Guaranteed On-Line Weakly-Hard Real-Time Systems

Guillem Bernat
University of York
Real-Time System Research Group
York - England
bernat@cs.york.ac.uk

Ricardo Cayssials
Universidad Nacional del Sur
Department of Electrical Engineering
Bahía Blanca - Argentina
iecayss@criba.edu.ar

Abstract

A weakly hard real-time system is a system that can tolerate some degree of missed deadlines provided that this number is bounded and guaranteed off-line. In this paper we present an on-line scheduling framework called Bi-Modal Scheduler (BMS) for weakly-hard real-time systems. It is characterised by two modes of operation. In normal mode *tasks can be scheduled with a generic scheduler (possibly best-effort). Weakly hard constraints are guaranteed to be satisfied by switching, whenever necessary, to a* panic mode *for which schedulability tests exist that guarantee that deadlines are met. Due to the sources of pessimism in the analysis (mainly WCET and critical instant assumptions) the worst case situations may never arise, thus almost all the time all deadlines are met, only at peak loads some deadlines may be missed, however the behaviour of the system is predicable and bounded. This allows building systems which maximise resource usage during normal operation and that resort to a guaranteed and predictable performance degradation specified by the weakly hard constraints should a transient overload arise.*

1. Introduction

Traditionally, real-time systems (or tasks) are classified as being *hard*, *soft* or *firm*. For hard real-time tasks, it is imperative that no deadline is missed, while for soft tasks, it is acceptable to miss some of the deadlines occasionally. It is still valuable for the system to finish the task, even if it is late, or just not finish that particular invocation. Firm tasks are also allowed to miss some of their deadlines, but as opposed to soft tasks there is no associated value if they finish after the deadline.

In practical engineering contexts, the occasional loss of some deadline can be usually tolerated (even in most hard real-time systems). This is either because the consequences of the loss can be negligible (due to the fact that the robustness of the involved algorithms imply the ability to react properly at the next invocation step without serious consequences), or because the effect of a single missed deadline is not noticeable by a user (as it would be the case of a missed audio packet 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. In fact most real-time systems that are classified as hard are not that hard. Some level of missed deadlines could be tolerated but the ambiguity of the definition of soft and firm is so inadequate that the system is considered to have to meet all deadlines when it really does not require so.

Assuming that some deadlines can be missed, 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 in a computer controlled system 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. To address the issues of real-time systems that can miss deadlines we introduce the concept of weakly-hard real-time system [3]:

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.*

We use the notion of weakly-hard real-time system to represent an appropriate conceptual framework for specifying real-time systems that can tolerate occasional losses of deadlines. Systems (tasks) 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(tasks)* whenever the distinction is relevant.

One feature that characterises real-time systems is that they are dimensioned for the worst case. This worst case scenario is very pessimistic and the load during this instant is generally much higher than the average load during the normal operation of the system. By specifying weakly hard constraints we can effectively lower down the resource demand during peak loads, therefore being able to schedule a system that otherwise would be deemed unschedulable.

There are three sources of pessimism in the analysis: (1) tasks do not always run for the worst case due to not following the worst path and due to the pessimism of the WCET analysis itself, (2) tasks do not rearrive always with their minimum interarrival time and (3) tasks do not suffer every time the worst case blocking factor. As a result, although the analysis may find out that some deadlines may be missed, at run-time almost all of them are actually met. In fact, during the normal operation of the system, the current load is much lower than the worst case scenario.

There are two research problems of interest related to real-time systems that can miss some of its deadlines:

- How to specify temporal constraints for weakly hard real-time systems
- How to schedule systems with weakly-hard constraints.

The specification of temporal constraints is generally done, for example, with a constraint $\binom{n}{m}$ meaning that a task has to meet at least n deadlines in any m consecutive invocations (other types of constraints are defined in section 2). If this constraint is not satisfied (e.g. more than n invocations miss its deadline in a window of m invocations), it is said that a *dynamic failure* occurs.

The scheduling approaches are concerned on how to use the weakly hard constraints to better schedule the system. The classification of current scheduling approaches can be done according to:

- Guaranteed/Best-effort: In a guaranteed framework, schedulability tests exist for guaranteeing that no dynamic failure can occur. Non-guaranteed schedulers use best-effort techniques to minimise the probability of a dynamic failure.
- Off-line/On-line schedules: An off-line approach follows a pre-built sequence of which tasks to skip. On the other hand, in on-line approaches the decision on what invocation is chosen to be executed is made at runtime, usually based on the recent history of the system.
- Complexity: In evaluating the suitability of a particular approach, the complexity of determining such approach has to be also considered. On on-line approaches, computationally expensive acceptance tests or priority calculations are clearly not adequate. On the contrary, off-line approaches may have a very low complexity, but they are not suitable when scheduling decision have to be taken at runtime.
- Process models: Weakly hard constraints apply to periodic (or pseudo-periodic) tasks only. However, real-time systems may be made up of other types of tasks. Some approaches are restrictive as they may only apply to task sets with restricted or uniform process models (all tasks are similar, same window size, $D_i = T_i$, only periodic tasks, etc). The general applicability of scheduling approach is its integration and coexistence with other scheduling approaches and process models.

The original contribution of this paper is a framework for scheduling real-time systems that has the following properties: Guaranteed: there exists off-line worst-case schedulability tests that guarantee that weakly-hard constraints are always satisfied (no dynamic failure can occur); On-line: scheduling decisions are taken at runtime, when the task is released and only once per task invocation; Low complexity: implementation of decision tests can be done with very few machine instructions; Heterogeneous process model: weakly hard tasks can coexist with other types of tasks (hard, sporadic, aperiodic servers, etc.) as schedulability tests exist that allow to model the worst case interference that they produce on each other. The approach presented here also allows any type of weakly hard constraint with arbitrary parameters. No restriction on tasks having the same size, or same weakly hard constraints is imposed.

