Tutorial on real-time scheduling

Emmanuel Grolleau
Laboratoire d'Informatique Scientifique et Industrielle
grolleau@ensma.fr

Abstract

This presentation is a tutorial given as a survey of the basic problems arising in real-time schedulability analysis on uniprocessor systems. It mainly focuses on on-line scheduling policies (fixed priority policies - FPP, and dynamic priority scheduling) on a preemptive scheduling scheme, and insists on the basic concepts of busy period and processor demand. This tutorial is an extract of a lecture given in Masters of Computer Science and most of the results presented here can be found in books [Liu00][But04].

1. Introduction

1.1. Real-Time systems and programming

A real-time system is interacting with a physical process (UAV, aircraft, car, etc.) in order to insure a correct behaviour. The system computes a view of the state of the process and of the environment through sensors (e.g. an inertial measurement unit) and acts using actuators (e.g. the flaps). For now, let's say that sensors can be passive or active: passive sensors are meant to be polled (the system has regularly to get its value), while active sensors send a value to the system, which is informed of the arrival of a new value by an interrupt.

Unlike a transformational system, which computes an output from an input (hopefully) in a strict deterministic behaviour (for the same input, the output is always the same), the behaviour of a real-time system is hardly repeatable (the environment is usually different from a test to another). This characteristic is shared with reactive systems (usually, real-time systems are in the category of reactive systems, since they react to external events). We can split reactive systems into two categories: the synchronous ones, and the asynchronous ones. The synchronous hypothesis assumes that the reaction to an event is instantaneous, therefore, the system is supposed to react immediately (or in a time compatible with the minimal inter-event duration) to an event in any case. The asynchronous hypothesis is used when the CPU load is too high for a synchronous hypothesis.

We focus on asynchronous real-time systems. Such systems are multitask because the rhythms involved are different in a reactive system: a polling task reading a sensor could have a period of 10 milliseconds, while another one has a period of 5 ms, and a third one is triggered by an interrupt, while a 4^{th} one should send a command to an actuator every 20 ms.

There are two ways to program a (asynchronous, we are now always in the asynchronous hypothesis) real-time system: event-driven programming, and time-driven programming. On one hand, in event-driven systems, events are interrupts. Events are either coming from sensors (including a keyboard or a pointing device) or from the internal clock. Thus, a task is released by an event, treats it, and then waits for the next event. On the other hand, time-driven systems are based on a time quantum: a task is awaken at its time, does a treatment anyway (even if nothing happened since its last release) and then sleeps until its next release time. Active sensors can also be used in time-driven systems: in this case, the data sent by the sensor is read by the interrupt service routine (ISR) and put into a buffer. The task in charge of this sensor will read the buffer during its next release (unless it's replaced in the meantime by a new data sent by the sensor). In time-driven systems, tasks act like if active sensors were passive sensors: they are polling their value which has been stored previously by the ISR. Thus, event-driven systems are more reactive to external events delivered by active sensors (the task is released as soon as possible after the occurrence of the event), but their release dates are unknown. This particularity is really important in the sequel and in the models used for schedulability analysis.

1.2. Different basic real-time task models

Supposing all the tasks are independent and don't share critical resources, or communicate (tough hypothesis!), we first consider independent task models.

1.2.1. Non concrete task systems

Since we only focus on time and processor requirement, Liu and Layland [LL73] proposed a periodic model (fitting to time-driven and event-driven programming) based on a worst case execution time C_i (WCET) needed by the CPU to complete a task τ_i and its release period T_i.

A task is thus denoted $\tau_i::=<C_i,T_i>$.

Note that this task model is non concrete: the first release date of a task is unknown (it corresponds to event-driven systems).

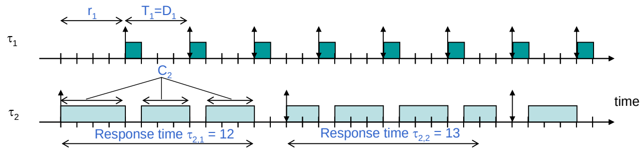

Figure 1: example of a task system $<C_i,T_i>$ $S=\{\tau_1<1,4>,\tau_2<10,14>\}$

Moreover, for such a model, the task has to finish before its next period: its relative deadline D_i is assumed to be T_i. The usual non concrete task model is denoted $\tau_i::=<C_i,D_i,T_i>$. Figure 1 shows a possible schedule for a non concrete task system when the release date (or offset) of τ_1 is $r_1=4$ while $r_2=0$.

Note that an instance $\tau_{i,j}$ of the task τ_i can be called a job.

The periodic task models fits with a lot of real-life task systems: in fact, most of the rhythms are periodic (polling passive sensors, then treatment chains, and even for active sensors, they usually behave periodically, or it is possible to find a minimal inter-event duration that can be used as a worst case period). Keep in mind that a WCET is a worst-case time, so when validating a system, we usually have to consider an execution time varying from 0 to the WCET of the task. It's the same for the period: the period should be considered varying from T_i to $+\infty$.

What is the worst-case scenario for a task, since the release dates, the WCET, and the periods may vary?

1.2.2. Concrete task model

When the system to study is time-driven, all the release dates are known: in this case, the task system is said concrete, and a task can be defined by $\tau_i::=<r_i,C_i,D_i,T_i>$ when r_i is the release date of the first instance of τ_i. When all the release dates are the same, the system is called synchronous (see Figure 2). When some tasks are not released for the first time at the same instant, the system is called non synchronous or asynchronous.

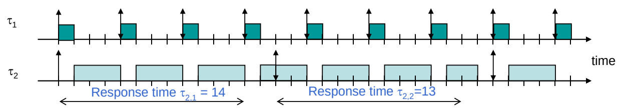

Figure 2: same system, but synchronous

1.3. Scheduling problems

A schedule is feasible if the worst-case response time of every job of every task is smaller than the task's relative deadline, in other words if all the deadlines are met during the life of the system. Most researchers propose scheduling policies or feasibility tests, or consider some quality of service or cope with more complex task models.

Except for basic problems, the feasibility problem is NP-hard, thus there are two choices: being exact at an exponential cost, or being pessimistic at a polynomial or pseudo-polynomial cost. Of course, better not to be optimistic when talking about feasibility.

Real-time scheduling community is usually bi-polar:

- on one hand, on-line scheduling (or priority-driven scheduling): during the execution of the system, a simple scheduling policy is used by the executive in order to choose the highest priority job in the set of ready jobs. This algorithm (usually called policy) is used when the executed job is finished or when it's blocked, or when another job is released or even sometimes at every time unit (quantum based scheduling). In this case researchers propose efficient schedulability tests (polynomials or pseudo-polynomials) that can be used off-line (i.e. in order to validate a scheduling policy for a system), rarely new scheduling policies, or they can study more specific task models. In fact, the more specific a model is for a problem, the less pessimistic the schedulability tests are. Some other interesting problems deal with optimality of scheduling algorithms or of feasibility tests;
- on the other hand, off-line scheduling (or time-driven scheduling) techniques, model based or using branch and bound or meta-heuristic algorithms, create a feasible schedule that can be executed endlessly by a dispatcher. In this case, researchers choose to deal with an exponential problem and have to cope with the state explosion problem.

