start this chapter by stating the system model in Section 6.2. Then, in Section 6.3, we present the intuitions lying behind the new optimal scheduling algorithm presented in this chapter. This scheduling algorithm named U-EDF is then exposed in detail in Section 6.4 for the scheduling of periodic tasks with implicit deadlines, for which an efficient implementation is presented in Section 6.5. This model is extended in Section 6.6 to the scheduling of sporadic tasks and dynamic systems. The problem of scheduling dynamic systems in discrete-time environments is addressed in Section 6.7. We prove in Section 6.8 that U-EDF respects all the task deadlines of dynamic systems composed of sporadic tasks with unconstrained deadlines (assuming that $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$ at any time t) in continuous-time environments. Two improvements of U-EDF are proposed in Section 6.9, while Section 6.10 discuss our simulation

results. The chapter is finally concluded in Section 6.11.

6.2 System Model

We tackle the problem of scheduling a real-time system τ composed of n tasks $\tau_1, \tau_2, ..., \tau_n$ on m *identical* processors $\pi_1, \pi_2, \ldots, \pi_m$. We consider that each task $\tau_i \stackrel{\text{def}}{=} \langle C_i, T_i \rangle$ is either *periodic* or *sporadic*. Each task has an *implicit deadline* and is characterized by a worst-case execution time C_i and a minimum inter-arrival time T_i. As in previous chapters, the utilization of a task τ_i is defined as $U_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$. We assume that $U \leq m$ and $U_i \leq 1$ for every task τ_i in τ.

Since we are considering implicit deadline tasks, each task has at most one active job at any time t implying that the terms job and task can be used interchangeably. Hence, as in the two previous chapters, we can refer to the current deadline of an active task τ_i at time t as the deadline of the current active job of τ_i at time t. Thus we have $d_i(t) > t$ for every task τ_i with an active job at time t and we define $d_i(t) = 0$ otherwise. In this entire chapter, we assume that tasks are prioritized following EDF rules. Hence, we say that a task τ_k has a higher priority than a task τ_i if and only if $d_k(t) < d_i(t)$ or $(d_k(t) = d_i(t) \ \wedge \ k < i)$.

We define the lower and the higher priority task set of τ_i at time t as follows

Definition 6.1 (*Lower priority task set*)

The lower priority tasks set $\text{lp}_i(t)$ *is the set of active tasks with lower priorities than task* τ_i *at time* t*. That is,*

$$\text{lp}_i(t) \stackrel{\text{def}}{=} \left\{ \tau_j : d_j(t) > 0 \wedge \Big(\big(d_j(t) > d_i(t)\big) \vee \big(d_j(t) = d_i(t) \ \wedge \ j > i\big) \Big) \right\}$$

Definition 6.2 (*Higher priority task set*)

The higher priority tasks set $\text{hp}_i(t)$ *is the set of active tasks with higher priorities than task* τ_i *at time* t*. That is,*

$$\text{hp}_i(t) \stackrel{\text{def}}{=} \left\{ \tau_j : d_j(t) > 0 \wedge \Big(\big(d_j(t) < d_i(t)\big) \vee \big(d_j(t) = d_i(t) \ \wedge \ j < i\big) \Big) \right\}$$

Note that because the priority of any task τ_x is dependent of its current deadline and because this deadline is modified at each new job released by τ_x, the content of $\text{hp}_i(t)$ and $\text{lp}_i(t)$ may be altered by every new job arrival in the system.

Finally, at any instant t, we define the (worst-case) remaining execution time $\text{ret}_i(t)$ of τ_i as the (maximum) number of time units that the active job of τ_i still has to execute before its deadline $d_i(t)$. If τ_i is not active at time t then $\text{ret}_i(t) = 0$.

6.3 Toward an Optimal EDF-based Scheduling Algorithm on Multiprocessor

Unlike in the rest of this chapter, in this section, we distinguish between the scheduling of jobs and tasks. Scheduling a set of *known* jobs assumes that all job release times are known at design time for the whole system lifespan. In contrast, scheduling sporadic tasks only assumes that the minimum inter-arrival time between two job releases of a same task is known beforehand. That implies that the schedule cannot be constructed offline since scheduling decisions depend on the actual job arrivals.

6.3.1 Scheduling Jobs

The Earliest Deadline First (EDF) algorithm optimally schedules any feasible set of jobs on a uniprocessor platform [Liu and Layland 1973]. As its name implies, this algorithm gives the highest priority to the active job with the earliest deadline.

In the next two paragraphs, we present two possible generalizations of EDF to multiprocessor (i.e., both reduce to EDF when the platform is composed of only one processor): the first generalization is said to be "vertical" while the second is "horizontal".

Vertical Generalization

The straightforward generalization of EDF to multiprocessor platforms is certainly *global EDF* (G-EDF). With G-EDF, jobs are still prioritized according to their deadlines and, at any instant t, the m processors of the platform are allocated to the m active jobs of highest priorities. Informally, we qualify this allocation rule as being vertical for the following reason: at any time t, in the usual representation of a schedule (see Figure 6.1(a)), the m highest priority jobs are vertically assigned to processors "from top to bottom" (but not necessarily in a decreasing priority order).

Unfortunately, as stated in [Dhall and Liu 1978] and shown in the following example, G-EDF is not optimal on multiprocessor.

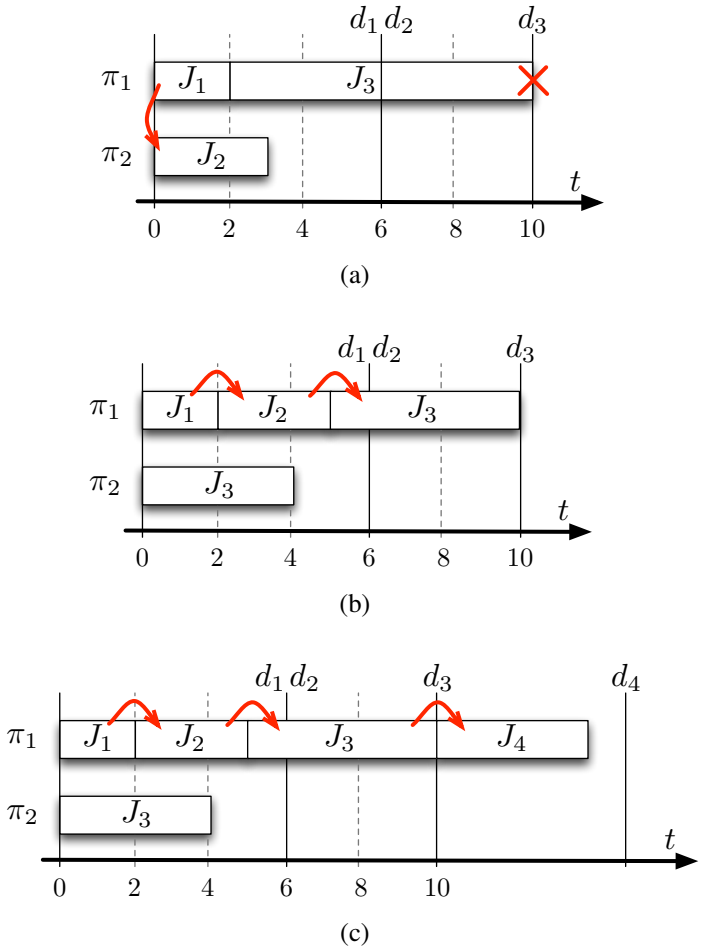


Figure 6.1: Vertical (a) and horizontal (b and c) generalizations of EDF.

Example:

Let us consider a platform composed of two processors and the three jobs $J_1 = \langle 2,6 \rangle$, $J_2 = \langle 3,6 \rangle$, $J_3 = \langle 9,10 \rangle$, *where each tuple* $\langle c,d \rangle$ *describes the computation time* c *and the deadline* d *of a job (as we consider jobs and not tasks, we do not specify any period). We assume that all these jobs are released at time 0. Using G-EDF to schedule them, we can see on Figure 6.1(a) that* J_3 *misses its deadline. The reason being that G-EDF always executes the highest priority jobs as soon as possible without caring for what could happen afterward. Consequently, after executing* J_1 *and* J_2*, it is already hopeless for* J_3 *to meet its deadline unless* J_3 *could execute in parallel on both processors.*

Horizontal Generalization

We should note that, EDF schedules highest priority jobs sequentially whereas G-EDF executes these same jobs in parallel. The rationale behind this parallel execution scheme is the will to execute highest priority jobs *as soon as* possible, in spite of the possibly negative impact on the future execution context.

But why should we hurry and start using more than one processor when it is not needed? An alternative approach would consist in scheduling highest priority jobs (i.e., jobs with earliest deadlines) *sequentially* on the first processor, and start to allocate π_2 only if processor π_1 cannot afford to schedule all jobs while respecting their deadlines. The idea is to maximize the workload executed by the first processor before starting to use the second one. Similarly, we will maximize the workload assigned on both π_1 and π_2 before starting to allocate tasks to the third processor π_3.

Example:

Consider the same three jobs executed on the same platform as in the previous example. The "horizontal" generalization is illustrated on Figure 6.1(b). Similarly to Figure 6.1(a), the first job J_1 is executed on π_1 between instants 0 and 2. However, contrarily to G-EDF, the job J_2 is also allocated to processor π_1 between instants 2 and 5 (instead of processor π_2 in G-EDF). The objective of this new approach is to maximize the workload on the already-allocated processors (here π_1) before using another processor (here π_2). Five time units are then allotted to J_3 on π_1, up to its deadline d_3. Obviously, we cannot further schedule J_3 on π_1 without missing its deadline. Hence, we consider that π_1 is "full", and continue the execution of J_3 on π_2, from time 0 to 4.

Note that it is not because the processor π_1 was considered as being "full" when assigning J_3 that next jobs with later deadlines cannot be assigned to π_1 afterwards.

Example:

Assume that a fourth job $J_4 \stackrel{\text{def}}{=} \langle 4, 15\rangle$ is released at time 0 in the previous example. Then, it can be entirely assigned to the processor π_1 within $[10, 14)$ and still respect its deadline as illustrated on Figure 6.1(c). The goal is still the same; maximizing the workload assigned to the first processor π_1 before utilizing the second processor π_2.

In contrast to G-EDF which vertically extends EDF, this generalization of EDF may be seen as being *horizontal*. Indeed, as shown in Figure 6.1(b) and previously detailed in our example, after scheduling J_1 on π_1, we schedule J_2 on the same processor, directly

after the completion of J_1. The second processor π_2 starts being allocated to a job only if this job cannot be entirely executed on π_1 by its deadline. Jobs are thus disposed one by one on the processors (following EDF), in a *horizontal* manner.

This horizontal approach is in line with the volition of EDF as it tries to maximize the amount of work executed by jobs of highest priorities before their respective deadlines. The difference between G-EDF and this "horizontal" generalization is that G-EDF is driven by the aforementioned will to execute jobs *as soon as* possible whereas the "horizontal" scheduler executes *as much as* possible on as few processors as possible.

6.3.2 A First Attempt of Horizontal Generalization: SA

The algorithm SA proposed by Khemka and Shyamasundar [1997] addresses the problem of scheduling a set of *periodic* tasks with implicit deadlines executed in a *discrete-time* environment. The scheduling algorithm SA produces a correct schedule if for every pair of two tasks τ_i and τ_j in the set τ, at least one of the two following conditions is respected:

C_1. T_i divides T_j, i.e., $\frac{T_j}{T_i}$ is a natural number (it is assumed without any loss of generality that $T_i \leq T_j$).

C_2. $\gcd(T_i, T_j) \times U_i$ and $\gcd(T_i, T_j) \times U_j$ are both integers.

The algorithm SA constructs a schedule on the entire hyper-period of the task set, i.e., from time 0 to $HP \stackrel{\text{def}}{=} \text{lcm}\{T_1, T_2, ..., T_n\}$. This schedule computed offline is then repeated online every HP time units.

The algorithm SA decomposes the time in slices of length $T \stackrel{\text{def}}{=} \gcd\{T_1, T_2, ..., T_n\}$. Then, it defines a set $\mathcal{F}$ as $\mathcal{F} \stackrel{\text{def}}{=} \left\{ \frac{T_i}{T}, \forall \tau_i \in \tau \right\} \cup \left\{ \gcd(\frac{T_i}{T_j}, \forall \tau_i, \tau_j \in \tau \right) \}$. Since every pair of tasks in τ respects at least one of the two conditions C_1 or C_2, for each of the factor $f \in \mathcal{F}$ there is at least one task $\tau_i \in \tau$ such that $f \times T \times U_i$ is a natural number. For each of the factor $f \in \mathcal{F}$ (considered in an increasing order), the SA algorithm takes the tasks $\tau_i \in \tau$ such that $f \times T \times U_i$ is a natural number and schedules these tasks for exactly $f \times T \times U_i$ time units between time 0 and $f \times T$ using the horizontal approach proposed in the previous section. That is, it first maximizes the utilization of the first processor π_1, then the second processor π_2 and so on. This schedule is then replicated in any interval extending from time $\big(k \times f \times T\big)$ to $\big((k+1) \times f \times T\big)$ where k is an integer greater than 0.

Note that if all tasks composing τ respect the condition C_1 then SA may allocate the tasks in an earliest deadline first order and therefore produces the horizontal EDF schedule

proposed in the previous section. Indeed, let us consider the following example already proposed in [Khemka and Shyamasundar 1997].

Example:

Let the task set τ be composed of six tasks $\tau_1 = \langle 6, 10\rangle$, $\tau_2 = \langle 5, 10\rangle$, $\tau_3 = \langle 11, 30\rangle$, $\tau_4 = \langle 17, 30\rangle$, $\tau_5 = \langle 31, 60\rangle$ and $\tau_6 = \langle 27, 60\rangle$ where $\tau_i \stackrel{\text{def}}{=} \langle C_i, T_i\rangle$. We have that $\frac{T_i}{T_j}$ is an integer for every pair of tasks τ_i and τ_j in τ. The condition C_1 is therefore respected for all tasks composing the task set and SA is able to produce a correct schedule. The time is divided in slices of length $T = \text{gcd}_{\tau_i \in \tau}\{\tau_i\} = 10$ and the schedule is constructed on the entire hyper-period $HP = \text{lcm}_{\tau_i \in \tau}\{\tau_i\} = 60$. The set $\mathcal{F}$ is given by $\mathcal{F} = \{1, 3, 6\}$. The two first tasks being considered by SA are τ_1 and τ_2 (the tasks with a deadline of $1 \times T = 10$ time units). The task τ_1 is first scheduled for 6 time units on the first processor π_1 between time 0 and 6 (see Figure 6.2(a)). Then, τ_2 is scheduled on π_1 from time 6 up to its first deadline at time 10. The remaining execution time of the first job of τ_2 is scheduled on processor π_2 from time 0 to 1. This schedule is repeated in every time slice of length $T = 10$ as illustrated on Figure 6.2(a). The next tasks to be considered are τ_3 and τ_4 (i.e., the tasks with a deadline of $3 \times T = 30$ time units). Since the processor π_1 is now completely allocated to the execution of the tasks τ_1 and τ_2, SA tries to maximize the utilization of processor π_2. Because τ_3 has a period of 30 time units, the first job of τ_3 is allocated to π_2 between instant 1 and 10 and then 11 and 13 (see Figure 6.2(b)). On the other hand, the first job of τ_4 is scheduled between instants 13 and 20 and from time 21 to 30 on processor π_2. Because the deadline of this first job is at time 30, SA cannot continue executing this job on processor π_2 without missing its deadline. Hence, it starts using processor π_3 on which it schedules the execution of the remaining part of first job of τ_4 within $[0, 1)$ (Figure 6.2(b)). This pattern of execution of τ_3 and τ_4 is then repeated every 30 time units (i.e., in every group of three time slices of length T). Finally, the tasks τ_5 and τ_6 (i.e., the tasks with a deadline of $6 \times T = 60$ time units) are scheduled in the remaining free "holes" in the schedule between instants 0 and 60 on processor π_3 (see Figure 6.2(c)).

This scheduling scheme is indeed a horizontal generalization of EDF as introduced in Section 6.3.1 since the allocation is performed in an increasing relative deadline order and it first maximizes the utilization of the first processor π_1, then processor π_2 and finally processor π_3. However, we should remember that SA is a horizontal generalization of EDF only when the condition C_1 is respected for all *tasks in τ.*

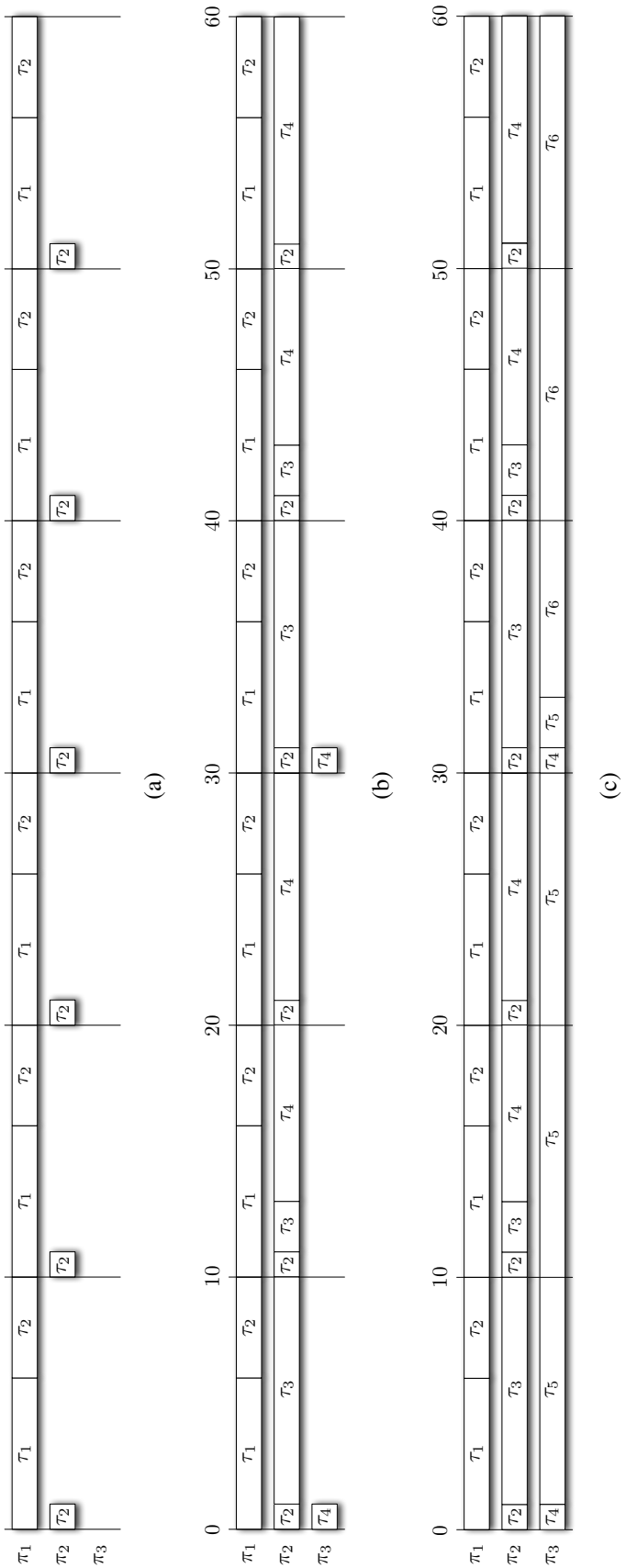


Figure 6.2: Illustration of the scheduling algorithm SA.

6.3.3 The Problem of Scheduling Sporadic Tasks

In addition to the restriction imposed on the task set — i.e., the task set must be composed exclusively of periodic tasks with implicit deadlines respecting either condition C_1 or C_2 —, the SA algorithm suffers from a major drawback; the schedule must be constructed offline, thereby implying that the application must be entirely defined at design time. Hence, it supposes that the task set is static (i.e., no task can dynamically leave or join the task set during the schedule) and every job arrival is known beforehand. Unfortunately, exact job arrival times cannot be known *a priori* when considering sporadic tasks. It results that offline solutions such as SA do not apply for the scheduling of sporadic tasks.

6.3.4 U-EDF: A New Solution for the Optimal Scheduling of Sporadic Tasks

In this chapter, we propose a new scheduling algorithm generalizing EDF in an horizontal manner. This algorithm named U-EDF is composed of two phases that will be discussed in detail in Sections 6.4.1 and 6.4.2, respectively. These two phases can be summarized as follows:

1) First, whenever a new job is released, U-EDF pre-allocates execution times to the active tasks using the horizontal approach introduced in Section 6.3.1. That is, tasks are considered in an increasing current deadline order and U-EDF tries to maximize the utilization of the first processor, then the second and so on. Nevertheless, during this execution time allocation, extra time proportionate to the utilization factor U_i of every task τ_i is preventively reserved on the platform in any time interval following the deadline of every task τ_i (see Section 6.4.1). The main idea of this reservation technique inspired by scheduling algorithms such as DP-Wrap (Section 5.2.2) or EKG (Section 5.2.4), is to save enough computing resources for the future jobs that might potentially be released and interfere with the schedule of the currently active jobs.

2) Then, according to the allocation decided during the first phase, active jobs are executed on each processor using a slight variation of EDF that avoids parallel execution of a same job. This EDF variation is named EDF-D and is presented in Section 6.4.2.

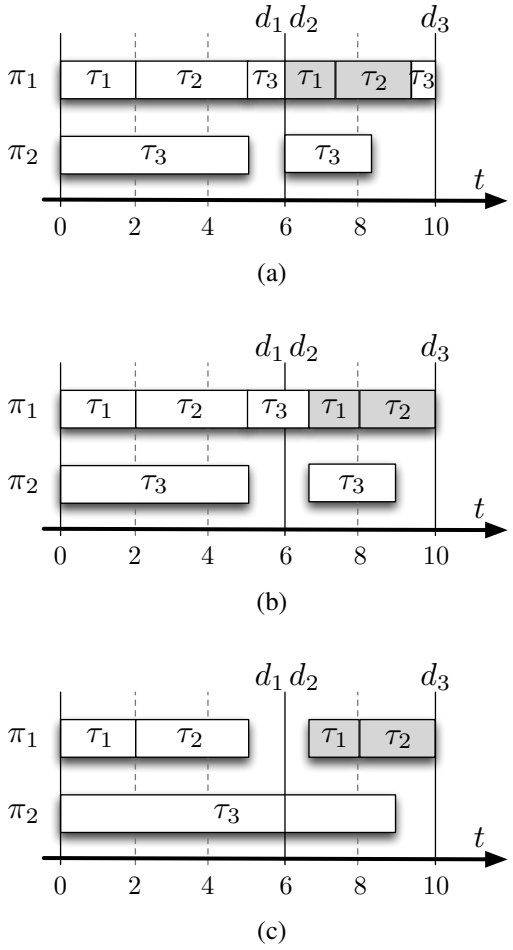


Figure 6.3: Generation of an U-EDF schedule at time 0: (a) Allocation and reservation scheme at time 0. (b) Actual schedule if no new job arrives before 10. (c) Effective schedule with processor virtualization.

Example:

Let us consider the example presented in Figure 6.3(a). We have three implicit deadline tasks $\tau_1 \stackrel{\text{def}}{=} \langle 2,6 \rangle$, $\tau_2 \stackrel{\text{def}}{=} \langle 3,6 \rangle$ *and* $\tau_3 \stackrel{\text{def}}{=} \langle 9,10 \rangle$. *Once* τ_1 *has been assigned to* π_1, *we reserve (in gray on the figure) a fraction* $1/3$ *of the processor time* ($U_1 = 1/3$) *for the execution of future jobs of* τ_1 *that might be released after its current deadline. Then,* τ_2 *is assigned to* π_1, *and we reserve an half* ($U_2 = 1/2$) *of a processor for the possible future job it might release after its deadline at time* 6. *Finally,* τ_3 *is allocated by filling the gaps, first on processor* π_1 *up to its deadline, and then on* π_2.

Note that during the scheduling phase, the time reserved for future jobs (in gray on Figure 6.3(a)) will actually be utilized only if new jobs arrive. On the other hand, if no new job is released by τ_1 or τ_2, task τ_3 will just keep on running as shown in Figure 6.3(b).

Of course, further improvements can still be carried out. For instance, on Figure 6.3(b), τ_3 instantly migrates from π_2 to π_1 at time 5, and inversely at time 7. A virtual processor mechanism presented in Section 6.9.1 avoids these pointless migrations and produces the schedule illustrated in Figure 6.3(c).

We would like to insist on the fact that the schedule presented on Figure 6.3(b) or 6.3(c) might be modified if new jobs are released. Indeed, U-EDF recomputes the task allocation whenever a new job arrives in the system. For that reason, U-EDF can be seen as bridging the notions of global and semi-partitioned algorithms. Indeed, with U-EDF, tasks are partitioned between the processors of the platforms. Some tasks being assigned to only one processor while others are assigned to many processors. However, the partitioning is updated at any new job release, thereby implying that the same tasks are not always assigned to the same processors.

6.4 U-EDF: Scheduling Algorithm for Periodic Tasks in a Continuous-Time Environment

As introduced in the previous section, U-EDF is composed of two different phases. First, it uses the "horizontal" technique to allot a time of execution to every task on each processor (this phase is repeated whenever a new job arrives). Then, between two such execution time pre-allocations (i.e., between two job releases), it schedules the active tasks according to their associated budgets, using a slight variation of EDF on each processor (named EDF with Delays because, in order to avoid intra-job parallelism, it may impose to delay the execution of some tasks in comparison with EDF).

Before explaining the pre-allocation process, we first define some notations.

Definition 6.3 (*Current job allotment*)

At any instant t, the allotment $\text{allot}_{i,j}(t)$ *of task τ_i is the amount of time allocated on processor π_j to the execution of the* active job *of τ_i.*

The allotment of τ_i on π_j is decreasing as τ_i is running on π_j. Hence, assuming that no new job arrives within $[t,t')$, if $\text{exec}_{i,j}(t,t')$ denotes the time during which τ_i has been running on processor π_j in the interval $[t,t')$ then

$$\text{allot}_{i,j}(t') \stackrel{\text{def}}{=} \text{allot}_{i,j}(t) - \text{exec}_{i,j}(t,t') \tag{6.1}$$

To be able to respect its deadline, the current job of a task τ_i needs that the sum of execution times allotted to its execution on the platform to be at least equal to its (worst-case) remaining execution time. A necessary condition to respect all job deadlines is therefore to have a valid assignment defined as follows

Definition 6.4 (*Valid assignment*)

An assignment is said to be valid at time t if and only if the allotment of every task $\tau_i \in \tau$ is sufficient to successfully schedule its remaining execution time. That is,

$$\forall \tau_i \in \tau : \sum_{j=1}^{m} \text{allot}_{i,j}(t) \geq \text{ret}_i(t)$$

Nevertheless, as already mentioned in Section 6.3.4, ensuring that a correct schedule may exist for the currently active jobs does not suffice. We must also make sure that the future jobs that are not yet arrived, will get enough time to execute prior to their deadlines. However, we do not want to compute the exact job arrival times as it would be time consuming and therefore impractical with an online algorithm. Hence, the fact that we do not know when the future jobs will be released increases the difficulty of taking appropriate scheduling decisions. We therefore simply reserve a portion of the platform computation capacity for the execution of these future jobs that might potentially arrive. The idea is to make sure that, after the current task deadline of any task τ_i, C_i time units are reserved for the execution of τ_i in any time interval of T_i time units. For each task τ_i, we therefore reserve

$$\text{res}_i(t_1,t_2) \stackrel{\text{def}}{=} U_i \times (t_2 - t_1)$$

time units in any time interval $[t_1,t_2)$ following the current deadline $d_i(t)$ of τ_i (with $t \leq t_1 < t_2$). Hence, we have that

$$\text{res}_i(t_1,t_1+T_i) = U_i \times T_i = C_i$$

implying that C_i time units are indeed reserved for the execution of τ_i in any time interval of T_i time units.

This time reservation is dispatched between the processors composing the platform using the following definition:

Definition 6.5 (*Future jobs reservation*)

The future jobs reservation $\text{res}_{i,j}(t_1,t_2)$ *computed at time t for a task τ_i, denotes the amount of time reserved on processor π_j for the execution of the* future jobs *of τ_i that*

might be released within $[t_1,t_2)$ *where* $d_i(t) \leq t_1 < t_2$. *This reservation is defined as follows:*

$$\text{res}_{i,j}(t_1,t_2) \stackrel{\text{def}}{=} u_{i,j}(t) \times (t_2 - t_1)$$

where

$$u_{i,j}(t) \stackrel{\text{def}}{=} \left[\sum_{\tau_x \in \text{hp}_i(t) \cup \tau_i} U_x \right]_{j-1}^{j} - \left[\sum_{\tau_x \in \text{hp}_i(t)} U_x \right]_{j-1}^{j}$$

with

$$[x]_a^b \stackrel{\text{def}}{=} \max\{a, \min\{b,x\}\}.^1$$

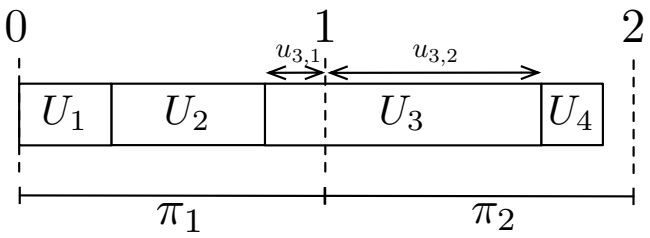


Figure 6.4: Computation of $u_{i,j}$.

The $u_{i,j}(t)$ value denotes the proportion of the utilization factor of τ_i that will be reserved on processor π_j for the scheduling of future jobs of τ_i. This value is obtained by aligning blocs of size U_i in an increasing current deadline $d_i(t)$ order, and then, by cutting this alignment in boxes of size 1 (see Figure 6.4). The portion of U_i contained in the j^{th} box corresponds to $u_{i,j}(t)$. Indeed, the term $\left[\sum_{\tau_x \in \text{hp}_i(t)} U_x \right]_{j-1}^{j}$ gives the part of $\sum_{\tau_x \in \text{hp}_i(t)} U_x$ contained in the j^{th} segment of size 1. Similarly, the term $\left[\sum_{\tau_x \in \text{hp}_i(t) \cup \tau_i} U_x \right]_{j-1}^{j}$ provides the part of $\sum_{\tau_x \in \text{hp}_i(t) \cup \tau_i} U_x$ which is greater than $j-1$ but smaller than j. Hence, by subtracting both values we get the part of U_i contained in the j^{th} segment of size 1.

[1] That is, the operator $[x]_a^b$ only considers the value of x if it is included within the interval $[a,b]$. If x is not in this interval it bounds its value to a if $x < a$ or b if $x > b$.

Example:

Let τ_1, τ_2 and τ_3 be three tasks with utilizations $U_1 = 0.7$, $U_2 = 0.8$ and $U_3 = 0.6$ respectively and with current deadlines $d_1(t) < d_2(t) < d_3(t)$. These tasks are scheduled on three processors π_1, π_2 and π_3.
According to the previous explanations on the computation of $u_{i,j}(t)$, we have that $u_{1,1}(t) = 0.7$ and $u_{1,2}(t) = u_{1,3}(t) = 0$. Indeed, the utilization U_1 of τ_1 is smaller than 1 and therefore entirely holds on the first processor π_1. On the other hand, we have that $U_1 + U_2 = 1.5$ which is greater than 1. Hence, we get that $u_{2,1}(t) = 1 - 0.7 = 0.3$, $u_{2,2}(t) = 1.5 - 1 = 0.5$ and $u_{2,3}(t) = 0$. Similarly, $U_1 + U_2 + U_3 = 2.1$ which is greater than 2. Hence, $u_{3,1}(t) = 0$ because we already entirely reserved π_1 for the tasks τ_1 and τ_2, but $u_{3,2}(t) = 2 - 1.5 = 0.5$ and $u_{3,3}(t) = 2.1 - 2 = 0.1$.
Note that $\sum_{j=1}^{3} u_{i,j}(t) = U_i$ for every task τ_i.

This reservation technique has been inspired by the DPFair approach (Section 5.2) and more particularly by the algorithms DP-Wrap (Section 5.2.2) and EKG (Section 5.2.4). However, unlike these two algorithms, it does not introduce any fairness in the resulting schedule of U-EDF since it only reserves time for jobs that are not arrived yet and thus not actually executed.

Two important properties that will be useful later in this chapter, are now enunciated and are proven in Appendix B:

Lemma 6.1

Let τ_i be any task in τ. It holds that

$$\sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t) = \left[\sum_{\tau_x \in \mathrm{hp}_i(t)} U_x \right]_{j-1}^{j} - (j-1)$$

Lemma 6.2

Let x be any real number and let s be a natural number greater than 0. It holds that

$$\sum_{k=1}^{s} \left([x]_{k-1}^{k} - (k-1) \right) = [x]_0^s$$

Furthermore, if $0 \leq x \leq s$ then

$$[x]_0^s = x$$

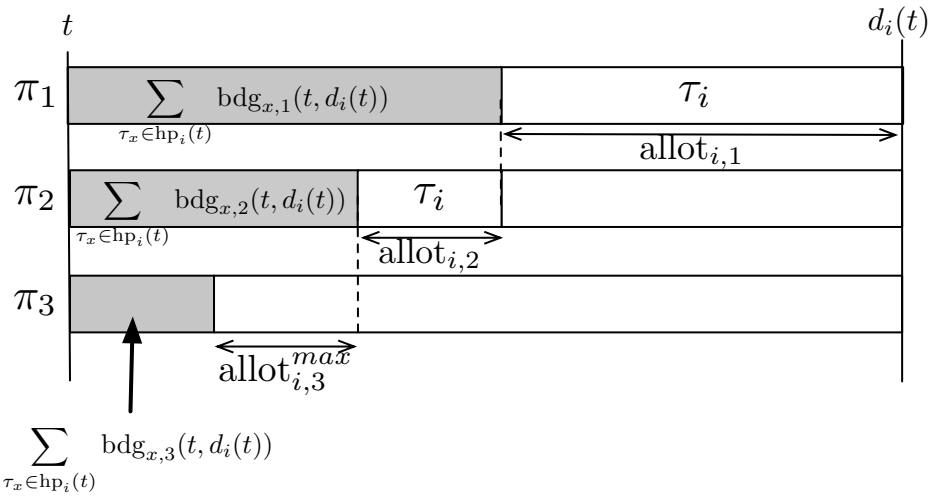


Figure 6.5: Computation of $\mathrm{allot}_{i,j}^{\max}(t)$ for τ_i on processor π_3.