The scheduling framework is based on a two mode scheduler, named *Bi-Modal* scheduler (BMS). In *normal mode* tasks are scheduled according to a flexible (generic) scheduling policy that may take into account past history of missed and met deadlines, computation times, value and deadlines. However, in order to prevent a dynamic failure the scheduler can switch to *panic mode* for which schedulability tests exist for guaranteeing that a task scheduled in this mode will meet its deadline. By changing task instances to panic mode adequately, dynamic failures are prevented. When there is no danger of a dynamic failure the scheduler switches back to normal mode. This change of operation can be done very efficiently and with very low overhead thus effectively providing a guaranteed minimum performance.

In the rest of this section we review the related work and in section 2 we formally introduce the concept of weakly-hard real-time system. After that, in section 3 we present our process model. We later introduce the concept of μ-pattern in section 4 to describe the history of missed and met deadlines. In section 5 we present the scheduling framework and in section 6 the schedulability analysis in panic mode. Finally in section 7 we present the evaluation of the approach through simulation. We present our main conclusions in section 8.

1.1. Related work

In [7] the concept of (m,k)-firm deadlines was introduced to model tasks that have to meet m deadlines every k consecutive invocations in the context of a stream of packets of communication networks. Later the same authors developed an analytic model for evaluating the expected probability of dynamic failure for an incoming stream given the other streams present in the system [8].

In [7], [8], [13] and [15], best effort on-line scheduling algorithms were proposed to schedule (m,k)-firm deadlines. In these approaches, the task's priority is raised if it is close to not meeting m deadlines on the last k invocations. As they are just best-effort approaches, it is possible that dynamic failures occur. Guaranteed approaches of (m,k)-firm systems were considered in [11] and [14]. In [11] an off-line scheduler is proposed and it is limited to be applicable to purely periodic tasks. Moreover, it does not exploit the fact that at run-time some invocations that were deemed to miss its deadline could have, indeed, finished on time. In [14], (m,k)-firm deadlines constraints are guaranteed over fixed windows when a *Dynamic Windows-Constrained Scheduling* (DWCS) is used. The process model is very restrictive as it requires all tasks to have the same window m and $D_i = T_i$. Dynamic failures may occur if consecutive (sliding) window are considered.

The *Skip-Over* model [9] is a particular case of (m,k)-firm model in which some invocations re skipped according to a *skip factor* s. If a task has a skip factor of s, it will have one invocation skipped out of s (it can be specified as a $(s-1, s)$-firm task). Skip-Over scheduling algorithms are on-line, guarantee and low complexity ones, but they cannot be extended to systems with arbitrary weakly hard constraints. In [4] and [5] skips are exploited to minimise the response time of aperiodic requests in an EDF scheduling context.

Bernat et al. [3] introduce the notion of weakly-hard real time constraints as a generalisation of the concept of (m,k)-firm to cover other types of missed invocation patterns (see section 2). It is a formally defined and general conceptual framework for specifying real time systems that can tolerate missed deadlines. This weakly-hard real time framework allows much richer types of temporal restrictions to be defined. The initial work provided off-line schedulability tests for fixed priority scheduled systems. An initial approach for the joint scheduling of aperiodic work with weakly hard schedulers, called enhanced dual priority scheduling, was presented in [2]. It is based on the dual priority mechanism [6], it only promoted task instances following a predefined pattern thus making more slack for aperiodic tasks. However, the approach was very rigid as the pattern was fixed beforehand and could not be altered if the tasks did not run for their worst case.

Previous approaches may be classified as: 1) based on best-effort algorithms and consequently no guarantees of avoiding dynamic failures are given or, 2) process model are very restrictive requiring either $D_i = T_i$ or same m for all tasks. Also, they are rigid and do not usually address the fact that worst case estimates are very pessimistic and that it is likely that during the normal operation of the system the behaviour would be much better.

Our approach addresses specifically these issues. It enables generic process models allowing, for instance, $D_i \leq T_i$, arbitrary weakly hard constraints for each task, and coexistence of different types of tasks including periodic and non-periodic tasks, aperiodic servers, blocking factors etc. It also exploits the fact that tasks do not always run for their worst case. The approach presented here encompass all previous approaches and allows to model them as particular cases. But most importantly, is that it provide schedulability tests that allow to obtain absolute guarantees that dynamic failures do never occur.

2. Weakly-hard real-time systems

The tolerance to missed deadlines cannot be adequately specified by a single parameter, for example with the percentage of deadlines to be met or missed (although it is common practice to do so). 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 900 deadlines met, which is clearly not the same. To capture this situation another parameter describing the window of time within which the number of deadlines must hold should be specified. Hence, the tolerance to missed deadlines is established within a window of $m > 0$ consecutive invocations of the task, which also correspond to a window of Tm units of time, where T is the period of a periodic task.

The effect of missed deadlines can be different depending on whether these deadlines are missed consecutively or non-consecutively. For instance, in a computer controlled system, missed deadlines result in loss of control performance. This performance loss depends on the total number of missed deadlines over a period of time. On the other hand, in digital audio systems, the quality of the output produced is more sensitive to the number of consecutive missed deadlines. If the same number of missed deadlines occur non-consecutively, its effects may not be noticeable. Consequently, it is required to introduce two classes of weakly-hard constraints that specify whether the deadlines can be missed consecutively or non-consecutively.

For symmetry purposes, it is also interesting to be able to specify the number of met deadlines as opposed the number of missed deadlines. For instance, in a certain control systems, after a deadline is missed it may be necessary to meet at least n deadlines in a row so that the state of the system is correctly updated. This introduces a further classification of weakly hard constraints.

The combination of the two considerations, (a) consecutiveness vs. non-consecutiveness, and (b) missed vs. met deadlines leads to four types of constraints ($n \geq 1, n \leq m$):

- A task τ "meets any n in m deadlines", denoted $\binom{n}{m}$, if, in any window of m consecutive invocations of the task, there are at least n invocations in any order that meet the deadline.

- A task τ "meets row n in m deadlines", denoted $\left\langle {n \atop m} \right\rangle$, if, in any window of m consecutive invocations of the task, there are at least n consecutive invocations that meet the deadline.

- A task τ "misses any n in m deadlines", denoted $\overline{\binom{n}{m}}$, if, in any window of m consecutive invocations of the task, no more of n deadlines are missed.

