Weakly Hard Real-Time Systems

G. Bernat, A. Burns,
University of York
Real-Time Systems Research Group
Department of Computer Science
YO1 5DD - York (England)
{bernat,burns}@cs.york.ac.uk

Albert Llamosí
Universitat de les Illes Balears
Dept. Matemàtiques i Informàtica
Cra. Valldemossa, Km 7.5
07010 - Palma (Spain)
llamosi@clust.uib.es

September 1, 1999

Abstract

In a hard real-time system it is assumed that no deadline is missed, whereas in a soft or firm real-time system deadlines can be missed, although, this usually happens in a non-predictable way. However, most hard real-time systems could miss some deadlines provided that it happens in a known and predictable way. Also adding predictability on the pattern of missed deadlines for soft and firm real-time systems is desirable, for instance to guarantee levels of quality of service.

We introduce the concept of weakly hard real-time systems to model real-time systems that can tolerate a clearly specified degree of missed deadlines. For this purpose we introduce several temporal constraints based on determining a maximum number of deadlines that can be missed during a window of time (a given number of invocations). Weakly hard real-time systems can be guaranteed off-line with an extension of traditional schedulability analysis. Also, new scheduling algorithm can be defined that specifically address weakly hard constraints. This report studies in detail different constraints, their properties and relationships and introduces new scheduling paradigms based on this concept.

Keywords: *Real-time systems, Specification, Temporal Constraints, Scheduling algorithms, Schedulability analysis, Transient overloads.*

1 Introduction

Real-time system are those ones in which the temporal aspects of their behaviour are part of their specification. The correctness of the system depends not only on the logical results of the computation, but also on the time at which the results are produced. This is due to the fact that these systems interact with their environment, which also changes in time. Real-time systems can be found in many different areas from control systems to multimedia video conferencing systems.

Most real-time programs are not intended to terminate, but to loop endlessly examining input data and reacting accordingly. Therefore, the specification of real-time systems relies on this cyclic nature and is expressed in terms of operations – also called tasks or jobs – that have to be performed repetitively and finish within a given deadline.

Traditionally, real-time systems (or tasks) are classified as being *hard* or *soft*. For hard real-time tasks, it is imperative that no deadline be missed, while for soft tasks, it is acceptable to miss some of them occasionally. It is still valuable for the system to finish the task, even it is late. There is another class of tasks called *Firm* tasks. Firm tasks are also allowed to miss some of their deadlines, however as opposed to soft tasks, producing the result after the deadline does not add value to the system.

In practical engineering contexts, the occasional loss of some deadline can be tolerated. This is either because the consequences of the loss can be negligible (due to the fact that the robustness of the involved control algorithms imply the ability to react properly at the next invocation step without serious consequences), or because the effect on a single deadline missed is not noticeable by a user, as it would be the case of a missed audio packed in a video conferencing system. In this case the quality of the service (QoS) provided would be lower.

Nevertheless, the term *occasional* is so ambiguous that it has no practical meaning for a specification. The extent to which a system may tolerate missed deadlines has to be stated precisely. However, the way that these missed and met deadlines are distributed is important. Some systems, for instance, are very sensitive to the consecutiveness of the missed deadlines. A missed deadline may correspond to a gap in the output signal, and if these gaps occur consecutively, the quality of the output is lower than if those gaps are non-consecutive.

Not having to meet *every* deadline allows the size of the system's resources to be smaller than it would be for meeting all of them. This permits the creation of simpler and most cost effective systems that make better use of the available resources while guaranteeing a good level of service.

The main contribution of this article is introducing temporal constraints for specifying tasks that can tolerate a well defined degree of missed deadlines and to present and study their properties and relationships.

1.1 Weakly-hard real-time systems

We present an appropriate conceptual framework for specifying real-time systems that can tolerate occasional losses of deadlines. We call these type of systems *weakly hard real-time system*. Formally:

Definition 1 *(Weakly hard) A weakly hard real-time system is a system in which the distribution of its met and missed deadlines during a window of time w is precisely bounded.*

In particular, systems that must meet all of their deadlines are a particular case of our definition and will be referred to as *strongly hard real-time systems* whenever the distinction is relevant.

This tolerance is established by specifying, for each task, the bounds for missed deadlines allowed in every window of m consecutive invocations. Which also corresponds to a window of $w = Tm$ units of time, where T is the period of the task.

By considering the number n of missed or met deadlines over a window on m consecutive invocations and by considering whether the consecutiveness of such missed deadlines is important or not, we introduce four different temporal constraints for weakly hard real-time systems.

The simplest one is the $\binom{n}{m}$ (read any n in m). We say that a task meets *any* n in m deadlines if in any window of m consecutive invocations of the task, there are at least n invocations in any order that meet the deadline. This constraint and the other three: $\overline{\binom{n}{m}}$, $\left\langle {n \atop m} \right\rangle$ and $\overline{\left\langle {n \atop m} \right\rangle}$ are formally defined in section 3.3.

Please note that the tolerance to missed deadlines in a real-time system can not be adequately specified with a single parameter, for example with the percentage of the number of deadlines to be met or missed (although it is a common practice). This is because, in general, a requirement like less than 10% of deadlines can be missed only represents average information over a large period of time. For example, it may mean that one deadline is missed every 10 task invocations or it may mean that 100 deadlines may be missed followed by 1000 met, which is not exactly the same. To capture this situation another parameter describing the window of time within which the number of deadlines must hold should be specified.

A missed deadline may correspond to any of the following scenarios:

- Delayed finalisation: the task runs until completion even though it finishes after the deadline
- Abortion: the task is terminated before finishing its computation either because it will not end its computation by the deadline or because another (more important) task needs the resource. This also requires some sort of cleaning up mechanism, last wishes, etc.
- Rejection: The task invocation is not accepted into the system.

- Skip: The task is not invoked and the whole invocation is not executed.

In the rest of the discussion, when we refer to a missed deadline we generally mean that the task suffers a delayed finalisation. Following task invocations may also miss the deadline or finish on time too. If only one deadline is missed in a row, of say, n consecutive invocations, it means that there is an upper bound on the maximum response time of the task. The response time of the task is bounded by twice the period of the task (in fact less than that because in the time interval of 2 periods, two tasks have to finish).

1.2 Motivation examples

Before going further in the definition and analysis of weakly hard real-time systems it is important to argue what is the effect of missing a clearly specified number of deadlines, and therefore why it should happen in a predictable way. This leads us to questioning traditional assumptions and hypothesis generally put forward by real-time systems researchers. Different studies have addressed the topic under different assumptions as:

- Deadlines can not be missed,
- Task periods are fixed and can not be modified,
- Deadline equals period ($D = T$),
- Average response time of soft tasks has to be minimised.

Period and deadline values of a real-time task are defined at the specification. These values can be either the result of a deep analysis of the system under control, as it is the case of a computer driven automatic control system, they may only describe a desired sampling rate, as in the monitoring of a non-critical signal, or they may derive from perceived quality of service (QoS) requirements as in a digital telephony system.

In order to argue in detail about these common assumptions and to analyse what are the effects of missing deadlines in a real-time system, we need to study in detail the semantics associated to a task and where the period and deadline parameters come from. We do this by studying three typical real-time scenarios, real-time control systems, real-time monitoring and multimedia systems.

1.2.1 Computer driven automatic control systems

In a computer driven automatic control system, tasks periodically sample input signals, perform some computation according to some control algorithm and send the commands to some actuators. Control systems theory allows the analysis of the system according to its physical characteristics and the derivation of the parameters that guarantee both the system to be stable and show a steady state error within the tolerance bounds. Among these parameters, the sampling rate plays an essential role.

A feedback loop is implemented as an Ada task as follows[1].

```
task Control; task body control is
  sampling_period  : Time_Span; -- period of time
  deadline         : Time_Span;
  s                : Sensors;   -- sensor information
  a                : Actuators; -- actuators information
  state            : Controller_States; --
  next             : Time;      -- time instant

begin
  sampling_period := milliseconds(200);-- say 5Hz
  deadline        := milliseconds( 40);-- say 40 ms
```

[1]types `Time_Span` and `Time` and function `Clock` are defined in the package `Ada.Real_Time`.

```
  state           := Initialise_State ;-- set initial state
                                      -- of the controller
  next            := Clock;           -- start now!

  loop
    delay until next; -- sleep until current time is
                      -- "at least" equal to "next"

    Read_Sensors(s);
    Compute_Action(state, s, a);
      -- compute output in terms of current state and
      -- sensor information. Also update state.
    Output_Actuators(a);

    -- if at this point (Clock - next) > deadline then
    -- deadline is missed

    next:= next + sampling_period; -- next sampling point
  end loop;
end control_task;
```

An important parameter of the analysis is the sampling period T (or sampling frequency $\omega = \frac{1}{T}$) [1]. The sampling period is then used as the period of the controller task which implements the control algorithm. There are some rules for defining the adequate sampling period. It is known that, in order to reconstruct a signal which has a maximum frequency ω, it is required a sampling frequency ω_s at least $\omega_s \geq 2\omega$. This result is known as the Nyquist theorem or Shannon theorem. As it is not generally necessary to reconstruct the original signal, and it is also not always possible to determine the exact maximum frequency, another measure is used. The bandwidth of a control system is defined as the frequency band between $\omega = 0$ and some maximum frequency ω_b. It gives a precise indication of the range of frequencies to which a system responds. In most cases, ω_b is considered to be the *half-power point* of the frequency response function. This value is generally chosen because it represents a good general value where large-frequency attenuation begins (this value also corresponds to the closed-loop 3dB bandwidth).

Control theory assumes sampling and output to occur *at exact* time instants $t_0 + kT$ ($k \in \mathbb{N}$). In fact, due to different delays (A/D, D/A conversion, sampling jitter, computation time and interference from other tasks), sampling and output occurs at any time between the activation time and the activation time plus the associated deadline. This effect produces an error similar to a quantization error (which can be also included in the analysis of the system [2]). To overcome the introduced errors associated to digital control, figures quoted in the literature [3, 4] suggest that, under favourable circumstances, control system sampling rates can be as low as 5 to 10 (may be 20, or even 40) times the bandwidth frequency ω_b (*i.e.* oversampling by a factor of 5 to 40). Another used heuristic for determining the minimum sampling rate is to guarantee that there are between 4 to 10 [1, 5] samples during the rise time of the response of the system to a step input. Also note that very large oversampling factors have undesirable effects in the system due to very small differences between one sample and the next.

The assumption that $D = T$ is unadequate for digital control. Automatic control systems require that $D \ll T$ so that both sampling and actuation occur at intervals as regular as possible [2]. Again, oversampling reduces the associated error of assuming $T = D$ to a negligible level. The assumption of $T = D$ came from the fact that, under this hypothesis, the analysis of the system made up of periodic tasks could be performed (*i.e.* the utilisation based Liu and Layland check [6]).

An example of the application of inadequate models in automatic control is the work done by Klein *et al.* in [7] in which they describe an industrial robot control system which is designed and analysed using "rate monotonic theory". In their work, servo loops are given deadlines equal to periods. These servo loops do not comply with normal timing requirements of sampled data systems as a result of the timing behaviour large deadlines allow. As Töngren indicates in [5], timing requirements appear to be postulated rather than based on control system requirements.

