Response Time Analysis of Asynchronous Periodic and Sporadic Tasks Scheduled by a Fixed-Priority Preemptive Algorithm

Manuel Coutinho
EDISOFT* / IST-UTL†
manuel.coutinho@edisoft.pt

José Rufino
FCUL‡
ruf@di.fc.ul.pt

Carlos Almeida
IST-UTL
cra@comp.ist.utl.pt

Abstract

Real-time systems usually consist of a set of periodic and sporadic tasks. Periodic tasks can be divided into two classes: synchronous and asynchronous. The first type does not define the task first release, contrary to the second. Hence, synchronous periodic tasks are assumed to be released at the worst instant: the critical instant. The schedulability test is reduced to check a single execution of the task under analysis. The integration of sporadic tasks is also straightforward: they are treated as a periodic task with maximum arrival frequency. On the other hand, asynchronous periodic tasks require a test for each release in the hyperperiod and the integration of sporadic tasks is not trivial: the worst release instant is unknown a priori. However, they do not assume that the tasks are released at the worst instant.

This paper presents a new schedulability analysis method based on the Response Time Analysis (RTA) to determine the worst response time of both asynchronous periodic and sporadic tasks, scheduled by a fixed-priority preemptive algorithm with general deadlines. It also presents another method that enables the introduction of a user configurable degree of pessimism, reducing the hyperperiod dependency.

1. Introduction

Traditionally, real-time system designers choose to implement both periodic and sporadic tasks due to their determinism and responsiveness. Periodic tasks are used typically for process control (e.g. attitude control in space systems) whereas sporadic tasks provide very fast responses to external events (e.g. airbag in automotive applications).

Two consecutive releases of a periodic task are separated by a well-known **fixed** time interval: the task period. On the other hand, sporadic tasks have a more non deterministic behaviour: two consecutive releases are separated by a well-known **minimum** time interval.

Periodic tasks can be divided into two main classes: synchronous and asynchronous[1]. This difference relates to definition (or lack of) of the first release instant. Sporadic tasks usually react to external events so there is little advantage in specifying their first release time.

The first release instant is **undefined** in synchronous periodic tasks. Hence, the schedulability analysis has to consider the worst possible release: the critical instant. Therefore, only one release has to be studied, allowing pseudo-polynomial tests to produce necessary and sufficient conditions [11, 12]. The integration of sporadic tasks with synchronous periodic tasks is straightforward: they are treated as synchronous periodic tasks, released at a critical instant and with period equal to its minimum inter-arrival time (maximum frequency).

On the other hand, the first release instant is **defined** in asynchronous periodic tasks. These are more difficult to analyze since the worst release instant is unknown *a priori*. The determination of sufficient and necessary schedulability conditions of asynchronous periodic tasks in a preemptive fixed-priority scheduling algorithm has been shown to be co-NP-complete [13]. In fact, an analysis through the task hyperperiod is required to determine that all releases meet their deadline [1]. Since the hyperperiod increases exponentially with the number of tasks, specially those with prime periods, this analysis can be heavy, even with few tasks. Furthermore, sporadic tasks are more difficult to analyze since the worst release instant is also unknown.

*Rua Quinta dos Medronheiros - Lazarim, Apartado 382, 2826-801 Caparica, Portugal. Tel: +351-21-2945900 - Fax: +351-21-2945999.

†Instituto Superior Técnico - Universidade Técnica de Lisboa, Avenida Rovisco Pais, 1049-001 Lisboa, Portugal. Tel: +351-21-8418397 - Fax: +351-21-8417499. This work was supported by EU and FCT, through Project POSC/EIA/56041/2004 (DARIO) WWW Page - http://pandora.ist.utl.pt/projects/dario.

‡Faculdade de Ciências da Universidade de Lisboa, Campo Grande - Bloco C8, 1749-016 Lisboa, Portugal. Tel: +351-21-7500254 - Fax: +351-21-7500084. This work was supported by EU and FCT, through Project POSC/EIA/56041/2004 (DARIO) and through the FCT Multiannual Funding Programme. This work was partially motivated by the research results of Project AIR, supported by ESA (European Space Agency) through the ITI program, ESTEC Contract 19912/06/NL/JD.

[1] Asynchronous periodic tasks are sometimes referred as concrete periodic tasks in the domain of non-preemptive schedulers [10].

However, asynchronous periodic tasks have an important advantage over the synchronous counterpart: they are not released at the critical instant. Hence, this generic methodology may be used to accurately represent real settings [16, 19, 18]. In addition, there can co-exist tasks that never interfere with each other, even with different periods. The use of asynchronous an periodic task model allows an increase of CPU usage and can guarantee higher system responsiveness thus avoiding the costs of buying and certifying faster equipment [16].

The hyperperiod dependency is also present in the time-triggered architecture, which to some degree demonstrates the practicability of this type of analysis. However, even though these systems generally allow a higher CPU usage, they have a low responsiveness to external events.

This paper proposes a method to analyze the schedulability of systems comprising asynchronous periodic and sporadic tasks with general deadlines (deadlines less than or equal to the period), scheduled by a fixed-priority preemptive algorithm. As far as our knowledge can say, the schedulability analysis of sporadic tasks together with asynchronous periodic tasks has not been addressed yet in the literature. The proposed methods follow the Response Time Analysis (RTA) approach, determining the response times of all periodic releases throughout the system hyperperiod. The determination of the worst response times also allows a greater insight to the amount of jitter associated to a periodic task and the number of missed deadlines (assuming the response time is larger than the deadline but smaller than the period). In control systems this is useful because, if not carefully bounded, the jitter can potentially degrade the system performance [17].

The worst release instant caused by the asynchronous task set, the *asynchronous critical instant*, is unknown *a priori*. This paper presents a method that determines a set of candidates to the asynchronous critical instant to analyze the schedulability of sporadic tasks.

Since many real-time embedded systems have large designing/implementing periods, the heavy schedulability analysis required is still manageable in most systems, as seen in the time-triggered architecture usage cases. However, if the hyperperiod is too large, another method is presented that produces a faster result at the expense of the introduction of a variable degree of pessimism.

2. Related work

The schedulability analysis has commonly used the critical instant as a means to produce very fast results. In fact, in systems where the release instants are not known *a priori* derive necessary and sufficient conditions, as the RTA [11]. This analysis gives the response time of a given task rendering the schedulability test trivial: the response time must be smaller than the deadline.

Attempts to remove the critical instant assumption by defining release instants for periodic tasks have produced some results. Perhaps the most widely known result is produced by the offset system model [3, 15, 4, 20, 14]. In this model, there is a set of periodic transactions, each one composed by a set of tasks (with the same period as the transaction and normally related by a precedence order). The tasks are released at a fixed time interval after the transaction has begun but the release of the transaction is unknown. As a consequence, tasks inside the same transaction have little or no interference with one another. However, tasks in different transactions are analyzed assuming they are released at the worst instant. Furthermore, tasks inside a transaction must share the same period. As a consequence, the hyperperiod is typically small (smaller number of tasks with different periods). Note also that this offset analysis can be made a particular case of asynchronous periodic tasks with the additional definition of the release instants of the transactions.