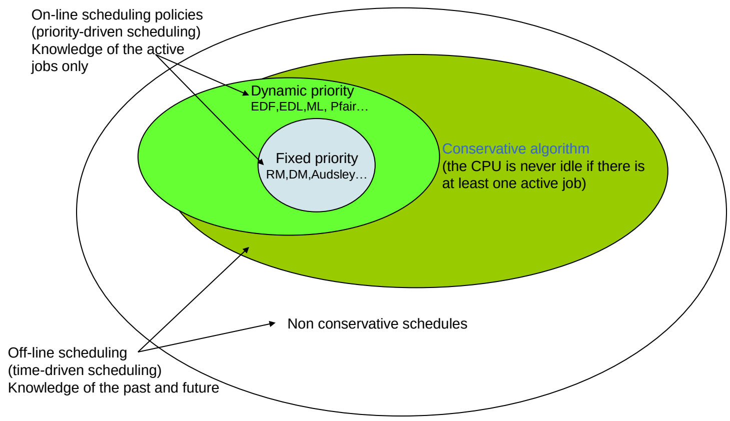

Figure 3: scheduling algorithms ordered by scheduling power

2. Fixed priority scheduling

The most widely used scheduling policy is FPP. The reason can be that most or all commercial off-the-shelf real-time executives offer FPP. The most important concepts to understand are the critical instant concept (for non concrete systems and synchronous concrete systems) and the busy period concept. We will see the impact of critical sections on the schedulability analysis in a third part. For this section, we assume that the tasks are ordered by priority level (priority(τ_1)> priority(τ_2)>…).

2.1. Critical instant

Since the duration, the periods, and for non concrete systems, the first release date may vary, it is important to study the worst-case behaviour of the tasks.

Critical instant theorem [LL73][Aud91][BHR93]: for independent task systems, in a context $<C_i,D_i,T_i>$ or $<r_i=0,C_i,D_i,T_i>$, the critical instant for a task τ_i, leading to its worst-case response time, occurs when τ_i is released simultaneously with all the higher priority tasks.

In Figure 1 and Figure 2, we can see an illustration of this theorem: the worst-case response time of τ_2 occurs when it's released at the same time as τ_1. A task is delayed by higher priority tasks releases.

2.2. Busy period

A level-i busy period is a time period where the CPU is kept busy by tasks whose priority is higher or equal to priority(τ_i), where there is no idle point. An idle point corresponds to a point where the Time Demand Function meets the Time line (it corresponds to a point where all the previous requests of this priority level have been completed). Figure 4 shows the "classic view" of a busy period: initially, τ_1 and τ_2 are released; therefore, the CPU has to compute C_1+C_2 time units. The processing power is given on the diagonal: the CPU can process 1 time unit of work per time unit. When the time demand function crosses the time (line Time demand=Time), it is the end of a busy period. When there is no demand, the CPU remains idle until the next release, which is the beginning of the next busy period. A flattened view is presented in Figure 5: the time is subtracted to the time demand function, giving the workload to process.

Theorem [Aud91][ABTR93]: the worst-case response time for a task τ_i occurs during the longest level-i busy period.

Theorem [Aud91] [ABTR93]: the longest busy period is initiated by the critical instant.

Therefore, we know the worst-case for a task τ_i (case of non concrete task systems and synchronous task systems): we just have to consider the critical instant, build the first level-i busy period, study all the jobs of τ_i occurring in the busy period, and claim the worst response time of these jobs as the worst-case response time of τ_i. Does a busy period always end? Yes, if and only if the processor utilization ratio $U=\Sigma_{i=1..n}C_i/T_i\leq 1$ which is a trivial necessary schedulability condition on one CPU.

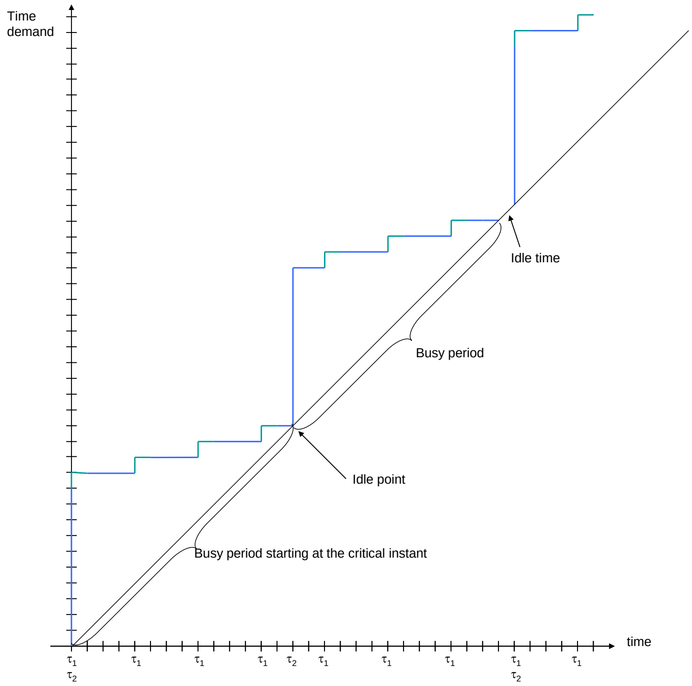

Figure 4: level-2 busy period for the task system S

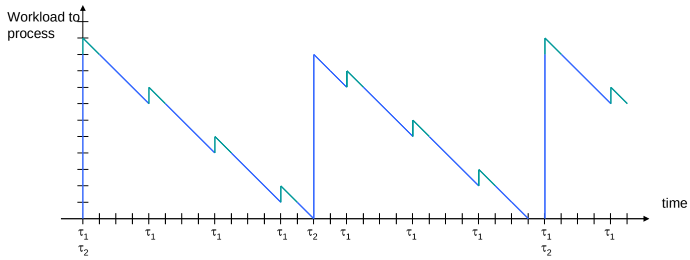

Figure 5: flattened view of the level-2 busy period for the task system S

We can notice that if the processor utilization ratio is less than 1, then the processor will remain idle at the same time instants (idle slots left into the level-n busy period) for any conservative algorithm. If the system is synchronous, there will be $LCM_{i=1..n}(T_i)\times(1\text{-}U)$ in any time period of length $LCM_{i=1..n}(T_i)$.

Only problem: it's exponential in time if we build the time demand function time unit per time unit! In fact with a processor utilization ratio of 100%, the level-n busy period ends at $LCM_{i=1..n}(T_i)$ which is bounded by $3^{\max(Ti)}$.

In fact, the end of a busy period is given by the first time the time demand function meets the line y=x (in Figure 4) except at 0. [JP86] gives a pseudo-polynomial test when only one job of τ_i can be in the busy period (the authors suppose that $D_i\leq T_i$, thus if 2 jobs of τ_i are in the busy period, the system is not feasible with the chosen FPP):

Starting from C_i the length of the level-i busy period R_i is given by the smallest fixed point of the equation:

$$R_i^{(0)} = C_i$$

$$R_i^{(n+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j$$

With hp(i) the set of indices of higher priority tasks than τ_i. $W_i(t)=\Sigma_{i=1..j}\lceil t/T_i \rceil C_i$ is called the processor demand function of level i: it represents the amount of CPU requested by tasks whose priority is greater or equal to priority(τ_i) in the interval $[0,t[$. Using this notation, $R_i^{(*)}$ is the smallest fixed-point of the equation $t=C_i+W_i(t)$.

This equation consists in taking C_i as the shortest possible busy period $R_i^{(0)}$. For the next step, we consider that the higher priority jobs released in the interval $[0..R_i^{(0)}[$ will grow the busy period by their WCET. We carry on until all the jobs in the busy period have been taken into account, let's note the length of the busy period $R_i^{(*)}$. If $R_i^{(*)} \leq T_i$ (thus τ_i doesn't occur more than once in the busy period), following the critical instant theorem, and Audsley's theorems, we can conclude that $R_i^{(*)}$ is the longest busy period, and that the worst-case response time of τ_i occurs in this busy period. Since r_i was assumed to be 0, the worst-case response time RT_i of τ_i is $R_i^{(*)}$.

