Faculté des Sciences
Département d'Informatique

Université Libre de Bruxelles
Université d'Europe

Integrating preemption costs in the real-time uniprocessor schedulability analysis

[WORKING COPY[1]]

Thomas Chapeaux

Supervisor
Joël Goossens

Mémoire présenté en vue de l'obtention du diplôme de Master en Sciences informatiques

Année académique 2013 - 2014

[1]The final version of this document is available at the Bibliothèque Sciences et Techniques of the Université Libre de Bruxelles

Acknowledgements

Je tiens d'abord à remercier mon superviseur, Monsieur le Professeur Joël GOOSSENS, pour sa disponibilité tout au long de l'année et pour sa grande rigueur sans laquelle ce travail n'aurait pas la même qualité.

De même, il me faut remercier l'équipe de l'ECE Paris qui m'a accueilli pour mon stage de recherche durant mon Master. C'est sous la direction de Monsieur Laurent GEORGE que sont nés les premières ébauches d'idées qui ont par la suite été travaillées ici. Des remerciements particuliers pour Messieurs Pierre COURBIN et Clément DUHART qui ont su me faire sentir accueilli dans une ville étrangère, ainsi qu'à Paul RODRIGUEZ qui effectuait son stage de recherche en même temps que moi.

En ce qui concerne directement le mémoire, je tiens finalement à remercier les membres de mon jury, Messieurs les Professeurs Gilles GEERAERTS et Bernard FORTZ, pour le temps qu'ils consacreront à la lecture de ce document et pour leur expertise lors de la défense.

De manière plus large, ce mémoire est pour moi la conclusion de quatre ans d'étude à l'Université Libre de Bruxelles, d'abord en Bachelier puis en Master en Sciences Informatique. Je tiens donc à remercier tous les professeurs et assistants pour leur enseignement de grande qualité. De même, j'aimerais remercier les différents étudiants avec qui j'ai eu le plaisir de travailler sur des projets ou de réviser des examens, toujours dans une ambiance d'entraide et de bonne humeur. Je leur souhaite à tous une grande réussite dans leur carrière future.

Finalement, je tiens à remercier mes parents Colette et Philippe, ma sœur Céline et ma petite amie Alizé pour leur soutien moral durant mes études et la rédaction de ce mémoire.

Que vous soyez tous et toutes sincèrement remerciés.

Contents

1 Introduction

Real-Time systems are defined as systems with timing constraints. That means that the expected results produced by the system must not only be correct, but also be available before a given deadline[BA06]. A well-known example is the ABS system in a car, which has to be operational within a given time frame.

Real-time systems have a wide range of applications, from virtual reality to life-critical flight control systems. We often distinguish between soft real-time, where a deadline miss degrades the quality of service and should be avoided if possible, with hard real-time, where a deadline miss is equivalent to a system failure and often has disastrous economical or human consequences. In the following we only consider hard real-time systems.

The theory of real-time systems feasibility analysis aims to offer a model allowing system designers to decide if a given system will be able to execute on a given hardware without missing a deadline. For this goal to be achieved, results given by the model must correspond to the actual behaviour of systems in a real hardware architecture. We call this concept **predictability**. If the results given by a model allow us to accurately predict the actual behaviour of systems, we say that the model has a good predictability. Else we say that it has bad predictability.

The first part of this master thesis consists of a literature review on the study of hard real-time systems for uniprocessor scheduling. The objectives and organization of the remainder of the master thesis will be discussed respectively in Section 1.2 and Section 1.3.

1.1 Real-Time Systems

1.1.1 Task system model

We use an extension of Liu and Layland model presented in [Liu00] to formalize real-time systems. This model represents such systems by a set of *tasks* generating *jobs*, where a job is a whole entity of computation with a given time of arrival and a given deadline.

Definition 1 (Task system)**.** A *task system* is a set $\tau = \{\tau_1, \ldots, \tau_n\}$, where each τ_i represents a task (see Definition 2).

Definition 2 (Task)**.** A *task* is defined as the tuple (O_i, T_i, D_i, C_i), where

- O_i is the minimal time at which the first job of the tasks is generated.
- T_i is the minimal time between two job generations.

- D_i is the relative deadline of its job.
- C_i is the execution time of its job.

Definition 3 (Job)**.** A task τ_i generates an infinite number of *jobs*, each denoted as the tuple $J_{i,j} = (a_{i,j}, d_{i,j}, c_{i,j})$ where

- $j = \{1, 2, ...\}$ is an unique identifier for the job within τ_i.
- $a_{i,j} \geq a_{i,j-i} + T_i$ is the arrival time (or *activation time*) of the job, with $a_{i,1} \geq O_i$.
- $d_{i,j} = a_{i,j} + D_i$ is an absolute deadline at which the job must be completed.
- $c_{i,j}$ is the time it takes to complete the job (*execution time*).

Definition 4 (State of a job)**.** We define the following possible states of a job.

- Jobs whose arrival is later than the current time are said to be *future jobs*.
- Jobs currently being executed are said to be *busy jobs*.
- Jobs who have been executed for $c_{i,j}$ units of time are said to be *completed jobs*.
- Jobs who have arrived but are neither completed nor busy are said to be *active jobs*.

We also assume a discrete time, meaning that we consider that time passes one unit (sometimes called *clock tick*) at a time. This means that the values listed above must be integers and that jobs will execute for whole units.

Tasks systems can be subdivided into different types based on their task parameters, some of which are described below.

A tasks system is said to be **periodic** when all inequalities in the definition are replaced by equalities, i.e. all jobs $j_{i,j}$ of the same tasks τ_i arrive precisely at the time $t = O_i + (j-1) \times T_i$. When a tasks system is not periodic, it is said to be **sporadic**. Sporadic tasks are often defined without offset values, which in our model translates to a value $O_i = 0$.

Similarly, we define the following property on the values of the deadline and the period:

- if $T_i = D_i \, \forall i$, we say that the system use **implicit deadlines**.
- if $T_i \leq D_i \, \forall i$, the system use **constrained deadlines**.
- if there are no constraints between those values, the system use **arbitrary deadlines**.

The last property we define is on the offset value. If $O_1 = O_2 = ... = O_n$, the tasks are said to be **synchronous**, otherwise they are said to be **asynchronous**. Note that adding or subtracting a constant value to every offset value in a system does not change the system. For that reason, we conventionally define the smallest offset to be 0 (and in the synchronous case we have $O_i = 0 \, \forall i$).

In the later sections of this master thesis, unless otherwise noted, we consider *periodic*, *constrained deadlines* systems (either *synchronous* or *asynchronous*).

The following notions are also defined:

Definition 5 (Task Utilization)**.** For a given task τ_i, we define its utilization

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

Definition 6 (System Utilization)**.** For a given system, its utilization U_{tot} is given by

$$U_{tot} = \sum_{i=1}^{n} U_i$$

1.1.2 Scheduling model

We know how to define a real-time system as a task set, we now define a mathematical representation of its execution and scheduling.

This master thesis, and thus the results presented in this section, is concerned with the uniprocessor case only, meaning that the whole system is executed on only one computational unit (sometimes referred as *processor* in the following). Thus, at each unit of time, at most one job can be executing. A similar summary of the schedulability results for the multiprocessor case can be found in [GR13].

Definition 7 (Schedule of a system [Goo99])**.** For a given task set, we define a *schedule* of it as a function $\sigma(t) : \mathbb{N} \to \mathbb{N}$ such that $\sigma(t)$ gives us i, the identifier of the task of the job executing at time t (or 0 for times at which no job is executing).

For example, consider the following system.

Task System 1.

	O_i	C_i	D_i	T_i
τ_1	0	1	3	3
τ_2	1	1	2	2

And the schedule of it described by Figure 1.

t	1	2	3	4	5	...
$\sigma(t)$	2	1	2	1	0	...

Figure 1: A schedule of System 1

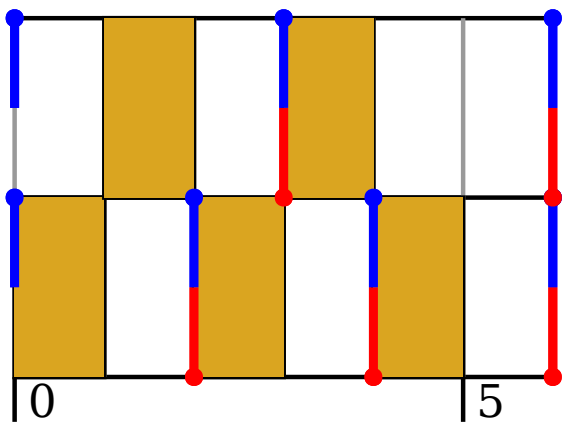

Figure 2: Schedule σ of System 1.

We represent this schedule as shown in Figure 2. In this figure, each row represent a different task (the top row being τ_1, the second row from the top being τ_2, etc.), and each unit of time is represented by a column where the row of the executing task is colourized. Job arrivals are represented on the task row by an arrow (with a circular head) pointing up on the left of the instant column. Job deadlines are represented similarly by an arrow pointing down. Deadline miss (not pictured in Figure 1) are represented by a big circle over the corresponding deadline arrow (as seen in, for example, figure 3).

Note that if $\sigma(t) = i$ and there are multiple jobs of τ_i active at t (in the case of arbitrary deadline systems), we execute the job with the earliest arrival.

Using the previously described model, we can define the following notions:

Definition 8 (Valid schedule)**.** A schedule for a given system is said to be *valid* if every job in the given system completes before its deadline is reached when the system follows this schedule.

Definition 9 (Feasibility)**.** A task set is said to be *feasible* if it exists at least one valid schedule of it.

Definition 10 ("Property"-Feasibility)**.** A task set is said to be *"property"-feasible* (for some property of schedulers), if it exists a scheduler respecting that property which produces a valid schedule of it. Examples of properties (discussed later) are FTP, FJP, and work-conserving.

We can derive a necessary condition of feasibility using the system utilization (Definition 6). Indeed, the utilization of a task is the fraction of time necessary for the processor to complete every job of this task. We thus have the following necessary condition of feasibility.

Theorem 1. *A necessary condition of feasibility of a task system $\tau = (\tau_1 \dots \tau_n)$ in the uniprocessor case is*

$$\sum_{i=1}^{n} \mathrm{U}_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

For example, the following task set is feasible. We can convince ourselves of this by looking at a valid schedule of it (see Fig. 3). Note that it also respects the necessary condition ($\mathrm{U}_1 + \mathrm{U}_2 + \mathrm{U}_3 = \frac{1}{10} + \frac{1}{2} + \frac{2}{5} = 1 \leq 1$).

Task System 2.

	O_i	C_i	D_i	T_i
τ_1	0	1	10	10
τ_2	1	1	2	2
τ_3	0	2	5	5

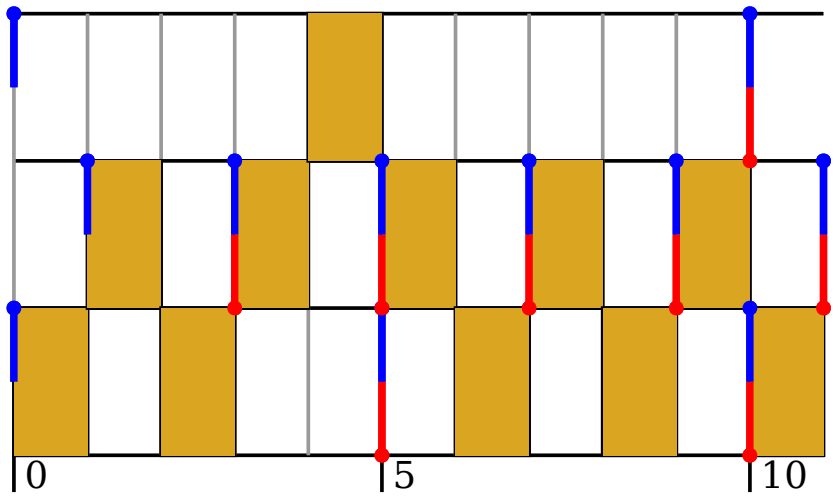

Figure 3: A valid schedule of system 2. After $t = 10$, the schedule is periodic

However, the following task set, constructed by increasing the value of C_1 by 1, is infeasible (Fig. 4 gives an example of a schedule where a deadline is not met). We can convince ourselves that there are no schedule which will respect every deadline by observing that it does not satisfy the necessary condition ($\mathrm{U}_{tot} = \frac{2}{5} + \frac{1}{2} + \frac{2}{10} > 1$).

Task System 3.

	O_i	C_i	D_i	T_i
τ_1	0	**2**	10	10
τ_2	1	1	2	2
τ_3	0	2	5	5

The notion of feasibility only characterizes the existence of a schedule allowing all jobs to be completed, giving no indication on the algorithm that can be used to determine such a schedule. We thus define the following notions:

Definition 11 (Schedulability (by a scheduling algorithm))**.** A task set is said to be *schedulable* by a scheduling algorithm if the algorithm applied to the task set produces a valid schedule.

If a system τ is schedulable by some scheduling algorithm S, we sometimes say that τ is S-schedulable (e.g. EDF-schedulable), or that it is *feasibly* schedulable by S.

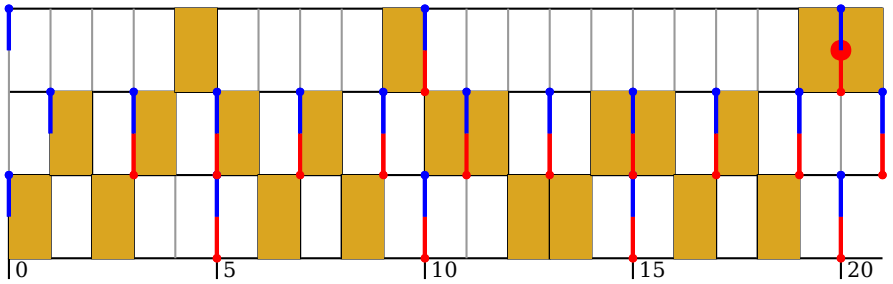

Figure 4: A schedule of system 3 where a deadline is missed at $t = 20$

Definition 12 (Optimal schedulign algorithm)**.** A scheduling algorithm is said to be *optimal* if every feasible task set is schedulable by this algorithm.

Definition 13 ("Property"-Optimal scheduling algorithm)**.** A scheduling algorithm is said to be *"property"-optimal* for some properties of scheduler if it respects that property and if every task set schedulable by at least one scheduler respecting this property is also schedulable by this algorithm.

If it exists an algorithm such that a task set is schedulable by it, the task set is feasible (Definition 9). Therefore, if we know an optimal scheduling algorithm, the schedulability of a system by this algorithm (which can be determined e.g. by simulation over a sufficiently long period of time) is a necessary and sufficient condition of feasibility of the system.

1.1.3 Examples of scheduling algorithms

In this section, we assume that scheduling algorithms are both *work-conserving* and *preemptive*, two notions defined here.

Definition 14 (Work-conserving scheduling algorithm)**.** A scheduling algorithm is said to be *work-conserving* (or *non-idling*) if the schedule produced by the algorithm for a system never contains an idle unit (a unit of time t such that $\sigma(t) = 0$) if there are active jobs in the system at time t.

Definition 15 (Preemptive scheduling algorithm)**.** A scheduling algorithm is said to be *preemptive* if the schedule produced by the algorithm for a system may contain preemptions, i.e. instants where a job that was executing the previous instant is not finished, but is interrupted so that another job may execute.

Scheduling algorithms are given a task system as input and have to produce

a schedule of it, i.e. decide for each instant which of the active jobs will be executing.

Thus, they can always be expressed as a function $\mathrm{prio}(J_{i,j}, t, \ldots)$ (where the "..." express that we do not restrict the information available to a scheduler in the general case) expressing the **priority** of each job at each instant, with the assumption that the job of highest priority at a time t is the one which will execute at that time.

Note that in order to be predictable, algorithms must have a deterministic way of handling cases where two jobs have the exact same probability. Often, when two jobs have the same priority at an instant t, we arbitrarily choose the job whose task has the lowest identifier i.

We distinguish between three classes of schedulers.

- **Fixed Task Priority** (FTP): Each task τ_i receives a constant priority $\mathrm{prio}_{\mathrm{FTP}}(i)$ offline based on its parameters. Jobs priorities are always equal to the priority of their task. More rigorously:

$$\mathrm{prio}_{\mathrm{FTP}}(J_{i,j}, t) = \mathrm{prio}_{\mathrm{FTP}}(i)\ \forall i, j, t$$

- **Fixed Job Priority** (FJP): Each job $J_{i,j}$ receives a constant priority $\mathrm{prio}_{\mathrm{FJP}}(J_{i,j}, a_{i,j})$ upon its arrival based on its parameters. The job keeps this priority until it is completed (or miss a deadline). More rigorously:

$$\mathrm{prio}_{\mathrm{FJP}}(J_{i,j}, t) = \mathrm{prio}_{\mathrm{FJP}}(J_{i,j}, a_{i,j})\ \forall i, j, t$$

- **Unconstrained Dynamic Priority** (DP): The most general case. Priorities of jobs are not constant over time.

Note that $\mathrm{FTP} \in \mathrm{FJP} \in \mathrm{DP}$.

FTP algorithms are the easiest to implement as they are able to set the task priorities offline (for this reason, they are often the only available option). FJP algorithms require to compute the priority of jobs as they are activated, introducing a small overhead which can be easily integrated in the execution time of the job. DP algorithms, however, need to update the priorities of every active jobs at a fixed interval of time (ideally the smallest possible), which can introduce some overhead (assumed negligible in this master thesis).

Here are some examples of well-known scheduling algorithms:

Deadline Monotonic (DM) FTP algorithm. Each task τ_i is assigned a priority:

$$\mathrm{prio}_{DM}(i, t) = \frac{1}{D_i}$$

It can be shown that DM is FTP-*optimal* for synchronous constrained deadline systems.

Earliest Deadline First (EDF) FJP algorithm. Each job $J_{i,j}$ is assigned a priority:

$$\text{prio}_{EDF}(J_{i,j}, t) = \frac{1}{d_{i,j}}$$

The authors of [LL73] (with a flaw corrected in [Goo99]) have shown that EDF is an optimal algorithm for all systems in the uniprocessor case.

Least-Laxity First (LLF): DP algorithm. At each instant, each job $J_{i,j}$ is assigned a priority:

$$\text{prio}_{LLF}(J_{i,j}, t) = \frac{1}{\ell_{i,j}(t)}$$

where $\ell_{i,j}(t) = d_{i,j} - t - (c_{i,j} - \epsilon_{i,j}(t))$ and $\epsilon_{i,j}(t)$ is the number of CPU unit the job has used. LLF is also called the *slack-time algorithm*. [KM83] has shown that LLF is also optimal for all systems in the uniprocessor case.

Concerning LLF, one has to realize that its major problem is that it can induces a great number of preemptions. Consider the following task set and its LLF schedule.

Task System 4.

	O_i	C_i	D_i	T_i
τ_1	0	4	8	10
τ_2	0	5	9	10

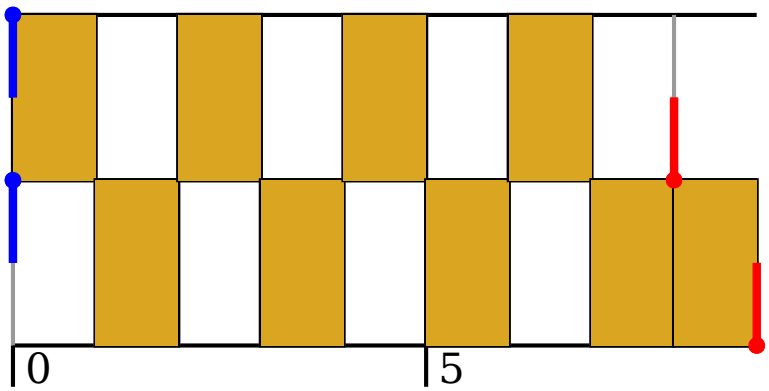

Figure 5: System 4 scheduled by the LLF algorithm. Note the unnecessarily high number of preemptions (at $t = 1, 2, 3, 4, 5, 6$)

As we can see in Fig. 5, the schedule will use 6 preemptions every 10 units of time. In the current model, we consider preemption times to be negligible. In later parts of this master thesis, we concentrate on models where the costs associated with these context switch are taken into account, and the high preemption count associated with LLF proves to be problematic.