The current job allotment and the future jobs reservation of a task τ_i represent altogether a budget of execution time allocated to τ_i on a processor π_j. We therefore define the budget of a task τ_i on a processor π_j in a time-interval $[t_1, t_2)$ as follows.

Definition 6.6 (*Task budget*)

The budget $\mathrm{bdg}_{i,j}(t_1, t_2)$ *denotes the total amount of time reserved for the task* τ_i *on processor* π_j *in the interval extending from* t_1 *to* t_2 *(with* $t_2 \geq d_i(t_1)$*). It includes the allotment* $\mathrm{allot}_{i,j}(t_1)$ *for the currently active job of* τ_i *at time* t_1 *and the reservation* $\mathrm{res}_{i,j}(d_i(t_1), t_2)$ *for the execution of the future jobs of* τ_i *that might potentially arrive within* $[d_i(t_1), t_2)$*, i.e.,*

$$\mathrm{bdg}_{i,j}(t_1, t_2) \stackrel{\text{def}}{=} \mathrm{allot}_{i,j}(t_1) + \mathrm{res}_{i,j}(d_i(t_1), t_2)$$

6.4.1 First phase: Pre-Allocation

As explained earlier, the first phase of our algorithm consists in pre-allocating time for the execution of tasks on processors, i.e., for each task determine the time budget allocated for its execution on each processor. Notice that, as we reassign tasks at every new job arrival, there is no guaranty that we will actually execute τ_i for its allocated time on π_j.

To ensure that a task will never have to run in parallel on two or more processors

Algorithm 6.1: Pre-allocation Algorithm.

Input:
TaskList := list of the n tasks sorted by increasing absolute deadlines;
t := current time;

```
1 forall the τ_i ∈ TaskList do
2     for j := 1 to m do
3         allot_{i,j}(t) := min{allot^max_{i,j}(t) , ret_i(t) − Σ_{y<j} allot_{i,y}(t)};
4     end
5 end
```

to complete its execution, we need to define the maximum amount of time a task τ_i can execute on a processor π_j without any risk of intra-job parallelism. This value will then be used during the first phase of the U-EDF algorithm which pre-allocates time to tasks for their execution on each processor.

Definition 6.7 (*Maximum allotment*)

At any time-instant t, the maximum allotment $\text{allot}_{i,j}^{\max}(t)$ of task τ_i on processor π_j is defined as the maximum amount of time that can be allocated on processor π_j to the active job of task τ_i in the time interval $[t, d_i(t))$. This upper-bound on $\text{allot}_{i,j}(t)$ is given by

$$\text{allot}_{i,j}^{\max}(t) \stackrel{\text{def}}{=} \big(d_i(t) - t\big) - \sum_{\tau_x \in \text{hp}_i(t)} \text{bdg}_{x,j}(t, d_i(t)) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

The first term of this expression, i.e., $(d_i(t) - t)$, is the amount of time between instant t and the deadline of the active job of τ_i, the second term $\sum_{\tau_x \in \text{hp}_i(t)} \text{bdg}_{x,j}(t, d_i(t))$ is the amount of time reserved in $[t, d_i(t))$ for the tasks with a higher priority than τ_i, and the third term $\sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$ is the amount of time already reserved for the active job of τ_i on the processors of lower indices than π_j. Of course, the task τ_i cannot be executed for more time than the time remaining until its deadline minus the time already reserved to execute the tasks with higher priorities than τ_i. Hence, $\text{allot}_{i,j}^{\max}(t) \leq \big(d_i(t) - t\big) - \sum_{\tau_x \in \text{hp}_i(t)} \text{bdg}_{x,j}(t, d_i(t))$. The third term $\sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$ on the other hand is subtracted in order to prevent the active job of τ_i from being executed concurrently on multiple processors (see Figure 6.5). Note that this value is pessimistic and only considers the worst-case scenario of execution. Nevertheless, it does not impact the optimality of U-EDF as it will be proven in Section 6.8.

Algorithm 6.1 presents the pre-allocation phase of U-EDF. Tasks are pre-allocated in

an increasing current deadline order. Algorithm 6.1 maximizes the time allotted to each task τ_i on each processor. Hence, for every task τ_i, it assigns on each processor π_j the minimum between $\text{allot}_{i,j}^{\max}(t)$ and the amount of remaining execution time of τ_i that has not yet been assigned to other processors.

We easily recognize the "horizontal" approach presented in Section 6.3.1. Indeed, Algorithm 6.1 allocates time to tasks in an increasing current deadline order and maximizes the utilization of the first processors before starting using next ones.

6.4.2 Second phase: Scheduling

After the pre-allocation of the processor execution time to the tasks, we use a variation of the EDF algorithm named EDF-D (which stands for EDF with Delays) to schedule the tasks on the processors until the next job arrival (where a reallocation of the tasks will be performed again). The core idea of EDF-D is to delay the execution of a task in comparison of the schedule that would have been produced by EDF, whenever the execution of this task would have led to some kind of intra-job parallelism (i.e., the same job running on two or more processors simultaneously).

EDF-D is a scheduling algorithm which operates as follows:

- It first builds the set $\mathcal{E}_j(t)$ of eligible tasks on π_j at time t. A task τ_i is eligible on π_j at time t if
  - **(i)** it is not currently running on a processor of *lower* index; and
  - **(ii)** τ_i still has to execute on processor π_j according to the current task allotment (i.e., $\text{allot}_{i,j}(t) > 0$).
- At any time t, EDF-D schedules the *eligible* task in $\mathcal{E}_j(t)$ with the earliest deadline.

In other words, EDF-D behaves the same way as EDF on each processor, except that as soon as a job is running on a processor π_j, it cannot be scheduled anymore on any processor with a higher index (see Figure 6.3(b) on page 170 for an illustration). This leads to the following property:

Property 6.1

If a task τ_i with $\text{allot}_{i,j}(t) > 0$ is not running on processor π_j at time t, then either a task with a higher priority than τ_i is executing on π_j, or τ_i is running on a processor

Algorithm 6.2: U-EDF scheduling algorithm.

```
   Data: t := current time
1  if t is the arrival time of a job then
       // Recompute the task allotment
2      Call Algorithm 6.1 ;
3  end
4  for j := 1 to m do
       // Use EDF-D
5      Update E_j(t) ;
6      Execute on π_j the task with the earliest deadline in E_j(t) ;
7  end
```

with a smaller index at time t.

Lemma 6.3

If EDF-D is applied during the time interval $[t_1, t_2)$ then there is no intra-job parallelism within this time interval.

Proof:

Since any task τ_i running on any processor π_j is removed from all the sets of eligible tasks of processors π_1 to π_{j-1}, τ_i cannot execute on two processors simultaneously. ■

6.4.3 Summary: the U-EDF algorithm

Algorithm 6.2 summarizes how U-EDF works. Whenever an event (i.e., a job arrival, a job deadline or a job completion) occurs in the system, Algorithm 6.2 is called. If the event is the arrival of a new job, U-EDF recomputes a new task allocation amongst the processors. Then, irrespective of the event type, EDF-D is used to schedule the tasks on the platform.

Note that U-EDF is a generalization of EDF as stated by the following property:

Property 6.2

U-EDF behaves exactly as EDF when the platform is composed of a single processor and $U \leq 1$.

Proof:

In the particular case where the number of processors m is equal to 1, all tasks are obviously assigned to the same and only processor during the pre-allocation phase.

Furthermore, a task cannot be delayed when using EDF-D as it never executes on another processor than the only existing processor. That is, no task is ever removed from the set of eligible tasks to be executed with EDF. Consequently, EDF-D reduces to EDF during the scheduling phase. ■

6.5 U-EDF: Efficient Implementation

At a first sight, the computation of $\text{allot}_{i,j}(t)$ may seem time consuming in Algorithm 6.1. Indeed, it seems to require the computation of the budget of each task belonging to $\text{hp}_i(t)$. The computation of $\text{allot}_{i,j}(t)$ would therefore have a complexity of $O(n)$ and because there are two imbricated loops with n and m iterations respectively, the run-time complexity of Algorithm 6.1 would be of $O\left(m \times n^2\right)$. This complexity is excessively high for an online scheduling algorithm. In these conditions, U-EDF could not viably be used in actual systems. However, Algorithm 6.1 can be implemented in an iterative manner, thereby reducing its complexity to $O\left(m \times n\right)$.

Let us consider the computation of $\text{allot}_{i,j}(t)$ at line 3 of Algorithm 6.1. We have

$$\text{allot}_{i,j}(t) := \min \left\{ \text{allot}_{i,j}^{\max}(t) \ , \ \text{ret}_i(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t) \right\}$$

with (using Definitions 6.6 and 6.7)

$$\text{allot}_{i,j}^{\max}(t) = \left(d_i(t) - t\right) - \sum_{\tau_x \in \text{hp}_i(t)} \left\{ \text{allot}_{x,j}(t) + \text{res}_{x,j}(d_x(t), d_i(t)) \right\} - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

Analyzing these expressions, we detect three terms that should be iteratively computed: (i) $\sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$, (ii) $\sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t)$ and (iii) $\sum_{\tau_x \in \text{hp}_i(t)} \text{res}_{x,j}(d_x(t), d_i(t))$. We can easily transform the terms (i) and (ii) in an iterative form. The term (iii) is more complex and needs more reflection.

(i) $\sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$.

Let $\text{PREV}_i^{(j-1)} \stackrel{\text{def}}{=} \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$ be the total amount of execution time already allocated to τ_i on previous processors when we consider processor π_j. It holds that:

$$\begin{cases} \text{PREV}_i^{(0)} = 0 \\ \text{PREV}_i^{(j)} = \text{PREV}_i^{(j-1)} + \text{allot}_{i,j}(t) \end{cases} \tag{6.2}$$

Because the processors are considered in an increasing index order in Algorithm 6.1, one step can be computed at each iteration of the "for" loop. Note that there is one such variable per task.

(ii) $\sum_{\tau_x \in \mathrm{hp}_i(t)} \mathrm{allot}_{x,j}(t)$.

Similarly to the first term, using $\mathrm{ALLOT}_j^{(i-1)} \stackrel{\text{def}}{=} \sum_{\tau_x \in \mathrm{hp}_i(t)} \mathrm{allot}_{x,j}(t)$ to denote the amount of execution time already allocated to tasks in $\mathrm{hp}_i(t)$ on processor π_j when we consider the pre-allocation of τ_i on π_j, we get

$$\begin{cases} \mathrm{ALLOT}_j^{(0)} = 0 \\ \mathrm{ALLOT}_j^{(i)} = \mathrm{ALLOT}_j^{(i-1)} + \mathrm{allot}_{i,j}(t) \end{cases} \tag{6.3}$$

Because the tasks are considered in an increasing deadline order in Algorithm 6.1, one step can be computed at each iteration of the "for" loop either. Note that unlike $\mathrm{PREV}_i^{(j)}$, there is one variable per processor π_j instead of one variable per task τ_i.

(iii) $\sum_{\tau_x \in \mathrm{hp}_i(t)} \mathrm{res}_{x,j}(d_x(t), d_i(t))$.

At a first sight, it seems that all terms of this sum must be recomputed at each iteration of the loop (i.e., whenever $d_i(t)$ is modified). However, using Definition 6.5 and assuming that the tasks are indexed in a decreasing priority order, we have

$$\begin{aligned} \sum_{\tau_x \in \mathrm{hp}_i(t)} \mathrm{res}_{x,j}(d_x(t), d_i(t)) &= \sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t)(d_i(t) - d_x(t)) \\ &= \sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t)(d_{i-1}(t) - d_x(t)) \\ &\quad + \sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t)(d_i(t) - d_{i-1}(t)) \\ &= \sum_{\tau_x \in \mathrm{hp}_i(t)} \mathrm{res}_{x,j}(d_x(t), d_{i-1}(t)) \\ &\quad + \sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t)(d_i(t) - d_{i-1}(t)) \end{aligned}$$

and applying Lemma 6.1

$$\sum_{\tau_x \in \text{hp}_i(t)} \text{res}_{x,j}(d_x(t), d_i(t)) = \sum_{\tau_x \in \text{hp}_i(t)} \text{res}_{x,j}(d_x(t), d_{i-1}(t)) + \left\{ \left\lfloor \sum_{\tau_x \in \text{hp}_i(t)} U_x \right\rfloor_{j-1}^{j} - (j-1) \right\} \times (d_{i-1}(t), d_i(t)) \tag{6.4}$$

Therefore, if $\text{RES}_j^{(i-1)} \stackrel{\text{def}}{=} \sum_{\tau_x \in \text{hp}_i(t)} \text{res}_{x,j}(d_x(t), d_i(t))$ represents the amount of time reserved on processor π_j for the execution of future jobs that could be released by the tasks in $\text{hp}_i(t)$ when we consider the pre-allocation of τ_i on π_j, then Expression 6.4 can be iteratively computed as follows

$$\begin{cases} W^{(0)} = 0 \\ \text{RES}_j^{(0)} = 0 \\ W^{(i)} = W^{(i-1)} + U_i \\ \text{RES}_j^{(i)} = \text{RES}_j^{(i-1)} + \left\{ \left\lfloor W^{(i)} \right\rfloor_{j-1}^{j} - (j-1) \right\} \times (d_i(t) - d_{i-1}(t)) \end{cases} \tag{6.5}$$

Using Expressions 6.2, 6.3 and 6.5, we can rewrite Algorithm 6.1 in an iterative manner as presented in Algorithm 6.3.

6.6 U-EDF: Sporadic and Dynamic Task System Scheduling in a Continuous-Time Environment

With static systems, the set of tasks that will be active during the system lifespan is completely known at design time (refer to Definition 2.8 on page 19). The tasks composing this task set never leave the system. It is therefore easy to reserve time for the execution of future jobs that they could potentially release after their current deadlines (see Definition 6.5). In dynamic task systems however, tasks might leave or join the system during the schedule (see Definition 2.9 on page 20). Hence, we never know which tasks will be active in the (near or far) future. Reserving execution time for these unknown tasks is consequently more complex.

Since tasks can leave and join the system, we are more interested by the tasks that are

Algorithm 6.3: Iterative version of the pre-allocation algorithm.

Input:
TaskList := list of the n tasks sorted by increasing absolute deadlines;
t := current time;

// Initialization of the iterative variables W, ALLOT_j and RES_j
1 $W := 0$;
2 **for** $j := 1$ **to** m **do**
3 $\quad \text{ALLOT}_j := 0$;
4 $\quad \text{RES}_j := 0$;
5 **end**
6 **forall the** $\tau_i \in$ TaskList **do**
$\quad$ // Initialization of the iterative variable PREV_i
7 $\quad \text{PREV}_i := 0$;
$\quad$ // Update of W
8 $\quad W := W + U_i$;
9 $\quad$ **for** $j := 1$ **to** m **do**
$\quad\quad$ // Update of RES_j
10 $\quad\quad \text{RES}_j := \text{RES}_j + \left\{ [W]_{j-1}^{j} - (j-1) \right\} \times (d_i(t) - d_{i-1}(t))$;
$\quad\quad$ // Computation of $\text{allot}_{i,j}(t)$
11 $\quad\quad \text{allot}_{i,j}^{\max}(t) := \big(d_i(t) - t\big) - \text{ALLOT}_j - \text{RES}_j - \text{PREV}_i$;
12 $\quad\quad \text{allot}_{i,j}(t) := \min\{\text{allot}_{i,j}^{\max}(t)\ ,\ \text{ret}_i - \text{PREV}_i\}$;
$\quad\quad$ // Update of PREV_i and ALLOT_j
13 $\quad\quad \text{PREV}_i := \text{PREV}_i + \text{allot}_{i,j}(t)$;
14 $\quad\quad \text{ALLOT}_j := \text{ALLOT}_j + \text{allot}_{i,j}(t)$;
15 $\quad$ **end**
16 **end**

active at any time t (i.e., the tasks that have released a job that has not reached its deadline yet) rather than the tasks that compose the task set τ at time t. We therefore define the *set of active tasks at time t* which is denoted by $\mathcal{A}(t)$. Formally,

Definition 6.8 (*Set of active tasks*)

The set of active tasks $\mathcal{A}(t)$ at time t contains all tasks which have an active job at time t. That is,

$$\mathcal{A}(t) \stackrel{\text{def}}{=} \{\tau_i \in \tau \mid d_i(t) > t\}$$

Note that even if a job completed its execution at time t, its corresponding task τ_i is still considered as being active at time t if the deadline of its current job is later than t (i.e., $d_i(t) > t$).

Because the set of tasks changes during the schedule, the total utilization of the system does not remain constant. We therefore define the instantaneous total utilization of the system as being the total utilization of the tasks composing $\mathcal{A}(t)$ at a specific instant t:

Definition 6.9 (*Instantaneous total utilization*)

The instantaneous total utilization $U(t)$ at time t is the sum of the utilizations of the tasks which are active at time t. That is,

$$U(t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \mathcal{A}(t)} U_i$$

This instantaneous total utilization varies with the tasks that leave and join the system and hence provides a snapshot of the platform utilization at time t.

Whenever a new task τ_i arrives in a dynamic system, the real-time operating system must decide if τ_i can be allowed to actually join the task set. That is, the RTOS must be sure that there is no risk for τ_i to cause any deadline miss. In the following, we will assume that the RTOS accepts a new task in τ if and only if the total utilization of the task set τ after the task arrival will not be greater than the number of processors in the processing platform. Consequently, because the active task set $\mathcal{A}(t)$ is a subset of τ, we can assume that the instantaneous total utilization always remains smaller than or equal to the number of processors m.

Note that even though there is a difference between the task set τ and the active task set $\mathcal{A}(t)$ for the RTOS, the active task set can be viewed as the actual scheduled system from the scheduling algorithm perspective. The scheduling of a sporadic task set therefore becomes a particular case of the scheduling of a dynamic task system. Indeed, in a sporadic task set, a task leaves the system of active tasks whenever its current job reaches its deadline, and joins the system whenever it releases a new job.

Most often, a scheduling algorithm which is optimal for the scheduling of sporadic tasks with implicit deadlines provided that $U \leq m$, is also optimal for the scheduling of dynamic task systems under the condition that the instantaneous total utilization always remains smaller than or equal to the number of processors m. While this is quite easy to demonstrate for scheduling algorithms that do not reserve any time for the execution of future jobs but instead take scheduling decisions considering only the active jobs (e.g. PD^2 [Srinivasan and Anderson 2005b], LRE-TL [Funk 2010], DP-Wrap [Funk et al. 2011], ...), it is not so straightforward for scheduling algorithms such as U-EDF that

anticipate the computation needs of future jobs that might potentially arrive.

6.6.1 Reservation of Computation Time for Dynamic Tasks

Because we never know which tasks will be active in the future, there is not any reason anymore to reserve time for a particular task after its current deadline. The notion of future job reservation $\text{res}_{i,j}(t_1,t_2)$ which reserves execution time for a particular task τ_i on a particular processor π_j should therefore be replaced by a *reservation for dynamic future jobs* $\text{res\_dyn}_j(t_1,t_2)$ which reserves execution time on processor π_j for any new job that could be released by any task within the time interval $[t_1,t_2)$.

Before to formally define this reservation for dynamic future jobs, we first propose a similar quantity built upon the iterative version of U-EDF studied in the previous section for the scheduling of periodic tasks in a static system. Then, we determine the modifications that should be made to handle a dynamic task system.

Let $\text{res}_j(t,d_i(t))$ be the time reserved on processor π_j for the execution of any job that could be released by the *periodic* tasks belonging to a *static* system τ within the time interval $[t,d_i(t))$. Expression 6.5 developed in the previous section, implies that this time reservation can be computed iteratively by calculating the amount of time that must be reserved in each time slice bounded by two successive deadlines. That is, assuming that the tasks in τ are indexed in a decreasing priority order (i.e., $d_{x-1}(t) \leq d_x(t)$ for all $\tau_x \in \mathcal{A}(t)$), Expression 6.5 yields

$$\text{res}_j(t,d_i(t)) \stackrel{\text{def}}{=} \sum_{x=1}^{i} \text{res}_j(d_{x-1}(t),d_x(t)) \tag{6.6}$$

with

$$\text{res}_j(d_{x-1}(t),d_x(t)) \stackrel{\text{def}}{=} \left\{ \left\lfloor w_x(t) \right\rfloor_{j-1}^{j} - (j-1) \right\} \times (d_x(t) - d_{x-1}(t)) \tag{6.7}$$

where $d_0(t) \stackrel{\text{def}}{=} t$ and $w_x(t) \stackrel{\text{def}}{=} \sum_{\tau_k \in \text{hp}_x(t)} U_k$.

Hence, for each time slice bounded by two successive deadlines $d_{x-1}(t)$ and $d_x(t)$, we reserve execution time on processor π_j for all the tasks that have already reached their own deadlines (i.e., tasks in $\text{hp}_x(t)$), which could therefore release a new job within the interval $[d_{x-1}(t),d_x(t))$. Recall that this execution time reservation is proportionate to the total utilization of all the tasks that could release a job and therefore belong to $\text{hp}_x(t)$, i.e., it is proportionate to $w_x(t) \stackrel{\text{def}}{=} \sum_{\tau_k \in \text{hp}_x(t)} U_k$.

In a dynamic system however, new tasks can join the system and therefore release a new job as long as the instantaneous utilization remains smaller than the number of processors in the platform. Therefore, at any time $t' \geq t$ such that $d_{x-1}(t) \leq t' < d_x(t)$, new tasks with a total utilization at most equal to $m - U(t')$ could become active and release new jobs in the system. The time reserved for these potential new job arrivals should thus be proportionate to $m - U(t')$ instead of $\sum_{\tau_k \in \text{hp}_x(t)} U_k$. Hence, we should have the following expression instead of Expression 6.7:

$$\text{res\_dyn}_j(d_{x-1}(t), d_x(t)) \stackrel{\text{def}}{=} \left\{ \left[m - U(t') \right]_{j-1}^{j} - (j-1) \right\} \times (d_x(t) - d_{x-1}(t))$$

Because Definition 6.8 implies that if no new job arrives within $[t, t')$, the active task set $\mathcal{A}(t')$ at time t' must be equal to the active task set at time t minus the tasks that reached their deadlines between t and t', we have $\mathcal{A}(t') = \mathcal{A}(t) \setminus \text{hp}_x(t)$. Definition 6.9 thus yields $U(t') = U(t) + \sum_{\tau_k \in \text{hp}_x(t)} U_k$, thereby leading to the following definition of the dynamic future jobs reservation:

Definition 6.10 (*Reservation for dynamic future jobs*)

The reservation for dynamic future jobs $\text{res\_dyn}_j(t, d_i(t))$ *on processor* π_j *represents the computation time reserved on processor* π_j *for the execution of future jobs that could potentially be released by any task within the time interval extending from time* t *to a deadline* $d_i(t)$*. Formally, we have that*

$$\text{res\_dyn}_j(t, d_i(t)) \stackrel{\text{def}}{=} \sum_{x=1}^{i} \text{res\_dyn}_j(d_{x-1}(t), d_x(t))$$

with

$$\text{res\_dyn}_j(d_{x-1}(t), d_x(t)) \stackrel{\text{def}}{=} \left\{ \left[\text{w}_x^{\text{dyn}}(t) \right]_{j-1}^{j} - (j-1) \right\} \times (d_x(t) - d_{x-1}(t))$$

where $d_0(t) \stackrel{\text{def}}{=} t$ *and* $\text{w}_x^{\text{dyn}}(t) \stackrel{\text{def}}{=} \left(m - U(t)\right) + \sum_{\tau_k \in \text{hp}_x(t)} U_k$.

The following lemma proven in Appendix B states an important property of the reservation for dynamic future jobs.

Lemma 6.4

Let t *and* t' *be two instants such that* $t < t'$ *and let* τ_i *be an active task at time* t *such*

that $d_i(t) > t'$. Then, there is

$$\text{res\_dyn}_j(t, d_i(t)) = \text{res\_dyn}_j(t, t') + \text{res\_dyn}_j(t', d_i(t))$$

6.6.2 Pre-allocation for Dynamic Systems

As already said, the dynamic future jobs reservation has been introduced in order to replace the future job reservation made for particular tasks in the initial pre-allocation algorithm of U-EDF. The computation of the maximum allotment must therefore be updated and replaced by the maximum allotment for dynamic systems defined as follows:

Definition 6.11 (*Maximum allotment for dynamic systems*)

At any time-instant t, the maximum allotment for dynamic systems $\text{allot\_dyn}_{i,j}^{\max}(t)$ *of task τ_i on processor π_j is defined as the maximum amount of time that can be allocated on processor π_j to the active job of task τ_i in the time interval $[t, d_i(t))$ assuming that the system of tasks is dynamic. This upper-bound on* $\text{allot}_{i,j}(t)$ *is given by*

$$\text{allot\_dyn}_{i,j}^{\max}(t) \stackrel{\text{def}}{=} \big(d_i(t) - t\big) - \text{res\_dyn}_j(t, d_i(t)) - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

Algorithm 6.1 is therefore replaced by Algorithm 6.4 for the allocation of the tasks with U-EDF.

Algorithm 6.4: Pre-allocation with for dynamic task systems.

Input:
TaskList := list of the n tasks sorted by increasing absolute deadlines;
t := current time;

```
1 forall the τ_i ∈ TaskList do
2     for j := 1 to m do
3         allot_{i,j}(t) := min{allot_dyn^max_{i,j}(t) , ret_i(t) − Σ_{y<j} allot_{i,y}(t)};
4     end
5 end
```

Note that Algorithm 6.4 works for *both* the scheduling of static and dynamic systems. Indeed, a static task system is a particular case of a dynamic task system where no task ever leaves or joins the task set τ.

Algorithm 6.4 can also be written under an iterative form as in Section 6.5. The only variation with Algorithm 6.3 is that the iterative variable W should now be initialized to $\big(m - U(t)\big)$ instead of 0 at line 1 of Algorithm 6.3.

6.7 U-EDF: Scheduling in a Discrete-Time Environment

As introduced in Chapters 2 and 4, the particularity of a discrete-time environment resides in the fact that the task parameters (i.e., the worst-case execution times C_i, the deadlines D_i and the minimum inter-arrival times T_i) and the task execution times must be natural multiples of a system time unit. Therefore, as we can reasonably assume that the task parameters are actually defined as natural multiples of the system time unit, to propose a discrete-time scheduling algorithm, we must yet ensure that U-EDF schedules the execution of the tasks on each processor for integer multiples of the system time unit.

Under U-EDF, the tasks are executed with respect to their respective allotments on each processor. If all the task allotments are integers and the jobs released by these same tasks have integral execution times (note that we say execution times and not necessarily worst-case execution times), then all tasks should be scheduled for natural multiples of the system time unit.

Our goal in this section, is therefore to make sure that the allotment computed with Algorithm 6.4 for any task τ_i on any processor π_j will always be an integer.

Recall that the task allotments are computed successively in Algorithm 6.4 with two imbricated "for" loops. The first loop iterates on the tasks in a decreasing priority order, and the second loop iterates on the processors in an increasing index order. Furthermore, the allotment $\text{allot}_{i,j}(t)$ computed for a task τ_i on processor π_j depends on the allotments of the tasks with higher priorities and the allotments of τ_i on processors with lower indices. Hence, we will analyze the computation of the allotment of τ_i on π_j assuming that the allotment $\text{allot}_{x,j}(t)$ previously computed for any task with a higher priority (i.e., for tasks $\tau_x \in \text{hp}_i(t)$) and the allotment $\text{allot}_{i,y}(t)$ computed for τ_i on processors with smaller indices (i.e., on processors π_y such that $y < j$) are all integers. This assumption is obviously true for the first allotment computed by Algorithm 6.4, i.e., the allotment of the highest priority task on the first processor. Then, this assumption becomes inductively true for the next tasks and processors.

These allotments are computed with line 3 of Algorithm 6.4 repeated hereafter

$$\text{allot}_{i,j}(t) := \min \left\{ \text{allot\_dyn}^{\max}_{i,j}(t) \ , \ \text{ret}_i(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t) \right\}$$

where (Definition 6.11)

$$\text{allot\_dyn}^{\max}_{i,j}(t) = \big(d_i(t) - t\big) - \text{res\_dyn}_j(t, d_i(t)) - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

We now analyze each term in these two expressions so as to identify the reasons that could cause the allotment $\text{allot}_{i,j}(t)$ not to be a natural number.

- t. Similarly to our reasoning on BF^2 in Chapter 4, we can reasonably assume that all jobs are released on system ticks. Since the pre-allocation algorithm (Algorithm 6.4) is called only when new jobs arrive, it holds that the time t at which the allotment $\text{allot}_{i,j}(t)$ is computed is a natural.

- $d_i(t)$. Because the job arrivals are synchronized on the system ticks, we have that the arrival time $a_i(t)$ of any active job of a task τ_i is a natural number. Then, because any task deadline D_i is a natural multiple of the system time unit, the deadline $d_i(t)$ of any task τ_i which is equal to $a_i(t) + D_i$, should also be synchronized on the system ticks. Consequently, $d_i(t)$ is a natural number.

- $\text{ret}_i(t)$. Since the worst-case execution time C_i of each task τ_i is a natural, if all tasks have been executed for integer multiples of the system time unit until time t then their (worst-case) remaining execution time $\text{ret}_i(t)$ at time t must also be an integer multiple of this system time unit.

- $\sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t)$ and $\sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$. We assumed that the allotments of all tasks with a higher priority and the allotments of τ_i on processor with smaller indices are integers. These two sums must therefore be integers too.

The only remaining term, which could cause the computed allotment $\text{allot}_{i,j}(t)$ *not* to be an integer is $\text{res\_dyn}_j(t, d_i(t))$, i.e., the reservation for dynamic future jobs.

This quantity represents the time reserved for the execution of future jobs that could potentially arrive within $[t, d_i(t))$. However, even though it is not proven yet, reserving an amount of time equal to the floor value of $\text{res\_dyn}_j(t, d_i(t))$ (i.e., $\lfloor \text{res\_dyn}_j(t, d_i(t)) \rfloor$)

should be sufficient to ensure the optimality of U-EDF. In this case, the floor operator makes certain that all terms present in the formulae computing the maximum allocation are all integers, thereby implying that $\text{allot}_{i,j}(t)$ will be an integer either.

We therefore define the reservation for dynamic future jobs in a discrete-time environment as follows

Definition 6.12 (*Reservation for dynamic future jobs in a discrete-time environment*)

The reservation $\text{res\_disc}_j(t, d_i(t))$ *for dynamic future jobs in a discrete-time environment on processor* π_j *represents the computation time reserved on processor* π_j *for the execution of future jobs that could potentially be released by any task within the time interval extending from time* t *to a deadline* $d_i(t)$*, assuming that the scheduling is based on a system tick. Formally, we have that*

$$\text{res\_disc}_j(t, d_i(t)) \stackrel{\text{def}}{=} \left\lfloor \text{res\_dyn}_j(t, d_i(t)) \right\rfloor$$

6.7.1 Pre-allocation for Dynamic Discrete-Time Systems

The computation of the maximum allotment must therefore be updated and replaced by the maximum allotment for dynamic systems scheduled in a discrete time-environment defined as follows:

Definition 6.13 (*Maximum allotment for dynamic systems in a discrete-time environment*)

At any time-instant t*, the maximum allotment for dynamic systems in a discrete-time environment* $\text{allot\_disc}_{i,j}^{\max}(t)$ *of task* τ_i *on processor* π_j *is defined as the maximum amount of time that can be allocated on processor* π_j *to the active job of task* τ_i *in the time interval* $[t, d_i(t))$ *assuming that the system of tasks is dynamic and the allotment must be a natural multiple of a system time-unit. This upper-bound on* $\text{allot}_{i,j}(t)$ *is given by*

$$\text{allot\_disc}_{i,j}^{\max}(t) \stackrel{\text{def}}{=} \big(d_i(t) - t\big) - \text{res\_disc}_j(t, d_i(t)) - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

Algorithm 6.1 is then replaced by Algorithm 6.5 for the allocation of the tasks with U-EDF.

Algorithm 6.5: Pre-allocation with for dynamic task systems in a discrete-time environment.

Input:
TaskList := list of the n tasks sorted by increasing absolute deadlines;
t := current time;

1 **forall the** $\tau_i \in$ TaskList **do**
2 **for** $j := 1$ **to** m **do**
3 $\text{allot}_{i,j}(t) := \min\{\text{allot\_disc}^{\max}_{i,j}(t)\ ,\ \text{ret}_i(t) - \sum_{y<j} \text{allot}_{i,y}(t)\}$;
4 **end**
5 **end**

Like Algorithm 6.1 and Algorithm 6.4, Algorithm 6.5 can be written in an iterative manner. Like for Algorithm 6.4, we must have $W = \big(m - U(t)\big)$ instead of $W = 0$ at line 1 of Algorithm 6.3. Furthermore, lines 11 and 12 of Algorithm 6.3 must be replaced by

$$\text{allot\_disc}^{\max}_{i,j}(t) := \big(d_i(t) - t\big) - \text{ALLOT}_j - \big\lfloor \text{RES}_j \big\rfloor - \text{PREV}_i;$$
$$\text{allot}_{i,j}(t) := \min\{\text{allot\_disc}^{\max}_{i,j}(t)\ ,\ \text{ret}_i - \text{PREV}_i\};$$

6.8 U-EDF: Optimality Proofs

In this section, we essentially prove that U-EDF is optimal for the scheduling of *sporadic* tasks with *implicit deadlines* in a *dynamic* task system scheduled in a *continuous-time* multiprocessor environment, in that sense that it always meets all deadlines when $U(t) \leq m$ and $U_i \leq 1$ for every $\tau_i \in \tau$ at every time t. Then, we extend this result and prove that sporadic tasks with *unconstrained deadlines* are schedulable by U-EDF provided that the instantaneous density $\delta(t)$ is never greater than m and the density of each task τ_i is not greater than 1.

Because all tasks have to be reassigned when a new job is released in the system (see Algorithm 6.2), the proof of optimality (Theorem 6.1) is made by induction on the job arrival times. The induction is the following: assuming that U-EDF is valid at the arrival time t_r of job J_r, running EDF-D from t_r to the arrival of the next job J_{r+1} at time t_{r+1}, we prove that (i) the U-EDF assignment is still valid at t_{r+1} (Lemmas 6.6 to 6.10) and (ii) all jobs with a deadline before or at t_{r+1} meet their deadline (Lemma 6.5).