What is really interesting in this test is the fact that the priority order of higher priority jobs has no influence on the response time of τ_i.

Nevertheless, if $R_i^{(*)}$ is greater than T_i, the busy period is not over, since τ_i is released at least a 2nd time. We thus have to carry on the test taking the following instances of τ_i into account. This is exactly what is proposed in [Leh90][LSST91]: k represents the number of occurrences of τ_i in the busy period. Starting with k=1 (obtaining exactly [JP86] test).

$$R_i^{(0)}(k) = kC_i$$

$$R_i^{(n+1)}(k) = kC_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j$$

The difference is that if $R_i^{(*)}(1)>T_i$ then the busy period initiated by the critical instant contains at least two occurrences of τ_i, therefore, the test has to be carried out for k=2. If $R_i^{(*)}(2)>2T_i$, we have to carry on for k=3 and so on until $R_i^{(*)}(k) \leq kT_i$. The worst-case response time of τ_i is found in this busy period, but it is not necessarily the first job's response-time. The response time of the job $\tau_{i,k}$ (k starting at 1) is $R_i^{(*)}(k)-(k-1)T_i$ (date of its termination minus date of its release).

As an example, consider the system ($<C_i,D_i,T_i>$) $S=\{\tau_1<26,26,70>, \tau_2<62,118,100>\}$.

The application of the formula is straightforward for τ_1 since there is no higher priority job:

$R_1^{(1)}(1)=C_1=26$

$R_1^{(2)}(1)=C_1=26=R_1^{(1)}(1)=R_1^{(*)}(1)$

$R_1^{(*)}(1) \leq T_1$, therefore, no additional job of τ_1 is involved in the busy period, and the worst-case response time $RT_1=R_1^{(*)}(1)-(1-1)T_1=26$. We can conclude that τ_1 always meets its deadline, since $RT_1 \leq D_i$.

For τ_2, the formula has a really interesting behaviour:

$R_2^{(1)}(1)=C_2=62$

$R_2^{(2)}(1)=C_2+\lceil 62/70 \rceil C_1=88$

$R_2^{(3)}(1)=C_2+\lceil 88/70 \rceil C_1=114$

$R_2^{(4)}(1)= C_2+\lceil 114/70 \rceil C_1=114=R_2^{(3)}(1)=R_2^{(*)}(1)$. The response time of the first job is 114, which meets the deadline, but $R_2^{(*)}(1)>T_2$. That means that the 2nd job is part of the same busy period. We thus have to continue for k=2:

$R_2^{(1)}(2)=2C_2=124$

$R_2^{(2)}(2)=2C_2+\lceil 124/70 \rceil C_1=176$

$R_2^{(3)}(2)=2C_2+\lceil 176/70 \rceil C_1=202$

$R_2^{(4)}(2)=2C_2+\lceil 202/70 \rceil C_1=202=R_2^{(3)}(2)=R_2^{(*)}(2)$. The 2nd job ends at the time 202. That means that its response time is $202-(2-1)T_2=102$. This response time is greater than the period, so the 3rd job is part of the busy period and the test has to be led for k=3.

We carry on for k=3 and the following until we reach k=7, where we finally find the end of the level-2 busy period:

$R_2^{(*)}(3)=316 \Rightarrow$ the response time of $\tau_{2,3}$ is 116

$R_2^{(*)}(4)=404 \Rightarrow$ the response time of $\tau_{2,4}$ is 104

$R_2^{(*)}(5)=518 \Rightarrow$ the response time of $\tau_{2,5}$ is 118

$R_2^{(*)}(6)=606 \Rightarrow$ the response time of $\tau_{2,6}$ is 106

$R_2^{(*)}(7)=696 \Rightarrow$ the response time of $\tau_{2,7}$ is 96 which is less than T_2, ending the busy period…

We see that the worst-case response time is given by the 5th job: $RT_2=118 \leq D_2$, thus all the tasks meet their deadline and the system is feasible.

We will see in the sequel that this test has been widely used with more specific task models, and constraints. Just note that the number of values to test for k can be exponential (up to $LCM_{j=1..n}(T_j)/T_j$ for each task τ_i).

2.3. Specific feasibility tests

The response-time calculation is not related to any specific policy, it is exact (necessary and sufficient condition), but it's not polynomial. Some authors proposed polynomial sufficient feasibility tests based on specific policies. These conditions consider only independent tasks, and don't give good results as soon as some critical sections are present in the system. The reader can refer to [ABDTW95] for a survey. A lot of results are presented in a practical handbook [RMA].

Rate Monotonic (RM) was the scheduling policy proposed in [Ser72][LL73]: the lower the period, the higher the priority. In the model studied by the authors, $D_i=T_i$. Thus the tasks with a lower period have a lower relative deadline: that makes RM the most intuitive FPP for tasks systems with $D_i=T_i$.

RM is optimal for synchronous, independent task systems with implicit deadline ($D_i=T_i$) [LL73] in the class of FPP. That means that if the system is schedulable with a FPP, then it is schedulable with RM. Keep in mind that the worst-case scenario occurs for non concrete task system when tasks are considered synchronous, therefore, results standing for synchronous task systems stand for non concrete task systems.

When $D_i<T_i$, the most used priority policy is know as Deadline Monotonic [LM80][LW82], where the lower the relative deadline, the higher the priority. In fact, RM is a particular case of DM. DM is optimal for synchronous independent tasks systems whose relative deadline is less or equal to their period, in the class of FPP.

Therefore, when the systems are concrete and synchronous, or when the system are non concrete, DM is the most widely used FPP. RM and DM have been intensively studied, and a lot of authors proposed polynomial time feasibility tests. Some tests are exact for some specific task systems, but they are necessary (thus pessimistic) conditions for the general case.

The best known test for RM is proposed in [LL73]: if a tasks system is synchronous, is composed of n independent tasks, whose $D_i=T_i$, then $U\leq n(2^{1/n}-1)$ is a sufficient necessary schedulability condition. This technique is reducing the field of uncertainty with a polynomial time test (see Figure 6). The more tasks in a system, the bigger the uncertainty (starting at 82% for 2 tasks, 78% for 3 and tending to a limit of 69% for an infinite number of tasks). This bound is quite low, since simulations [LSD89] showed that the average bound was around 88%. We can note that [DG00] showed that the proof in [LL73] was incomplete and completed it.

Liu and Layland's test has been tweaked in [KM91], where the simply periodic task sets are used (a simply periodic task set is such that for every couple τ_i and τ_j of the set, if period $T_i>T_j$, then T_i is an integer multiple of T_j. In this case, if there are k simply periodic task sets, then the necessary condition is $U\leq k(2^{1/k}-1)$. That means that if a system contains only simply periodic tasks, k=1, and the system is feasible with RM if and only if $U\leq 1$. When the tasks are not simply periodic, the test of [LL73] can still be improved using the fact that the closer the tasks are to being simply periodic, the larger the utilization can be [BLOS96]. Another exact test for RM can be found in [LSD89]: based on the processor demand function $W_i(t)=\Sigma_{i=1..j}\lceil t/T_i\rceil C_i$, the test consists, for each priority level, in checking the fact that the processor demand function meets the time line (i.e. $W(t)/t\leq 1$) at least once in the interval $]0,T_i]$. Since the local minima of this function correspond to the release date of the higher priority tasks, and to the release of the next instance of τ_I, only these points need to be tested.