1.1.4 Optimal algorithms and feasibility conditions

Optimal algorithms allow us to test the feasibility of a system by simulation: any system feasibly scheduled by an optimal algorithm is feasible, and any system for which an optimal algorithm causes a deadline miss is infeasible. However, it is not always necessary to simulate the execution of some systems to determine feasibility. In some special cases, it is possible to determine feasibility by analysis of the tasks properties.

In this section, we look at necessary and sufficient conditions allowing us to know if a system is feasible, based on schedulability by a known optimal algorithm.

To this aim, we introduce the following notions:

Definition 16 (Response time of a job)**.** The *response time* of job $J_{i,j}$ is noted r_i^j and is the time between the arrival of a job and its completion.

Definition 17 (Critical instant of a task)**.** A *critical instant* of a task τ_i in a system τ is a context of arrival of one of its job $J_{i,j}$ and of of other jobs of other tasks such that r_i^j is maximal.

It then follows that if the response time of the job arriving at the critical instant of τ_i is $\leq D_i \; \forall i$ when scheduled by an optimal algorithm, the system is feasible.

Let us first consider FTP-feasibility of synchronous constrained tasks. The authors of [LL73] have shown that the critical instant of all tasks when scheduled by DM (an FTP-optimal algorithm) occurs at the arrival of their first job ($t = 0$ as we consider synchronous systems). We find in [Aud91] a formula to find the response time of each task in that case (and thus to determine FTP-feasibility), as

$$r_i \leq D_i$$

$$r_i^1 = C_i + \sum_{i=1}^{i-1} \lceil \frac{r_i^1}{T_j} \rceil C_j$$

This result cannot be extended to arbitrary deadline task set. In this case, the response time of the first job is not necessarily the maximum. To analyse the feasibility of such systems, we use the notion of feasibility intervals:

Definition 18 (Feasibility Inverval)**.** The *feasibility interval* of a system is a finite interval of time such that, if no job deadline is missed within it, no other deadline will be missed by the system.

If a feasibility interval is known for a system, it is sufficient to simulate an optimal scheduler on a system during this interval to determine feasibility.

For synchronous arbitrary deadline, [Goo99] has shown that $[0, \lambda_n]$ is a feasibility interval, with λ_n being the first idle point (see Definition 19) after $t = 0$. The value of λ_n is given by the formula

$$\lambda_n = \sum_{i=1}^{n} \lceil \frac{\lambda_n}{T_i} \rceil C_i$$

Definition 19 (Idle point, idle time)**.** A time t_i in a schedule of a system is said to be an *idle point* (or *idle time*) if at time t_i, every job released strictly before t_i is completed.

For asynchronous task set, [LW82] have shown that $[\mathrm{O}_{\max}, \mathrm{O}_{\max} + 2\,\mathrm{H}]$ is a feasibility interval, where $\mathrm{O}_{\max} = \max\{O_i\}$ and $\mathrm{H} = \mathrm{lcm}\{T_i\}$. An optimal FTP algorithm is Audsley's algorithm ([ATB93]).

Now we consider FJP-feasibility. As we saw, we can use EDF, which is optimal (not just FJP-optimal!).

For synchronous implicit deadline, the condition $\mathrm{U}_{\mathrm{tot}} \leq 1$ is actually necessary and sufficient. For arbitrary deadline, we can reuse the feasibility interval previously defined ($[0, \lambda_n]$) and for asynchronous tasks, we can also reuse $[0, \mathrm{O}_{\max} + 2\,\mathrm{H}]$. Smaller intervals (of the same asymptotic complexity) can be used in some cases, one of which will be discussed in Section 3.4.2.

As EDF is optimal (at least for uniprocessor), we do not need to study any DP algorithms.

The previous results on feasibility conditions of real-time systems are summarized in Figure 6.

Model	Sched. Class	Optimal Algo.	Feasibility Interval
Synchr. Constr.	FTP	DM	$[0, \mathrm{H}]$
Asynchr. Constr.	FTP	Audlsey	$[0, \mathrm{O}_{\max} + 2\,\mathrm{H}]$
Synchr. Impl.	General	EDF	Immediate ($\mathrm{U}_{\mathrm{tot}} \leq 1$)
Synchr. Constr.	General	EDF	$[0, \lambda_n]$
Asynchr. Constr.	General	EDF	$[0, \mathrm{O}_{\max} + 2\,\mathrm{H}]$

Figure 6: Summary of the feasibility conditions

1.1.5 Worst-Case Execution Time

The model assumes a constant execution time (C_i) for every job of the same task. In practice, the execution time of a job often depends on its input. Therefore, it may be different for each job of the same task and its exact value is not known when doing an offline analysis (in which we would like to decide the feasibility of a system for any possible job execution time).

To solve this problem, we introduce the notion of sustainability, as defined in [BB06]. We recall this definition here, with some parameter names changed to fit the rest of the text.

Definition 20 (Sustainability [BB06])**.** A schedulability test for a scheduling policy is *sustainable* if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one of more individual job(s) are changed in any, some, or all of the following ways.

- Decreased execution times (C_i)
- Later arrival times ($a_{i,j}$)
- Smaller jitter[2]
- Larger relative dealines (D_i)

Note that in the case of later arrival times ($a_{i,j}$), the distance between the arrival of two successive jobs is still lower-bounded by the period of the task (T_i).

We can further define sustainability with regard to a subset of parameters. For example a schedulability test for a scheduling policy is *sustainable with regard to the execution times* if any system deemed schedulable by the schedulability test remains schedulable when the execution times of one of more individual job(s) are decreased.

In the same paper, the authors of [BB06] show that any exact schedulability tests for EDF in the uniprocessor model is sustainable, the main argument being that any of these changes leads to another feasible system, which is thus schedulable by EDF as it is optimal.

Therefore, if we know the **worst-case execution time (WCET)** of a task, i.e. an upper bound of the execution time of its jobs, we know that any job during the execution will have an execution time smaller or equal to the WCET. If we set the C_i value of each task of a system to its WCET and if this system is deemed schedulable by EDF, the property of sustainability with

[2]Where the jitter is the minimal time between the arrival of a job and the first time unit where it executes. In this master thesis, we assume all tasks have a jitter of 0.

regard to the execution times assures us that it will be schedulable for any possible execution time of the jobs of the system.

Task WCET analysis in itself is a whole branch of study. In [WEE+08], an overview of the existing techniques to determine the WCET of a task, the authors differentiate between *static methods*, which analyse the task code to determine the worst control flow with a very abstract hardware model, and *measurement-based methods*, in which the task is executed on the given hardware or a simulator of it with different inputs.

Whichever technique is used, the task is always supposed to be in isolation in the system when studying its execution time. It thus does not take into account side-effects caused by the execution of other tasks, such as the preemption costs.

1.1.6 Negligibility of preemption cost

As we saw in section 1.1.3, previous results for preemptive schedulers assume that the duration of a preemption, or *preemption cost*, is negligible compared to the duration of a time unit or the execution times of the jobs. In this section, we discuss the validity of that claim in practice.

The authors of [BBY13] distinguish between the following four types of preemption costs.

- *Scheduling cost:* At each preemption, the scheduling algorithm has to handle the suspension of the preempted task, the restoration of the preempting task, the context switch and it has to update its internal data.
- *Pipeline cost:* Time taken to flush and refill the processor pipeline.
- *Cache-related cost:* Time taken to reload the cache lines evicted by the preempting task.
- *Bus-related cost:* Additional cache misses when accessing the RAM.

Note that in the literature, *preemption costs* sometimes refers only to the first two, i.e. the time interval from the moment we decide to preempt the current job until the preemptive job starts executing. The last two delays are then called *interference* (or *task interference*), and represents the additional execution time imposed to a preempted job.

The authors of [BCSM08] have shown that in some cases, cache-related costs alone account to as much as 30% of the WCET, which is *far from negligible*. If we consider that a given job can be preempted multiple times during its execution, we see that preemptions could easily add a consequent

load to the system and make a supposedly optimal scheduler miss a deadline (LLF, for example, is an optimal scheduler known to cause a high number of preemptions).

It follows that studying a model where preemptions costs are not neglected will greatly increase the predictability of our feasibility predictions.

1.2 Objectives of the master thesis

Based on the observations made in Section 1.1.6, we claim that the extended Liu and Layland model as presented previously has bad predictability in situations where the preemption times are not negligible with regard to the execution times of the job.

To this aim, the objectives of this master thesis are multiple. First, we want to review the existing contributions on the integration of the preemption costs. We want to see which techniques are proposed in the scientific community and in practice and see if they can be improved.

Then, we want to propose an efficient algorithm to optimally schedule systems with good predictability in contexts where the preemption costs are not negligible. Ideally, we would want to have an algorithm as light and easy to implement as EDF. If that is not possible, we want to at least propose a good heuristic and/or show that the problem is complex (e.g. NP-hardness).

Finally, we want to validate our results with a simulator. The simulator could also be used to test ideas and hypothesis during our research, and produce figures for the master thesis.

1.3 Organization of the master thesis

In this chapter, we presented a model used to study hard real-time systems. We showed how the model allows us to determine feasibility of those systems under different assumptions, such as negligible preemption times. Then in Section 1.1.6 we discussed the pertinence of this assumption in real applications with regard to the predictability.

The remaining part of this master thesis explore models where this assumption is not made. First, in Section 2, we present approaches where the impact of preemptions is reduced, allowing us to assume that they are negligible while still maintaining good predictability.

In Section 3, we study another model, called the NNP model, where preemptions costs are taken into account during the execution of the schedule. We

then study how previous results on schedulability analysis translate to the NNP model. We also quantify the complexity of scheduling in this model.

In Section 4, we propose two of our own scheduling algorithms adapted to the NNP model.

In Section 5, we present the simulator we used and show the performance of our scheduling algorithms against state-of-the-art algorithms.

Globally, this master thesis makes three significant contributions.

- A list of significant properties of scheduling in the NNP model, in Section 3.3.
- The implementation and description of our simulator for the NNP model, in Section 5.2.
- The description (Section 4) and empirical analysis (Section 5.3) of two heuristic scheduling algorithms for the NNP model.

2 Reducing the impact of preemptions

As we saw, one of the strong assumptions made when studying real-time systems is that preemption times are negligible compared to the execution times of the jobs. This may be problematic (i.e. the model will have bad predictability) in situations with a great number of preemptions (an extreme example being certain cases of LLF scheduling), or in situations in which we know that the preemption times are not negligible. This section reviews alternative models in which preemptions times are assumed to be negligible, but with techniques in place to reduce their cost to a minimum, with the aim of having a better predictability.

2.1 Surestimation of the WCET

What is often done in practice in order to improve the predictability without having to change the model much is to artificially boost the WCETs by a fixed value (often between 20 and 30 %). This added load is supposed to be an upper bound of the preemption costs and can be seen as a margin of error: if the system is schedulable with it, then the preemption costs in the actual system will add less load and, because of the property of sustainability (Definition 20), the system will always be schedulable if we use a sustainable algorithm such as EDF.

Note that this improve the predictability when the model show that a system in feasible. In situations where the model show that a system in infeasible, it is possible that the system would actually be possible (i.e. bad predictability).

A more refined approach for FTP scheduling would be to apply a multiplicative factor to the WCET that is inversely proportional to the priority of the task, as tasks of lower priority will get preempted more often on average. However, this is still a really rough approach, as the number of preemptions is not constant for each job of the same task.

In summary, this approach consider a pessimistic worst-case, which leads to resources waste. Moreover, as we saw in section 1.1.5, the WCET is supposed to be an upper bound of the execution time of the task alone in the system. Thus it does not make sense to integrate the preemption times into it as it is the task of the scheduler to ensure that the deadline are met, and thus to take the preemptions times into account.

While this approach can serve as a quick estimation of feasibility, more rigorous and extensive approaches (as those presented in later sections of this master thesis) are preferred.

2.2 Non-preemptive scheduling

An extreme approach to reduce the impact of preemption on predictability is to absolutely forbid them. This cause less systems to be feasible but allows us to get rid of preemption costs entirely (i.e. we trade feasibility for more predictability).

To illustrate this, consider the following system.

Task System 5.

	O_i	C_i	D_i	T_i
τ_1	1	1	1	3
τ_2	0	2	3	3

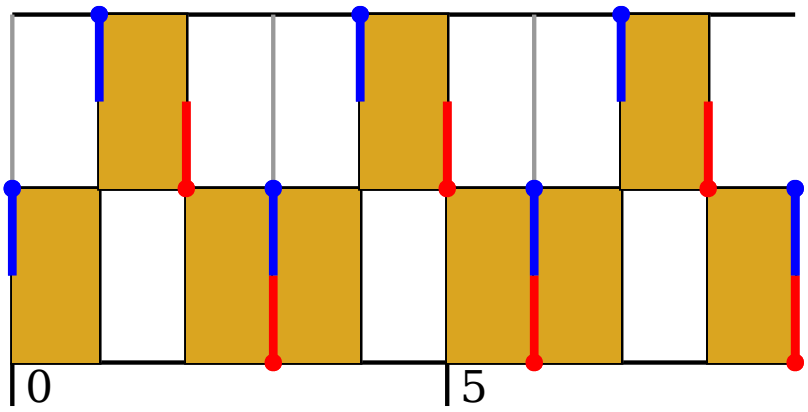

Figure 7: A valid schedule of system 5 which require preemptions (at $t = 1, 3, 7, \ldots$). After $t = 9$, the schedule is periodic.

System 5 is schedulable by a preemptive version of EDF (Figure 7 shows the only valid schedule), and is clearly not schedulable by any non-preemptive schedulers. By definition, we have that non-preemptive schedulers are a subset of preemptive schedulers (as a preemptive scheduler may create non-preemptive schedules), thus this example shows that non-preemptive-feasible $\subset$ preemptive-feasible (strict).

The remaining of this section reviews the feasibility analysis of the non-preemptive approach, based mostly on [JSM91] with some results from [GMR$^+$95]. The feasibility loss of non-preemptive scheduling is also discussed.

2.2.1 Idling vs. Non-idling

In the preemptive case with no preemption times, we assumed the algorithms to be non-idling. This was not a strong assumption as we would not gain anything by allowing the scheduler to not execute any jobs while some were active. Indeed, as we can decide which job is executing at each unit of time without constraining our future choices, there are no reasons to use idle units. Moreover, this assumption allowed us to simplify the analysis and helped to prove some results.

In the non-preemptive case however, the non-idling assumption could prevent us to schedule systems that would otherwise be feasible. Consider the following system.

Task System 6.

	O_i	C_i	D_i	T_i
τ_1	1	1	1	5
τ_2	0	3	5	5

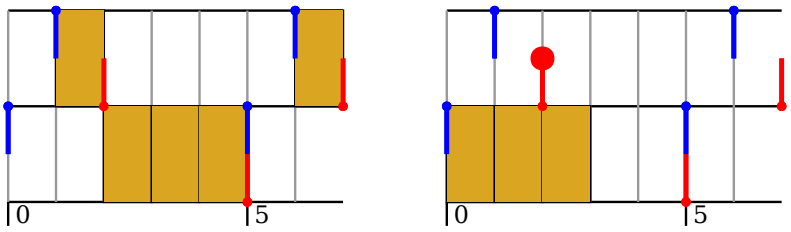

Figure 8: Left: a valid schedule with an idling unit at $t = 0$. Right: NINP-EDF (an optimal non-idling non-preemptive scheduler) missing a deadline at $t = 2$.

As we can see in Figure 8, at $t = 0$, as the system is non-idling, the scheduler has to choose the only active job, $J_{2,1}$ (the first job of τ_2). But because we are in a non-preemptive model, this forces the schedule to complete the execution of this job before being able to choose another. The system is thus locked until $t = 3$, at which point $J_{1,1}$, the first job of τ_1, will have missed its deadline. This means that no non-preemptive non-idling scheduling algorithm can schedule this system. However, an *idling* non-preemptive algorithm could schedule the system by delaying the execution of $J_{2,1}$ until $t = 3$, allowing $J_{1,1}$ to complete on time.

This example shows that in non-preemptive algorithms, idling and non-idling algorithms have to be considered separately.

2.2.2 Non-idling algorithms

When considering non-idling algorithms, [JSM91] shows that NINP-EDF (*Non-Idling Non-Preemptive EDF*) is optimal. Then, it suffices to check if this algorithms feasibly schedule a task set (by simulating it on a given feasibility interval) to know if the task set is schedulable.

For *asynchronous periodic tasks*, the authors of [GMR$^+$95] use the results obtained in the preemptive case to find an upper bound of the feasibility interval of, again, $[0, \mathrm{O}_{\max} + 2\,\mathrm{H}]$ with the added condition that an idle time must occur in the interval $[\mathrm{O}_{\max}, \mathrm{O}_{\max} + 2\,\mathrm{H}]$.

This results is refined for *synchronous periodic tasks*, where the feasibility interval can be reduced to $[0, P]$ with the added condition that P must be an idle instant.

2.2.3 Idling algorithms

The problem of deciding if a task set is feasible with idling schedule has been shown in [JSM91] to be NP-hard in the strong sense, and indeed an exhaustive search of a valid schedule asks to consider an exponential number of schedules.

Thus, heuristic approaches can be used (for example, tests based on the schedulability of the task set by NINP-EDF) but we have seen that there are task sets which are feasible and which are not schedulable by a non-idling scheduler. Another approach is to reduce the complexity by optimal decomposition (as in [YSA94]), but it is not always possible.

The authors of [GMR$^+$95] propose a branch-and-bound approach based on *valid prompt schedules* (schedules in which every job starts at its arrival time or exactly at the end of another job). This greatly reduces the number of schedule considered. Even if the theoretical complexity of the algorithm is still $O(n!)$, the authors say that it shows good performances.

2.2.4 Feasibility loss

The authors of [TDP13] study non-preemptive feasibility compared to EDF-feasibility for synchronous systems.

They do this by deriving an upper bound for the *speed-up factor* S, i.e. the minimal required processor speed-up to be able to schedule any EDF-schedulable system non-preemptively. Which means that if we had a new

processor able to compute S time units or more in the time it took the previous processor to compute one, then any EDF-schedulable system on the previous processor could be scheduled without preemption by the new processor.

Their upper-bound of S is system-dependent and is

$$S \leqslant \frac{4C_{\max}}{D_{\min}}$$

. This means that unless every deadline is significantly bigger than every execution times, the necessary speed-up factor could be as high as 4. Which would mean that to be able to schedule most systems that EDF is able to schedule, we need to multiply the speed of the processor by 4.

This shows that the great predictability of non-preemptive scheduling comes at a high cost with regard to feasibility.

2.3 Limited preemptive algorithm

As we saw, non-preemptive algorithms remove the problem of preemption times. However, they are also non-optimal, in the sense that there are some systems which are not non-preemptive-feasible. Limited preemptive algorithms are a compromises between fully preemptive and non-preemptive algorithms which greatly reduce the number of preemptions (thus diminishing their cost to the system) and which are able to schedule systems which are not non-preemptive-feasible.

In this section, we present the three main approaches of limited preemptive FTP scheduling, based on [BBY13] (we could not find study of limited preemptive FJP or DP scheduling).

To study those approaches, we use System 7, the example found in [BBY13] and which is recalled here.

Task System 7.

