A Fast Slack Stealing method for embedded Real-Time Systems

José M. Urriza, Ricardo Cayssials and Javier D. Orozco

Departamento de Ingeniería Eléctrica y Computadoras

Universidad Nacional del Sur / CONICET

8000 Bahía Blanca, Argentina

`{jurriza, iecayss, ieorozco}@criba.edu.ar`

Abstract:

In this paper we present a low-cost Slack-Stealing method, named Fast Slack, to be employed in a Fixed Priority scheduling mechanism. The slack obtained is intended to improve the execution of non-critical tasks without jeopardizing the schedulability of the critical ones. The method is compared with the most important one in the real-time theory. We show that the complexity of this method is suitable to be implemented online in embedded real-time systems. Besides, we can determine online the amount of time that the calculus of the slack will take.

Key Words: Slack Stealing, Embedded Systems, Real-Time Scheduling

1 Introduction

Real-time systems are applied to the most diverse areas nowadays. A real-time system consists of a set of tasks that must be executed by the processor before a certain deadline. If all deadlines are wanted to be met, then the system has to be designed in order to satisfy each one of the deadlines under the worst case of load. Since the worst case of load rarely occurs during runtime, designing under this premise could lead to a pessimistic implementation and consequently the processor is under utilised most of the time.

The problem of jointly scheduling both hard and soft deadline tasks is an important issue in many real-time systems. This is due to the tension between the scheduling requirements of tasks in the two categories. Typically, soft tasks benefit from being delivered as early as possible, whilst hard tasks need to be guaranteed to meet their deadlines [1].

Several approaches have been developed for scheduling mixed task sets under a fixed priority pre-emptive discipline. The simplest and perhaps the least effective of these is to execute soft deadline tasks at lower priority level than any of those with hard deadlines. This effectively relegates the soft tasks to background processing. Alternatively, soft tasks may be run at higher priority under the control of a pseudo hard real-time server task, such as a simple polling server.

A polling server is a periodic task that is executed at the highest priority and its execution time is predefined off-line. Improvements of this mechanism were proposed in *Priority Exchange* and *Deferrable Server* [2], *Extended Priority Exchange* [3], *Sporadic Server* [4] and *Total Bandwidth Server*.[5]

The polling server will usually significantly improve the response times of soft tasks over background processing. However, if the ready soft tasks exceed the capacity of the server, then some of them will have to wait until its next release, leading to potentially long response times. Conversely, no soft tasks may be ready when the server is released, wasting its high priority capacity.

The Slack Stealer algorithm [6] services aperiodic requests by making any spare processing time available as soon as possible. In doing so, it effectively steals slack from the hard deadline periodic tasks. A means of determining the maximum amount of slack which may be stolen, without jeopardizing the hard timing constraints, is thus key to the operation of the algorithm. A setback of these mechanisms is its complexity. In [1, 7] and [8] different approaches have been proposed to reduce the complexity of the slack stealing. In [8], Tia proved that there is not an optimal on-line algorithm that minimises the response time of non-critical tasks nor its average.

In [1, 6-8], it is described how the slack available can be found. At run-time, a set of counters are used to keep track of the slack which may be stolen at each priority level. These counters are decremented depending on which tasks, if any, are executing and updated by a certain criterion, at the complexion of each task. Whenever, the counters indicate that there is slack available at all priority levels, then soft tasks may be executed at the highest priority level.

In this paper, we present an online algorithm to determine the amount of slack that a real-time system leaves. We show that the complexity of this method is very low and consequently it makes it suitable for embedded real-time applications. We compare its performance with the methods proposed in [1].

The paper is organized as follows: Section 2 describes the Real-Time Process model considered in this paper. Section 3 shows an example which uses the slack available. In Section 4, the Fast Slack method is mathematically proved. Section 5 describes the implementation of the Fast Slack method to be used online. Comparisons and experimental results are shown in Section 6. Conclusions are drawn in Section 7.