More recently, Bini and al. proposed two tests for RM: the hyperbolic bound (H-bound) [BBB03] and the δ-HET [BB04]. The H-bound is simply $\Pi_{i=1..n}(C_i/T_i + 1)\leq 2$ which is a sufficient condition for a system to be feasible with RM. H-bound has been proven to be the tightest possible test based on the processor utilization. δ-HET is based on [LSD89] test, wisely studied as an hyperplane representation, allowing the authors to provide a test that can be tuned to control the complexity from polynomial (sufficient condition) to exact pseudo-polynomial time with less steps than a classic response-time analysis.

We can find an exact test for the DM policy in [LSST91]. A test in $O(n.2^n)$ is proposed in [MA98].

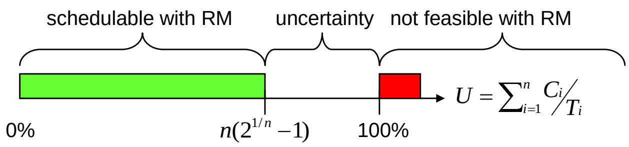

Figure 6: reducing uncertainty with the processor utilization

2.4. Non synchronous tasks

All the discussed tests assume a critical instant to exist, while it's not always the case when the task system is asynchronous ($\exists i,j/\ r_i\neq r_j$) in a concrete system. In fact, forbidding the critical instant to happen can be interesting in order to increase the schedulability of a system, that wouldn't be feasible otherwise. There are mainly two problems: choosing the right release dates to avoid the critical instant (offset free systems), and feasibility analysis.

For example, if two tasks τ_i and τ_j should never be released at the same time, there are $gcd(T_i,T_j)$-1 possible integer values for their relative offset [Goo03]. Then choosing wisely the release times may improve schedulability, moreover, [Aud91] proposes an optimal priority assignment for such systems by testing $O(n^2)$ priorities. Nevertheless, testing the feasibility of a priority assignment for asynchronous independent task systems is NP-hard [LW82]. We can't just focus on one busy period and conclude, but all the busy periods have to be studied, depending on the task system, at least until $LCM(T_i)$ up to $\max(r_i)+2\times LCM(T_i)$ [LW82][GG04].

2.5. Practical factors

The practical factors are the most interesting ones for the researchers' community of uniprocessor scheduling: most citations for independent "classic" task systems are dating from the 70's to the mid-90's. Usually, it seems that when someone has a problem involving a new practical factor, he is proposing a feasibility test, or even a new scheduling algorithm, improved later by other people. So starting in the 90's, researchers have been proposing adaptations of classic scheduling theory to the real world.

2.5.1. Critical sections and non-preemptible tasks

Except for deadlock potential problems, the respect of mutual exclusion introduces new problems in real-time scheduling: scheduling anomalies, and priority inversion.

A scheduling anomaly is presented in Figure 7: recall that for on-line scheduling, the WCET is a worst-case time. Therefore, even if on a simulation starting at the critical instant the system given in the form $<C_i,D_i,T_i>$ $S=\{\tau_1<2,15,16>, \tau_2<6,15,16>, \tau_2<6,16,16>\}$ seems to be feasible with DM, it is not (note: it would be feasible if we were using the schedule in a dispatcher). When $C_2=6$, all the deadlines are met in the schedule, but not when $C_2=5$. This phenomenon is known as a scheduling anomaly: reducing the execution time or increasing the period can be worse than the worst-case temporal parameters. Therefore, even if the simulation could be used to validate a system composed with independent tasks, it can't be used as soon as critical sections are involved.

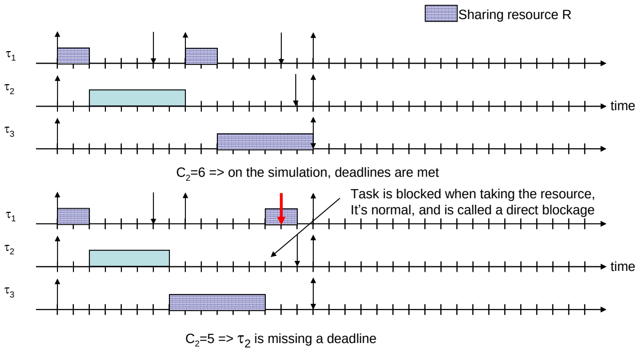

Figure 7: a scheduling anomaly due to resource sharing

The problem of priority inversion is illustrated by the Figure 8: a priority inversion occurs when a task is delayed by a lower priority task that does not share a resource with it. In this figure, τ_2 is running while the highest priority job is waiting for τ_3 to complete its critical section.

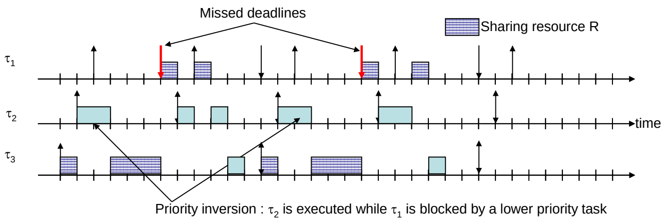

Figure 8: a priority inversion due to resource sharing

An intuitive way to avoid the priority inversion is to use the Priority Inheritance Protocol (PIP) [SRL90]: a task holding a resource which is blocking a higher priority task inherits the higher priority task's priority until it frees the resource (see Figure 9).

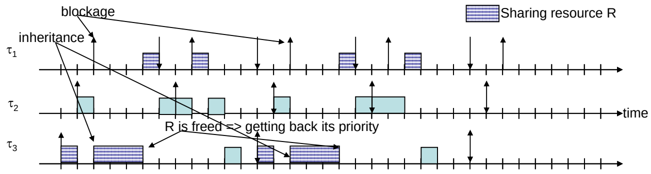

Figure 9: priority inheritance protocol

The PIP avoids any priority inversion, but it does not reduce the number of blockages that a task can suffer when trying to enter in a critical section: in Figure 9, if τ_3 was using another resource R_2 while using R, and if R_2 was already used by a lower priority task, then τ_1 would have to wait for both critical sections to end. Moreover, a task using several resources can be blocked each time it's trying to enter in critical section. Studying a graph of resource uses, we can compute for a system how many resources can block a job, and how long the longest critical section would be. We can deduce a blocking factor B_i of a job. Note that during a level-i busy period, a task can be blocked at most once, thus, the worst-case response time of a task is written:

$$R_i^{(0)}(k) = B_i + kC_i$$

$$R_i^{(n+1)}(k) = B_i + kC_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j$$

We assume the worst-case scenario as being an instant where all the higher priority jobs are released at the critical instant, while all the lower priority jobs have just started their longest critical section, implying the longest blocking time. Note that when using this protocol, a task can be delayed by a lower priority task even if it's not sharing a resource with it. This is called indirect blocking. The task τ_2 is indirectly blocked by τ_3 in Figure 9.

Sha and al. [SLR90] use PIP as an intuitive protocol but they show its inefficiency compared to the priority ceiling protocol (PCP). In PCP each resource R has a ceiling π_R, defined as the highest priority among the tasks using it. The system ceiling is defined as $\pi_S=\max_{\forall \text{ resource in use } R}(\pi_R)$. The protocol functions exactly like the PIP, with an additional resource access rule: a task can access a resource if its priority is strictly higher than the system ceiling or if it is itself the cause of the value of the system ceiling. PCP avoids any priority inversion (like PIP), moreover, a task can be blocked only once per busy period, even if it is using several resources. A blocking time can't exceed the length of one critical section. This is due to the rule introduced by PCP: if there is a critical section using a resource R1 required by a task τ_i (thus, $\pi_{R1} \geq priority(\tau_i)$ and $\pi_S \geq \pi_{R1}$), then no other task can enter in critical section unless its priority is strictly greater than the priority of τ_i (because $\pi_S \geq priority(\tau_i)$). An interesting side effect of PCP is that no deadlock can occur.