- A task τ "misses row n in m deadlines", denoted $\overline{\left\langle {n \atop m} \right\rangle}$, if, in any window of m consecutive invocations of the task, it is never the case that n consecutive invocations miss their deadline.

The term λ denotes a weakly hard constraint and Γ the set of all possible weakly hard constraints of these four types (see [3] for a formal analysis). Although weakly hard constraints with $n = 0$ could be defined, they have no useful meaning. For this reason, in the rest of the paper the case $n = 0$ is not considered.

We denote by a `0` a deadline missed and by a `1` a deadline met. We can therefore characterise a task by the sequence of zeros and ones (we formally introduce this concept and its properties in section 4). A task that has a pattern `11001101` does satisfy a $\binom{2}{4}$ because in any 4 consecutive symbols there are always at least 2 "`1`'s", however it does not satisfy $\binom{1}{2}$ because there is a window of two symbols without a `1`.

For example, the $\overline{\binom{n}{m}}$ constraint is equivalent to $\binom{m-n}{m}$, also for the $\overline{\left\langle {n \atop m} \right\rangle}$ constraint the parameter m is redundant, therefore, we would write simply $\overline{\langle n \rangle}$. Also, $\left\langle {n \atop m} \right\rangle$ is equivalent to a strongly hard ($\left\langle {n \atop n} \right\rangle$) if $2n - 1 \geq m$. Clearly, the (m, k)-firm corresponds to $\binom{m}{k}$.

Depending on the application model, the notion of "missed deadline" can be associated to: (1) Delayed completion: the task invocation runs until completion even though it finishes after the deadline. (2) Abortion: the task invocation 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. (3) Rejection: the task is not accepted into the system, so it does not even start to run. (4) Skip: the task invocation is not released and the whole invocation is not executed.

This categorisation is important because it identifies one of the potential applications of weakly-hard constraints: *resource adaptation*. A scheduler may decide to skip some task invocations in order to "make space" for new tasks during a peak load. Thus effectively reducing the instantaneous resource request (load).

3. Process model

The process model, from the implementation point of view, consists of a set Π, of N periodic and non-periodic tasks. Each task, τ_i is characterised by either its period in case of periodic tasks or minimum interarrival for non-periodic ones, T_i, deadline, D_i, worst-case execution time, C_i, a weakly hard constraint, λ_i, and a priority when it is scheduled in panic mode , P_i.

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

Each time that a task requires the processor to be executed, it will be said that task is either *invoked* or an *invocation takes place*. A system can contain periodic as well as non periodic tasks. However it has only sense to attach weakly hard constraints to periodic tasks or to sporadic tasks that arrive regularly (called repeating tasks). Sporadic tasks that occur rarely (called isolated tasks) should be considered strongly hard.

When an invocation of a task takes place, a state is assigned to it according to the following criterion:

- *Non-Critical State*: if the current invocation could be missed without jeopardising the satisfaction of the weakly hard real time constraints.

- *Critical State*: if the current invocation has to be met in order to prevent missing its weakly hard real time constraint.

The expressions *non-critical invocation* and *critical invocation* can be used to denote an invocation in non-critical state and critical state respectively. The scheduling approach therefore consists in a two level scheduler. A normal mode scheduler for non-critical tasks, and a scheduler for panic mode in which critical task invocations can be guaranteed to finish on time.

4. μ-Patterns and weakly hard constraints

The only information a scheduler for weakly hard systems uses on the history of the task (and possibly of the possible future of the tasks) is the pattern of zeros and ones that represent missed and met deadlines. We call these patterns a *μ-pattern*.

Definition 2 *(μ-pattern) A μ-pattern α of a task is a sequence of symbols of $\Sigma = \{\mathit{0}, \mathit{1}\}$ that characterises the execution of the task. $|\alpha| = p$ is the length of the pattern. $\alpha(k) \in \Sigma$, $1 \leq k \leq p$. A 1 means that a task has met its deadline, a 0 means that task has missed it. $\alpha(1)$ is the oldest invocation and $\alpha(p)$ is the most recent invocation.*

We will use the usual notation of regular expressions to manipulate μ-patterns, also $\alpha(a..b)$ is the sub string of α from position a to b. We will also call $\alpha(a..b)$ a window or subwindow of α.

For example, the μ-pattern `010011` has length 6. It has 3 subwindows of length 4: $\{\texttt{0100}, \texttt{1001}, \texttt{0011}\}$, where `0011` is the most recent subwindow.

We redefine the notion of task satisfying a weakly hard constraint in terms of μ-patterns. Clearly, a μ-pattern α, with $|\alpha| \geq m$, satisfies a weakly hard constraint λ, of window size m, and we write $\alpha \vdash \lambda$, if λ is satisfied in *each* subwindows of length m of α. For instance, for a $\binom{n}{m}$ constraint, it is satisfied if there are at least n `1`'s in any subwindow of the μ-pattern of length of m. If the constraint is not satisfied for a particular subwindow, then there is a dynamic failure. The satisfiability of the other types of constraint is defined similarly.

We are interested in a particular type of μ-pattern. We consider two on-line μ-patterns that hold, at each instant t the past of the execution the task, and an indication of the potential future behaviour of the task.

A *past μ-pattern* is a log of the real execution of a task. If α is a past μ-pattern then $\alpha(k) = 1$ means that the task *did meet* its deadline, and $\alpha(k) = 0$ means that the task *did miss* it (or was aborted/skipped). A *future* μ-pattern, β, is an estimation of the way the invocations in the future may occur.

To distinguish between the past μ-pattern and the future μ-pattern, symbols of the future μ-pattern will be denoted (borrowing the notation from [9]) by `r` (for red) as a synonym for `1` and `b` (for blue) as a synonym for `0`, whenever the distinction is useful. Thus at a given time t the system is characterised by the pattern $\alpha \cdot \beta$. For example:

$$\xrightarrow{\text{time}}$$
$$\overbrace{\texttt{111 010 111 010}}^{\text{past}(\alpha)}\overbrace{\texttt{rrr bbb rrr bbb rrr}}^{\text{future}(\beta)}$$

At the deadline of a particular invocation it is known whether that invocation actually met or missed the deadline. This is the adequate time to update the past μ-pattern.

