MEMO-F-508 : MASTER THESIS

Integrating preemption costs in the real-time uniprocessor schedulability analysis

Author:
Thomas CHAPEAUX

Supervisor:
Joël GOOSSENS

Academic Year 2013-2014

Abstract

In this document, we present an overview of the analysis of real-time systems as periodic and sporadic task sets in the uniprocessor case. Then, we point out the oversimplification that is made in considering the preemption times to be negligible and explain why this might cause unpredictability. We thus present some of the methods that could be used to remove this unpredictability. Some methods that are presented reduce the importance of the problem by reducing the impact of additional preemption costs, while others define a new model to be able to take that cost into account. Finally, we point out some open questions that could serve as basis for our master thesis.

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 timeframe.

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.

In this section, We first present in 1.1 the model that we use to study hard real-time systems. We show how the model allows us to determine feasibility of those systems under different assumptions, such as negligible preemption times. Then we try to remove this assumption in two manners : first, in 2, we present approaches to reduce the impact of preemptions. Then, in 3, we allow costly preemption but we try to measure its effect on feasibility.

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.

A system is thus a set $\tau = \{\tau_1, \cdots, \tau_n\}$, with each τ_i representing a task 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.

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*).

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.

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**.

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**. In the synchronous case we can assume $O_i = 0 \, \forall i$ without loss of generality.

We also define the following notions:

Definition 1 (Task Utilization) *For a given task τ_i, we define its utilization*

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

Definition 2 (System Utilization) *For a given system, its utilization U_{tot} is given by*

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

1.1.2 Worst-Case Execution Time

This model assumes a constant execution time for every job of the same task (C_i). In practice, it often depends on the input and thus is different for each job. A classic approach is thus to set the C_i value of a task to an upper bound equal to its longest known execution time, called the **worst-case execution time (WCET)**. In later sections, we sometimes use WCET to refer to C_i.

This approach relies on the idea that if a system is feasible for some pessimistic C_i values, any similar system whose jobs have lower execution times will stay feasible[1].

In our model, C_i is the number of time units spent on a job necessary to complete it. This is different from its *response time*, which is the time it needs

[1] While this is true under our assumptions, it should be noted that this is not always true (e.g. multiprocessor model or non-preemptive scheduling)

to complete when considering the other jobs in the system. This means that the value of C_i, and the WCET, must be calculated by considering the job alone in the system.

1.1.3 Scheduling model

Once the system is defined as a task set, we may define a mathematical representation of the execution and scheduling of the system.

We are concerned with the uniprocessor case, 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 summary of the schedulability results for the multiprocessor case can be found in [GR13].

For a given task set, similarly as in [Goo99], we define one of its **schedule** as a function $\sigma(t) : \mathbb{N} \to \mathbb{N}^2$ such that $\sigma(t)$ gives us the (i, j) identifier of the job executing at time t (or $(0, 0)$ for times at which no job is executing).

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

Definition 3 (Feasibility) *A task set is said to be* feasible *if it exists a schedule such that every job in the task set completes before its deadline is reached.*

We can derive a necessary condition of feasibility using the system utilization (Definition 2). Indeed, the utilization of a task is the fraction of time necessary for the processor to complete every job of this task. It thus follows that

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

is a necessary condition of feasibility in the uniprocessor case.

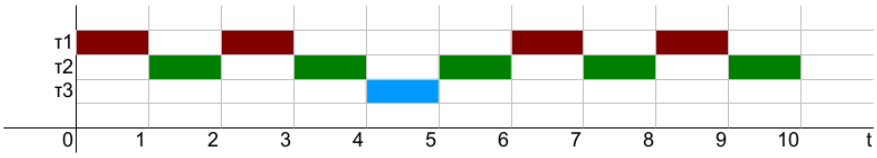

Figure 1: A feasible task set

For example, the following task set is feasible. We can convince ourselves of this by looking at a schedule in which all deadlines are met (see Fig. 1)