While PIP can't be implemented efficiently, and has a poor behaviour regarding the value of B_i, PCP can be implemented efficiently in its immediate version (having the behaviour of the super priority protocol proposed in [Kai82]). The exact same worst-case behaviour takes place when the inheritance occurs as soon as a task enters in a critical section. As a result, Immediate PCP is the most widely used protocol in commercial off-the-shelf real-time executives (e.g. POSIX, OSEK/VDX, Ada standards).

Non-preemptible tasks are a particular case of tasks sharing resources (we can consider that all the tasks share the same resource), thus scheduling anomalies can occur too (even if, of course, priority inversion can't occur). Validating a non-preemptible task system is NP-hard [LRKB 77][JSM 91]. Their behaviour is closer, though, to the non-preemptible critical section [Mok83].

2.5.2. Precedence constraints

The task model considers communicating tasks to be in a canonical form (e.g. if a task has to wait for a message, the message has to be awaited at the beginning of the task, and messages are sent at the end): it supposes that the original communicating tasks are split into several canonical tasks. The period of the tasks are assumed to be the same. When the priorities are not consistent with the precedence constraints (a higher priority task waiting for a lower priority task to complete), scheduling anomalies can occur (releasing a precedence constraint, or reducing the duration of a job can lead to a worse behaviour) [RRGC02]..

2.5.3. Multiframe model and transactions

Alternative more accurate models than the one of [LL73] have been introduced in the last decade. We focus here on the multiframe and the transaction models. Different task models are presented in [Bar98].

The multiframe model has been introduced by [MC96][MC97]. A multiframe task is non concrete, and characterized by $<T_i, P_i>$ where T_i is the period of the task, and P_i is a set of execution times. For example, $<10,\{3,2,1,5\}>$ represents a task of period 10, whose first job has a WCET of 3, 2^{nd} job of 2, 3^{rd} job a WCET of 1, 4^{th} job a WCET of 5, 5^{th} job a WCET of 3, and so on, repeatedly. In works concerning multiframe tasks, this task has 4 frames. The longest one is called the peak frame. The relative deadline is equal to the period. This model can be used when tasks have various amounts of data to treat during their execution. Note that a periodic task is a particular case of a multiframe task. Mok and Chen proposed a utilization-based sufficient feasibility test for RM, improved in [HT97][KCLL03][LLWS07] . Some other tests are based on a fixed-point lookup like in [BCM99].

The main problem is that determining the worst-case scenario for a multiframe set is intractable in general [MC96]: determining the critical instant requires to compute all the combinations of the releases of the tasks in each multiframe task ($\Pi_{i=1..n}n_i$ combinations).

For some particular patterns, when the peak frame and the successive frames (modulo the number of frames) always generate the worse interference pattern, the task is said Accumulatively Monotonic (AM). For an AM task, by construction, there is only one task that can lead to the worse-case interference on a lower priority task. Therefore in this case, when there are only AM tasks, the problem is tractable since there is only one known worst-case scenario which is the one where a frame (the validation is lead frame by frame for a task) is released at the same time as all the higher priority peak frames.

The multiframe model has been extended in [BCGM99] as the generalized multiframe model (gmf) where the frames don't have the same deadline, and not the same period (i.e. not the same interval between successive frames of a task). In this model, a task is thus characterized by 3 vectors (WCET, relative deadline, minimal interval to the next frame (called period)). They study the time demand function of the tasks in order to validate the frames.

In parallel to the development of this model, the transaction model, derived from Tindell's task model with offsets, has been investigated. This model is a little similar to the gmf, except that the priority of the frames can differ, that the frames can have a jitter, may overlap, and of course that the vocabulary is quite different. A transaction is defined as a set of tasks. In fact, the transaction itself is non concrete (event-driven), but the tasks inside of a transaction are released a certain time after the release of the transaction, this time is the offset of the task (note that the difference between the offsets of two successive tasks would be the period of the first task in the gmf model), thus defined as the offset compared to the beginning of the transaction. This model has been introduced in [PH98]: a transaction $\Gamma_i=<\{\tau_{i,1},\ldots, \tau_{i,|\Gamma i|}\},T_i>$ where T_i is the period of the transaction (minimal interval between 2 successive activations), and each task of a transaction is defined as $\tau_{i,j}:=<C_{i,j}, \phi_{i,j}, D_{i,j}, J_{i,j}, B_{i,j}, P_{i,j}>$ where $C_{i,j}$ is the WCET, $\phi_{i,j}$ is the offset relative to the beginning of the transaction, $D_{i,j}$ the relative deadline, $B_{i,j}$ the blocking factor due to resource sharing, $P_{i,j}$ the priority, and $J_{i,j}$ the release jitter. The concept of release jitter has been introduced in [Tin94]. A release jitter translates the fact that a task can have to wait up to a certain time before being able to start after its release date. For example, a task awaiting a message coming from a network could be activated between a planed release time and this release time plus its jitter (which could be the difference between the latest arrival time of the message and its earliest arrival time). This parameter is widely used in the holistic analysis used to validate distributed real-time systems.

Going back to the transaction model, let's call 0 the date when transaction Γ_i is released, the task $\tau_{i,j}$ is released at the date $\phi_{i,j}$ but may be delayed until the date $\phi_{i,j}+J_{i,j}$.

[PH98] proposed a interference based sufficient method using the time demand function, whose calculus is optimized in [TN04]. The test has been improved in [TN05]: the authors noticed that the classic time demand function had chances to miss the fixed-point and slowed it down, by forbidding it to grow faster than the time. The obtained worst-case is far less pessimistic than in [PH98]. [TGC06] showed that the transactions were a generalization of the gmf model (itself generalizing the multiframe model), and studied similar properties as the ones used for the multiframe model (AM transactions), not taking the jitter into account.

2.5.4. Miscellaneous

Other practical factors have been studied, like the tasks that self-suspend (e.g. during an I/O operation) There are scheduling anomalies when tasks can self-suspend, and the feasibility problem is NP-hard [RRC04]. Therefore, the self-suspension can be replaced, like in the case of critical sections, by a blocking factor [Liu00]. Some studies split the self suspension tasks and use the jitter to compute a worst-case response time [KCPKH95]. An exact but exponential worst-response time calculation method is proposed in [RR06] and several approximation tests are compared.

Different other practical factors have been studied recently, like energy aware scheduling that takes profit of CPU ability to change their execution speed in order to save energy; another example is taking into account the bounded number of priority levels of some executives, considering hierarchical schedulers, take the context switch time into account, etc. Some authors focused on relaxing the timing constraints, since for several kinds of real-time systems, the deadlines don't have to be all met (e.g. model (m,k)-firm, Quality of Service, etc.).

3. Dynamic priority scheduling

3.1. Optimality

The most well know algorithm is Earliest Deadline First (EDF), where the priority increases with the urgency (proximity of the deadline). The first known version was called Earliest Due Date, and [Jack55] proves its optimality regarding the lateness minimization, in the rule called Jackson's rule: any algorithm executing tasks in a non decreasing order of deadlines is optimal for minimizing the maximum lateness. The proof is really nice, and based on the lateness of a task τ_I, noted $L_i=RT_i-d_i$ where d_i is the deadline of τ_i, and TR_i its response time. Note that this proof assumes τ_i to be a job released at the beginning of the application, but [Horn74] generalized it to non synchronous jobs, so it can be taken for periodic tasks.