1.2.2 Monitoring system

Another example of a real-time task is a monitor. A task periodically performs a set of monitoring actions. The sampling period may be carefully computed (and probably overestimated) or decided as a rule of thumb. Again, missing an occasional deadline means that the action from monitoring will be delayed by some bounded period of time. Provided that the effect of such delay can be tolerated deadlines can be missed.

As in the previous section, periods are generally defined as a rough estimation of the desired sampling rate, therefore they may be reduced if design convenience require to.

1.2.3 Multimedia systems

A multimedia system is one which exhibits different media simultaneously. From the real-time point of view, we are interested in media which changes as a function of time, for instance digital audio and video. These systems are generally considered as soft systems as missing a deadline has no catastrophic consequences, unless very high fidelity has to be certified.

Consider, for example, a digital video system. The system decodes some video streams and plays the frames at some rate (say, 20 frames per second). If it does not have enough time to finish decoding a frame it is either skipped or only partially displayed. Not displaying a frame on time is considered a deadline missed.

In this case, a deadline missed results in a lower quality of the service provided by the computer. If it is not known how deadlines missed are distributed over time, several deadlines missed may occur in a row that manifest as having the screen frozen for short while.

In these systems, measures like setting the desired percentage of deadlines met are generally provided. These measures are inadequate as for example missing two consecutive deadlines has a different visible effect than missing one, meeting the next and missing the third one. It is clear that the adequate measure should capture the number of deadlines missed during a period of time, for example that no more than 5 deadlines can be missed every second (every 20 frames) and with a maximum number of consecutive deadlines missed, for example than no more than two missed deadlines in a row can be tolerated).

Another common problem in systems that can tolerate missed deadlines is related to using some sort of buffering. Assume a buffer of some size, and producer and consumer tasks that periodically add and remove elements of the buffer. If the consumer task misses its deadline it may mean that the buffer has grown in size temporarily. The buffer will not overflow if there is a guaranteed maximum number of deadlines missed in a row.

As a final note, real-time scheduling algorithms that schedule a mixture of hard and soft tasks usually partition the task in these two classes: hard and soft. As it can not be guaranteed how many deadlines soft tasks may miss, the criteria commonly used for comparing their performance is by minimising the average response time of soft (or firm) tasks, generally ignoring completely that those soft (firm) tasks do have a deadline as well. This actually leads to the situation in which hard tasks suffer high delays and high output jitter in order to give preference to soft (firm) tasks. This is an unfair mechanism because soft (firm) tasks are given preference over hard ones even though hard tasks may have shorter deadlines. A more sensible mechanism is to consider all tasks in the system to be weakly hard (some of them would still be strongly hard) and where firm and soft tasks have weakly hard constraints. A guaranteed scheduling algorithm may schedule all tasks according to their weakly hard constraints with the secondary objective, if required, of minimising the average response time of the tasks (or a subset of them).

The cases analysed in this section show that hard real-time systems are not that hard because some degree of missed deadlines can be tolerated by the algorithms themselves. On the other hand soft (firm) real-time systems are not that soft as it is generally required to specify upper bounds on the number and pattern of deadlines missed during a period of time. The notion of weakly hard real-time system captures all of these scenarios in a unified way.

1.3 Scope of the research

The study of weakly hard real-time systems can be done from three points of view: the specification, the implementation and the analysis.

The specification relates to setting weakly hard constraints on tasks which describe their desired or required temporal properties. For this purpose it is necessary to study in detail the different temporal constraints and how they relate to each other, for example, by identifying when a weakly hard constraint is harder to satisfy than another. The semantics associated to setting this weakly hard constraints are also an important issue which depends on the particular application domain. This is one of the main contributions of this article and it is described in detail in section 3.

At the implementation, the specified tasks are mapped into implementation threads which will be scheduled with a given scheduling algorithm (or even implemented as a cyclic executive). The scheduling algorithm can be any traditional fixed priority or dynamic priority algorithm or be based on weakly hard constraints like the enhanced dual priority scheduling [8, 9], skip over [10], etc. The issue of scheduling algorithms which are based on weakly hard constraints is revised in section 5.

The analysis allows to determine whether weakly hard constraints will be satisfied. This depends on the scheduling algorithm itself. There are two main scheduling strategies for the implementation of a real-time system. The scheduling strategy can be *guaranteed* or *best-effort*. In guaranteed systems, an off-line analysis technique exists that actually assures that the temporal constraints will be always satisfied. On the other hand, a best-effort scheduling is not based on obtaining a-priori guarantees that the constraints are going to be satisfied. Instead, it is based on maximising (or minimising) some criteria, for example maximising the number of weakly hard constraints satisfied.

For guaranteed systems, off-line schedulability checks are required to determine the satisfiability of the system. Section 4 describes such a check for a fixed priority scheduler. For best-effort scheduling algorithms simulation is most often applied for obtaining average properties of the system's execution.

1.4 Previous Work

There have been some previous approaches to the specification and design of real-time systems that can tolerate occasional losses of deadlines.

Hamdaoui and Ramanathan in [11] present the notion of (m, k)-firm deadlines to specify tasks that are desired to meet m deadlines in any k consecutive invocations. They use this concept in the context of scheduling messages in a communication system. They present a scheduling algorithm based on raising the priority of a class of messages if it is likely that m messages of the class out of k consecutive messages are not going to meet their deadlines. This approach has, on average, a better behaviour than a simple EDF approach. This approach is a best-effort scheduling algorithm. It does not provide any guarantee on the number of deadlines a task may miss. They also assume that all tasks have the same m, k parameters. They use the concept of (m, k)-firm deadlines as a desired level of service as opposed to a guaranteed level of service.

Another approach is the introduction by Koren and Shasha [12] of the *skip factor*, s. If a task has a skip factor of s it will have one invocation skipped out of s. They introduce the D-over dynamic priority scheduling algorithm which is based on deciding which task invocations to skip. Later in [10] the same authors introduce two scheduling algorithms for dealing with skips: the Red Tasks Only (RTO) and Blue When Possible (BWP). They also show that making optimal use of skips is an NP-hard problem.

Koren and Shasha deal with overloaded systems. They reduce the overload by skipping some task invocations. A selected number of task invocations are skipped, even though the tasks (and lower priority tasks) could meet the deadline if they were scheduled. Note that this is essentially different from guaranteeing that a task will only miss one deadline in every s consecutive invocations. This work has been followed by Caccamo and Buttazzo in [13] where they schedule hybrid task sets consisting of skippable periodic and soft tasks. They present a scheduling algorithm based on EDF which exploits skips to minimise the average response time of the soft tasks.

It is known that under overload conditions, deadline based scheduling systems perform badly due to the domino effect [14]. This has lead to the definition of scheduling strategies which rely on the value of the task rather than the deadline under overload conditions. For example, Buttazzo *et al.* in [15] studies the value based scheduling under overloads.

Other works are just based on mechanisms that integrate hard and firm tasks and that are able to deal with transient overloads which implies that some deadlines (of the firm tasks) will be missed. The work of Spuri *et al.* [16] is an example. However, even though some invocations are rejected, there is no mechanism to quantify how many of them are actually being lost, neither to distribute the number of lost deadlines among the tasks.

In [17] Wedde *et al.* presents a distributed operating system, called Melody, in which it introduces the notion of sensitivity and criticality as a measure of the number of consecutive deadlines a task may miss based on the history of missed deadlines of the task and associated scheduling policies.

There has been increasing interest in the real-time community for quality of service (QoS) based systems. QoS is generally used in multimedia systems. Different strategies allow to offer a negotiated level of service to each of the tasks in the system. In most of the cases, current tasks compete for the available resources and it is necessary a mechanism to distribute it between them. For example, see [18] for a review of related work on QoS. In these systems, it is generally assumed that deadlines can be missed and therefore decrease the quality of the service (as in [19])

These works have introduced the idea of systems which tolerate some missed deadlines but their contributions do not separate the specification of such constraints from the analysis or scheduling algorithm. This makes it difficult to generalise their ideas and therefore to be able to apply their work in a more general framework.

We present a general framework for specifying real-time systems that tolerate a degree of missed deadlines. We later show that all of these approaches shown above are particular cases of our definitions. We also generalise their scheduling models to all of the weakly hard constraints.

1.5 Structure of the article

The main emphasis of the article is on the specification of weakly hard constraints and on the study of the properties and relationships of tasks with such constraints. However, we also discuss the issues of schedulability analysis and the implementation of scheduling algorithms based on weakly hard constraints. Therefore the article is organised as follows. The next section introduces the process model used for specifying the constraints. We first introduce a pure periodic model and then complete it with the specification of sporadic tasks. After this section, the formal definitions of weakly-hard constraints are introduced and their properties and relationships studied in detail. We then present the schedulability checks for two scheduling models, fixed priority and dynamic priority (EDF). Finally, we introduce on-line scheduling algorithms based on the weakly-hard specifications.

2 Process Model

In the real world things happen in parallel and a system that interacts with this real world has to mimic this parallel behaviour. This is usually done by building the system as a set of cooperating sequential operations that interact with each other and with the environment, and that execute continuously. Each of these operations models a small part of the system and can be either *periodic* or *non-periodic*.

A periodic operation consists of a computation that is executed repeatedly in a regular and cyclic pattern. The duration interval between the start of one execution, the invocation time, and the next one is a constant, called the period (denoted by T). Non-periodic operations are usually executed in response to asynchronous events. For non-periodic operations, a measure of the arrival rate is usually provided. The most commonly used is the minimum separation between two consecutive invocations (also denoted by T).

These operations are usually called tasks or jobs. This term does not have to be confused with the same term when it is used in the implementation. A task in a specification context means a goal that must be achieved periodically. A task in an implementation context means an execution thread. Although a specification task can be mapped on an implementation thread, other alternatives exist as it is the case of cyclic executives. In this work, a ***task*** refers to the operation concept when in the context of a specification and to the thread concept in the context of the implementation.

Each task invocation should have finished before of after the deadline. For weakly hard systems, the notion of deadline is much more relaxed than in traditional hard systems. This parameter may describe a maximum upper bound on the completion time of the task as it is the case of strongly hard tasks, or it may have a more relaxed meaning, for instance a period of time before which the task is expected to finish.

2.1 Periodic tasks

The basic process model, from the specification point of view, consists of N periodic tasks, where each task , τ_i is characterised by the period, deadline and a weakly hard constraint λ which describes the tolerance to missed deadlines. The weakly hard constraints are introduced formally in section 3[2].

$$\Pi = \{\tau_i = (T_i, D_i, \lambda_i)\}_{1 \leq i \leq N}$$

All periodic tasks are assumed to be released together at $t = 0$. This time instant is also called the *critical instant*. In this case, the tasks will be released together again at the least common multiple of the periods of the tasks, this period of time is called the *hyperperiod* and it is denoted by $\mathbf{H}$:

$$\mathbf{H} = \mathsf{lcm}\{T_i \mid \tau_i \in \Pi\}$$

We denote by $A_i = \frac{\mathbf{H}}{T_i}$ the number of activations (or invocations) of a task τ_i in the hyperperiod. Under fixed priority scheduling, a task can suffer interference from higher priority tasks and potentially a blocking due to lower priority tasks. We can, therefore define the *hyperperiod at level* i as the hyperperiod only considering tasks of equal or higher priority than task τ_i. We denote the hyperperiod at level i by h_i and the number of invocations of a task in its hyperperiod by $a_i = \frac{h_i}{T_i}$.