In Lemma 6.5, we prove that all tasks respect their deadlines between two job arrivals assuming that the assignment made by U-EDF was valid. Note that we *do not* assume

that each job is executed for its worst-case execution time in this lemma. Hence, if a job does not need to execute for its whole worst-case execution time, then all task deadlines are still respected. The jobs will just finish earlier than initially expected.

Lemma 6.5

Let t_a be the very first arrival time of a job after t. Let us assume that we executed the pre-allocation algorithm of U-EDF (Algorithm 6.1) at time t. If the U-EDF assignment is valid at time t and EDF-D is applied within $[t, t_a)$ then all jobs with a deadline such that $d_i(t) \leq t_a$ meet their deadlines without intra-job parallelism.

Proof:

From Lemma 6.3, intra-job parallelism can never occur in a time interval when EDF-D is used to schedule the tasks within this time interval. It therefore remains to prove that all jobs with a deadline at or before t_a respect their deadlines applying EDF-D. Let $\text{rem}_i(t)$ be the time τ_i must still execute before its deadline. This actual remaining execution time of τ_i may be smaller than or equal to the worst-case remaining execution time $\text{ret}_i(t)$ of τ_i. We must prove in this lemma that for all tasks such that $d_i(t) \leq t_a$

$$\sum_{j=1}^{m} \text{exec}_{i,j}(t, d_i(t)) = \text{rem}_i(t) \leq \text{ret}_i(t) \tag{6.8}$$

where $\text{exec}_{i,j}(t, d_i(t))$ is the time that the task τ_i has been running on processor π_j within the time interval $[t, d_i(t))$. That is, we must prove that all tasks with a deadline no later than t_a are executed for their remaining execution time.

Since the U-EDF assignment is valid at time t, Definition 6.4 imposes that

$$\forall \tau_i \in \tau : \sum_{j=1}^{m} \text{allot}_{i,j}(t) = \text{ret}_i(t) \tag{6.9}$$

Analyzing Expressions 6.8 and 6.9 altogether, we deduce that each task τ_i with a deadline $d_i(t) \leq t_a$ respects its deadline, if the following expression holds

$$\forall \pi_j : \text{exec}_{i,j}(t, d_i(t)) = \text{rem}_{i,j}(t) \leq \text{allot}_{i,j}(t)$$

where $\text{rem}_{i,j}(t)$ is the time τ_i should execute on π_j to respect its deadline.

By contradiction, let us assume that the execution of the current job of τ_i on

processor π_j is not finished before the deadline of τ_i at time $d_i(t) \leq t_a$. That is,

$$\text{exec}_{i,j}(t, d_i(t)) < \text{rem}_{i,j}(t)$$

Property 6.1 says that if a task τ_i with $\text{allot}_{i,j}(t') > 0$ is not running on processor π_j at time $t' \in [t, d_i(t))$, then either a task with a higher priority than τ_i (i.e., a task in $\text{hp}_i(t)$) is executing on π_j, or τ_i is running on a processor with a smaller index at time t' (i.e., on a processor π_y such that $y < j$). Since τ_i executed for less than $\text{rem}_{i,j}(t)$ time units in the time interval $[t, d_i(t))$, it must therefore hold that

$$\text{rem}_{i,j}(t) + \sum_{\tau_x \in \text{hp}_i(t)} \text{exec}_{x,j}(t, d_i(t)) + \sum_{y<j} \text{exec}_{i,y}(t, d_i(t)) > (d_i(t) - t)$$

and because $\text{rem}_{i,j}(t) \leq \text{allot}_{i,j}(t)$

$$\text{allot}_{i,j}(t) + \sum_{\tau_x \in \text{hp}_i(t)} \text{exec}_{x,j}(t, d_i(t)) + \sum_{y<j} \text{exec}_{i,y}(t, d_i(t)) > (d_i(t) - t) \tag{6.10}$$

Since $d_i(t)$ is an instant at or before t_a and because t_a corresponds to the very first job arrival after t, only the current active jobs of the tasks in $\text{hp}_i(t)$ might have been executed within $[t, d_i(t))$. Hence, each task τ_x could have been executed for at most its current job allotment on processor π_j. Therefore,

$$\sum_{y<j} \text{exec}_{i,y}(t, d_i(t)) \leq \sum_{y<j} \text{allot}_{i,y}(t) \tag{6.11}$$

and

$$\sum_{\tau_x \in \text{hp}_i(t)} \text{exec}_{x,j}(t, d_i(t)) \leq \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) \tag{6.12}$$

Injecting Expressions 6.11 and 6.12 in Expression 6.10, we get

$$\text{allot}_{i,j}(t) + \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) + \sum_{y<j} \text{allot}_{i,y}(t) > (d_i(t) - t)$$

and rearranging the terms

$$\text{allot}_{i,j}(t) > (d_i(t) - t) - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

Finally using Definitions 6.11 and 6.13, it respectively results that

$$\text{allot}_{i,j}(t) > \text{allot\_dyn}_{i,j}^{\max}(t)$$

and

$$\text{allot}_{i,j}(t) > \text{allot\_disc}_{i,j}^{\max}(t)$$

which is a contradiction with line 3 of the U-EDF pre-allocation algorithm for dynamic system in a continuous and discrete-time environment (Algorithms 6.4 and 6.5, respectively). ■

The proof of the optimality is made by induction. Lemma 6.6 first proves that the assignment is obviously valid at the start of the schedule assuming that the tasks have not released any job yet. Then, Lemmas 6.7 to 6.10 prove that $\sum_{j=1}^{m} \text{allot}_{i,j}(t) = \text{ret}_i(t)$ is still respected at every new job arrival. Finally, Theorem 6.1 states the optimality of U-EDF.

Lemma 6.6

Let t_0 be the very beginning of the schedule, i.e., when no job has been released yet. For every $\tau_i \in \tau$ we have

$$\sum_{j=1}^{m} \text{allot}_{i,j}(t_0) = \text{ret}_i(t_0)$$

Proof:

Since at time t_0 no job has been released yet, we have $\text{ret}_i(t_0) = 0$ for every $\tau_i \in \tau$. Consequently, Algorithm 6.1 does not allocate any time unit for the execution of these tasks, i.e., $\text{allot}_{i,j}(t_0) = 0$, $\forall i, j$. Hence, it obviously holds that $\sum_{j=1}^{m} \text{allot}_{i,j}(t_0) = \text{ret}_i(t_0)$. ■

Now that the basic statement has been made, we will prove through Lemmas 6.7 to 6.10 that the condition $\sum_{j=1}^{m} \text{allot}_{i,j}(t_a) = \text{ret}_i(t_a)$ is satisfied at any time $t_a \geq t_0$ corresponding to a new job arrival.

We first define some new notations. Let J_a of task τ_a denote the job released at time t_a. According to Algorithm 6.2, all tasks have to be reallocated on the processors on J_a's release (i.e., at time t_a) using Algorithm 6.1. Since the set of currently active jobs is modified due to the release of J_a at time t_a, we distinguish between two instants t_a^- and t_a^+ defined as follows

- t_a^- considers the task set just *before* the release of J_a.

- t_a^+ considers the system right *after* the release of J_a and the execution of Algorithm 6.1.

It is essential to understand that from a theoretical point of view, these two instants are *equal to* t_a but correspond to two different states of the system. Hence, Lemma 6.7 provides information on the system until time t_a^-. Then, Lemmas 6.8 to 6.10 prove that the expression $\sum_{j=1}^{m} \text{allot}_{i,j}(t_a) = \text{ret}_i(t_a)$ is satisfied at the instant t_a^+.

Lemma 6.7

Let t_a be the instant of the next arrival of a job after t and let t' be any time-instant such that $t < t' \leq t_a^-$. If we have for every task τ_i and every processor π_j

$$\text{allot}_{i,j}(t) \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t) & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t) & \text{if the time is discrete} \end{cases}$$

and tasks are scheduled using EDF-D in $[t,t')$, then it holds for every task τ_i and every processor π_j

$$\text{allot}_{i,j}(t') \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t') & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t') & \text{if the time is discrete} \end{cases}$$

Proof:

For all tasks τ_i that have completed their execution at time t', we have $\text{ret}_i(t') = 0$ and $\text{allot}_{i,j}(t') = 0$ for all π_j. Consequently, the condition $\text{allot}_{i,j}(t') \leq \text{allot\_dyn}_{i,j}^{\max}(t')$ (or $\text{allot}_{i,j}(t') \leq \text{allot\_disc}_{i,j}^{\max}(t')$) is obviously satisfied for those tasks and the remainder of the proof considers only the tasks which have not completed their execution at time t', i.e., the tasks τ_i with $\text{ret}_i(t') > 0$. Note that the active job of each such task τ_i cannot have a deadline $d_i(t)$ before time t'. Otherwise, since $\text{ret}_i(t') > 0$, it would mean that τ_i has missed its deadline, leading to a direct contradiction with Lemma 6.5. That is, we have $d_i(t) > t'$ and thus $d_i(t) = d_i(t')$. From this point onward, we will therefore use the notation d_i instead of $d_i(t)$ or $d_i(t')$ for the sake of conciseness.

Let us denote by $\text{exec}_{i,j}(t,t')$ the time during which τ_i has been running on pro-

cessor π_j in the time interval $[t,t')$. We have from Expression 6.1 on page 171

$$\text{allot}_{i,j}(t) \stackrel{\text{def}}{=} \text{allot}_{i,j}(t') + \text{exec}_{i,j}(t,t') \tag{6.13}$$

By assumption, we have $\forall \tau_i, \pi_j$

$$\text{allot}_{i,j}(t) \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t) & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t) & \text{if the time is discrete} \end{cases}$$

which gives, together with Expression 6.13,

$$\text{allot}_{i,j}(t') + \text{exec}_{i,j}(t,t') \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t) & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t) & \text{if the time is discrete} \end{cases}$$

Then, using either Definition 6.11 or Definition 6.13, this last expression can be rewritten as

$$\begin{aligned} \text{allot}_{i,j}(t') + \text{exec}_{i,j}(t,t') \leq (d_i - t) - \text{res\_d}_j(t,d_i) \\ - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t) \end{aligned} \tag{6.14}$$

where $\text{res\_d}_j(t,d_i)$ denotes $\text{res\_dyn}_j(t,d_i)$ if we are in a continuous-time environment or $\text{res\_disc}_j(t,d_i)$ if we are in a discrete-time system.

By replacing the term $\sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t)$ of Expression 6.14 with Expression 6.13, we get

$$\begin{aligned} \text{allot}_{i,j}(t') + \text{exec}_{i,j}(t,t') \leq (d_i - t) - \text{res\_d}_j(t,d_i) - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') \\ - \sum_{\tau_x \in \text{hp}_i(t)} \text{exec}_{x,j}(t,t') - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t) \end{aligned}$$

Then, subtracting $\text{exec}_{i,j}(t,t')$ to both sides, we obtain

$$\begin{aligned} \text{allot}_{i,j}(t') \leq (d_i - t) - \text{res\_d}_j(t,d_i) - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') \\ - \sum_{\tau_x \in \text{hp}_i(t) \cup \{\tau_i\}} \text{exec}_{x,j}(t,t') - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t) \end{aligned} \tag{6.15}$$

Let us now focus on the term $\sum_{\tau_x \in \text{hp}_i(t) \cup \{\tau_i\}} \text{exec}_{x,j}(t,t')$ in the above inequality. Intuitively, this term expresses the amount of time during which the active job of the tasks in $\text{hp}_i(t) \cup \{\tau_i\}$ are executed on processor π_j in the interval $[t,t')$. Obviously, this amount of time cannot be greater than the time elapsed between instants t and t', i.e., we cannot use processor π_j during more than $(t'-t)$ time units within the interval $[t,t')$. Hence, two cases may arise:

Case 1: $\sum_{\tau_x \in \text{hp}_i(t) \cup \{\tau_i\}} \text{exec}_{x,j}(t,t') = (t'-t)$
Injecting this equality in Expression 6.15, we get

$$\text{allot}_{i,j}(t') \leq (d_i - t') - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') - \text{res\_d}_j(t,d_i) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t)$$

Note that the amount of time $\text{allot}_{i,y}(t)$ allotted to the active job of τ_i on a processor π_y decreases as τ_i executes on π_y (Expression 6.13). Hence, $\text{allot}_{i,y}(t') \leq \text{allot}_{i,y}(t)$ and it consequently results that

$$\text{allot}_{i,j}(t') \leq (d_i - t') - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') - \text{res\_d}_j(t,d_i) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t')$$

Similarly, Lemma 6.4 implies that

$$\text{res\_dyn}_j(t,d_i) > \text{res\_dyn}_j(t',d_i) \tag{6.16}$$

Hence by the property of the floor operator, we have that $\lfloor \text{res\_dyn}_j(t,d_i) \rfloor \geq \lfloor \text{res\_dyn}_j(t',d_i) \rfloor$ and by Definition 6.12

$$\text{res\_disc}_j(t,d_i) \geq \text{res\_disc}_j(t',d_i) \tag{6.17}$$

Hence, combining Expressions 6.16 and 6.17, there is

$$\text{res\_d}_j(t,d_i) \geq \text{res\_d}_j(t',d_i) \tag{6.18}$$

which leads to

$$\text{allot}_{i,j}(t') \leq (d_i - t') - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') - \text{res\_d}_j(t',d_i) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t')$$

Finally, using Definition 6.11 and 6.13, we obtain that

$$\text{allot}_{i,j}(t') \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t') & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t') & \text{if the time is discrete} \end{cases}$$

Case 2: $\sum_{\tau_x \in \text{hp}_i(t) \cup \{\tau_i\}} \text{exec}_{x,j}(t,t') < (t'-t)$

In this case, there are some time in $[t,t')$ during which processor π_j is either idle or it executes at least one task with a lower priority than τ_i (i.e., a task from the subset $\text{lp}_i(t)$). There can be only two reasons for that to happen by using EDF-D (see Property 6.1):

a) The active job of τ_i at time t completed its execution on π_j during the time interval $[t,t')$. Hence $\text{allot}_{i,j}(t') = 0$ and it obviously holds that

$$\text{allot}_{i,j}(t') \leq \text{allot\_dyn}_{i,j}^{\max}(t')$$

and

$$\text{allot}_{i,j}(t') \leq \text{allot\_disc}_{i,j}^{\max}(t')$$

b) The execution of the active job of τ_i at time t was delayed on π_j because, in the time interval $[t,t')$, it was executed on at least one other processor π_y such that $y < j$ (see Property 6.1). Let $\text{delay}_{i,j}(t,t')$ be the time during which the execution of τ_i was delayed on processor π_j, we have

$$\sum_{\tau_x \in \text{hp}_i(t) \cup \{\tau_i\}} \text{exec}_{x,j}(t,t') + \text{delay}_{i,j}(t,t') = (t'-t)$$

Specifically, the amount of time by which the execution of τ_i is delayed on π_j is upper-bounded by $\sum_{y=1}^{j-1} \text{exec}_{i,y}(t,t')$ (i.e., the time during which τ_i executed on processors with lower indices). That is, $\text{delay}_{i,j}(t,t') \leq \sum_{y=1}^{j-1} \text{exec}_{i,y}(t,t')$ and we therefore get

$$\sum_{\tau_x \in \text{hp}_i(t) \cup \{\tau_i\}} \text{exec}_{x,j}(t,t') + \sum_{y=1}^{j-1} \text{exec}_{i,y}(t,t') \geq (t'-t) \tag{6.19}$$

Moreover, it holds from Expression 6.13 that

$$\sum_{y=1}^{j-1} \text{exec}_{i,y}(t,t') = \sum_{y=1}^{j-1} \left[\text{allot}_{i,y}(t) - \text{allot}_{i,y}(t')\right] \tag{6.20}$$

Using Expressions 6.15, 6.19 and 6.20 altogether, we finally obtain

$$\text{allot}_{i,j}(t') \leq (d_i - t') - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') - \text{res\_d}_j(t,d_i) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t')$$

Because $\text{res\_d}_j(t,d_i) \geq \text{res\_d}_j(t',d_i)$ (Expression 6.18), there is

$$\text{allot}_{i,j}(t') \leq (d_i - t') - \sum_{\tau_x \in \text{hp}_i(t)} \text{allot}_{x,j}(t') - \text{res\_d}_j(t',d_i) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t')$$

and using Definition 6.11 and Definition 6.13, it results that

$$\text{allot}_{i,j}(t') \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t') & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t') & \text{if the time is discrete} \end{cases}$$

Therefore, we get in all cases $\text{allot}_{i,j}(t') \leq \text{allot\_dyn}_{i,j}^{\max}(t')$ if the time is continuous or $\text{allot}_{i,j}(t') \leq \text{allot\_disc}_{i,j}^{\max}(t')$ if the time is discrete. Hence, this states the lemma for continuous and discrete-time systems, respectively. ■

From this point onward, we will denote by $\text{res\_dyn}_j^-(t_1,t_2)$ (with $t_a \leq t_1 < t_2$) the time that was reserved on processor π_j within the interval $[t_1,t_2)$, before that task τ_a actually release a job at time t_a. Similarly, $\text{res\_dyn}_j^+(t_1,t_2)$ will denote the time that is reserved on processor π_j within the interval $[t_1,t_2)$, after that task τ_a becomes active at time t_a^+.

Furthermore, for the sake of readability, we will assume from this point onward that the tasks are indexed according to their priorities at time t_a^+. Hence, if $r < s$ then τ_r has a higher priority than τ_s at time t_a^+, implying that $\tau_r \in \text{hp}_s(t_a^+)$ and $d_r(t_a^+) \leq d_s(t_a^+)$.

Lemma 6.8

Let $t_a \geq 0$ be any instant in the scheduling of τ at which a new job is released and let m be the number of processors in the platform. If the task τ_a is the only task releasing a new job at time t_a, we have for all $j \leq m$ and all tasks $\tau_x \in \mathcal{A}(t_a^+)$:

$$\begin{cases} \displaystyle\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) \geq \sum_{p=1}^{j} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) & \text{if } d_x(t_a^+) \leq d_a(t_a^+) \\ \qquad\qquad - U_a \times (d_x(t_a^+) - d_{x-1}(t_a^+)) & \\ \displaystyle\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) & \text{otherwise} \end{cases}$$

with $d_0(t_a^+) \stackrel{\text{def}}{=} t_a$.

Proof:

Since τ_a is the only task releasing a job a time t_a, the task τ_a is the only tasks becoming active between t_a^- and t_a^+. Hence, we have that $\mathcal{A}(t_a^+) = \mathcal{A}(t_a^-) \cup \tau_a$ (from Definition 6.8), thereby implying (from Definition 6.9)

$$U(t_a^+) = U(t_a^-) + U_a \tag{6.21}$$

Furthermore, since τ_a is inactive at time t_a^- and because the higher priority task set $\text{hp}_x(t_a^-)$ computed at time t_a^- for any task τ_x contains the *active* tasks with a higher priority than τ_x (Definition 6.2), it holds for every task $\tau_x \in \tau$ that $\tau_a \notin \text{hp}_x(t_a^-)$. On the other hand, because τ_a becomes active at time t_a^+ and because τ_a as a lower priority than any task τ_x such that $d_x(t_a^+) \leq d_a(t_a^+)$ but a higher priority than every task τ_x such that $d_x(t_a^+) > d_a(t_a^+)$, we have that

$$\begin{cases} \tau_a \notin \text{hp}_x(t_a^+) & \text{if } d_x(t_a^+) \leq d_a(t_a^+) \\ \tau_a \in \text{hp}_x(t_a^+) & \text{otherwise} \end{cases}$$

Consequently, because τ_a is the only task that joins the system between t_a^- and t_a^+, we get

$$\text{hp}_x(t_a^+) = \begin{cases} \text{hp}_x(t_a^-) & \text{if } d_x(t_a^+) \leq d_a(t_a^+) \\ \text{hp}_x(t_a^-) \cup \tau_a & \text{otherwise} \end{cases} \tag{6.22}$$

Moreover, Definition 6.10 applied at time t_a^+ provides

$$\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \left(\left[\left(m - U(t_a^+)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^+)} U_k \right]_{p-1}^{p} - (p-1) \right) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) \tag{6.23}$$

(where $d_0(t_a^+) = t_a$).

We now study two different cases:

- $d_x(t_a^+) > d_a(t_a^+)$. Using Expressions 6.21 and 6.22 with Expression 6.23, we get

$$\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \Big(\Big[\left(m - U(t_a^-) - U_a\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k + U_a \Big]_{p-1}^{p} - (p-1) \Big) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

Simplifying

$$\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \Big(\Big[\left(m - U(t_a^-)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \Big]_{p-1}^{p} - (p-1) \Big) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

and using Definition 6.10 applied at time t_a^-, it leads to

$$\sum_{j=p}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) \tag{6.24}$$

- $d_x(t_a^+) \leq d_a(t_a^+)$. Using Expressions 6.21 and 6.22 with Expression 6.23, we get

$$\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \Big(\Big[\left(m - U(t_a^-) - U_a\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \Big]_{p-1}^{p} - (p-1) \Big) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

Applying Lemma 6.2, it holds that

$$\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \left(\left[\left(m - U(t_a^-) - U_a\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \right]_0^j \right) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

Because $[y]_a^b \stackrel{\text{def}}{=} \max\{a, \min\{b, y\}\}$ and because

$$\max\{a, \min\{b, y-z\}\} \geq \max\{a, \min\{b, y\}\} - z$$

the previous expression yields

$$\begin{aligned}\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) \geq & \left[\left(m - U(t_a^-)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \right]_0^j \\ & \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) \\ & - U_a \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)\end{aligned}$$

First, reusing Lemma 6.2 and then applying Definition 6.10 at time t_a^-, we get

$$\begin{aligned}\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) &\geq \sum_{p=1}^{j} \left(\left[\left(m - U(t_a^-)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \right]_{p-1}^{p} - (p-1) \right) \\ &\quad \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) - U_a \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) \\ &\geq \sum_{p=1}^{j} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) \qquad (6.25) \\ &\quad - U_a \times (d_i(t_a) - d_{i-1}(t_a))\end{aligned}$$

Expressions 6.24 and 6.25 prove the lemma. ■

Lemma 6.9

Let $t_a \geq 0$ be any instant in the scheduling of τ at which a new job is released and let m be the number of processors in the platform. If the task τ_a is the only task releasing a new job at time t_a, we have for all tasks $\tau_x \in \mathcal{A}(t_a^+)$:

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \begin{cases} \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) & \text{if } d_x(t_a^+) \leq d_a(t_a^+) \\ \quad - U_a \times (d_x(t_a^+) - d_{x-1}(t_a^+)) & \\ \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) & \text{otherwise} \end{cases}$$

(where $d_0(t_a^+) \stackrel{\text{def}}{=} t_a$) provided that $U(t_a^+) \leq m$.

Proof:

Using the same argumentation as for Lemma 6.8, we get that $\mathcal{A}(t_a^+) = \mathcal{A}(t_a^-) \cup \tau_a$ (from Definition 6.8), thereby implying (from Definition 6.9)

$$U(t_a^+) = U(t_a^-) + U_a \tag{6.26}$$

and as already stated in the previous lemma, because τ_a is the only task becoming active at time t_a^+, we have that

$$\text{hp}_x(t_a^+) = \begin{cases} \text{hp}_x(t_a^-) & \text{if } d_x(t_a^+) \leq d_a(t_a^+) \\ \text{hp}_x(t_a^-) \cup \tau_a & \text{otherwise} \end{cases} \tag{6.27}$$

We now study the two same cases than in Lemma 6.8:

- $d_x(t_a^+) > d_a(t_a^+)$. In Lemma 6.8, we proved for $d_x(t_a^+) > d_a(t_a^+)$ that

$$\sum_{p=1}^{j} \text{res\_dyn}_p^+(d_{i-1}(t_a^+), d_i(t_a^+)) = \sum_{p=1}^{j} \text{res\_dyn}_p^-(d_{i-1}(t_a^+), d_i(t_a^+))$$

Hence, we get for $j = m$,

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{i-1}(t_a^+), d_i(t_a^+)) = \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{i-1}(t_a^+), d_i(t_a^+)) \tag{6.28}$$

- $d_x(t_a^+) \leq d_a(t_a^+)$. Definition 6.10 applied at time t_a^+ provides

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{j} \left(\left[\left(m - U(t_a^+)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^+)} U_k \right]_{p-1}^{p} - (p-1) \right) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) \quad (6.29)$$

(where $d_0(t_a^+) = t_a$). Since by assumption $U(t_a^+) = \sum_{\tau_k \in \mathcal{A}(t_a^+)} U_k \leq m$ and because $\text{hp}_x(t_a^+)$ is a subset of $\mathcal{A}(t_a^+)$ (i.e., by Definition 6.2, $\text{hp}_x(t_a^+)$ is composed of *active* tasks at time t_a^+), there is

$$0 \leq \left(\left(m - U(t_a^+)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^+)} U_k \right) \leq m$$

Therefore, using Lemma 6.2 on Expression 6.29, it holds that

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \left(\left(m - U(t_a^+)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^+)} U_k \right) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

Injecting Expressions 6.26 and 6.27 in this last equation, we get

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \left(\left(m - U(t_a^-)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \right) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) - U_a \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

Since $U(t_a^-) < U(t_a^+)$ (Expression 6.26), we have that

$$U(t_a^-) = \sum_{\tau_k \in \mathcal{A}(t_a^-)} U_k \leq U(t_a^+) \leq m$$

and because $\text{hp}_x(t_a^-)$ is a subset of $\mathcal{A}(t_a^-)$ (i.e., by Definition 6.2, $\text{hp}_x(t_a^-)$ is composed of *active* tasks at time t_a^-), we get that

$$0 \leq \left(\left(m - U(t_a^-)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \right) \leq m$$

Thus, applying Lemma 6.2

$$\sum_{p=1}^{m} \text{res\_dyn}_p(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{m} \left(\left[\left(m - U(t_a^-)\right) + \sum_{\tau_k \in \text{hp}_x(t_a^-)} U_k \right]_{p-1}^{p} - (p-1) \right) \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) - U_a \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right)$$

Finally, using Definition 6.10, we get

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{x-1}(t_a^+), d_x(t_a^+)) = \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{x-1}(t_a^+), d_x(t_a^+)) - U_a \times \left(d_x(t_a^+) - d_{x-1}(t_a^+)\right) \tag{6.30}$$

Expressions 6.28 and 6.30 prove the lemma. ■

Lemma 6.10

Let $t \geq 0$ be any time-instant in the scheduling of τ and let $t_a \geq t$ be the first instant after (or at) time t at which a job is released. If the assignment is valid at time t for every $\tau_i \in \tau$ and EDF-D is used to schedule tasks within $[t, t_a)$, then assuming that only one job is released at this instant t_a, we have for all tasks $\tau_i \in \mathcal{A}(t_a^+)$:

$$\sum_{j=1}^{m} \text{allot}_{i,j}(t_a^+) = \text{ret}_i(t_a^+)$$

(i.e., there is a valid assignment after the reallocation of tasks at time t_a) provided that $U(t_a^+) \leq m$ and $U_i \leq 1, \forall \tau_i \in \mathcal{A}(t_a^+)$.

Proof:

Let τ_a denote the (only) task releasing a job at time t_a. We already proved in Lemma 6.7, that at time t_a^- (i.e., just before the tasks reallocation), we have $\forall i, j$:

$$\text{allot}_{i,j}(t_a^-) \leq \begin{cases} \text{allot\_dyn}_{i,j}^{\max}(t_a^-) & \text{if the time is continuous} \\ \text{allot\_disc}_{i,j}^{\max}(t_a^-) & \text{if the time is discrete} \end{cases} \tag{6.31}$$

Furthermore, the assignment was valid at time t and because Algorithm 6.4 does

not allow $\sum_{j=1}^{m} \text{allot}_{i,j}(t)$ to be greater than $\text{ret}_i(t)$, it implies that for all $\tau_i \in \mathcal{A}(t)$

$$\begin{aligned} &\sum_{j=1}^{m} \text{allot}_{i,j}(t) = \text{ret}_i(t) \\ \Leftrightarrow &\sum_{j=1}^{m} (\text{allot}_{i,j}(t) - \text{exec}_{i,j}(t, t_a^-)) = \text{ret}_i(t) - \sum_{j=1}^{m} \text{exec}_{i,j}(t, t_a^-) \end{aligned}$$

Hence using Expression 6.1 and because the remaining execution time of τ_i at time t_a^- is equal to its remaining execution time at time t minus the time it was executed on all the processors within $[t, t_a^-)$, we get for every task $\tau_i in \mathcal{A}(t)$

$$\sum_{j=1}^{m} \text{allot}_{i,j}(t_a^-) = \text{ret}_i(t_a^-) \tag{6.32}$$

Since by assumption, τ_a is the only task releasing a new job at time t_a, there is $\mathcal{A}(t_a) \subseteq \mathcal{A}(t) \cup \tau_a$ and the deadlines and the remaining execution times of tasks $\tau_i \neq \tau_a$ do not change between t_a^- and t_a^+, i.e., $d_i(t_a^-) = d_i(t_a^+)$ and $\text{ret}_i(t_a^-) = \text{ret}_i(t_a^+)$. Hence, Expression 6.32 implies that

$$\forall \tau_i \neq \tau_a : \sum_{j=1}^{m} \text{allot}_{i,j}(t_a^-) = \text{ret}_i(t_a^+) \tag{6.33}$$

On the other hand, task τ_a has a new deadline $d_a(t_a^+) = t_a + D_a$ after the arrival of its new job at time t_a^+, and its worst-case remaining execution time is now equal to its worst-case execution time, i.e., $\text{ret}_a(t_a^+) = C_a$. Hence, it holds that

$$U_a \times (d_a(t_a^+) - t_a) = U_a \times D_a = C_a = \text{ret}_a(t_a^+) \tag{6.34}$$

The remainder of the proof is subdivided in two subproofs. In Subproof 1, we show that

$$\begin{aligned} \forall i, j : \quad &\sum_{p=1}^{j} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^+) + \text{res\_dyn}_p^+(t_a, d_i(t_a^+)) \right\} \geq \\ &\sum_{p=1}^{j} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^-) + \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) \right\} - \min \left\{ C_a \ , \ U_a \times (d_i(t_a^+) - t_a) \right\} \end{aligned}$$

with $d_0(t_a^+) \stackrel{\text{def}}{=} t_a$.

Then, in Subproof 2, we deduce from this last expression that $\sum_{j=1}^{m} \text{allot}_{i,j}(t_a^+) = \text{ret}_i(t_a^+)$ for every task τ_i in $\mathcal{A}(t_a^+)$, thereby proving the lemma.

Subproof 1: The first subproof is made by induction.
Let us assume that for task τ_{i-1} we have for all processors $k = \{1, 2, ..., m\}$:

$$\sum_{p=1}^{k} \left\{ \sum_{q=1}^{i-1} \text{allot}_{q,p}(t_a^+) + \text{res\_dyn}_p^+(t_a, d_{i-1}(t_a^+)) \right\} \geq$$
$$\sum_{p=1}^{k} \left\{ \sum_{q=1}^{i-1} \text{allot}_{q,p}(t_a^-) + \text{res\_dyn}_p^-(t_a, d_{i-1}(t_a^+)) \right\} - \min\left\{C_a \ , \ U_a \times (d_{i-1}(t_a^+) - t_a)\right\} \tag{6.35}$$

We prove for task τ_i that

$$\sum_{p=1}^{k} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^+) + \text{res\_dyn}_p^+(t_a, d_i(t_a^+)) \right\} \geq$$
$$\sum_{p=1}^{k} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^-) + \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) \right\} - \min\left\{C_a \ , \ U_a \times (d_i(t_a^+) - t_a)\right\} \tag{6.36}$$

Basic statement: Notice that the induction hypothesis (i.e., Expression 6.35) is obviously respected when $i = 1$ since both sides of Expression 6.35 are then equal to 0 (recall that $d_0(t_a^+) = t_a$).

Induction Step: We proved in Lemma 6.8 that

$$\sum_{p=1}^{k} \text{res\_dyn}_p^+(d_{i-1}(t_a^+), d_i(t_a^+)) \geq \sum_{p=1}^{k} \text{res\_dyn}_p^-(d_{i-1}(t_a^+), d_i(t_a^+)) \quad \text{if } d_i(t_a^+) \leq d_a(t_a^+)$$
$$- U_a \times (d_i(t_a^+) - d_{i-1}(t_a^+))$$
$$\sum_{p=1}^{k} \text{res\_dyn}_p^+(d_{i-1}(t_a^+), d_i(t_a^+)) = \sum_{p=1}^{k} \text{res\_dyn}_p^-(d_{i-1}(t_a^+), d_i(t_a^+)) \quad \text{otherwise}$$

with $d_0(t_a^+) \stackrel{\text{def}}{=} t_a$.

Because, if $d_i(t_a^+) > d_a(t_a^+)$ then $U_a \times (d_i(t_a^+) - t_a) \geq U_a \times (d_a(t_a^+) - t_a) = C_a$ (Expression 6.34), adding this expression to the induction hypothesis (Expression 6.35),

we get

$$\sum_{p=1}^{k}\left\{\sum_{q=1}^{i-1}\text{allot}_{q,p}(t_a^+)+\text{res\_dyn}_p^+(t_a,d_i(t_a^+))\right\}\geq$$
$$\sum_{p=1}^{k}\left\{\sum_{q=1}^{i-1}\text{allot}_{q,p}(t_a^-)+\text{res\_dyn}_p^-(t_a,d_i(t_a^+))\right\}-\min\left\{C_a\ ,\ U_a\times(d_i(t_a^+)-t_a)\right\}\quad(6.37)$$

Now, we first consider the case of task τ_a. Because it was not active at time t_a^-, its allotment on any processor at time t_a^- is equal to 0. Hence, it obviously holds that

$$\forall j\ :\ \text{allot}_{a,j}(t_a^+)\geq\text{allot}_{a,j}(t_a^-)$$

Adding this expression to Expression 6.37 when $k=j$ and $i=a$, we get

$$\sum_{p=1}^{j}\left\{\sum_{q=1}^{a}\text{allot}_{q,p}(t_a^+)+\text{res\_dyn}_p^+(t_a,d_a(t_a^+))\right\}\geq$$
$$\sum_{p=1}^{j}\left\{\sum_{q=1}^{a}\text{allot}_{q,p}(t_a^-)+\text{res\_dyn}_p^-(t_a,d_a(t_a^+))\right\}-\min\left\{C_a\ ,\ U_a\times(d_i(t_a^+)-t_a)\right\}$$

which proves Expression 6.36 for task τ_a.