2 Real-Time Process Model

Real-time systems theory allows to analyse the temporal properties of a set of concurrent tasks. The general process model of a real-time system, from the implementation point of view, consists of a set $\mathbf{\Pi}$ 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, and worst-case execution time, C_i.

$$\mathbf{\Pi} = \{T_i, D_i, C_i\} \quad 1 \le i \le n$$

Each time that a task requires the processor to be executed, it is said that the task is either *invoked* or an *invocation takes place*. When a task invocation is completed, it will not require to be executed until the next invocation.

Only one task can run on a processor at the same time. The selection of the task is done by the *scheduler* that applies a priority discipline among the tasks of the systems that are requiring to be executed. The scheduler is one of the main tasks of a real-time operating system (RTOS). The fixed priority (FP) discipline is one of the most important ones in real-time and most of RTOS implement it. In a FP discipline each task is assigned with a priority in the design time and it remains fixed during runtime.

Exact, necessary and sufficient scheduling analysis conditions exist to guarantee that the temporal requirements will be satisfied in a real-time system. The most advanced technique computes the worst case response time of any task. This is done by building the critical instance, named the *worst case of load*, as the time instant when a task will suffer its maximum interference from higher priority tasks. This actually happens when all tasks are released simultaneously. The worst case response time R_i of a task τ_i can be computed by finding the smallest $R_i > 0$ that is a solution to the following fix point equation [9, 10]

$$R_i^+ = C_i + W_{i-1}(t) = C_i + \sum_{j=1}^{i-1} C_j \left\lceil \frac{R_i}{T_j} \right\rceil = \sum_{j=1}^{i} C_j \left\lceil \frac{R_i}{T_j} \right\rceil \tag{1}$$

where $W_i(t)$ is the workload of the subsystem defined as:

$$W_i(t) = \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil$$

This can be solved by forming a recurrence with $R_i^0 = 0$ and stopping when $R_i^n = R_i^{n+1}$ or when $R_i^n > D_i$. The system is schedulable iff $R_i \le D_i \ \forall\, i,\ 1 \le i \le n$.

The complexity of the slack stealing mechanisms based on these schedulability testing is **O**(max(T_i).n^2) and consequently they cannot be applied on-line.

3 Using the slack: an example

In this section we show how slack can be used to execute non-critical tasks. Let's consider a real-time system with 3 tasks whose real-time parameters are shown on Table 1

Task τ_i	T_i	C_i	D_i
1	3	1	3
2	4	1	4
3	6	1	6

Table 1

Figure 1 shows how tasks are scheduled after a critical instant. Arrows represent the time when task is invoked. The real-time system leaves three idle intervals at times 5, 10 and 11.

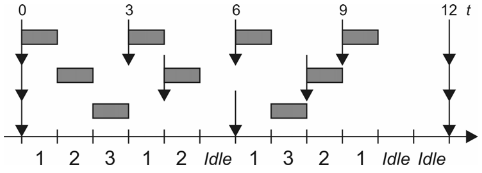

Figure 1. Tasks scheduled by a Fixed Priority discipline after a critical instant.

Postponing the execution of real-time tasks as much as possible, we can produce idle intervals at times 0, 6 and 7. Figure 2 shows the scheduling diagram that leaves such idle intervals.

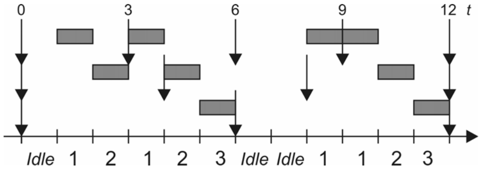

Figure 2. Execution of hard real-time tasks delayed the maximum amount of time.

When hard real-time tasks are delayed the maximum amount of time that task can support, the response time of soft tasks is improved. If no soft task is ready for execution, then a hard task is executed and the slack remains available in the next interval.

4 Slack time analysis

The slack analysis is based on considering the schedulability of each task τ_i at some arbitrary time t. As it is defined in [1], we assume that at time t, the following data is available via the operating system (typically derived from data stored in a task control block):

$l_i(t)$ The time at which task τ_i was last invoked.

$x_i(t)$ The earliest possible next release of task τ_i . Typically $x_i(t) = l_i(t) + T_i$.

$d_i(t)$ The next deadline on an invocation of task τ_i . (Note, if the current invocation of task τ_i is complete, then $d_i(t) = x_i(t) + D_i$, i.e. $d_i\,(t)$ is the deadline following the next release).