Regarding asynchronous periodic tasks, schedulability analysis has been addressed in [1] and in [6, 8]. In [1] it is presented an optimum priority assignment algorithm for fixed-priority schedulers with $O(n^2)$ complexity. This algorithm requires a sufficient and necessary schedulability test (also known as a feasibility test) which is also presented. However, this analysis determines the interference of each individual higher priority task release, which requires a sorting procedure and a high number of iterations. In [5] the author treats both asynchronous periodic and sporadic tasks, but only provides a sufficient test when considering sporadic tasks.

Both [6] and [8] determine the response time in dynamic priority schedulers with arbitrary deadlines (deadlines can be larger than the period), whereas fixed-priority scheduling is briefly addressed in [7]. However, this work can be simplified when applied to fixed-priority systems.

Other work has addressed offset-free systems, characterized by allowing the scheduler to choose the best (first) release instants of the periodic tasks. In [9] near-optimal priority and release assignments are produced using a heuristic and using the schedulability test described in [1, 2].

Fixed-priority non-preemptive schedulers have been considered in [10], where it is demonstrated that the complexity of the schedulability analysis is NP-hard in the strong sense, but it did not provide a schedulability test.

3. System Model

The system model considered in this paper integrates both periodic and sporadic tasks, scheduled by a fixed-priority preemptive algorithm. It is assumed that the tasks are independent, co-exist in a single processor and the con-

text switching overhead is null. Each task has a unique priority but periodic and sporadic can intertwine their relative priorities, that is, the priority of a given sporadic task can be in between the priorities of two other periodic tasks.

Concerning only the periodic task set, it is commonly referred as an asynchronous task set with general deadlines. In this model, the periodic tasks are defined by the parameters $\Gamma_i = \{c_i, T_i, r_i, D_i\}$ where c_i represents the Worst Case Execution Time (WCET), T_i the task period, r_i the release time of the task and D_i the relative deadline from the task release. The condition of general deadlines specifies that the deadline is smaller than or equal to the period. Hence, the parameters are conditioned to $0 \leq c_i \leq D_i \leq T_i$ and $0 \leq r_i$. A job is a particular execution of a periodic task.

Similarly, the sporadic task set is characterized by $\tau_m = \{e_m, MIT_m, H_m\}$ where e_m is the WCET, MIT_m is the Minimum Inter-Arrival Time between consecutive task releases and H_m is the relative deadline from the task release. Similarly, the deadline is inferior to the minimum inter-arrival time, hence $0 \leq e_m \leq H_m \leq MIT_m$.

Both tasks sets are ordered by increasing priority, that is, task Γ_i has a higher priority than task Γ_{i+1} and τ_m has also a higher priority than τ_{m+1}.

Additional Definitions

- $\overline{R}_i$ - Response time (instant) of the synchronous periodic counterpart of the asynchronous periodic task Γ_i
- ${}^l r_i = r_i + l\,T_i$ - Release instant of the lth job of the asynchronous periodic task Γ_i
- ${}^l R_i$ - Response instant of the lth job of the asynchronous periodic task Γ_i
- R_m - Response instant of the sporadic task τ_m
- ${}^l R_i - {}^l r_i$ - Response time of the lth job of an asynchronous periodic task Γ_i
- Λ_i - Hyperperiod of the asynchronous periodic task Γ_i

4. Response Time Analysis for Synchronous Tasks

The Response Time Analysis (RTA) for synchronous periodic tasks and sporadic tasks is a well-known fast method that produces necessary and sufficient schedulability conditions [11]. This section presents the essential details of how this method works, since it is used as a starting point of the analysis described latter on.

In synchronous periodic tasks, the worst instant where the tasks can be released is at a critical instant: all higher priority tasks start at the same instant. Hence, the response time of the synchronous counterpart of Γ_i, given by $\overline{R}_i$, is the solution of the equation

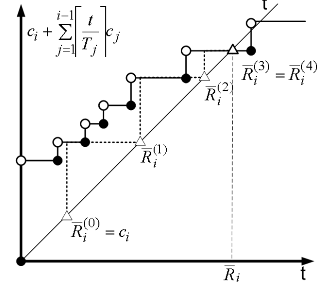

Figure 1. Iterations of the Response Time Calculation, Equation (2)

$$\overline{R}_i = c_i + \sum_{j=1}^{i-1} \left\lceil \frac{\overline{R}_i}{T_j} \right\rceil c_j \quad (1)$$

Due to the non-linear ceiling operator $\lceil x \rceil$, this equation can not be easily solved from direct analysis. Instead, an iterative fixed-point method is used to determine its value

$$\begin{cases} \overline{R}_i^{(0)} & = & c_i \\ \overline{R}_i^{(n+1)} & = & c_i + \sum_{j=1}^{i-1} \left\lceil \frac{\overline{R}_i^{(n)}}{T_j} \right\rceil c_j \end{cases} \quad (2)$$

The basic philosophy behind this iterative method is to incrementally increase the response time based on the higher priority task demanded workload. Each iteration gives the instant where the workload request has been fulfilled. However, between the last iteration and the present one, new workload may have been demanded. Hence, the algorithm continues to update the new workload demanded until it stops increasing. At this point the method halts, where $\overline{R}_i^{(n+1)} = \overline{R}_i^{(n)} = \overline{R}_i$. The schedulability test is reduced to check if $\overline{R}_i \leq D_i$. If so, then the task is schedulable. Figure 1 shows an example of how the iterations progress until the response time is found.

The integration of sporadic tasks is straightforward within this model: they are treated as periodic tasks, released at the critical instant and with period equal to their MIT. As such, there is little difference analysing synchronous periodic or sporadic tasks.

5. Response Time Analysis of Asynchronous Periodic Tasks

This section provides a new method to determine the response time of all asynchronous periodic jobs. For simplicity of exposition, sporadic tasks are not yet considered.

The introduction of the release instant of periodic tasks has to be firstly reflected into the workload function. Hence, the workload requested in the interval $[0, t[$ by the tasks $\Gamma_1, \ldots, \Gamma_i$ is given by

$$w_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t - r_j}{T_j} \right\rceil_0 c_j \tag{3}$$

where the $\lceil x \rceil_0$ operator gives the $max\{0, \lceil x \rceil\}$. This operator is introduced to reflect that the number of times a task is invoked cannot be lower than zero.

The response instant of an asynchronous periodic task released at ${}^l r_i$ is obtained by adding its execution time, c_i, and the interference from the higher priority tasks, $I_{i-1}({}^l r_i, {}^l R_i)$, to the instant it was released, ${}^l r_i$

$${}^l R_i = {}^l r_i + c_i + I_{i-1}({}^l r_i, {}^l R_i) \tag{4}$$

The interference from higher priority tasks is the only unknown element. This interference can be divided into the interference remaining at ${}^l r_i$, designated by $I^b_{i-1}({}^l r_i)$, and the interference that arises after the task release until its response instant, $I^a_{i-1}({}^l r_i, {}^l R_i)$.

$$I_{i-1}({}^l r_i, {}^l R_i) = I^b_{i-1}({}^l r_i) + I^a_{i-1}({}^l r_i, {}^l R_i) \tag{5}$$

The term $I^b_{i-1}({}^l r_i)$ reflects the interference from higher priority tasks requested before the release instant. In [1], this term is calculated by searching backwards all the higher priority task releases in the interval $[{}^{l-1} r_i + D_i, {}^l r_i[$ plus the workload remaining at ${}^{l-1} r_i + D_i$. This set of task releases is then sorted chronologically and individually analyzed.