In some circumstances, however, the starting times of some of the tasks may not occur at $t = 0$. In this case we also consider that a task may have an *offset*, denoted by O_i which is the time elapsed since the first invocation of that task.

For a given task τ_i, we denote by $S_i(k),\ k \geq 1$, as the invocation time of the k-th release of the task.

$$S_i(k) = (k-1)T_i + O_i \tag{1}$$

We also use R_i to denote the worst case response time of the task. We study the behaviour of the tasks at different invocations, for this purpose we introduce $F_i(k)$ as an upper bound on the worst case finalisation time of task τ_i at invocation k. This values could be computed after a scheduling strategy and worst case computation times of the tasks are obtained.

We also define $R_i(k)$ as the worst case response time of the task at invocation k. It is given by:

$$R_i(k) = F_i(k) - S_i(k)$$

In general $R_i(k) \neq R_i(l)$ for $k \neq l$ because the amount of interference a task may suffer from other tasks depends on the phasing with other tasks which is different at each invocation.

[2]Note that we do not include the worst case computation time C_i in this process model as it is not necessary from the requirements point of view. It it will be only necessary for checking the schedulability of the system.

2.2 Non-periodic tasks: repeating and isolated tasks

We also consider that the system can be made up of non periodic tasks. Traditionally, hard non-periodic tasks are called *sporadic* while soft non-periodic tasks are usually called *aperiodic*[3].

In our context we need to further distinguish between two classes of non-periodic tasks: *repeating* and *isolated*.

On the one hand, repeating tasks are non-periodic tasks which are continuously invoked, although this does not happen at fixed periods of time. Over long periods of time these tasks behave almost as periodic. Repeating tasks model quasi-periodic activity, as it would be the case of signals generated by an angular encoder in which the rate of arrivals of signals would depend on the angular velocity which changes in time. It could also model messages from a network generated by a periodic task. In average messages arrive periodically, but with relative large jitter between them. Another example could be a video streaming system in which video and audio packets are sent through a network. The receiver node sees a stream of signals which are continuously arriving but can not be modeled as periodic due to the different sources of delay.

On the other hand, isolated sporadics are tasks that, even though they may have a minimum interarrival time for analysis purposes, they are not expected to be invoked consecutively nor it has sense to consider the behaviour of some task invocations together. Isolated tasks can be seen as the tasks attached to alarm or rare events in the system.

Periodic and repeating tasks may have weakly hard constraints assigned to them, however it has no sense to set weakly hard constraints to isolated tasks.

In the rest of the article, τ denotes a periodic task, whereas σ denotes a non-periodic task. A task which may be either periodic or non-periodic is denoted by ρ. For repeating tasks, the vector $R_i(k)$ can not be computed in the same way as for periodic tasks because arrival instants are not fixed. The issue of using repeating tasks is discussed in section 3.4.

3 Specification of weakly hard real-time systems

This section studies the way a bounded distribution of met and lost deadlines can be specified, as well as the relationships and properties of the systems so specified.

We first present a task set as an example to illustrate the distribution of met and missed deadlines under fixed priority scheduling. After this we introduce the concept of μ-pattern and based on this we define the weakly hard constraints.

3.1 Example

The following case study, based on the one described by Locke *et al.* in [20] will be useful to illustrate our concepts. The parameters of the task set are shown in table 1.

The reader should realise that the table also includes the columns C_i, holding the worst case computation time of each cycle of the tasks, and R_i, which holds the worst case response time of the tasks assuming fixed priority scheduling. Tasks are sorted in deadline monotonic priority ordering.

Under fixed priority scheduling, the traditional worst case response time schedulability analysis would conclude that the system is not schedulable because τ_9 and τ_{10} miss the deadline in the worst case. However, even though tasks τ_9 and τ_{10} miss their deadlines at the critical instant, they meet almost all deadlines.

Figure 1 shows $R_9(k)$, the worst case response times at each invocation k within the hyperperiod at priority level 9 (the way $R_i(k)$ is discussed later in section 4). Each vertical line represents the worst case response time of the task at invocation k. The deadline of the task is $D_9 = 80$. The hyperperiod is $\mathbf{H} = 118000$, whereas $h_9 = 23600$. This means that the task is invoked $a_9 = 295$ times within its hyperperiod at level 9. The following table shows the exact distribution of the

[3]This terminology is unfortunate and confusing as these names do not represent the notion behind the tasks. However they are accepted terms and therefore we will stick to them.

Table 1: Example task set. Tasks τ_9 and τ_{10} miss their deadlines in the worst case

τ_i	T_i	D_i	C_i	R_i
τ_1	200	5	4	4
τ_2	25	25	2	6
τ_3	25	25	6	12
τ_4	40	40	3	15
τ_5	50	50	3	18
τ_6	50	50	5	23
τ_7	59	59	8	39
τ_8	80	80	9	75
τ_9^*	80	80	2	97
τ_{10}^*	100	100	5	139
τ_{11}	200	200	1	145
τ_{12}	200	200	3	148
τ_{13}	200	200	1	149
τ_{14}	200	200	1	150
τ_{15}	200	200	3	199
τ_{16}	1000	1000	1	200
τ_{17}	1000	1000	1	295

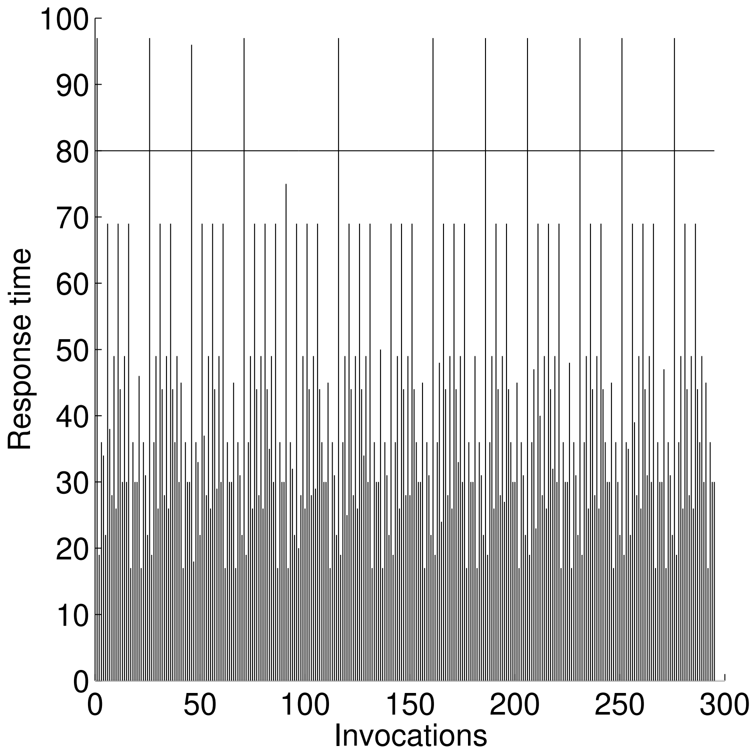

Figure 1: Response times of task τ_9 having $D_9 = 80$ for all invocations in the hyperperiod.

missed and met deadlines for invocations 1 to 295 of task τ_9. The pattern is repeated cyclically. A 0 represents a deadline missed and a 1 a deadline met.

```
  1- 40: 01111 11111  11111 11111  11111 01111  11111 11111
 41- 80: 11111 01111  11111 11111  11111 11111  01111 11111
 81-120: 11111 11111  11111 11111  11111 11111  11111 01111
121-160: 11111 11111  11111 11111  11111 11111  11111 11111
161-200: 01111 11111  11111 11111  11111 01111  11111 11111
201-240: 11111 01111  11111 11111  11111 11111  01111 11111
241-280: 11111 11111  01111 11111  11111 11111  11111 01111
281-295: 11111 11111  11111
```

The task does not meet the deadline at the critical instant at t_0. However, it may miss it at most 11 times during its hyperperiod, (11 times every 295 invocations), also note that the invocations when the task may miss its deadline are spread out regularly along the hyperperiod. The minimum separation between two consecutive deadlines missed is 20 invocations. This is because the task only misses the deadline when it overlaps with the invocation of a subset of higher priority tasks that produce enough interference to make the task miss its deadline. Note that this situation is very common.

This example illustrates some typical properties of task executions: even though some tasks may miss their deadline, the response times are bounded, additionally, the points in time when those deadlines may be missed are spread out and occur at known points in time.

We now formalise the the string that characterises the execution of the tasks and based on this we introduce the different temporal constraints.

3.2 μ-patterns

The way a task achieves and misses its deadlines is characterised by a binary sequence that we call the task's μ-pattern.

Definition 2 *The tasks μ-pattern is the worst case pattern of met and lost deadlines of task τ_i. It is denoted by μ_i and it is defined as:*

$$\mu_i(k) = \begin{cases} 1 & \textit{if } R_i(k) \leq D_i \\ 0 & \textit{otherwise} \end{cases} \quad k \geq 1$$

The analysis of μ-patterns will require the use of some elementary concepts on strings on the alphabet $\Sigma = \{0, 1\}$. As usual, we will denote as Σ^* the set of words over the alphabet Σ. We will call *sequences* the elements of $\Sigma^* \cup \Sigma^\infty$. Please note that a μ-pattern is a sequence.

Given a sequence α, the length of the sequence, $|\alpha|$, is the number of symbols the sequence has. α_i is the i-th symbol of α starting with $i = 1$. A slice or subsequence $\alpha(i..j)$ is the word obtained from α by getting the symbols starting at position i and ending at position j (both included). It is only defined for $1 \leq i \leq i - 1 + j$ and $j \leq |\alpha|$. Also $\alpha(i, p)$ denotes the window of length p starting at i, $\alpha(i..i + p - 1)$. A negative value of p in $\alpha(i, -p)$ denotes a string that *ends* at i and has length p. This corresponds to $\alpha(i - p + 1..i)$. The set of slices of length p of a given sequence or word α is denoted by $W^p(\alpha)$:

$$W^p(\alpha) = \{\alpha(i, p)\} \quad 1 \leq i \leq |\alpha| - p + 1 \tag{2}$$

3.3 Weakly hard temporal constraints

Some systems are sensitive to the consecutiveness of lost deadlines while others are only sensitive to the number of deadlines missed. For each one, we have considered the points of view of both missed and met deadlines. This gives us four types of constraints:

Definition 3 *A task τ "meets any n in m deadlines" ($m \geq 1$ and $0 \leq n \leq m$), and it is denoted by $\tau \vdash \binom{n}{m}$, if in any window of m consecutive invocations of the task, there are at least n invocations in any order that meet the deadline.*

Formally, the task satisfies $\binom{n}{m}$ if its μ-pattern satisfies that

$$\forall \alpha \in W^m(\mu),\ \alpha \text{ has at least } n \text{ 1's}$$

Definition 4 *A task τ "meets row n in m deadlines" ($m \geq 1$ and $0 \leq n \leq m$), and it is denoted by $\tau \vdash \left\langle {n \atop m} \right\rangle$, if in any window of m consecutive invocations of the task, there are at least n consecutive invocations that meet the deadline.*

Formally, the task satisfies $\left\langle {n \atop m} \right\rangle$ if its μ-pattern satisfies that

$$\forall \alpha \in W^m(\mu),\ \text{the word } 1^n \text{ is a substring of } \alpha$$

Definition 5 *A task τ "misses any n in m deadlines" ($m \geq 1$ and $0 \leq n \leq m$), and it is denoted by $\tau \vdash \overline{\binom{n}{m}}$, if in any window of m consecutive invocations of the task, no more than n deadlines are missed.*

Formally, the task satisfies $\overline{\binom{n}{m}}$ if its μ-pattern satisfies that

$$\forall \alpha \in W^m(\mu),\ \alpha \text{ has at most } n \text{ 0's}$$

Definition 6 *A task τ "misses row n in m deadlines" ($m \geq 1$ and $0 \leq n \leq m$), and it is denoted by $\tau \vdash \overline{\left\langle {n \atop m} \right\rangle}$, if in any window of m consecutive invocations of the task, there are no n consecutive invocations that miss the deadline.*

Formally, the task satisfies $\overline{\left\langle {n \atop m} \right\rangle}$ if its μ-pattern satisfies that

$$\forall \alpha \in W^m(\mu),\ \text{the word } 0^n \text{ is not a substring of } \alpha$$

Realize that for the $\overline{\left\langle {n \atop m} \right\rangle}$ constraint, the parameter m is not required. Therefore, we will write $\overline{\langle n \rangle}$.