It is easy to see that if λ is a weakly hard constraint, α is a μ-pattern of length $p \geq m$ such as $\alpha \vdash \lambda$ and $c \in \Sigma$ a symbol corresponding to the satisfiability of the next invocation, then in order to test the satisfiability of $\alpha' = \alpha \cdot c$, only the last subwindow of α', $\alpha'(p - m + 2..p + 1)$ needs to be considered. The other windows of length m in α' satisfied λ as they are the same windows of α. This means that for practical purposes, for a task that has a weakly-hard constraint λ of size m, the past μ-pattern only needs to hold the last m invocations of the task. For this reason, in the rest of the paper, we consider μ-patterns of length m (we do not require that all μ-patterns are of the same length, as each task may have a different weakly hard constraint, the length of its associated μ-pattern is also different). The operation of shifting the μ-pattern to the left to accommodate a new symbol c is very common and we will denote it by $\texttt{LSH}(\alpha, c)$. For example, $\texttt{LSH}(\texttt{101}, \texttt{1}) = \texttt{011}$.

The past μ-pattern of a task when it is first released is 0^m although no dynamic failure check should be performed until the firm m invocations of the tasks are finished.

In order to determine how important for satisfying the weakly hard constraint the following invocation is, we need a metric to measure how close is the μ-pattern to miss its weakly hard constraint. This is measured by the criticality function.

4.1. Criticality functions

We define the criticality of a μ-pattern as a function that determines the number of consecutive successive deadlines that a μ-pattern can miss without missing its weakly-hard constraint. As we do not assume any future behaviour of the task, we define the criticality as function of the past μ-pattern of the task only:

Definition 3 *(Criticality) Given a constraint λ with window size m and a μ-pattern α of length $|\alpha| = m$. A criticality function of α is $\delta^\lambda(\alpha) : \Sigma^m \to \mathbb{Z}$ which has the following properties (assume α, β are μ-patterns of length m):*

1. $\forall p \geq 0\ ,\ \alpha \cdot \mathit{1}^p \vdash \lambda \quad \Leftrightarrow \quad \delta^\lambda(\alpha) \geq 0$
2. $\exists p \geq 0 \mid \alpha \cdot \mathit{1}^p \not\vdash \lambda \quad \Leftrightarrow \quad \delta^\lambda(\alpha) < 0$
3. *let* $\beta = \mathit{LSH}(\alpha, \mathit{1})$, *then* $\delta^\lambda(\beta) \geq \delta^\lambda(\alpha)$
4. *let* $\beta = \mathit{LSH}(\alpha, \mathit{0})$, *then* $\delta^\lambda(\beta) \leq \delta^\lambda(\alpha)$

Criticality is positive or zero if the constraint is satisfied, and if it will be still satisfied in the future if all next deadlines are met. It is negative if it is not satisfied or, even assuming that all future deadlines are met, if the constraint will be not satisfied (this may happen for $\left\langle {n \atop m} \right\rangle$ constraints). The last two properties capture the fact that when the next deadline is met, then the criticality is at least as high as before. On the contrary, if the deadline is missed, the criticality either decreases or stays the same.

If $\delta^{\lambda}(\alpha) = 0$ for some μ-pattern α, then we say that the task is in a *critical state*. In this case the following task invocations have to meet their deadlines to guarantee the satisfiability of the weakly-hard constraint. When the task has a dynamic failure criticality is negative.

Several criticality functions can be defined for the four types of the weakly-hard constraints. We propose the following ones:

$$\delta^{\binom{n}{m}}(\alpha) = \begin{cases} g_n(\alpha) - 1 & \text{if } \sum_{i=1}^{m} \alpha(i) \geq n \\ \sum_{i=1}^{m} \alpha(i) - n & \text{otherwise} \end{cases} \tag{1}$$

where, $g_n(\alpha)$ is the rightmost point in α such that $\alpha(g_n(\alpha)..m)$ has n ones:

$$g_n(\alpha) = \max\left\{ q \mid 1 \leq q \leq m \ , \ \sum_{i=q}^{m} \alpha(i) = n \right\}$$

If the constraint is satisfied, the δ function holds how many deadlines can be missed in the future. If it is not satisfied it actually computes the number of overrun deadlines. The maximum value of $\delta^{\binom{n}{m}}(\alpha)$ is $m - n$, and the minimum is $-n$. For example, for a weakly hard constraint $\binom{3}{10}$ and a past μ-pattern $\alpha = \texttt{1010101001}$, then $g_3(\alpha) = 5$ and consequently $\delta^{\binom{3}{10}}(\alpha)$=4. This can be shown graphically as:

$$\overbrace{\underbrace{\texttt{1010}}_{g_3(\alpha)-1}\ \underbrace{\texttt{101001}}_{3\ 1\text{'s}}}^{m}$$

Criticality 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 `111 1111 000`, 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, the sequence becomes `1111 000 111`, when the next deadline is met, the 4 1's in a row occur at the right end of the sequence, therefore satisfying the constraint. If any of the last four deadlines was not met, then a dynamic failure would inevitably occur. Thus,

$$\delta^{\left\langle {n \atop m} \right\rangle}(\alpha) = \begin{cases} e_n(\alpha) - n & \text{if } e_n(\alpha) \geq n \\ e_n(\alpha) - n + z(1, \gamma(m, n, \alpha)) & \text{otherwise} \end{cases} \tag{2}$$

where $e_n(\alpha)$ is the left index of the rightmost sequence 1^n in α or :

$$e_n(\alpha) = \begin{cases} \max\{e \mid \alpha(e..e+n-1) = 1^n\} & \text{if } 1^n \in \alpha \\ 0 & \text{otherwise} \end{cases}$$

where $z(c, \alpha)$ is the number of consecutive symbols $c \in \Sigma$ that there are at the right end of the sequence.