Instead, our approach requires the determination of the last idle instant before ${}^l r_i$, designated by $L_{i-1}({}^l r_i)$. This last idle instant is defined as the last instant where all higher priority tasks previously demanded workload has been fulfilled. Hence, from $L_{i-1}({}^l r_i)$ to ${}^l r_i$ the system is processing the higher priority tasks. The remaining interference is easily found by subtracting the workload demanded with the workload processed by the higher priority tasks in the interval $[L_{i-1}({}^l r_i), {}^l r_i[$.

$$I^b_{i-1}({}^l r_i) = \underbrace{w_{i-1}({}^l r_i) - w_{i-1}(L_{i-1}({}^l r_i))}_{\text{workload demanded}} - \underbrace{({}^l r_i - L_{i-1}({}^l r_i))}_{\text{workload processed}} \tag{6}$$

As will be shown later on, the calculation of the last idle instant follows a method similar to the RTA, so it does not analyze each higher priority task prior release individually, nor does it require a sorting operation.

The term $I^a_{i-1}({}^l r_i, {}^l R_i)$ accounts for the total workload request by the higher priority tasks after the release, that is, in the interval $[{}^l r_i, {}^l R_i[$. In [1] the author determines all the higher priority tasks releases in the interval $[{}^l r_i, {}^l r_i + D_i[$. This set of releases is ordered chronologically and individually analyzed. The interference found at ${}^l r_i + D_i$ is used to calculate the remaining interference for the next job, $I^b_{i-1}({}^{l+1} r_i)$, as previously stated.

Our approach determines the interference after the release until the response instant, corresponding to the interval $[{}^l r_i, {}^l R_i[$ and it is given by

$$I^a_{i-1}({}^l r_i, {}^l R_i) = w_{i-1}({}^l R_i) - w_{i-1}({}^l r_i) \tag{7}$$

Joining equations (4), (5), (6) and (7), the response instant is given by

$${}^l R_i = L_{i-1}\left({}^l r_i\right) + c_i + w_{i-1}\left({}^l R_i\right) - w_{i-1}\left(L_{i-1}\left({}^l r_i\right)\right) \tag{8}$$

The smallest solution (larger than ${}^l r_i$) of this equation gives the response instant of a task released at ${}^l r_i$. The equation starts from the last idle instant, as opposed to ${}^l r_i$ in equation (4). It then adds the task execution time to the workload demanded by the higher priority tasks in the interval $[L_{i-1}({}^l r_i), {}^l R_i[$.

This equation has to be verified for all jobs of Γ_i released inside the hyperperiod. In [1] the hyperperiod is proven to correspond to the interval

$$[\max(r_1, ..., r_i), \max(r_1, ..., r_i) + LCM(T_1, ..., T_i)[$$

where the *LCM* function corresponds to the Least Common Multiple. The hyperperiod is the minimum interval in which the task set starts repeating itself. Note that it starts after a initial transient phase. This initial transient phase does not need to be analyzed since there is less interference than the worst case (not all tasks have been made active). Hence, the hyperperiod can start at any instant after $\max(r_1, ..., r_i)$ as long as it maintains the same length. Due to the sporadic task interference, we begin the hyperperiod at $S_i = \max(r_1, ..., r_i) + T_i$

$$\Lambda_i = [S_i, S_i + LCM(T_1, ..., T_i)[\tag{9}$$

This new interval has the same length and starts after the transient phase, so it maintains the same properties. Section 6.2 explains the reasons for the change made to the hyperperiod. In short, it is necessary to analyze the interval $]{}^{l-1} r_i, {}^l r_i]$ to determine ${}^l R_i$, so this interval must not contain a transient phase.

The minimum and maximum l such that ${}^l r_i \in \Lambda_i$ are given by

$$\begin{aligned} l_i^{min} &= \left\lceil \frac{S_i - r_i}{T_i} \right\rceil \\ l_i^{max} &= \left\lceil \frac{S_i + LCM(T_1, ..., T_i) - r_i}{T_i} \right\rceil - 1 \end{aligned}$$

The schedulability analysis can be summarized in the condition

$$\begin{cases} \forall_{l: {}^l r_i \in \Lambda_i} \quad {}^l R_i - {}^l r_i \le D_i \quad , \quad \Gamma_i \text{ is } \textbf{schedulable} \\ \exists_{l: {}^l r_i \in \Lambda_i} \quad {}^l R_i - {}^l r_i > D_i \quad , \quad \Gamma_i \text{ is } \textbf{unschedulable} \end{cases} \tag{10}$$

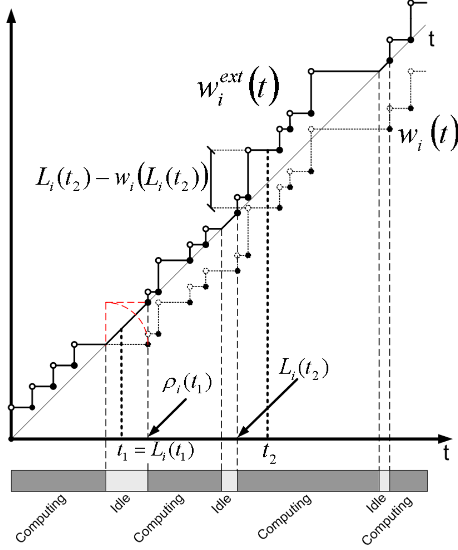

Figure 2. Extended Workload Request

5.1. Last Idle Instant Calculation

The response time analysis presented requires the determination of the last idle instant of the higher priority periodic tasks for a given time, $L_i(t)$. Even though previous work in [1] analyzes the small interval $[{}^{l-1}r_i + D_i, {}^{l}r_i[$ to determine the remaining workload at ${}^{l}r_i$, it also has to add the remaining workload at ${}^{l-1}r_i + D_i$. So, in essense, it has to analyze the whole time sequence since the beggining of time to determine the remaining workload at ${}^{l}r_i$. In fact, all previous work regarding asynchronous systems required determining the schedule since the beggining of time [1, 5]. Our method also has to analyze the interval $[0,t[$ but, instead of searching every individual higher priority task release as [1], our approach follows a method closely related with the fast iterations of the RTA algorithm.

Our aim is to extend the workload function with the knowledge of the idle periods: $w_i^{ext}(t)$, Figure 2. When the system is in an idle state (at the current priority level) the extended workload is equal to t (instant t_1 in Figure 2), so $w_i^{ext}(t) - t$ gives the remaining workload at each instant.

By tracking the extended workload function, the idle periods can be determined and the last idle instant found. In summary, when the system is non idle (computing), the method advances in time similarly to the RTA method; when it is in idle, a new function, $\rho_i(t)$, determines the next computing instant.

The transition between an idle instant to a computing instant requires the determination of the workload requested at a particular instant. To do this, the auxiliary function $\overline{w}_i(t)$ is introduced

$$\overline{w}_i(t) = \sum_{j=1}^{i} \left\lfloor 1 + \frac{t - r_j}{T_j} \right\rfloor_0 c_j \tag{11}$$

where the non-linear operator $\lfloor x \rfloor$ denotes the floor function and $\lfloor x \rfloor_0 = max\{0, \lfloor x \rfloor\}$. The function $\overline{w}_i(t)$ gives the workload request in the interval $[0,t]$. Notice that this interval is closed at both ends. Hence, the workload requested at t can be given by $\overline{w}_i(t) - w_i(t)$.