	O_i	C_i	D_i	T_i
τ_1	0	1	4	6
τ_2	0	3	9	10
τ_3	0	6	12	18

Note that System 7 is feasible in the general case, which we prove by giving a valid EDF schedule in Figure 9. However, it is neither DM-schedulable (Figure 11) nor non-preemptive-DM-schedulable (Figure 10). We will see that it is, however, schedulable by some limited preemptive schedulers.

The following notion is defined for FTP non-preemptive scheduling and is used in later sections.

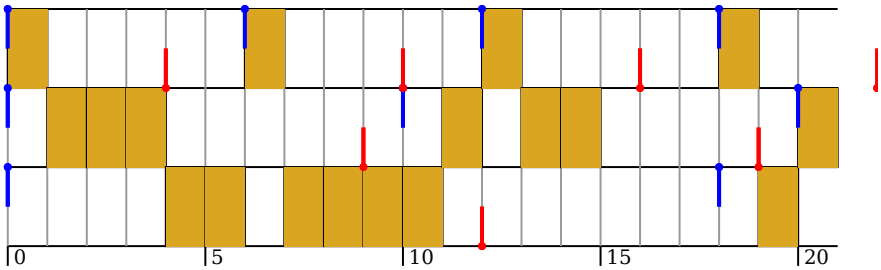

Figure 9: System 7 feasibly scheduled by EDF

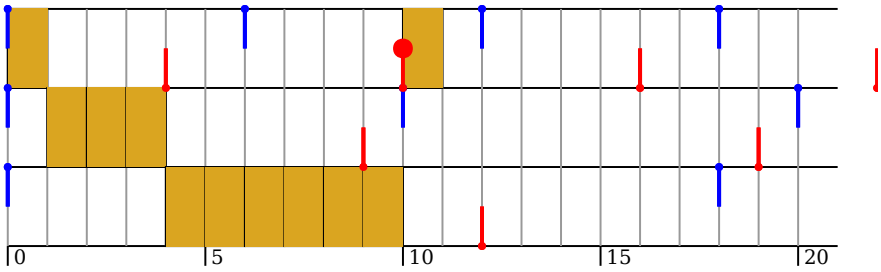

Figure 10: System 7 infeasibly scheduled by non-preemptive DM

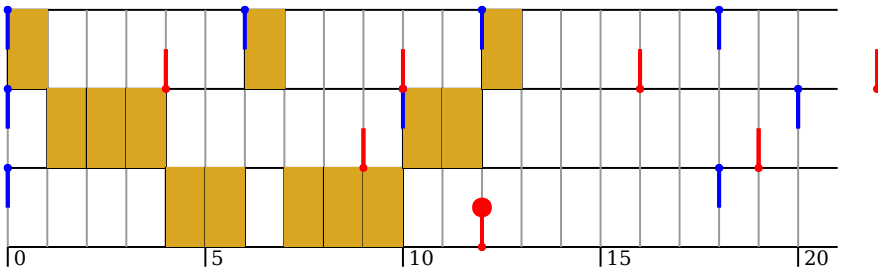

Figure 11: System 7 infeasibly scheduled by fully-preemptive DM

Definition 21 (Blocking time of a task)**.** The *blocking time* of a task is the maximal time a job of the task can be delayed because of lower-priority tasks.

With non-preemptive scheduling, it is equal to the longest computation time among lower-priority tasks, minus one (as a job has to start before the arrival of another job to be able to block it). The blocking time of the lowest-priority task is set to 0. We thus have

$$B_i = \max_{j:\mathrm{prio}(j)<\mathrm{prio}(i)} \{C_j - 1\}$$

The blocking time of a task can be used to compute the response time of a task (Definition 16), which gives the exact feasibility condition $r_i^j \leq D_i \; \forall i, j$.

We will now look at the feasibility of each of the three approaches, as given in [BBY13].

2.3.1 Preemption Thresholds Scheduling (PTS)

In this approach, each task τ_i is assigned a **preemption threshold** θ_i. Only jobs of task τ_k such that $\mathrm{prio}(k) > \theta_i$ are allowed to preempt jobs of τ_i.

Observe that with $\theta_i = \mathrm{prio}(i) \; \forall i$, the scheduling is equivalent to a fully preemptive one. Similarly, with $\theta_i = \mathrm{prio}(k) \; \forall i$ (τ_k being the task of highest priority), the scheduling is equivalent to a fully non-preemptive one. Thus it is easy to see that PTS allows us to define a compromise between fully preemptive and fully non-preemptive.

For example, System 7 scheduled by DM using preemptions thresholds $\theta_1 = 3$, $\theta_2 = 3$, $\theta_3 = 2$ produce a valid schedule (see Figure 12).

Feasibility analysis of PTS is done based on the blocking time. A task τ_i may only be blocked by tasks of lower priority whose preemption threshold is higher than $\mathrm{prio}(i)$. The blocking time is thus

$$B_i = \max_j \{C_j - 1 \mid \mathrm{prio}(j) < \mathrm{prio}(i) \leq \theta_j\}$$

This blocking time is used to compute the worst-case response-time of each task, which gives us a condition of feasibility.

It remains to show how to define the preemptions thresholds for a given system such that the preemptions are reduced to a minimum and the system remains schedulable. The authors of [SW00] give an heuristic approach which increase incrementally the preemption thresholds of each task until the system is infeasible.

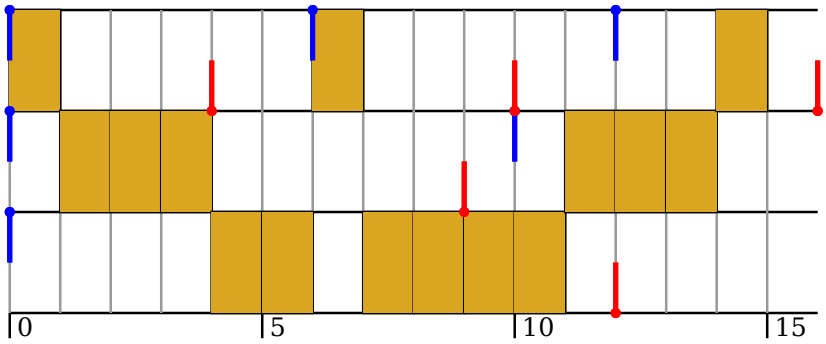

Figure 12: System 7 scheduled with the PTS approach and with $\theta_1 = 3$, $\theta_2 = 3$, $\theta_3 = 2$

2.3.2 Deferred Preemption Scheduling (DPS)

Here, each task τ_i has a non-preemptive interval of duration q_i during which it cannot be preempted. The authors differenciate between the **floating model**, where non-preemptive sections are activated by system calls in the task code, and with **activation-triggered model**, where those sections are activated by the arrival of higher priority tasks.

The activation-triggered model is a pessimistic approach which gives us a precise worst case in which the analysis can be limited to the first job of each task.

For example, System 7 scheduled by DM using maximal non-preemptive interval length of $q_2 = 2$ and $q_3 = 1$ (there is no need to define a value for q_1 as it has the highest priority) in the activation-triggered model produces a valid schedule, as seen in Figure 13.

The maximal blocking time of each task is given by the maximal non-preemptive section within the lower-priority tasks. In the floating model, we have to substract one unit of time has the non-preemptive section has to start before the arrival of a task of higher priority to be able to block it.

The blocking time of each task are thus given by:

$$B_i = \max_{j:P_j<P_i} \{q_j\} \text{ (activation-triggered model)}$$

$$B_i = \max_{j:P_j<P_i} \{q_j - 1\} \text{ (floating model)}$$

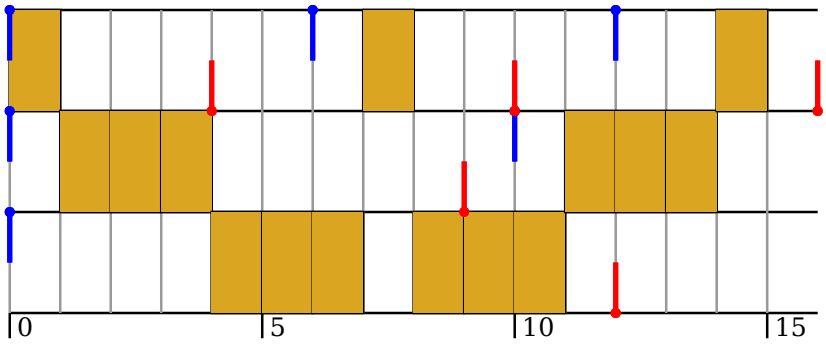

Figure 13: System 7 scheduled with the DPS approach. The non-preemptive regions begin in $t = 6$, $t = 10$ and $t = 11$

It remains to find how to determine the longest blocking time with which the system is still feasible. This has been done by the authors of [YBB09] using the notion of *blocking tolerance*, whose definition is recalled here.

Definition 22 (Blocking Tolerance [YBB09])**.** The *blocking tolerance* β_i of a task τ_i is the maximum amount of blocking τ_i can tolerate without missing any of its deadlines.

If we know the β_i values, or lower bounds of them, we can use them to compute the length of the non-preemptive section.

$$Q_i = \min_{k:\text{prio}(k)>\text{prio}(i)} \{\beta_k + 1\}$$

2.3.3 Fixed Preemption Points (FPP)

Here, tasks can specify **preemption points** in their task code, which are the only points in the execution of a job at which they can be preempted. The scheduling of the system is then similar to a non-preemptive scheduling where jobs are subdivided in multiple subjobs.

The following values are defined for each task τ_i.

- m_i is the number of subjobs into which each job of τ_i is subdivised.
- $q_{i,j}$ is the execution time of the k^{th} subjob of τ_i.
- $q_i^{\max} = \max_{k\in[1,m_i]}\{q_{i,k}\}$ is the longest execution time of the subjobs of τ_i.

- $q_i^{last} = q_{i,m_i}$ is the execution time of the last subjob of τ_i.

For example, consider System 7 scheduled by DM with the following subjobs.

τ_i	m_i	$q_{i,j}$	$q_i^{\max}$	q_i^{last}
τ_1	1	(1)	1	1
τ_2	2	(2, 1)	2	1
τ_3	2	(4, 2)	4	2

This schedule produces a valid schedule, as seen in Figure 14.

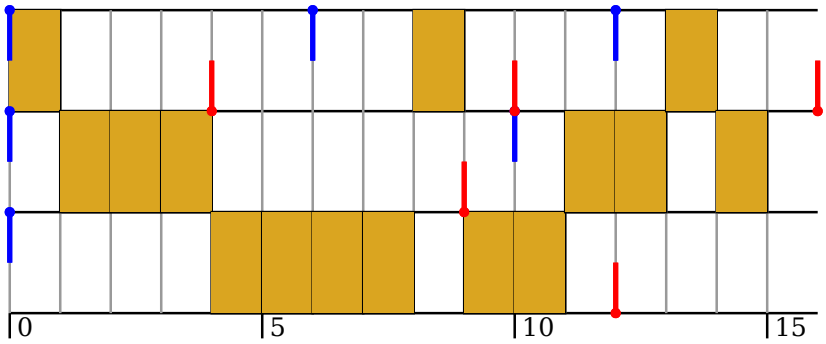

Figure 14: System 7 scheduled with the FPP approach. Jobs of τ_2 are separated into two subjobs of size 2 then 1, jobs of τ_3 are separated into two subjobs of size 4 and 2

Then, the blocking time of each task is equal to the maximal length of the subjobs of lower priority, minus one as it has to have already started to be blocking.

$$B_i = \max_{j:\text{prio}(j)<\text{prio}(i)} \{q_j^{max} - 1\}$$

The feasibility analysis is then derived from that of non-preemptive scheduling, using the subjobs instead of the tasks.

2.4 Carousel

As we saw in Section 1.1.6, preemptions costs come from many sources, one of which is the cache (or Local RAM, i.e. a fast, easily accessible memory shared by all jobs and which allows them to store temporary information).

Indeed, it is easy to see that the execution time of a job may be dependent on the state of the cache. If a job is preempted then re-activated at a later time, the state of the cache may have changed, which will increase its execution time as cache entries will have to be recreated (this effet is sometimes called task interference).

A common approach to reduce the cache-related costs is *static partitioning* ([RP07]), which consists of allocating a fixed portion of memory for each tasks, and thus results on no cache state modified by a preemption but also on reduced memory space available for each task, and an increased need for memory optimisation.

In this spirit, the authors of [WA12] suppose that the highest preemptions costs are the cache-related costs and present the Carousel approach, which reduces the occurrence of cache misses by requiring that each job starts its execution by saving the section of memory that it needs to use, and ends its execution by restoring it. Because this is always done and takes a constant amount of time, it can be easily integrated into the WCET and ensures that no task interference occurs.

This approach forces the scheduler to schedule jobs in a stack order: Suppose a job J preempts a job J_{prev}. When J finishes its execution (and thus restores the context of J_{prev}, the following job has to be J_{prev} (in which case J_{prev} will continue executing in the right context) or a new job (which will starts its execution by saving the context of J_{prev}).

In their paper, the authors assume that the scheduling algorithms is in FTP, but we think the approach could be trivially adapted to FJP schedulers.

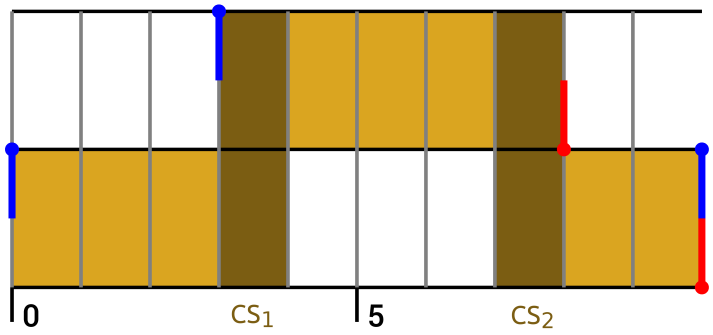

Figure 15: Representation of the added cost of Carousel. CS_1 and CS_2 are the (constant) scheduler costs before and after each preemption

If we suppose a constant scheduler cost, and because all the other costs are integrated in the WCET of the preemptive task, we can check the worst-case response time of each task (as in section 1.1.4) to check for feasibility. The

formula becomes

$$R_i = C_i + B_i + CS^1 + CS^2 + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil (C_j + CS_1 + CS_2)$$

where R_i is the worst-case response time of task τ_i, CS_1 and CS_2 are the scheduler costs, $hp(i)$ is the set of higher priority task and B_i is the blocking time caused by lower-priority tasks (induced by Carousel).

3 Integrating the preemption cost into the model

Results from Section 1.1 assumed a negligible preemption cost, which is not guaranteed in all situations. The techniques presented in Section 2 allow to reduce the impact of preemptions on the predictability of systems schedulability, but come at various costs (resource waste, feasibility loss, etc).

In this section, we consider a new model taking preemption costs into account, called the *Non-Negligible Preemption model* (NNP model). First, we describe the NNP model in Section 3.1. Then, we present results from [MYS07] giving exact feasibility conditions for FTP schedulers under this model. We then present our own results on feasibility for any scheduler.

3.1 Non-Negligible Preemption model

As we saw in Section 1.1.6, the increased load to the system caused by a preemption comes from various factors, mostly the cost of saving and restoring the job context (which is a local measurable cost) and cache misses (distributed over the execution period of the job and dependent on other factors).

Studying a model describing the behaviour of preemptions to that level of detail seems far-fetched for the scope of this master thesis, we thus make the assumption that it is possible to give an upper-bound of the cost of each preemption in the system, which we denote α and call the **preemption cost**. Moreover, we will consider that the added load caused by preemption is similar to a non-interruptible period of size α that must be completed each time a preempted job is re-activated.

The *Non-Negligible Preemption model* (NNP) is based on the popular model presented in Section 1.1, with a different behaviour when handling preemptions. The preemption behaviour of the NNP is presented in Figure 16. The bottom task is preempted twice, in $t = 4$ and in $t = 7$, following an EDF schedule. Each time its job is re-activated, it enters a non-interruptible **preemption recovery period (PRP)** of size α (drawn using a darker color). When this period is over, if the job still has the highest priority (which is the case in $t = 10$ but not in $t = 7$), it finally begins its execution and can be preempted again. Only the time units spent on the job outside of a PRP count towards the completion of a job (i.e. the bottom task has an execution time equals to 5 even if it holds the processor for 9 time units).

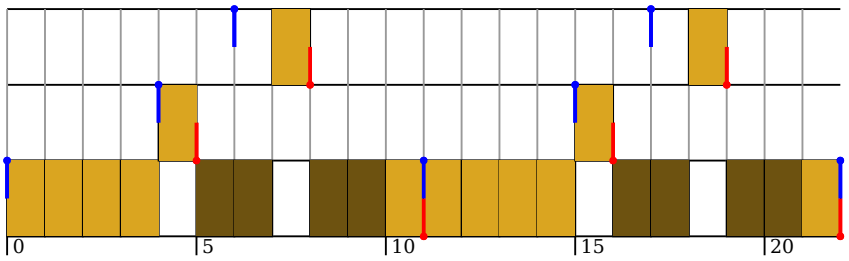

Figure 16: System scheduled by EDF in the preemption model. Note the non-interruptible preemption in $t = 6$

3.2 FTP-optimal scheduling algorithm

In [MYS07], we find feasibility conditions for the NNP model derived from those presented in section 1.1 for synchronous implicit deadline task set and considering FTP schedules only. We will present these results here.

The following values are defined:

- $N_p(J_{i,j})$: number of preemptions of the job $J_{i,j}$
- $C_{i,j} = C_i + N_p(J_{i,j}) \times \alpha$: Preemption Execution Time (PET)
- $R_{i,j}$: response time of $J_{i,j}$
- $R_i = \max_{j \in \mathbb{N}}\{R_{i,j}\}$: response time of τ_i
- $hep(\tau_i)$: set of tasks of higher priority than τ_i
- $H_i = lcm\{T_j\}_{j \in hep(\tau_i)}$: hyperperiod at level i
- $\sigma_i = \frac{H_i}{T_i}$: number of times a job of τ_i arrives during an hyperperiod at level i.

We also define the **exact total utilization factor** to be the utilization when considering preemption costs, or

$$\mathrm{U}_n^* = \mathrm{U}_n + \sum_{i=1}^{n} \frac{1}{\sigma_i} \left(\sum_{j=1}^{\sigma_i} \frac{N_p(J_{i,j}) \times \alpha}{T_i} \right)$$

We observe that when $\alpha = 0$, the exact total utilization factor is equal to the utilization.

We also have that $\mathrm{U}_n^* \leq 1$ is a necessary condition. In fact, [MYS07] shows that this condition is actually sufficient. But it remains to find a way to compute the values of $N_p(J_{i,j})\ \forall i, j$.

To find them (and schedule the system), the authors propose a 13-step backtracking algorithm for each task (from the task of highest priority to the one of lowest priority) based on the slots occupied by higher priority task, organised in T_i-mesoid.

Note that those results were recently extended to the multiprocessor model by [Ndo14].

3.3 Main properties of optimal online scheduling

Using the NNP model rather than the popular model of Section 1.1 implies that previous results may not hold.

In this section, we concentrate on online (during runtime) scheduling of tasks in the NNP model, as the offline scheduling was mostly covered in Section 3.2. When considering online schedulers, we must also consider the load that it adds to the system, as a heavy load not taken into account would weaken the predictability of the analysis. We thus consider online job schedulers, with no (or few) knowledge of periodicity (i.e. arrival times). This will be discussed in Section 3.3.3.

The results are mostly negative and indicate that the complexity of an optimal online scheduler is greater than in the popular model.

The entirety of this section is personal, original work unless otherwise noted.

3.3.1 EDF and LLF

In Section 1.1, we introduced the notion of *optimal scheduler*, i.e. schedulers which feasibly schedule any feasible system. With negligible preemption costs, the two main optimal schedulers were the Earliest-Deadline-First (EDF) and Least-Laxity-First (LLF) schedulers, with other optimal schedulers often being an extension of one of the two optimized for some property.