$$z(c, \alpha) = \max\{p \mid \alpha(m - p + 1..m) = c^p\}$$

and $\gamma(n, m, \alpha)$ is the rightmost subsequence of α of length $n - e_n(\alpha)$:

$$\gamma(n, m, \alpha) = \alpha(m - n + e_n(\alpha) + 1..m)$$

This criticality function can be explained as follows. When there are only n symbols at the left of the rightmost 1^n pattern, then the criticality is zero. In the worst case n invocations need to be satisfied in order to have n consecutive 1's in the μ-pattern. The criticality is, actually, the number of deadlines that can be missed until the previous situation arises. For example, for a weakly hard constraint $\left\langle {2 \atop 10} \right\rangle$ and a past μ-pattern $\alpha = \texttt{0100111011}$, then $e_n(\alpha) = 9 (\geq 2)$ and consequently $\delta^{\left\langle {2 \atop 10} \right\rangle}(\alpha) = 7$. If the past μ-pattern is $\alpha = \texttt{1100101010}$, then $z(c, \alpha) = 0$ and consequently $\delta^{\left\langle {2 \atop 10} \right\rangle}(\alpha) = -1$. In this last case α satisfies $\left\langle {2 \atop 10} \right\rangle$ but its criticality is negative since a future dynamic failure cannot be avoided. When there are less than $n - 1$ symbols to the left of e_n then a dynamic failure is inevitable. The second expression is always negative and represents by how much the constraint was missed. For example, $\delta^{\left\langle {3 \atop 7} \right\rangle}(\texttt{0111000}) = -1$ because a dynamic failure will occur and it will last for 1 invocation.

As $\overline{\binom{n}{m}}$ is equivalent to $\binom{m-n}{m}$, the same criticality function for the *any* constraint can be used.

For the miss row constraint, a criticality function is defined as the number of consecutive zeros that can be tolerated:

$$\delta^{<\bar{n}>}(\alpha) = n - z(0, \alpha) \tag{3}$$

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

It is straightforward to see that the functions defined here are criticality functions. The scheduling framework that is described later in section 5 uses this notion of criticality to schedule tasks.

4.2. Minimal future μ-patterns

We now introduce a characterisation of the potential behaviour of the task invocations in the future. Let λ be a weakly hard constraint of size m, and let α be a past μ-pattern of a task τ of length m.

Definition 4 *(Guaranteed future μ-pattern) A future μ-pattern β is a guaranteed future μ-pattern for a weakly-hard constraint λ if for all possible past μ-patterns that have criticality greater than zero ($\delta(\alpha) \geq 0$), the combined μ-pattern $\gamma = \alpha \cdot \beta$ does not have a dynamic failure, $\gamma \vdash \lambda$.*

A guaranteed future μ-pattern captures the notion of a future behaviour of the task that is compatible with any possible past behaviour of that task, here compatible means that it will not generate a dynamic failure. For instance, the μ-pattern 1^m is a guaranteed future μ-pattern for any weakly-hard constraint.

Among all possible guaranteed future μ-patterns, we shall be interested in those that have the minimum number of 1's.

Definition 5 *(Minimal future μ-pattern) A guaranteed future μ-pattern β is a minimal future μ-pattern if when any single `1` in β is switched to a `0`, then the pattern is no longer a guaranteed future μ-pattern*

Minimal μ-patterns represent a possible future behaviour with a minimum number of invocation deadlines to meet. Among the multiple minimal future μ-patterns we are interested in the ones that allow a schedulability analysis to be performed.

The combination of the criticality function of the past μ-patterns and the notion of minimal future μ-pattern are the key for the scheduling approach. The criticality function is used to determine when a task is critical, the minimal future μ-pattern can then be used to provide a guaranteed schedulability test. For this purpose we are interested in defining a set of minimal future μ-patterns for each of the weakly-hard constraints so that response-time based schedulability tests can be defined for them. Among the possible minimal future μ-patterns, the following ones are adequate for the schedulability analysis described in section 6.

Definition 6 *For a constraint $\lambda = \binom{n}{m}$, a minimal future μ-pattern, $\hat{\omega}^\lambda$, is given by:*

$$\hat{\omega}^\lambda = \overbrace{\texttt{rr..r}}^{n}\overbrace{\texttt{bb..b}}^{m-n}\overbrace{\texttt{rr..r}}^{n}\overbrace{\texttt{bb..b}}^{m-n}\ldots$$

Similarly, for the $\left\langle {n \atop m} \right\rangle$ constraint:

Definition 7 *For a constraint $\lambda = \left\langle {n \atop m} \right\rangle$, a minimal future μ-pattern, $\hat{\omega}^\lambda$, is given by:*

$$\hat{\omega}^\lambda = \overbrace{\texttt{rr..r}}^{n}\overbrace{\texttt{bb..b}}^{m-2n+1}\overbrace{\texttt{rr..r}}^{n}\overbrace{\texttt{bb..b}}^{m-2n+1}\ldots$$

As the *miss any* constraint is equivalent to the *any* constraint, its minimal future μ-pattern is also equivalent. Finally, for the $\overline{\langle n \rangle}$ constraint:

Definition 8 *For a constraint $\lambda = \overline{\langle n \rangle}$, a minimal future μ-pattern, $\hat{\omega}^\lambda$, is given by:*

$$\hat{\omega}^\lambda = \texttt{r}\overbrace{\texttt{bb..b}}^{n-1}\texttt{r}\overbrace{\texttt{bb..b}}^{n-1}\ldots$$

It is straightforward to prove that the μ-patterns defined above are guaranteed μ-patterns.

4.3. Critical state invocations

In a weakly hard-scheduled system, every time a task is released, the scheduler should need to determine whether the invocation is in a critical state. The invocation of a task is in a critical state if the criticality function of its past μ-pattern is zero. Note that because of the way the criticality function is defined, to determine whether a particular invocation of a task with a constraint λ of length m is critical it is only necessary to examine the past μ-pattern. The following theorem gives the condition to guarantee that no dynamic failures will occur.

Theorem 1 *Any weakly-hard scheduling algorithm that guarantees the deadline of an invocation when its criticality is zero does never produce a dynamic failure.*

Proof: Immediate from property 3 of the definition of criticality function.