The method advances through the computing periods through a fixed-point method. Let $\Upsilon(t)$ be the total idle time until a given instant. At each iteration, the fixed-point method gives the instant where the workload demanded has been fulfilled. Since from the last iteration to the present one new workload may have been demanded, the method continues to update the new workload until it stops increasing

$$w_i^{(n+1)} = \sum_{j=1}^{i} \left\lceil \frac{w_i^{(n)} - r_j}{T_j} \right\rceil_0 c_j + \Upsilon(t) \tag{12}$$

When $w_i^{(n+1)} = w_i^{(n)}$ the method stops and $w_i^{(n)}$ is equal to the instant where the computing period has ended. The diagram in Figure 4 shows how this iterative method finds the next point computing instant.

After finding the instant where the workload as been fulfilled, which also corresponds to an idle instant, the function $\rho_i(t)$ gives the next computing instant, which corresponds to the closest task release

$$\rho_i(t) = \min_{j=1\ldots i} \left(\left\lceil \frac{t - r_j}{T_j} \right\rceil_0 T_j + r_j \right) \tag{13}$$

Being t the last computing instant of a computing period, the difference between $\rho_i(t)$ and t gives the next idle period. By sequentially adding these idle periods to the workload request function we get the extended workload function, Figure 2.

$$w_i^{ext}(t) = w_i(t) + \underbrace{L_i(t) - w_i(L_i(t))}_{\text{total idle time}} \tag{14}$$

An iterative method is thus built by sequentially determining the computing and idle periods. The transition from an idle to a computing period is accomplished by adding the workload request at that instant, given by $\overline{w}_i(\rho_i(t)) - w_i(\rho_i(t))$. The pseudo-code that implements this method is illustrated in Figure 3.

An example of how this pseudo-code works is shown on Figure 4 . As illustrated, the algorithm adds to the workload the idle time periods so as to follow the extended workload request function.

```
function L_i(t){
    if( i = 0 ) return t; // there are no higher priority tasks
    last_idle_instant = 0;
    total_idle_time = 0;
    iterator = 0;
    while(true) {
        last_iterator = iterator;
        iterator = w_i(iterator) + total_idle_time;
        if(iterator > t)
            return last_idle_instant;
        if(last_iterator = iterator) { // arrived at an idle instant
            if(iterator ≤ t ≤ ρ_i(iterator))
                return t;
            last_idle_instant = ρ_i(iterator);
            total_idle_time += ρ_i(iterator) − iterator;
            iterator = ρ_i(iterator) + w̄_i(ρ_i(iterator)) − w_i(ρ_i(iterator))
}}}
```

Figure 3. Pseudo-code to determine $L_i(t)$

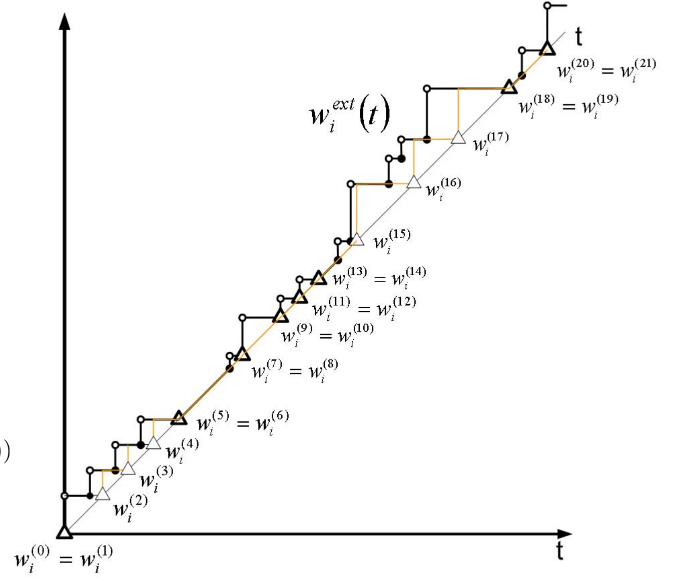

Figure 4. Iterations of the function $L_i(t)$

5.2. Response Instant Calculation

Like the classic RTA iterative method, the response instants can also be found iteratively. Hence, the response instants of Γ_i, ${}^{l}R_i$, are determined using the fixed-point method

$$\begin{cases} {}^{l}R_i^{(0)} & = & c_i + {}^{l}r_i \\ {}^{l}R_i^{(n+1)} & = & c_i + w_{i-1}({}^{l}R_i^{(n)}) + L_{i-1}({}^{l}r_i) - w_{i-1}(L_{i-1}({}^{l}r_i)) \end{cases}$$

When ${}^{l}R_i^{(n+1)} = {}^{l}R_i^{(n)}$ the response instant has been found.

5.3. Application Example

An example of how this method can be applied is presented in Table 1 and the first time sequences are shown in Figure 5. The Example 1 has a 98,67% CPU usage and $\sum \frac{c_i}{D_i} = 282,54\%$. Notice also that the larger hyperperiod, Λ_{10}, is quite large, above 60 million time units. This was made to show that even for very large hyperperiods, the amount of time required by the analysis is still managable. The schedulability method presented thus far has very demanding processing times[2], so the analysis followed an improved method, discussed in Section 7.1. The Table 1 compares the number of iterations of the presented method with the one described in [1][3]. As depicted, the number of iterations required is smaller by up to 95% and the performance gain increases as the hyperperiod becames larger.

From Table 1 we can see that the response time calculated assuming a critical instant, $\overline{R}_i$, renders four tasks unschedulable: $\Gamma_{2,6,7,8}$. Note in particular that Γ_2 is deemed

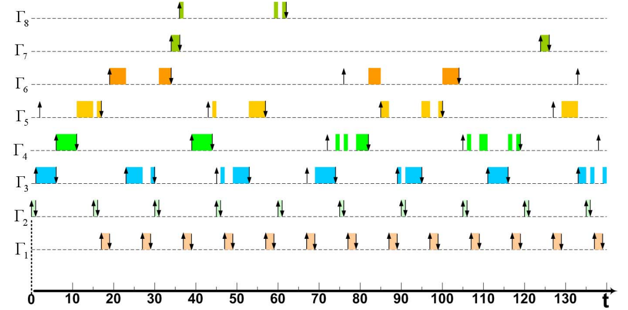

Figure 5. Scheduling Example 1 for the tasks $\Gamma_1, \ldots, \Gamma_8$. The symbol $\uparrow$ denotes the task release whereas $\downarrow$ denotes the response instant

schedulable from Figure 5. In fact, this task never suffers interference from the higher priority task Γ_1. For the tasks $\Gamma_{7,8}$, we show that their response time is always inferior to the deadline by analysing every job in the hyperperiod: Figure 6. Following a similar approach, task Γ_6 is also deemed schedulable. Note that to determine if the task is schedulable the determination of $\overline{R}_i$ (or other, even faster schedulability test) can be sufficient in most cases. There are, however, scenarios where the classic schedulability analysis gives too pessimisitic results. This analysis can be taken in these situations and when there is enough time.

This method provides the worst response time for all jobs and therefore allows a better view as to how much response jitter is introduced. For example, as seen on Figure 5, Γ_2 never suffers the interference of Γ_1. Hence its jitter is only bounded by its minimum and maximum execution time, $[c_2^-, c_2]$, making it an almost "jitter-free" task.

[2]A rough estimation gives more than 14000 hours to analyze Γ_{10}.