Let A be an algorithm minimising the maximal lateness, and τ_a and τ_b with $d_a \leq d_b$ such that τ_b ends right before τ_a. Let σ be the schedule produced by A. Note that A doesn't fit Jackson's rule. In σ, $L_{max}(\tau_a, \tau_b)=L_a$ (see Figure 10). Let σ' be the same schedule except that the execution of τ_a and τ_b are reverted. In σ', $L'_{max}(\tau_a,\tau_b)=\max(L'_a, L'_b)$, and $L'_a \leq L_a$ and $L'_b \leq L_a$. Therefore, the maximal lateness of the schedule can't be increased. This technique can be repeated until all the tasks fit Jackson's rule.

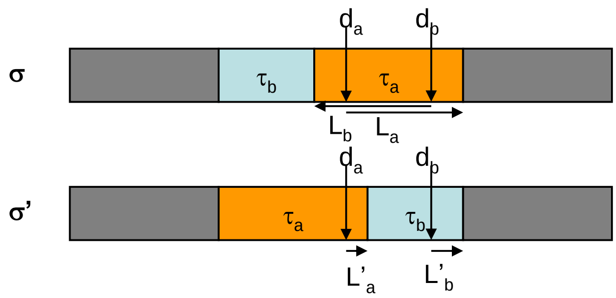

Figure 10: illustration of Jackson's rule

[Der74] and [Lab74] showed the optimality of EDF in meeting the deadlines, and [LL73] showed that a necessary and sufficient condition for a system of periodic independent tasks with $T_i=D_i$ was $U\leq 1$. Off course, few real-life system meet these conditions, therefore, studies have been led to take practical factors into account.

Even if we will focus on EDF in this presentation, other algorithms have been studied (like Minimal Laxity, a.k.a. Least Laxity, a.k.a. Least-Slack-Time First [Mok83] or Earliest Deadline Last, that both have the same optimality properties for independent task systems).

3.2. Processor demand concept

As soon as $D_i \neq T_i$, feasibility tests can use the concept of processor demand. This concept is applied for concrete and synchronous tasks systems. For non concrete and concrete asynchronous systems, it is hard to determine what the worst-case scenario for a task is. [Spu96] showed that for non-concrete task systems, a worst-case scenario for a task occurs when all the other tasks are released simultaneously, but one has to check different release dates for the task under analysis.

For concrete synchronous task systems, [JS93] proposed a feasibility test based on the processor demand: let B_p be the length of the first busy period (that would correspond to the level-"lowest priority" busy period in a FPP), obtained as the smallest fixed point of the equation $W(L)=\Sigma_{\forall task\ \tau i}\lceil L/T_i \rceil C_i$. A concrete synchronous independent task system with $D_i=T_i$ is feasible with EDF if and only if:

$$\boxed{\forall L \le \min(LCM(T_i), B_p), L \ge \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i}$$

This test doesn't look very efficient, since feasibility in this context can be tested just by computing the processor utilization. Nevertheless, it's helping to understand what's underlying EDF behaviour: $\lfloor L/T_i \rfloor$ represents the number of jobs of τ_i that must be completed at time L. Therefore, $\lfloor L/T_i \rfloor C_i$ is the amount of time that the schedule must have given to τ_i in the time interval [0,L]. If at any time, it has not been the case, then a deadline has been missed.

An efficient version of this test is given in [BRH90], it takes relative deadlines into account:

$$\boxed{\begin{aligned} &D = \left\{ d_{i,k} \middle| d_{i,k} = kT_i + D_i, d_{i,k} \le \min(LCM(T_i), B_p) \right\} \\ &\forall L \in D, L \ge \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \end{aligned}}$$

D is the set of deadlines in the busy period, thus all the deadlines have to be checked. $\lfloor (L-D_i)/T_i \rfloor+1$ is the number of completed deadlines during [0,L].

3.3. Practical factors

Several protocols have been proposed to handle resource sharing with EDF: [CL90] proposed a dynamic version of the priority ceiling protocol but this implies a high overhead due to the updates of the priority ceilings of the resources. A better version, using the concept of preemption ceiling level (rather than priority ceiling) can be found in [Bak91]. It has the same properties as the PCP in FPP.

3.4. Fixed priority vs. dynamic priority scheduling

Dynamic priority scheduling is optimal for independent task systems, so its scheduling power is strictly higher than the fixed priority scheduling. Moreover, Jackson's rule shows that integrating sporadic traffic in a deadline driven system is optimal to minimise maximal lateness. Nevertheless, when a task misses its deadline, and is carried on anyway, other tasks may miss their deadlines (it's called the domino effect). Moreover, dynamic priority scheduling is less predictable than FPP (keep in mind that the task parameters may vary, and that a lot of real-world applications are event-driven). On the other hand, FPP are easy to understand, and there is a notion of importance that comes naturally with the priority. When a job is late at a priority level k, it does not affect the higher priority jobs. A side effect of FPP is that the regularity of higher priority jobs is higher than with a dynamic priority scheduling. Moreover, all the commercial off-the-shelf executives offer FPP scheduling.

4. Non periodic traffic

According to [Liu00] there are two main categories of non periodic tasks: aperiodic and sporadic ones. Since they are handled job by job, we will talk about jobs. We will say non periodic jobs for sporadic or aperiodic jobs.

Sporadic jobs are hard deadline tasks, which can be accepted by the scheduler if it is possible to meet their deadline without missing any deadline of periodic tasks or previously accepted sporadic tasks. The problem with sporadic tasks is to create really efficient acceptance tests that are run on-line. Sporadic tasks, in J.W.S. Liu's point of view, don't have any inter-arrival time constraint. Note that in a non concrete model, a sporadic task which has a minimal inter-arrival time Δ_i, a WCET C_i and a relative deadline D_i (we can talk about a sporadically-periodic task) can be modelled by a periodic task τ_i with the same relative deadline and WCET, such that $T_i=\Delta_i$ (the parameter T_i can be greater on-line than in the model, thus T_i represents the minimal time between two consecutive activations of sporadically periodic tasks). In a concrete model, a sporadically periodic task can be modelled by a polling server: a polling server τ_S is a periodic task having whose parameters are such that $C_S=C_i$, $T_i+D_i \le \min(D_i,\Delta_i)$. Therefore, we will consider that a sporadic (non sporadically-periodic) job is characterized by $<r_i,C_i,D_i>$ where the release date r_i is known only at run-time, when the sporadic request arrives.

Aperiodic jobs don't have a deadline and are handled in a best-effort way, and the scheduler tries to complete them as soon as possible, without causing the periodic tasks or the accepted sporadic jobs to miss their deadline. An aperiodic job is characterized by $<r_i,C_i>$. Like for sporadic jobs, r_i is known only at run-time. Note that non periodic traffic is composed of independent tasks only.

We can think about two basic ways to handle non periodic traffic: the background treatment, and an interrupt-driven treatment.

Background treatment consists in using the idle slots left by the periodic/accepted sporadic traffic in order to compute non periodic traffic. However, the execution of the non periodic may be delayed unnecessarily, and the acceptance conditions of sporadic jobs would be drastic. Of course, one could use a periodic task as a sporadic server, whose WCET would be a bandwidth used to execute the non periodic jobs. Nevertheless, if it does not preserve its bandwidth when it's not used by a non periodic job, a job would have to wait for the next release of the server in order to be executed. On the other hand, an interrupt-driven treatment would consist in executing the non periodic jobs as soon as they arrive, which, of course, would cause the periodic/accepted sporadic tasks to miss their deadline.