Lemma 1. *Within the NNP preemption model, neither EDF nor LLF are optimal schedulers*

Proof. Let us look at Task System 8:

Task System 8.

	O_i	C_i	D_i	T_i	α
τ_1	0	3	6	6	2
τ_2	1	2	4	4	2

Figure 17 shows that it is in fact possible to schedule this system such that no preemptions occur and every deadline is respected (i.e. system 8 is feasible).

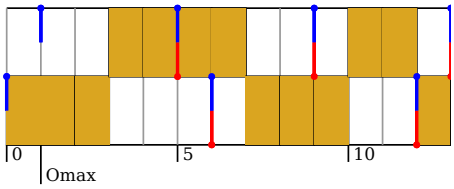

Figure 17: Task System 8 feasibly scheduled. After $t = 12$, the behavior is periodic.

However, Figure 18 shows the same system scheduled by EDF, with a deadline miss. This means that there is at least one feasible system that EDF does not schedule, i.e. that EDF is not optimal with the NNP preemption model. LLF produces the exact same schedule, which means it is not optimal either.

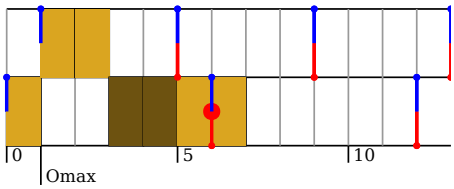

Figure 18: Task System 8 scheduled by EDF (or LLF), with a deadline miss at $t = 6$

□

So neither EDF nor LLF are optimal for the NNP preemption model. Furthermore, we argue that they are poor choice to study the feasibility of systems in this model as schedulability test based on them are not sustainable (see Definition 20), as proven in the following theorems and lemmas.

Lemma 2. *Within the NNP model, no schedulability test for EDF (resp. LLF) is sustainable with regard to the execution times.*

Proof. Consider the following system, with two different values for C_3.

Task System 9.

	O_i	C_i	D_i	T_i	α
τ_1	3	2	2	9	2
τ_2	0	4	9	9	2
τ_3	0	2 / 3	5	9	2

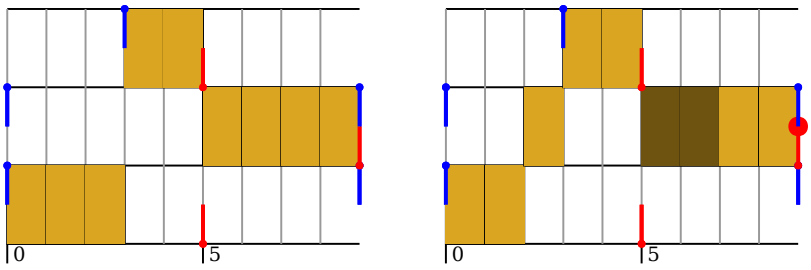

Figure 19: System 9 scheduled by EDF (or LLF) with different values of C_3. When $C_3 = 3$ (left), EDF (or LLF) produces a valid schedule. When $C_3 = 2$ (right), EDF (or LLF) causes a deadline miss at $t = 9$. This illustrates the non-sustainability of EDF and LLF w.r.t. the execution times.

Figure 19 shows system 9 scheduled by EDF (or LLF), with different values of C_3. With the longest execution time ($C_3 = 3$), every deadline is met. However, when the execution time of τ_3 is lower ($C_3 = 2$), the execution of $J_{2,1}$ begins earlier, at $t = 2$, then is immediately stopped at $t = 3$ by the arrival of $J_{1,1}$ (which has an earlier deadline, thus a higher priority). This causes $J_{2,1}$ to enter a preemption recovery period at $t = 5$, and a deadline miss to occur at $t = 9$. It follows that the system with the bigger C_3 is schedulable by EDF (or LLF) while the system with the smaller C_3 is not.

□

Theorem 2. *Within the NNP model, no schedulability test for EDF (resp. LLF) is sustainable.*

Proof. Direct consequence of Lemma 2 (by Definition 20). □

Remember that the sustainability of EDF and LLF with regard to the execution times was the justification for using WCET values to model the different possible execution times of the jobs of a same task (see Section 1.1.5). Lemma 2 means that in the NNP model, a system deemed schedulable by EDF or LLF may actually miss a deadline during its execution if a job finishes its execution earlier than the WCET.

While it is possible to simulate the WCET for all jobs by idling during the time between the actual and expected end of execution, this illustrates the fact that EDF and LLF are poor choices to study the feasibility of systems in the NNP model.

So neither EDF nor LLF are sustainable in general, because they are not

sustainable with regard to the execution times. We now discuss their sustainability with regard to other parameters.

Lemma 3. *Within the NNP model, no schedulability test for EDF (resp. LLF) is sustainable with regard to the relative deadlines.*

Proof. Consider the following system with two possible values for D_1.

Task System 10.

	O_i	C_i	D_i	T_i	α
τ_1	0	1	4/6	7	2
τ_2	0	2	5	7	2
τ_3	1	2	2	7	2

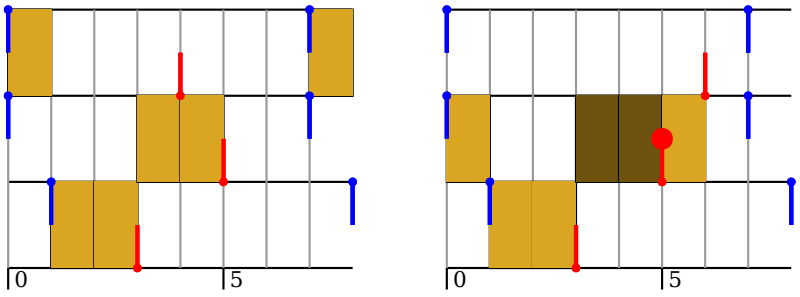

Figure 20: System 10 scheduled by EDF (or LLF) with two possible D_1 values (Left: lower value. Right: higher value). With the *longest* value, a deadline miss occurs at $t = 5$. This illustrates the non-sustainability of EDF (or LLF) w.r.t. the relative deadline.

Figure 20 shows system 10 scheduled by EDF (or LLF) with two different values of D_1. With the shortest value, all job deadlines are met. With the longest value, EDF (and LLF) choose to execute $J_{2,1}$ at $t = 0$ instead of $J_{1,1}$, which causes a preemption at $t = 1$, a preemption recovery period at $t = 3$, and ultimately a deadline miss at $t = 5$.

Note that for LLF, the laxity of $J_{1,1}$ and $J_{2,1}$ is equal ($\ell = 3$). In Section 1.1.3, we said that such ties were decided (deterministically) by choosing the job of lowest task identifier. This means that the system is LLF-schedulable when described as such, but not if we switch the order in which τ_1 and τ_2 are defined.

This system is thus an example of system schedulable by EDF and LLF with the shortest deadline value, and non-schedulable by EDF or LLF with the longest value. Schedulability by EDF or LLF is thus non-sustainable with regard to the relative deadline.

□

Lemma 4. *Within the NNP model, no schedulability test for EDF (resp. LLF) is sustainable with regard to the arrival times.*

Proof. Consider the following system with two possible values for O_1.

Task System 11.

	O_i	C_i	D_i	T_i	α
τ_1	0 / 1	1	3	5	2
τ_2	0	2	5	5	2
τ_3	1	1	1	5	2

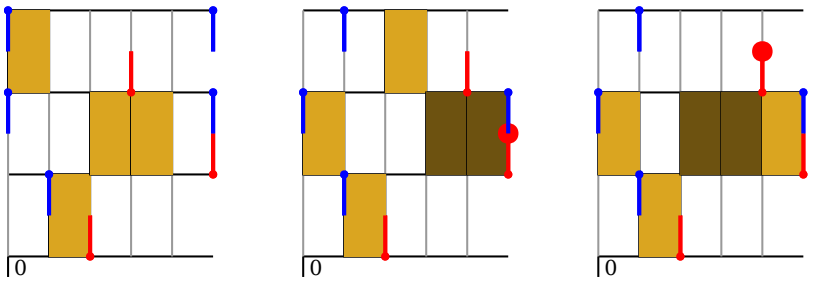

Figure 21: System 11 scheduled by EDF (or LLF) with two possible A_1 values (Left: EDF or LLF with earlier value. Center: EDF with later value. Right: LLF with later value). With the *latest* value, a deadline miss occurs at $t = 4$ (EDF) or $t = 5$ (LLF). This illustrates the non-sustainability of EDF (or LLF) w.r.t. the arrival time.

Figure 21 shows system 11 scheduled by EDF (or LLF) with two different values of A_1. With the earliest value, all job deadlines are met. With the longest value, EDF (and LLF) choose to execute $J_{2,1}$ at $t = 0$, which causes a preemption at $t = 1$, and thus a preemption recovery period which ultimately causes a deadline miss.

This system is thus an example of system schedulable by EDF and LLF with the earliest arrival value, and non-schedulable by EDF or LLF with the latest value. Schedulability by EDF or LLF is thus non-sustainable with regard to the arrival times.

□

We continue our review of EDF and LLF for feasibility analysis by showing another flaw they possess in this model, based on the feasibility interval (Definition 18. As we have seen in Section 1.1.4, when the preemption

times are negligible we have that $[0, \mathrm{O_{max}} + 2\,\mathrm{H}]$ is a feasibility interval for EDF.

Lemma 5. *Within the NNP,* $[0, \mathrm{O_{max}} + 2\,\mathrm{H}]$ *is not a feasibility interval for EDF, even when* $\mathrm{U_{tot}} \leq 1$.

Proof. We recall Definition 18.

> The *feasibility interval* of a system is a finite interval of time such that, if no job deadline is missed within it, no other deadline will be missed by the system.

Then, consider the following system.

Task System 12.

	O_i	C_i	D_i	T_i	α
τ_1	6	4	11	11	3
τ_2	4	1	1	11	3
τ_3	0	5	11	11	3

And its EDF schedule, as shown in Figure 22.

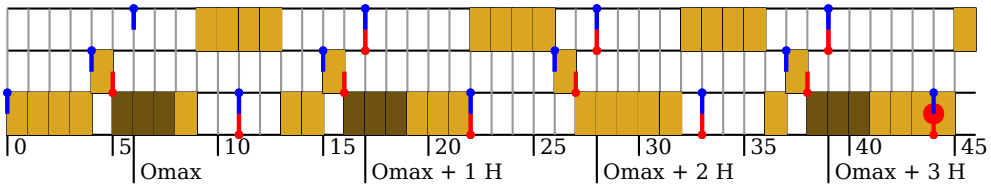

Figure 22: EDF schedule of system 12. A deadline miss occurs at $t = 44$, with $44 > \mathrm{O_{max}} + 2\,\mathrm{H} = 28$

We see that the EDF schedule is invalid, as a deadline miss occurs at $t = 44$. However, no deadline misses occurs before $\mathrm{O_{max}} + 2\,\mathrm{H} = 6 + 2 * 11 = 28$. It directly follows from the definition that $[0, \mathrm{O_{max}} + 2\,\mathrm{H}]$ is not a feasibility interval.

□

Lemma 5 implies that the feasibility test based on EDF simulation until $\mathrm{O_{max}} + 2\,\mathrm{H}$ is not sufficient (and thus not exact either). Indeed, if we applied it to System 12, we would have $\mathrm{U_{tot}} = \frac{10}{11} \leq 1$ and no deadline miss during the feasibility interval. This test would thus deem System 12 feasible when it does in fact miss a deadline at $t = 44$.

3.3.2 No optimal schedulers is in FJP

In Section 1.1.3, we described three types of schedulers: FTP, which assign constant priorities to each *task*; FJP, which assign constant priorities to each *job*, and DP, with which job priorities may change at any instant. Note that $FTP \in FJP \in DP$.

With negligible preemption costs, EDF was a FJP scheduler which was optimal. Thus, it is possible to determine feasibility by simulation of EDF on the feasibility interval. Simulating a FJP scheduler require little overhead to compute priority as they must only be computed at job arrivals, rather than at each instant as in DP schedulers.

This property does not hold in the preemption model:

Lemma 6. *Within the NNP preemption model, no FJP scheduler is optimal.*

Proof. Consider the following system:

Task System 13.

	O_i	C_i	D_i	T_i	α
τ_1	0	10	16	16	2
τ_2	0	1	4	4	2

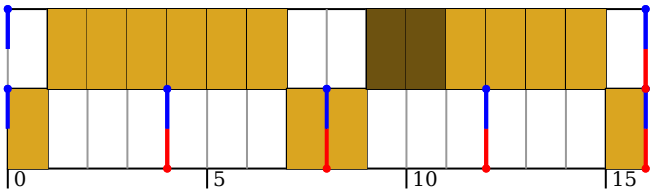

Figure 23: The only valid schedule of Task System 13

Figure 23 shows the only valid schedule of this system. Indeed, any other choice made during the execution leads to more preemptions, and thus a deadline miss (because each preemption adds load to the system, and the system is already at full capacity).

For example, if we start with $J_{1,1}$ rather than $J_{2,1}$ at $t = 0$, Then, at $t = 3$, it will be absolutely necessary to preempt it for $J_{2,1}$ to complete before its deadline. In that case, at $t = 4$, $J_{2,1}$ is finished and $J_{1,1}$ has still 2 units of preemption recovery period and 7 units of execution remaining. It will thus not be completed at $t = 7$, when it will be absolutely necessary to preempt it for $J2, 2$ to complete before its deadline. This second preemption

will increase the load of the system above an utilization of 1, resulting in a deadline miss.

Another example: let us preempt $J_{1,1}$ at $t = 4$ in order to execute $J_{2,2}$ earlier. In that case, $J_{1,1}$ will finish its preemption recovery period at $t = 7$, with 7 execution units left. Thus, at $t = 11$, it will not be finished and it will be absolutely necessary to preempt it for $J_{2,3}$ to complete before its deadline. Again, this second preemptions will cause a deadline miss.

Other strategies are either similar to the two examples, or completely counter-productive. We leave the rest of the proof as an exercise to the reader. Note that because there are only two tasks and a hyperperiod $\mathrm{H} = 16$, there are at most 2^{16} possible schedules (most of which may be quickly eliminated).

Once we are convinced that the schedule of figure 23 is the only valid schedule of system 13, we show that no FJP schedule could have produced this exact schedule.

Let us prove this by contradiction. Suppose that the schedule of figure 23 was produced by a FJP scheduler. At $t = 4$, $J_{1,1}$ continues its execution even though $J_{2,2}$ has entered the system, which means that $p(J_{1,1}, 4) > p(J_{2,1}, 4)$ (where $p(J, t)$ gives the priority of J at time t). However, at $t = 7$, $J_{2,2}$ preempts $J_{1,1}$, which means that $p(J_{1,1}, 7) < p(J_{2,2}, 7)$. This implies that $p(J_{1,1}, 4) \neq p(J_{1,1}, 7)$ or $p(J_{2,2}, 4) \neq p(J_{2,2}, 7)$ (or both). This is a contradiction has the priority of a job remains constant in a FJP schedule.

To conclude:

- System 13 is not schedulable by any FJP scheduler.
- System 13 is feasible, as shown in figure 23.
- No FJP scheduler is optimal.

□

3.3.3 Context-Free scheduling and Clairvoyance

We define the following notions:

Definition 23 (Job configuration)**.** The configuration $C^t_{i,j}$ of the job $J_{i,j} = (a_{i,j}, d_{i,j}, c_{i,j})$ of task $\tau_i = (O_i, T_i, D_i, C_i)$ at instant t is

$$C^t_{i,j} = (\tau_i, J_{i,j}, \theta_{i,j}, \epsilon_{i,j}, \mathrm{exec}_{i,j}, \mathrm{preemp}_{i,j})$$

Where

- $\theta_{i,j} = t - a_{i,j}$ is the number of time units elapsed since $a_{i,j}$)

- $\epsilon_{i,j}$ is the number of time units during which $J_{i,j}$ has executed before t.
- $\mathrm{exec}_{i,j}$ is equal to 1 if the job was executing at instant $t-1$, 0 otherwise
- $\mathrm{preemp}_{i,j}$ is equal to the number of time units left in the preemption recovery period. If $J_{i,j}$ has not be preempted yet or has finished its preemption recovery period, we let $\mathrm{preemp}_{i,j} = 0$

Note that while this definition is similar to the notion of configuration as found in the literature, we added the attributes related to the preemption model, i.e. $\mathrm{preemp}_{i,j}$ and $\mathrm{exec}_{i,j}$ (as this is known at runtime and allow a scheduler to know if a priority assignment may cause a preemption or not).

Definition 24 (Context-free schedulers)**.** A scheduler is said to be *context-free* if the priority it gives to a job $J_{i,j}$ at a time t can be expressed as a function of its job configuration $C^t_{i,j}$.

In other words, a context-free schedulers does not use information about other jobs to determine one job priority. Note that FTP, FJP and DP schedulers may all be either context-free or not.

All the schedulers we discussed previously (except those discussed in section 2.2) were context-free. Indeed, they can be defined by a function on a job configuration:

- $\mathrm{prio}_{RM}(C^t_{i,j}) = 1/T_i$
- $\mathrm{prio}_{DM}(C^t_{i,j}) = 1/D_i$
- $\mathrm{prio}_{EDF}(C^t_{i,j}) = 1/d_{i,j}$
- $\mathrm{prio}_{LLF}(C^t_{i,j}) = 1/(D_{i,j} - \theta_{i,j} - (c_{i,j} - \epsilon_{i,j}))$

Furthermore, we define the following subclass of non-context-free schedulers:

Definition 25 (Clairvoyant scheduler)**.** A scheduler is said to be *clairvoyant* if it is not context-free and the priority it gives to a job $J_{i,j}$ at a time t can be expressed as a function of the configuration of **all** the current jobs and the arrival times and initial configuration of some future jobs.

Note that the priority assignment of any context-free scheduler can be expressed as a function of all the current jobs and the information on future jobs (as a function is not forced to use all of its parameter). We thus explicitly exclude context-free schedulers from Definition 25.

Definition 26 (Horizon)**.** The *horizon* of a clairvoyant scheduler is the maximal distance between t and an arrival used to determine the priority of a job at t.

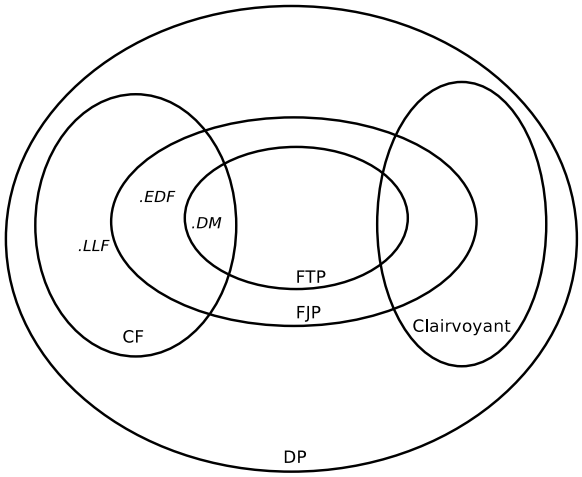

Figure 24: Relationship between classes of schedulers

Figure 24 shows the relationship between the different classes of scheduling. Remember that FTP is a particular case where all the priorities are fixed before the first instant, whereas both FJP and DP may be run during the execution of the system. Therefore a clairvoyant FTP scheduler only makes sense for sporadic systems.

Note that the notion of clairvoyance might seem irrelevant with periodic systems as the arrival of all jobs is known in advance by definition. All schedulers with the knowledge of the periodicity of the system are thus clairvoyant. However a scheduler making use of the future arrivals will have a complexity dependent on the number of tasks and its horizon, and a high complexity may cause significant overhead that hurts the predictability. We thus use this notion in a periodic context to quantify the complexity of a non-context-free scheduler.