[3]This result may vary somewhat with the implementation since the algorithm in [1] requires a sorting procedure.

System Parameters					Schedulability Results					
Γ_i	c_i	T_i	r_i	D_i	$\overline{R}_i$	max $\left({}^lR_i - {}^lr_i\right)$	$LCM(T_1,\ldots,T_i)$	Analysis Time (s)	# Iterations	Improvement over [1]
Γ_1	2	10	17	2	2	2	10	0.015	2	0%
Γ_2	1	15	0	2	**3**	1	30	0.047	17	12%
Γ_3	5	22	1	10	8	8	330	0.062	173	20%
Γ_4	5	33	6	20	15	15	330	0.078	176	43%
Γ_5	5	42	1	42	28	21	2 310	0.172	1 145	69%
Γ_6	7	57	19	47	**58**	44	43 890	2.094	20 654	83%
Γ_7	2	90	34	90	**98**	89	131 670	5.406	48 387	83%
Γ_8	3	120	36	120	**148**	101	526 680	19.78	181 318	85%
Γ_9	17	345	0	340	329	329	12 113 640	305.9	3 754 897	88%
Γ_{10}	2	700	0	700	660	622	60 568 200	1196	9 794 992	95%

Table 1. Parameters of Example 1. The schedulability analysis followed the improved method described in Section 7.1, Figure 13

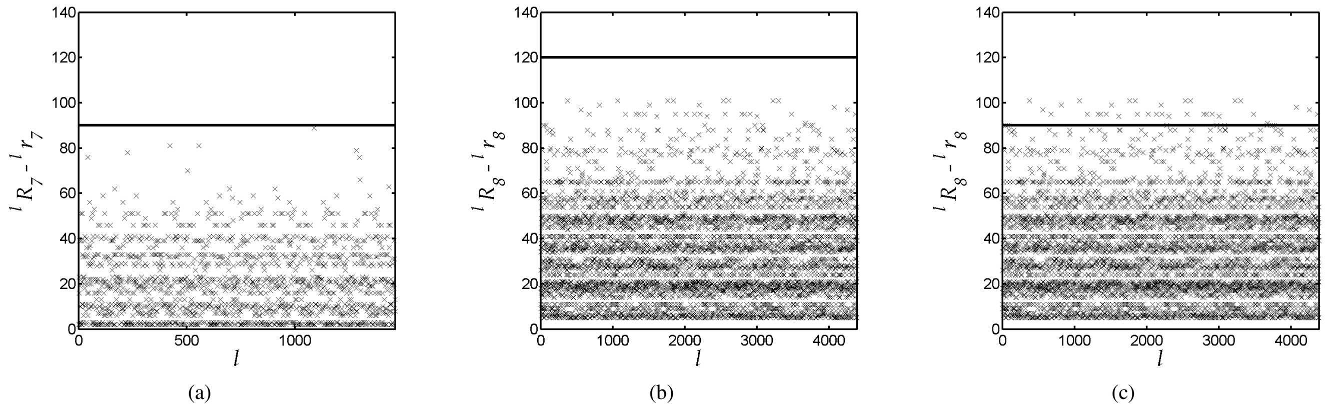

Figure 6. Response time of each job in the hyperperiod for the tasks: (a) Γ_7; (b) Γ_8; (c) Γ_8 with $D_8 = 90$

In a more general case, the determination of the worst jitter requires two analysis: one with the minimum execution times, $c_1^-,\ldots,c_i^-$; the other with the worst execution times, $c_1,\ldots,c_i$. The interval between the best response time of the first case with the worst response time of the latter gives the largest jitter. If, for example, $c_i^- = c_i$, then Figure 6 can tell us the jitter of tasks Γ_7 and Γ_8.

This analysis also covers systems that support occasional missed deadlines. For example, by lowering D_8 to 90 time units, there are some jobs that do not fulfill the deadline, as depicted in Figure 6. Note however, that since the responses are all lower than T_8, if one job misses its deadline, the next job will not suffer any temporal interference. If a job has a response time higher than its period, than this analysis does not suffice and a model extension (to incorporate arbitrary deadlines, for example) is needed.

Since the knowledge of $L_{i-1}({}^lr_i)$ is required to determine the response time for each job inside a hyperperiod, the overall complexity to determine if a given task Γ_i is schedulable is $O(E\,LCM(T_1,...,T_i)^2)$, where E is the average number of steps required to calculate the next idle instant since the previous one. The algorithm is dependent on the quadratic power of the LCM because, when calculating $L_{i-1}({}^lr_i)$, the method starts from the beginning for every job. For a $LCM(T_1,...,T_i)/T_i$ number of jobs, the number of total iterations required is $E\;1 + E\;2 + ... + E\;LCM(T_1,...,T_i)/T_i = O(E\;LCM(T_1,...,T_i)^2)$. In Section 8.1 we will describe how to make this $O(E\,LCM(T_1,...,T_i))$.

6. Integration of Sporadic Tasks

One major advantage of using event-driven systems with a combination of periodic and sporadic tasks comes from the high responsiveness and low resource consumption. Suppose a high priority sporadic task with a very low deadline (H_i) but with a very large MIT: the sporadic task will execute seldomly having a low impact on the remaining tasks schedulability. A pure time-triggered system, on the other hand, would need a high frequency polling task to provide such responsiveness.

The synchronous approach assumes that all tasks,

whether periodic or sporadic, are released at the same instant: critical instant. However asynchronous periodic tasks are released at fixed instants which may not coincide with the critical instant. Therefore it is necessary to know the worst possible instant where a sporadic task can be released: *asynchronous critical instant*.

It is also necessary to account for the interference produced by the sporadic tasks. Hence, assuming that all sporadic tasks are released at $t = 0$, the workload demanded by the tasks $\tau_1, \ldots, \tau_m$, given by $\omega_m(t)$, is equal to

$$\omega_m(t) = \sum_{k=1}^{m} \left\lceil \frac{t}{MIT_k} \right\rceil_0 e_k \tag{15}$$

If the sporadic tasks are released at t_0 then their workload function is $\omega_m(t - t_0)$. As will be shown latter on, the worst instant comes when all sporadic tasks are released at the same instant.

6.1. Asynchronous Critical Instant

This section presents a method to determine the asynchronous critical instant, κ_i, imposed by the higher priority asynchronous periodic tasks. This instant corresponds to the release instant where low priority sporadic tasks are released to produce the worst response time.

Observing Figure 7, which illustrates the function $w_i^{ext}(t) - t$, we conclude that the asynchronous critical instant can only be in the beginning of the computing periods/end of the idle periods. This can easily be proved by assuming the contrary: if the asynchronous critical instant is in the idle period before, then the response instant will also be sooner by at least the same amount, Figure 7; if it is in the computing period after, then the response instant is equal but, since it started later, the response time is smaller, Figure 7. Hence, we need only concern with the start of the computing periods because these will give the candidates for the asynchronous critical instant. The worst release instant for the lower priority sporadic tasks is therefore one of these candidates.

To determine the candidates for the asynchronous critical instant we need a method to determine the next idle instant, $\varphi_i(t)$. This new method is very similar to the iterations of the $L_i(t)$ algorithm. As specified in Figure 8, the function $\varphi_i(t)$ receives a work instant. This is the first work instant of the computing period. The first iteration adds the work demanded at that instant. The following iterations are very similar to the other methods, adding at each iteration the new workload demanded. An example of the $\varphi_i(t)$ iterations is shown in Figure 9.