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

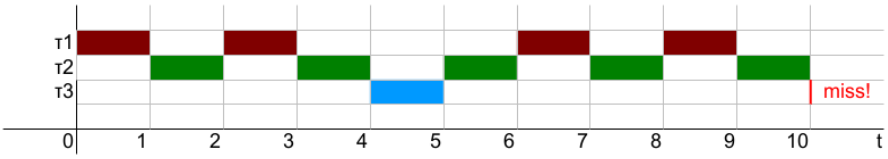

Figure 2: An unfeasible task set

And the following task set is unfeasible (Fig. 2 gives an example of a schedule where a deadline is not met). We can convince ourselves that no schedule will respect every deadline by observing that it does not satisfy the necessary condition ($\mathrm{U}_{tot} = \frac{2}{5} + \frac{1}{2} + \frac{2}{10} > 1$).

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

The notion of feasibility only caracterizes 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:

Schedulability (by a scheduling algorithm) : A task set is said to be *schedulable* by a scheduling algorithm if the algorithm gives a schedule of the system such that it does not miss any deadline.

Optimal scheduling algorithm : A scheduling algorithm is said to be *optimal* if every feasible task set is schedulable by this algorithm.

If it exists an algorithm such that a task set is schedulable by this algorithm, the task set is feasible (by definition). Moreover, If we know an optimal scheduling algorithm, it is sufficient to determine the schedulability of a task set by this optimal scheduling algorithm to determine its feasibility.

1.1.4 Examples of scheduling algorithms

In this section, we assume the following properties of the scheduling algorithms:

- *Non-idling* (or *work-conserving*) : The schedule produced by the algorithm cannot contains an idle unit (a unit of time t such that $\sigma(t) = (0, 0)$) if there are active jobs at this time.
- *Preemptive* : The schedule is allowed to use preemptions, i.e. to pause the execution of a job to execute another job in its place. For now, we assume that a preemption can occur instantaneously between two time units, i.e.

that the preemption time is negligible w.r.t. a time unit. A model where this assumption is not made is described in later sections.

Scheduling algorithms have to decide for each instant which of the active jobs will be executing. This is done by comparing the **priority** of each job at each instant and executing the one with the highest priority. Each scheduling algorithm will have its own definition of priority for a given job[2], but we can distinguish between three classes :

- **Fixed Task Priority** (FTP) : Each task τ_i is affected a constant priority p_i based on its parameters. Jobs priorities are equal to the priority of their task.
- **Fixed Job Priority** (FJP) : Each job $J_{i,j}$ is affected a constant priority $p_{i,j}$ based on its parameters.
- **Unconstrained Dynamic Priority** (DP) : The most general case. Priorities of jobs are not constant.

Note that $FTP \in FJP \in DP$. 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 an important overhead.

Here are some examples of well-known scheduling algorithms:

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

$$p_i = \frac{1}{D_i}$$

It can be shown that DM is *FTP-optimal*[3] for synchronous constrained deadline systems.

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

$$p_{i,j} = \frac{1}{d_{i,j}}$$

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

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

$$p_{i,j,t} = \frac{1}{\ell_{i,j}(t)}$$

[2] In order to be deterministic, algorithms must have a deterministic way of handling cases where two jobs have the exact same probability. Often, we trivially choose the job whose task has the lowest identifier i. When two jobs of the same task have the same priority, we choose the job with the lowest execution time.

[3] that is, it schedules all task set that are feasible by a FTP algorithm

where $\ell_{i,j}(t) = d_{i,j} - t - (e - \epsilon_{i,j}(t))$ 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 task set in the uniprocessor case.