Nevertheless, we still have to prove this expression for every task τ_i that does not release a job at time t_a (i.e., every $\tau_i\neq\tau_a$).

Algorithm 6.4 maximizes the allotment of τ_i on π_j at time t_a^+. Hence, according to line 3 of this algorithm, we either have $\sum_{p=1}^{j}\text{allot}_{i,p}(t_a^+)=\text{ret}_i(t_a^+)$ or $\text{allot}_{i,j}(t_a^+)=\text{allot\_dyn}_{i,j}^{\max}(t_a^+)$:

Case 1. If $\sum_{p=1}^{j}\text{allot}_{i,p}(t_a^+)=\text{ret}_i(t_a^+)$, then, it means that τ_i has been entirely allocated on processors π_1 to π_j. Since, according to Expression 6.33, τ_i has a budget of execution of exactly $\text{ret}_i(t_a^+)$ time units dispersed among the processors at time t_a^-, it must hold that

$$\sum_{p=1}^{j}\text{allot}_{i,p}(t_a^+)=\sum_{p=1}^{m}\text{allot}_{i,p}(t_a^-)$$

implying that

$$\sum_{p=1}^{j}\text{allot}_{i,p}(t_a^+)\geq\sum_{p=1}^{j}\text{allot}_{i,p}(t_a^-)$$

The modified version of the induction hypothesis (Expression 6.37) gives for $k = j$,

$$\sum_{p=1}^{j}\left\{\sum_{q=1}^{i-1}\text{allot}_{q,p}(t_a^+)+\text{res\_dyn}_p^+(t_a,d_i(t_a^+))\right\}\geq$$
$$\sum_{p=1}^{j}\left\{\sum_{q=1}^{i-1}\text{allot}_{q,p}(t_a^-)+\text{res\_dyn}_p^-(t_a,d_i(t_a^+))\right\}-\min\left\{C_a\ ,\ U_a\times(d_i(t_a^+)-t_a)\right\}$$

Therefore, adding the two previous expressions leads to

$$\sum_{p=1}^{j}\left\{\sum_{q=1}^{i}\text{allot}_{q,p}(t_a^+)+\text{res\_dyn}_p^+(t_a,d_i(t_a^+))\right\}\geq$$
$$\sum_{p=1}^{j}\left\{\sum_{q=1}^{i}\text{allot}_{q,p}(t_a^-)+\text{res\_dyn}_p^-(t_a,d_i(t_a^+))\right\}-\min\left\{C_a\ ,\ U_a\times(d_i(t_a^+)-t_a)\right\}$$

which proves Expression 6.36.

Case 2. $\text{allot}_{i,j}(t_a^+)=\text{allot\_dyn}_{i,j}^{\max}(t)$.

If $\text{allot}_{i,j}(t_a^+)=\text{allot\_dyn}_{i,j}^{\max}(t)$ then it holds from Definition 6.11, that

$$\text{allot}_{i,j}(t_a^+)=(d_i(t_a)-t_a)-\text{res\_dyn}_j^+(t_a,d_i(t_a^+))-\sum_{x=1}^{i-1}\text{allot}_{x,j}(t_a^+)-\sum_{y=1}^{j-1}\text{allot}_{i,y}(t_a^+)$$

Rearranging the terms, we get

$$(d_i(t_a)-t_a)=\text{res\_dyn}_j^+(t_a,d_i(t_a^+))+\sum_{x=1}^{i}\text{allot}_{x,j}(t_a^+)+\sum_{y=1}^{j-1}\text{allot}_{i,y}(t_a^+)\qquad(6.38)$$

Furthermore, from Definition 6.11, we have at time t_a^-,

$$\text{allot}_{i,j}^{\max}(t_a^-)=(d_i(t_a^-)-t_a)-\text{res\_dyn}_j^-(t_a,d_i(t_a^-))-\sum_{\tau_x\in\text{hp}_i(t_a^-)}\text{allot}_{x,j}(t_a^-)-\sum_{y=1}^{j-1}\text{allot}_{i,y}(t_a^-)$$

Yielding together with Expression 6.31

$$\text{allot}_{i,j}(t_a^-)\leq(d_i(t_a^-)-t_a)-\text{res\_dyn}_j^-(t_a,d_i(t_a^-))-\sum_{\tau_x\in\text{hp}_i(t_a^-)}\text{allot}_{x,j}(t_a^-)-\sum_{y=1}^{j-1}\text{allot}_{i,y}(t_a^-)$$

Since $\text{hp}_i(t_a^+) = \text{hp}_i(t_a^-)$ or $\text{hp}_i(t_a^+) = \text{hp}_i(t_a^-) \cup \tau_a$ (Expression 6.27) and because $\text{allot}_{a,j}(t_a^-) = 0$ (recall that τ_a was not active at time t_a^-), there is

$$\text{allot}_{i,j}(t_a^-) \leq (d_i(t_a) - t_a) - \text{res\_dyn}_j^-(t_a, d_i(t_a^-)) - \sum_{\tau_x \in \text{hp}_i(t_a^+)} \text{allot}_{x,j}(t_a^-) - \sum_{y=1}^{j-1} \text{allot}_{i,y}(t_a^-)$$

Rearranging the terms and using our convention on the task indices, we get

$$(d_i(t_a^-) - t_a) \geq \text{res\_dyn}_j^-(t_a, d_i(t_a^-)) + \sum_{x=1}^{i} \text{allot}_{x,j}(t_a^-) + \sum_{y=1}^{j-1} \text{allot}_{i,y}(t_a^-)$$

Now, because for every task τ_i which does not release a job at time t_a (i.e., every $\tau_i \neq \tau_a$) we have $d_i(t_a^-) = d_i(t_a^+)$, we get

$$(d_i(t_a^+) - t_a) \geq \text{res\_dyn}_j^-(t_a, d_i(t_a^+)) + \sum_{x=1}^{i} \text{allot}_{x,j}(t_a^-) + \sum_{y=1}^{j-1} \text{allot}_{i,y}(t_a^-) \tag{6.39}$$

Hence, combining Expressions 6.38 and 6.39 leads to

$$\begin{aligned} &\text{res\_dyn}_j^+(t_a, d_i(t_a^+)) + \sum_{x=1}^{i} \text{allot}_{x,j}(t_a^+) + \sum_{y=1}^{j-1} \text{allot}_{i,y}(t_a^+) \geq \\ &\text{res\_dyn}_j^-(t_a, d_i(t_a^+)) + \sum_{x=1}^{i} \text{allot}_{x,j}(t_a^-) + \sum_{y=1}^{j-1} \text{allot}_{i,y}(t_a^-) \end{aligned} \tag{6.40}$$

Furthermore, Expression 6.37) gives for $k = j-1$

$$\begin{aligned} &\sum_{p=1}^{j-1} \left\{ \sum_{q=1}^{i-1} \text{allot}_{q,p}(t_a^+) + \text{res\_dyn}_p^+(t_a, d_i(t_a^+)) \right\} \geq \\ &\sum_{p=1}^{j-1} \left\{ \sum_{q=1}^{i-1} \text{allot}_{q,p}(t_a^-) + \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) \right\} - \min\left\{ C_a \ , \ U_a \times (d_i(t_a^+) - t_a) \right\} \end{aligned}$$

Therefore, adding this expression to Expression 6.40 leads to

$$\begin{aligned} &\sum_{p=1}^{j} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^+) + \text{res\_dyn}_p^+(t_a, d_i(t_a^+)) \right\} \geq \\ &\sum_{p=1}^{j} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^-) + \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) \right\} - \min\left\{ C_a \ , \ U_a \times (d_i(t_a^+) - t_a) \right\} \end{aligned}$$

which proves Expression 6.36 and ends Subproof 1.

Subproof 2. We now prove that $\sum_{p=1}^{m} \text{allot}_{q,p}(t_a^+) = \text{ret}_q(t_a^+)$ for all tasks $\tau_q \in \mathcal{A}(t_a^+)$.

Using Expression 6.36, proven in Subproof 1, we have for all tasks $\tau_i \in \mathcal{A}(t_a^+)$ for $j = m$

$$\begin{aligned} &\sum_{p=1}^{m} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^+) + \text{res\_dyn}_p^+(t_a, d_i(t_a^+)) \right\} \geq \\ &\sum_{p=1}^{m} \left\{ \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^-) + \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) \right\} - \min\left\{C_a \ , \ U_a \times (d_i(t_a^+) - t_a)\right\} \end{aligned} \tag{6.41}$$

Moreover, we proved in Lemma 6.9 that because we have by assumption $U(t_a^+) \leq m$, it holds $\forall \tau_i \in \mathcal{A}(t_a^+)$ that

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{i-1}(t_a^+), d_i(t_a^+)) = \begin{cases} \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{i-1}(t_a^+), d_i(t_a^+)) & \\ - U_a \times (d_i(t_a^+) - d_{i-1}(t_a^+)) & \text{if } d_i(t_a^+) \leq d_a(t_a^+) \\ \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{i-1}(t_a^+), d_i(t_a^+)) & \text{otherwise} \end{cases}$$

with $d_0(t_a^+) \stackrel{\text{def}}{=} t_a$.

Consequently, we get that $\sum_{q=1}^{i} \sum_{p=1}^{m} \text{res\_dyn}_p^+(d_{q-1}(t_a^+), d_q(t_a^+))$ equals

$$\begin{cases} \sum_{q=1}^{i} \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{q-1}(t_a^+), d_q(t_a^+)) & \\ - \sum_{q=1}^{i} \left(U_a \times (d_q(t_a^+) - d_{q-1}(t_a^+))\right) & \text{if } d_i(t_a^+) \leq d_a(t_a^+) \\ \sum_{q=1}^{i} \sum_{p=1}^{m} \text{res\_dyn}_p^-(d_{q-1}(t_a^+), d_q(t_a^+)) & \text{otherwise} \end{cases}$$

and using Definition 6.10

$$\sum_{p=1}^{m} \text{res\_dyn}_p^+(t_a, d_i(t_a^+)) = \begin{cases} \sum_{p=1}^{m} \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) & \text{if } d_i(t_a^+) \leq d_a(t_a^+) \\ - \left(U_a \times (d_i(t_a^+) - t_a)\right) & \\ \sum_{p=1}^{m} \text{res\_dyn}_p^-(t_a, d_i(t_a^+)) & \text{otherwise} \end{cases} \tag{6.42}$$

Because $U_a \times (d_i(t_a^+) - t_a) \leq C_a$ if $d_i(t_a^+) \leq d_a(t_a^+)$ (from Expression 6.34), subtracting Expressions 6.42 to Expression 6.41 leads to

$$\sum_{p=1}^{m} \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^+) \geq \begin{cases} \sum_{p=1}^{m} \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^-) & \text{if } d_i(t_a^+) \leq d_a(t_a^+) \\ \sum_{p=1}^{m} \sum_{q=1}^{i} \text{allot}_{q,p}(t_a^-) + C_a & \text{otherwise} \end{cases}$$

Finally, applying Expressions 6.32 and 6.34, this leads to

$$\forall i \ : \ \sum_{q=1}^{i} \sum_{p=1}^{m} \text{allot}_{q,p}(t_a^+) \geq \sum_{q=1}^{i} \text{ret}_q(t_a^+) \tag{6.43}$$

Since Algorithm 6.4 never allocates more than $\text{ret}_q(t_a^+)$ to a task τ_q on the platform at time t_a^+ (see line 3 of Algorithm 6.4), there must be

$$\sum_{p=1}^{m} \text{allot}_{q,p}(t_a^+) \leq \text{ret}_q(t_a^+)$$

for every task τ_q. Therefore, Expression 6.43 is true only if,

$$\forall q : \sum_{p=1}^{m} \text{allot}_{q,p}(t_a^+) = \text{ret}_q(t_a^+)$$

This proves the lemma. ■

Theorem 6.1

U-EDF is optimal for the scheduling of a dynamic system that is comprised of sporadic tasks with implicit deadlines in a continuous-time environment provided that

$U(t) \leq m$ and $U_i \leq 1$ for every task $\tau_i \in \tau$ at any time t.

Proof:

By Lemma 6.6, we know that there is a valid U-EDF assignment at time $t = t_0$ assuming that no job has been released in the system, yet. Then, assuming that $U(t) \leq m$ and $U_i \leq 1, \forall \tau_i \in \mathcal{A}(t)$ at every time t, Lemma 6.10 proves that, if EDF-D is used to schedule the tasks between two consecutive task allocations, then a valid U-EDF assignment exists at any instant corresponding to a new job arrival (Notice that, if t_a is the arrival time of the new job, t_a can be equal to t in Lemma 6.10. Therefore, if two or more jobs arrive at the same instant t_a, we just have to apply Lemma 6.10 as many times as there are new jobs arriving at time t_a).

Finally, Lemma 6.5 proves that all deadlines between two such job arrivals are met. Furthermore, if no new job arrives after a given time instant t, Theorem 6.5 ensures that all tasks meet their deadlines by assuming that the next job arrival is $t_a = +\infty$. Therefore, U-EDF is optimal for the scheduling of a dynamic system that is comprised of sporadic tasks with implicit deadlines in a continuous-time environment. ■

Finally, using Theorem 3.6, it directly follows from Theorem 6.1 that

Theorem 6.2

U-EDF respects all the deadlines of a dynamic system that is comprised of sporadic tasks with unconstrained deadlines in a continuous-time environment provided that $\delta(t) \leq m$ and $\delta_i \leq 1$ for every task $\tau_i \in \tau$ at any time t.

6.9 Some Improvements

In this section, we propose two complementary improvements for the implementation of the basic U-EDF algorithm. While easy to implement, they allow to drastically improve the performances of U-EDF in terms of preemptions and migrations. However, they are executed in addition to U-EDF and do not change the algorithm itself.

6.9.1 Virtual Processing

If implemented exactly as described in Section 6.4, U-EDF can cause unnecessary preemptions and migrations. Indeed, the remaining execution time of each task is assigned to multiple processors. Moreover, this assignment varies after each job arrival.

To mitigate this issue, we apply a similar approach to the *"switching technique"* proposed in [Megel et al. 2010]. That is, we can see the theoretical schedule produced by U-EDF as a *virtual schedule* executed on *m virtual processors*. At any instant t, there is a bijection (i.e. a unique association in both directions) between the *virtual processors* (denoted π_j) and the *physical processors* (denoted $\mathcal{P}_k$) that composed the computing platform.

The main idea is that, if a schedule is correct at time t assuming a given virtual-physical processor association A, then the schedule remains correct if we permute two physical processors in A. Indeed, the situation has not been modified for the scheduler since it schedules tasks on virtual processors and not on physical processors.

Using this technique, it is possible to adapt the virtual-physical processor association such that if, at time t_p, a task τ_i instantly migrates from a processor π_b to π_a in the virtual schedule built by U-EDF then the physical processor $\mathcal{P}_k$ which was associated to π_b before t_p is associated to π_a after t_p (see Figure 6.6). Furthermore, if π_a was associated to $\mathcal{P}_\ell$ before t_p, then we associate $\mathcal{P}_\ell$ to π_b after t_p. That is, we keep the task τ_i running on the same physical processor by permuting $\mathcal{P}_k$ and $\mathcal{P}_\ell$ in the virtual-physical processor association. We therefore save one unnecessary preemption in the actual schedule executed on the physical platform.

To minimize the number of unnecessary preemptions, we need at most m permutations in the virtual-physical processor associations (one for each task executed on the platform).

Similarly, if a task τ_i is preempted when running on the physical processor $\mathcal{P}_k$, then when τ_i will be re-executed in the virtual schedule, we will try to associate its virtual processor with $\mathcal{P}_k$. Therefore, if the association is possible (i.e., the physical processor was not yet associated to another task), we save a migration in the real schedule executed on the physical platform. Again, because the number of tasks scheduled to be running at any time t is at most m, the complexity of this improvement is $O(m)$.

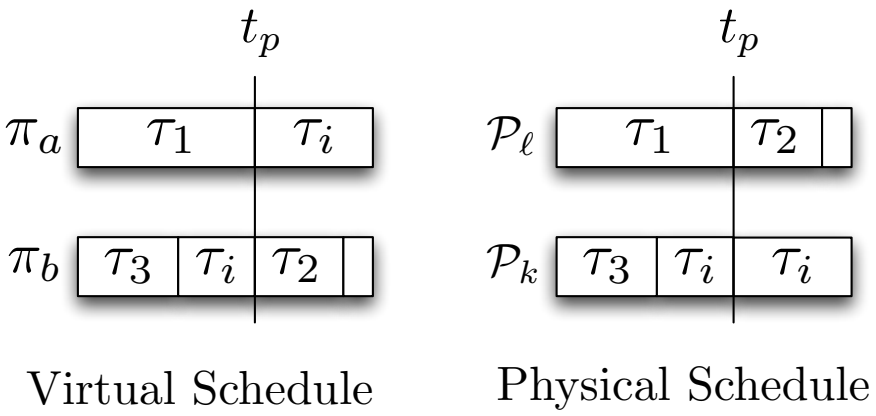


Figure 6.6: Virtual and physical schedules when an instantaneous migration of task τ_i occurs at time t_p.

6.9.2 Clustering

As explained in Chapter 5, the algorithm EKG proposes to partition the task set in clusters through the definition of a parameter k. This parameter defines the number of processors in each cluster. Then, a bin-packing algorithm can be applied to dispatch the tasks among the clusters such that the total utilization in each cluster does not exceed k. EKG is then independently used on each cluster for the scheduling of their associated tasks. Furthermore, in order to minimize the amount of preemptions and migrations, every task with a utilization greater than or equal to $\frac{k}{k+1}$ receives its own processor.

Using this approach, EKG can correctly schedule any task set with a utilization bound of $\left(\frac{k}{k+1} \cdot m\right)$ [Andersson and Tovar 2006].

We use the same technique with U-EDF. That is, we always compute the smallest value for k such that a given task set τ respects the following condition:

$$\forall t \ : \ U(t) \leq \left(\frac{k}{k+1} \cdot m\right)$$

Then, we dispatch the tasks among the clusters using a *worst fit* algorithm.

A more accurate choice on the number of processors in each cluster could be done using the results presented in [Qi et al. 2011]. However, for its facility of comparison, we preferred the solution proposed by EKG. Indeed, for a given total utilization, all the task sets will have the same parameter k.

Dividing the system in clusters reduces the number of interacting tasks in each cluster.

Hence, the number of preemptions and migrations should diminish. Moreover, we have less job arrivals, which reduces the number of reallocation of tasks during the schedule, thereby lessening the scheduling overheads but also the task preemptions and migrations that could be caused by the reallocation process.

6.10 Simulation Results

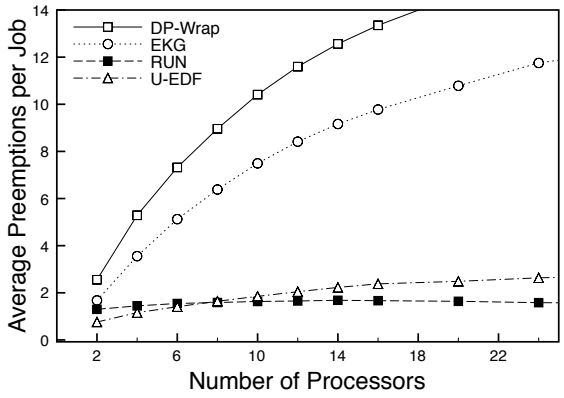
We evaluated the performance of U-EDF through extensive simulations. For each set of parameters, we simulated the scheduling of 1,000 task sets from time 0 to time 100,000. Each task had a minimum inter-arrival time randomly chosen within $[5, 100]$ using a uniform integer distribution. Task utilizations were randomly generated between 0.01 and 0.99 until the targeted system utilization was reached.

Both the clustering and the virtual processing techniques presented in the previous section were implemented for U-EDF.

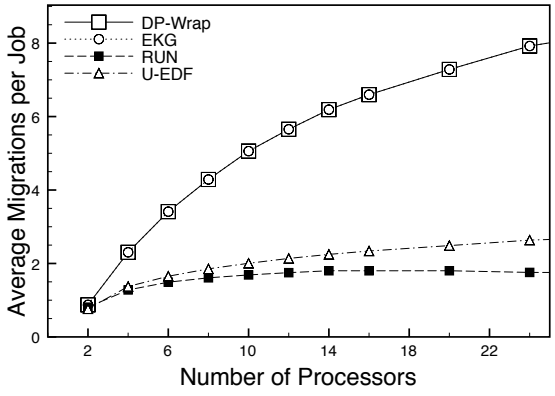
For Figures 6.7(a) and 6.7(b), we simulated the scheduling of implicit deadlines periodic tasks with a total utilization of 100% of the platform on a varying number of processors. We compared the average number of preemptions and migrations generated by U-EDF with that of the three most efficient *optimal* scheduling algorithms for *periodic* tasks on multiprocessor, namely DP-Wrap, EKG and RUN. Both RUN and U-EDF clearly outperform EKG and DP-Wrap. This result can be explained by the absence of fairness in the schedule produced by RUN and U-EDF. Furthermore, U-EDF shows to be slightly better than RUN for a number of processors smaller than 8. Hence, U-EDF seems to be the best alternative to use on platforms composed of clusters containing less than 8 processors. However, the standard deviation of both U-EDF and RUN is around 0.35 for preemptions and 0.25 for migrations. Therefore, none of these two algorithms is drastically better than the other. Nevertheless, unlike RUN which cannot schedule sporadic tasks on multiprocessor platforms, U-EDF is easily extendable to the scheduling of more general task models. For now, U-EDF is the only existing optimal algorithm for the scheduling of *sporadic* tasks which is not based on the notion of fairness.

Figure 6.7(c) presents two different results on a platform composed of 8 processors with a total utilization varying between 5 and 100%. First, we compare the average number of preemptions per job generated by U-EDF and Partitioned-EDF (P-EDF)[2] when tasks are periodic. Then, we show the results obtained for the scheduling of sporadic

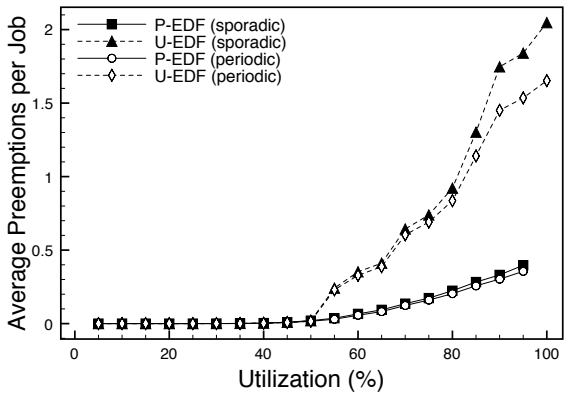
[2]Tasks are partitioned with the worst fit decreasing utilization heuristic



(a)



(b)



(c)

Figure 6.7: (a) and (b) periodic tasks running on a number of processors varying between 2 and 24 with a total utilization of 100%; (c) periodic and sporadic tasks running on 8 processors with a total utilization within 5 and 100%.

tasks. For each sporadic task we randomly selected the maximum delay that jobs can incur, in the range $[1, 100]$. Then, each job release was delayed by a random number of

time units uniformly generated between 0 and this maximum delay. Note that for P-EDF results, only task sets for which all task deadlines were met are taken into account in the calculation of the number of preemptions and migrations. We can see on Figure 6.7(c) that U-EDF and P-EDF have the same average number of preemptions when the system utilization does not exceed 50%. This particularity is the result of the clustering technique explained in Section 6.9.2. Indeed, when the total utilization is smaller than $\frac{m}{2}$ then we have only one processor in each cluster. Hence, U-EDF reduces to a fully partitioned algorithm. Furthermore, because U-EDF behaves exactly as EDF when executed on only one processor (Property 6.2), U-EDF is similar to P-EDF in this situation.
The second information we can draw from these graphs is that the average number of preemptions generated by U-EDF is slightly higher for sporadic tasks. This is due to the fact that even if there are few active tasks at some time t, the pre-allocation phase assigns these tasks on as few processors as possible. Hence there are more tasks interacting with each other than with P-EDF. Nevertheless, in its current version, U-EDF is not work conserving (i.e., a processor might stay idle, even if there is a task with remaining execution time which is not running on the platform). However, if we used the processors idle times (which are more likely in sporadic systems) to schedule pending tasks in a least remaining execution time order for instance, then we could certainly be able to further reduce the average number of preemptions incurred by jobs under U-EDF, and therefore have similar results than P-EDF. Note that even though the U-EDF curves deviate from the P-EDF curves for system utilizations greater than 50%, unlike U-EDF, Partitioned-EDF is not optimal and can miss task deadlines.

Figure 6.8 shows a comparison of the U-EDF's results with those obtained with EKG-Skip and EKG-Swap. We can see that U-EDF always performs better than EKG-Skip and has similar results than EKG-Swap. This is again understandable by the fact that EKG-Swap is the most "unfair version" of EKG. However, we must remember that U-EDF is optimal for the scheduling of sporadic and dynamic task sets while EKG-Skip and EKG-Swap can only schedule periodic tasks.

6.11 Conclusion

In this chapter, we presented the first *optimal* algorithm for the scheduling of *sporadic* tasks on *multiprocessor* that does not fairly distribute the platform processing capacity amongst tasks (or group of tasks). Furthermore, it was shown that U-EDF is a gener-

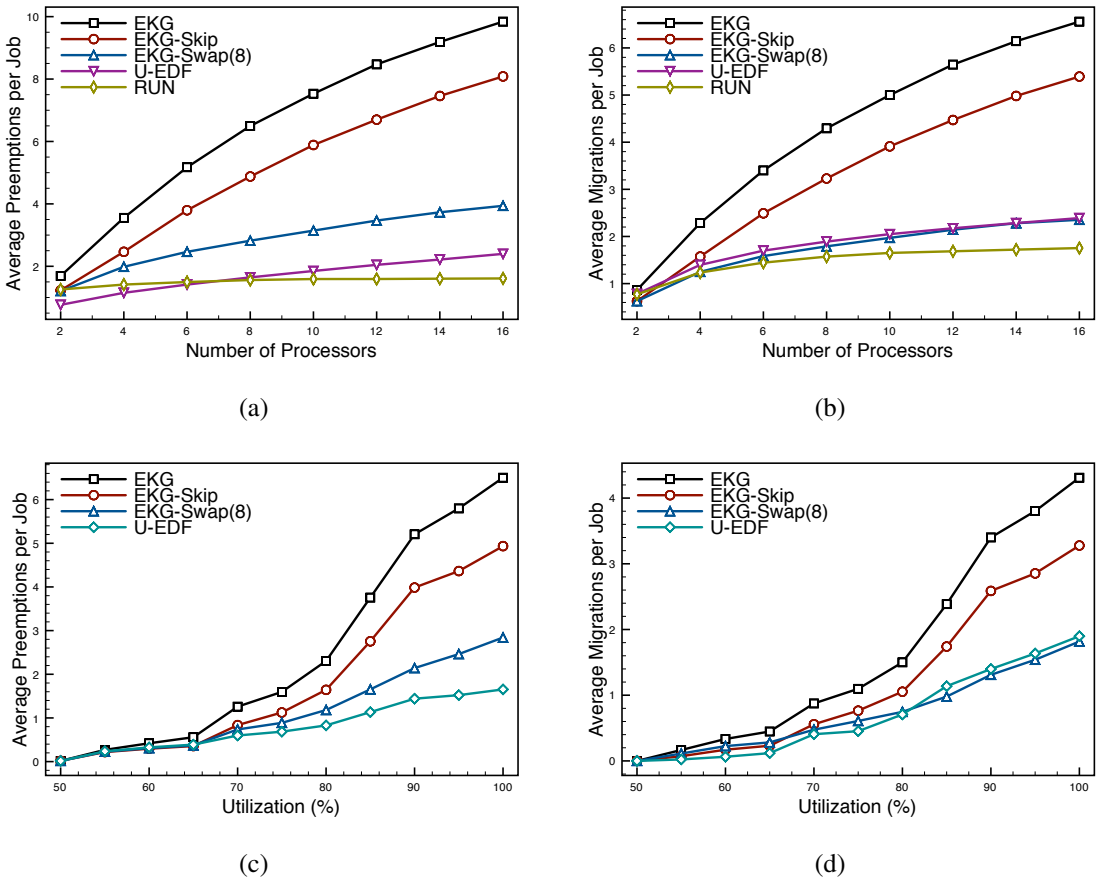


Figure 6.8: Comparison with EKG-Skip and EKG-Swap: for fully utilized processors varying between 2 and 16 (a) and (b); for 8 processors with a varying utilization (c) and (d).

alization of the optimal uniprocessor scheduling algorithm EDF. Although other works already tried to extend EDF to the scheduling of *sporadic* tasks on multiprocessor, none of them succeeded while both preserving its optimality and keeping a small number of preemptions and migrations. This fact can be explained by their will to schedule tasks *as soon as* possible. U-EDF, on the contrary, adopts an approach where *as much* work *as* possible is scheduled to execute on as few processors as possible following the prioritization order of EDF. This approach resulted in an optimal algorithm on multiprocessor platforms. This new technique has shown to be efficient in terms of preemptions and migrations. Moreover, U-EDF has been extended to the scheduling of dynamic systems and has been proven optimal in continuous-time environments. A modification of U-EDF for discrete-time systems is also provided. However, it was not proven optimal yet.

Further improvements and extensions of U-EDF are still possible. We investigate one

of them in the next chapter. We indeed propose to extend the model of schedulable tasks to the multi-threaded parallel task model.

CHAPTER

7

Generalizing Task Models

Scheduling Parallel Real-Time Tasks

"We build too many walls and not enough bridges."

Isaac Newton

Contents

Abstract

In the three previous chapters, we have presented many scheduling algorithms for multiprocessor platforms. With these algorithms, we have always assumed that the tasks may be running on only one processor at a time. However, the number of cores available in computational platforms has drastically grown during these last few years. Concurrently, a new coding paradigm dividing tasks into smaller execution instances called *threads*, was developed. These threads can be executed in parallel on many processors, taking advantage of the inherent parallelism of large multiprocessor platforms. However, only few methods were proposed to efficiently schedule hard real-time multi-threaded tasks on multiprocessor.

In this chapter, we propose techniques optimizing the number of processors needed to schedule such sporadic parallel tasks with constrained deadlines, still using the scheduling algorithms presented in the previous chapters. We first define an optimization problem determining, for each thread, an intermediate (artificial) deadline minimizing the number of processors needed to schedule the whole task set. The scheduling algorithm can then schedule threads as if they were *independent sequential* sporadic tasks. The second contribution is an *efficient* and nevertheless *optimal* algorithm that can be executed online to determine the thread's deadlines. Hence, it can be used in dynamic systems were all tasks and their characteristics are not known *a priori*. We finally prove that both techniques achieve a resource augmentation bound on the processor speed of 2 when the tasks have implicit deadlines and the threads are scheduled with optimal scheduling algorithms for dynamic task systems such as those presented in the three previous chapters (e.g., U-EDF, DP-Wrap, *etc*.). The resource augmentation bound on the processor speed becomes equal to 3 for EDF-US $\left[\frac{1}{2}\right]$ and EDF$^{(k_{\min})}$

Note: This work was published in the work-in-progress session of RTSS '11 [Nelissen et al. 2011b] and as a full paper at ECRTS '12 [Nelissen et al. 2012a].

7.1 Introduction

These last years, we have witnessed a dramatic increase in the number of cores available in computational platforms. For instance, Intel designed a 80 cores platform [Intel® 2012b] and today, Tilera currently sells processors with up to 100 identical cores [Tilera 2011]. To take advantage of this high degree of parallelism, a new coding paradigm has been introduced using programming languages and API such as OpenMP [OpenMP Architecture Review Board 2011; Chandra et al. 2001], MPI [Argonne National Laboratory 2012], CilkPlus [Intel® 2012a], Go [Google 2012], Java [Lea 2000], *etc*. These various tools allow the software designer or the compiler to explicitly or implicitly divide the execution of tasks in parallel instances, usually called *threads*, that can run concurrently on the processing platform. With this new programming paradigm, we can for instance parallelize the execution of a task which would not have been able to respect its deadlines while executed on a single processor. Hence, we can use many processors simultaneously to accelerate the computation of the task instead of increasing the processors speed which would have resulted in an increase of the power consumption of the platform [Chen and Kuo 2007].

In this chapter, we consider multi-threaded tasks modeled as follows (refer to Figure 7.2 on page 233):

- each task τ_i is defined as a sequence of segments $\sigma_{i,j}$ which must be executed while respecting some precedence constraints;
- each segment is a collection of threads and the $n_{i,j}$ threads of a same segment $\sigma_{i,j}$ may be scheduled *simultaneously* on (at most) $n_{i,j}$ different processors.

A task τ_i is therefore represented by a *directed acyclic graph* (DAG). Each node of the graph is a segment which can itself contains a set of threads that can be executed concurrently. Although the number of threads in each segment is usually defined by the developer or the compiler at design time, it is not systematic. Hence, in dynamic systems, the number of threads in the segments of a task may be defined right at the release of a new job depending on the current state of the system. The task characteristics may therefore not be known *a priori*.

Multi-threaded parallel tasks have been studied in only few previous works. One may cite [Korsgaard and Hendseth 2011] in which the authors propose a sufficient (but pessimistic) schedulability test for such systems scheduled with global EDF. Interesting

approaches were also proposed in [Lakshmanan et al. 2010] and [Saifullah et al. 2011, 2012] to optimize the utilization of the platform when tasks follow the *fork-join* model. Other works investigated solutions for different models of parallel tasks that will be extensively presented in the next section.

Contribution: In this chapter, we propose techniques optimizing the number of processors needed in the computational platform to ensure that all parallel tasks respect their deadlines. To reach this goal, we propose two different approaches: an optimization problem which must be solved offline; and an efficient (optimal) algorithm which can be executed online. Both techniques compute, for each thread, an intermediate (artificial) deadline so that the online scheduler may manage the execution of each thread as an independent *sequential sporadic* task with *constrained* deadline. The intermediate deadlines are optimally derived from a *sufficient* schedulability test of multiprocessor schedulers for dynamic systems such as U-EDF (Chapter 6), LRE-TL [Funk 2010] or DP-Wrap [Levin et al. 2010]. Further, these results can easily be extended to any other scheduling algorithm for multiprocessor platforms according that their schedulability tests are exclusively based on utilizations and densities.