Figure 8 also describes a method, $\kappa_i(t_{min}, t_{max})$, that captures all candidates inside the interval $]t_{min}, t_{max}]$. This method is either calculating the next work instant, using

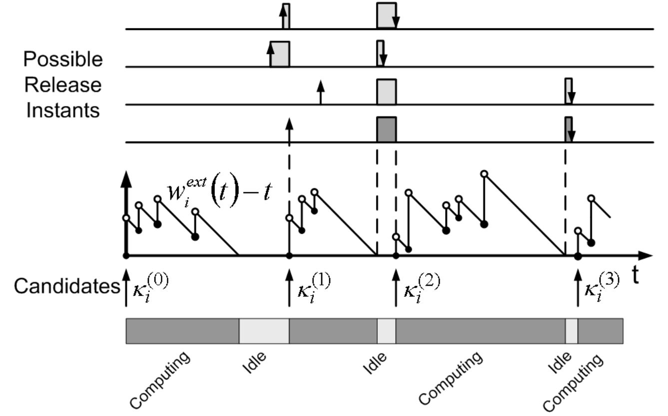

Figure 7. Asynchronous Critical Instants Candidates

$\rho_i(t)$, or the next idle instant, using $\varphi_i(t)$. When the work instant found is greater than t_{max} the method stops.

The complexity of determining the asynchronous critical instant candidates in Λ_i is similar to the determination of the last idle instant: $O(E\, LCM(T_1, \ldots, T_i))$.

6.2. Response Time Analysis of Asynchronous Periodic Tasks

The worst interference that sporadic tasks can produce to the lower priority periodic tasks can be determined with knowledge of the asynchronous critical instant candidates.

Suppose a periodic job released at ${}^{l}r_i$ and that τ_K is the lowest priority sporadic task with higher priority than Γ_i. In Figure 10 is illustrated the higher priority periodic tasks remaining workload at each instant, $w_i^{ext}(t) - t$. There are four candidate instants between ${}^{l-1}r_i$ and ${}^{l}r_i$: $\kappa_i^{(1)}, \ldots, \kappa_i^{(4)}$.

From Figure 10 it is clear that the worst interference that high priority sporadic tasks induce into the low priority periodic tasks happens when they are released at the critical instant candidates. If the sporadic tasks are released earlier, then they will be executed (since the system is idle). If they are released after the candidate, then the interval where they can cause interference is smaller.

The worst release instant in Figure 10 is $\kappa_i^{(2)}$. This release instant removes the idle period between $\kappa_i^{(2)}$ and ${}^{l}r_i$, making one single computation period. In fact, if there are idle periods between the release of the sporadic tasks and ${}^{l}r_i$ then that cannot be the worst release instant since all the workload has been fulfilled at one instant and hence will not be added to the next computing period (take example of the release at $\kappa_i^{(1)}$ in Figure 10).

Since from the worst release instant to ${}^{l}r_i$ there cannot exist idle periods, if the worst release is before ${}^{l-1}r_i$ then the $l-1$ job could not be executed and would miss its deadline. Hence, assuming that all previous jobs fulfill their deadline,

```
function κi(tmin, tmax){
    next_work_instant = Li(tmin);
    work_instants = {};
    while(true) {
        next_idle_instant = φi(next_work_instant);
        next_work_instant = ρi(next_idle_instant);
        if( next_work_instant > tmax )
            return work_instants;
        work_instants = work_instants ∪ {next_work_instant};
} }
function φi(work_instant){ //returns the next idle instant
    total_idle_time = work_instant − wi(work_instant);
    iterator = work_instant + w̄i(work_instant) − wi(work_instant);
    while(true) {
        last_iterator = iterator;
        iterator = wi(iterator) + total_idle_time;
        if( iterator = last_iterator )
            return iterator;
} }
```

Figure 8. Pseudo-code to determine all the candidates of the asynchronous critical instant in the interval $]t_{min}, t_{max}]$

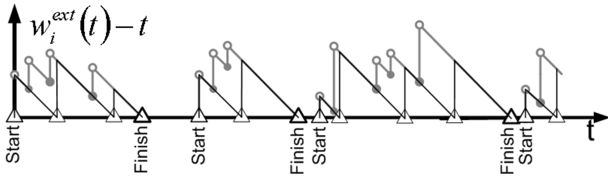

Figure 9. Examples of the $\varphi_i(t)$ iterations

the candidates to the worst release instant are in the interval $]^{l-1}r_i, {}^l r_i]$. If there are no critical instant candidates, then the task is not schedulable since during that interval the system never completed its workload and hence the previous job did not fullfill its deadline.

The interval $]^{l-1}r_i, {}^l r_i]$ must not be inside the initial transient phase. So the hyperperiod must start T_i after the initial phase has ended. Hence the need to make $S_i = \max(r_1, ..., r_i) + T_i$ in Equation (9). This change in the hyperperiod does not affect significantly the analysis time since the LCM is still by far the main factor.

Since the idle periods from the worst release instant, κ_i, to ${}^l r_i$ are now occupied processing the higher priority tasks, these have to be subtracted when calculating the response instant. The total idle time from κ_i to ${}^l r_i$ is given by

$$\underbrace{[L_{i-1}({}^l r_i) - w_{i-1}(L_{i-1}({}^l r_i))]}_{\text{total idle time until } {}^l r_i} - \underbrace{[L_{i-1}(\kappa_i) - w_{i-1}(L_{i-1}(\kappa_i))]}_{\text{total idle time until } \kappa_i}$$

Since at κ_i the higher priority periodic tasks are idle, then $L_{i-1}(\kappa_i)) = \kappa_i$. The response instant of an asynchronous periodic task is found by adding to equation (8) the workload

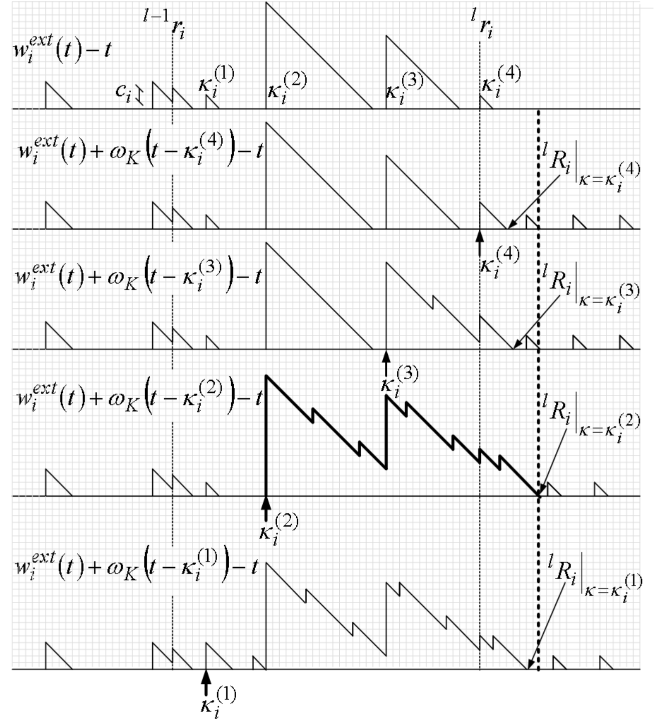

Figure 10. Analysis of the candidates to the worst sporadic release

from the higher priority sporadic tasks and subtracting the total idle time

$$^l R_i = \kappa_i + c_i + w_{i-1}(^l R_i) - w_{i-1}(\kappa_i) + \omega_K(^l R_i - \kappa_i) \quad (16)$$

where the smallest solution (larger than ${}^l r_i$) corresponds to the response instant. This equation can be solved using the fixed-point method with ${}^l R_i^{(0)} = {}^l r_i + c_i$. This equation has to be solved for every $\kappa_i \in]^{l-1}r_i, {}^l r_i]$ and the worst response instant found is used in expression (10) to determine if Γ_i is schedulable. Note that if a particular κ_i is not the worst instant and it does not completely remove the idle periods, then the corresponding response instant calculated will be smaller than it should be (because we are subtracting all the idle periods), but since this is not the worst case it will not be considered in the final schedulability test.