Please note that it is required the constraint to be satisfied *for all subsequences* of length m.

If the sequence has length shorter than $\lambda.m$, then we say the the satisfiability is *undefined* (more details are given in next section).

In this context, we will use λ to denote a constraint and Γ as the set of all possible temporal constraints of these four types. We will also refer to the n and m parameter of a constraint λ as $\lambda.n$ and $\lambda.m$. We also call the $\lambda.m$ the *window* of the constraint. The four constraints are summarised in table 2.

Table 2: Weakly hard constraints

	Met deadlines	Missed deadlines
Any order	$\binom{n}{m}$	$\overline{\binom{n}{m}}$
Consecutive	$\left\langle {n \atop m} \right\rangle$	$\overline{\langle n \rangle}$

These constraints model other notations introduced previously by other researches. For instance, the notion of (m, k)-firm deadlines [11] maps directly to the $\binom{m}{k}$ constraint. The skip factor s [12] maps to the $\overline{\binom{1}{s+1}}$ or $\binom{s}{s+1}$.

3.4 Weakly hard constraints for repeating tasks

Up to now, we have only considered weakly hard constraints for periodic tasks. However, these constraints can not be applied directly to non-periodic tasks. There are essential differences between periodic tasks and sporadic tasks. For example, consider a system in which a sporadic (isolated) task misses the deadline if it is activated at the critical instant. In the worst case, this sporadic task may never meet a deadline as it might arrive at the multiples of the hyperperiod. Repeating sporadic tasks do not show this problem. For repeating tasks, the definition of weakly hard constraint has to be reformulated to specify the exact window of time in which the constraint has to hold.

For periodic tasks, we have considered the satisfiability of a given number of deadlines over a number of task invocations. That is, for a window of Tm time units. For repeating sporadics, as the arrival instants of the tasks are not known, we can not specify the size of the window as the number of invocations. Instead it is required to specify a window of time in which the constraint has to be checked. This window of time has to be large enough to have, during normal operation at least m invocations of the task. This leads us to rewrite the weakly hard constraint for repeating tasks as follows:

Definition 7 *(Weakly hard constraint for repeating tasks) Given a period of time w, and a weakly hard constraint λ, we denote by λ_w a weakly hard constraint for a window time w. A given task ρ satisfies λ_w, and it is denoted by $\rho \vdash \lambda_w$ if all the possible μ-patterns of task invocations within a window w do satisfy λ.*

Note that for periodic tasks, this definition is equivalent to the definitions 3 to 6 by considering $w = Tm$.

For a given time t, we consider the window of time $(t - w, t]$, and we obtain the slice of the μ-pattern that corresponds to the task invocations within this interval. Therefore, the sporadic task will satisfy the constraint if for all possible time instants t the corresponding slice does satisfy the constraint. Fortunately, it is not required to check all possible time instants, it is only required to check the time instants that correspond to the invocation or finalisation of an sporadic task. If μ_i is the μ-pattern of a repeating task ρ_i, where $\mu_i(k)$ corresponds to the k invocation, then at a given time t, the slice, denoted by $\mu_i^w(t)$ is given by all invocations of the task that overlap with the window w:

$$\mu_i^w(t) = \mu_i(l..r) \tag{3}$$

where $l = \min\{s \mid F_i(s) \geq t - w\}$ and $r = \max\{s \mid S_i(s) \leq t\}$. Therefore, the satisfiability of the constraint can be written as:

$$\rho \vdash \lambda_w \quad \Leftrightarrow \quad \forall t,\ \rho \vdash \mu^w(t)$$

Please note that it is not required to evaluate all time instants t, in fact, only a subset of time points of interest are required to determine the satisfiability. These correspond to the invocation and finalisation points of all task instances. for a given task ρ_i, we the set of points of interest is

$$\{S_i(k)\}_{k>0} \quad \cup \quad \{F_i(k)\}_{k>0}$$

A natural question arises as whether the constraint is satisfied when the length of the slice is shorter than $\lambda.m$. This situation corresponds to the case when the task has been invoked less than $\lambda.m$ times in $[t - w, t]$.

Under this situation there are not enough symbols in the slice and we would say that the constraint is *undefined*. However this is a *lazy* evaluation because with fewer than m symbols in the μ-pattern it may be possible to determine, assuming some future behavior, whether the constraint will be satisfied. We call this evaluation an *eager* evaluation.

Given a constraint λ_w (let $m = \lambda_w.m$), and μ-pattern μ of a repeating task. If t is such that $p = \mu^w(t)$ where $p < m$ then :

- The constraint is *satisfied* at t if even assuming that the next $m-p$ deadlines are missed then the constraint is satisfied:

$$\mu^w(t)\cdot 0^{m-p} \vdash \lambda \;\Rightarrow\; \mu^w(t) \stackrel{\text{def}}{\vdash} \lambda_w$$

- The constraint is *unsatisfied* if even assuming that the next $m-p$ deadlines are met, the constraint will still be missed:

$$\mu^w(t)\cdot 1^{m-p} \not\vdash \lambda \;\Rightarrow\; \mu^w(t) \stackrel{\text{def}}{\not\vdash} \lambda_w$$

- If it can not be determined whether the constraint is satisfied or unsatisfied, then the constraint is *undefined* at t.

These previous concepts are illustrated in figure 2. It shows a pattern of the behaviour of a repeating task that misses some of its invocations (the 4th, 7th, 8th, and 18th) with a $\binom{3}{4}$ constraint and with a window size of 9 units. The figure displays the the state of the constraint for the lazy and eager evaluation modes.

The *lazy* bar shows the status of the constraint assuming that it is undefined unless there $\mu^w(t)$ has at least 4 invocations. At $t2$ four tasks are finished and it is known that the 4th invocation missed the deadline, however as the previous three were met, then the constraint is satisfied. This holds until time $t3$ when $\mu^w(t3)$ does not hold 4 instances of the task. at $t5$, the task missed the deadline and the weakly hard constraint is not satisfied as two deadlines were missed. this holds until $t6$ where the constraint is again satisfied as $\mu^w(t6) = 01111$.

The *eager* bar is able to decide sooner whether the constraint is either satisfied or missed. For instance at $t1$, as three deadlines were met, independently of the fourth invocation $\binom{3}{4}$ is satisfied. In the same way, at $t4$, as two deadlines were missed in the last three, the constraint is not satisfied anyway.

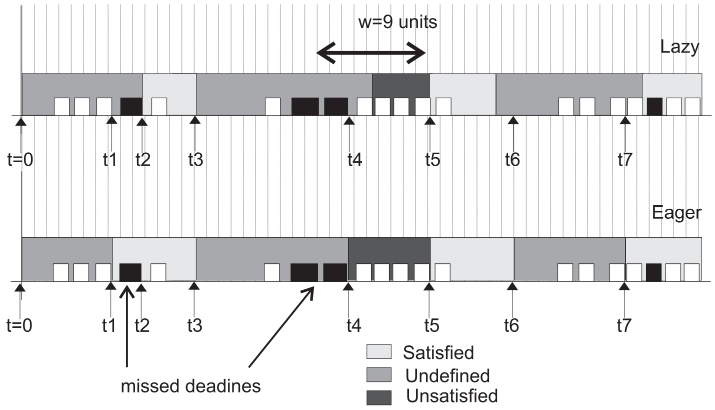

Figure 2: State of constraint $\binom{3}{4}$ for a repeating task under lazy and eager evaluation

However, updating $\mu^w(t)$ at finalisation and invocation times of the task introduces considerable overloads as it requires to time stamp each symbol of the μ-pattern and to determine which are the last symbols in w at t. As we seek for intuitive and simple mechanisms for checking weakly hard constraints we propose a relaxation on the satisfiability of the constraint. We assume that after a task is started, (*i.e.* after the first m invocations have finished), the last m symbols of the μ-pattern are the relevant history of the task. This assumption holds if the window size is larger than m times the average interarrival times of the task and the task does not have long periods of inactivity. Is so, then mechanisms for determining whether the constraint is undefined should be used.

3.5 An algebra of temporal constraints

A natural question arises as to whether different types of constraints are comparable. For this purpose we shall define the relation between sequences before. As our sequences are binary, we can naturally extend to them the logical operators (*i.e.* operations symbol to symbol). As a one in the μ-pattern represents a met deadline, we introduce the following definition to be able to compare μ-patterns.

Definition 8 *Given either two words u, v of the same length or two infinite sequences u, v (i.e. $u, v \in \Sigma^l$, $l > 0$, or $u, v \in \Sigma^\infty$) we say that v is easier than u (u is harder than v), if u has, at least, the same 1's and in the same position as v. Formally,*

$$u \preceq v \quad \Leftrightarrow \quad u \wedge v = u \tag{4}$$

It is easy to prove that the relation $\preceq$ is a partial order. Additionally, $(\Sigma^l, \preceq, \wedge, \vee)$ for all $l \geq 1$ is a bounded lattice. For any two words $u, v \in \Sigma^l$, their infimum is $u \wedge v$ and their supremum is $u \vee v$. The minimal element is the word 1^l and the maximal element is the word 0^l. The same applies to Σ^∞.

Given a constraint λ, it is also of interest to consider the set of sequences that satisfy the constraint:

Definition 9 *(Satisfaction set) The satisfaction set of a constraint λ, denoted by $\mathcal{S}(\lambda)$, is the set of all $\alpha \in \Sigma^\infty$ that satisfy λ.*

With this definition we can then introduce the comparison between constraints.

Definition 10 *Given two constraints, $\lambda, \lambda' \in \Gamma$, we say that λ' is easier that λ, denoted by $\lambda \preceq \lambda'$ if all sequences that satisfy λ also satisfy λ'. Formally,*

$$\lambda \preceq \lambda' \quad \Leftrightarrow \quad \mathcal{S}(\lambda) \subseteq \mathcal{S}(\lambda') \tag{5}$$

The relation $\preceq$ induces naturally the notion of constraint equivalence as

$$\lambda \equiv \lambda' \quad \Leftrightarrow \quad \lambda \preceq \lambda' \ \wedge \ \lambda' \preceq \lambda \tag{6}$$

It is easy to prove that $\equiv$ is an equivalence relationship. There are two classes of particular interest. Namely, $[\binom{1}{1}]$ and $[\binom{0}{1}]$. These two classes have the property that are the *hardest* and the *weakest* classes of constraints.