The interest of the offline approach resides in the fact that it can be generalized without too much efforts for systems using models of tasks that are more complex. It is thus perfectly suited to complex task systems which must however be static. On the other hand, the online algorithm targets dynamic systems where all tasks and their characteristics are not known *a priori*. However, the model of tasks is simpler since two segments of the same task cannot be executed concurrently. Further, we prove the optimality of the online algorithm (amongst the algorithms adding intermediate deadlines to tasks modeled by a sequence of sequential segments), and show that both approaches achieve a resource augmentation bound on the processor speed (see Definition 3.7 on Page 42) of 2 (respectively, 3) when the threads are scheduled using U-EDF, LRE-TL, ... (respectively, using EDF-US$\left[\frac{1}{2}\right]$ or EDF$^{(k_{\min})}$).

Note that we never mentioned RUN [Regnier et al. 2011; Regnier 2012] as being a valid alternative for the scheduling of the set of threads. Indeed, we need a scheduling algorithm that can schedule *sporadic* tasks with constrained deadlines. Unlike U-EDF which was presented in Chapter 6, this is currently not the case of RUN which has not been extended to the scheduling of sporadic tasks yet.

Organization of this chapter: Section 7.2 presents various results on the real-time scheduling of parallel tasks. The basic model of tasks that we are using in this chap-

ter is explained in Section 7.3, and Section 7.4 presents the problem that we are trying to solve. An offline approach based on an optimization problem is presented in Section 7.5, while an online solution is proposed in Section 7.6. Then, two resource augmentation bounds are proven in Section 7.7. One for algorithms such as U-EDF or DP-Wrap, and an other for EDF-US[$\frac{1}{2}$] and EDF$^{(k_{\min})}$. Next, Section 7.8 proposes extensions of the offline solution for the determination of segment deadlines of more general task models. Finally, simulation results are presented in Section 7.9 and Section 7.10 concludes the chapter.

7.2 Related Works

The study of parallel tasks is not new. It has already been extensively studied in the '80s s and '90s to take advantage of the high number of processors available in supercomputers [Ousterhout 1982; Edler et al. 1985; Feitelson 1990; Zahorjan et al. 1991; Feitelson and Rudolph 1992; Drozdowski 1996]. The goal was already to accelerate the execution of the tasks and improve the utilization of the machines by dividing the tasks in smaller instances which can simultaneously be executed on different processors.
Various scheduling policies were designed and implemented to handle this new execution paradigm [Ousterhout 1982; Edler et al. 1985; Feitelson 1990; Feitelson and Rudolph 1992]. However, at that time, these parallel systems were not subject to real-time constraints and the tasks did not have to respect any kind of deadlines.

Nowadays, there are two main models of parallel tasks which may be used in practical applications:

- the *gang* model in which all threads of a same task must be executed simultaneously;
- the *independent threads* approach in which the threads released by a same task can be executed independently one from another.

7.2.1 Gang Scheduling

The *gang scheduling* or *coscheduling* was first introduced in [Ousterhout 1982] as a way to effectively use the processing platform when executing a set of highly interacting threads. Interaction between threads of a same parallel application can either be implicit (such as a message passing) or explicit (such as a synchronization barrier). In the case

of highly interacting systems where threads must often synchronize their executions or exchange data, it was shown that executing all the interacting threads simultaneously, is the best solution to get an effective synchronization and therefore minimize their execution times [Zahorjan et al. 1991; Feitelson and Rudolph 1992]. This approach is named *gang scheduling* because the interacting threads are grouped in a "gang" which must be considered as a single schedulable entity. Feitelson and Rudolph [1992] highlighted the fact that if highly interacting threads were not gang scheduled, they would spend more time waiting the non-running threads than actually executing on the platform. In this case, either the threads would be idling on the processors, or the processors should switch their execution contexts so often that the overheads induced by these context switches would even exceed the actual execution time of the threads. On the other hand, if all interacting threads start and stop their executions simultaneously, an effective busy waiting synchronization protocol can be used to reduce the synchronization overheads.

Note that gang scheduling is only necessary for threads with *"fine grain"* interactions. On the other hand, if the time between two interactions is reasonably long in comparison with the execution time of the tasks, it is preferable to execute all threads independently as it will be presented in Section 7.2.2 [Feitelson and Rudolph 1992].

In the context of real-time systems, we have always supposed that tasks are described by an application code which is repeatedly executed. Each single execution of the code of a particular task τ_i corresponds to the release of a new job by τ_i. The same happens with parallel tasks. However, instead of being sequential as it was assumed in the previous chapters, the task code can be parallelized using threads. In the case of gang scheduling, jobs released by a task may be categorized in three different families [Goossens and Berten 2010]:

Rigid A job $J_{i,j}$ is said to be rigid if the number of parallel threads of $J_{i,j}$ that must be executed synchronously is defined externally to the scheduler, *a priori* and does not change throughout the execution.

Moldable A job $J_{i,j}$ is said to be moldable if the number of parallel threads of $J_{i,j}$ that must be executed synchronously is defined by the scheduler and does not change throughout the execution. That is, the scheduler may take decisions on the number of created threads regarding, for instance, the current workload on the platform.

Malleable A job $J_{i,j}$ is said to be malleable if the number of parallel threads of $J_{i,j}$ that must be executed synchronously can be modified by the scheduler during the execution of $J_{i,j}$. That is, the scheduler may decide to increase or decrease the degree

of parallelism of the job regarding the evolution of the workload on the platform during the execution.

Rigid Jobs Scheduling

Two main works on this topic were published in [Kato and Ishikawa 2009] and [Goossens and Berten 2010]. Both tried to extend to the scheduling of parallel tasks composed of a fixed number of threads, previous scheduling policies initially devoted to the scheduling of sequential tasks. The inherent difficulty with the gang scheduling of rigid jobs resides in the fact that a job which is constituted of n_i threads needs exactly n_i processors to start running.

Example:

Let us consider a job $J_{i,j}$ constituted of 4 threads. If only 3 processors are not utilized by jobs with highest priorities at time t, then $J_{i,j}$ cannot start its execution before at least one of the thread currently running on the platform finishes its execution. Hence, the job $J_{i,j}$ cannot take advantage of the three available processors.

The situation depicted in the previous example can even lead to the execution of a low priority job before a high priority one, thereby resulting in a kind of priority inversion.

Example:

In the previous example, consider that another job $J_{k,\ell}$ with a lower priority than $J_{i,j}$ is active at time t. If $J_{k,\ell}$ is constituted of only 3 threads or less, it can start executing on the three available processors even if $J_{i,j}$ is still waiting for the release of one more processor.

Kato and Ishikawa [2009] provide an extension of EDF to the gang scheduling of rigid jobs, while in [Goossens and Berten 2010], the authors propose four different fixed job and fixed task priority scheduling policies together with their exact schedulability tests.

Moldable Jobs Scheduling

In the case of moldable jobs, the schedulers can decide how many threads — and therefore how many processors — the job will use during its entire execution. More threads means smaller execution time. However, increasing the number of threads should not be considered as being free. Usually, multiplying the number of threads leads to an increase

of the synchronization overheads between these threads. Hence, we should assume that for a same job $J_{i,j}$ divided in k or ℓ threads (with $k < \ell$) with a respective worst-case execution time of $c^k_{i,j}$ and $c^\ell_{i,j}$, there will be $c^k_{i,j} > c^\ell_{i,j}$ but $k \times c^k_{i,j} < \ell \times c^\ell_{i,j}$. On the other hand, if the parallelization was free (i.e., $k \times c^k_{i,j} = \ell \times c^\ell_{i,j}$ for any number of threads k and ℓ), the best possible solution in terms of schedulability would be to divide all jobs in as many threads as processors composing the platform, and then apply the EDF scheduling algorithm to schedule each gang of threads as if they were single jobs running on a single processor [Berten et al. 2011]. Unfortunately, free parallelization is not realistic.

Deciding in how much threads each job has to be divided in is not a simple problem. If taken online, the decision must indeed be taken considering the current workload of the system but with a limited or no knowledge at all of the future load requests. Hence, most works propose off-line solutions deriving the number of threads of each job before starting to execute the application [Han and Lin 1989; Manimaran et al. 1998; Berten et al. 2011]. In [Berten et al. 2011], the authors also propose two on-line solutions. One is based on a partial anticipation on the future load requests and the second takes its decisions only considering the current state of the system without any assumption on its past or future.

Malleable Jobs Scheduling

Scheduling malleable jobs is certainly the most flexible type of gang scheduling. Indeed, the scheduler can modify the parallelism degree of tasks during the schedule. Collette et al. [2007, 2008] provide an exact feasibility test for sporadic tasks releasing such malleable jobs. They present a theoretically optimal offline scheduling algorithm which determines the evolution of the number of threads that must be allocated to each job. They also investigate techniques reducing the number of preemptions of the scheduled jobs.

The biggest problem with this last model of gang scheduling is its actual implementation. Modifying the number of threads allocated to a job at run-time is not an easy matter in terms of implementation. A first step toward a maybe more realistic solution is the model proposed in [Berten et al. 2009]. In this work, the authors propose a sufficient schedulability test for tasks composed of a set of segments which are sequentially executed, i.e., the second segment of a task cannot start executing before the first segment of this same task has been finished. Each segment has a maximum degree of parallelism and the scheduler must decide the number of threads in which each segment has to be divided (while respecting the maximum degree of parallelism of these segments). Hence, even though it is not yet the case in [Berten et al. 2009], the number of threads in a segment

could be fixed and not vary during its execution. The time-instants lying between two segments of a same job could therefore be seen as a synchronization point in the code at which decisions on the number of threads allocated to the execution of the job can easily be taken (for example, at the beginning of a *"for"* loop where we can decide to parallelize the execution of the loop iterations).

7.2.2 Independent Thread Scheduling

As already previously stated, if the threads of a same task do not have "fine grain" interactions, it is not required to schedule them synchronously. In these conditions, gang scheduling should be avoided.

In [Lupu and Goossens 2011], the authors consider that a task repeatedly releases jobs composed of a single set of *independent* threads. These threads only share the same deadline. Hence, once a thread has been released, it can be executed on the processing platform independently of the execution of the other threads. Lupu and Goossens [2011] define several kinds of real-time schedulers for this task model and their associated schedulability tests.

This last model of task is however rather simple. OpenMP for instance, allows the application designer to define points in the application code where the task can *fork* in many threads and then *join* again in a single thread. Hence, each job can be seen as a sequence of segments, alternating between sequential and parallel phases. A sequential segment executes only one master thread which is split in multiple parallel independent threads during the parallel phases. Of course, this execution pattern is still quite basic and OpenMP or any other tool such as those cited in the introduction, can be used to let the application designer or the compiler define a more complex execution pattern than this strict alternation between sequential and parallel segments. For instance, the master thread can fork in many threads and one of these newly created thread can fork again in a new set of threads (see Figure 7.3(a) on page 251 for such an example). The task can therefore be represented by a *directed acyclic graph* where each node represents a segment containing one or many threads (see Figure 7.5(b) on page 255 for instance). However, there is currently no results proposing any solution for the scheduling of such recurrent real-time tasks. Hence, before addressing this problem in Section 7.8, we present some previous works assuming simpler task models.

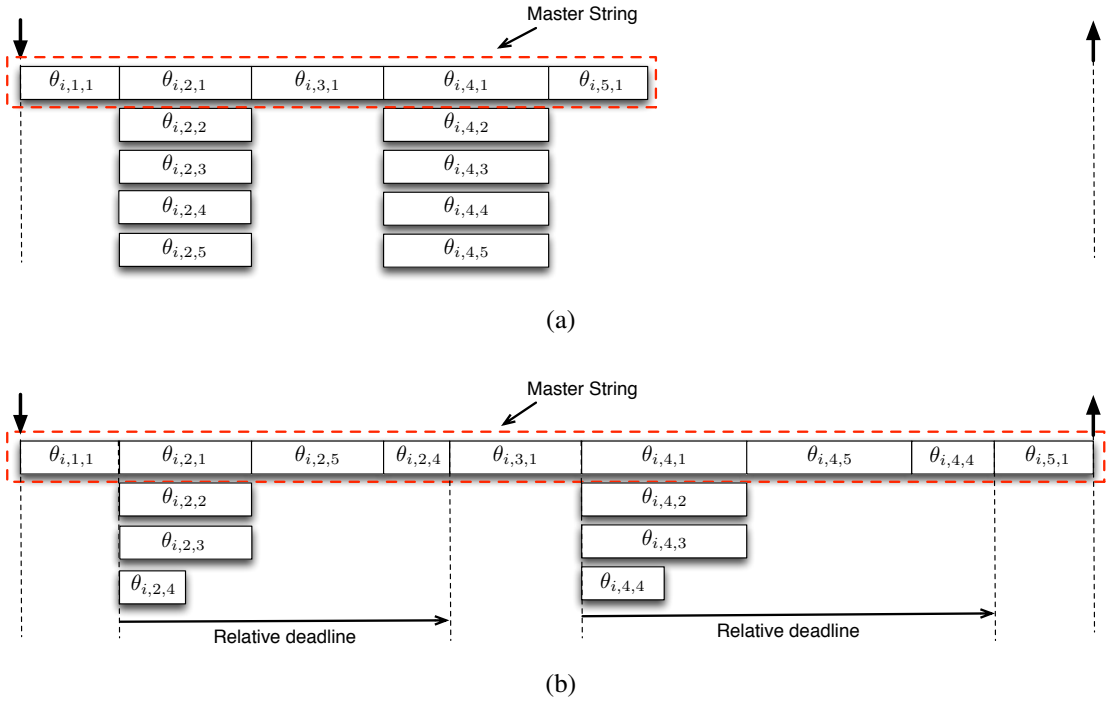


Figure 7.1: Multi-threaded task with a strict alternation between sequential and parallel segments (a) before and (b) after the transformation using the method presented in [Lakshmanan et al. 2010]

Strict Alternation between Sequential and Parallel Segments

Lakshmanan et al. [2010] propose a scheduling algorithm for the *fork-join* model with a strict alternation between sequential and parallel segments. To this first constraint, three others must be added:

- all the parallel segments are assumed to have the same number of parallel threads;
- the number of threads in a parallel segment may never exceed the number of processors in the platform;
- all threads belonging to a same parallel segment have the same worst-case execution time.

The core idea of Lakshmanan et al. [2010] is to minimize the interferences of a task τ_i with the other tasks during the schedule. To achieve this goal, they define a *master string* for each task τ_i. Initially, the master string of a task is constituted of the master thread of each sequential segment and the first thread of each parallel section (see Figure 7.1(a)). Note that the total worst-case execution time of this sequence of threads has

to be smaller than or equal to the task deadline. Otherwise, the task would never be able to respect its deadline. In order to minimize the number of threads of τ_i interfering with other tasks, Lakshmanan et al. [2010] propose to *"stretch"* the master string by adding other threads of the parallel sections of τ_i, so as to reach a total execution time equal to the deadline of τ_i (see Figure 7.1(b)). Hence, a single processor can be assigned to the execution of this master string, and the threads belonging to this master string will not interfere with the schedule of other tasks. The remaining threads of the parallel sections of τ_i that cannot be added to the master string, are then assigned an artificial deadline computed so as to respect the precedence constraints in the execution of the segments[1] (see Figure 7.1(b)) but maximizing their slack (i.e., the difference between their deadline and their worst-case execution time). Then, these remaining threads are individually partitioned among the processors using the FBB-FDD (which stands for Fisher Baruah Baker - First Fit Decreasing) algorithm [Fisher et al. 2006], and finally scheduled with the Deadline Monotonic algorithm on each processor independently.

Lakshmanan et al. [2010] proved that their scheduling approach has a resource augmentation bound on the processor speed of 3.42.

This work was recently improved in [Fauberteau et al. 2011], minimizing the number of migrating threads for each task.

Arbitrary Number of Threads in each Segment

The strict alternation between parallel and sequential segments has been raised in [Saifullah et al. 2011, 2012]. The authors now consider that each segment is composed of an arbitrary number of threads. However, all threads belonging to a same parallel segment must still have the same worst-case execution time.

Drawing inspiration from Lakshmanan et al. [2010] who impose an artificial deadline to the threads to respect the precedence constraints during their execution, Saifullah et al. [2011] propose another method to impose one artificial deadline per *segment*. This deadline is then shared by all threads belonging to the same segment. Moreover, it also corresponds to the release time of the threads of the next segment. Once those deadlines have been defined, the scheduling of the task set is equivalent to the scheduling of a set of independent sequential sporadic tasks, each thread being considered as an independent

[1] Note that adding artificial deadlines to synchronize the execution of many portions of code, was not a new idea: it was already proposed in [Sun and Liu 1996] to synchronize the execution of different parts of strictly periodic tasks running on different processors.

task [Saifullah et al. 2011, 2012].

The methodology proposed in [Saifullah et al. 2011, 2012] to determine the segment intermediate deadlines is the following:

1. For each task, it categorizes each segment as being either "light" or "heavy". This decision is based on the number of threads in the segment compared to the total slack of the task.

2. If a task contains some heavy segments, the light segments receive no slack (i.e., the deadline of a light segment is defined as being equal to the worst-case execution time of its threads) and the slack of the task is distributed between heavy segments so that they all share the same density.

3. If there is not any heavy segment in a task, then the slack of the task is proportionally distributed between all light segments (according to their respective workloads).

The transformation method proposed in [Saifullah et al. 2011, 2012] has a resource augmentation bound on the processor speed of 4 when global EDF is used to schedule the threads (basing the scheduling decisions on their new artificial deadlines). The resource augmentation bound becomes equal to 5 when the EDF scheduling algorithm is replaced by a partitioned deadline monotonic (DM) scheduling scheme.

Comparison with our solution. The research presented in this chapter, also adds artificial deadlines to the segments of each task. However, it strictly dominates the work proposed in [Saifullah et al. 2011, 2012] in the sense that if the method in [Saifullah et al. 2011, 2012] needs m' processors to schedule a task set τ, then, we need $m \leq m'$ processors to schedule τ.

Actually, we prove that, when comparing with any solution imposing intermediate deadlines for the segment executions, our technique is optimal. In the experimental section, we provide examples where the method of [Saifullah et al. 2011, 2012] requires up to three times more processors than our approach. Furthermore, as presented in the next sections, we deal with more general models of parallel tasks.

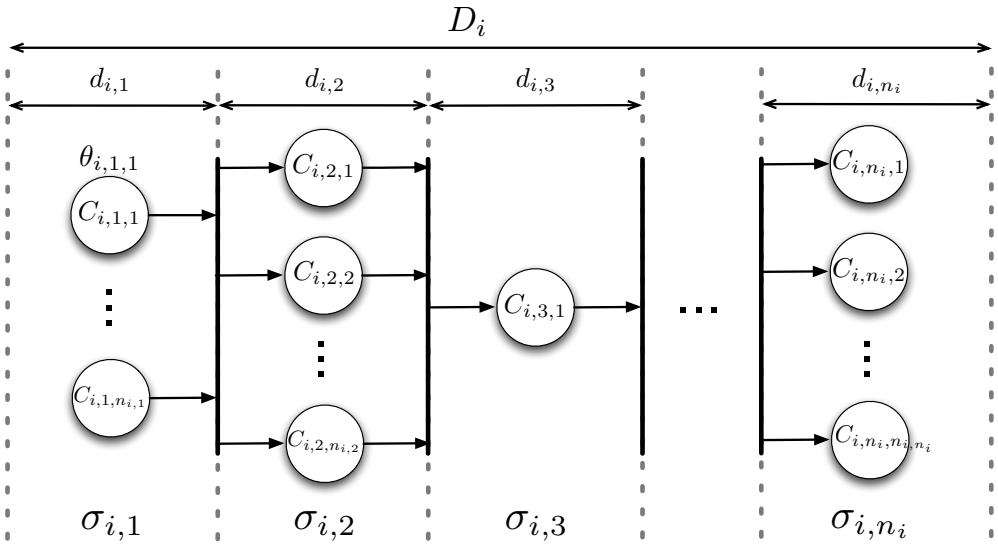


Figure 7.2: Parallel task τ_i composed of n_i segments $\sigma_{i,1}$ to σ_{i,n_i}.

7.3 System Model

We assume a set $\tau \stackrel{\text{def}}{=} \{\tau_1, ..., \tau_n\}$ of n sporadic tasks with constrained deadlines. Each task τ_i is a sequence of n_i segments denoted $\sigma_{i,j}$ (with $j \in \{1, ..., n_i\}$). Each segment $\sigma_{i,j}$ is a set of $n_{i,j}$ threads denoted $\theta_{i,j,k}$ (with $k \in \{1, ..., n_{i,j}\}$) and each thread $\theta_{i,j,k}$ has a worst-case execution time of $C_{i,j,k}$ (see Figure 7.2). Each task τ_i is a sporadic task with constrained deadline, meaning that τ_i has a relative deadline D_i not larger than its minimal inter-arrival time T_i. This implies that τ_i has never more than one active job at any time t.

We define the minimum time needed to execute $\sigma_{i,j}$ as $C_{i,j}^{\min} \stackrel{\text{def}}{=} \max_k \{C_{i,j,k}\}$ and the total worst-case execution time of a segment $\sigma_{i,j}$ as the sum of its threads worst-case execution time, i.e., $C_{i,j} \stackrel{\text{def}}{=} \sum_{k=1}^{n_{i,j}} C_{i,j,k}$. Therefore, $C_{i,j}^{\min}$ represents the minimum amount of time needed to complete the execution of all threads belonging to the segment $\sigma_{i,j}$ assuming that it can permanently use $n_{i,j}$ processors for its execution. Similarly, $C_{i,j}$ is the maximum amount of time needed to execute all threads in $\sigma_{i,j}$ if there is only a single processor.

Note that a task τ_i can only respect its deadline if

$$\sum_{j=1}^{n_i} C_{i,j}^{\min} \leq D_i$$

The total worst-case execution time C_i of any task τ_i is equal to the sum of the worst-

case execution time of its constituting segments, i.e., $C_i \stackrel{\text{def}}{=} \sum_{j=1}^{n_i} C_{i,j}$. Hence, C_i is the maximum amount of time needed to complete the execution of the entire task τ_i on a single core.
Therefore, if $C_i \leq D_i$ then all threads of τ_i can be scheduled sequentially and still respect the task deadline.

In this work, we assume that all threads of a same segment $\sigma_{i,j}$ are synchronous. That is, all threads of $\sigma_{i,j}$ are released simultaneously. Furthermore, all threads of a segment $\sigma_{i,j}$ must have completed their execution before starting to execute the next segment $\sigma_{i,j+1}$. There is therefore a precedence constraint between the segment constituting the task. Note that different models of precedence constraints will be further investigated later in Section 7.8.

Remember that the density of a task τ_i is given by $\delta_i \stackrel{\text{def}}{=} \frac{C_i}{D_i}$. This quantity represents the average computing capacity needed for the execution of a job of τ_i between its arrival and its deadline occurring D_i time units later.

The notations used in this chapter are summarized in Table 7.1.

7.4 Problem Statement

As previously explained, our goal is to transform the threads composing the parallel tasks in a set of sporadic *sequential* tasks with constrained deadlines. This approach, already described in [Lakshmanan et al. 2010; Saifullah et al. 2011, 2012], implies that for each thread, an arrival time and a relative deadline must be defined.

We assume in our model that all threads of a same segment $\sigma_{i,j}$ are released simultaneously and the threads of $\sigma_{i,j+1}$ can only start running once the execution of the threads of $\sigma_{i,j}$ is completed. We therefore assume that all threads in a segment $\sigma_{i,j}$ have the same deadline $d_{i,j}$ and threads in $\sigma_{i,j+1}$ are released as soon as the deadline $d_{i,j}$ of the previous segment has been reached. Hence, the minimum inter-arrival time of any thread of τ_i is equal to the minimum inter-arrival time T_i of τ_i. Consequently, our problem consists in determining the relative deadlines $d_{i,j}$ of every segment $\sigma_{i,j}$ so that the number of required processors to respect all deadlines is minimized.

Let the instantaneous total density $\delta(t)$ be the sum of the densities of all the *active jobs* at time t. Some existing scheduling algorithms such as U-EDF, DP-Wrap or LRE-TL can schedule any set of sequential *sporadic* tasks as long as the instantaneous total

Table 7.1: System notations

System characteristics	
τ	Set of tasks
n	Number of tasks in τ
m	Number of processors
Task characteristics	
τ_i	Task number i
n_i	Number of segments in τ_i
D_i	Relative deadline of τ_i
T_i	Minimal inter-arrival time
C_i	Worst-case execution time of τ_i ($C_i \stackrel{\text{def}}{=} \sum_{j=1}^{n_i} C_{i,j}$)
δ_i	Density of τ_i ($\delta_i \stackrel{\text{def}}{=} \frac{C_i}{D_i}$)
Segment characteristics	
$\sigma_{i,j}$	j^{th} segment of τ_i
$n_{i,j}$	Number of threads in $\sigma_{i,j}$
$C_{i,j}$	Worst-case execution time of $\sigma_{i,j}$ ($C_{i,j} \stackrel{\text{def}}{=} \sum_{k=1}^{n_{i,j}} C_{i,j,k}$)
$C_{i,j}^{\min}$	Minimum time needed to execute $\sigma_{i,j}$ ($C_{i,j}^{\min} \stackrel{\text{def}}{=} \max_k \{C_{i,j,k}\}$)
Thread characteristics	
$\theta_{i,j,k}$	k^{th} thread of $\sigma_{i,j}$
$C_{i,j,k}$	Worst case execution time of $\theta_{i,j,k}$

density $\delta(t)$ does not exceed the number m of processors constituting the processing platform [Srinivasan and Anderson 2005b; Funk 2010; Levin et al. 2010]. This property is expressed through the following *sufficient feasibility test*:

Property 7.1

A task set τ is feasible on a platform of m identical processors if, at any time t, we have $\delta(t) \leq m$.

According to Property 7.1, if we want to minimize the number of processors needed to schedule τ, then we must minimize the maximum value reachable by $\delta(t)$. Hence, we define the density of $\sigma_{i,j}$ as $\delta_{i,j} \stackrel{\text{def}}{=} \frac{C_{i,j}}{d_{i,j}}$ and since two segments $\sigma_{i,j}$ and $\sigma_{i,\ell}$ $(j \neq \ell)$ of the same task τ_i cannot be active simultaneously, we have to minimize the following expression:

$$\delta^{\max} \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \max_j \left\{ \delta_{i,j} \right\} = \sum_{\tau_i \in \tau} \max_j \left\{ \frac{C_{i,j}}{d_{i,j}} \right\} \tag{7.1}$$

Indeed, since the parallel tasks are sporadic, we do not know which segment of each task will be active at each instant t. Therefore, in the worst case, each task has its segment with the largest density active at time t.

If, after the optimization process, we have $m \geq \delta^{\max}$ then the task set τ is schedulable on the processing platform using a scheduling algorithm such as those previously cited (i.e., U-EDF, LRE-TL, *etc.*).

7.5 Offline Approach: An Optimization Problem

As stated in the previous section, optimizing the number of processors m implies to minimize $\delta^{\max}$. However, some constraints on the relative deadlines of each segment must be respected:

- The total execution time granted to all segments composing τ_i must be smaller than the relative deadline of τ_i. That is, for each task τ_i, it must hold that

$$\sum_{j=1}^{n_i} d_{i,j} \leq D_i \tag{7.2}$$

  In practice, we will impose that $\sum_{j=1}^{n_i} d_{i,j} = D_i$ as the segment densities decrease if their deadlines increase (we remind that $\delta_{i,j} \stackrel{\text{def}}{=} \frac{C_{i,j}}{d_{i,j}}$).

- The relative deadline of any segment $\sigma_{i,j}$ cannot be smaller than its minimum execution time $C_{i,j}^{\min}$. That is,

$$d_{i,j} \geq C_{i,j}^{\min} \stackrel{\text{def}}{=} \max_k \left\{ C_{i,j,k} \right\} \tag{7.3}$$

  Indeed, because a thread cannot be executed in parallel on two (or more) processors, the relative deadline of the associated segment cannot be smaller than its largest thread worst-case execution time.

Hence, the optimization problem can be written as

$$\begin{aligned} \textbf{Minimize:} \quad & \delta^{\max} = \sum_{\tau_i \in \tau} \max_j \left\{ \frac{C_{i,j}}{d_{i,j}} \right\} \\ \textbf{Subject to:} \quad & \forall \tau_i \in \tau \ , \ \sum_{j=1}^{n_i} d_{i,j} = D_i \\ & \forall \sigma_{i,j} \ , \ d_{i,j} \geq C_{i,j}^{\min} \end{aligned}$$

Nevertheless, because the constraints on the relative deadlines of any segment $\sigma_{i,j}$ of a task τ_i are independent of the constraints on any other segment $\sigma_{a,b}$ ($a \neq i$) of another task τ_a, we can subdivide this optimization problem in n sub-problems consisting, for every task τ_i, in the minimization of $\max_j \left\{ \frac{C_{i,j}}{d_{i,j}} \right\}$. Indeed, the minimization of a sum with independent terms, is equivalent to the minimization of every term independently. We therefore get $\forall \tau_i \in \tau$,

$$\textbf{Minimize:} \quad \max_j \left\{ \frac{C_{i,j}}{d_{i,j}} \right\} = \max_j \{ \delta_{i,j} \} \tag{7.4}$$

$$\textbf{Subject to:} \quad \sum_{j=1}^{n_i} d_{i,j} = D_i \tag{7.5}$$

$$\forall \sigma_{i,j} \in \tau_i \ , \ d_{i,j} \geq C_{i,j}^{\min} \tag{7.6}$$

This new formulation has two different interests:

1. It allows to determine the relative deadlines of each task *independently*. Therefore, if the task set is modified, we do not have to recompute the properties of every task, but only for the new and altered tasks in τ.

2. It highly reduces the solution research space visited by the optimization algorithm. In fact, for each task τ_i, both the number of variables and the number of constraints

in the optimization problem, increase linearly with the number of segments n_i in τ_i.

We should also note that, as already stated in Section 7.3, by Equations 7.5 and 7.6, a solution to the optimization problem exists if and only if $\sum_{j=1}^{n_i} C_{i,j}^{\min} \leq D_i$.

7.5.1 Problem Linearization

The optimization problem can be linearized by maximizing $\min_j \left\{ \frac{d_{i,j}}{C_{i,j}} \right\}$ instead of minimizing $\max_j \left\{ \frac{C_{i,j}}{d_{i,j}} \right\}$. It is indeed equivalent to minimize a function than maximize its inverse. Therefore, by replacing the objective function by this dual version, we get

$$
\begin{aligned}
& \forall \tau_i \in \tau, \\
\textbf{Maximize:} \quad & \min_j \left\{ \frac{d_{i,j}}{C_{i,j}} \right\} \\
\textbf{Subject to:} \quad & \sum_{j=1}^{n_i} d_{i,j} = D_i \\
& \forall \sigma_{i,j} \in \tau_i \ , \ d_{i,j} \geq C_{i,j}^{\min}
\end{aligned}
$$

which is equivalent to the following linear program:

$$
\begin{aligned}
& \forall \tau_i \in \tau, \\
\textbf{Maximize:} \quad & x \\
\textbf{Subject to:} \quad & \sum_{j=1}^{n_i} d_{i,j} = D_i \\
& \forall \sigma_{i,j} \in \tau_i \ , \ d_{i,j} \geq C_{i,j}^{\min} \\
& \forall \sigma_{i,j} \in \tau_i \ , \ \left\{ \frac{d_{i,j}}{C_{i,j}} \right\} \geq x
\end{aligned}
$$

This problem can be solved using effective linear programming techniques such as those presented in [Dantzig et al. 1955; Wright 1997].

7.6 Online Approach: An Efficient Algorithm

In this section, we explain how we can derive from the optimization problem presented in the previous section, an algorithm that optimally determines the segment intermediate

deadlines while considering the sufficient schedulability test expressed in Property 7.1. The proposed solution is a *greedy* algorithm. That is, it iteratively builds the optimal solution by fixing one intermediate deadline at each step. Whenever an intermediate deadline is determined, it will never be updated. By smartly choosing the order in which segments are considered, we can guarantee that the optimality of the global solution will not be affected by the local decision taken for a particular segment.

The motivations lying behind the implementation of an online algorithm determining the segment intermediate deadlines, is the following:
Let us assume that we are scheduling a dynamic task set. New tasks can join the system at any time and the task characteristics such as the number of threads in some segments may be altered during the schedule. Therefore, whenever a task τ_i releases a new (unknown) job in the system, we want to be able:

- to compute the segment intermediate deadlines minimizing the number of processors needed for the execution of τ_i;
- to check if the system is overloaded. In the case of an affirmative answer, the operating system should take appropriate decisions such as stopping the execution of some tasks, re-dispatching the workload between different clusters, switching a processor on, *etc.*

In the previous section, we showed that a linear programming technique can be used to optimally solve the problem of finding intermediate deadlines for a parallel task τ_i. Unfortunately, a scheduler cannot afford to solve a linear optimization problem at run-time. However, the problem studied in the previous section can be solved with a simple greedy algorithm. This algorithm has a low run-time complexity (as later proven in Section 7.6.2), thereby implying that it can be executed online.

7.6.1 Algorithm Presentation

Notations

Our algorithm must determine the optimal intermediate deadlines $d_{i,j}^{\text{opt}}$ for the set of segments $\sigma_{i,j} \in \tau_i$ (i.e., the intermediate deadlines minimizing the maximal segment density), while respecting the constraints (7.5) and (7.6) described in the previous section. The density corresponding to the optimal intermediate deadline $d_{i,j}^{\text{opt}}$ is the optimal density which

is defined as $\delta_{i,j}^{\text{opt}} \stackrel{\text{def}}{=} \frac{C_{i,j}}{d_{i,j}^{\text{opt}}}$.

In the following, we denote by S_i^ℓ the set of segments we still need to consider at the ℓ^{th} step of the greedy algorithm (i.e., the segments $\sigma_{i,j}$ that do not have an associated deadline yet). Hence, it initially holds that $S_i^1 \stackrel{\text{def}}{=} \{\sigma_{i,j} \in \tau_i\}$. Similarly, L_i^ℓ denotes the amount of time that may be distributed amongst the segments in S_i^ℓ, assuming that $L_i^1 \stackrel{\text{def}}{=} D_i$. Therefore, at the ℓ^{th} iteration, there is a segment $\sigma_{i,j}$ such that $S_i^{\ell+1} = S_i^\ell \setminus \sigma_{i,j}$, and $L_i^{\ell+1} = L_i^\ell - d_{i,j}^{\text{opt}}$.