If there are no higher priority sporadic tasks then the only κ_i candidate that does not have any idle periods between κ_i and ${}^l r_i$ is $L_{i-1}({}^l r_i)$ ($\kappa_i^{(4)}$ in Figure 10), so equation (16) becomes equivalent to equation (8).

6.3. Response Time Analysis of Sporadic Tasks

To determine the worst possible response time of a sporadic task we have to consider all asynchronous critical in-

stant candidates. Consider Γ_i to be the lowest priority periodic task with higher priority than τ_m. The worst response instant of a sporadic job is found when all higher priority sporadic tasks are released at the same asynchronous critical candidate. Hence, it is the smallest solution (larger than κ_i) of

$$R_m = \kappa_i + e_m + w_i(R_m) - w_i(\kappa_i) + \omega_{m-1}(R_m - \kappa_i) \quad (17)$$

Likewise, this equation can be solved using the fixed-point method, with $R_m^{(0)} = e_m + \kappa_i$. The schedulability analysis can be summarized in the condition

$$\begin{cases} \forall_{\kappa_i \in \Lambda_i} \quad R_m - \kappa_i \le H_m \quad , \quad \tau_m \text{ is } \textbf{schedulable} \\ \exists_{\kappa_i \in \Lambda_i} \quad R_m - \kappa_i > H_m \quad , \quad \tau_m \text{ is } \textbf{unschedulable} \end{cases} \quad (18)$$

As an example take the interference given by $\Gamma_1, ..., \Gamma_3$ (Table 1). The candidates for the asynchronous critical instant are (first 10): 37; 45; 57; 60; 67; 75; 77; 87; 89; 97. In total, there are 55 candidates in Λ_3. The response times of a sporadic task with $e_1 = 1$ and released at the candidates instants are respectively: 3; 9; 3; 2; 8; 2; 3; 9; 7; 3. For a $e_1 = 10$ the response times are: 20; 21; 23; 21; 20; 21; 20; 23; 21; 20. Note that if the task is released at $t = 57$, for a $e_1 = 1$ the response time is very low compared to other instants, but when $e_2 = 10$ it has the highest response time. Therefore all critical instants candidates must be considered, even if for lower WCET they give very good response times. This is because for a task with a very low WCET, then the worst candidate corresponds to the single longest computation period. But for a task with a large WCET it can be the longest chain of computing periods with little idle periods in between.

7. Improving the Response Time Calculation

To speedup the schedulability test of a particular task it is possible to determine the response time assuming a critical instant, $\overline{R}_i$. If the task is schedulable under this condition then it is always schedulable, whatever the release time [1].

7.1. Exploiting Last Idle Instant Knowledge

As described earlier, the complexity of determining the last idle instant for all jobs of a given task Γ_i is $O(E\, LCM(T_1, ..., T_i)^2)$. The quadratic dependency comes from the reinitialization of the iterative method to determine $L_i(t)$ for each job. But by keeping track of the last known idle instant, the algorithm can start from this point instead of doing the same analysis since the beginning. This reduces the overall complexity to $O(E\, LCM(T_1, ..., T_i))$. Since the LCM is obviously the limiting factor, this improvement is very significant.

```
function L̃_i(last_known_idle, t){
    if( i = 0 ) return t; // there are no higher priority tasks
    last_idle_instant = last_known_idle;
    total_idle_time = last_known_idle − w_i(last_known_idle);
    iterator = last_known_idle;
    while(true) {
        last_iterator = iterator;
        iterator = w_i(iterator) + total_idle_time;
        if(iterator > t)
            return last_idle_instant;
        if(last_iterator = iterator) { // arrived at an idle instant
            if(iterator ≤ t ≤ ρ_i(iterator))
                return t;
            last_idle_instant = ρ_i(iterator);
            total_idle_time += ρ_i(iterator) − iterator;
            iterator = ρ_i(iterator) + w̄_i(ρ_i(iterator)) −
                    w_i(ρ_i(iterator));
}}}
```

Figure 11. Improved pseudo-code to determine the $L_i(t)$ starting from the last known idle instant

The Figure 11 illustrates the improved pseudo-code that determines the last idle instant starting from the last known idle instant. Note the changes made only in the initialization phase. Figure 12 describes the improved pseudo-code that determines κ_i using the new function $\tilde{L}_i(last, t)$.

Finally, Figure 13 illustrates the improved pseudo-code to determine the response instants for the jobs $l_{initial}, \ldots, l_{final}$. Note that the last idle instant is updated after the maximum known critical instant candidate since, as previously stated, the critical instant candidates are themselves idle instants. If the response time is larger than the task period, then this analysis cannot determine the response instants of the following jobs (since the next job will have some additional interference) and the method halts.

As an example of the improvement made, the determination of the response times for the Γ_7 jobs took approximately one hour with the standard method[4]. With the improved method it took 5 seconds - Table 1. A similar approach was also taken in [1].

7.2. Handling Large Hyperperiods

The schedulability analysis is rigidly dependent on the hyperperiod defined by the periodic tasks. For a high number of tasks, this analysis can be deemed unfeasible, specially when the periods are co-prime.

This paper introduces an innovative idea to handle large hyperperiods by allowing some degree of pessimism. Instead of considering that all tasks are released at a critical

[4]Using MatLab under Windows XP on a PIII processor at 1300 MHz.

function $\tilde{\kappa}_i(t_{min}, t_{max}, last_idle_instant)\{$
    $next_work_instant = \tilde{L}_i(last_idle_instant, t_{min});$
    $work_instants = \{\};$
    while(true) {
        $next_idle_instant = \varphi_i(next_work_instant);$
        $next_work_instant = \rho_i(next_idle_instant);$
        if($next_work_instant > t_{max}$)
            return $work_instants$;
        $work_instants = work_instants \cup \{next_work_instant\};$
} }

Figure 12. Improved pseudo-code to determine κ_i in the interval $]t_{min}, t_{max}]$ using the function $\tilde{L}_i(last, t)$

function $\tilde{R}_i(l_{initial}, l_{final})$ {
    $idle = 0$; // last known idle instant
    for$(l = l_{initial}\ ;\ l \leq l_{final}\ ;\ l{+}{+})\{$
        ${}^{l}R_i = {}^{l}r_i + c_i$; // smallest possible value of ${}^{l}R_i$
        $\kappa_i = \tilde{\kappa}_i({}^{l-1}r_i, {}^{l}r_i, idle);$
        $idle = max(\kappa_i)$; // update last known idle instant
        foreach$(\kappa_i^{(x)} \in \kappa_i)\{$
            $R = {}^{l}r_i + c_i$; // initial iteration
            do {
                $last_R = R;$
                $R = \kappa_i^{(x)} + c_i + w_{i-1}(R) - w_{i-1}(\kappa_i^{(x)}) + \omega_K(R - \kappa_i^{(x)});$
                if$(R\ >\ {}^{l+1}r_i)$
                    return “cannot determine response instant”;
            } **while**$(last_R \neq R)$;
            if$(R\ >\ {}^{l}R_i)$ // update ${}^{l}R_i$ with largest response instant
                ${}^{l}R_i = R;$
    } }
    return ${}^{l_{min}}R_i, \ldots, {}^{l_{max}}R_i$;
}

Figure 13. Improved pseudo-code to determine ${}^{l_{initial}}R_i, \ldots, {}^{l_{final}}R_i$ using the function $\tilde{\kappa}_i(t_{min}, t_{max}, last)$

instant, it is possible to allow that only a small portion is released at pessimistic instants. Hence it is possible to compromise between the pessimism induced and the number of iterations required.