$[\binom{1}{1}]$ is the class of constrains that are only satisfied by the sequence 1^*, and the class $[\binom{0}{1}]$ is the class of constraints that are satisfied by any sequence.

Now we shall relate the relation between constraints and the relation between sequences. This is achieved by two monotonicity properties.

If a sequence α satisfies a constraint λ, any other sequence which is harder than α will also satisfy the constraint. Formally,

Theorem 1 *Given $\alpha \in \Sigma^\infty$ and $\lambda \in \Gamma$ such that $\alpha \vdash \lambda$, then $\forall \alpha' \in \Sigma^\infty$, $\alpha' \preceq \alpha \Rightarrow \alpha' \vdash \lambda$.*

Likewise, if a constraint λ is satisfied by a sequence α, any other constraint which is weaker than λ will be also satisfied by that sequence. Formally,

Theorem 2 *Given $\lambda \in \Gamma$ and $\alpha \in \Sigma^\infty$ such that $\alpha \vdash \lambda$, then $\forall \lambda' \in \Gamma;$, $\lambda \preceq \lambda' \Rightarrow \alpha \vdash \lambda'$.*

The proofs of these theorems are simple and they are left to the reader.

Another interesting property is the fragility. It measures how close is a particular sequence to not satisfying its weakly hard constraint.

Definition 11 *A sequence $\alpha \in \Sigma^\infty$ is p-fragile for a constraint λ, if $\alpha \vdash \lambda$ and $\exists \alpha'$ obtained from α by switching p 1's in α to 0, so that α' no longer satisfies λ.*

As a particular case, a 1-fragile sequence will be simply called a fragile sequence.

3.6 Tolerance functions

We can reformulate the fragility concept for one particular problem of interest. An on-line scheduling algorithm has to decide at run-time which task to schedule or to accept in the system. For weakly hard real-time systems, it is necessary a mechanism that allows reasoning on how close it is from missing its weakly hard constraint assuming some future behaviour and history of invocations of the task (possibly only the last $\lambda.m$ invocations).

Under this assumptions we introduce the concept of *tolerance* of a task to denote the number of missed deadlines it may tolerate next for it to keep satisfying its constraint. Several tolerance functions must be defined for each temporal constraint. We present the general definition of tolerance functions and some examples.

Definition 12 *(tolerance) Given a constraint λ with window size m and a sequence α of length $|\alpha| = m$. A tolerance function of λ is $f^\lambda : \Sigma^m \to \mathbb{Z}$ which has the following properties (assume u, v are m sequences):*

- $u \vdash \lambda \;\Leftrightarrow\; f^\lambda(u) \geq 0$
- $u \not\vdash \lambda \;\Leftrightarrow\; f^\lambda(u) < 0$
- *let* $v = u(2..m) \cdot 1$, *then* $f^\lambda(v) \geq f^\lambda(u)$
- *let* $v = u(2..m) \cdot 0$, *then* $f^\lambda(v) \leq f^\lambda(u)$

Tolerance is positive or zero if the constraint is satisfied, negative if it is not satisfied[4]. The last two properties capture the fact that when the next deadline is met, then tolerance is at least as high as before. On the contrary, if the deadline is missed, then tolerance either decreases or stays level.

If tolerance is 0 then we say that the task is *fragile* or *critical.* In this case next task invocations have to meet their deadlines to guarantee the satisfiability of the weakly hard constraint.

Tolerance functions can be: *three-valued* if $f^\lambda(w) \in \{-1, 0, 1\}$, or *multivalued* if $f^\lambda(w) \in [-M, +M]$.

3.6.1 Three-valued tolerance functions

The easiest tolerance functions that we can define are three-valued ones. The family of three-valued functions, denoted by B, are defined in the following way (let $m = \lambda.m$, length of $|w| = m$ and $w = \lambda$),

$$B^\lambda(w) = \begin{cases} +1 & \text{if } w(2..m) \cdot 0 \vdash \lambda \\ 0 & \text{if } w(2..m) \cdot 0 \not\vdash \lambda \text{ and } w(2..m) \cdot 1 \vdash \lambda \\ -1 & \text{if } w(2..m) \cdot 0 \not\vdash \lambda \text{ and } w(2..m) \cdot 1 \not\vdash \lambda \end{cases}$$

A Note that due to the monotonicity of the constraints if $w(2..m) \cdot 0 \vdash \lambda$ then also $w(2..m) \cdot 1 \vdash \lambda$.

If the constraint is fragile, then tolerance is 0; if it is not fragile, then tolerance is 1, finally if even though assuming that the next invocation is met the constraint is not satisfied, then tolerance is -1.

This simple functions, however do not capture the fact that some constraints allow more deadlines to be missed than others. To address this issue multivalued tolerance functions are introduced.

[4]This may happen with best-effort scheduling algorithm under heavy loads.

3.6.2 Multivalued tolerance functions

We need to introduce different tolerance functions for each of the four types of constraints.

For the any n m constraint, we can map the number of deadlines the task may miss in the future with the n and m parameters.

$$f_1^{(n\ m)}(w) = \begin{cases} p(w) - 1 & \text{if } \sum_{i=1}^{m} w_i \geq n \\ \sum_{i=1}^{m} w_i - n & \text{if } \sum_{i=1}^{m} w_i < n \end{cases}$$

where, $p(w)$ is the rightmost point in w such that $w(p..m)$ has n ones:

$$p(w) = \max \left\{ p \mid \sum_{i=p}^{m} w_i = n \right\}$$

Another possible definition of the tolerance of a $\binom{n}{m}$ constraint is actually the number of ones above the minimum number of deadlines that must be met:

$$f_2^{(n\ m)}(w) = \sum_{i=1}^{m} w_i - n$$

The graph in figure 3 shows the values of the $f_2^{(2\ 5)}$ function for the sequence $w = 1011111011$ 0110011111 1101000101 00. The first subsequence is $w(1..5) = 10111$ that has a tolerance of 2. The next sequence is $w(2..6) = 01111$ which has tolerance 3 and so forth.

Because of the way $f^{(n\ m)}$ is defined, a shift of the subsequence can only result in a change on tolerance of $1, 0$ or -1. If we consider $w = a\alpha$ and $w' = \alpha b$ where w' represents the next subsequence of w, then we can represent a transition from w to w' by a/b. Then an increment of $+1$ occurs under $0/1$. A decrement of -1 occurs under $1/0$ and when $0/0$ or $1/1$ the tolerance doesn not change.

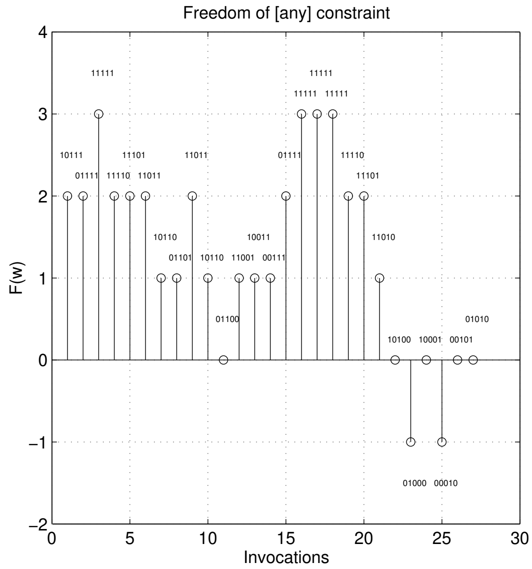

Figure 3: Tolerance graph of tolerance function $f^{(2\ 5)}$ for all subsequences $\alpha(i, 5)$ of sequence $\alpha = 10111\ 11011\ 01100\ 11111\ 11010\ 00101\ 00$.

Tolerance functions for *row* constraints are essentially different. A sequence has zero tolerance when the next symbol has to be 1 for guaranteeing the satisfiability of the constraint. For example, the sequence $w = 1111111000$, has tolerance 0 for a $\left\langle {4 \atop 10} \right\rangle$ constraint, because if the next four deadlines are not $1's$, then the constraint will not be satisfied. Assuming that the next three deadlines were met, then w becomes $w = 1111000111$, when the next deadline is met, the 4 1's in a row occur at the right end of the sequence. Thus,

$$f^{<n\ m>}(w) = \begin{cases} \max\{0, m-(2n-1)\} - e_n(w) & \text{if } 1^n \in w \\ n - g(w) & \text{otherwise} \end{cases}$$

where $e_n(w)$ is the right index of the rightmost sequence 1^n in α:

$$e_n(w) = \max\{e \mid w(e-n+1..e) = 1^n\}$$

and $g(w)$ is the length of the largest sequence of 1's in w:

$$g(w) = \max\{p \mid 1^p \in w\}$$

For the miss row constraint, a tolerance function is defined, simply as zero if the next deadline can not be a zero:

$$f^{<\bar{n}>}(w) = n - z(w)$$

where $z(w)$ is the number of consecutive zeros that there are at the right end of the sequence.

$$z(w) = \max\{p \mid w(m-p+1, m) = 0^p\}$$

This function decreases the tolerance as long as more deadlines are missed in a row. When one single deadline is met, the tolerance value reaches the maximum value n.

If multiple constraints are set on a task, then the tolerance of the task is the minimum of the tolerance associated to each constraint.

The issue of using tolerance functions at run-time is revised again in section 5.

3.7 Constraint relationship properties

After having introduced weakly hard constraints, and having analysed their properties, the next step is obtaining expressions that directly relate when two particular constraints are equivalent, harder or weaker. In this section we present some expressions that allow us to obtain whether the relationship between two constraints as a function of their n and m parameters. The proofs of these theorems can be found in [9].

We first start with some straightforward equivalences between constraints.

Theorem 3 $\binom{n}{m} \equiv \overline{\binom{m-n}{m}}$

Meeting n deadlines out of m, in any order is equivalent to specifying that no more than $m-n$ deadlines can be missed out of m.

Theorem 4 $\overline{\left\langle {n \atop m} \right\rangle} \equiv \overline{\left\langle {n \atop m'} \right\rangle} \quad \forall m, m' \geqslant n$

The theorem states that specifying that no more than n deadlines in a row can be missed is independent on the size of the window m.

We now move to relationships between constraints of the same type.

Theorem 5 $\binom{n}{m} \preceq \binom{p}{q} \Leftrightarrow$
$p \leqslant \max\left\{\left\lfloor \frac{q}{m} \right\rfloor n, q + \left\lceil \frac{q}{m} \right\rceil (n-m)\right\}$

The two following corollaries give us insight on constraint properties:

Corollary 1 $\binom{n}{m} \preceq \binom{p}{m} \Leftrightarrow p \leqslant n$

If two constraints have the same window size m, the higher the n, the harder the constraint.

Corollary 2 $\binom{n}{m} \preceq \binom{n}{q} \Leftrightarrow q \geqslant m$

If two constraints require the same number of deadlines met, the wider the window the weaker the constraint.