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

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

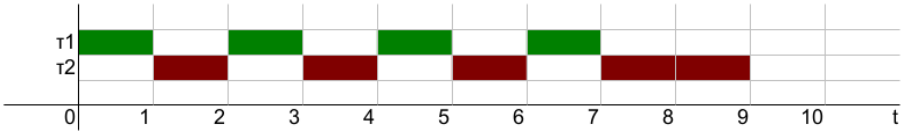

Figure 3: A task system scheduled by LLF

As we can see in Fig. 3, the schedule will use 6 preemptions every 10 units of time. In our model, we consider preemption times to be negligible, but in a real application the constant context switch would delay the schedule and may cause a deadline miss.

1.1.5 Optimal algorithms and feasibiliy conditions

Optimal algorithms allow us to be sure that any feasible system will be scheduled correctly. In this section, we will look at sufficient conditions allowing us to know if a system is feasible.

To this aim, we introduce the following notions:

Definition 4 (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 5 (Critical instant of a task) *the* critical instant *of a task τ_i is the time of arrival of a job of τ_i with the maximal response time.*

It then follows that if the response time of the job arriving at the critical instant of τ_i is $\leq D_i \forall i$, the system is FTP-feasible (schedulable by DM). For synchronous constrained tasks, [LL73] have shown that the critical instant of all tasks is equal to the arrival of their first job ($t = 0$). [Aud91] gives us a formula to find the response time of each task in that case, as

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

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 analyze the feasibility of such systems, we use the notion of feasibility intervals:

Definition 6 (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 7) 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 7 (Idle point) *An* idle point *is a time unit at which all jobs which arrived strictly before it are completed.*

For asynchronous task set, [LW82] have shown that $[O_{max}, O_{max} + 2P]$ is a feasibility interval, where $O_{max} = max\{O_i\}$ and $P = lcm\{T_i\}$. 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 $\mathrm{U}_{tot} \leq 1$ 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 $[O_{max}, O_{max} + 2P]$.

As EDF is optimal, we do not need to study any DP algorithms.

1.1.6 Negligibility of preemption cost

As we saw in 1.1.4, previous results for preemptive schedulers assume that the duration of a preemption, or *preemption cost*, is negligible w.r.t. 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 [BBY09] distinguish between the following four types of preemption costs [4] :

[4] 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.

- *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.

[BCSM08] has shown that cache-related costs alone can 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 correctness of our feasibility predictions.

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 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 techniques where we continue to assume that preemptions times are negligible, but we modify the model so that this assumption is closer to reality.

2.1 Surestimation of the WCET

A way of integrating the preemption cost without changing our model is to artificially boost the WCETs by a fixed value (often between 20 and 30 %). A more refined approach would be to apply a multiplicative factor to the WCET that is inversely proportional to the priority of the task. However, this is a really rough approach, as the number of preemptions is not constant for each job of the same task. This approach must thus consider a pessimistic worst-case, which leads to resources waste.

Moreover, as we saw in section 1.1.2, 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 the following) are preferred.

2.2 Non-preemptive scheduling

An extreme approach to avoid the negligible preemption cost assumption is to absolutely forbid preemptions. This cause less systems to be feasible but allows us to not drastically change our model and to get rid of preemption costs entirely (i.e. we trade feasibility for more predictability). To illustrate this, consider the following system :

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

This system is schedulable by a preemptive version of EDF (see Figure 4), but is not schedulable by any non-preemptive scheduling algorithm. This example shows that there exist preemptive-feasible systems which are not non-preemptive-feasible. However, as non-preemptive algorithms are a subclass of

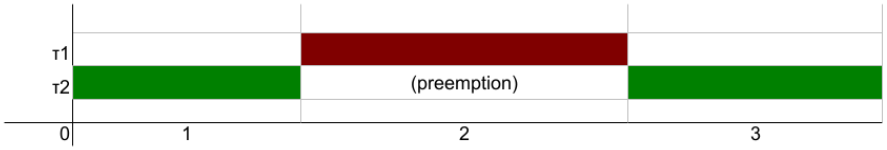

Figure 4: A task set which require preemptions to be feasible

preemptive algorithms, every non-preemptive-feasible system is preemptive-feasible. To summarize, non-preemptive-feasible $\subset$ preemptive-feasible (strict).