The theorem states that if there is a mechanism by which the next release of a task can be guaranteed to finish by its deadline when a task has zero criticality, then no dynamic failure can occur. There are two key points to consider: No constraint is put in the way that tasks are scheduled, only that a task has to be guaranteed to finish by its deadline. Secondly, only the past μ-pattern of the last m invocations needs to be considered to determine whether the next invocation is in a critical state.

5. Scheduling with weakly hard Constraints

After having introduced the theory behind on-line weakly-hard schedulers we can now present a scheduling framework that is flexible, dynamic and that guarantees that no dynamic failure ever occurs.

A guaranteed scheduler must perform a well-known behaviour leading to avoid any dynamic failure. However, when weakly hard constraints are satisfied, the scheduler must allow tasks to be scheduled under any arbitrary discipline in order to maximise a certain criterion of performance of the specific system. For this purpose we present a Bi-Modal Scheduler (BMS). It has two operating modes: Panic mode and Normal mode.

- When a task *i* is in panic mode it is run under a fixed priority mechanism with a priority P_i strictly higher than any other task which is not in panic mode. In this mode the maximum interference a task may suffer from other tasks can be precisely measured and therefore an schedulability test can be defined (See sec. 6).
- When a task is in normal mode it can be scheduled according to any arbitrary discipline, possibly a best-effort one. Scheduling disciplines based on minimising the response time of tasks or a discipline to avoid entering in panic mode or specific-system disciplines can be applied. For example the simple EDF may suffice, other approaches that are fairer when deadlines are missed, or specialised scheduling approaches based on the notion of criticality can be defined. In this paper we do not investigate normal mode schedulers any further. This is the subject of a future work.

The scheduler has to change to panic mode when any weakly hard constraint is in danger of being missed. This can be done as soon as a task with zero criticality is invoked. However, it is possible to delay such transition as there is slack in the system. A critical task i will be scheduled under normal mode until a time t, in what follows denoted L_i, in which if it is not put into panic mode there is the risk of missing the deadline. The former approach is called *immediate panic mode*, whereas the latter is called *delayed panic mode*. The scheduler has to return to normal mode, as soon as possible but only when there is no risk to miss a weakly hard constraint. Obviously, non-critical invocations will not produce a change to panic mode of the scheduler, because they are not close to generate a dynamic failure.

The scheduling framework can be summarised as follows:

- There are two scheduling policies defined, a panic mode scheduler and a normal mode scheduler. The panic mode is implemented as a fixed priority mechanism with priorities strictly higher than any task scheduled under the normal mode scheduler. No restriction is placed on the normal mode scheduler.
- Whenever a task is invoked, its criticality is computed. If criticality is zero then task invocation may be in panic mode. Otherwise it is in normal mode.
- The transition from normal mode to panic mode is simply implemented as a priority promotion. We use the dual priority mechanism [6] to achieve that. Other slack management techniques could be also used for this purpose.
- Whenever a task finishes (or at the deadline if the task has not finished by the deadline) the past μ-pattern is updated accordingly.
- If there are no task in panic mode then the scheduler may choose a ready task to execute according to any scheduling policy. We do not impose the way to schedule tasks in non-panic mode. Although we provide some criticality based ones later.

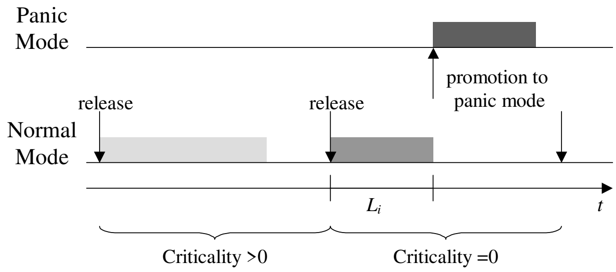

Figure 1. Example of delayed panic mode.

Figure 1 shows an example of the interaction of the two schedulers. The first invocation is considered non-critical as its criticality is greater than 0. Therefore, it is scheduled in normal mode and it may or may not meet its deadline depending on the scheduling discipline used in normal mode. Assuming that the deadline is missed, the second invocation is critical because its criticality has dropped to zero. Under the delayed panic mode, the tasks starts running under normal mode but as it has not finished by L_i, it is promoted to panic mode where it is guaranteed that it will finish by the deadline and therefore a dynamic failure is prevented. The computation of L_i and the schedulability test is developed in the following section.

6. Schedulability analysis

In this section, a schedulabity analysis is performed to guarantee that task invocations in panic mode will always meet its deadline.

When a task is in panic mode, it may only suffer interference from other invocations which are also in panic mode. In fact, as tasks are scheduled using fixed priorities, each task *i* has assigned a fixed priority, P_i, for panic mode and can only suffer interference from other task invocations which are also in panic mode and with higher priority. With this scenario, the worst case of requirements that panic invocations can produce must be determined.

Lemma 1 *A task invocation in panic mode may receive, at most, interference from the red invocations of the minimal future μ-pattern of higher priority tasks.*

Proof: From the definition of minimal future μ-pattern, all blue invocations correspond to non-panic releases as the criticality function at the release of the blue invocation is strictly greater than zero. Only red invocations may become panic, and therefore these are the only ones that can produce interference on other (lower priority) panic invocations.

Lemma 2 *The worst case response time for a task invocation in panic mode happens when it is simultaneously released with all higher priority tasks in panic mode and all these higher priority tasks have the minimum criticality.*

Proof: The worst case response time of a task will occur when it will suffer the maximum interference from higher priority tasks. When all tasks with higher priority in panic mode are released simultaneously, it will be preempted at most for each of their red releases in their minimal future μ-pattern. Moreover, because of the way the minimal future μ-patterns are defined, they correspond to the conditions of multiframe tasks [10], for which the definition of critical instant is equivalent.

Lemma 3 *The worst case interference a task τ_i with weakly-hard constraint λ_i and minimal future μ-pattern $\hat{\omega}^\lambda$ may produce on lower priority tasks in any window of time of length t, denoted by $W_i(t)$, is given by:*

$$W_i(t) = \sum_{k=1}^{\left\lceil \frac{t}{T_i} \right\rceil} \hat{\omega}_i^\lambda(k) C_i$$

Proof: From theorem 1, the task only produces interference from its red ($\hat{\omega}_i^\lambda(k) = 1$) invocations. The maximum interference is obtained when the task is released simultaneously with all higher priority tasks (Lemma 3), therefore from the high priority task point of view, it will produce at most C_i units of interference every period that this task is red.