By knowing the asynchronous critical instants imposed by the periodic tasks, the analysis can encompass a reduced set of high priority periodic tasks and determine the worst instants in which lower priority tasks (periodic and sporadic) can be released.

For example, suppose a task set of 40 periodic and 40 sporadic tasks. If the hyperperiod defined by the 40 periodic tasks is too large, the system designer determines the asynchronous critical instants of only the first 20 periodic tasks. A sufficient test can thus be found by assuming that all lower priority tasks, periodic and sporadic, are released at these instants. Note that even though the number of candidates can be very large, it is fixed, making the analysis of the lower priority tasks take relatively the same amount of time.

As a concrete example take the schedulability analysis of Γ_8 in Example 1 (Table 1). The determination of all candidates of the asynchronous critical instant of the tasks $\Gamma_1, \ldots, \Gamma_7$ in Λ_7 took 4.28s and the determination of the response times of Γ_8, assuming it was released at these candidates, took 9.2s (not shown in Table 1). The worst response time is found when the task is released at $t = 925; 49435; 97945$ and equals to 110. Since the deadline is 120, this analysis is sufficient to say that Γ_8 is schedulable. If the worst response time is larger than the deadline, then nothing can be concluded.

8. Conclusions

This paper presents an algorithm to analyze the schedulability of asynchronous periodic and sporadic tasks scheduled by a fixed-priority preemptive algorithm. The results presented in this paper can be used to

- Diminish the pessimism induced by the critical instant
- Allow offset relationships
- Determine the worst response time of each job
    - analyze response jitter
    - account for missed deadlines
- Calculate the *asynchronous critical instant*
    - integrate sporadic tasks
    - increase analysis speed

By assuming that the release times are not equal to zero and hence not considering the critical instant scenario, more than just one response time calculation is required. The determination of the response times of all jobs under a hyperperiod is necessary. This makes the overall computation time strongly dependent on the LCM of the tasks periods. Therefore, the proposed methods do not scale well against the number of periodic tasks, especially task sets using co-prime periods. Hence, it is only expected to be used as an offline test. However, there are a number of applications where the schedulability analysis presented is manageable and useful, such as the automotive, satellite and other aerospace industries, where the typical task frequency is around 10-100Hz.

The inclusion of the sporadic tasks also enables a new faster schedulability method by manipulating the degree of pessimism introduced. In essence, the lower priority tasks

are considered to be sporadic tasks released at a set of *asynchronous critical instants* (worst possible instants). This method strongly reduces the hyperperiod under analysis.

The incorporation of major model extensions found for synchronous systems, such as blocking factors, context switch overhead, non-preemptive schedulers, dynamic priority schedulers, arbitrary deadlines, etc, is very appealing as future work. The introduction of release jitter is also of great interest. Furthermore, a study is being performed to allow the determination of $L_i(t)$ without having to analyze the whole interval $[0,t[$. This enables an efficient distribution of the overall method through several independent machines, each one analysing a small portion of the hyperperiod with a very small overhead.

Acknowledgments

This work was motivated by our research activities under the scope of Projects DARIO (Distributed Agency for Reliable Input/Output), AIR (ARINC 653 Interface in RTEMS) [16] and RTEMS CENTRE [19]. The authors are in debt to researchers and technical officers at ESTEC (ESA - European Space Agency, Noordwijk) for some informal discussions on task timing characteristics of typical spacecraft on-board software applications.

References

[1] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, Department of Computer Science, University of York, 1991.

[2] N. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[3] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Fifth Euromicro Workshop on Real-time Systems*, pages 36–41, Oulu, Finland, 1993. IEEE Computer Society Press.

[4] I. Bate and A. Burns. Schedulability analysis of fixed priority real-time systems with offsets. In *Proc. of 9th Euromicro Workshop on Real-Time Systems*, pages 153–160, Toledo, Spain, June 1997. IEEE Computer Society.

[5] G. Bernat. Response time analysis of asynchronous real-time systems. *Real-Time Syst.*, 25(2-3):131–156, 2003.

[6] R. Devillers and J. Goossens. General response time computation for the deadline driven scheduling of periodic tasks. *Fundamenta Informaticae*, 40(2–3):199–219, November-December 1999.

[7] J. Goossens. *Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constrains*. PhD thesis, Faculté des Sciences, December 1999.

[8] J. Goossens and R. Devillers. Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. In I. C. Society, editor, *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 54–61, December 1999.

[9] M. Grenier, J. Goossens, and N. Navet. Near-optimal fixed priority preemptive scheduling of offset free systems. In *Proceedings of the 14th International Conference on Network and Systems (RTNS'2006)*, Poitiers, France, May 2006.

[10] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In I. C. S. Press, editor, *Proceeedings of the 12th IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[11] M. Joseph and P. Pandaya. Finding response times in a real-time system. *The Computer Journal*, 29:390–395, 1986.

[12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171, Santa Monica, USA, 1989.

[13] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[14] J. Mäki-Turja and M. Nolin. Fast and tight response-times for tasks with offsets. In *17th EUROMICRO Conference on Real-Time Systems*, page 10, Palma de Mallorca Spain, July 2005. IEEE.

[15] R. Pellizzoni and G. Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems*, 30(1-2):105–128, 2005.

[16] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor. ARINC 653 interface in RTEMS. In *Proceedings of the DASIA 2007 "DAta Systems In Aerospace" Conference*, Naples, Italy, June 2007. EUROSPACE.

[17] K. Shin and X. Chui. Computing time delay and its effects on real-time control systems. *IEEE Transactions on Control Systems Technology*, 3(2):218–224, June 1995.

[18] H. Silva, A. Constantino, D. Freitas, M. Coutinho, S. Faustino, and M. Zulianello. RTEMS CENTRE - support and maintenance CENTRE to RTEMS operating system. In *Proceedings of the DASIA 2008 "DAta Systems In Aerospace" Conference*, Palma de Majorca, Spain, May 2008. EUROSPACE.

[19] H. Silva, A. Constantino, M. Mota, D. Freitas, and M. Zulianello. RTEMS CENTRE - support and maintenance to RTEMS operating system. In *Proceedings of the DASIA 2007 "DAta Systems In Aerospace" Conference*, Naples, Italy, June 2007. EUROSPACE.

[20] K. Tindell. Adding time-offsets to schedulability analysis. Technical report, University of York, 1994.