Theorem 6 $\left\langle {n \atop m} \right\rangle \in [\left\langle {1 \atop 1} \right\rangle] \Leftrightarrow 2n > m$

The previous theorem states that specifying that more than $\frac{m}{2}$ deadlines have to be met out of m is equivalent as specifying that *all* deadlines have to be met. This can be clearly seen with an example. For instance, specifying $\left\langle {3 \atop 5} \right\rangle$ requires that all deadlines must be met. Just consider the subsequence $\alpha = 11011$ of 1...11011...1 which has only one zero. α does not have three ones in a row and therefore $\left\langle {3 \atop 5} \right\rangle$ is not satisfied. The next theorems show the relationship between *row* constraints which are not strongly hard.

Theorem 7 *For* $\left\langle {n \atop m} \right\rangle, \left\langle {p \atop q} \right\rangle \notin [\left\langle {1 \atop 1} \right\rangle]$, $\left\langle {n \atop m} \right\rangle \preceq \left\langle {p \atop q} \right\rangle \Leftrightarrow$
$(q < m \;\wedge\; p \leqslant n - \lceil \frac{m-q}{2} \rceil) \;\vee\; (q \geqslant m \;\wedge\; p \leqslant n)$

In the same way as theorem 5, there are pairs of constraints which are not comparable. Also theorems 1 and 2 have their correspondents for the 'row' constraint:

Corollary 3 *For* $\left\langle {n \atop m} \right\rangle, \left\langle {p \atop m} \right\rangle \notin [\left\langle {1 \atop 1} \right\rangle]$
$\left\langle {n \atop m} \right\rangle \preceq \left\langle {p \atop m} \right\rangle \Leftrightarrow p \leqslant n$

Corollary 4 *For* $\left\langle {n \atop m} \right\rangle, \left\langle {n \atop q} \right\rangle \notin [\left\langle {1 \atop 1} \right\rangle]$
$\left\langle {n \atop m} \right\rangle \preceq \left\langle {n \atop q} \right\rangle \Leftrightarrow q \geqslant m$

If we know that a *row* constraint is harder than another one, with the same window size, then we can relate the parameters of the constraint as the following theorem shows:

Theorem 8 $\left\langle {n \atop m} \right\rangle \preceq \left({p \atop m} \right) \Rightarrow$
$p \leqslant 4n - m - 2,\ p \leqslant m,\ p \geqslant 0.$

Finally, a similar property as of theorem 1 can be obtained for the *miss row* constraint:

Theorem 9 $\overline{\langle n \rangle} \preceq \overline{\langle p \rangle} \Leftrightarrow n \leqslant p$

The proofs of these theorems can be found in [9].

3.8 Closed sequences

We have considered the satisfiability of weakly hard constraints in terms of infinite sequences of task invocations. However, for most systems the μ-pattern is made up of the cyclic repetition of a word. In such a case we say that the μ-pattern is closed. Formally,

Definition 13 *(closed sequence) Given a sequence $\alpha \in \Sigma^\infty$, we say that α is closed with cycle c if :*

$$\forall k, r \;\; k \geq 1, r > 0, \;\; \alpha(k) = \alpha(k + rc) \tag{7}$$

For closed sequences the check of whether $\alpha \vdash \lambda$ can be done by checking only the slices within a single cycle (plus the size of the window), as the following theorem shows.

Theorem 10 *If a sequence $\alpha \in \Sigma^\infty$ is closed then for all constraints with window size m,*

$$\alpha \vdash \lambda \;\;\Leftrightarrow\;\; \alpha(1..c + m - 1) \vdash \lambda$$

For closed sequences, the check of whether $\alpha \vdash \lambda$ can be computed by elementary algorithms whose cost is linear with respect to the length of the slice. Closed sequences are of special interest, as we shall see in the next sections. In most practical environments, periodic task μ-patterns are closed.

4 Fixed priority schedulability analysis

After having presented weakly hard constraints and studied their properties and relationships, we move to obtaining off-line schedulability checks for some process models under fixed priority basis. We leave for section 5 the discussion of scheduling approaches based on weakly hard constraints.

4.1 Introduction

A real-time system can be implemented in several ways, no matter its specification is strongly or weakly hard. However, the techniques for analysing whether a specification is met are different.

In this section, we will develop only the implementation consisting on mapping the specification jobs into execution threads. Moreover, we will restrict our analysis to fixed priority scheduling of such threads or tasks. The purpose of doing so is to provide insight about how weakly hard specifications can be usefully related to a particular implementation before extending the specification discussion.

4.2 Implementation issues

Of particular relevance is the assignment of priorities to the tasks. In many texts, the priority of the task is denoted by the task's subindex of the task, 1 being the highest priority. However, as we shall see later, this notation introduces some problems when the tasks may have various priorities or when these priorities may change. To avoid such notation problems we will use P_i as the priority of the task instead of the subindex.

We shall also have, for each task τ_i, the worst case computation time corresponding to a single invocation of the task, denoted by C_i. Moreover, the tasks may communicate data through common variables. If so, we must also take into account the blocking factors due to blocking operations upon these variables under the priority ceiling protocol [21, 22]. The maximum blocking time a task may suffer from lower priority tasks is denoted by B_i. With these definitions, each task is therefore characterised by:

$$\tau_i = (T_i, D_i, C_i, B_i, O_i, \lambda_i)$$

We shall also introduce the utilisation of the task set which is given by:

$$U = \sum_{\tau_i \in \Pi} \frac{C_i}{T_i}$$

we may also consider the contribution of each individual task to the utilisation, and we denote it by $U_i = C_i/T_i$.

4.3 Analysis

The analysis of a system made up of cyclic tasks scheduled with fixed priorities requires two steps. The first one is computing the upper bounds for the response times of each invocation of the tasks. The second one is checking the resulting such μ-pattern satisfies the constraint. As we will show that if the system is feasible the resulting μ-pattern is closed, this check can be performed easily.

4.3.1 Computing μ-patterns

In order to check whether the tasks of a system satisfy their weakly hard specifications we must start computing their μ-pattern. Fortunately, if the utilisation is not larger than 1 these μ-patterns are closed, so their analysis can be reduced to a finite sequence. However, we shall first prove this property.

Theorem 11 *Given a task set A, made up of independent periodic tasks scheduled under fixed priority, then all tasks are closed if and only if $U_A \leq 1$. Moreover the cycle of each task τ_i is given by $c_i = \frac{h_i}{T_i}$.*

If $U \leq 1$ then we can extend the system by an additional task τ_{N+1} at the lowest priority having $T_{N+1} = \mathbf{H}$ and $C_{N+1} = \mathbf{H}(1-U)$.

Such a task makes the new system have an utilization rate $U' = 1$. C_{N+1} will always be an integer because it is exactly the amount of time that the original system left unused. The response time of the only invocation of τ_{N+1} within $\mathbf{H}$ can be computed by the usual expression.

$$R_{N+1} = C_{N+1} + \sum_{\tau_j \in A} \left\lceil \frac{R_{N+1}}{T_j} \right\rceil C_j$$

$\mathbf{H}$ is a solution of the previous equation, because

$$C_{N+1} + \sum_{\tau_j \in A} \left\lceil \frac{\mathbf{H}}{T_j} \right\rceil C_j =$$

$$\frac{\mathbf{H}}{T_{N+1}} C_{N+1} + \sum_{\tau_j \in A} \frac{\mathbf{H}}{T_j} C_j = \mathbf{H}U' = \mathbf{H}$$

This means that the lowest priority task finishes its computation exactly by the hyperperiod, therefore the last time unit within the hyperperiod is used by τ_{N+1}. So any task from the original set A have completed all their invocations within the hyperperiod.

In that case the next hyperperiod starts with the same conditions as the first one. Therefore, the upper bounds on the response times of the tasks at any invocation have cyclic pattern of period $\mathbf{H}$.

The same argument also applies to the subset of tasks $\mathsf{hp}(\tau_i)$. In this case, the cycle of task τ_i is repeated every h_i time units, and therefore the task is closed with cycle $c_i = \frac{h_i}{T_i}$. □

Under the operation of a protocol for shared resources, the same result applies as, even though a task may have an additional interference from lower priority tasks, the total amount of computation time requested is the same as if no protocol was used.

The computation of the μ-pattern for task τ_i requires computing the worst case finalisation time of task i at invocation k, $F_i(k)$, within the first hyperperiod at level i. $F_i(k)$ is the sum of three factors. First, the time of computation of the first k invocations of the task. Second, the time that higher priority tasks have been computing before $F_i(k)$. Third, the time the processor has not been used by tasks with priority higher than or equal to τ_i. This is due to the fact that the invocations of these tasks can not start before their invocation times. So they can leave the processor idle. Therefore $F_i(k)$ is the minimum $t > 0$ that makes the following equation hold.

$$t = kC_i + \mathsf{Intf}_i(t) + \mathsf{Idle}_i(S_i(k)) \tag{8}$$

The idle time at level i, $\mathsf{Idle}_i(t)$, is the amount of time the processor can be used by tasks of lower priority than τ_i within period of time $[0, t)$.

The value $F_i(k)$ can be computed with a generalisation of the worst case response time formulation[9]. It is the minimum t that makes equation 8 hold; t is the time required to perform k computations of the task, plus the interference from higher priority tasks suffered during that period, plus the time the processor is used by lower priority tasks from $t = 0$ until the start time of the task. Formally,

$$t_{n+1} = kC_i + \mathsf{Intf}_i(t_n) + \mathsf{Idle}_i(S_i(k)) \tag{9}$$

This equation does not have a direct solution for t. But it can be computed with a recurrence formula. Starting with $t_0 = 0$, the recurrence finishes when $t_{n+1} = t_n$. The recurrence finishes if $U \leq 1$. The interference function $\mathsf{Intf}_i(t)$ is the amount of time higher priority tasks have requested before t and it can be computed by:

$$\mathsf{Intf}_i(t) = \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{t - O_j}{T_j} \right\rceil C_j \tag{10}$$

$\mathsf{Idle}_i(t)$ is the time the processor can be used by lower priority tasks between $[0, t)$. This can be computed by adding a virtual task $\bar{\tau} = \left(\bar{T} = t, \bar{D} = t, \bar{C}, \bar{O} = 0\right)$ of lower priority than τ_i and computing the maximum value $\bar{C}$, such that task $\bar{\tau}$ meets its deadline. Formally,

$$\mathsf{Idle}_i(t) = \max\left\{\bar{C} \mid \bar{\tau} \text{ is schedulable}\right\} \tag{11}$$

The finish time for task $\bar{\tau}$ is given by:

$$t = \bar{C} + \sum_{\tau_j \in \mathsf{hep}(\tau_i)} \left\lceil \frac{t - O_j}{T_j} \right\rceil C_j \tag{12}$$

and checking if $t < \bar{D}$. Equation 12 can be computed by a recurrence on the values of t. The range of values to check for $\bar{C}$ range from 0 to t and can be computed by a dichotomic search. See [9] for effective upper and lower bounds on the values to check.

Note that $\mathsf{Idle}_i(S_i(k))$ does not depend on t and therefore needs not to be recomputed at each iteration of the recurrence of equation 8.

4.3.2 Closed sequences

The fact that a system is closed is very important as it allows to study the the behaviour of the weakly hard constraints by analysing the invocations of the tasks within one single hyperperiod of each task. As we have shown in theorem 11, if $U \leq 1$ and all task have no offsets ($O_i = 0$) then the task set is closed and therefore the μ-pattern is repeated cyclically at every hyperperiod.

However, when some of the tasks may have offsets different from zero the system is also closed if $U \leq 1$ but in this case: (a) the worst case response time does not happen at $t = 0$, and (b) the finish time of the tasks may not be closed within the hyperperiod at level i (some tasks can overrun over the hyperperiod at level i).