The remaining of this section quantifies the cost of the non-preemptive approach with regards to the overall feasibility of systems, based on [GMR+95] and [BBY09].

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 schedule to not schedule 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 times. Moreover, this assumption allowed us to simplify the analysis and helped to prove some results.

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

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

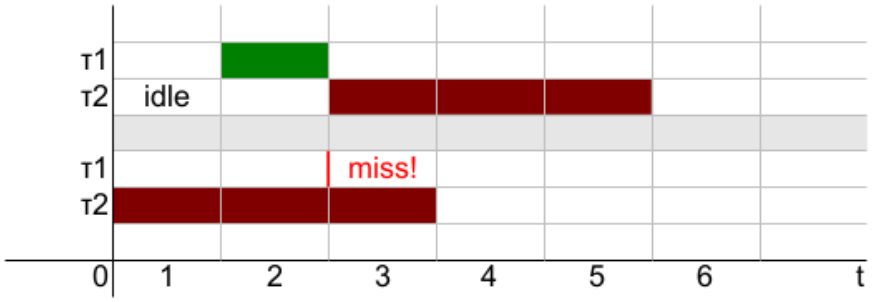

Figure 5: Above, a feasible schedule with an idling unit. Below : an optimal non-idling non-preemptive schedule (e.g. NINP-EDF) missing a deadline

As we can see in Fig. 5, 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 $t = 3$, which is exactly when $J_{1,1}$, the first job of τ_1, will miss 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

In this case, [GMR$^+$95] shows that NINP-EDF (*Non-Idling Non-Preemptive EDF*) is optimal (by showing that every valid schedule can be re-ordered into a NINP-EDF schedule). 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, $[0, O_{max} + 2 \times P]$ with the added condition that an idle time must occur in the interval $[O_{max}, O_{max} + 2 \times P]$ (with O_{max} being the maximal O_i and P the LCM of the T_i).

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 [JIS79] to be NP-complete, and indeed an exhaustive search of a valid schedule asks to consider $O(n!)$ 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.

2.3 Limited preemptive algorithm

As we saw, non-preemptive algorithms remove the problem of preemption times. However, they also prevent us from feasibly scheduling some systems. Limited preemptive algorithms are a compromises between fully preemptive and non-preemptive algorithms which greatly reduce the number of preemptions (allowing us to study the execution of the system more easily) and which are able to schedule systems which are not non-preemptive-feasible.

The authors of [BBY09] distinguish between three types of limited preemptive algorithms

- **Preemption thresholds scheduling (PTS)** Each task τ_i is assigned a priority p_i (as in FTP scheduler) and a **preemption threshold** θ_i. Only jobs of task τ_k such that $p_k > \theta_i$ are allowed to preempt jobs of τ_i.
- **Deferred preemption scheduling (DPS)** Each task τ_i is assigned a no-preemption interval q_i, during which it cannot be preempted. The authors distinguish between two implementation variant of DPS : in the **floating model**, non-preemptive sections are activated by system calls in the task code, and in **activation-triggered model**, those sections are activated by the arrival of higher priority tasks.
- **Fixed Preemption Points (FPP)** 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.

We will now look at the feasibility of each of those approaches, as given in [BBY09]. It should be noted that the authors only consider FTP scheduling.

In FTP non-preemptive scheduling, we define the following notion:

Definition 8 (Blocking time of a task) *The* blocking time *of a task, defined as the maximal time a job of the task can be delayed because of lower-priority tasks. 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). That is,*

$$B_i = \max_{j:P_j<P_i} \{C_j - 1\}$$

(The blocking time of the lowest-priority task is set to 0).

2.3.1 Preemption Thresholds Scheduling (PTS)

First, we can observe that if $\theta_i = p_i \, \forall i$, the scheduling is equivalent to a fully preemptive one. Similarly, if $\theta_i = p_k \, \forall i$, with 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.