We define the worst-case execution time of the remaining segments at iteration ℓ as

$$C_i^\ell \stackrel{\text{def}}{=} \sum_{\sigma_{i,j} \in S_i^\ell} C_{i,j} \tag{7.7}$$

and the average density at iteration ℓ as $\delta_i^\ell \stackrel{\text{def}}{=} \frac{C_i^\ell}{L_i^\ell}$. Notice that $C_i^1 = C_i$ and $\delta_i^1 = \delta_i$.

Similarly, the maximum instantaneous density of the set of segments S_i^ℓ is defined as

$$\delta_i^{\ell\max} \stackrel{\text{def}}{=} \max_{\sigma_{i,j} \in S_i^\ell} \{\delta_{i,j}\}$$

According to Constraint (7.6), the relative deadline $d_{i,j}$ of a segment $\sigma_{i,j}$ cannot be smaller than $C_{i,j}^{\min}$. Since $C_{i,j}^{\min}$ is a lower bound on the relative deadline of $\sigma_{i,j}$, we define the density upper bound of $\sigma_{i,j}$ as

$$\widehat{\delta_{i,j}} \stackrel{\text{def}}{=} \frac{C_{i,j}}{C_{i,j}^{\min}} \tag{7.8}$$

Table 7.2 summarizes all these new notations.

Core Idea of the Algorithm

Two situations can be encountered at any step ℓ of our algorithm. Either the density upper bound $\widehat{\delta_{i,j}}$ of every segment $\sigma_{i,j} \in S_i^\ell$ is greater than or equal to the average density $\delta_i^\ell \stackrel{\text{def}}{=} \frac{C_i^\ell}{L_i^\ell}$, or there is a segment $\sigma_{i,j} \in S_i^\ell$ such that $\widehat{\delta_{i,j}} < \delta_i^\ell$. In the first situation, we prove through Lemma 7.2 that the optimal solution consists in assigning intermediate deadlines to the segments such that all the segments get a density equal to the average density δ_i^ℓ. On the other hand, if there is a segment $\sigma_{i,j} \in S_i^\ell$ with a density which cannot be increased up

Table 7.2: Problem notations

Variables

$d_{i,j}$	Relative deadline of segment $\sigma_{i,j}$
$\delta_{i,j}$	Density of segment $\sigma_{i,j}$ ($\delta_{i,j} \stackrel{\text{def}}{=} \frac{C_{i,j}}{d_{i,j}}$)
$\delta_i^{\max}$	Maximum instantaneous density of τ_i ($\delta_i^{\max} \stackrel{\text{def}}{=} \max_{\sigma_{i,j} \in \tau_i} \left\{ \delta_{i,j} \right\}$)
$\delta^{\max}$	Maximum instantaneous density of τ ($\delta^{\max} \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \delta_i^{\max}$)

Properties

$\widehat{\delta_{i,j}}$	Upper bound on $\delta_{i,j}$ ($\widehat{\delta_{i,j}} \stackrel{\text{def}}{=} \frac{C_{i,j}}{C_{i,j}^{\min}}$)

Optimization problem

S_i^ℓ	Set of segments of τ_i that remain to consider at the ℓ^{th} step of the algorithm
L_i^ℓ	Amount of time that has still to be dispatched amongst the segments of S_i^ℓ
C_i^ℓ	Worst-case execution time of S_i^ℓ ($C_i^\ell \stackrel{\text{def}}{=} \sum_{\sigma_{i,j} \in S_i^\ell} C_{i,j}$)
δ_i^ℓ	Average density of S_i^ℓ on L_i^ℓ time units ($\delta_i^\ell \stackrel{\text{def}}{=} \frac{C_i^\ell}{L_i^\ell}$)
$d_{i,j}^{\text{opt}}$	Optimal value of $d_{i,j}$
$\delta_{i,j}^{\text{opt}}$	Density of $\sigma_{i,j}$ in the optimal solution ($\delta_{i,j}^{\text{opt}} \stackrel{\text{def}}{=} \frac{C_{i,j}}{d_{i,j}^{\text{opt}}}$)
$\delta_i^{\ell\max}$	Maximum density of the segments in S_i^ℓ ($\delta_i^{\ell\max} \stackrel{\text{def}}{=} \max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j} \right\}$)

to the average density δ_i^ℓ (i.e., $\widehat{\delta_{i,j}} < \delta_i^\ell$), then the optimal solution consists in minimizing the deadline of this segment $\sigma_{i,j}$ (and therefore maximizing its density) so as to maximize the remaining time for the execution of the other segments. This second claim is proven in Lemma 7.3.

All this reasoning is based on a simple property that can be expressed as follows:

Property 7.2

Let L_i^ℓ be the time that must be distributed amongst the deadlines of the segments belonging to S_i^ℓ. The maximum instantaneous density $\delta_i^{\ell\max}$ ($\stackrel{\text{def}}{=} \max_{\sigma_{i,j} \in S_i^\ell} \delta_{i,j}$) cannot be smaller than the average density δ_i^ℓ of S_i^ℓ (i.e., $\delta_i^{\ell\max} \geq \delta_i^\ell$, $\forall \ell$).

That is, the maximum of a set of values is never lower than the average value of this set.

Another interesting property which will be used into the proofs of Lemmas 7.2 and 7.3, is presented through Lemma 7.1 and its corollary.

Lemma 7.1

If the time L_i^ℓ that must be distributed amongst the segments belonging to a set S_i^ℓ increases, then $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j}^{\text{opt}} \right\}$ can only decrease.

Proof:

If the time L_i^ℓ increases by Δ time units, we can distribute Δ amongst the segments with the largest densities $\delta_{i,j}^{\text{opt}}$. The densities $\delta_{i,j}^{\text{opt}}$ of these segments are then reduced which in turn implies that $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j}^{\text{opt}} \right\}$ decreases. ■

Corollary 7.1

If the time L_i^ℓ that must be distributed amongst the segments belonging to a set S_i^ℓ decreases, then $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j}^{\text{opt}} \right\}$ of S_i^ℓ can only increase.

We now prove our claims on the optimal solutions.

Lemma 7.2

If, for every segment $\sigma_{i,j} \in S_i^\ell$, we have $\widehat{\delta_{i,j}} \geq \delta_i^\ell$, then, in order to minimize the maximum instantaneous density $\delta_i^{\ell\max}$ of S_i^ℓ, we must impose $\delta_{i,j} = \delta_i^\ell$ to every segment $\sigma_{i,j} \in S_i^\ell$ (i.e., $\delta_{i,j}^{\text{opt}} = \delta_i^\ell$, $\forall \sigma_{i,j} \in S_i^\ell$). Furthermore, we get $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j}^{\text{opt}} \right\} = \delta_i^\ell$.

Proof:

Let us assume that every segment $\sigma_{i,j}$ in S_i^ℓ has a density $\delta_{i,j}$ equal to δ_i^ℓ. We obviously get $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j}^{\text{opt}} \right\} = \delta_i^\ell$.

Furthermore, by Property 7.2, the maximum instantaneous density $\delta_i^{\ell\max}$ cannot be smaller than δ_i^ℓ. Therefore, because $\delta_i^{\ell\max}$ is defined as $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j} \right\}$, $\delta_i^{\ell\max}$ is minimum when $\delta_{i,j} = \delta_i^\ell$ for every segment $\sigma_{i,j}$ in S_i^ℓ. ■

Lemma 7.3

Let S_i^ℓ be a set of segments and δ_i^ℓ be its average density when we have L_i^ℓ time units to distribute between the deadlines of the segments in S_i^ℓ. If there is a segment $\sigma_{i,j} \in S_i^\ell$ such that $\widehat{\delta_{i,j}} < \delta_i^\ell$, then, in order to minimize the maximum density $\delta_i^{\ell\max}$ of S_i^ℓ on L_i^ℓ, we must impose $\delta_{i,j} = \widehat{\delta_{i,j}}$ (i.e., $\delta_{i,j}^{\text{opt}} = \widehat{\delta_{i,j}}$).

Proof:

Let C_i^ℓ be the total worst-case execution time of S_i^ℓ. Let us assume that $\sigma_{i,j}$ has a worst-case execution time $C_{i,j}$. We must determine the relative-deadline $d_{i,j}$ for $\sigma_{i,j}$ so that the maximum density of S_i^ℓ is minimized. Let $S_i^{\ell+1}$ denote the set $S_i^\ell \setminus \sigma_{i,j}$. We have $C_i^{\ell+1} \stackrel{\text{def}}{=} C_i^\ell - C_{i,j}$ and $L_i^{\ell+1} \stackrel{\text{def}}{=} L_i^\ell - d_{i,j}$.

By Property 7.2, $\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j} \right\}$ cannot be smaller than δ_i^ℓ. Since, by assumption, the maximum density $\widehat{\delta_{i,j}}$ reachable by $\sigma_{i,j}$ is smaller than δ_i^ℓ, it results that

$$\max_{\sigma_{i,j} \in S_i^\ell} \left\{ \delta_{i,j}^{\text{opt}} \right\} = \max_{\sigma_{i,j} \in S_i^{\ell+1}} \left\{ \delta_{i,j}^{\text{opt}} \right\} \tag{7.9}$$

The density of $\sigma_{i,j}$ can be expressed as $\delta_{i,j} \stackrel{\text{def}}{=} \frac{C_{i,j}}{d_{i,j}}$. Therefore, it holds that $\delta_{i,j} \times d_{i,j} = C_{i,j}$. Since $C_{i,j}$ is constant, if $\delta_{i,j}$ decreases then $d_{i,j}$ increases. Hence, the time $L_i^\ell - d_{i,j}$ that must be distributed amongst the segments belonging to $S_i^{\ell+1}$ decreases. Consequently, by Corollary 7.1, $\max_{\sigma_{i,j} \in S_i^{\ell+1}} \left\{ \delta_{i,j}^{\text{opt}} \right\}$ can only increase.

Thus, from Expression 7.9, $\delta_{i,j}$ must be maximized to minimize $\delta_i^{\ell\max}$. As a result, we have to impose $\delta_{i,j} = \widehat{\delta_{i,j}}$ which states the lemma. ■

Algorithm 7.1: Deadlines assignment algorithm.

Input: τ_i;

1 $S_i^1 := \{\sigma_{i,j} \in \tau_i\}$; `// Ordered by` $\widehat{\delta_{i,j}}$
2 $C_i^1 := C_i$;
3 $L_i^1 := D_i$;
4 **for** $\ell : 1 \rightarrow n_i$ **do**
5 $\quad \delta_i^\ell := \frac{C_i^\ell}{L_i^\ell}$;
6 $\quad \sigma_{i,j} :=$ first segment of S_i^ℓ (i.e., with the minimal $\widehat{\delta_{i,j}}$) ;
7 $\quad$ **if** $\widehat{\delta_{i,j}} < \delta_i^\ell$ **then**
8 $\quad\quad d_{i,j}^{\text{opt}} := C_{i,j}^{\text{min}}$; `// Rule 2`
9 $\quad$ **else**
10 $\quad\quad \forall \sigma_{i,j} \in S_i^\ell : d_{i,j}^{\text{opt}} := \frac{C_{i,j}}{\delta_i^\ell}$; `// Rule 1`
11 $\quad\quad$ break;
12 $\quad$ **end**
13 $\quad S_i^{\ell+1} := S_i^\ell \setminus \sigma_{i,j}$;
14 $\quad C_i^{\ell+1} := C_i^\ell - C_{i,j}$;
15 $\quad L_i^{\ell+1} := L_i^\ell - d_{i,j}^{\text{opt}}$;
16 **end**

Algorithm Description

Since $\delta_{i,j} = \frac{C_{i,j}}{d_{i,j}}$, we derive from Lemmas 7.2 and 7.3 the two following rules to minimize the maximum instantaneous density $\delta_i^{\ell\,\text{max}}$ at any step ℓ.

Rule 7.1

If, for every segment $\sigma_{i,j} \in S_i^\ell$ we have $\widehat{\delta_{i,j}} \geq \delta_i^\ell$ then $d_{i,j}^{\text{opt}} = \frac{C_{i,j}}{\delta_i^\ell}$ for all segments in S_i^ℓ.

Rule 7.2

Let $\sigma_{i,j}$ be a segment in S_i^ℓ. If $\widehat{\delta_{i,j}} < \delta_i^\ell$ then we have $d_{i,j}^{\text{opt}} = C_{i,j}^{\text{min}}$.

These two rules lead to Algorithm 7.1 which is used to determine the segment relative deadlines of a task τ_i.

First, the algorithm computes the maximum reachable density $\widehat{\delta_{i,j}}$ of every segment $\sigma_{i,j}$ belonging to τ_i and sorts the segments in an increasing $\widehat{\delta_{i,j}}$ order (Line 1). Then, at step ℓ, Algorithm 7.1 selects the segment $\sigma_{i,j}$ with the smallest density upper bound amongst the remaining segments, and compares $\widehat{\delta_{i,j}}$ with the average density δ_i^ℓ of S_i^ℓ (Line 6). If $\widehat{\delta_{i,j}} < \delta_i^\ell$ then we apply Rule 7.2 (Line 8). Otherwise, we can use Rule 7.1 to

compute the segment deadline (Line 10). Finally, whenever the optimal deadline $d_{i,j}^{\text{opt}}$ of a segment $\sigma_{i,j}$ has been determined, $\sigma_{i,j}$ is removed from the problem, and the values $S_i^{\ell+1}$, $C_i^{\ell+1}$ and $L_i^{\ell+1}$ are computed accordingly (Lines 13 to 15).

7.6.2 Optimality and Run-Time Complexity

Theorem 7.1

If, $\forall i, \sum_j C_{i,j}^{\min} \leq D_i$, Algorithm 7.1 provides an optimal solution to the problem of dispatching D_i time units amongst the n_i segments of τ_i.

Proof:

To prove the optimality of the solution proposed by Algorithm 7.1, we must prove that $\delta_i^{\max}$ is minimum and all constraints expressed by Equations 7.5 and 7.6 are respected.

1. Since the segments are ordered in an increasing $\widehat{\delta_{i,j}}$ order, we first compute the relative deadlines of the segments with the smallest $\widehat{\delta_{i,j}}$ values. Therefore, if $\widehat{\delta_{i,j}} \geq \delta_i^{\ell}$, then all the remaining segments in S_i^{ℓ} have $\widehat{\delta_{i,j}} \geq \delta_i^{\ell}$. Hence, we can apply Lemma 7.2 to determine the optimal deadline of $\sigma_{i,j}$ while minimizing $\delta_i^{\ell\max}$. It is exactly what is done at line 10 of Algorithm 7.1. On the other hand, if $\widehat{\delta_{i,j}} < \delta_i^{\ell}$, then, according to Lemma 7.3 we must maximize the density of $\sigma_{i,j}$ in order to minimize $\delta_i^{\max}$. Consequently, Line 8 in Algorithm 7.1 applies Rule 7.2 which is the direct consequence of Lemma 7.3.

   Hence, in both cases, Algorithm 7.1 applies the correct rule to minimize $\delta_i^{\max}$.

2. Proving that $\sum_j d_{i,j}^{\text{opt}} = D_i$ is straightforward if the algorithm enters the "else" bloc (Line 10). Indeed, by applying Rule 7.2, the algorithm dispatches the remaining L_i^{ℓ} times units between all the remaining segments in S_i^{ℓ}. Then, $\sum_{\sigma_{i,j} \in S_i^{\ell}} d_{i,j}^{\text{opt}} = L_i^{\ell}$. As, by construction (see Line 15), $L_i^{\ell} = D_i - \sum_{\sigma_{i,j} \notin S_i^{\ell}} d_{i,j}^{*}$, we have that $\sum_{\sigma_{i,j} \in \tau_i} d_{i,j}^{\text{opt}} = D_i$.

   We still need to prove that Algorithm 7.1 eventually enters the "else" bloc. By contradiction, let us assume that we always have $\widehat{\delta_{i,j}} < \delta_i^{\ell}$. It means that for the very last segment in S_i^{ℓ} (say, $\sigma_{i,a}$; then $C_i^{\ell} = C_{i,a}$), we have $\widehat{\delta_{i,a}} = \frac{C_{i,a}}{C_{i,a}^{\min}} < \delta_i^{\ell} = \frac{C_i^{\ell}}{L_i^{\ell}} = \frac{C_{i,a}}{L_i^{\ell}}$. Then, $C_{i,a}^{\min} > L_i^{\ell}$. As, by Lines 8 and 15 (and we never enter the "else"), $L_i^{\ell} = D_i - \sum_{j \neq a} C_{i,j}^{\min}$, we have that $\sum_j C_{i,j}^{\min} > D_i$, which contradicts with the hypothesis of the theorem.

3. For every segment $\sigma_{i,j}$ belonging to τ_i, Algorithm 7.1 determines $d_{i,j}^{\text{opt}}$ according to Rules 7.1 and 7.2. These rules are based on Lemmas 7.2 and 7.3, respectively. Since Lemmas 7.2 and 7.3 impose that $\delta_{i,j} \leq \widehat{\delta_{i,j}}$, it yields $d_{i,j}^{\text{opt}} \geq C_{i,j}^{\min}$ from Equation 7.8. Hence, the constraint formulated by Equation 7.6 is respected.

Consequently, Algorithm 7.1 optimally resolves the optimization problem described in Section 7.5. ■

Theorem 7.2

The computational complexity to determine the optimal segment deadlines for a task τ_i with Algorithm 7.1 is $O(n_i \times \log n_i)$.

Proof:

Algorithm 7.1 starts by sorting the n_i segments belonging to τ_i in an increasing $\widehat{\delta_{i,j}}$ order (Line 1). This manipulation has a computational complexity of $O(n_i \times \log n_i)$. Then, the *for* loop executed at Lines 6 to 15, iterates at most n_i times (once for each segment of τ_i). Since the computation time of $d_{i,j}^{\text{opt}}$ is constant, the computational complexity of the loop is $O(n_i)$. Therefore, the total computational complexity of Algorithm 7.1 is dominated by the sorting algorithm, and is then $O(n_i \times \log n_i)$. ■

7.7 Resource Augmentation Bounds

Let τ^* denote the task set τ when intermediate deadlines have been allocated to the segments using one of the two techniques proposed in Sections 7.5 and 7.6. We first prove in Lemma 7.4 that every segment $\sigma_{i,j}$ of any task in τ^* has a density $\delta_{i,j} < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$. This property is then used to prove that both the offline and the online technique presented in Sections 7.5 and 7.6, respectively, achieve a resource augmentation bound on the processor speed of 2 when tasks have implicit deadlines and threads in τ^* are scheduled using algorithms such as U-EDF, LLREF or DP-Wrap.

We then use the same approach to derive a resource augmentation bound of 3 for EDF-US[$\frac{1}{2}$] [Srinivasan and Baruah 2002] and EDF$^{(k_{\min})}$ [Baker 2005]. Note that the reasonings utilized to derive these bounds have been strongly influenced by those provided in [Saifullah et al. 2011].

Lemma 7.4

Every segment $\sigma_{i,j}$ of every task in τ^* has a density

$$\delta_{i,j} < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$$

Proof:

According to Algorithm 7.1, we have $\delta_i^\ell = \frac{C_i^\ell}{L_i^\ell}$. Furthermore, Algorithm 7.1 implies that $L_i^\ell = D_i - \sum_{\sigma_{i,k} \notin S_i^\ell} C_{i,k}^{\min}$ (because, as soon as the condition of line 7 is not valid anymore, the loop ends) and using Expression 7.7, we get

$$\delta_i^\ell = \frac{\sum\limits_{\sigma_{i,q} \in S_i^\ell} C_{i,q}}{D_i - \sum\limits_{\sigma_{i,k} \notin S_i^\ell} C_{i,k}^{\min}} \tag{7.10}$$

Moreover, because $S_i^\ell \subseteq \tau_i$, it holds that

$$\sum_{\sigma_{i,q} \in S_i^\ell} C_{i,q} \leq \sum_{\sigma_{i,q} \in \tau_i} C_{i,q} = C_i \tag{7.11}$$

and

$$\sum_{\sigma_{i,k} \notin S_i^\ell} C_{i,k}^{\min} < \sum_{\sigma_{i,k} \in \tau_i} C_{i,k}^{\min} \tag{7.12}$$

Thus, using Expressions 7.11 and 7.12 in Expression 7.10, we get

$$\delta_i^\ell < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$$

Because at every step ℓ, Algorithm 7.1 imposes:

- $\delta_{i,j} = \widehat{\delta_{i,j}}$ if $\widehat{\delta_{i,j}} < \delta_i^\ell$
- $\delta_{i,j} = \delta_i^\ell$ otherwise.

We have in both cases: $\delta_{i,j} \leq \delta_i^\ell < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$.

Hence, the lemma is proven for Algorithm 7.1.

Since the resolution of the optimization problem presented in Section 7.5, minimizes the maximum density reachable by the segments composing τ_i, and because

Algorithm 7.1 gets $\delta_{i,j} < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$ for every segment $\sigma_{i,j}$, it must hold that $\delta_{i,j} < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$ with the optimization algorithm.

This states the lemma. ■

Theorem 7.3

If a multi-threaded task set τ where tasks have implicit deadlines is schedulable on m unit-speed identical processors using any optimal algorithm, then the task set τ^* is schedulable on m identical processors of speed 2, using any algorithm respecting Property 7.1.

Proof:

According to Lemma 7.4 any segment of any task in τ^* has a density $\delta_{i,j} < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$ on a unit-speed processor. Hence, the maximum instantaneous density achievable in τ^* is given by

$$\delta^{\max} = \sum_{\tau_i \in \tau} \max_j \left\{ \delta_{i,j} \right\} < \sum_{\tau_i \in \tau} \left\{ \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}} \right\}$$

Furthermore, if the processors in the platform are v times faster, then all the worst-case execution times are divided by v. Hence, the maximum instantaneous density becomes

$$\delta^{\max,v} < \sum_{\tau_i \in \tau} \left\{ \frac{\frac{C_i}{v}}{D_i - \frac{\sum_{k=1}^{n_i} C_{i,k}^{\min}}{v}} \right\}$$

It was stated in Section 7.3 that a task τ_i can respect its deadline only if $\sum_{j=1}^{n_i} C_{i,j}^{\min} \leq D_i$, implying that

$$\delta^{\max,v} < \sum_{\tau_i \in \tau} \left\{ \frac{\frac{C_i}{v}}{D_i - \frac{D_i}{v}} \right\} = \frac{1}{v-1} \sum_{\tau_i \in \tau} \frac{C_i}{D_i}$$

A task set with implicit deadlines is schedulable by any optimal algorithm on m unit-speed processors if and only if $\sum_{\tau_i \in \tau} \frac{C_i}{D_i} \leq m$ (Theorem 3.4). Hence,

$$\delta^{\max,v} < \frac{1}{v-1} m \tag{7.13}$$

Furthermore, any algorithm respecting Property 7.1 imposes that $\delta^{\max,v} \leq m$. This

condition is therefore respected if

$$\frac{1}{v-1}m \leq m$$

therefore, by imposing a processor speed v of 2, we guarantee that no task will ever miss any deadline. ■

Note that the same approach can be used to derive a resource augmentation bound for other global scheduling algorithms as long as their schedulability tests are based on densities. For instance, it can be proven that EDF-US[$\frac{1}{2}$] — an hybrid scheduling algorithm which gives the highest priority to tasks (i.e., threads in our case) with a density above a threshold of $\frac{1}{2}$ and uses EDF to schedule the others — has a resource augmentation bound of 3 using the schedulability test proposed in [Baker 2005; Baker and Baruah 2008]. This bound is also true for EDF$^{(k_{\min})}$ since this algorithm has the same schedulability test [Baker 2005].

Theorem 7.4

If a multi-threaded task set τ where tasks have implicit deadlines is schedulable on m unit-speed identical processors using any optimal algorithm, then the task set τ^* is schedulable on m identical processors of speed 3, using EDF-US[$\frac{1}{2}$] or EDF$^{(k_{\min})}$.

Proof:

According to Lemma 7.4 any segment of any task in τ^* has a density $\delta_{i,j} < \frac{C_i}{D_i - \sum_{k=1}^{n_i} C_{i,k}^{\min}}$ on a unit-speed processor. Then, using the same reasoning as in Theorem 7.3, we obtain Expression 7.13 repeated hereafter.

$$\delta^{\max,v} < \frac{1}{v-1}m$$

According to the schedulability tests proposed in [Baker 2005], a task set is schedulable with EDF-US[$\frac{1}{2}$] (and EDF$^{(k_{\min})}$, respectively) on a v-speed processor if

$$\delta^{\max,v} \leq \frac{m+1}{2}$$

This sufficient schedulability condition is therefore respected if

$$\frac{1}{v-1}m \leq \frac{m+1}{2}$$

or even if

$$\begin{aligned} & \frac{1}{v-1}m \leq \frac{m}{2} \\ \Leftrightarrow\ & 2 \leq v-1 \\ \Leftrightarrow\ & v \geq 3 \end{aligned}$$

It results that by imposing a processor speed $v = 3$, we can guarantee that no task will ever miss any deadline. ■

Same kind of results could be obtained for other scheduling algorithms, including partitioned scheduling algorithms as it was done in [Lakshmanan et al. 2010] and [Saifullah et al. 2011]. However, we stop here as the presentation of these results could start looking fastidious but of small interests due to their redundancy.

7.8 Task Model Generalization

In the model considered so far, we assumed that threads of a same segment can be executed in parallel, but segments of a same task cannot. Although this task representation is realistic for many applications, it is also simplistic. For instance, let us consider the task τ_i represented on Figure 7.3(a). Its execution starts with a single thread (numbered 1). Then, the first thread forks in three threads indexed 2, 3 and 4. However, contrarily to threads 2 and 3, at a certain point of its execution the thread 4 can be parallelized in two other threads numbered 5 and 6. Finally, all active threads must complete their execution before executing the last thread 7. In this example, we have one segment composed of threads 2 and 3 which is running concurrently with two other segments containing thread 4 and threads 5 and 6, respectively.

Therefore, in this section we generalize the task model, allowing several segments of the same task to run in parallel. Then, we propose variations of the offline approach to minimize the number of processors needed to schedule such parallel tasks.

7.8.1 One Parallel Segment

Let us first consider a task τ_i (pictured on Figure 7.3(b)) containing one (and only one) segment $\sigma_{i,\ell}$ running in parallel with p consecutive segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$. We remind

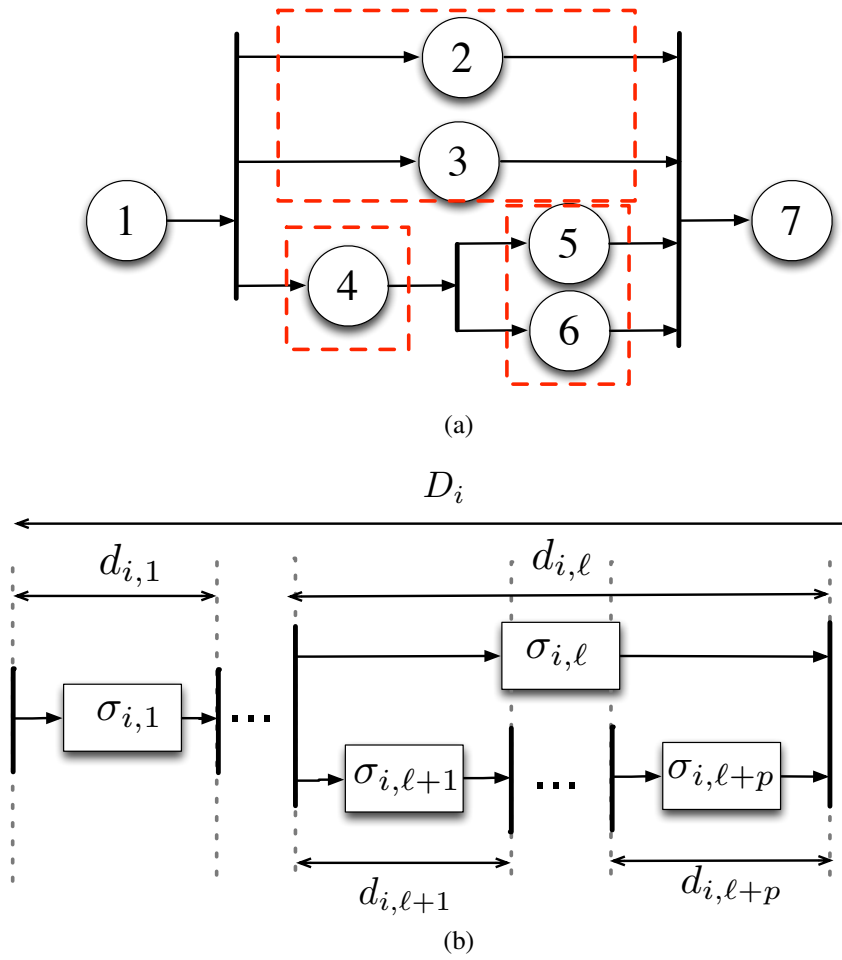


Figure 7.3: (a) Generalized task model where threads can run in parallel with several successive segments. (b) General representation of the same problem where a segment $\sigma_{i,\ell}$ is simultaneously active with segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$.

the reader that each segment is composed of several threads (as illustrated in the example of Figure 7.3(a)).

We say that the segments $\sigma_{i,\ell}$ to $\sigma_{i,\ell+p}$ belong to a *parallel phase*. Three different phases are identifiable in this task: a sequential phase composed of the $\ell-1$ first segments, a parallel phase with $p+1$ segments, and a last sequential phase, with $n_i-\ell-p$ segments[2]. The parallel phase is composed of two branches: a first branch containing only one segment $\sigma_{i,\ell}$, and a second branch constituted of p successive segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$. For the sake of clarity, we assume that there is only one parallel phase. However, the solution presented in this section is straightforwardly extendable to tasks containing

[2]Note that when we talk about parallel or sequential phases here, we are actually referring to the segment executions. The threads belonging to these segments can still execute in parallel even during a sequential phase.

several parallel phases (see Section 7.8.3 for the most general solution).

The technique computing the segment intermediate deadlines is similar to the offline approach proposed in Section 7.5. Since the instantaneous density of a task at time t is equal to the sum of the densities of its segments running at time t, we define an optimization problem considering that the densities of segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$ are increased by the density $\delta_{i,\ell}$ of the segment $\sigma_{i,\ell}$ which is executed concurrently. That is, we define $\delta'_{i,j}$ such that

$$\delta'_{i,j} \stackrel{\text{def}}{=} \begin{cases} \delta_{i,j} + \delta_{i,\ell} & \text{if } j \in [\ell+1, \ldots, \ell+p] \\ \delta_{i,j} & \text{otherwise} \end{cases}$$

Furthermore, as we can see on Figure 7.3(b), the deadline $d_{i,\ell}$ of the segment $\sigma_{i,\ell}$ must equal the sum of the deadlines allocated to segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$, i.e., we must have $\sum_{q=1}^{p} d_{i,\ell+q} = d_{i,\ell}$.

Therefore, the optimization problem can be expressed as follows: $\forall \tau_i \in \tau$,

$$\textbf{Minimize:} \quad \max_{j \neq \ell} \left\{ \delta'_{i,j} \right\} \tag{7.14}$$

$$\textbf{Subject to:} \quad \sum_{j \neq \ell} d_{i,j} = D_i \tag{7.15}$$

$$\forall \sigma_{i,j} \in \tau_i \ , \ d_{i,j} \geq C_{i,j}^{\min} \tag{7.16}$$

$$\sum_{q=1}^{p} d_{i,\ell+q} = d_{i,\ell} \tag{7.17}$$

$$\text{where } \delta'_{i,j} \stackrel{\text{def}}{=} \begin{cases} \delta_{i,j} + \delta_{i,\ell} & \text{if } j \in [\ell+1, \ldots, \ell+p] \\ \delta_{i,j} & \text{otherwise} \end{cases}$$

This new optimization problem is *non-linear*. However, the constraints and objective functions are convex. Hence, a non-linear programming technique such as [Wright 1997] can be used to efficiently solve this new problem.

7.8.2 Splitting the Parallel Segment

The approach proposed above is rather pessimistic. Indeed, it assumes that the segment $\sigma_{i,\ell}$ is uniformly executed in parallel with the segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$, meaning that it increases the densities of all segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$ by the same value $\delta_{i,\ell}$. That is, $\sigma_{i,\ell}$ is fairly spread amongst the segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$. A better technique consists in dispatching $\sigma_{i,\ell}$'s workload so that more work is executed in parallel with segments with low

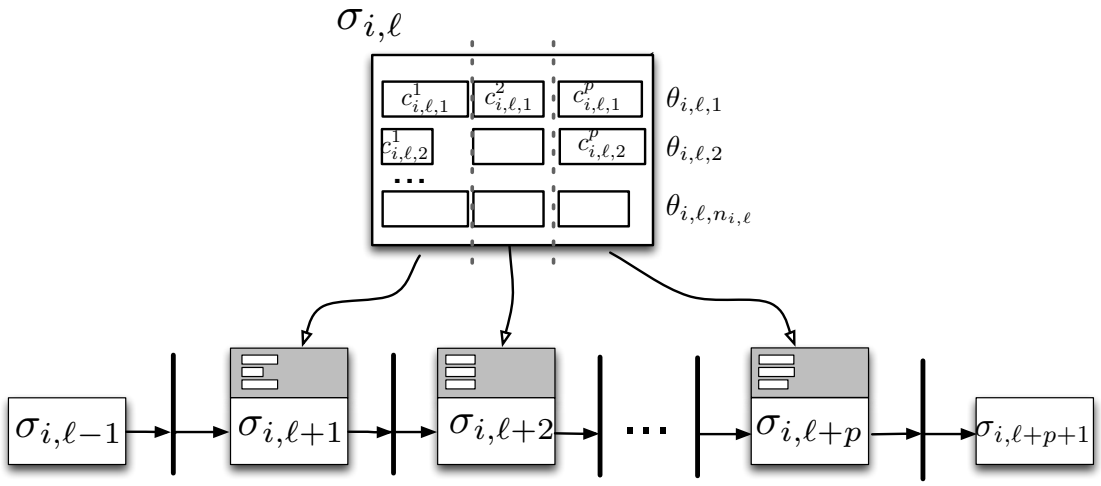


Figure 7.4: Work of segment $\sigma_{i,\ell}$ being dispatched amongst the p successive segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$. Each thread of $\sigma_{i,\ell}$ is split in p parts.

densities and less work on densest segments. Hence, the maximum on the segment densities constituting the task τ_i may be smaller, thereby reducing the number of processors needed to schedule this task.