We can distinguish 2 effective ways to handle non periodic traffic, the slack stealing, and the polling server preserving unused bandwidth. Most of the techniques can be used to handle aperiodic jobs, or sporadic jobs using an online acceptance test (feasibility test). For FPP, this test can be based on the time demand function or a polynomial-time estimation of the time demand, or on the processor utilization ratio. For deadline-driven scheduling, the acceptance test can use the processor demand, or the density (C_i/D_i).

Slack stealing consists in using the slack of periodic/accepted sporadic tasks to compute the sporadic/aperiodic jobs. The slack (or laxity) of a job is the difference between the remaining time until the next deadline and the time needed to complete the job. It is characterizing how long a job can be delayed without missing its deadline, in other words, its non-urgency. The idea behind slack stealing is to use this non-urgency in order to treat non periodic jobs. [LR92] proposed a slack-stealing algorithm for FPP scheduling. Even if it's optimal, this method uses the time demand analysis method on-line, which would imply an important overhead (pseudo-polynomial algorithm) in order to compute the slack time. Note that it is possible to use polynomial time approximation tests in order to implement this server. [CC89] presents a slack stealing mechanism using a characterization of EDL algorithm in order to compute efficiently the slack time in a deadline-driven system. For more online efficiency, some servers use a pre-computed slack-time table.

Polled execution with bandwidth-preserving consists in using a polling server (periodic task) that preserves its bandwidth when it's not needed, in order to be able to handle future non periodic requests until the next replenishment (next release) of the server. The basic bandwidth-preserving server is the deferrable server [LSS87][Str88]. For FPP scheduling, the server is validated like a task with a release jitter (the jitter represents the fact that the server can be delayed in order to keep its bandwidth when there is no non periodic task to compute). [GB95] uses a deferrable server in a deadline driven system. Task systems containing tasks with jitter are tougher to validate than without jitter, since the time demand is higher at the critical instant. Therefore, in order to avoid this problem, [SSL89][GB95] propose the sporadic server, where in any time interval of the period of the server, only its capacity can be used. Under this condition, the server can be considered as a periodic task with no jitter. Other authors proposed different bandwidth-preserving servers, especially for deadline-driven systems: the total bandwidth server [SB96], the constant utilization server [DLS97].

5. Conclusion

This paper tried to give a little survey of uniprocessor real-time scheduling problems and some solutions. In fixed priority scheduling, there are basically two categories of periodic task systems: the non concrete/concrete simultaneous systems that have the same worst-case behaviour in fixed priority scheduling. This worst-case is obtained at the critical instant. The second category is the concrete asynchronous systems for which finding the worst-case scenario is NP-hard. There are two kinds of feasibility tests for the FPP: time demand based tests, exact for independent tasks, working for any FPP, but requiring a pseudo-polynomial time; and processor utilization ratio based, polynomial-time tests, which are sufficient and not necessary (thus pessimistic) feasibility condition for any non trivial cases.

For dynamic priority scheduling (mainly EDF), it is usually assumed that the tasks are concrete, since the non-concrete case is hard to characterize. The acceptance tests are based on the processor demand, or on density.

Adding any practical factor leads to scheduling anomalies, and to NP-hard feasibility problems, which can be handled using worst-case blocking times, or more ad-hoc techniques.

While in the late 80's and 90's, different ways were explored in order to handle the non periodic jobs, some new models, closer to the reality than the classic $<C_i,D_i,T_i>$ model arose in the last decade.

A lot of areas are still opened: unexplored practical factors (that will open new research paths), more specific models (mix between time-driven and event-driven models), handling non periodic traffic into new tasks models (multiframe, transactions), etc.

6. References

[ABTR93] N.C. Audsley, A. Burns, K. Tindell, M. Richardson, A. Wellings, "Applying a new scheduling theory to static priority preemptive scheduling", Software Engineering Journal, vol. 5(5), pp. 284-292, 1993.

[ABDTW95] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, A.J. Wellings, "Fixed priority pre-emptive scheduling: an historical perspective", Real-Time Systems, Vol. 8, pp. 173-198; 1995.

[Aud91] N.C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times", University of York, YCS 164, 1991.

[Bak91] T.P. Baker, "Stack-based scheduling of real-time processes", Real-Time Systems, Vol. 3(1), pp. 67-99, 1991.

[Bar98] S. Baruah, "A general model for recurring real-time tasks", Proc. IEEE Real-Time Systems Symposium, pp. 114-122, Madrid, Spain. Dec. 1998.

[BB04] E. Bini, G.C. Buttazzo, “Schedulability analysis of periodic fixed priority systems”, IEEE Transactions on Computers, Vol. 53(11), pp. 1462-1473, November 2004.

[BBB03] E. Bini, G.C. Buttazzo, G.M. Buttazzo, “Rate Monotonic analysis: the hyperbolic bound”, IEEE Transactions on Computers, Vol. 52(7), pp. 933-942, July 2003.

[BCGM99] S.K. Baruah, D. Chen, S. Gorinsky, A.K. Mok, “Generalized Multiframe Tasks”, The International Journal of Time-Critical Computing Systems,Vol. 17, pp. 5-22, 1999.

[BCM99] S. K. Baruah, D. Chen, A.K. Mok, “Static-priority scheduling of multiframe tasks”, P*roc.* 11th *Euromicro Conference on Real-Time Systems*, pp. 38–45, June 1999.

[BLOS96] A. Burchard, J. Liebeherr, Y. Oh, S.H. Son, “New strategies for assigning real-time tasks to multiprocessor systems”, IEEE Transactions on Computers, vol. 44(12) pp. 1429-1442, 1996.

[BHR93] S.K. Baruah, R.R. Howell, L.E. Rosier, “Feasibility problems for recurring tasks on one processor”, Theoretical Computer Science, Vol. 1(118), 1993.

[BCGM99] S.K. Baruah, D. Chen, S. Gorinsky, A.K. Mok, “Generalized Multiframe Tasks”, Real-Time Systems, Vol. 17(1) pp. 5-22, 1999.

[BRH90] S.K. Baruah, L.E. Rosier, R.R. Howell, “Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor”, Real-Time Systems, vol. 2, pp. 301-324, 1990.

[But04] G. Buttazzo, “Hard real-time computing systems: predictable scheduling algorithms and applications”, Ed. Springer, 425p., 2004.

[CC89] H. Chetto, M. Silly-Chetto, “Some results of the earliest deadline scheduling algorithm”, IEEE Transactions on Software Engineering, vol. 15(10), pp. 1261-1269, 1989.

[CL90] M. Chen and K. Lin, “Dynamic priority ceilings: a concurrency control protocol for real-time systems”, Real-Time Systems, vol. 2(4), pp. 325-346, 1990.

[Der74] M.L. Dertouzos, “Control robotics: the procedural control of physical processes”, proc. IFIP Congress, pp. 807-813, 1974.

[DG00] R. Devillers, J. Goossens, “Liu and Layland’s Schedulability Test Revisited,” Information Processing Letters, vol. 73(5), pp. 157-161, Mar. 2000.

[DLS97] Z. Deng, J.W.S. Liu, J. Sun, “A scheme for scheduling hard real-time applications in open systems environment”, Proc. 9th Euromicro Workshop on Real-Time systems, pp. 191-199, June 1997.

[GB95] T.M. Ghazalie, T.P. Baker, “Aperiodic servers in a deadline scheduling environment”, Real-Time Systems, Vol. 9(1), 1995.

[GG04] A. Geniet, E. Grolleau, “Minimal schedulability interval for real-time systems of periodic tasks with offset”, Theoretical Computer Science, vol. 310, pp. 117-134, 2004.

[Goo03] J. Goossens, “Scheduling of offset-free systems”, Real-Time Systems, Vol. 24(2), pp. 239-258, 2003.