The blocking time of each task thus becomes

$$B_i = \max_j \{C_j - 1 \mid P_j < P_i \leq \theta_j\}$$

The authors then use this blocking time and previous results to obtain a formula giving the worst-case response-time of each task, which can be checked against the corresponding deadline.

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 interrupted. The authors only give a feasibility condition for the floating model, where the non-preemptive tasks are activated by system calls within the task, it follows that the maximal blocking time of each task is given by

$$B_i = \max_{j:P_j<P_i}\{q_j - 1\}$$

and we can use the worst response time for our feasibility test

$$R_i = B_i + \sum_{h:P_h>P_i} \lceil \frac{R_i}{T_h} \rceil C_h$$

In the activation-triggered model, non-preemptive sections begin when the job should have been preempted by a higher-priority job. This is a pessimistic approach which gives us a precise worst case and the analysis can be limited to the first job of each task.

2.3.3 Fixed Preemption Points (FPP)

Here, we define the following values 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.

Then, we can define the blocking time of each task τ_i as

$$B_i = \max_{j:P_j<P_i}\{q_j^{max} - 1\}$$

and again, the authors propose a feasibility approach based on the worst response time.

2.4 Carousel

Another approach, presented in [WA12], supposes that the highest costs during a preemption are the cache-related costs, and thus proposes an approach to greatly reduce those. A common approach to achieve this ([RP07]) is *static partitionning*, which consists of allocating a fixed portion of memory for each tasks, and thus results on reduced cache misses but also on reduced memory

space available for each task, and an increased need for memory optimisation.

In this spirit, the authors of [WA12] present the Carousel approach, which reduces the occurrence of cache misses by requiring that each job begins by saving the section of memory that it needs to use, and ends by restoring it. This can be done in a fixed time (which can thus be easily integrated in the WCET) and ensures that no task interference occurs.

This approach forces the scheduler to schedule jobs in a LIFO manner : when a preemptive job $J_{i,j}$ finishes its execution, the following job has to be a new job or the job preempted by $J_{i,j}$. If this is not the case, $J_{i,j}$ will restore the wrong context at the end of its execution. The authors also assume the scheduling to be FTP, but we think the approach could be easily adapted to FJP schedulers.

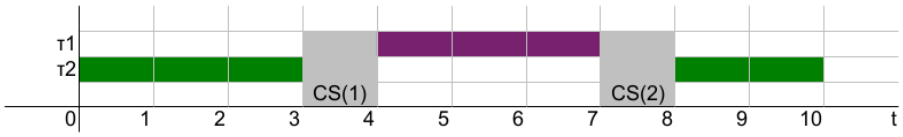

Figure 6: CS_1 and CS_2 are the (constant) scheduler costs before and after each preemption

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 1.1.5) 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 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 model taking preemption costs into account. First, we describe our preemption model. Then, we present a result 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 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 new model describing the exact behavior of preemptions 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 **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.

This preemption model is based on the classic model presented in section 1.1, with a different behavior when handling preemptions. The preemption behavior of the model is presented in Figure 7. 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 grayed out color). When this period is over, if the job still has the highest priority (which is not the case 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).

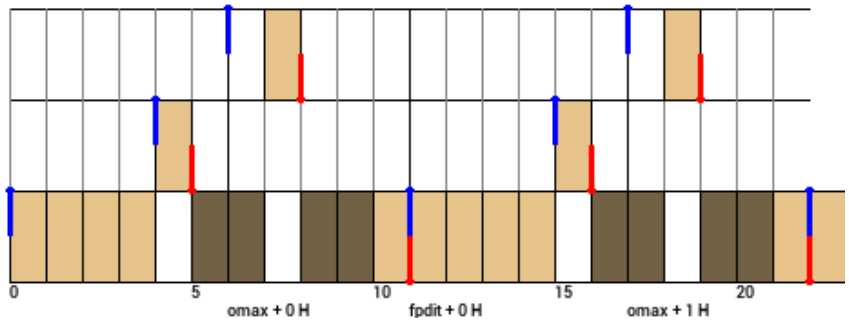

Figure 7: System scheduled by EDF in the preemption model. Note the non-interruptible preemption in $t = 6$

3.2 Previous results

In [MYS07], we find feasibility conditions derived from those presented in section 1.1 but integrating the preemption cost (assumed to be constant) for synchronous implicit deadline task set and considering only FTP schedules. We will present this approach here.