Indeed, if a scheduler needs more than the job configuration to determine the job priority, how do we quantify "more"? The previously studied scheduler were context-free, the input they required to decide each job priority was of constant size. A non-context-free non-clairvoyant scheduler would require the configuration of all the jobs currently in the system, which is an input size linear in the number of tasks. As for clairvoyant schedulers, the input size is dependent on its horizon, which could possibly go as high as H (input size exponential in the number of tasks).

With that in mind, we now show that an optimal scheduler cannot be context-free by first proving the stronger statement that an optimal scheduler must be clairvoyant.

Lemma 7. *Within the NNP preemption model, an optimal scheduler must be*

clairvoyant.

Proof. Consider the two following systems, identical except for the offset value of their first task.

Task System 14.

	O_i	C_i	D_i	T_i	α
τ_1	10	1	1	12	1
τ_2	0	5	12	12	1
τ_3	4	5	6	12	1

Task System 15.

	O_i	C_i	D_i	T_i	α
τ_1	9	1	1	12	1
τ_2	0	5	12	12	1
τ_3	4	5	6	12	1

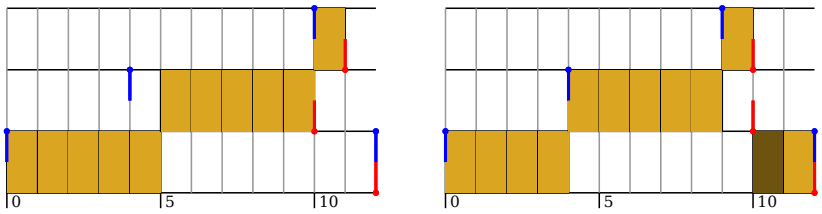

Figure 25: The only valid schedule of systems 14 and 15. After $t = 12$, both behaviors are periodic.

Figure 25 shows the only valid schedule of those systems. Indeed, for system 14, $J_{1,1}$ cannot execute at any other times that $t = 10$ and $t = 11$, meaning that those times are not accessible for $J_{3,1}$, while $J_{2,1}$ must still execute for 5 units of time between $t = 4$ and $t = 9$. The presented execution pattern is the only valid one under these constraints. We have a similar situation for system 15, where $J_{1,1}$ may only execute at $t = 9$, therefore $J_{2,1}$ is forced to execute between $t = 4$ and $t = 8$, and this then forces $J_{3,1}$ to execute in the remaining slots.

We now show that both schedule cannot be obtained by a non-clairvoyant scheduler. Indeed, consider $t = 4$ in both execution, the scheduler must choose between a job of τ_2 that has been executing for 4 unit of time, and a job of τ_3 that has just arrived. The configuration of every job is thus the same, however for system 14 the job of τ_2 must selected and for system 15 the job of τ_3 must be selected. This means that the scheduler should return different priority values for the same job configurations, hence it need access to the only difference between the two execution, i.e. the date of arrival of the first job of τ_1.

To summarize, both systems are feasible as there exist one (and only one) valid schedule for each of them (figure 25). These two schedules cannot both be achieved by the same non-clairvoyant scheduler, as it would need to make different choice with the same input at $t = 4$. Therefore no non-clairvoyant scheduler is able to feasibly schedule both those feasible systems, therefore no non-clairvoyant scheduler is optimal □

Lemma 8. *Within the NNP preemption model, no optimal algorithm is context-free.*

Proof. Follows directly from lemma 7, as an optimal scheduler must be clairvoyant, and no clairvoyant scheduler can be context-free (by definition). □

In conclusion, an optimal scheduling algorithm for the NNP preemption model has to be of a more complex class of algorithms that those previously studied. It remains an open question how much more complex it has to be, as the minimal horizon necessary for an optimal algorithm is not known. Possible candidates are:

- until the next idle instant (if there is one) (see Definition 19)
- until the latest deadline of the currently active jobs
- until stabilization of the schedule (i.e. exponential complexity)

3.3.4 Idling scheduling strictly dominates non-idling scheduling

In the model with negligible preemption costs, we differentiated between idling schedulers (which could inject idle units in the schedule even when active jobs were present), and non-idling (or work-conserving) schedulers. This was useful because, even though idling schedulers seems to be a more general class, both EDF and LLF were optimal and work-conserving, meaning that any system which was feasible with idle units had at least one valid schedule without any (the one produced by EDF or LLF, which is valid by definition).

Indeed, if there is no penalty for switching from a job to another, filling the idle units with active jobs will always result in shorter response times, i.e. more feasibility.

We now show that this property is not conserved in the NNP preemption model.

Lemma 9. *Within the NNP preemption model, idling scheduling strictly dominates non-idling scheduling.*

Proof. Consider Task System 16:

Task System 16.

	O_i	C_i	D_i	T_i	α
τ_1	1	1	6	6	2
τ_2	0	10	12	12	2

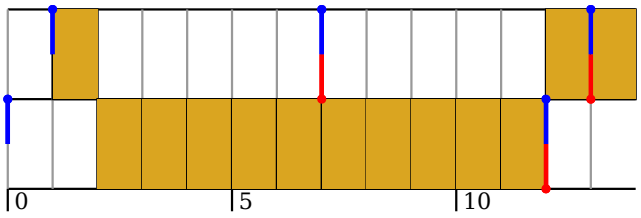

Figure 26: The only valid schedule of Task System 16, with an idle unit at $t = 0$. After $t = 13$, the behavior is periodic.

Figure 26 shows an idling schedule of system 16, as no jobs execute at $t = 0$ when there is an active job ($J_{2,1}$).

It is also the only valid schedule of the system. To convince ourselves of this, note that $2C_1 + C_2 = 12$ units of computations must be executed before $t = 13$, which means that we cannot encounter a single preemption (which would require 2 units of preemption recovery period). With that in mind, consider that if $J_{2,1}$ starts earlier, it will be preempted before $t = 7$ and will thus lead to a deadline miss. $t = 0$ must thus be an idle unit. At $t = 1$, we have to choose between $J_{1,1}$ and $J2, 1$. Choosing $J_{2,1}$ would lead to a preemption before $t = 7$, so we must choose $J_{1,1}$. The rest of the schedule is the only valid option as we cannot preempt or inject another idle unit.

Because the only valid schedule of system 16 may only be obtained from an idling scheduler, this system is an example of a feasible system which is not schedulable by any work-conserving scheduler. Because such a system exists, and because idling scheduling is a more general case of work-conserving scheduling, idling scheduling strictly dominates work-conserving scheduling. □

Lemma 9 was to be expected, as it was also the case for strictly non-preemptive scheduling. However, in non-preemptive scheduling, NINP-EDF (which can be implemented as plain EDF if you force the executing job to not be preempted until completion) was *non-idling optimal*, i.e. it scheduled any system that was feasible by a non-idling algorithm.

The following lemmas shows that this result does not translate into the NNP model.

Lemma 10. *In the NNP preemption model, EDF is not non-idling optimal.*

Proof. From the proof of lemma 6, system 13 has only one valid schedule and no FJP scheduler could produce that schedule. Or, EDF is a FJP scheduler, and the valid schedule does not have any idle unit. It follows that EDF does not schedule a system which has a valid non-idling schedule, i.e. EDF is not non-idling optimal. □

The following lemmas give some intuition on the optimal behaviour of idling units. They are mostly intuitive, but serve to show that even though we now consider idling algorithms, the complexity does not necessarily increase a lot because idle units are only necessary in very specific contexts.

Lemma 11. *If a valid schedule of a system exists where instant t is an idle unit with at least one active job in the system, we either have that one of the following is true*

- $t = 0$
- *a job finished its execution at $t - 1$*
- *there was another idle unit at $t - 1$*

or we have that there exists another valid schedule with the same execution before t and where t is not an idle unit.

Proof. Suppose a valid schedule s with an idle unit t which does not respect these conditions. Then, a job $J_{i,j}$ was running at $t - 1$ and is preempted to allow for the idle unit at instant t. We will show that there always exist another valid schedule without such an idle unit.

Consider the schedule s' similar to s except that $J_{i,j}$ is allowed to execute during t. At instant $t + 1$, the state of the system is the same except that $J_{i,j}$ has executed one more unit, a net positive in terms of schedulability. Therefore if s was valid, it is possible to construct a valid schedule after t in s'. □

Lemma 12. *If there exist a schedule of a system with an idle unit t where active jobs are present, either the next job arrival is at a time t' such that $t' - t < \alpha$, or there exist another valid schedule with the same execution before t and where t is not an idle unit.*

Proof. Suppose there exists a valid schedule s with such an idle unit. We will show that it can be transformed into another valid schedule without such an idle unit. Indeed, consider that there are two possibilities.

- No job execute between $t+1$ and t' (case 1)
- At least one job starts its execution before t'. In particular, we call the first job to execute j and the instant at which it starts t'', such that $t+1 \leq t'' < t'$ (case 2)

Consider the schedule s' where instead of an idle unit, we start executing one of the active jobs at t (any active job in case 1, j in case 2). Then we have the following possibilities.

- In case 1, the job finishes before t'. Then the outcome of s' is a net positive over s (one job finishes sooner without preemption), so the idle unit was not necessary.
- In case 1, the job is preempted at t'. Then it has been executing for $t' - t \geq \alpha$ units but will need to go through a preemption recovery period of size α. In other words, the amount of computation necessary to complete the job (including the preemption recovery period) has decreased by $t' - t - \alpha \geq 0$. Therefore this never has any negative implication to the load of the system.
- In case 2, j finishes before t''. Then at t'' the system is in the same state except that j is already finished, a net positive.
- In case 2, job j execute between t and t''. At t'', we have the same state of the system as in s except that job j has executed for $t'' - t$ more time units, a net positive.

In conclusion, we are always able to modify the schedule to remove such an idle unit while maintaining the validity of the schedule. □

Lemmas 11 and 12 give us the following theorem.

Theorem 3. *A scheduling algorithm does not need to consider putting an idle unit at instant t if any of the following is true*

- *The next job arrival is at a time t' such that $t' - t \geq \alpha$*
- *A job was executing at $t-1$ and is not finished*

Proof. Direct consequence of lemmas 11 and 12. □

This shows us how the possibility of idle units affects the complexity of scheduling. Indeed, idling scheduling is often found to be of exponential complexity because it require an exhaustive search amongst the possible idling schedules. Theorem 3 gives us a sufficient condition for the absence of an idling unit, which can be used to discard most of those schedules.

The next theorem gives us a sufficient condition to justify the *presence* of an idling unit.

Theorem 4. *If a system has a valid schedule where a job $J_{i,j}$ begins its execution at a time t, then preempted at a time $t' > t$ such that $t' - t < \alpha$, then there exists another valid schedule where the execution is the same before t, but t is an idle unit.*

Proof. Consider a schedule s where there is such a time t and t'. Then consider the schedule s' where $J_{i,j}$ is **not** started at t, and the interval $[t, t')$ is composed of idle units. The job that preempted $J_{i,j}$ is started at t' as in s.

In that case, after t', the state of the system is the same as in s except that $J_{i,j}$ has executed for $t' - t < \alpha$ time units less and was not preempted, its load has thus increased of $\alpha - (t' - t) > 0$ compared to s, a net positive in terms of schedulability.

We have thus constructed a valid schedule s' without such time t and t'. □

Theorem 4 gives us a sufficient condition to justify the presence of an idle unit at t: if every active jobs at t would have been preempted less than α time units later, then it is better to idle.

Note that in order to use these results, a scheduler needs to be clairvoyant as the next arrival time must be known.

To conclude on the subject of idling, we introduce this interesting corner case.

Lemma 13. *If a system τ with $\alpha = 1$ has at least one valid schedule, then there always exist a valid schedule of τ such that there are no time unit t such that there are active jobs at t and t is a idle unit.*

Proof. Directly follows from theorem 3. If $\alpha = 1$, then at time t the next job arrival is at least at time $t' = t + 1$, so the condition $t' - t \geq \alpha$ is always verified. □

This means that in the particular case of $\alpha = 1$, idling algorithms do not strictly dominate work-conserving algorithms. Indeed, in that case, the penalty of idling for one unit is equal to the penalty of launching a job and preempting it immediately. Therefore there are never any strict advantage of idling, unless we want to minimize preemption. In that case, idling algorithms may feasibly schedule a system with less preemption that an optimal work-conserving algorithm.

3.4 On the complexity of the feasibility analysis

3.4.1 NP-hardness

When studying scheduling algorithms for the preemption model, the question of the possible NP-hardness of the scheduling problem must be asked. A first, pessimistic result is the following.

Lemma 14. *The problem of deciding if a system is feasible in the NNP preemption model for all values of α is NP-hard in the strong sense.*

Proof. We know that in the NNP model, it is necessary to consider idling schedules for some systems. We can thus prove the NP-hardness by restriction (see [GJ79], section 3.2.1) to idling non-preemptive scheduling (Section 2.2) by considering only α values such that $\alpha > D_i - C_i \; \forall i$.

Indeed, in that case, a preemption will cause a preemption recovery period longer than the initial laxity of the preempted job, which will always lead to a deadline miss. □

The idea of Lemma 14 is that if we allow any α value, the problem is obviously at least as complex as non-preemptive scheduling because great values of α are equivalent to forbidding preemptions. However, even if the model was conceived to study cases where preemptions times are not negligible compared to the duration of the time unit, the motivational examples taken from actual systems expressed preemption costs as a fraction of the execution times, which is far from the preemption times necessary to make scheduling completely non-preemptive.

3.4.2 With restricted preemption cost values

It would then be more interesting to study the complexity of scheduling systems within the NNP preemption model for α values where some preemptions can still occur, i.e. $\alpha < \max_i(D_i - C_i)$. In that case, we cannot prove NP-completeness by restriction.

The exact complexity class of the scheduling problem in that case is still an open question. However, there are few hopes for it not being at least as hard as the scheduling problem with negligible preemptions, for which the simpler feasibility problem has already been shown to be co-NP-hard in [LW82]. Indeed, in their review of the complexity of scheduling with negligible preemptions, the authors of [BG04] write

> "An exponential-time feasibility test [for negligible preemptions] is known, which consists essentially of simulating the behavior

> of EDF upon the periodic task system for a sufficiently long interval." [BG04]

In the general case, this interval has a length of at least $\mathrm{H} = \mathrm{lcm}(T_1, \ldots, T_n)$, which has been shown to be exponential in the number of tasks in [GM01].

In the NNP model, Lemma 5 tells us that this interval is not a feasibility interval for EDF. The minimal length of a feasibility interval for EDF, or for an optimal scheduler, is still an open question.

An upper bound of the length of a feasiblity interval for any deterministic scheduler will be discussed in Section 5.2.6 in order to determining the termination conditions of our simulator.

3.4.3 Using Definitive Idle Times

In some cases, we can define a feasibility interval for an optimal schedulers, even without knowing one. To that aim, we recall Definition 19.

> A time t_i in a schedule of a system is said to be an *idle point* (or *idle time*) if at time t_i, every job released strictly before t_i is completed.

Idle *times* are not to be confused with idle *units*, which are units where no jobs execute. They hold many interesting properties, one of which is described by the authors of [CGG04] and is reworded in Theorem 5. Note that this result trivially holds in the NNP model.

Theorem 5. *(Adapted from [CGG04]) In a schedule s of some system τ, if an idle time occurs at a time $t_c > \mathrm{O}_{\max}$ and another idle time occurs exactly at $t_c + \mathrm{H}$, then the state of the system at $t_c + 1$ and $t_c + \mathrm{H} + 1$ is exactly the same, and s can be repeated for the rest of the execution.*

Proof. For the complete proof, we redirect the reader to the original paper. The general idea is that because t_c and $t_c + \mathrm{H}$ are both idle times and occurs at a distance H, the configuration and arrival pattern of all subsequent jobs is going to be the same. □

Figure 27 illustrates Theorem 5. We see that the patterns of arrival and deadline of all jobs repeat with a period of H. Furthermore, there are two idle times separated by an interval of H ($t_c = 20$ and $t_c + \mathrm{H} = 40$), and we can see that after $t_c + \mathrm{H}$ the execution patterns is repeated.

It follows that if it exists such a time t_c, then $[0, t_c + \mathrm{H} + 1]$ is a feasibility interval.

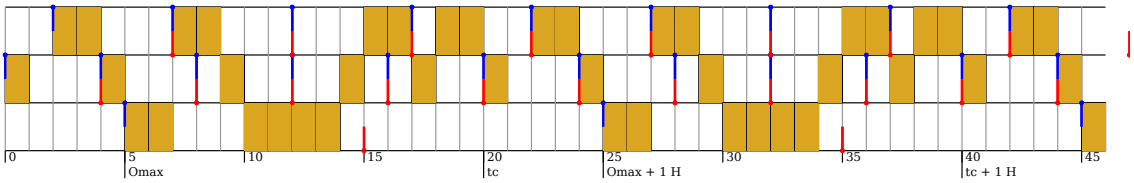

Figure 27: An idle time occurs both at $t_c = 20$ and $t_c + \mathrm{H} = 40$. We see that the configuration and arrival of subsequent jobs is the same.

Another contribution by the authors of [CGG04] is Theorem 6. It remains an open question if this result hold in the NNP model.

Theorem 6. *(Adapted from [CGG04]) A time t_c as described in Theorem 5 always exists in feasible systems, and it occurs before* $\mathrm{O}_{\max} + \mathrm{H} + 1$.

If Theorem 6 does hold in the NNP model, we know that an optimal algorithm would have $[0, \mathrm{O}_{\max} + 2\,\mathrm{H}]$ as a feasibility interval. In the meantime, we may check for the existence of idle times during the simulation of an algorithm. If two occurs at a distance H and no deadline misses have occurred before, we know that no further deadline misses will occur.

Even without simulation or the knowledge of an optimal scheduler, we are able to predict the existence of idle times in some cases, which we will describe now. We introduce the following notion, defined in [LGBB13].

Definition 27 (Definitive Idle Time [LGBB13])**.** A time t_d is said to be a *definitive idle time* (DIT) for a system if at time t_d, there is no job released strictly before t_d having an absolute deadline strictly after t_d.

Note that the DIT is defined for a system, whereas idle instants were defined for the schedule of a system. If a schedule contains a DIT t_d, then t_d is guaranteed to be an idle instant in every valid schedule.

In the NNP, the presence or absence of DIT is also independent of the value of α.

In a previously published work ([CRGG13]), we introduced the following notion.

Definition 28 (First Periodic Idle Time [CRGG13])**.** The *First Periodic Idle Time* (FPDIT) of a system is the first DIT occurring after or at $\mathrm{O}_{\max}$.

An example of FPDIT is given in Figure 28.

By definition, if there is a t_d such that t_d is a FPDIT, then t_d and $t_d + \mathrm{H}$ will always be two idle times (as the pattern of arrival and deadlines will

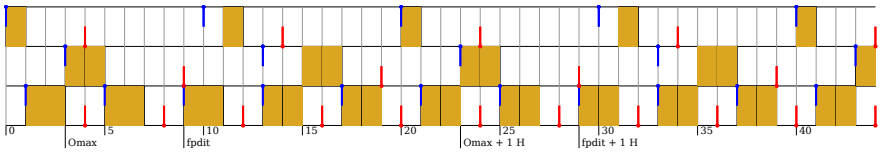

Figure 28: FPDIT of a system. The FPDIT occurs at $t = 9$, which is greater than $\mathrm{O}_{\max} = 3$ and has no jobs which arrived previously with a later deadline.

repeat). By Theorem 5, we will thus have that $[0, t_d + \mathrm{H} + 1]$ is a feasibility interval.

A result presented in [LGBB13] is that the FPDIT always exists for synchronous systems, as in those case $t = \mathrm{H}$ is always a DIT. Because this property holds in the NNP model, we have the following theorem.