Audsley [23] showed that, for hard tasks, it is only necessary to compute $R_i(k)$ for values of k such that $S_i(k) \leq h_i + \max\{O_j\}$. For tasks that can miss some deadlines, this check is not sufficient. If the task response times are bounded, the worst case response time occurs between the first two hyperperiods at level i. It can be obtained by computing $R_i(k)$ for all k in the first hyperperiod at level i, plus some invocations in the second hyperperiod. We compute $R_i(k)$ for $a_i + 1 \leq k \leq a_i + p$, where $p \geq 0$ is the first value such that $R_i(p) \geq R_i(a_i + p)$. Formally,

$$\hat{R}_i(k) = \begin{cases} R_i(a_i + k) & 1 \leq k \leq p \\ R_i(k) & p + 1 \leq k \leq a_i \end{cases} \tag{13}$$

Therefore, to compute $\hat{R}_i(k)$ we need to compute $a_i + p$ task invocation finish times. In most of the cases, $p = 1$ should suffice. Only in highly overloaded task sets with large offsets it will be necessary to compute $R_i(k)$ for all invocations in the second hyperperiod. We compute the worst case μ-pattern in the same way. For $1 \leq k \leq a_i$.

$$\hat{M}_i(k) = \begin{cases} 1 & \text{if } \hat{R}_i(k) \leq D_i \\ 0 & \text{Otherwise} \end{cases} \tag{14}$$

Note that, $R_i(k)$, can also be computed with an event driven simulation but, as it is shown in [9], the formulation presented here is much more efficient. However, there are two major problems associated to building the schedule.

- The schedule can only be build for perfectly synchronous systems. *i.e.* without blocking factors and without sporadic tasks which limits the applicability of the analysis to a wide range of task models.

- The schedule has to be build for all the tasks independently of whether they meet all of their deadlines.

The formulation presented here actually computes the worst case response time of each invocation assuming the maximum blocking it may suffer and the maximum interference from sporadic tasks. Each task is analysed independently and only within its hyperperiod at level i. Therefore only tasks which may miss the deadline at the critical instance may require to compute the invocations in one hyperperiod.

One common argument against analysing the system up to the hyperperiod is that, in general, the hyperperiod can be extremely large. In theory this is the case but we note that in real engineering scenarios this pessimism can be overridden by noting that the periods of the tasks can be slightly modified to remove common prime factors. Also note that the overhead of the computation is small in comparison to the cost of, for instance, compiling the whole system. The check may take, in a current desktop computer, in the order of seconds to compute. Although, it is not applicable as an on-line check.

The previous analysis can be easily extended to include the maximum blocking time B_i due to the use of semaphores provided that the protocol for locking the semaphores is like the immediate ceiling protocol [24]. Non-periodic tasks can also be incorporated in the analysis, although in general it is not possible to guarantee off-line whether their weakly hard constraints will be satisfied.

Note that most of the tasks will meet all of their deadlines and therefore weakly hard checks would have to be performed on a small subset of the tasks only.

5 Scheduling weakly hard real-time systems

In the previous section we have provided a technique to determine whether a task set is weakly hard schedulable under fixed priority scheduling. If so, then the system can be scheduled under a traditional fixed priority scheduler and weakly hard constraints are always satisfied. We also note that similar analysis can be developed for the traditional dynamic priority schedulers like EDF.

In general, finding optimal schedules for tasks that allow skips is a hard problem. Several results are known. For instance Leung And Wong in [25] show that a very similar problem, minimising the number of late tasks with the constraint that the total error is no more than a given threshold is NP-Hard for a single processor, even if all tasks have identical deadlines and release times. Also Shih *et al.* [26] showed that the problem of minimising the total error for imprecise computation tasks with optional components is NP-Hard. Similar results can be found in [27]. The conclusion that can be deduced from this results is that trying to obtain optimal algorithms is unlikely to succeed and therefore research should be directed to obtain good heuristics with low overheads.

An alternative approach of using fixed priority scheduling or EDF is to introduce new scheduling algorithms that specifically address the issue of dealing with weakly hard constraints. The algorithms can be classified according to the following categories:

- Guaranteed: The algorithm guarantees that weakly hard constraints are always satisfied, *i.e.* off-line schedulability checks are required.

- Best-effort: No guarantee can be obtained, but the objective is to minimise the number of tasks that miss their weakly hard constraints. When all constraints can be met, any secondary objective can be specified, like minimising the number of missed deadlines, fair distribution of missed deadlines among competing tasks, etc...

- Acceptance tests: The scheduler is based on rejecting some task instances, mainly for guaranteeing that the set of ready tasks is not overloaded. The acceptance tests are based on weakly hard constraints. The acceptance test itself can be either guaranteed or best-effort.

Please note that the notion of how weak a task is (actually the tolerance) can be mapped into the value of the function and therefore any value based scheduler can then be used.

Algorithms that deal with overloaded conditions are usually structured in two component. One component that is used in normal conditions (not overload) and one component (may be different than the previous one) to be used under overloaded conditions. We mainly present scheduling checks that deal with the overloaded conditions.

The algorithms should also provide a level of fairness. While it is important to guarantee that all tasks will meet their weakly hard constraints, a secondary objective is to distribute among the tasks the number of deadlines missed (or maximum error, maximum tardiness, etc) so that not always the same tasks are the ones that miss the deadlines.

The main idea behind the schedulers is to use the tolerance functions as the basic mechanism for taking the scheduling decisions. If three-valued tolerance functions are only ones used, then the tasks are classified in two classes, strongly hard (need to meet the next deadline) or weakly hard (may miss the next deadline). If multi-valued tolerance functions are used, more sophisticated decisions can be taken.

5.1 Scheduling algorithms

In this section, we present some weakly hard scheduling algorithms. The first group has already been published in the literature although their results and analysis have been mainly used only for a very particular domain and were not applicable to the general case. The important point to note is that all the algorithms presented here are particular cases of the definition of weakly hard constraints and tolerance functions.

- *RTO*: The simplest algorithm (under fixed priority and dynamic priority) is called the *Red Task Only (RTO)* for dynamic priorities and *Rate Monotonic RTO (RM-RTO)*. Introduced by Koren *et al.* in [10]. The algorithm was only applied to use constraints of the $\binom{m-1}{m}$ type. The algorithm skips all tasks invocations which are not strongly hard. This naive approach has no benefits as tasks invocations are discarded event though there may be available computation resources to finish on time.

- *BWP*: The extended version of this algorithm is called *Blue When Possible (BWP)*. In which non critical tasks can be scheduled if there are no critical tasks ready. Also the approach is too simple. Anyway, it may be useful for simple constraints.

- *FP*: Traditional fixed priority scheduling with weakly hard constraints is another algorithm. All tasks are scheduled even though it is known that some of them will miss their deadlinesm although, it is guaranteed off-line that weakly hard constraints are satisfied.

- *EDF*: Earliest deadline first can also be used with weakly hard constraints. If $D \leq T$ and $U \leq 1$, then system is bounded and tasks can be weakly hard. Off-line checks can be derived to determine whether weakly hard constraints will be always satisfied.

- *(m,k)-firm deadlines*: Is a scheduling algorithm introduced by Hamdaoui and Ramanathan in [11]. In this approach, they introduce the (m,k)-constraint which is equivalent to the $\binom{m}{k}$ constraint. They present their scheduling algorithm for scheduling messages in a communication system. They assume that all tasks have the same m and k parameters and based on this, they define the priority of the task as, in fact, the tolerance of the task. Therefore task which are closer to miss their (m,k)-firm constraint have higher priority. However, they do not address scheduling issues of tasks with different parameters or other constraints.

- *EDPS*: The enhanced dual-priority scheduling algorithm introduced by Bernat *et al.* in [8, 9] is based on dual priority mechanism, first introduced by Davies *et al.* in [28]. Priorities are divided in two bands, higher and lower. Each task has assigned two priorities, one in the upper band and one in the lower. Soft task may only have priorities assigned in the lower band. Hard tasks start running in the lower band and if after a period of time. Each task has a priority promotion point which is the latest time by which if the task is promoted to the upper band it is guaranteed that it will meet the deadline. As the task in the upper band has a bounded interference, the priority promotion time can be easily computed. The main idea of the algorithm is to promote a task only when the task is strongly hard. Therefore competing for CPU with the rest of the tasks. Schedulability checks exists that allow to guarantee that even in the worst case hard tasks will always meet their weakly hard constraints.

- The *Melody* approach presented by Wedde and Lind in [17] introduces two concepts, the criticality and sensitivity which at run time are used to define the relative criticality and relative sensitivity at any invocation k. These relative criticality (and sensitivity) depend on pairs of boundaries that define the allowed criticality (sensitivity) of the task and the number of consecutive deadlines the task has missed. This is very similar to a specification of a $\overline{\langle n \rangle}$ constraint. However, the notions of criticality and sensitivity are far from intuitive neither the formulas they use to compute the relative values. Their work is supported with simulation studies but no analysis is provided in order to guarantee some off-line properties of the system, therefore it is only a best-effort technique.

These algorithms have been developed with different objectives and requirements in mind, it is clear that all of them can are particular instances of how to use weakly hard constraints and tolerance functions. We can now propose new scheduling algorithms ones also based on the tolerance concepts. A detailed study of their properties is required but it is out of the scope of this paper. We present their informal definitions only.

- *Least Tolerance First (LTF).* Based on the tolerance function, schedule the tasks with the smallest tolerance. Under the likely case of multiple tasks with the same tolerance value, use any known scheduling policy (fixed priority or EDF, value density, etc).

- *Weighted Deadline-Tolerance (WDT).* Schedule the task with least weighted tolerance-deadline ($(\mathsf{tolerance}+1) \times D_i$). This compensates for tasks with short deadlines competing with tasks with long deadlines.

- *Tolerance Acceptance (TA).* Use tolerance as a criteria for acceptance. When a task is released (or gets its first tick) the task is rejected if the system is overloaded and tolerance is greater than zero.

- *Predictable Degradation Mode (PDM).* Different weakly hard constraints can be associated to the same task under different modes of operation. For instance, under normal operation even non-critical tasks have $\binom{1}{1}$ constraints assigned. However, when a fault is detected, non-critical tasks have their performance degraded to $\binom{n}{m}$ so that there is plenty more time for the recovery mechanisms to execute. When the system arrives to a stable state again, degraded tasks run with $\binom{1}{1}$ constraints again. This would require detailed analysis of mode changes for weakly hard tasks which is actually being performed.

- *Dynamic guaranteed QoS (DGQ).* QoS requirements can be mapped onto weakly hard constraints. For instance the quality of a video player can be mapped to the number of deadlines met in a window of time. When multiple tasks with QoS requirements compete for available resources, then an adaptation mechanism is required. Whenever a new task is started, the scheduler recomputes the n in m parameters of all the tasks in the system so that the available processor can be shared between the tasks. The range of values that can be changed for each task depends on the requested minimum level of service negotiated for each task.