To do so, every thread $\theta_{i,\ell,k}$ belonging to $\sigma_{i,\ell}$ is divided in p (i.e., the length of the parallel phase) parts associated with the p segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$ (see Figure 7.4). Each part has a worst-case execution time $c^q_{i,\ell,k}$ representing the portion of $C_{i,\ell,k}$ executed in parallel with segment $\sigma_{i,\ell+q}$ ($q \in [1,\ldots,p]$). This quantity is a new variable in our optimization problem. Note that at the end of the dispatching of any thread $\theta_{i,\ell,k}$ between the segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$, it must hold that $\sum_{q=1}^{p} c^q_{i,\ell,k} = C_{i,\ell,k}$. That is, the sum of the work executed for $\theta_{i,\ell,k}$ in each segment must still be equal to the total worst-case execution time of $\theta_{i,\ell,k}$.

We re-define the optimization problem as follows: $\forall \tau_i \in \tau$,

$$\textbf{Minimize:} \quad \max_{j \neq \ell} \left\{ \delta''_{i,j} \right\} \tag{7.18}$$

$$\textbf{Subject to:} \quad \sum_{j \neq \ell} d_{i,j} = D_i \tag{7.19}$$

$$\forall \sigma_{i,j} \in \tau_i \ , \ d_{i,j} \geq C^{\min}_{i,j} \tag{7.20}$$

$$\forall \theta_{i,\ell,k} \in \sigma_{i,\ell} \ , \ \sum_{q=1}^{p} c^q_{i,\ell,k} = C_{i,\ell,k} \tag{7.21}$$

$$\forall j \in [\ell+1,\ldots,\ell+p], \forall k, \ d_{i,j} \geq c^{j-\ell}_{i,\ell,k} \tag{7.22}$$

$$\text{where } \delta''_{i,j} \stackrel{\text{def}}{=} \begin{cases} \frac{C_{i,j}+\sum_{k=1}^{n_{i,\ell}} c_{i,\ell,k}^{j-\ell}}{d_{i,j}} & \text{if } j \in [\ell+1,\ldots,\ell+p] \\ \delta_{i,j} & \text{otherwise.} \end{cases}$$

Constraint (7.21) ensures that the entire workload of $\sigma_{i,\ell}$ is dispatched amongst the segments $\sigma_{i,\ell+1}$ to $\sigma_{i,\ell+p}$. On the other hand, constraint (7.22) makes sure that the execution time allotted to a thread $\theta_{i,\ell,k}$ in a segment $\sigma_{i,q}$ of the parallel phase is not greater than the deadline of this segment (because otherwise this thread should be executed on more than one processor simultaneously to be able to respect its deadline, which is of course forbidden).

As for the case of one parallel segment uniformly spread in the parallel phase, the problem is non linear but is still convex, and can therefore be solved efficiently [Wright 1997].

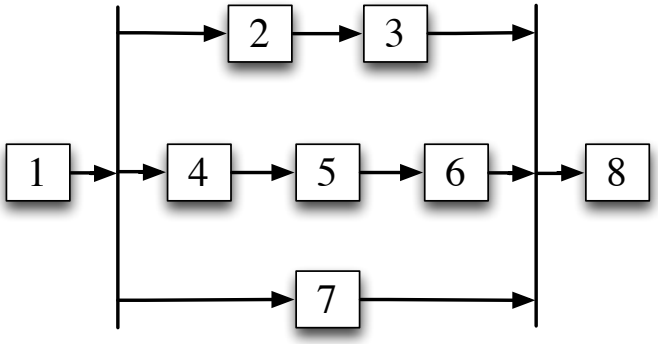
7.8.3 General Representation of a Parallel Task with a DAG

The model defined in the previous section can still be further generalized. Instead of considering one branch containing only one segment and another branch containing several segments, we may consider that the parallel phase is composed of several branches with, respectively, $p_1, p_2, p_3, \ldots$ segments. Figure 7.5(a) shows an example where $p_1 = 2$, $p_2 = 3$ and $p_3 = 1$. In this case, finding the maximal density of a task requires to consider all the combinations of segments that may run concurrently. For instance, in Figure 7.5(a), segment number 1 will never run in parallel with any other segment, but segments 2, 4 and 7 may run in parallel. In this example, all the following combinations of segments running in parallel are possible: $\{1\}$, $\{2,4,7\}$, $\{2,5,7\}$, $\{2,6,7\}$, $\{3,4,7\}$, $\{3,5,7\}$, $\{3,6,7\}$, $\{8\}$. Then, because the density of task running at time t is equal to the sum of the densities of its segments running at time t, in order to minimize the maximum density that can be reached by the task τ_i at any time t, we now have to minimize the following expression in place of (7.14):

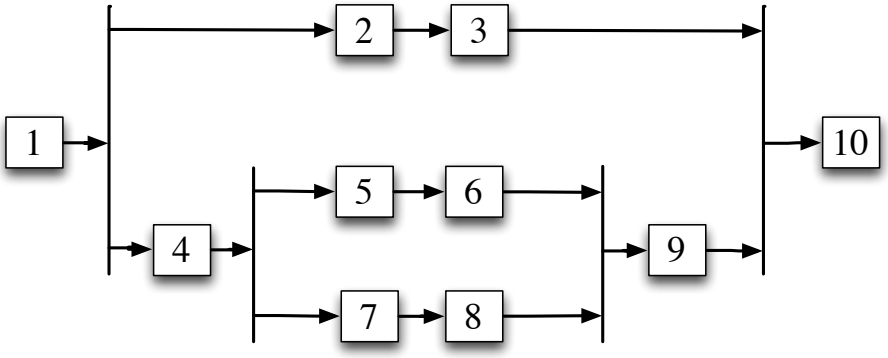
$$\max\{\delta_{i,1}, \delta_{i,2}+\delta_{i,4}+\delta_{i,7}, \delta_{i,2}+\delta_{i,5}+\delta_{i,7}, \ldots, \delta_{i,8}\}$$

Furthermore, as we did with constraints (7.15) and (7.17), we need to make sure that

$$\begin{aligned} D_i &= d_{i,1}+d_{i,2}+d_{i,3}+d_{i,8} \\ &= d_{i,1}+d_{i,4}+d_{i,5}+d_{i,6}+d_{i,8} \\ &= d_{i,1}+d_{i,7}+d_{i,8} \end{aligned}$$



(a)



(b)

Figure 7.5: Two examples of more general models. (a) A task containing a parallel phase composed of several branches of various sizes. (b) A task described by a DAG.

There is then as many such constraints as there are branches in task τ_i. Of course, we still need to make sure that $\forall \sigma_{i,j}, d_{i,j} \geq C_{i,j}^{\min}$ (constraint (7.16)).

More generally, we may consider any DAG to represent the dependencies between segments, as in Figure 7.5(b). Our objective function needs now to consider any set of segments that could possibly run in parallel. Note that such a segment combination is called a *maximal anti-chain* in the literature. Algorithms enumerating all the maximal anti-chains in a graph can be found in [Tsukiyama et al. 1977]. In our example, there are 14 maximal anti-chains which are partially enumerated hereafter:
$\{1\}, \{2,4\}, \{2,5,7\}, \{2,5,8\}, \{2,6,7\}, \ldots, \{3,9\}, \{10\}$.

The objective function that must be minimized is therefore:

$$\max\{\delta_{i,1}, \delta_{i,2} + \delta_{i,4}, \delta_{i,2} + \delta_{i,5} + \delta_{i,7}, \ldots, \delta_{i,3} + \delta_{i,9}, \delta_{i,10}\}$$

Even though there are only 14 possible segment combinations in our example, in some

particular cases, the number of maximal anti-chains could grow exponentially with the number of segments. Furthermore, the number of constraints depends on the structure of the DAG: for any possible "path" going through the graph (called a *maximal chain* in the literature), the sum of intermediate deadlines has to be equal to D_i. In our example, we have only 3 maximal chains ($\{1,2,3,10\}, \{1,4,5,6,9,10\}$ or $\{1,4,7,8,9,10\}$). Hence, for each maximal chain we get one more constraint. Note that the number of maximal chains could also grow exponentially with the number of segments in some pathological cases.

More formally, let MA_i (respectively MC_i) denote the set of maximal anti-chains (respectively, the set of maximal chains) of task τ_i. Then, let $A_{i,\ell}$ (respectively $B_{i,\ell}$) be a set of segments corresponding to a maximal anti-chain (respectively, a maximal chain) belonging to MA_i (respectively MC_i). The optimization problem that minimizes the maximum instantaneous density for a task τ_i that is represented by a DAG, can be enunciated as follows:

$$\textbf{Minimize:} \quad \max_{A_{i,\ell} \in \mathrm{MA}_i} \left\{ \sum_{\sigma_{i,j} \in A_{i,\ell}} \delta_{i,j} \right\} \tag{7.23}$$

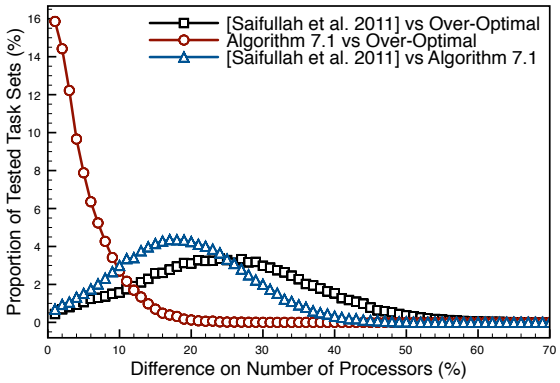
$$\textbf{Subject to:} \quad \forall B_{i,\ell} \in \mathrm{MC}_i \ , \ \sum_{\sigma_{i,j} \in B_{i,\ell}} d_{i,j} = D_i \tag{7.24}$$

$$\forall \sigma_{i,j} \in \tau_i \ , \ d_{i,j} \geq C_{i,j}^{\min} \tag{7.25}$$

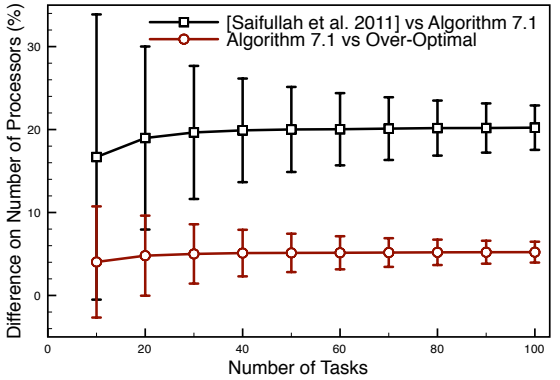
Note that this last optimization problem is actually a generalization of the optimization problem presented in Section 7.5 for the simpler parallel task model where all segments are executed sequentially. Indeed, if all segments are executed sequentially, there is only one maximal chain (i.e., a path from the first to the last segment) which contains all segments constituting the task τ_i. Hence, the constraint (7.24) becomes identical to the constraint (7.5). Furthermore, if all segments are executed sequentially, each individual segment is a maximal anti-chain (i.e., only one segment can be executed at a time). The objective function (7.23) therefore reduces to the objective function (7.4) of the simpler problem studied in the previous sections.

7.9 Simulation Results

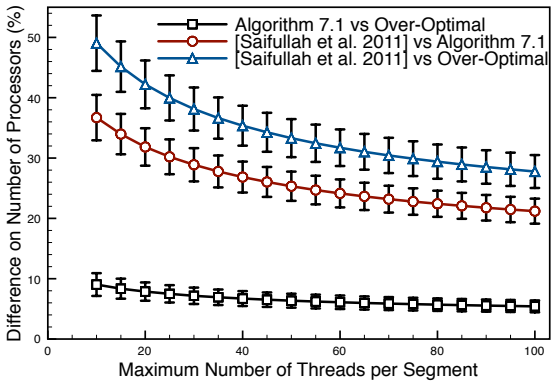
We evaluated the performances of Algorithm 7.1 through three different experiments. Note that Algorithm 7.1 provides an optimal solution for the optimization problem pre-



(a)



(b)



(c)

Figure 7.6: Simulation results comparing the number of processors needed to schedule a task set τ, when using Algorithm 7.1, the algorithm proposed in [Saifullah et al. 2011] and the "over-optimal" schedulability bound ($m = \sum_{\tau_i \in \tau} \delta_i$).

sented in Section 7.5. Hence, the simulation results discussed in this section also hold for the offline technique. Furthermore, the generalization of the offline technique presented in

Section 7.8 cannot be compared with any other solution since our work is the first which addressed this problem. Note that the offline approach always found a solution in less than a second. This is far more than reasonable for an algorithm executed offline.

In the first experiment, we randomly generated $100,000$ task sets and computed the number of processors needed to ensure that all tasks respect their deadlines. We compared our method with the methodology proposed in [Saifullah et al. 2011]. We also compared both solutions with a lower bound on this number of processors, obtained by summing up the density δ_i of all tasks (i.e., $m = \sum_{\tau_i \in \tau} \delta_i$). We named this bound "over-optimal" as even an optimal algorithm cannot always reach this bound when the tasks are sporadic with implicit deadlines (this claim was proven in [Lakshmanan et al. 2010]) and today's existing algorithm cannot guarantee to find a scheduling solution for multi-threaded sporadic tasks with constrained deadlines with the same bound on the total density. Nevertheless, it gives a good idea on the quality of the solution found with Algorithm 7.1.

In our tests, each task set is composed of 50 tasks. Each task has a number of segments n_i randomly chosen in a uniform distribution within $[1,30]$. To be coherent with the model presented in [Saifullah et al. 2011], we assume that all threads in a segment $\sigma_{i,j}$ have an identical worst-case execution time $C_{i,j}^{\min}$ randomly chosen within $[1,100]$[3]. The number of threads $n_{i,j}$ belonging to a segment $\sigma_{i,j}$ is randomly chosen within $[1,50]$. The deadline D_i of the task τ_i belongs to the interval extending from $\sum_{\sigma_{i,j} \in \tau_i} C_{i,j}^{\min}$ to $\sum_{\sigma_{i,j} \in \tau_i} n_{i,j} \times C_{i,j}^{\min}$. Indeed, as stated in Section 7.3, the task set is not feasible if $D_i < \sum_{\sigma_{i,j} \in \tau_i} C_{i,j}^{\min}$. On the other hand, all threads in τ_i can be executed consecutively (without any kind of parallelism) and still respect the task deadline when $D_i > \sum_{\sigma_{i,j} \in \tau_i} n_{i,j} \times C_{i,j}^{\min}$. Hence the solution would have been trivial and identical for Algorithm 7.1 and the method proposed in [Saifullah et al. 2011]. In all experiments, we assume that the tasks are scheduled with an optimal algorithm such as U-EDF, LRE-TL or DP-Wrap.

The results are presented in Figure 7.6(a). The curve *"[Saifullah et al. 2011] vs Algorithm 1"* in Figure 7.6(a) for instance, represents the distribution of the task sets regarding the difference between the number of processors needed with Algorithm 7.1 and the solution provided by [Saifullah et al. 2011]. We can observe that our algorithm performs quite well against the "over-optimal" bound. Indeed, in average, it needs less than 5% more processors than the "over-optimal" bound (which may not be reachable). The median value

[3]Notice that the fact that all threads have the same worst-case execution time does not impact the quality of the simulation results. Indeed, the only parameters actually used in Algorithm 7.1 are $C_{i,j}$ and $C_{i,j}^{\min}$. The solution is therefore independent of the particular $C_{i,j,k}$ values.

is even under 4%. On the other hand, the algorithm proposed in [Saifullah et al. 2011] needs in average 25.5% processors more than the over-optimal bound. Furthermore, its distribution is widely spread leading to a standard deviation greater than 11.5%. Therefore, the algorithm in [Saifullah et al. 2011] logically needs almost 20% more processors than our methodology (in average). Even worst, we detected in some particular cases, a difference of 210% between the number of processors needed with our solution and the algorithm proposed in [Saifullah et al. 2011]. That is, the method presented in [Saifullah et al. 2011] may need three times more processors than ours for some specific task sets.

The second experiment gives the average value of the relative difference between the number of processors needed with the three techniques when the number of tasks in τ varies within $[1,100]$. Each point in Figure 7.6(b) is the result of $10,000$ simulations. We can observe that the average value is small when there are few tasks in the system, but increases when the number of tasks grows. Inversely, the standard deviation is high for small task systems but decreases when the system grows. We note that the average difference between the number of processors needed with [Saifullah et al. 2011] and our solution seems to stabilize around 20% when the number of tasks is greater than 20. On the other hand, the average relative difference on m when we compare our algorithm with the over-optimal solution, never exceeds 6%. This means that even if we constrain the system by imposing deadlines to the task segments, the number of processors needed when using our methodology in conjunction with a scheduling algorithm such as U-EDF, remains close to the minimum number of processors which could ever be reached.

Finally, the third experiment shows the sensitivity of the solutions produced by Algorithm 7.1 and the algorithm proposed in [Saifullah et al. 2011], to the maximum number of threads $n_{i,j}$ composing each segment $\sigma_{i,j}$ (see Figure 7.6(c)). Again, each point is the result of $10,000$ simulations. The variation of the average value of the relative difference between our solution and the algorithm in [Saifullah et al. 2011] is due to the fact that the methodology presented in [Saifullah et al. 2011] divides the segments in two groups: the light and the heavy segments. The distinction between these two kinds of segments is based on the number of threads $n_{i,j}$ composing each segment $\sigma_{i,j}$. When all segments are considered as being heavy, the solution proposed by [Saifullah et al. 2011] is optimal and identical to ours. On the other hand, when all segments are light, the solution proposed by [Saifullah et al. 2011] is almost systematically suboptimal. Since there are more and more task sets with only heavy segments when the maximum value of $n_{i,j}$ increases, and since there are more and more task sets with only light segments when the maximum value of $n_{i,j}$ decreases, the trend of Figure 7.6(c) can easily be explained.

7.10 Conclusion

In this chapter, we proposed two new techniques determining the relative deadlines that must be applied to each segment belonging to a parallel task τ_i in order to optimize the number of processors needed to schedule the task set. The first approach consists in an optimization problem that can be solved offline. The second approach is a greedy algorithm which optimally solves the optimization problem with a complexity of $O(n_i \times \log(n_i))$.
The advantage of the first technique is that it can be easily extended to very general and realistic models of tasks, where segments, instead of being sequential, are organized in a DAG. However, this generalization is at the expense of an increasing complexity of the optimization problem.
The advantage of the second technique lies in its very low complexity, which enables the execution of this algorithm online. Hence, this second solution is particularly well suited for dynamic task systems where all task information are not known at design time but may be modified at run-time.

We proved that both techniques achieve a resource augmentation bound on the processor speed of 2 when tasks have implicit deadlines and threads are scheduled with algorithms such as U-EDF, LRE-TL or DP-Wrap. In order to show that the argumentation proving this bound can be utilized for other scheduling algorithms, the same reasoning was then applied to prove a resource augmentation bound of 3 for EDF-US$\left[\frac{1}{2}\right]$ and EDF$^{(k_{\min})}$. We also showed through simulations that our methodology to compute the segment deadlines may drastically improve the number of processors needed in the processing platform compared to previous works (up to three times less). Furthermore, even if we constrain the system by imposing deadlines to the task segments, the number of processors needed when using our methodology in conjunction with a scheduling algorithm such as U-EDF, remains close to the number of processors needed with an ideal "over-optimal" scheduling algorithm which may even not exist.

CHAPTER

8 Conclusion and Perspectives

Real-time systems are almost everywhere and their number grows every day. We find them in domains as diversified as the medical, telecommunication, aerospace, automotive or military industry. They are characterized by their need of respecting strict timing constraints. Indeed, missing deadlines in a real-time system may cause a diminution of the quality of service provided by the system but may also endanger the safety of the product, the environment or the user. For instance, if the regulation system of a power-plant stops working or if the auto-pilot of an airplane does not update the altitude correctly, the consequences may be dramatic. Real-time system designers could therefore be tempted to over-dimension the computing system. However, this may be very costly and impractical in many situations where resources are strictly limited (e.g., in aero-spatial systems such as satellites or in any device with a limited source of energy). An other approach consists in providing an accurate mathematical proof certifying that the system will work and meet all its task deadlines. This is the solution that was investigated in this thesis. This approach previously showed to be very efficient in the case of uniprocessor systems. However, nowadays, real-time systems are facing a new challenge. The computing platform architectures have changed and the uniprocessor platforms have now given place to multiprocessor systems. Already in 1969, it was stated that the presence of more than one processor in the computing platform brings non trivial problems to the real-time scheduling theory [Liu 1969a]. Since then, the real-time community worked hard to find effective scheduling algorithms for multiprocessor platforms.

As exposed in Chapter 3, a scheduling algorithm that is optimal has the advantage to respect all the task deadlines of any feasible task set. Hence, the utilization of the computing platform is maximal. The only schedulability test which is known today is based on the total density of the task set. Hence, either the total density of the task set is smaller than the number of processors, in which case the task set is feasible, or the total density is greater than m, thereby implying that the task set is infeasible if the tasks have implicit deadlines but might be feasible otherwise. However, there is no feasibility test in the latter case and it was proven that no online scheduling algorithm can be optimal for

Discrete-time	PFair	BFair	Unfair
Periodic	PF, PD, PD^2	PL, BF, **BF^2**	/
Sporadic	PD^2	**BF^2**	/
Dynamic	PD^2	**BF^2**	/

Table 8.1: Existing optimal algorithms for multiprocessor platforms for discrete-time systems (assuming $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$). The algorithms developed in this thesis are written in bold and colored in red.

such task sets [Fisher et al. 2010]. Within all this dissertation, we therefore focused on systems respecting the constraints $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$.

Many optimal multiprocessor scheduling algorithms, have been designed over the last 10 years. They can be classified in two broad categories regarding if they target discrete or continuous-time systems. Optimal algorithms for continuous-time systems are more numerous as they are simpler to design. However, discrete-time models are more in line with existing embedded real-time operating systems. In this thesis, we thus proposed new optimal scheduling algorithms for both discrete and continuous-time environments. Hence, we provide solutions that can be implemented in real-time operating systems respecting either of these two models of time representation.

Throughout this document, we addressed the problem of reducing the number of preemptions and migrations imposed by the scheduling algorithm to the jobs released by the tasks. Indeed, as explained in Chapter 2, these preemptions and migrations cause overheads that must be added to the worst-case execution times of the tasks. Consequently, the more preemptions and migrations there are, the less task sets are schedulable. An efficient optimal scheduling algorithm is therefore an optimal algorithm that minimizes the preemptions and migrations overheads incurred by the jobs.

Historically, all the optimal real-time scheduling algorithms for multiprocessor platforms were based on the notion of fairness. Hence, all tasks or groups of tasks were scheduled for a time proportional to their platform utilization. However, it was shown in Chapter 5 that the average numbers of preemptions and migrations incurred by the jobs during the schedule are decreasing when the quality of the fairness is reduced. That is, PFair algorithms cause more preemptions and migrations than BFair or DP-Fair algorithms. Similarly, hierarchical scheduling algorithms that use a combination of DP-Fair and EDF scheduling policies, are more efficient than BFair or DP-Fair algorithms. And still following this same logic, completely unfair scheduling algorithms are those which cause the smallest numbers of preemptions and migrations in average.

Continuous-time	DP-Fair	DP-Fair + Supertasking	Unfair
Periodic	DP-Wrap, LRE-TL, LLREF	EKG, VC-IDT	RUN, **U-EDF**, **EKG-Skip/Swap**
Sporadic	DP-Wrap, LRE-TL	EKG Sporadic, NPS-F, VC-IDT	**U-EDF**
Dynamic	DP-Wrap, LRE-TL	/	**U-EDF**

Table 8.2: Existing optimal algorithms for multiprocessor platforms for continuous-time systems (assuming $\delta \leq m$ and $\delta_i \leq 1, \forall \tau_i \in \tau$). The algorithms developed in this thesis are written in bold and colored in red.

Tables 8.1 and 8.2 show the existing optimal multiprocessor scheduling algorithms for discrete and continuous-time systems, respectively. As we can see in Table 8.1, the few previously existing algorithms for discrete-time systems are either PFair and therefore cause an excessive amount of preemptions and migrations (up to m preemptions and migrations per system tick), or they are boundary fair (BFair) but can only handle periodic tasks with implicit deadlines. Our first attempt in the scheduling of discrete-time systems, has therefore been to design a new BFair scheduling algorithm that is optimal for the scheduling of sporadic tasks. This algorithm named BF^2 was the first new result in many years, that improved the model of schedulable tasks by a discrete-time scheduling algorithm. This first result therefore tended to prove that there was still space for more research in the discrete-time domain.

For continuous-time systems, the existing optimal scheduling algorithms were already more diversified (see Table 8.2). Moreover, their performances in terms of preemptions and migrations are better than those of discrete-time scheduling algorithms. Our first work in the continuous-time world, has consisted in improving the already existing EKG algorithm. We designed a swapping and a skipping algorithm, which both reduce the fairness in the schedule computed by EKG. Using the skipping and swapping algorithms in conjunction with EKG, we showed that the number of preemptions and migrations can be brought back closer to or even outperform the results obtained with RUN — an efficient algorithm for the scheduling of periodic tasks with implicit deadlines in continuous-time environments. Hence, we confirmed our initial thought that less fairness in the schedule also reduces the number of preemptions and migrations. Moreover, unlike RUN, EKG is a semi-partitioned algorithm. That is, only a few tasks migrate while most of them are assigned to only one processor. This technique improves the code locality in memories and has therefore a positive impact on the preemption overheads. When we designed our

skipping and swapping algorithms, we therefore took care to preserve both the optimality and the semi-partitioned scheme of EKG.

The main result of this thesis is certainly U-EDF. Indeed, this algorithm presented in Chapter 6, is the first real-time scheduling algorithm which is optimal for the scheduling of sporadic and dynamic systems without being based on the notion fairness. Instead, it extends the uniprocessor algorithm EDF to the multiprocessor scheduling problem. This choice in the design of U-EDF led to very good performances in terms of preemptions and migrations. Hence, a job incurs only 2 preemptions and migrations in average on a fully utilized platform composed of 16 identical processors (for periodic tasks with implicit deadlines). These results are similar to those obtained with the best today's optimal algorithm — namely RUN —, but unlike RUN, U-EDF is optimal for the scheduling of sporadic and dynamic task sets. Moreover, a modification of U-EDF for the scheduling of dynamic systems in discrete-time environments is also provided. However, even though we strongly believe in its optimality, it was not proven optimal yet.

Note that there is still a limitation to all the scheduling algorithms presented in this work; they are all designed for the scheduling of independent sequential tasks. However, as the number of processors available in a single chip increases, coding techniques evolve from a sequential to a parallel task model so as to take advantage of the high degree of parallelism of new computing platforms. It results that the number of real-time applications using parallel tasks will rapidly grow and the real-time scheduling algorithms designed for the scheduling of sequential tasks could thus become obsolete. In the last chapter of this thesis, we therefore proposed techniques transforming a set of independent parallel tasks in a set of independent sequential tasks schedulable with the today's existing scheduling algorithms. We proved that the resulting set of sequential tasks is schedulable by U-EDF — or any other optimal scheduling algorithm for dynamic systems composed of sequential tasks in a continuous-time environment — on a platform running twice as fast as a platform on which an optimal scheduling algorithm for parallel tasks is utilized. However, there is not any optimal scheduling algorithm for parallel tasks yet. Therefore, scheduling algorithms for sequential tasks are currently our best option.

In conclusion, we solved many open problems in the field of the multiprocessor real-time scheduling. First, we proved that it is possible to design an optimal BFair algorithm for the scheduling of sporadic tasks in discrete-time environments. The newly proposed scheduling algorithm, named BF^2, is currently the most efficient solution in terms of preemptions and migrations for discrete-time systems that is proven optimal. However, it must still be implemented in a real operating system to show that it indeed reduces the

total overhead imposed to tasks. Then, we showed that reducing the fairness in the schedule has a positive impact on the number of preemptions and migrations. This was shown concurrently to the improvement of a semi-partitioned optimal scheduling algorithm for periodic tasks with implicit deadlines (named EKG). Hence, we showed that the amount of preemptions and migrations caused by EKG can be brought back at the same level than other efficient optimal scheduling algorithms for periodic tasks. However, as it follows a semi-partitioned scheme, the preemption overheads are smaller than with global scheduling algorithms. This approach should still be extended to the sporadic version of EKG. With our third contribution, we proved that the fairness is not needed for the optimal scheduling of sporadic and dynamic systems. We designed U-EDF which is the first algorithm that is not based on the notion of fairness to reach the optimality for the scheduling of sporadic and dynamic systems. As we could expect, U-EDF is efficient in terms of preemptions and migrations as it is completely unfair in its schedule. Furthermore, U-EDF is a direct generalization of the uniprocessor algorithm EDF to the multiprocessor scheduling problem. A discrete-time version of U-EDF has also been exposed but must still be proven optimal. Finally, we showed that the previously cited algorithms initially designed for the scheduling of independent sequential tasks, can also be utilized to schedule parallel multi-threaded tasks. The next step would consist in applying the same approach to the scheduling of inter-dependent (parallel) tasks.

"Daddy's flown across the ocean
Leaving just a memory
A snapshot in the family album
Daddy what else did you leave for me?
Daddy, what'd you leave behind for me?
All in all it was just a brick in the wall
All in all it was all just bricks in the wall"

Pink Floyd

Bibliography

J. H. Anderson and A. Srinivasan. A new look at pfair priorities. Technical Report TR00-023, Departement of Computer Science, University of North Carolina, September 1999.

J. H. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000)*, pages 35–43, Stockholm, Sweden, June 2000a. IEEE Computer Society. doi: 10.1109/EMRTS.2000.853990.

J. H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *Proceedings of the 7th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2000)*, pages 297–306, Cheju Island, South Korea, December 2000b. IEEE Computer Society. ISBN 0-7695-0930-4. doi: 10.1109/RTCSA.2000.896405.

J. H. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*, pages 76–85, Delft, The Netherlands, June 2001. IEEE Computer Society.

J. H. Anderson, P. Holman, and A. Srinivasan. Fair scheduling of real-time tasks on multiprocessors. In J. Y.-T. Leung, editor, *Handbook of Scheduling*, chapter 31. Chapman & Hall/CRC, 2005.

B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS 2008)*, pages 243–252, Prague, Czech Republic, July 2008. IEEE Computer Society. ISBN 978-0-7695-3298-1. doi: 10.1109/ECRTS.2008.9.

B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*, pages 322–334, Sydney, Australia, August 2006. IEEE Computer Society. doi: 10.1109/RTCSA.2006.45.

Argonne National Laboratory. The message passing interface (mpi) standard, 2012. URL `http://www.mcs.anl.gov/research/projects/mpi/`.

N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995. ISSN 0922-6443.

J.-L. Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2010. ISBN 978-0-521-76992-1.

T. P. Baker. An analysis of edf schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8):760–768, August 2005. ISSN 1045-9219. doi: 10.1109/TPDS.2005.88.

T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In I. Lee, J. Y.-T. Leung, and S. H. Son, editors, *Handbook of Realtime and Embedded Systems*, chapter 3. Chapman & Hall/CRC, 2008.

M. Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, June 1997.

M. Barr. *Embedded Systems Glossary*. Barr Group, 2007. URL `http://www.barrgroup.com/Embedded-Systems/Glossary`.

S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, January 2003. ISSN 0922-6443. doi: 10.1023/A:1021711220939.

S. K. Baruah. The non-cyclic recurring real-time task model. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS '10)*, pages 173–182, San Diego, California, USA, December 2010. IEEE Computer Society. ISBN 978-0-7695-4298-0. doi: 10.1109/RTSS.2010.19.

S. K. Baruah and J. Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *Journal of Embedded Computing*, 1(2):169–178, April 2005. ISSN 1740-4460.

S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC 1993)*, pages 345–354, San Diego, California, USA, May 1993. ACM. ISBN 0-89791-591-7. doi: 10.1145/167088.167194.

S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proccedings of the 9th International Parallel Processing Symposium (IPPS '95)*, pages 280–288, Santa Barbara, California, USA, April 1995. IEEE Computer Society. ISBN 0-8186-7074-6.

S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

S. K. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, July 1999. ISSN 0922-6443. doi: 10.1023/A:1008030427220.

A. Bastoni, B. B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In S. M. Petters and P. Zijlstra, editors, *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, pages 33–44, July 2010a.

A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the 31th IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 14–24, San Diego, California, USA, December 2010b. IEEE Computer Society.

A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is semi-partitioned scheduling practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, pages 125–135, Porto, Portugal, July 2011. IEEE Computer Society.

G. Bernat, A. Burns, and A. Llamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001. ISSN 0018-9340. doi: 10.1109/12.919277.

V. Berten, S. Collette, and J. Goossens. Feasibility test for multi-phase parallel real-time jobs. In D. Zhu, editor, *Proceedings of the Work-in-Progress session of the IEEE Real-Time Systems Symposium 2009*, pages 33–36, Washington, D.C., USA, December 2009.

V. Berten, P. Courbin, and J. Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *Proceedings of the 5th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2011)*, pages 9–12, Nantes, France, September 2011.

K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS '09)*, pages 447–456, Washington, DC, USA, December 2009. ISBN 978-0-7695-3875-4. doi: 10.1109/RTSS.2009.16.

K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Systems*, 47:319–355, 2011. ISSN 0922-6443.

B. B. Brandenburg. *Scheduling And Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, USA, 2011. URL `http://www.cs.unc.edu/~bbb/diss/brandenburg-diss.pdf`.

B. B. Brandenburg and J. H. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 214–224, Washington, DC, USA, December 2009. IEEE Computer Society.

A. Burns, R. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D scheme. In S. Baruah and Y. Sorel, editors, *Proceedings of the 18th International Conference on Real-Time and Network Systems (RTNS 2010)*, pages 169–178, Toulouse, France, November 2010.

A. Burns, R. I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. *Real-Time Systems*, 48(1):3–33, 2011. ISSN 0922-6443.

G. C. Buttazzo. Real-time scheduling and ressource management. In I. Lee, J. Y.-T. Leung, and S. H. Son, editors, *Handbook of Realtime and Embedded Systems*, chapter 2. Chapman & Hall/CRC, 2008.

G. C. Buttazzo and M. Caccamo. Minimizing aperiodic response times in a firm real-time environment. *IEEE Transactions on Software Engineering*, 25(1):22–32, January 1999. ISSN 0098-5589. doi: 10.1109/32.748916.

M. Caccamo and G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 223–231, Hiroshima, Japan, October 1998. IEEE Computer Society. ISBN 0-8186-9209-X.

J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS 2006)*, pages 111–126, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society. ISBN 0-7695-2761-2. doi: 10.1109/RTSS.2006.27.