Theorem 7. *In the NNP model,* $[0, \mathrm{H}]$ *is a feasibility interval for synchronous systems scheduled by an optimal scheduler.*

In the general case, the existence of the FPDIT is not guaranteed. The determination of the existence and value of the FPDIT is one of the contribution of [CRGG13]. A system has a FPDIT if the following condition was respected.

$$\exists\, t_d \; s.t. \; \forall \tau_i \in \tau, \exists a_i \in [D_i, T_i] \; : \\ \begin{cases} t_d > O_i \\ t_d - O_i \equiv a_i \pmod{T_i} \end{cases}$$

Where $a \equiv b \pmod{n}$ means that a and b are congruent modulo n.

A method to solve this equation and find the a_i values is discussed in [CRGG13]. Another property that is discussed is that if the FPDIT exists, then it occurs before $\mathrm{O}_{\max} + \mathrm{H}$.

In the case where a FPDIT exist, we thus have the following theorem.

Theorem 8. *In the NNP model, for a system* τ *with a FPDIT* t_d*,* $[0, t_d + \mathrm{H}]$ *is a feasibility interval when scheduled by an optimal scheduler.*

3.4.4 Conclusion

The complexity of scheduling with negligible preemption is of exponential complexity, even though EDF (a work-conserving, FJP, context-free scheduler with a known upper bound for the length of a feasibility interval) is known to be optimal.

In the NNP model, we do not know an optimal algorithm, but we know that it is at least idling, in DP and not context-free, and we do not know any upper bound on the length of the feasibility interval in the general case, hinting at a greater complexity.

However, using the notion of Definitive Idle Times (DIT), we can at least give an upper bound on the length of a feasibility interval for some systems, which is of H for all synchronous systems and $t_d + \mathrm{H}$ for systems who contains a FPDIT t_d.

4 Our solutions - heuristic schedulers

In this section, we present two of our own schedulers for the NNP model. In 4.2, we present a scheduler based on LLF which gives more priority to the executing job. In 4.1, we present an approach which never preempts a job unless it detects that it is absolutely necessary. Finally in 5.3, we compare both those approach with state-of-the-art algorithms.

Before we start, note that even if we showed in Section 3.3.1 that both EDF and LLF are not optimal for the NNP preemption model, they are still based on principles inherent to real-time scheduling (jobs with low laxity, or whose deadline is close, must generally be chosen over other tasks). Therefore our proposed solutions are extensions of those algorithms, whose performance is improved thanks to observations made in section 3.

4.1 PMImp

The first approach we present is **PMImp** (Preempion Minimizer - Implicit), which was conceived with the properties of implicit systems in mind, but shows good performance for constrained-deadline systems as well.

4.1.1 Overview

Definition 29 (Cumulative laxity)**.** The **cumulative laxity** of a job $J_{i,j}$ at instant t where it is active is equal to its laxity minus the remaining execution time of all other active jobs with higher priority (at instant t), and minus α if $J_{i,j}$ was preempted.

Expressed with job configurations (definition 23), we have, at time t

$$\ell_{i,j}^{\text{cum}}(t) = \ell_{i,j} - \Big(\sum_{i',j' \ s.t. \ p_{i',j'}(t) > p_{i,j}(t)} C_{i',j'} - \epsilon_{i',j'} \Big) - \text{preemp}_{i,j}$$

The main idea behind PMImp is that once a job has begun executing, we should try our best to not preempt it until completion. In fact, PMImp will only preempt it if the cumulative laxity (using EDF priority to determine which jobs have a higher priority than others) of one of the active jobs reaches 0, which means that if we spend one more unit on the current job, a deadline will be missed.

If we apply this principle, it remains to decide how to chose between several active jobs after an idle instant or a job completion. In that case, we simply decide using EDF priority.

Note that the algorithm is work-conserving and non-clairvoyant. It is thus non-optimal (per Lemma 9 or Lemma 7).

4.1.2 Algorithm

The priority assignment of PMImp is described in Algorithm 1. As PMImp is non-clairvoyant, it is described as a function taking as parameters a job configuration and the configuration of all the other active jobs, and returning the corresponding job priority.

Algorithm 1 PMImp priority assignment

Require: $C^t_{i,j}$: the job configuration that is evaluated
Require: waitingJobs : job configurations of other active jobs
Require: PRIO_MIN : a priority smaller than anything returned by EDF
Require: PRIO_MAX : a priority higher than anything returned by EDF
if $\text{exec}_{i,j} == 1$ **then**
  cumulExecTime $\leftarrow$ 0
  sort waitingJobs by increasing deadline value
  for all $C^t_{i',j'}$ in waitingJobs **do**
    cumulLaxity $\leftarrow \ell_{i',j'}$ - $\text{preemp}_{i',j'}$ - cumulExecTime
    if cumulLaxity $\leq$ 0 **then**
      return PRIO_MIN
    end if
    cumulExecTime $\leftarrow$ cumulExecTime + (C'_i - $\epsilon_{i',j'}$)
  end for
  return PRIO_MAX
else
  return 1 / $e_{i,j}$ {EDF priority}
end if

4.1.3 Example

Consider task system 17.

Task System 17.

	O_i	C_i	D_i	T_i	α
τ_1	4	1	1	18	2
τ_2	3	2	4	18	2
τ_3	2	3	8	18	2
τ_4	1	4	13	18	2
τ_5	0	4	18	18	2

System 17 is an example of system that EDF does not feasibly schedule (as shown in Figure 29). EDF is work-conserving, so it executes $J_{5,1}$ at $t = 0$.

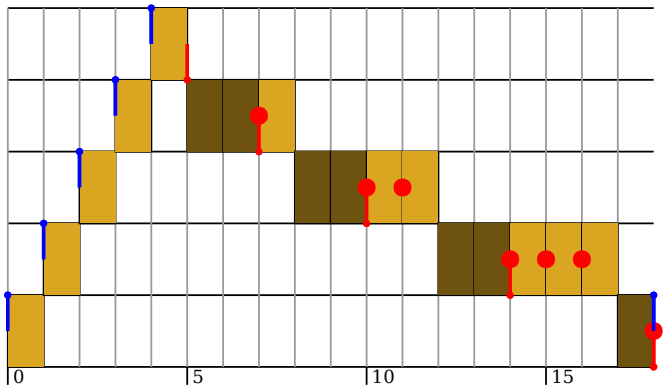

Figure 29: System 17 scheduled by EDF. The system was allowed to continue executing after the first deadline miss (causing further deadline misses)

However, at $t = 1$, the arrival of $J_{4,1}$ preempts the execution of $J_{5,1}$ as its deadline is earlier.

Each subsequent job arrivals creates a preemption, and each forces a preemption recovery period which adds α to the load of the system. This is an extreme example of EDF reacting really badly to the preemption model.

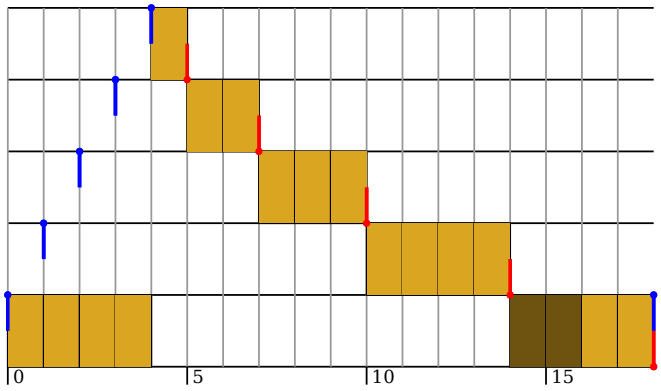

Figure 30: System 17 feasibly scheduled by PMIMP. After $t = 18$, the behavior is periodic

On the other hand, Figure 30 shows that PMImp feasibly schedules system 17. Indeed, $J_{5,1}$ is the only job at $t = 0$ so it is chosen for execution

t	ℓ_1^{cum}	ℓ_2^{cum}	ℓ_3^{cum}	ℓ_4^{cum}	ℓ_5^{cum}
1	∄	∄	∄	9	(executing)
2	∄	∄	5	5	(executing)
3	∄	2	2	2	(executing)
4	(executing)	0	0	0	(preempted)
5	∄	(executing)	2	2	2
6	∄	(executing)	1	1	1
7	∄	∄	(executing)	3	3
9	∄	∄	(executing)	1	1
10	∄	∄	∄	(executing)	4
14	∄	∄	∄	∄	(executing)

Figure 31: The cumulative laxity of the active jobs of system 17 scheduled by PMImp. Here, ℓ_i^{cum} is the cumulative laxity of the active job of task τ_i (if any)

(recall that PMImp is work-conserving).

When $J_{4,1}$, $J_{3,1}$ and $J_{2,1}$ arrive in the system, their cumulative laxity is not enough to preempt $J_{5,1}$ (see Figure 31). However, when $J_{1,1}$ arrives, its initial cumulative laxity is 0, so $J_{5,1}$ is preempted to leave place to the other jobs.

When $J_{1,1}$ finishes, jobs are chosen with EDF priority, so $J_{2,1}$ is selected. Note that $J_{2,1}$ is not preempted because when it starts executing, its remaining execution time is not taken into account in the cumulative laxity of the remaining jobs, which means their cumulative laxity is superior to 0 until $t = 7$.

4.1.4 Non-optimality

Because it is both non-clairvoyant and work-conserving, there were no hope of PMImp being optimal. Indeed, consider system 16 used in the proof of lemma 9. Its schedule by PMImp is shown on Figure 32 to miss a deadline at $t = 12$.

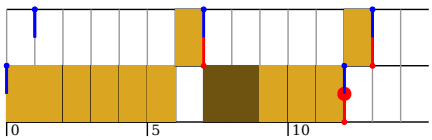

Figure 32: A feasible system infeasibly scheduled by PMImp.

4.1.5 Non-strict dominance of EDF by PMImp

There are systems schedulable by EDF which are not schedulable by PMImp, even if we restrict ourselves to implicit deadline systems. Consider system 18.

Figure 34 shows the system feasibly scheduled by EDF, while 33 shows that the system misses a deadline when scheduled by PMImp.

If we look more closely at the schedules, we see that at $t = 41$, PMImp is actually in a better state that EDF: PMImp did not encounter any preemption and has already finished every active job. EDF, on the other hand, caused a preventable preemption at $t = 32$ and is thus "late" of one time unit compared to the PMImp schedule.

This continues until $t = 69$, where PMImp finishes the current job of τ_2. At that point, because it is work-conserving, it is forced to begin the job of τ_1, which has to be preempted at $t = 72$. Because EDF was late, it finishes the job of τ_2 right on its deadline (which is also the arrival of the next job), it is thus able to begin the execution of the next job of τ_2 immediately, without starting the job of τ_1 which causes a preemption. Once the job of τ_1 is preempted, it is already too late. The system cannot recover from this additional load and finally misses a deadline at $t = 84$.

Task System 18.

	O_i	C_i	D_i	T_i	α
τ_1	0	4	21	21	2
τ_2	7	7	9	9	2

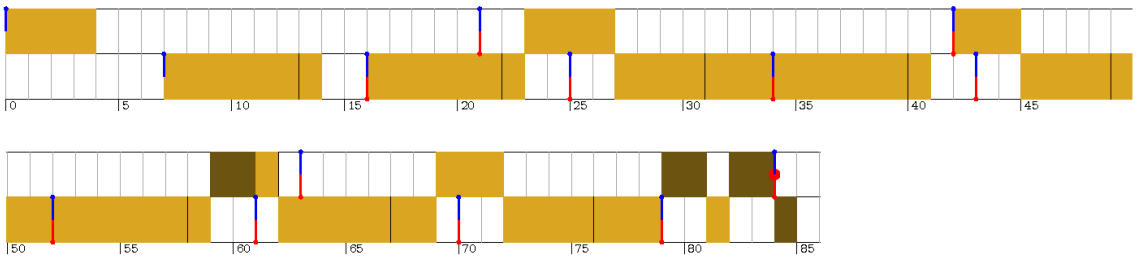

Figure 33: System 18 scheduled by PMImp. The system encounter a deadline miss at $t = 84$

4.2 PALLF

PALLF stands for **Preemption-Aware Least Laxity First**. It is a modification of LLF taking preemption costs into account.

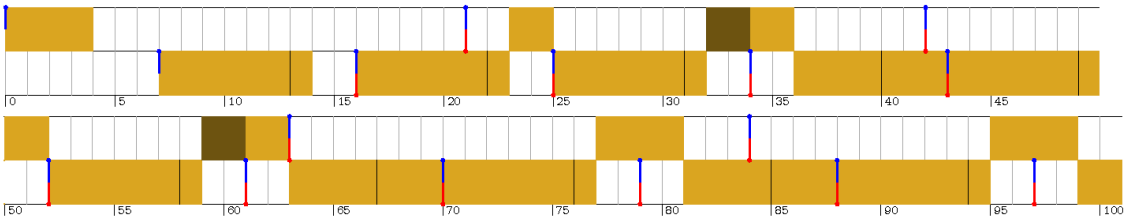

Figure 34: System 18 feasibly scheduled by EDF.

4.2.1 Overview

PALLF is based around three ideas.

The first idea is that once a job is started, it is not fair to compare its laxity to the laxity of other jobs to determine priority. Indeed, the laxity is supposed to quantify how much the job can "afford" not to be executed at the current unit. If a job was executing at the previous unit and is not executing at the current unit, it is preempted. So not executing it at the current unit will actually increase its load of α time units, which should be taken into account when comparing priorities.

The second idea is based on the idling properties discussed in Section 3.3.4, especially theorem 4 which is recalled here.

> If a system has a valid schedule where a job $J_{i,j}$ begins its execution at a time t, then preempted at a time $t' > t$ such that $t' - t < \alpha$, then there exists another valid schedule where the execution is the same before t, but t is an idle unit.

The proof of this theorem showed that not executing $J_{i,j}$ in that case never harmed the schedulability of the system. Therefore if we have the knowledge that starting any active jobs at a time t will result in a preemption in less than α time units, we know it is better to idle.

Finally, the third idea is similar to what we proposed with PMImp: while a job is executing, we do not interrupt it unless another active jobs will miss a deadline if it is not activated immediately.

PALLF is thus idling and clairvoyant. Its horizon is of length α (to determine the idling units). It is thus in the complexity class of an hypothetical optimal scheduling algorithms, but we will see that it is not.

4.2.2 Algorithm

In order to know if starting a job at time t will result in a preemption in less than α time unit, we use Algorithm 2 to determine when a job will be preempted next when scheduling with PALLF. The general idea is to iterate through the next arrivals of the other tasks and compare the expected laxity of the considered job and the initial laxity of the next job of the task.

Algorithm 2 `PALLF.earliestPreemptArrival` method

Require: $C^t_{i,j}$
1: finishTime $\leftarrow t + C_i - \epsilon_{i,j}$
2: candidate $\leftarrow$ NULL
3: **for all** task in the system **do**
4: {Expected priorities (based on laxity)}
5: nextArrival $\leftarrow$ next known arrival of task after t
6: jobExecLeftAtArrival $\leftarrow$ max(0, finishTime - nextArrival)
7: arrivalLax $\leftarrow$ D_i - C_i
8: **if** nextArrival $<$ finishTime **and** arrivalLax $<$ jobExecLeftAtArrival **then**
9: candidate $\leftarrow$ min(nextArrival, candidate)
10: **end if**
11: **end for**
12: **return** candidate

The priority assignment of PALLF is described in Algorithm 3. As it is clairvoyant, it is described as a priority function taking as parameters a job configuration, the configuration of all the other active jobs and a list of future arrivals.

4.2.3 Example

Consider Task System 19.

Task System 19.

	O_i	C_i	D_i	T_i	α
τ_1	0	26	45	45	2
τ_2	22	3	20	45	2
τ_3	2	3	20	45	2
τ_4	12	1	12	45	2
τ_5	0	1	12	45	2

Figure 35 shows Task System 19 scheduled by PALLF. We can observe several interesting behaviours. At $t = 1$, the scheduler includes an idle unit even if there are active jobs ($J_{1,1}$) present. This is because of the arrival of $J_{3,1}$ at

Algorithm 3 PALLF priority assignment

Require: $C^t_{i,j}$: the job configuration that is evaluated
Require: waitingJobs : job configurations of other active jobs
Require: futureArrivals : list of future arrivals and associated jobs
1: lax $\leftarrow \ell_{i,j}$
2: **if** $\text{exec}_{i,j} == 1$ **then**
3: {LLF adjusted to take preemption costs into account}
4: **return** 1 / (lax - α)
5: **else**
6: epa $\leftarrow$ earliestPreemptArrival(job, simu) {see Algorithm 2}
7: **if** epa - t $< \alpha$ **then**
8: **return** PRIO_MIN
9: **end if**
10: **end if**
11: $J_{i',j'} \leftarrow$ job in waitingJobs s.t. $\text{exec}_{i',j'} == 1$
12: **if** $C'_i - \epsilon_{i',j'} \leq$ lax **then**
13: **return** PRIO_MIN
14: **end if**

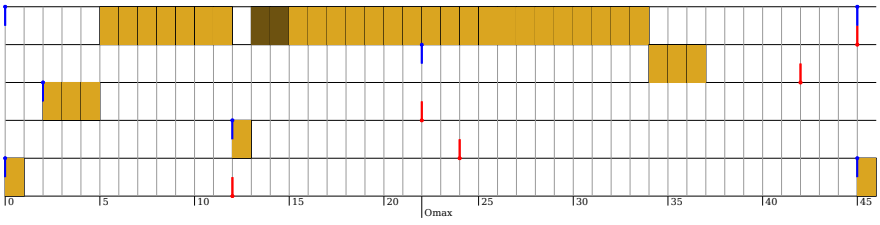

Figure 35: Task System 19 scheduled by PALLF

$t = 2$, which the scheduler detects will cause an immediate preemption as it is impossible for $J_{1,1}$ to execute entirely before the deadline of $J_{3,1}$.

The situation is different at $t = 5$: here, the scheduler is also able to detect that the future arrival of $J_{4,1}$ at $t = 12$ will surely cause a preemption, but it also detects that $J_{1,1}$ may execute for $7 > \alpha$ time units before this becomes a problem. PALLF thus schedule $J_{1,1}$ until $t = 12$, where it is immediately preempted.

Another interesting instant is $t = 22$, when $J_{2,1}$ arrives in the system. It has a earlier deadline than $J_{1,1}$, the executing job (which would cause a preemption with EDF) but the scheduler does not preempt it. Similarly, at $t = 30$, we have that $\ell_{2,1} < \ell 1, 1$ (which would cause a preemption with LLF), but the execution continues because it is possible for both jobs to complete without using a preemption.

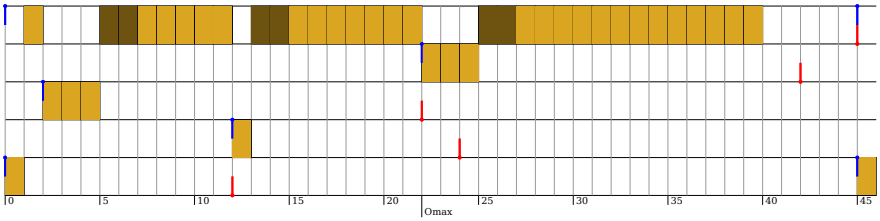

Figure 36: Task System 19 scheduled by EDF

For comparison, Figure 36 shows the same system scheduled by EDF. We see that the number of preemptions is higher. Looking at both schedules side by side, it is clear that the different choices made by PALLF (an idle unit at $t = 1$ and no preemption by $J_{2,1}$) have been entirely beneficial to the system.

4.2.4 Non-optimality of PALLF and dominance by EDF

Consider the following system.

Task System 20.

	O_i	C_i	D_i	T_i	α
τ_1	6	1	1	12	2
τ_2	3	3	6	12	2
τ_3	0	6	12	12	2

EDF feasibly schedules this system, as seen in Figure 37. In fact, the EDF schedule is the only valid schedule of system 20.