Lemma 4 *The worst case response time, denoted by R_i, of a task invocation in panic mode is the smallest $R_i > 0$ which is a solution to the fixed point equation:*

$$R_i = C_i + \sum_{j \in \mathsf{hp}(i)} W_j(R_i)$$

where $\mathsf{hp}(i)$ denotes the set of tasks of higher priority than task τ_i in panic mode. R_i can be computed with a recurrence formula [1]: $R_i^0 = 0$, $R_i^{n+1} = C_i + \sum_{j \in \mathsf{hp}(i)} W_j(R_i^n)$. The iteration stops when $R_i^{n+1} = R_i^n$ or when $R_i^n > D_i$.

Theorem 2 *For a system Π of weakly hard real time tasks scheduled according to the scheme described in section 5, all the weakly hard real time constraints can be met (and therefore no dynamic failure can occur) in panic mode if*

$$\forall \tau_i \in \Pi \;\; R_i \leq D_i$$

Proof: The criticality function guarantees that a task is in normal mode only when it does satisfy its weakly hard constraint. Therefore a task in normal mode does never have a dynamic failure. If it does not produce a dynamic failure in panic mode (theorem 1), then no dynamic failure can occur.

Corollary 1 *A system Π does never produce a dynamic failure if critical invocations are promoted to panic mode at least R_i slots before their deadline ($L_i \leq D_i - R_i$).*

This final corollary gives an interesting result. Independently on how tasks are scheduled in normal mode, if they are promoted to panic mode with enough time to complete when their criticality is zero and a schedulability test exists in panic mode, then no dynamic failure can ever occur.

Also note that the guarantee is based on the ability of computing the maximum interference a task may receive from higher priority tasks. This has been done with an extension of response time analysis techniques. This formulation can be easily extended to include other factors like blocking factors due to the operation of a protocol for sharing resources like the priority ceiling protocol [12], release jitter effects, kernel overheads [1], etc.

The examination of Lemma 3 has an additional important implication. For weakly hard systems the load at the critical instant is significantly lowered. Thus systems that with traditional schedulability analysis techniques would be found to be unschedulable they can be weakly hard schedulable. Most importantly, experimental evaluation suggests that although the schedulability analysis indicates that some deadlines can be missed, the pessimism in the WCET analysis leads to all task deadlines actually being met.

Panic invocations are scheduled under a fixed priority discipline. Therefore, a priority must be assigned to panic invocations of each task. It is easy to find counterexamples where deadline monotonic or rate monotonic assignments are not optimal when tasks have weakly hard constraints. An optimal priority assignment can be found using the partition method. The details can be found in [3]

7. Evaluation

In this section we present two examples of the applicability of weakly hard constraints: an illustration of the reduction of resource requests at peak loads and a comparison of the number of dynamic failures of other approaches.

Table 1. Example Task set. Total Utilisation $= 1.2$, but weakly hard schedulable under BMS

Task	T_i	C_i	D_i	λ_i	P_i
1	45	22	45	$\binom{2}{4}$	1 (highest)
2	70	22	70	$\binom{4}{6}$	2
3	245	54	245	$\binom{1}{1}$	3
4	1200	198	1200	$\binom{1}{1}$	4 (lowest)

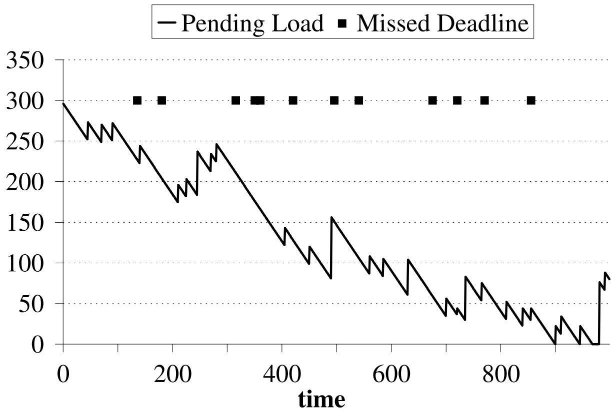

Figure 2. Pending load and distribution of missed deadlines

7.1. Resource requests

Consider the system shown in table 1. It is not strongly hard schedulable because it has a total utilization factor of 1.2. However, it meets the conditions of theorem 2, it is schedulable under a BMS scheduler.

Figure 2 shows the trace of the pending load (number of requirements pending considering maximum execution time of each task) if an EDF scheduling algorithm in normal mode is used (assume $L_i = 0$). It can be noted that load has a peak at $t = 0$ but later pending load decreases gradually. Some high priority tasks miss their deadlines, thus effectively leaving more resources available. After the initial busy period, the number of missed deadlines decreases significantly and its distribution is more sparse.

7.2. Comparison

Previous attempts on weakly hard real time systems were focused on application-specific scenarios which makes them difficult to apply to different contexts. Skip-over scheduler are guaranteed, online schedulers but the temporal constraints that they can manage are very restrictive. On the other hand, (m, k)-firm systems are based on best-effort schedulers and no guarantee can be obtained. We compare

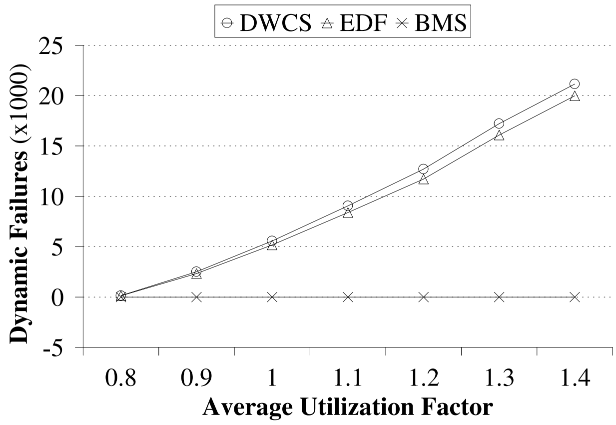

Figure 3. Comparison of the number of dynamic failures for DWCS, EDF and BMS

our BMS scheduler with the DWCS approach for (m, k)-firm constraints [14].