J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In J. Y. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 30, pages 30–1 – 30–19. Chapman and Hall/CRC, 2004.

R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, academic press edition, 2001.

Y.-H. Chao, S.-S. Lin, and K.-J. Lin. Schedulability issues for EDZL scheduling on real-time multiprocessor systems. *Information Processing Letters*, 107(5):158–164, August 2008. ISSN 0020-0190. doi: 10.1016/j.ipl.2008.02.014.

J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proceedings of the 13th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pages 28–38, Daegu, Korea, 2007. IEEE Computer Society. ISBN 0-7695-2975-5. doi: 10.1109/RTCSA.2007.37.

H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, pages 101–110, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society. ISBN 0-7695-2761-2. doi: 10.1109/RTSS.2006.10.

H. Cho, B. Ravindran, and E. D. Jensen. Synchronization for an optimal real-time scheduling algorithm on multiprocessors. In *International Symposium on Industrial Embedded Systems 2007 (SIES 2007)*, pages 9–16, Lisbon, Portugal, 2007. IEEE Computer Society. ISBN 1-4244-0840-7.

M. Cirinei and T. P. Baker. EDZL scheduling analysis. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS '07)*, pages 9–18, Pisa, Italy, 2007. IEEE Computer Society. ISBN 0-7695-2914-3. doi: 10.1109/ECRTS.2007.14.

S. Collette, L. Cucu, and J. Goossens. Algorithm and complexity for the global scheduling of sporadic tasks on multiprocessors with work-limited parallelism. In I. Puaut, editor, *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07)*, pages 123–128, Nancy, France, March 2007.

S. Collette, L. Cucu, and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, May 2008.

R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, Octobre 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978814.

R. I. Davis and S. Kato. FPSL, FPCL and FPZL schedulability analysis. *Real-Time Systems*, 48:750–788, 2012. ISSN 0922-6443. doi: 10.1007/s11241-012-9149-x.

M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989. ISSN 0098-5589. doi: 10.1109/32.58762.

U. C. Devi and J. H. Anderson. Improved conditions for bounded tardiness under EPDF pfair multiprocessor scheduling. *Journal of Computer and System Sciences*, 75(7):388–420, November 2009. ISSN 0022-0000. doi: 10.1016/j.jcss.2009.03.003.

R. Devillers and J. Goossens. Liu and layland's schedulability test revisited. *Information Processing Letters*, 73(5-6):157–161, 2000. ISSN 0020-0190. doi: 10.1016/S0020-0190(00)00016-8.

S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1): 127–140, February 1978.

F. Dorin, P. M. Yomsi, J. Goossens, and P. Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. In *Proceedings of the 18th International Conference on Real-Time and Network Systems (RTNS 2010)*, pages 207–216, Toulouse, France, November 2010.

M. Drozdowski. Scheduling multiprocessor tasks – an overview. *European Journal of Operational Research*, 94(2):215–230, October 1996.

A. Easwaran, I. Shin, and I. Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, Septembre 2009. ISSN 0922-6443. doi: 10.1007/s11241-009-9073-x.

J. Edler, A. Gottlieb, and J. Lipkis. Considerations for massively parallel unix systems on the nyu ultracomputer and ibm rp3. *Ultracomputer Note*, 91:1–28, December 1985.

P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proccedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, Brussels, Belgium, July 2010.

F. Fauberteau, S. Midonnet, and M. Qamhieh. Partitioned scheduling of parallel real-time tasks on multiprocessor systems. *SIGBED Rev.*, 8(3):28–31, September 2011. ISSN 1551-3688. doi: 10.1145/2038617.2038623.

D. G. Feitelson. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–77, May 1990. ISSN 0018-9162. doi: 10.1109/2.53356.

D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306 – 318, 1992. ISSN 0743-7315. doi: 10.1016/0743-7315(92)90014-E.

N. Fisher, S. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS '06)*, pages 118–127, Dresden, Germany, July 2006. IEEE Computer Society. ISBN 0-7695-2619-5. doi: 10.1109/ECRTS.2006.30.

N. Fisher, J. Goossens, and S. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, June 2010. ISSN 0922-6443. doi: 10.1007/s11241-010-9092-7.

K. Funaoka, S. Kato, and N. Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, pages 13–22, Prague, Czech Republic, July 2008. IEEE Computer Society.

S. Funk. *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, USA, 2004.

S. Funk. LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Systems*, 46:332–359, 2010. ISSN 0922-6443.

S. Funk and S. K. Baruah. Restricting EDF migration on uniform heterogeneous multiprocessors. *Technique et Science Informatique*, 24/8:917–938, 2005.

S. Funk and V. Nadadur. LRE-TL: An optimal multiprocessor algorithm for sporadic task sets. In M. Chetto and M. Sjödin, editors, *Proceedings of the 17th International Conference on Real-Time and Network Systems (RTNS 2009)*, pages 159–168, Paris, France, October 2009.

S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt. DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, 47:389–429, 2011. ISSN 0922-6443.

Google. Go, 2012. URL `http://code.google.com/p/go`.

J. Goossens and V. Berten. Gang ftp scheduling of periodic and parallel rigid real-time tasks. In S. Baruah and Y. Sorel, editors, *Proceedings of the 18th International Conference on Real-Time and Network Systems (RTNS 2010)*, pages 189–196, Toulouse, France, November 2010.

J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003. ISSN 0922-6443. doi: 10.1023/A:1025120124771.

R. Ha. *Validating timing constraints in multiprocessor and distributed real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign, Computer Science Departement, 1995.

C.-C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. In *Proceedings of 10th IEEE Real-Time Systems Symposium (RTSS '89)*, pages 59–67, Santa Monica, California, USA, December 1989. IEEE Computer Society.

S. Heath. *Embedded System Design (2nd edition)*. Newnes, 2003.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (Fourth Edition)*. Morgan Kaufmann Publishers Inc., 2006. ISBN 0123704901.

P. Holman and J. H. Anderson. Guaranteeing pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 203–212, London, UK, December 2001. IEEE Computer Society. ISBN 0-7695-1420-0. doi: 10.1109/REAL.2001.990612.

P. Holman and J. H. Anderson. The staggered model: Improving the practicality of pfair scheduling. In *Proceedings of the Work-in-Progress Session of the 24th IEEE Real-time Systems Symposium*, pages 125–128, December 2003a.

P. Holman and J. H. Anderson. Using supertasks to improve processor utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 41–50, Porto, Portugal, July 2003b. IEEE Computer Society. ISBN 0-7695-1936-9. doi: http://doi.ieeecomputersociety.org/10.1109/EMRTS.2003.1212726.

K. S. Hong and J. Y.-T. Leung. On-line scheduling of real-time tasks. In *Proceedings of the 9th Real-Time Systems Symposium (RTSS '88)*, pages 244–250, Huntsville, Alabama, USA, December 1988. IEEE Computer Society.

K. S. Hong and J. Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992. ISSN 0018-9340. doi: 10.1109/12.166609.

W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1): 177–185, March 1974.

IEEE. IEEE standard for information technology - standardized application environment profile (aep) - posix realtime and embedded application support. Technical Report Std 1003.13-2003, IEEE Computer Society, 2003.

Intel®. Cilkplus, 2012a. URL `http://software.intel.com/en-us/articles/intel-cilk-plus`.

Intel®. Intel's teraflops research chip: Advancing multi-core technology into the tera-scale era., 2012b. URL `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`.

D. S. Johnson. *Near-optimal Bin Packing Algorithms*. PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, 1973.

D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3): 272–314, 1974.

J. T. Kao and A. P. Chandrakasan. Dual-threshold voltage techniques for low-power digital circuits. *IEEE Journal of Solid-State Circuits*, 35:1009—1018, July 2000.

S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 459–468, Washington, DC, USA, December 2009. IEEE Computer Society.

S. Kato and N. Yamasaki. Global EDF-based scheduling with efficient priority promotion. In *Proceedings of the 14th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*, pages 197–206, Kaohisung, Taiwan, August 2008a. IEEE Computer Society. ISBN 978-0-7695-3349-0. doi: 10.1109/RTCSA.2008.11.

S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software (EMSOFT 2008)*, pages 139–148, Atlanta, GA, USA, October 2008b. IEEE Computer Society. ISBN 978-1-60558-468-3. doi: 10.1145/1450058.1450078.

S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS 2009)*, pages 249–258, Dublin, Ireland, July 2009. IEEE Computer Society. ISBN 978-0-7695-3724-5. doi: 10.1109/ECRTS.2009.22.

A. Khemka and R. K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of Parallel and Distributed Computing*, 43(1):37–45, May 1997.

H. Kim and Y. Cho. A new fair scheduling algorithm for periodic tasks on multiprocessors. *Information Processing Letters*, 111(7):301–309, March 2011.

M. Korsgaard and S. Hendseth. Schedulability analysis of malleable tasks with arbitrary parallel structure. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2011)*, volume 1, pages 3–14, Toyama, Japan, August 2011. IEEE Computer Society.

R. Krten and QNX Software Systems. Getting started with QNX neutrino: A guide for realtime programmers. Technical report, QNX Software Systems International Corporation, June 2012.

K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31th IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 259–268, San Diego, California, USA, December 2010. IEEE Computer Society.

D. Lammers. Intel cancels tejas, moves to dual-core designs. *EETimes*, July 2004.

D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande (JAVA '00)*, pages 36–43, San Francisco, California, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337465.

S. K. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *Proceedings of TENCON'94 - 1994 IEEE Region 10's 9th Annual International Conference on: 'Frontiers of Computer Technology'*, volume 2, pages 607–611. IEEE Computer Society, August 1994.

J. Leung and J. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Perfomance Evaluation*, 2(4):237–250, 1982.

G. Levin, C. Sadowski, I. Pye, and S. Brandt. Sns: A simple model for understanding optimal hard real-time multiprocessor scheduling. Technical Report ucsc-soe-11-09, UCSC, 2009. URL `http://www.soe.ucsc.edu/share/technical-reports/2011/ucsc-soe-11-09.pdf`.

G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-Fair: A simple model for understanding optimal multiprocessor scheduling. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, pages 3–13, Brussels, Belgium, July 2010. IEEE Computer Society.

C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 2(37-60):28–31, November 1969a.

C. L. Liu. lnformation systems: Scheduling algorithms for hard real-time multiprogramming of a single processor. *JPL Space Programs Summary*, 2(37-60):31–37, November 1969b.

C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. ISSN 0004-5411. doi: 10.1145/321738.321743.

D. Liu, X. S. Hu, M. D. Lemmon, and Q. Ling. Firm real-time system scheduling based on a novel qos constraint. *IEEE Transactions on Computers*, 55(3):320–333, March 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.41.

I. Lupu and J. Goossens. Scheduling of hard real-time multi-thread periodic tasks. In A. Burns and L. George, editors, *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS 2011)*, pages 35–44, Nantes, France, September 2011.

Lynux Works. Lynxos user's guide. lynxos release 4.0. Technical Report DOC-0453-02, Lynux Works, 2005.

G. Manimaran, C. S. R. Murthy, and K. Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Systems*, 15(1):39–60, 1998. ISSN 0922-6443.

R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959. ISSN 00251909.

T. Megel, R. Sirdey, and V. David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *Proceedings of the 31th IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 37–46, San Diego, California, USA, December 2010. IEEE Computer Society.

M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 294–303, Phoenix, AZ, USA, December 1999. IEEE Computer Society. ISBN 0-7695-0475-2.

A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachussetts Institute of Technology, 1983. Available as a technical report No. MIT/LCS/TR-297.

A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, Octobre 1997. ISSN 0098-5589. doi: 10.1109/32.637146.

V. Nélis. *Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems*. PhD thesis, Université Libre de Bruxelles, 2010.

V. Nélis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, pages 151–160, Dublin, Ireland, July 2009. IEEE Computer Society.

V. Nélis, B. Andersson, J. Marinho, and S. Peeters. Global-EDF scheduling of multimode real-time systems considering mode independent tasks. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, pages 205–214, Porto, Portugal, July 2011. IEEE Computer Society.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. An optimal multiprocessor scheduling algorithm without fairness. In *Proceedings of the Work-in-Progress Session of the 31th IEEE Real-Time Systems Symposium*, San Diego, California, USA, December 2010. URL `http://cse.unl.edu/~rtss2008/archive/rtss2010/WIP2010/4.pdf`.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2011)*, volume 1, pages 15–24, Toyama, Japan, August 2011a. IEEE Computer Society.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the Work-in-Progress Session of the 32th IEEE Real-Time Systems Symposium*, pages 5–8, Vienna, Austria, December 2011b. IEEE Computer Society.

G. Nelissen, S. Funk, J. Goossens, and D. Milojevic. Swapping to reduce preemptions and migrations in EKG. In E. Bini, editor, *Proceedings of the Work-in-Progress Session of the 23rd Euromicro Conference on Real-Time Systems*, pages 35–38, Porto, Portugal, July 2011c.

G. Nelissen, S. Funk, J. Goossens, and D. Milojevic. Swapping to reduce preemptions and migrations in EKG. *SIGBED Review*, 8(3):36–39, September 2011d. ISSN 1551-3688.

G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 321–330, Pisa, Italy, July 2012a. IEEE Computer Society.

G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic. U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 13–23, Pisa, Italy, July 2012b. IEEE Computer Society.

G. Nelissen, S. Funk, and J. Goossens. Reducing preemptions and migrations in ekg. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012)*, pages 134–143, Seoul, South Korea, August 2012c. IEEE Computer Society.

OpenMP Architecture Review Board. *OpenMP Application Program Interface (version 3.1)*, july 2011. URL `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, Florida, USA, October 1982. IEEE Computer Society.

S. M. Petters and G. Färber. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems*, London, United Kingdom, december 2001.

C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97)*, pages 140–149. ACM, 1997. ISBN 0-89791-888-6. doi: 10.1145/258533.258570.

X. Qi, D. Zhu, and H. Aydin. A study of utilization bound and run-time overhead for cluster scheduling in multiprocessor real-time systems. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010)*, pages 3–12, Macau, SAR, China, August 2010. IEEE Computer Society. ISBN 978-0-7695-4155-6. doi: 10.1109/RTCSA.2010.15.

X. Qi, D. Zhu, and H. Aydin. Cluster scheduling for real-time systems: utilization bounds and run-time overhead. *Real-Time Systems*, 47(3):253–284, May 2011. ISSN 0922-6443. doi: 10.1007/s11241-011-9121-1.

QNX Software Systems Limited. Qnx neutrino realtime operating system library reference. Technical report, QNX Software Systems Limited, 2012.

P. Regnier. *OPTIMAL MULTIPROCESSOR REAL-TIME SCHEDULING VIA REDUCTION TO UNIPROCESSOR*. PhD thesis, Universidade Federal da Bahia, 2012.

P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proceedings of the 32th IEEE Real-Time Systems Symposium (RTSS 2011)*, pages 104–115, Vienna, Austria, December 2011. IEEE Computer Society.

K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91 (2):305–327, February 2003. doi: 10.1109/JPROC.2002.808156.

RTEMS, 2012. URL `http://rtems.com/`.

A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32th IEEE Real-Time Systems Symposium (RTSS 2011)*, pages 217–226, Vienna, Austria, November 2011. IEEE Computer Society.

A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, pages 1–32, October 2012. ISSN 0922-6443. URL `http://dx.doi.org/10.1007/s11241-012-9166-9`. (first online).

L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989. ISSN 0922-6443. doi: 10.1007/BF00365439.

L. Sha, T. Abdelzaher, K.-E. årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004. ISSN 0922-6443.

I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, pages 181–190, Prague, Czech Republic, July 2008. IEEE Computer Society. ISBN 978-0-7695-3298-1. doi: 10.1109/ECRTS.2008.28.

P. B. Sousa, B. Andersson, and E. Tovar. Implementing slot-based task-splitting multiprocessor scheduling. In *Proceedings of 6th IEEE International Symposium on Industrial Embedded Systems (SIES 2011)*, pages 256—265, Vasteras, Sweden, March 2011a. IEEE Computer Society. ISBN 978-1-61284-818-1.

P. B. Sousa, K. Bletsas, B. Andersson, and E. Tovar. Practical aspects of slot-based task-splitting dispatching in its schedulability analysis. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2011)*, volume 1, pages 224–230, Toyama, Japan, August 2011b. IEEE Computer Society.

P. B. Sousa, K. Bletsas, E. Tovar, and B. Andersson. On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems. In *13th Real-Time Linux Workshop*, pages 207–218, Prague, Czech Republic, October 2011c.

A. Srinivasan and J. H. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, pages 189–198, Montréal, Quebec, Canada, May 2002. ACM. ISBN 1-58113-495-9. doi: 10.1145/509907.509938.

A. Srinivasan and J. H. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. *Journal of Embedded Computing*, 1(2):285–302, 2005a. ISSN 1740-4460.

A. Srinivasan and J. H. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 77(1):67–80, Avril 2005b. ISSN 0164-1212. doi: 10.1016/j.jss.2003.12.041.

A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002. ISSN 0020-0190. doi: 10.1016/S0020-0190(02)00231-4.

J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 38–45, Hong Kong, May 1996. IEEE Computer Society. ISBN 0-8186-7398-2.

N. Tchidjo Moyo, E. Nicollet, F. Lafaye, and C. Moy. On schedulability analysis of non-cyclic generalized multiframe tasks. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS '10)*, pages 271–278, Brussels, Belgium, July 2010. IEEE Computer Society. ISBN 978-0-7695-4111-2. doi: 10.1109/ECRTS.2010.24.

Tilera. Tile-gx 3000 series overview, 2011. URL `http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx3000SeriesBrief.pdf`.

K. Tindell, A. Burns, and A. Wellings. Mode changes in priority pre-emptively scheduled systems. In *Proceedings of the 13th Real-Time Systems Symposium (RTSS '92)*, pages 100–109, Phoenix, Arizona, USA, December 1992. IEEE Computer Society.

S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

Wind River Systems, Inc. VxWorks: Application programmer's guide 6.9, edition 2. Technical report, Wind River Systems, Inc., May 2011.

S. J. Wright. *Primal-dual interior-point methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. ISBN 0-89871-382-X.

K.-S. Yeo and K. Roy. *Low Voltage, Low Power VLSI Subsystems*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2005.

V. Yodaiken and M. Barabanov. Real-time linux, application and design. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, January 1997.

J. Zahorjan, E. Lazowska, and D. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2): 180–198, April 1991.

D. Zhu, D. Mossé, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS 2003)*, pages 142–151, Cancun, Mexico, December 2003. IEEE Computer Society. ISBN 0-7695-2044-8.

D. Zhu, X. Qi, D. Mossé, and R. Melhem. An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, 71(10): 1411–1425, 2011. ISSN 0743-7315. doi: 10.1016/j.jpdc.2011.06.003.

CHAPTER

A

Proof of Optimality for PD^{2*}

In this appendix, we present the updated proof of the only lemma of the proof of optimality of PD^2 for the scheduling of sporadic tasks (Lemma 11 in [Srinivasan and Anderson 2002]), which is impacted by the new definition of the group deadline proposed in Section 4.6.1. For the sake of clarity, every modification to the initial proof presented in [Srinivasan and Anderson 2002] is colored in red in the current document. Note that Lemma 11 of [Srinivasan and Anderson 2002] is identical to Lemma 12 introduced in [Srinivasan and Anderson 2005b] which proves the optimality of PD^2 for the scheduling of dynamic task sets under some constraints. Hence, PD^{2*} is also optimal for the scheduling of dynamic task sets under the same constraints (the other lemmas are not impacted in [Srinivasan and Anderson 2005b]).

Remember that in [Srinivasan and Anderson 2002], the proof of optimality of PD^2 is made by contradiction. Hence, they assume that there exists a task set τ respecting some properties noted (T1), (T2) and (T3) such that a deadline is missed during the schedule S produced with PD^2. The first time-instant after the beginning of the schedule corresponding to a deadline miss is denoted by t_d.

Note that in the following proof, we use the notion of displacement defined in [Srinivasan and Anderson 2002]. A displacement is denoted by $\Delta_i = \left\langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \right\rangle$ and means that the subtask $X^{(i)}$ which was initially scheduled at time t_i is now replaced by the subtask $X^{(i+1)}$ which was scheduled at time t_{i+1}. A displacement Δ_i is therefore valid if and only if $X^{(i+1)}$ is eligible to be scheduled at time t_i (i.e., $e\left(X^{(i+1)}\right) \leq t_i$).

Lemma 11

Let $\tau_{i,j}$ be a subtask of a light task scheduled at $t' < pd\left(\tau_{i,j}\right)$ in S. If the eligibility time of the successor of $\tau_{i,j}$ is at least $pd\left(\tau_{i,j}\right)+1$, then there cannot be holes in both $pd\left(\tau_{i,j}\right)$ and $pd\left(\tau_{i,j}\right)+1$.

Proof:

Note that by Lemma 1 (in [Srinivasan and Anderson 2002]), $pd\left(\tau_{i,j}\right) \leq t_d$. Therefore, $t' < t_d$. Let $pd\left(\tau_{i,j}\right) = t$. If $t = t_d$, then t satisfies the stated requirements

because there is no holes in slot t_d (by Lemma 3 in [Srinivasan and Anderson 2002]). In the rest of the proof we assume that $t < t_d$, and hence $t+1 \leq t_d$. Suppose that there are holes in both t and $t+1$. Because there is a hole in slot t and (from the statement of the lemma) the eligibility time of the successor of $\tau_{i,j}$ is at least $t+1$, by Lemma 10 (in [Srinivasan and Anderson 2002]), either $pd\left(\tau_{i,j}\right) < t$ or $pd\left(\tau_{i,j}\right) = t \ \wedge \ b\left(\tau_{i,j}\right) = 1$. Because $pd\left(\tau_{i,j}\right) = t$, the latter in fact must hold, i.e., $pd\left(\tau_{i,j}\right) = t \ \wedge \ b\left(\tau_{i,j}\right) = 1$. We now show that $\tau_{i,j}$ can be removed without causing the missed deadline to be met, contradicting (T2) from [Srinivasan and Anderson 2002]. In particular, we show that the sequence of left-shifts caused by removing $\tau_{i,j}$ does not extend beyond slot $t+1$.

Let the chain of displacements caused by removing $\tau_{i,j}$ be Δ_1, Δ_2, ..., Δ_k, where $\Delta_i = \left\langle X^{(i)}, t_i, X^{(i+1)}, t_{i+1} \right\rangle$, $X^{(1)} = \tau_{i,j}$ and $t_1 = t'$. By Lemma 8 (in [Srinivasan and Anderson 2002]), we have $t_{i+1} > t_i$ for all $i \in [1,k]$. Also, the priority of $X^{(i)}$ is greater than the priority of $X^{(i+1)}$ at t_i, because $X^{(i)}$ was chosen over $X^{(i+1)}$ in S. Because $pd\left(\tau_{i,j}\right) = t \ \wedge \ b\left(\tau_{i,j}\right) = 1$, this implies the following property (from Rules 4.4 presented in Section 4.6.1):

(P) For all $i \in [1,k+1]$, **(i)** $pd\left(X^{(i)}\right) > t$ or **(ii)** $pd\left(X^{(i)}\right) = t$ and $b\left(X^{(i)}\right) = 0$ or **(iii)** $pd\left(X^{(i)}\right) = t$ and $GD^*\left(X^{(i)}\right) \leq GD^*\left(\tau_{i,j}\right)$.

Suppose this chain of displacements extends beyond $t+1$, i.e., $t_{k+1} > t+1$. Let h be the smallest $i \in [1,k+1]$ such that $t_i > t+1$. Then, $t_{h-1} \leq t+1$.

If $t_{h-1} < t+1$, then $X^{(h)}$ is eligible to be scheduled in slot $t+1$ because $e\left(X^{(h)}\right) \leq t_{h-1}$ (by the validity of displacement Δ_{h-1}). Because there is a hole in slot $t+1$ in S, $X^{(h)}$ should have been scheduled there in S (which is a contradiction with the assumption that $t_h > t+1$). Therefore, $t_{h-1} = t+1$ and by Lemma 9 (in [Srinivasan and Anderson 2002]), $X^{(h)}$ must be a successor of $X^{(h-1)}$. By similar reasoning, because there is a hole in slot t, $t_{h-2} = t$ and $X^{(h-1)}$ must be the successor of $X^{(h-2)}$ (see Figure 7(b) in [Srinivasan and Anderson 2002]).

By (P), either $pd\left(X^{(h-2)}\right) > t$ or $pd\left(X^{(h-2)}\right) = t \ \wedge \ b\left(X^{(h-2)}\right) = 0$ or $pd\left(X^{(h-2)}\right) = t \ \wedge \ GD^*\left(X^{(h-2)}\right) \leq GD^*\left(\tau_{i,j}\right)$. In either case, $pd\left(X^{(h-1)}\right) > t+1$ or $pd\left(X^{(h-1)}\right) = t+1 \ \wedge \ b\left(X^{(h-1)}\right) = 0$. To see this, note that if $pd\left(X^{(h-2)}\right) > t$, then because $X^{(h-1)}$ is the successor of $X^{(h-2)}$, by (6) in [Srinivasan and Ander-

son 2002] and (8) (in [Srinivasan and Anderson 2002]), $pd\left(X^{(h-1)}\right) > t+1$. If $GD^*\left(X^{(h-2)}\right) = t$ and $b\left(X^{(h-2)}\right) = 0$, then using (6) (in [Srinivasan and Anderson 2002]) and (8) (in [Srinivasan and Anderson 2002]), it holds that $pd\left(X^{(h-1)}\right) > t+1$. Moreover, because $b\left(\tau_{i,j}\right) = 1$, Definition 4.14 yields $GD^*\left(\tau_{i,j}\right) = pd\left(\tau_{i,j}\right) + 1 = t+1$. Hence, if $pd\left(X^{(h-2)}\right) = t$ then $GD^*\left(X^{(h-2)}\right) \leq t+1$. By Definition 4.14, $GD^*\left(X^{(h-2)}\right) \geq pd\left(X^{(h-2)}\right) = t$ implying that $GD^*\left(X^{(h-2)}\right) = t$ or $GD^*\left(X^{(h-2)}\right) = t+1$. If $GD^*\left(X^{(h-2)}\right) = t$ then Definition 4.14 imposes that $b\left(X^{(h-2)}\right) = 0$. Therefore, using (6) (in [Srinivasan and Anderson 2002]) and (8) (in [Srinivasan and Anderson 2002]), it holds that $pd\left(X^{(h-1)}\right) > t+1$. On the other hand, if $GD^*\left(X^{(h-2)}\right) = t+1$ then, according to Definition 4.14, either $GD^*\left(X^{(h-2)}\right) = pd\left(X^{(h-2)}\right) + 1 \quad \wedge\ pd\left(X^{(h-1)}\right) - pd\left(X^{(h-2)}\right) \geq 2$, or $GD^*\left(X^{(h-2)}\right) = pd\left(X^{(h-1)}\right) \wedge b\left(X^{(h-1)}\right) = 0$. Hence, in all cases, $pd\left(X^{(h-1)}\right) > t+1$ or $pd\left(X^{(h-1)}\right) = t+1 \wedge b\left(X^{(h-1)}\right) = 0$.

Now, because $X^{(h-1)}$ is scheduled at $t+1$, by Corollary 1 (in [Srinivasan and Anderson 2002]), the successor of $X^{(h-1)}$ is not eligible before $t+2$, i.e., $e\left(X^{(h)}\right) \geq t+2$. This implies that the displacement Δ_{h-1} is not valid. Thus, the chain of displacements cannot extend beyond $t+1$ and because $t+1 \leq t_d$, removing $\tau_{i,j}$ cannot cause a missed deadline at t_d to be met. This contradicts (T2) (in [Srinivasan and Anderson 2002]). Therefore, there cannot be holes in both t and $t+1$. ■

CHAPTER

B Missing Proofs in Chapter 6

Lemma 6.1

Let τ_i be any task in τ. It holds that

$$\sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t) = \left\lceil \sum_{\tau_x \in \mathrm{hp}_i(t)} U_x \right\rceil_{j-1}^{j} - (j-1)$$

Proof:

Using Definition 6.5, we have that

$$\sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t) = \sum_{\tau_x \in \mathrm{hp}_i(t)} \left(\left\lceil \sum_{\tau_y \in \mathrm{hp}_x(t) \cup \tau_x} U_y \right\rceil_{j-1}^{j} - \left\lceil \sum_{\tau_y \in \mathrm{hp}_y(t)} U_y \right\rceil_{j-1}^{j} \right)$$

Let us assume that the tasks are indexed in a decreasing priority order. That is, for any pair of tasks τ_k and τ_ℓ in τ such that $k < \ell$ then $\tau_k \in \mathrm{hp}_\ell(t)$ and $\tau_\ell \in \mathrm{lp}_k(t)$. Then, the last expression can be rewritten as

$$\begin{aligned}
\sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t) &= \sum_{x=1}^{i-1} \left(\left\lceil \sum_{y=1}^{x} U_y \right\rceil_{j-1}^{j} - \left\lceil \sum_{y=1}^{x-1} U_y \right\rceil_{j-1}^{j} \right) \\
&= \left\lceil \sum_{y=1}^{1} U_y \right\rceil_{j-1}^{j} - \left\lceil \sum_{y=1}^{0} U_y \right\rceil_{j-1}^{j} + \left\lceil \sum_{y=1}^{2} U_y \right\rceil_{j-1}^{j} - \left\lceil \sum_{y=1}^{1} U_y \right\rceil_{j-1}^{j} \\
&\quad + \cdots + \left\lceil \sum_{y=1}^{i-2} U_y \right\rceil_{j-1}^{j} - \left\lceil \sum_{y=1}^{i-3} U_y \right\rceil_{j-1}^{j} + \left\lceil \sum_{y=1}^{i-1} U_y \right\rceil_{j-1}^{j} - \left\lceil \sum_{y=1}^{i-2} U_y \right\rceil_{j-1}^{j}
\end{aligned}$$

It is easy to see that all these terms cancel each others out at the exceptions of

$\left\lfloor \sum_{y=1}^{0} U_y \right\rfloor_{j-1}^{j}$ and $\left\lfloor \sum_{y=1}^{i-1} U_y \right\rfloor_{j-1}^{j}$. That is,

$$\begin{aligned}
\sum_{\tau_x \in \mathrm{hp}_i(t)} u_{x,j}(t) &= \left\lfloor \sum_{x=1}^{i-1} U_x \right\rfloor_{j-1}^{j} - \left\lfloor \sum_{x=1}^{0} U_x \right\rfloor_{j-1}^{j} \\
&= \left\lfloor \sum_{x=1}^{i-1} U_x \right\rfloor_{j-1}^{j} - \Big\lfloor 0 \Big\rfloor_{j-1}^{j} \\
&= \left\lfloor \sum_{x=1}^{i-1} U_x \right\rfloor_{j-1}^{j} - (j-1) \\
&= \left\lfloor \sum_{\tau_x \in \mathrm{hp}_i(t)} U_x \right\rfloor_{j-1}^{j} - (j-1)
\end{aligned}$$

which proves the lemma. ■

Lemma 6.2

Let x be any real number and let s be a natural number greater than 0. It holds that

$$\sum_{k=1}^{s} \Big([x]_{k-1}^{k} - (k-1) \Big) = [x]_0^s$$

Furthermore, if $0 \leq x \leq s$ then

$$[x]_0^s = x$$

Proof:

Because $[x]_{k-1}^{k} \stackrel{\text{def}}{=} \max(k-1, \min(k,x))$, the term $[x]_{k-1}^{k} - (k-1)$ represents the portion of x which is greater than $k-1$ but smaller than k. Hence, when we sum all the portions of x lying between 0 and s (i.e. $\sum_{k=1}^{s} \Big([x]_{k-1}^{k} - (k-1) \Big)$), we get the portion of x which is greater than 0 but smaller than s (i.e., $[x]_0^s - 0$). Therefore,

$$\sum_{k=1}^{s} \Big([x]_{k-1}^{k} - (k-1) \Big) = [x]_0^s$$

Furthermore, if $0 \leq x \leq s$, then the portion of x lying between 0 and s is x. Hence, $[x]_0^s = x$ when $0 \leq x \leq s$. ■

Lemma 6.4

Let t and t' be two instants such that $t < t'$ and let τ_i be an active task at time t such that $d_i(t) > t'$. Then, there is

$$\text{res\_dyn}_j(t, d_i(t)) = \text{res\_dyn}_j(t, t') + \text{res\_dyn}_j(t', d_i(t))$$

Proof:

From Definition 6.10, we have that

$$\text{res\_dyn}_j(t, d_i(t)) \stackrel{\text{def}}{=} \sum_{x=1}^{i} \text{res\_dyn}_j(d_{x-1}(t), d_x(t))$$

Let t' be any instant between t and $d_i(t)$ such that $d_q(t) \leq t' < d_{q+1}(t)$ (with $q < i$). Applying Definition 6.10, we have

$$\begin{aligned}
\text{res_dyn}_j(t, d_i(t)) &= \sum_{x=1}^{q} \text{res_dyn}_j(d_{x-1}(t), d_x(t)) + \text{res_dyn}_j(d_q(t), d_{q+1}(t)) \\
&\quad + \sum_{k=q+2}^{i} \text{res_dyn}_j(d_{k-1}(t), d_k(t)) \\
&= \sum_{x=1}^{q} \text{res_dyn}_j(d_{x-1}(t), d_x(t)) \\
&\quad + \left\{ \left[\text{w}_{q+1}^{\text{dyn}}(t) \right]_{j-1}^{j} - (j-1) \right\} \times (d_{q+1}(t) - d_q(t)) \\
&\quad + \sum_{k=q+2}^{i} \text{res_dyn}_j(d_{k-1}(t), d_k(t)) \\
&= \sum_{x=1}^{q} \text{res_dyn}_j(d_{x-1}(t), d_x(t)) \\
&\quad + \left\{ \left[\text{w}_{q+1}^{\text{dyn}}(t) \right]_{j-1}^{j} - (j-1) \right\} \times (t' - d_q(t)) \\
&\quad + \left\{ \left[\text{w}_{q+1}^{\text{dyn}}(t) \right]_{j-1}^{j} - (j-1) \right\} \times (d_{q+1}(t) - t') \\
&\quad + \sum_{k=q+2}^{i} \text{res_dyn}_j(d_{k-1}(t), d_k(t)) \\
&= \text{res_dyn}_j(t, t') + \text{res_dyn}_j(t', d_i(t))
\end{aligned}$$

This proves the lemma. ■