However, PALLF does not schedule this system feasibly, as seen in Figure 38. Indeed, because it tries to continue executing $J_{3,1}$ until it is absolutely nec-

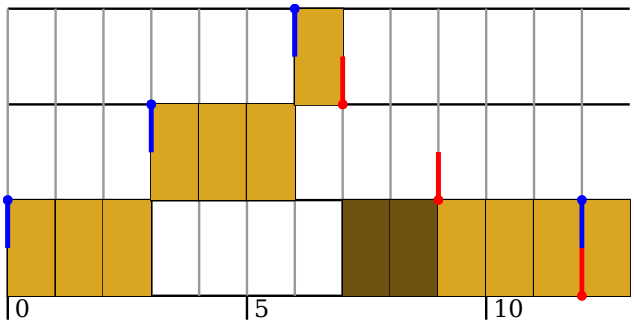

Figure 37: System 20 feasibly scheduled by EDF. Note that this is also the only valid schedule.

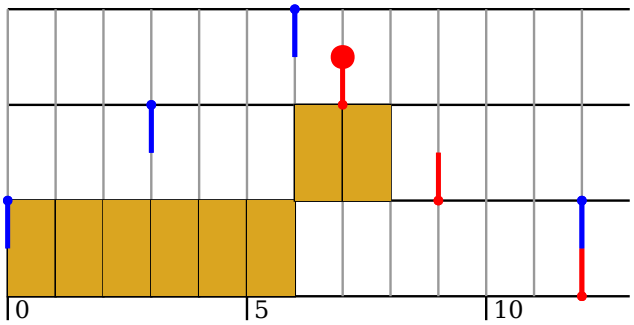

Figure 38: System 20 scheduled by PALLF, with a deadline miss at $t = 7$.

essary to preempt, it is taken by surprise by the unexpected arrival of $J_{1,1}$, at which point it is already too late.

Indeed, PALLF only uses its clairvoyance property to determine if an instant should be idle, not during the execution of an active job. This would be an interesting direction to take in order to extend this scheduler.

4.3 Conclusion

The two proposed approaches are based on trial and error and on the properties described in Section 3.3. They are not optimal, as an optimal scheduling algorithm has been shown in Section 3.4 to be of a greater complexity class. Their performance are studied within a simulator in the next section.

5 Performance evaluation

In order to study real-time systems and the proposed scheduling algorithms within the NNP preemption model, a Python simulator was built from scratch[3] . In this Section, we first explain in Section 5.1 the motivation behind our choice of building our own solution, with a review of the other possibilities. Section 5.2 describes the architecture of our simulator and justify some of the choices that were made.

Finally, Section 5.3 uses the simulator to compare our scheduling algorithms presented in Section 4 to the current state of the art.

5.1 Motivation

5.1.1 Objectives

We wanted to use a simulator to study the feasibility of systems in the NNP preemption model described in Section 3. At that time, we had only reviewed the state of the art and intended to use the simulator to test and confirm the properties presented in Section 3.3. This meant that we needed a simulator with the following features:

- Fast iteration: At this point, our work was highly theoretical and we did not know what results we would find. Therefore we needed to be able to adapt the simulator to results as we found them, to generate random systems, to save interesting ones, to generate clear visual execution logs, etc.
- Permissive license: Because our result were to be published in this master thesis and (hopefully) submitted to experts in the domain, we needed to be able to also submit the (possibly modified) simulator itself for review, which is not permitted by some commercial license.
- Preemption behavior customization: To study the behavior of systems in the preemption model, we needed to be able to implement it. That meant choosing a well-structured and relatively simple open-source simulator, or one with existing preemption features.
- DP Scheduling: Because we suspected that no FJP scheduler would be optimal in our case (as shown later in Lemma 6), we wanted to study DP scheduling algorithms DP and and compare their performances. Therefore we needed a simulator which could handle such schedulers.

[3] At the time of writing, the full source code of the simulator is available at `https://github.com/tchapeaux/py-rt-simulator`.

Note that computational efficiency, while appreciated, was less important than having a clean and modular code.

5.1.2 Review of existing tools

Multiple tools already exist to study the execution of real-time systems. While we chose to build our own in the end, we reviewed several projects before coming to that decision.

Commercial projects such as TimeWiz (TimeSys Corp.) or RapidRM (Tri-Pacific Software Inc.) are mostly closed-source and focused on FTP approaches, which made them unusable for our purpose. They are also focused on the design of one particular system, rather than on the study of feasibility in general.

On the other hand, for academic projects we can cite MAST [HGGM01] which allow to describe systems and study their properties with great details. It is however restrained to FTP schedule. Cheddar [SLNM04] allows both simulation and analysis while being open-source. However, it doesn't seem to have been updated since 2008 and propose a model far more complex that what we intend to study at this point. Finally, the tool from Université Libre de Bruxelles [DVGH96] is similar to Cheddar while being closer to the complexity level we intend to study, but does not seem to be freely available and hasn't been updated since 1996. Other academic tools are too specific or where not build with the intention of being used by others.

5.1.3 Conclusion

After reviewing the existing tools, and because we knew that for this master thesis a relatively simple simulator would suffice, we decided to build our own tool in Python. If we were to extend our work to a more complex task model, we could eventually transfer our work to Cheddar, which already implement various task properties and offer a clean GUI.

5.2 Design of the simulator

5.2.1 Unit vs. Event based

Two paradigms exist in the construction of real-time system simulators:

- Unit-based simulator: where the state at each time unit is computed from the state at the previous time unit, starting from the initial state.

- Event-based simulator: where the simulator keeps a list of future events and handle the events in chronological order. The handling of an event may create more events, and the initial list of events is often the arrival of the first job of each task.

For example, here are some obvious types of event used in an event-based simulator, with the corresponding course of action:

- Job arrival: add the job to the active job list, check if the CPU is idle or if the active job with highest priority preempts the current job. Finally add a new event for the next job arrival.
- Job computation finishes: check that its deadline has not been met yet. Choose the job with highest priority.
- End of preemption recovery period: check if the active job with highest priority preempts the current job.
- Tick: For a dynamic priority scheduler, update the priority of each active jobs. Finally add a new event at the next tick.

While unit-based simulators are conceptually simpler because they follow the natural behavior of systems, event-based simulators are more efficient as they skip redundant time units. For a unit-based simulator, the simulation time of a system can be artificially increased by multiplying the (O, C, D, T, α) values of each task by a constant. This would not affect the execution time of an event-based simulator as the number of events remains the same.

However, event-based simulators do not work well with DP schedulers. Indeed, this is the more general class, with no restriction on the priority of each job at each unit of time. In the worst case, the priority values must be computed at each unit of time, resulting in the same complexity as a unit-based scheduler. This worst case is not an extreme case, as the optimal LLF scheduler modifies the priority of the current job at each non-idle unit of time.

Because Lemma 6 tells us that no FJP schedulers is optimal, we know that the simulator will be run with a majority of DP schedulers. The complexity gain of an event-based scheduler will thus be minimized. For this reason, we choose to implement our simulator with the simpler unit-based paradigm.

5.2.2 Overview

Figure 39 shows the components of the simulator. Upon creation of a `Simulator` object, the user must provide a valid `TaskSystem` and `Scheduler`. A termination time is either provided by the user or set at a default value.

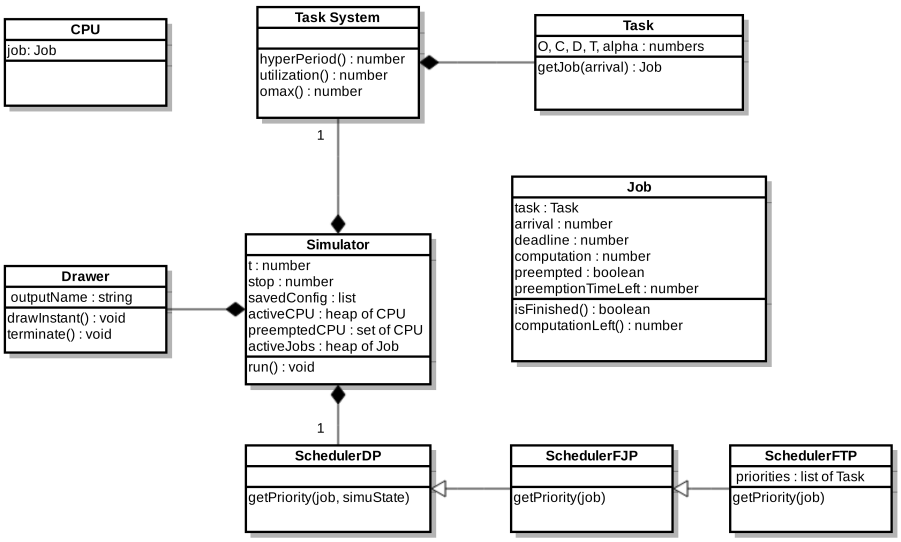

Figure 39: UML Class diagram of the simulator, showing the relationship between most components

The simulation begins when the user calls the method `Simulator.run()`, optionally specifying which termination conditions is or is not enabled. By default, the simulator stop whenever

- The provided termination time is met
- A job deadline is missed
- The execution is stable, i.e. a repeating pattern is detected

After each time unit, the current state is transmitted to the `Reporter`, which generate a report of the execution. Once the simulation is stopped, the method `Reporter.terminate()` is called and an output file is generated.

5.2.3 Data Structures

Figure 40 shows the three basic objects used by the simulator. An external user of the simulator will mostly manipulates `TaskSystem`, while `Simulator` is principally interested in instances of `Job`. `Simulator` generates jobs using the `Task.getJob()` method, passing the current time as a parameter.

Another important object class is the `CPU`. The handling of the CPU may seem unnecessarily complex for uniprocessor scheduling because the simulator is built to also support global scheduling in multiprocessor context (with m processors). Because this master thesis only concentrate on uniprocessor

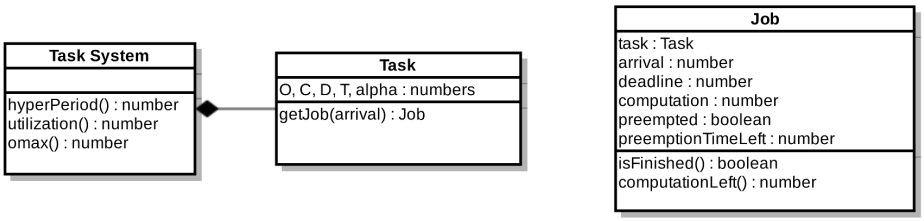

Figure 40: UML Class diagram of the jobs, tasks and tasks systems

contexts, consider $m = 1$. The simulator keeps track of the m CPUs using three data structures linking to the instances of `CPU`:

- `Simulator.CPUs`: a list of the m CPU in a fixed order.
- `Simulator.preemptedCPUs`: a set of the CPUs associated with a recovering jobs
- `activeCPUs`: a heap of CPUs not in `Simulator.preemptedCPUs`, ordered in increasing order according to the priority of the job executing on the CPU (or 0 if the CPU is not executing any job).

The simulator thus keeps two references to each CPU: a constant one in `Simulator.CPUs` and one which will either be in `Simulator.activeCPUs` or in `Simulator.preemptedCPUs`.

During the simulation, all active jobs are stored in either `Simulator.activeJobs`, which is a heap of unfinished jobs not executing (ordered in decreasing order according to job priority), or in the `job` attribute of an instance of `CPU` (which is then available through either `Simulator.activeCPU` or `Simulator.preemptedCPU`.

Concerning systems, instances of `TaskSystem` can be generated from a text file describing the system in the following format described by Figure 41.

```
1    # Example system
2    # Task format: (O, C, D, T, alpha)
3    (1, 2, 3, 4, 0)
4    (3, 2, 6, 10, 0)
5    (0, 1, 4, 10, 0)
```

Figure 41: File format used to describe tasks system

Another possibility is to generate a Task System randomly. The function `generateTasks()` (Algorithm 4) of the module `TaskGenerator` returns a list of `Task` and takes the following parameters:

- `Utot`: the total utilization of the system

- `n`: the number of tasks
- `maxHyperT`: the maximal hyperperiod of the system
- `preemptionCost`: the α value of the system
- `synchronous`: If some offset values may be greater than 0
- `CDF` (Constrained Deadline Factor): Determine the range of permitted values for the task deadline. A CDF of 0 yields an implicit deadline system, while a CDF of 1 means that the deadline of a task may have any value between the execution time and the period.

To generate the utilization values, we use the UUniFast algorithm described in [BB05] (Section 3.6). For the period values, we use the method described in [GM01] which allows the value of H to be bounded by the given value.

Algorithm 4 Function `generateTasks()`

Require: `Utot`, `n`, `maxHyperT`, `preemptionCost`, `synchronous`, `CDF`

1: utilizations = UUniFast(Utot) {[BB05]}
2: divisors = divisors of `maxHyperT`
3: $\tau \leftarrow \{\}$
4: **for all** u in utilizations **do**
5: O = random value (0 if `synchronous` is **true**)
6: T = random choice in divisors
7: C = $\max(1, \lfloor u * T \rfloor)$
8: D = random integer between $T - (T - C) * CDF$ and T
9: α = `preemptionCost`
10: τ.append(Task(O, C, D, T, α))
11: **end for**
12: translate offsets so that the minimal offset is 0
13: **return** τ

5.2.4 Main loop

Let us now review the main loop of the simulation. The simulator is unit-based, which means that at each succeeding unit of time, the following operations are performed in order:

- Check for jobs whose preemption recovery period has ended in `Simulator.preemptedCPUs` and move them to `Simulator.activeCPUs` (see Algorithm 5)
- Remove the busy jobs whose computation has reached 0 of their corresponding CPUs in `Simulator.activeCPU` (see Algorithm 6)

Algorithm 5 Reactivating CPU whose preemption recovery period has ended

Require: preemptedCPUs: set of CPUs with recovering jobs
Require: activeCPUs: heap of the other CPUs
1: **for all** cpu in preemptedCPUs **do**
2: **if** cpu.job.preemptionTimeLeft == 0 **then**
3: preemptedCPUs.remove(cpu)
4: activeCPUs.add(cpu)
5: **end if**
6: **end for**

Algorithm 6 Deleting finished jobs

Require: activeCPUs: a heap of CPU (with executing jobs)
1: **for all** cpu in activeCPUs **do**
2: **if** cpu.job is finished **then**
3: cpu.job $\leftarrow$ NULL
4: **end if**
5: Reorder heap
6: **end for**

- If there are no more active, busy or recovering jobs in the system, increment the idle instant counter.
- At fixed interval of H, save the system state and compare it to previous states. If the current state has been met before, the execution is stable (i.e. repeating). See section 5.2.6.
- Initialize the scheduler (call a function `Scheduler.initInstant()` which does nothing by default, but is used by some schedulers.)
- Check for deadline misses in all jobs of the system. If one is found, add it to the `Simulator.deadlineMisses` list (see Algorithm 7). This never cause the simulation to terminate immediately, as the termination conditions are checked at the end of each iteration of the main loop.

Algorithm 7 Checking for deadline misses

1: **for all** job in active, busy or recovering jobs **do**
2: **if** $t >$ job.deadline **then**
3: deadlineMisses.add((t, job))
4: **end if**
5: **end for**

- Check for job arrivals and add new jobs if there are any (Algorithm 8).
- Compute new priorities if needed (DP schedulers)

Algorithm 8 Checking for job arrivals

Require: activeJobs: heap of active jobs
Require: system: `TaskSystem`
Require: scheduler: `Scheduler`
1: **for all** task in system.tasks **do**
2: **if** $t \geqslant$ task.O **and** ($t\%$ task.T) == (task.O $\%$ task.T) **then**
3: newJob $\leftarrow$ task.getJob(t)
4: newJob.priority = scheduler.getPriority(job, simulator)
5: activeJobs.push(newJob)
6: **end if**
7: **end for**

- Handle preemptions (see section 5.2.5)
- Increment the `completion` counter of the busy jobs
- Decrement the `preemptionTimeLeft` counter of the recovering jobs

5.2.5 Preemption

Simulator based on the assumption that preemption times are negligible are able to simulate preemption in a relatively simple manner. Indeed, scheduling algorithms are often at least FJP and work-conserving, and preempted jobs can be started again with no constraints. In our case however, we consider idling, DP scheduling algorithm and preempted jobs must go through a preemption recovery period when being re-activated.

In our simulator, preemptions are handled at each instant between the arrival of new jobs and the execution of the jobs associated with a CPU. Each job gets a new priority (when the scheduler is DP) and the heap is re-arranged according to these new priority values (see Algorithm 9).

Algorithm 9 New priority values for each jobs

Require: scheduler: `SchedulerDP`
Require: activeJobs: heap of `Job`
Require: activeCPU: heap of `CPU`
1: **for all** job in active or executing jobs **do**
2: job.priority $\leftarrow$ scheduler.getPriority(job, simulator)
3: **end for**
4: Re-order activeJobs
5: Re-order activeCPU

When this is done, we compare the top of `Simulator.activeJobs` (i.e. the active job of highest priority) to the top of `Simulator.activeCPUs`

(i.e. the active CPU of lowest priority) (we do not need to worry about the CPUs of `Simulator.preemptedCPUs` as preemption recovery periods are non-interruptible). If the job of highest priority has a higher priority than the CPU of lowest priority, a preemption occurs as seen in algo 10. Note that at this point, it is possible that the CPU of lowest priority was idle (priority of 0), in which case no job is preempted and no penalty must be applied.

Algorithm 10 Handling a preemption

Require: activeJobs: heap of active jobs, ordered by decreasing priority
Require: activeCPU: heap of active CPUs, ordered by increasing priority

```
highP ← activeJobs.top().priority
lowP ← activeCPUs.top().priority
if highP > lowP then
    {We suppose that priorities are never equals}
    preemptiveJob ← activeJobs.pop()
    preemptedCPU ← activeCPUs.pop()
    preemptedJob ← preemptedCPU.job
    {Push the new CPU in the correct CPU data structure}
    preemptedCPU.job = preemptiveJob
    if preemptiveJob.preempted == true then
        preemptiveJob.preempted ← false
        preemptedCPUs.push(preemptedCPU)
    else
        activeCPUs.push(preemptedCPU)
    end if
    {Put the preempted job back in the active job heap}
    if preemptedJob ≠ NULL then
        preemptedJob.preempted ← true
        preemptedJob.preemptionTimeLeft ← α
        activeJobs.push(preemptedJob)
    end if
end if
```

5.2.6 Termination Condition

Because the systems are periodic, they could theoretically be simulated forever, we thus have to determine when to stop the simulation. Idealistically, the simulation should stop whenever we know for sure if the scheduler will produce a deadline miss or not.

We introduced in section 1.1.4 the notion of feasibility interval (definition 18). We showed that $O_{max} + 2H$ was an upper bound of the feasibility interval for any system scheduled by EDF (or any deterministic scheduler).

This was due to the fact that after that time, either a deadline miss had occurred or the behavior of the system was periodic.

We showed with Lemma 5 that this feasibility interval did not hold for EDF, where some systems did not miss any deadline and did not have a periodic behaviour at $t = O_{max} + 2H$. We do not currently know an upper limit for the length of the transitive period of the execution of a system for any scheduling algorithm.

However, we know that if two identical system configuration occurs at two instant t_1 and t_2 such that $t_2 - t_1 = nH$ (with $n \in \mathbb{Z}_0$), then the behavior is periodic after that point. Because there is a finite number of possible system configuration, this is bound to happen at some point.

Thus, we can save the system configurations at the instants $t = kH$ (with $k = 0, 1, 2, \ldots$). For each new k value, we check if the configuration has been met previously. When this eventually happen, we know the system is stable.