A set of one thousand systems of 20 tasks each were randomly generated. Task periods were generated uniformity distributed between 10 and 500. Total utilisation factor was set to 1.4 for each system, but average execution times were randomly decreased to achieve an average utilisation factor varying between 0.8 and 1.4 (when average utilisation factor is 1.4, average execution time was just equal to the maximum execution time). Weakly hard constrains were introduced to reduce the weakly hard utilisation factor down to 0.7. Task sets that were not schedulable (theorem 2) were rejected. Only constraints of *any* type were chosen because they are the only constraints supported by DWCS.

Simulations were carried for at least 1000 invocations of the task with the longest period. Execution times were generated assuming an exponential distribution. Figure 3 shows the average number of dynamic failures that occur for each group of utilisation factors for the DWCS, simple EDF and for BMS with EDF as the scheduler in normal mode. Although under the “fixed window” assumption of the DWCS approach there should not be dynamic failures, in the more restricted model of sliding windows several dynamic failures do occur. Moreover there are even more dynamic failures than with a simple EDF approach. As expected, no dynamic failure takes place if systems are schedule under the BMS algorithm.

Figure 4 shows the performance of each scheduling algorithm, based on the effective utilisation factor. The effective utilisation factor is defined as the processor time (per unit) used by invocations that meet their deadlines. When load is low, deadline based algorithms (DWCS is a pseudo-deadline based one) achieve the maximum effective utilisation factor (best-effort algorithms try to optimise the maximum average effective utilisation factor). However, when

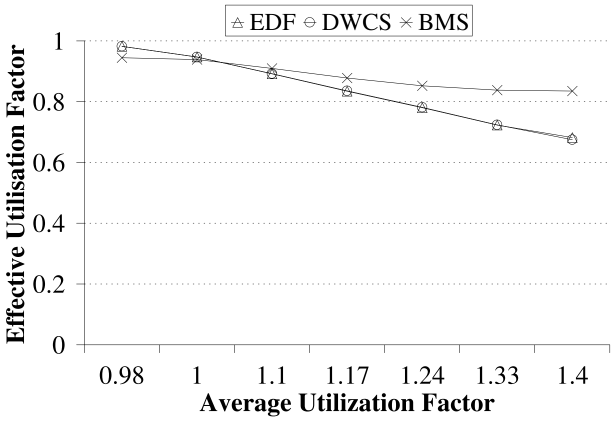

Figure 4. Effective utilisation factors for EDF, DWCS and BMS with EDF as the scheduler in normal mode.

load increases deadline based algorithms performance starts to decrease because of the domino effect. Although BMS does not perform as well under low loads as the two other approaches, it achieves a consistent high load even for high total utilisation factors. The low performance may be due to the underlying EDF scheduler used in normal mode. Other schedulers in normal mode may produce better results. This is the topic of current ongoing research.

8. Conclusions

Future real-time systems require mechanisms to specify in a clear, predictable, and bounded way that some deadlines can be missed [3]. We use weakly-hard constraints to specify such bounds which are a generalisation of the concept of $(m, k)-$firm deadlines. Also, efficient and robust scheduling techniques that address weakly hard real-time systems should be available. Previous attempts have not been satisfactory as they are not guaranteed, too constrained or applicable to reduced application contexts only.

In this paper, a general scheduling framework, called BMS, is proposed for weakly hard real time systems. It is based on a simple and robust mechanism that has two modes of operation. Whenever there is the risk that a task may not satisfy its weakly-hard constraint the scheduler enters in panic mode for which schedulability tests exists that guarantee that deadlines will be always met. When there is no such a risk, a generic scheduler can be used. Due to the inherent pessimism of the analysis it is very likely that the scheduler would rarely switch to panic mode.

With weakly hard constraints the resource requirements around the critical instant are lowered, thus allowing systems with a potential strongly hard utilisation greater than 100% to be scheduled with weakly-hard constraints. Due to the pessimism in the WCET analysis it is expected that most deadlines are indeed met.

References

[1] N. Audsley, A. Burns, M. Richardson, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] G. Bernat and A. Burns. Combining (n m)-hard deadlines with dual priority scheduling. In *18th IEEE Real-Time systems symposium. RTSS San Francisco, CA*, December 1997.

[3] G. Bernat, A. Burns, and A. Llamosí. Weakly-hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001.

[4] G. Buttazzo and M. Caccamo. Minimizing aperiodic response times in a firm real-time environment. *IEEE Transactions on Software Engineering*, 25(1):22–32, January 1999.

[5] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *17th IEEE Real-Time Systems Symposium. San Francisco, CA.*, pages 330–339, December 1997.

[6] R. Davis and A. Wellings. Dual priority scheduling. In *proc. 16th IEEE Real-Time Systems Symposium*, pages 100–109, December 1995.

[7] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[8] M. Hamdaoui and P. Ramanathan. Evaluating dynamic failure probability for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 46(12):1325–1337, December 1997.

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

[10] A. Mok and D. Chen. A multiframe model for real-time tasks. In *proc. 17th IEEE Real-Time Systems Symposium*, December 1996.

[11] G. Quan and X. Hu. Enhanced fixed-priority scheduling with (m, k)-firm guarantee. In *12th IEEE Euromicro Conference on Real-Time Systems*, Sweden, June 2000.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[13] H. F. Wedde, J. A. Lind, and G. Segbert. Distribuited real-time task monitoring in the safety-critical system melody. In *11th IEEE Euromicro Conference on Real-Time Systems*, pages 158–165, York. UK, June 1999.

[14] R. West and C. Poellabauer. An optimal, on-line window-constrained scheduler for real-time heterogeneous activities. Technical Report GIT-CC-99-11, College of Computing, Georgia Institute of Technology, Atlanta, 1999.

[15] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *21st IEEE Real-Time Systems Symposium. Florida. USA*, November 2000.