3.2.1 Definitions

The following values are defined:

- α : (constant) cost of one preemption.
- $N_p(J_{i,j})$: 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.

3.2.2 Feasibility test

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.

3.3 General properties of scheduling with the preemption model

Modifying our model implies that previous results may not hold. In this section, we show how most assumptions that were made in Section 1.1 do not hold with the preemption model.

3.3.1 Optimal schedulers

In Section 1.1, we introduced *optimal scheduler*, i.e. schedulers which correctly 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.

None of these schedulers are optimal with the preemption model. To convince ourselves our this, we look at Task System 1:

Task System 1

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

Figure 8 shows that it is in fact possible to schedule this system such that no preemptions occur and every deadline is respected.

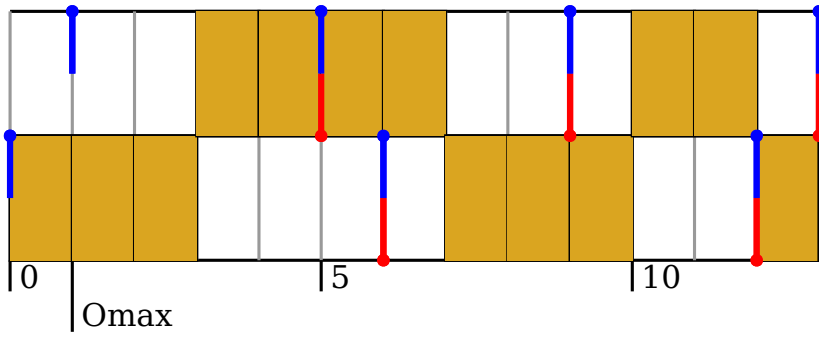

Figure 8: Task System 1 correctly scheduled. After $t = 12$, the behavior is periodic

However, Figure 9 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, and so that it is not optimal with our preemption model. LLF produces the exact same schedule, which means it is also not optimal.

Non-Optimality Within our preemption model, neither EDF nor LLF are optimal schedulers

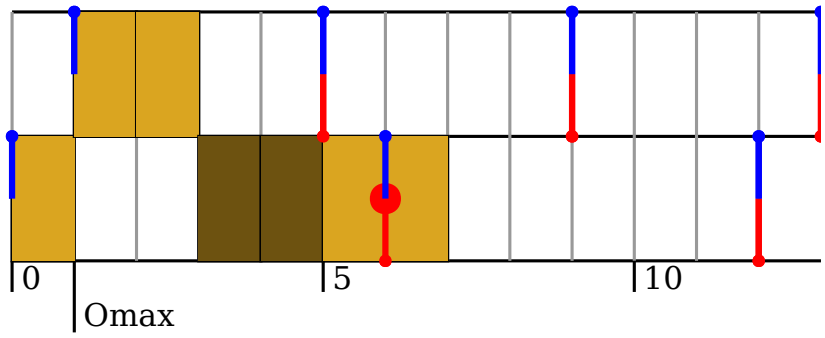

Figure 9: Task System 1 scheduled by EDF, with a deadline miss at $t = 6$

4 Implementation

5 Minimizing preemptions with PM-EDF

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.

[BBY09] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems: a survey. 2009.

[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.

[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 multiprocesseurs. *Theoretical Easter Physics*, 2013. (For ETR'13).

[JIS79] M Tech CSE Syllabus JIS. Computers and intractability a guide to the theory of np-completeness. 1979.

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

[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*, Pisa, Italie, 2007.

[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.

[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.

[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.