Note that the number of possible configuration is astronomic. Indeed, we can find a pessimistic upper bound by considering that for each task, we have at most one job configuration which can take the following value:

- $\theta_{i,j}$ (number of time units since the job arrival) : T_i possible values
- $\epsilon_{i,j}$ (number of executed time units) : C_i possible values
- $P_{i,j}$ (number of PRP time units) : α possible values

We thus have the following upper bound on the number of possible different system configurations:

$$\prod_{i=1}^{n} T_i C_i \alpha$$

However in practice, with a sensible scheduling algorithm, the stabilization often occurs in a sensible time ($t < O_{max} + 10H$). For a great majority of systems, it occurs at the second possible time, $O_{max} + 2H$.

5.2.7 Schedulers Implementation

The organization of schedulers within the project and some of the provided schedulers are shown in Figure 42.

Schedulers must inherit from the abstract `SchedulerDP` class. As we saw before, the method `Scheduler.priority` is called for each job at each in-

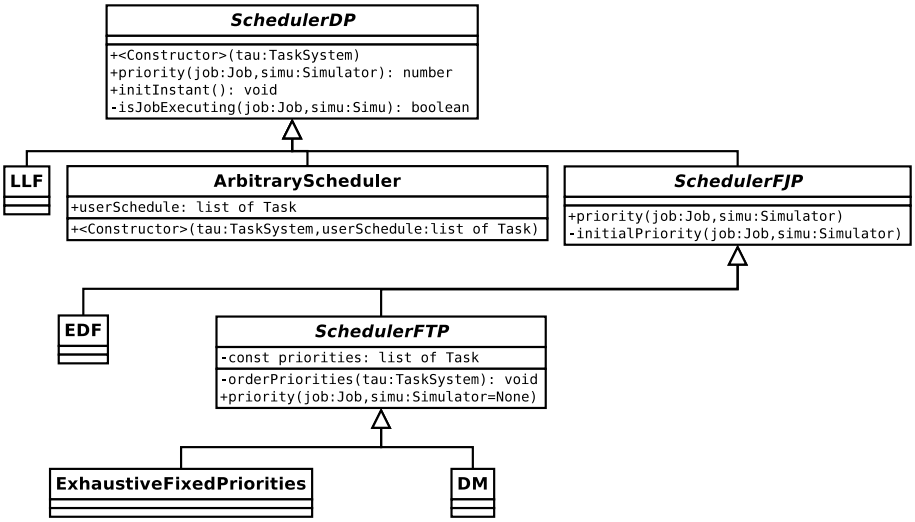

Figure 42: Class diagram of the different scheduler classes

stant. This is the method that new schedulers must define to work within the project.

As no restrictions are imposed on the schedulers, they are free to define new attributes and use any information provided by the `simu` parameters of their `priority` method.

Because we are in a context where preemptions costs must be taken into account, schedulers have access to a `isJobExecuting` method, which allow schedulers to know if their priority assignment will cause a preemption. Note that this information is part of the job configuration (Definition 23), and is thus available to context-free schedulers.

The provided `ArbitraryScheduler` class allow users to provide their own schedule for a given system, in the form of the `userSchedule` parameter. This parameter is a list of reference to tasks of the system. The i^{th} element must be the task whose earlier job is to be executed at $t = i$. Because the schedule is infinite and the list has a finite size, we suppose the provided schedule to be periodic.

Algorithm 11 shows how `ArbitraryScheduler` assign priorities.

To simulate FJP schedulers on top of DP schedulers, we use the abstract `SchedulerFJP` class. This class redefine the `priority` method as shown in Algorithm 12. New FJP schedulers must thus define their priority assignment in the `initialPriority` method, which will only be used at the job arrival.

FTP Schedulers are implemented using the abstract `SchedulerFTP` class.

Algorithm 11 `ArbitraryScheduler.priority` method

Require: job: the job that is evaluated
Require: t: the time that is evaluated
Require: userSchedule: the provided schedule
1: t $\leftarrow$ t $\%$ (length of userSchedule)
2: **return** userSchedule[t]

Algorithm 12 `SchedulerFJP.priority` method

Require: job : the job that is evaluated
Require: simu : the state of the system
1: {If a job was evaluated previously, its `priority` argument will be set}
2: **if** job.priority is not NULL **then**
3: **return** job.priority
4: **else**
5: **return** initialPriority(job, simu)
6: **end if**

This class forces the scheduler to compute its priority assignment at startup, then use the computed assignment at runtime. To do so, the default constructor calls the `orderPriorities` abstract method (and thus defined by children classes), which return a sorted list of the tasks of the system.

This sorted list is stored in the `priorities` attribute, which is used by default `priority` function at runtime (as shown in Algorithm 13). Note that the lowest priority is 1, as 0 is reserved for idling scheduler (and every FTP scheduler is work-conserving by definition).

Algorithm 13 `SchedulerFTP.priority` method

Require: job : the job that is evaluated
1: jobTask $\leftarrow$ job.task
2: i $= 1$
3: **for all** task in self.priority **do**
4: **if** task $==$ jobTask **then**
5: **return** i
6: **end if**
7: i $\leftarrow$ i + 1
8: **end for**

Finally, both PMImp (Section 4.1) and PALLF (Section 4.2) are implemented in the simulator. Algorithm 14 describes the `priority` method of PMIMP, and Algorithm 15 describes the `priority` method of PALLF.

Algorithm 14 `PMImp.priority` method

Require: job : the job that is evaluated
Require: simu : the state of the simulation
if isJobExecuting(job) **then**
  waitingJobs ← simu.getActiveJobs()
  sort waitingJobs by increasing deadline value
  cumulExecTime ← 0
  for all waitingJ in waitingJobs **do**
    cumulLaxity ← laxity(waitingJ) - cumulExecTime
    if job was preempted **then**
      cumulLaxity ← cumulLaxity - α
    end if
    if cumulLaxity $\leq$ 0 **then**
      return 0 {Always lower than EDF priority}
    end if
    cumulExecTime ← cumulExecTime + waitingJ.computationLeft()
  end for
  return 1 {Always higher than EDF priority}
else
  return 1 / job.deadline
end if

Algorithm 15 `PALLF.priority` method

Require: job : the job that is evaluated
Require: simu : the state of the simulation
lax ← laxity of job
if isJobExecuting(job) **then**
  {LLF adjusted to take preemption costs into account}
  return 1 / (lax - α)
else
  epa ← earliestPreemptArrival(job, simu) {see Algorithm 2}
  if epa - t $< \alpha$ **then**
    return 0
  end if
end if
busyJ ← executing job
if busyJ.computationLeft $\leq$ lax **then**
  return 0
end if

5.2.8 The Reporter module

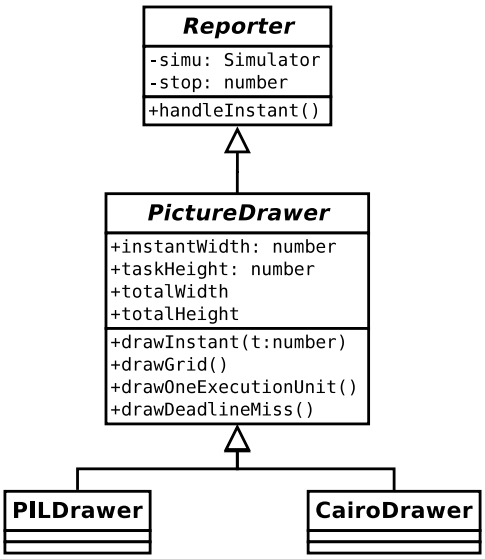

Figure 43: Class diagram of the reporter classes

The Reporter class (Figure 43) is called after each step in the simulation and is used to display reports of the execution. New classes may inherits from the `Reporter` class, whose method `Reporter.handleInstant` is called after each iteration of the main loop.

`PictureDrawer` is an abstract reporter used to display a visual representation of the execution, such as those presented in this master thesis. It is implemented by two different classes, `PILDrawer` and `CairoDrawer`, which use different drawing libraries.

5.3 Comparison with state-of-the-art

Both PMImp and PALLF were tested on randomly generated tasks system against EDF and LLF, both optimal algorithms when preemptions times are not taken into account.

Systems were generated using the `generateTask` function of the simulator described in Algorithm 4, with the following parameters.

- `Utot`: Between 0.1 and 1 (see graphs)
- `n`: chosen uniformly between 2 and 10
- `maxHyperT`: 6300

- `preemptionCost`: 2
- `synchronous`: `False`
- `CDF` : 0 for implicit systems, 1 otherwise
- α : 2

Each point of the graphs is the percentage of feasible systems out of 1000 systems with the given total utilization.

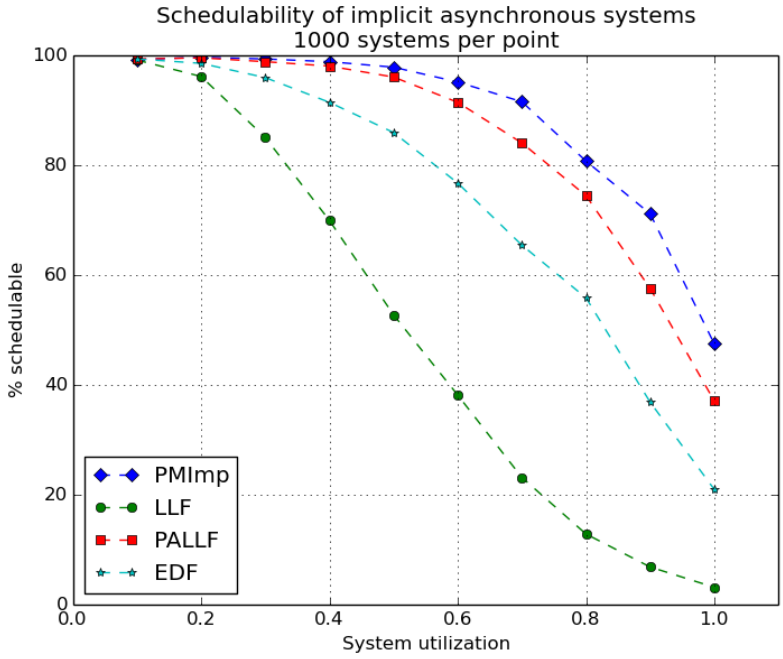

Figure 44: Comparison of schedulability of constrained implicit systems

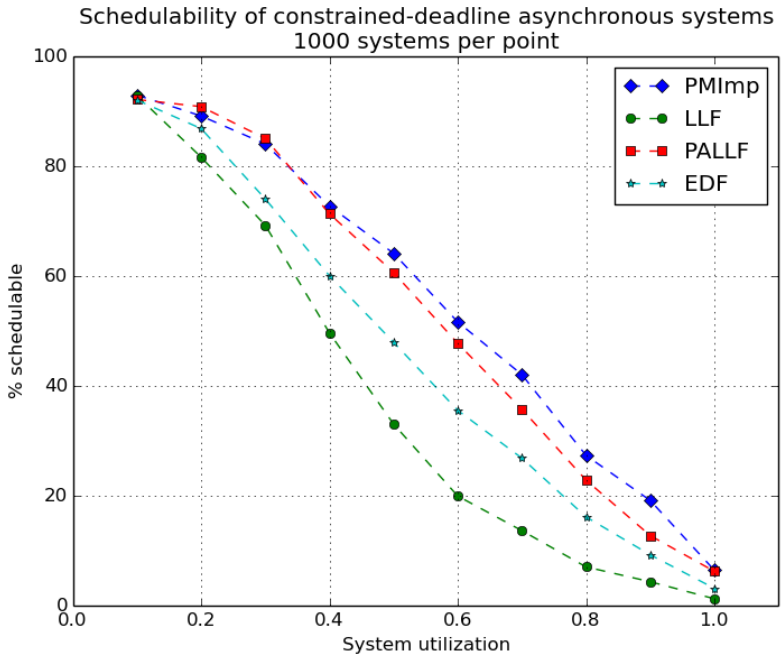

Figure 45: Comparison of schedulability of constrained deadline systems

6 Conclusion

6.1 Review of the objectives

In Section 1.2, we defined multiple objectives for the master thesis. We will now review each of them and discuss our results.

6.1.1 State of the Art

The first objective was to review the state of the art on the subject. We presented many methods, mostly studied in the context of offline (FTP) scheduling.

Of those methods, non-preemptive scheduling gave the best predictability with regard to the preemption costs. Indeed, as they were non-existent, the preemption costs could easily be assumed negligible without any negative impact. However, this predictability comes at a great feasibility cost as most systems are not non-preemptive-schedulable.

Another method with great predictability with regard to the preemption costs was Carousel. With Carousel, the cache-related preemptions costs are

constant for each job and can thus be integrated into the execution time. The flaws of Carousel were the small overhead causing some resource waste, and the fact that it requires system designer to know which portion of the cache each task is going to use.

The method which seemed to be the most frequently used in practice was to artificially increase the WCET values by some arbitrary value. We discussed why this was a really rough estimation and probably lead to a great amount of resource waste.

Finally, we presented the NNP preemption model which directly integrated the preemption costs into the scheduling model, and a previous result giving an optimal algorithm for FTP scheduling.

6.1.2 Optimal online scheduling algorithm

We wanted to propose an optimal online scheduling algorithm for the NNP model. This turned out to be more complex than expected.

Indeed, unlike the well-known model with negligible preemptions, an optimal scheduler for the NNP model needed to be in (TODO FIXME: on doit pouvoir lire intro + conclusion et comprendre) DP, use idling units, and be clairvoyant. The complexity of the scheduling in the NNP model was discussed, with a trivial proof of NP-hardness in the general case (for all values of α, even ridiculously high ones).

Instead of an optimal online scheduling algorithm, we proposed two heuristic algorithms with good performance against EDF and LLF, which are both optimal when the preemptions costs are negligible.

6.1.3 Simulator

The last objective was to be able to validate our results with a simulator. Because of the lack of solutions integrating the NNP model, we build our own simulator and implemented our heuristic algorithms (as well as others).

The fact that we had written the simulator ourselves allowed us to be very familiar with the source code and adapt it very easily when necessary. This proved invaluable during the theoretical part of our research (as hypotheses could be quickly implemented and tested against real data) and during the redaction of this master thesis (as some figures needed special annotations, unimplemented schedules or other corner cases).

We intend to continue to improve the simulator and release its code with an open-source license. At the time of writing its `git` repository is available on

GitHub[4].

6.2 Contributions

The main contributions of this master thesis were

- A list of significant properties of scheduling in the NNP model.

  In particular:

  - The non-optimality (Definition 12) and non-sustainability (Definition 20) of EDF and LLF
  - The fact that an optimal online schedulers had to be idling, clairvoyant and in DP.
  - An incomplete (but promising) description of the idling patterns of an optimal online scheduler
- The implementation and description of our simulator for the NNP model.
- The description and empirical analysis of PMImp and PALLF, two heuristic scheduling algorithms for the NNP model. Both have been shown to have a success ratio at least 10% higher than that of EDF or LLF.

Furthermore, during our research, we published and presented a paper ([CRGG13]) which presented a method to guarantee the existence of a feasibility interval (Definition 18) for systems with a FPIT (Definition 28), which gives an upper bound on the time complexity of the scheduling problem.

We also plan to finalize and submit this summer another research paper on feasibility intervals for EDF in the NNP model. The invalidity of the current bound will be discussed, and the value of the actual bound will be presented as an open problem.

6.3 Future works

This master thesis presented the first results for online scheduling in the NNP model. This work can be extended along multiple paths.

The obvious candidate is to extend this work to multiprocessor scheduling. In that case, we would have to also take into account migration costs (when a job is interrupted on a processor then reactivated on another). Even with

[4] `https://github.com/tchapeaux/py-rt-simulator`

negligible preemption costs, multiprocessor scheduling analysis is still being actively researched today and some big open question remains, so we should expect only partial results in that domain.

During our research, we encountered a number of open questions which could also be the basis for future works. The following list contains the most significant ones in our opinion.

- Is there an upper bound for the length of a minimal feasibility interval for EDF or LLF scheduling in the NNP?
- Is there an upper bound for the horizon of an optimal scheduling algorithm for the NNP model?
- Does Theorem 6 (Existence of cyclic idle times) holds for the NNP model?

We are currently in the process of applying for a PhD thesis during which these questions will be studied.

List of definitions

List of Figures

References

[ATB93] Neil Audsley, Ken Tindell, and Alan Burns. The end of the line for static cyclic scheduling. In *Proceedings of the 5th Euromicro Workshop on Real-time Systems*, pages 36–41, 1993.

[Aud91] Neil C Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. 1991. Technical report.

[BA06] Mordechai Ben-Ari. *Principles of concurrent and distributed programming*. Addison-Wesley Longman, 2006.

[BB05] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[BB06] Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 159–168. IEEE, 2006.

[BBY13] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *Industrial Informatics, IEEE Transactions on*, 9(1):3–15, 2013.

[BCSM08] Bach D Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 101–110. IEEE, 2008.

[BG04] Sanjoy Baruah and Joël Goossens. Scheduling real-time tasks: Algorithms and complexity. *Handbook of scheduling: Algorithms, models, and performance analysis*, 3, 2004.

[CGG04] Annie Choquet-Geniet and Emmanuel Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theoretical computer science*, 310(1):117–134, 2004.

[CRGG13] Thomas Chapeaux, Paul Rodriguez, Laurent George, and Joël Goossens. The space of feasible execution times for asynchronous periodic task systems using definitive idle times. In *Brazilian Symposium on Computing Systems Engineering*, 2013.

[DVGH96] Stephane De Vroey, Joël Goossens, and Christian Hernalsteen. A generic simulator of real-time scheduling algorithms. In

Simulation Symposium, 1996., Proceedings of the 29th Annual, pages 242–249. IEEE, 1996.

[GJ79] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[GM01] Joël Goossens and Christophe Macq. Limitation of the hyper-period in real-time periodic task set generation. In *Proceedings of the RTS Embedded System (RTS'01*, 2001.

[GMR+95] Laurent George, Paul Muhlethaler, Nicolas Rivierre, et al. Optimality and non-preemptive real-time scheduling revisited. 1995. Research report.

[Goo99] Joël Goossens. *Scheduling of hard real-time periodic systems with various kinds of deadline and offset constraints*. PhD thesis, Université Libre de Bruxelles, 1999.

[GR13] Joël Goossens and Pascal Richard. Ordonnancement temps réel multiprocesseur. *État de l'art—ETR*, 2013, 2013.

[HGGM01] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and J. M. Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE, 2001.

[JSM91] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

[KM83] Aloysius Ka and Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*, volume 1. MIT Thesis, May, 1983.

[LGBB13] Giuseppe Lipari, Laurent George, Enrico Bini, and Marko Bertogna. On the average complexity of the processor demand analysis for earliest deadline scheduling. In *Proceedings of a conference organized in celebration of Professor Alan Burns's sixtieth birthday*, page 75, 2013.

[Liu00] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[LL73] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

[MYS07] Patrick Meumeu Yomsi and Yves Sorel. Extending Rate Monotonic Analysis with Exact Cost of Preemptions for Hard Real-Time Systems. In *Proceedings of 19th Euromicro Conference on Real-Time Systems, ECRTS'07*, 2007.

[Ndo14] Falou Ndoye. *Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation*. PhD thesis, Université Paris Sud-Paris XI, 2014.

[RP07] Rakesh Reddy and Peter Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 198–207. ACM, 2007.

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM, 2004.

[SW00] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 25–34. IEEE, 2000.

[TDP13] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Quantifying the sub-optimality of non-preemptive real-time scheduling. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 113–122. IEEE, 2013.

[WA12] Jack Whitham and Neil C Audsley. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 3–12. IEEE, 2012.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[YBB09] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Embedded and Real-Time Computing Sys-*

tems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on, pages 351–360. IEEE, 2009.

[YSA94] Xiaoping George Yuan, Manas C Saksena, and Ashok K Agrawala. A decomposition approach to non-preemptive real-time scheduling. *Real-Time Systems*, 6(1):7–35, 1994.