- *Pessimism Reduction (PR).* Schedulability analysis is performed to determine whether *in the worst case* all tasks meet their deadlines. The pessimism involved gets worse with powerful processors that use advanced features like caches and pipelines because they are very difficult to model for the worst case. As a result the worst case analysis values are much greater than the real execution times. Either the system is underloaded in order to guarantee that all deadlines are met, or deadlines may be missed. Tasks should meet all their deadlines assuming its average response time [5] and weakly hard schedulable using worst case execution time. During normal execution of the system it is likely that a deadline is never missed because of the pessimism introduced in the analysis. However, worst case weakly hard behaviour is guaranteed. In this case, A good and simple scheduling strategy could be to schedule the tasks at their priority level.

- *Minimum Transaction Tolerance (MTT).* In transaction based systems, a given operation requires a group of tasks. This tasks are called a transaction. It is desirable to set weakly hard constraints to the transaction. This can be done by extending any previous scheduling model, and to consider a transaction tolerance function to be the minimum of the tolerance of any member of the transaction. If R is the set of tasks of a transaction then, the actual tolerance Tol_R of the transaction is given by:

$$\mathsf{Tol}_R = \min\{f^{\lambda}(\mu_i)\}_{\forall \tau_i \in R} \qquad (15)$$

  This guarantees that the behaviour of the whole transaction is of the task with less tolerance. Also ensures that that the performance of the whole transaction is guaranteed over time. If one of the first tasks of the transaction misses its deadline, then it reduces the number of deadlines task in the rest of the transaction.

- *Synchronised Tolerance (ST).* In a similar way as the transactions. Groups of tasks may require to perform synchronously. For example, for data fusion from multiple sources. To ensure that all tasks which should work synchronously behave in that way, the actual tolerance of the task should be kept constant for all the tasks in the synchronised set. If S is the set of tasks of the synchronised set, then the actual tolerance of each task in the set is equal to the tolerance when the first task in the set started. When the last task in the synchronised set finishes, the new tolerance for the set is computed from the tolerance of each individual task, for example as the minimum of the tolerance of the task in the set.

Table 3 shows a summary of some of the properties of the algorithms presented here. The legends are as follows: *Fix* means that the algorithm is based on fixed priorities, *Dyn* means that it is essentially used with dynamic priorities. *Dyn/Fix* means that the idea can be used with both fixed priority and dynamic priority schedulers. *BE* stands for best effort. The algorithm is an heuristic without schedulability checks that will guarantee that all weakly hard constraints will be met. *G* means guaranteed. Off-line schedulability checks can be used to guarantee that all constraints will be satisfied. Finally, *Acc* means that it can be used as an acceptance test.

The previous algorithms are just an indication of possible scheduling algorithms that could be derived from exploiting weakly hard constraints of the tasks. Further work is required to determine schedulability checks for them, comparison of their fairness, behaviour of maximum lateness, etc.

Also, it is important to note that implementation details are of major importance. A scheduling algorithm which is hard to implement, uses high amounts of memory and/or consumes time, and most important, it is not intuitive to see how it works, is unlikely to succeed.

All these algorithms presented here introduce very low overheads in computation time and memory requirements [6] and they are easy to implement.

6 Conclusion

New generation of real-time systems require mechanisms to specify in a clear, predictable and bounded way that some deadlines can be missed. This is mainly for:

[5] More appropriate is to use a value in between the average case and worst case. i.e. $a \in [0,1]$ and $C = \text{average} + a(\text{worst} - \text{average})$

[6] Most of the operations on the history of the task can be done with bit operations with registers.

Table 3: Summary of properties of on-line weakly hard scheduling algorithms

Algorithm	Fix/Dyn	Guar. / B-E	Acceptance
RTO	Dyn	G	Yes
RM-RTO	Fix	G	Yes
BWP	Dyn	G	Yes
FP	Fix	G	No
EDF	Dyn	G	No
(m,k)-firm	Dyn	B	No
EDPS	FP	G	No
Melody	?	B	Yes
LTF	Fix/Dyn	G	Yes
WDT	Fix/Dyn	G	Yes
TA	Fix/Dyn	G/B	Yes
PDM	Fix/Dyn	G/B	Yes
DGQ	Fix/Dyn	G/B	Yes
PR	Fix/Dyn	G	Yes
MTT	Fix/Dyn	G/B	Yes
ST	Fix/Dyn	G/B	Yes

- alleviating the pessimism in the parameters of the system (mainly C_i) and the worst case scenarios (assuming all sporadics arriving simultaneously),
- providing a mechanism for fair degradation of the quality of the service tasks provide, and
- obtaining fair mechanism for deciding which tasks need to be skipped during transient overloads.

This allows building systems with worst case utilisation far above 100% , but with average case utilization well below 100% . This means that, generally, all tasks meet their deadlines, but even in the worst case a minimum level of service is given to the tasks.

The other main property of such systems is that allows to predict what the behaviour of a task will be in case deadlines are missed. Current scheduling algorithms do not remember which tasks missed the deadline and how many times. This leads to unfairness between the tasks and to non-guaranteed behaviour.

Weakly hard real-time systems are based on

- clear and intuitive constraints for specifying the number of deadlines missed and met over a period of time,
- Study of the properties of the tasks specified with weakly hard constraints,
- Off-line schedulability checks for determining whether those constraints will be satisfied under traditional scheduling algorithms, and
- New scheduling algorithms specifically designed to deal with weakly hard tasks based on the concept of *tolerance*.

Acknowledgments

This research has been partially funded by a grant from "Caixa de Balears. Sa Nostra" and by the Universitat de les Illes Balears.

References

[1] Stuart Bennett, *Real-time computer control: an introduction*, Prentice Hall Intl., 1994.

[2] Constantine Houpis, *Digital control systems. Theory, Hardware and Software*, McGraw-Hill, 1985.

[3] K. Åström and B. Wittenmark, *Computer Controlled Systems. Theory and Design*, Prentice Hall, 2nd edition, 1990.

[4] C.C. Bissell, *Control Engineering*, Chapman & Hall, 2nd edition, 1998.

[5] Martin Törngren, "Fundamentals of implementing real-time control applications in distributed computer systems," *Real-Time Systems*, vol. 14, pp. 219–250, 1998.

[6] C.L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[7] Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar, "Rate-monotonic analysis for real-time industrial computing," *Computer*, pp. 25–32, 1994.

[8] Guillem Bernat and Alan Burns, "Combining (n m)-hard deadlines with dual priority scheduling," in *18th IEEE Real-Time systems symposium. San Francisco, CA*, December 1997.

[9] Guillem Bernat, *Specification and Analysis of Weakly Hard Real-Time Systems*, Ph.D. thesis, Departament de Ciències Matemàtiques i Informàtica. Universitat de les Illes Balears. Spain, 1998.

[10] G. Koren and D. Shasha, "Skip-over: Algorithms and complexity for overloaded systems that allow skips," in *proc. 16th IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995, pp. 110–117.

[11] Moncef Hamdaoui and Parameswaran Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1443–1451, December 1995.

[12] G. Koren and D. Shasha, "D-over: An optimal on-line scheduling algorithm for overloaded real-time systems," in *proc. 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, USA, December 1992, pp. 290–300.

[13] Marco Caccamo and Giorgio Buttazzo, "Exploiting skips in periodic tasks for enhancing aperiodic responsiveness," in *17th IEEE Real-Time Systems Symposium. San Francisco, CA.*, December 1997, pp. 330–339.

[14] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio Buttazzo, *Deadline Scheduling for real-time systems: EDF and Related Algorithms*, Kluwer academic publishers, 1998.

[15] Giorgio Buttazzo, Marco Spuri, and Fabrizio Sensini, "Value vs. deadline scheduling in overload conditions," *proc. 16th IEEE Real-Time Systems Symposium*, pp. 90–99, December 1995.

[16] Marco Spuri, Giorgio Buttazzo, and Fabrizio Sensini, "Robust aperiodic scheduling under dynamic priority systems," in *16th IEEE Real-Time Systems Symposium. Pisa, Italy*, December 1995, pp. 210–219.

[17] Horst F. Wedde and Jon A. Lind, "Integration of task scheduling and file services in the safety-critical system MELODY," in *10th IEEE Euromicro Workshop on Real-Time Systems*, Berlin. Germany, June 1998, pp. 18–25.

[18] Scott Brandt and Gary Nutt, "A dynamic quality of service middleware agent for mediating application resource usage," in *19th IEEE Real-time systems symposium. Madrid*, December 1998, pp. 307–317.

[19] Luca Abeni and Giorgio Buttazzo, "Integrating multimedia applications in hardware real-time systems," in *19th IEEE Real-Time Systems Symposium. Madrid*, December 1998, pp. 4–14.

[20] D. Locke, D. Vogel, and T. Mesler, "Building a predictable avionics platform in Ada: A case study," in *proc. 12th Real-Time Systems Symposium*, San Antonio, Texas, December 1991, pp. 181–189.

[21] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[22] Neil Audsley, Alan Burns, M. Richardson, and Andy J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[23] N. C. Audsley, *Flexible Scheduling of Hard Real-Time Systems*, Ph.D. thesis, Department of Computer Science. University of York, 1993.

[24] Lui Sha, Ragunathan Rajkumar, Sang Hyuk Son, and Chun-Hyon Chang, "A real-time locking protocol," *IEEE Transactions on Computers*, vol. 40, no. 7, pp. 793–800, 1991.

[25] J. Leung and C.S. Wong, "Minimizing the number of late tasks with error constraint," *Information and Computation*, vol. 106, pp. 83–108, 1993.

[26] W.K. Shih, J.W.S. Liu, and J.Y. Chung, "Algorithms for scheduling imprecise computations with timing constraints," in *proc. 10th IEEE Real-Time Systems Symposium*, 1989.

[27] Swaminathan Natarajan, *Imprecise and Approximate Computation*, Kluwer Academic Press, 1995.

[28] Robert Davis and Andy Wellings, "Dual priority scheduling," in *proc. 16th IEEE Real-Time Systems Symposium*, December 1995, pp. 100–109.

Contents

1 Introduction **1**
- 1.1 Weakly-hard real-time systems . . . 2
- 1.2 Motivation examples . . . 3
  - 1.2.1 Computer driven automatic control systems . . . 3
  - 1.2.2 Monitoring system . . . 5
  - 1.2.3 Multimedia systems . . . 5
- 1.3 Scope of the research . . . 6
- 1.4 Previous Work . . . 6
- 1.5 Structure of the article . . . 7

2 Process Model **7**
- 2.1 Periodic tasks . . . 8
- 2.2 Non-periodic tasks: repeating and isolated tasks . . . 9

3 Specification of weakly hard real-time systems **9**
- 3.1 Example . . . 9
- 3.2 μ-patterns . . . 11
- 3.3 Weakly hard temporal constraints . . . 11
- 3.4 Weakly hard constraints for repeating tasks . . . 13
- 3.5 An algebra of temporal constraints . . . 15
- 3.6 Tolerance functions . . . 16
  - 3.6.1 Three-valued tolerance functions . . . 16
  - 3.6.2 Multivalued tolerance functions . . . 17
- 3.7 Constraint relationship properties . . . 18
- 3.8 Closed sequences . . . 19

4 Fixed priority schedulability analysis **20**
- 4.1 Introduction . . . 20
- 4.2 Implementation issues . . . 20
- 4.3 Analysis . . . 20
  - 4.3.1 Computing μ-patterns . . . 20
  - 4.3.2 Closed sequences . . . 22

5 Scheduling weakly hard real-time systems **23**
- 5.1 Scheduling algorithms . . . 24

6 Conclusion **26**