[Horn74] W. Horn, “Some simple scheduling algorithms”, Naval Research Logistic Quarterly, 21, 1974.

[HT97] C.C. Han, H.Y. Tyan, “A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms”, Proc. IEEE Real-Time Systems Symp., pp. 36-45, Dec. 1997.

[HKL91] M.G. Harbour, M.H. Klein, J.P. Lehoczky, “Fixed priority scheduling of periodic tasks with varying execution priority”, Proc. IEEE Real-Time Systems Symposium, San Antonio, Texas, pp. 116-128, Dec 4-6 1991.

[Jac55] J.R. Jackson, “Scheduling a production line to minimize maximum tardiness”, Management Science Research Project 43, University of California, Los Angeles, 1955

[JP86] M. Joseph and P. Pandya, “Finding response times in real-time system”, The Computer Journal, vol. 29(5), pp. 390-395, 1986.

[JS93] K. Jeffay and D.L. Stone, “Accounting for interrupt handling costs in dynamic priority task systems", proc. IEEE Real-Time Systems Symposium, Raleigh-Durham, NC, USA, 1993

[JSM91] K. Jeffay, D.F. Stanat, C.U. Martel, “On non-preemptive scheduling of periodic and sporadic tasks”, Proc. IEEE Real-Time Systems Symposium, San Antonio, Texas, pp. 129-139, Dec 4-6 1991.

[Kai82] C. Kaiser, « Exclusion mutuelle et ordonnancement par priorité », Technique et Science Informatiques, Vol. 1, pp. 59-68, 1982.

[KCLL03] T. Kuo, L. Chang, Y. Liu, K. Lin, “Efficient on-line schedulability tests for real-time systems”, IEEE Trans. On Software Engineering, Vol. 29(8), 2003.

[KCPKH95] I.G. Kim, K.H. Choi, S.K. Park, D.Y. Kim, MP. Hong, “Real-time scheduling of tasks that contain the external blocking intervals”. *Real-Time and EmbeddedComputing Systems and Applications(RTCSA’95)*, 1995.

[KM91] T.W. Kuo, A.K. Mok, “Load adjustment in adaptive real-time systems”, Proc. IEEE Real-Time systems Symposium, 1991.

[Lab 74] J. Labetoulle, « Un algorithme optimal pour la gestion des processus en temps réel », Revue Française d'Automatique, Informatique et Recherche Opérationnelle, pp. 11-17, Fevr 1974.

[Leh90] J.P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines”, IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, USA, 1990.

[Liu00] J.W.S. Liu, “Real-time systems”, Ed. Prentice Hall, 610 p., 2000.

[LL73] C.L. Liu and J.W. Layland, “Scheduling algorithms for multiprogramming in real-time environment”, Journal of the ACM, vol. 20(1), pp. 46-61, 1973.

[LLWS07] W.C. Lu, K.J. Lin, H.W. Wei, W.K. Shih, “New schedulability conditions for real-time multiframe tasks”, *19th Euromicro Conference on Real Time Systems, (ECRTS07)*, Pisa, Italy, July 4-6 2007.

[LM80] J. Leung and M. Merrill, "A note on preemptive scheduling of periodic real-time tasks", Information Processing Letters, vol. 11(3), pp. 115-118, 1980.

[LR92] J.P. Lehoczky, S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks infixed-priority preemptive systems", Proc. Real-Time Systems Symposium, pp. 110-123, Phoenix, AZ, 2-4 Dec. 1992.

[LRKB77] J.K. Lenstra, A.H.G. Rinnooy Kan, P. Brucker, "Complexity of machine scheduling problems", Ann. Discrete Math., 1, pp. 343-362, 1977.

[LSD89] J.P. Lehoczky, L. Sha, Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behaviour", Proc. 10th Real-Time Systems Symposium, pp. 166-171, 1989.

[LSS87] J.P. Lehoczky, L. Sha, J.K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments", Proc. IEEE Real-Time systems Symposium, pp. 261-270, 1987.

[LSST91] J.P. Lehoczky, L. Sha, J.K. Strosnider, H. Tokuda, "Fixed priority scheduling theory for hard real-time systems", Foundations of Real-Time Computing: Scheduling and resource management, Kluwer Academic Publishers, pp. 1-30, 1991.

[LW82] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks", Performance Evaluation, vol. 2, pp. 237-250, 1982.

[MA98] Y. Manabe, S. Aoyagi, "A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling", Real-Time Systems, 14(2), pp. 171-181, 1998.

[MC96] A.K. Mok, D. Chen. "A multiframe model for real time tasks", proc. IEEE International Real Time System Symposium, pp. 22–29, 1996.

[MC97] A.K. Mok, D. Chen. "A multiframe model for real-time tasks", *IEE Trans. on Software Engineering*, Vol. 23(10), pp. 635-645, 1997.

[Mok83] A.K. Mok, "Fundamental design problems for the hard real-time environments", Ph.D. MIT, May 1983.

[PH98] J. Palencia Gutierrez, M.Gonzalez Harbour, "Schedulability analysis for tasks with static and dynamic offsets", Proc. 19th IEEE Real-Time System Symposium, December 1998.

[RMA] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, "A practitioner's handbook for real-time analysis: guide to rate monotonic analysis for real-time systems", 712 p., Kluwer Academic Publishers, 1994.

[RR06] F. Ridouard, P. Richard, "Worst-case analysis of feasibility tests for self-suspending tasks", Proc. Real-Time and Network Systems, RTNS'06, pp. 15-24, Poitiers, France, May 30-31st, 2006.

[RRC04] F. Ridouard, P. Richard, F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions", Proc. 25th IEEE International Real-Time Systems Symposium, pp. 47-56, December 2004.

[RRGC02] M. Richard, P. Richard, E. Grolleau, F. Cottet., "Contraintes de précédences et ordonnancement mono-processeur", Real Time and Embedded Systems, ed. Teknea, pp. 121-138, 2002.

[Ser72] O. Serlin, "Scheduling of time critical processes", proc. Spring Joint Computers Conference, pp. 925-932, 1972.

[SB96] M. Spuri, G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems", Real-Time Systems, Vol. 10(2), pp.179-210, 1996.

[Spu96] M. Spuri, "Analysis of deadline scheduled real-time systems", Research Report INRIA, 2772, Jan, 1996.

[SRL90] L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority inheritance protocols : an approach to real-time synchronization", IEEE Transactions on Computers, Vol. 39(9), pp. 1175-1185, 1990.

[Str88] J.K. Strosnider, "Highly Responsive Real-Time Token Rings", PhD thesis, Carnegie Mellon Univ., 1988.

[SSL89] B. Sprunt, L. Sha, J.P. Lehoczky, "Aperiodic task scheduling for hard real-time systems", Real-Time Systems, Vol. 1(1), pp. 27-60, 1989.

[Tin94] K. Tindell, "Addind Time-Offsets to Schedulability Analysis", Technical Report YCS 221, Dept of Computer Science, University of York, England, January 1994.

[TGC06] K. Traoré, E. Grolleau, F. Cottet, Characterization and analysis of tasks with offsets : monotonic transactions, Real-Time Computing Systems and Applications, RTCSA'06, Sydney, Australia, Augt 16-18, 2006.

[TN04] J. Mäki-Turja, M. Nolin, "Faster response time analysis of tasks with offsets", Proc. 10th IEEE Real-Time Technology and Applications Symposium (RTAS), May 2004.

[TN05] J. Mäki-Turja, M. Nolin, "Fast and tight response-times for tasks with offsets", 17th EUROMICRO Conference on Real-Time Systems, Palma de Mallorca Spain, July 2005.