$c_i(t)$ The amount of time that the current invocation of task τ_i was executed.

$e_i(t)$ The time at which last completed invocation was finished.

In this section we proposed a low-complexity mechanism to get the maximum amount of slack time which may be stolen at priority level i at time t, denoted $S_i(t)$, whilst guaranteeing that task τ_i meets its deadline. Note, $S_i(t)$ may not actually be available for soft task processing due to the constraints on hard deadline tasks with priorities lower than i. To guarantee that task τ_i will meet its deadline, we need to analyse the worst case scenario from time t onwards. We therefore assume that all tasks j are re-invoked at their earliest possible next release $x_j(t)$ and subsequently with a period of T_j.

Next lemma determines the maximum amount of slack time that task τ_i can support without missing its deadline.

Lemma 1:

Given a set of n real-time tasks scheduled by a Fixed Priority discipline, the maximum available slack for task τ_i at the interval, denoted S_{i,t_c} , is given by:

$$\begin{aligned} &S_{i,t_c} = \max_{[t_c,\, d_{i,t_c}]} k_i \\ &d_{i,t_c} \geq \wedge t \mid t - t_c = k_i + \sum_{j=1}^{i} \left[C_j \left(\left\lceil \frac{t}{T_j} \right\rceil - \left\lfloor \frac{t_c}{T_j} \right\rfloor \right) - c_{j,t_c} \right] \end{aligned} \tag{2}$$

Proof:

In [8], it was proved that the workload function for a real time system with i tasks, in the interval $[t_c , t]$, is

$$W_{i,t_c} = \sum_{j=1}^{i} \left[C_j \left(\left\lceil \frac{t}{T_j} \right\rceil - \left\lfloor \frac{t_c}{T_j} \right\rfloor \right) - c_{j,t_c} \right]$$

A system is schedulable in interval [t_c, t] if there exists enough time to execute each task before its deadline

$$d_{i,t_c} \geq \wedge t \mid t - t_c = W_{i,t_c}$$

Then, the maximum available slack in the interval [t_c, t] is given by the maximum time we can postpone the execution of the workload without missing a deadline

$$\begin{aligned} &S_{i,t_c} = \max_{[t_c\,,\,t]} k_i \\ &d_{i,t_c} \geq \wedge t \mid t - t_c = k_i + W_{i,t_c} \end{aligned} \quad \square$$

The calculus of equation (2) requires an exhaustive iteration on k in the interval [t_c , d_{i,t_c}] which makes it not suitable for online mechanisms.

We propose a low-complexity method to find a solution of equation (2) in such a way that it can be applied online. This method is base on two criteria:

I. Reducing the set of values in which equation (2) is evaluated. We find a reduced set of values which contains the solution of equation (2).

II. Confining the inspection interval close to the solution of equation (2). We restrict the set of values which contains the solution of equation (2) to a close interval.

Applying both criteria we get a reduced set of values which contain the solution of equation (2) and consequently the complexity of the method is reduced.

4.1 Reducing the set of values to evaluate

In this section we get a reduced set of values which contains the t that is solution of equation (2).

Lemma 2:

A schedulable task τ_i, that has suffered the maximum delay, will finish its execution either at its deadline or when a higher priority task invokes.

Proof:

Suppose task τ_i finishes its execution, after the maximum delay that it can support, an interval before both its deadline and an invocation of a higher priority task. If that happens, then there exists an idle interval before the deadline of task τ_i when no task is executed. Therefore, the execution of task τ_i could be delayed that interval. This opposes the supposition and proves the lemma. □

We can conclude from *Lemma 2*, that the t that satisfies equation (2) for the task τ_i is either its deadline or a time when a higher priority task invokes. So, the set of values defined by *Lemma 2* for task τ_i at time t, denoted $\mathbf{V}_i(t)$, which contains the solution of equation (2), is

$\mathbf{V}_i(t)=\{\mathrm{U}\, x_j(t') \text{ for all } t \le t' \le d_i(t)\ \} \ \mathrm{U}\ d_i(t)$.

equation (2) should be evaluated for each value of the set **V** and, if the system is schedulable, it can be expressed as:

$$\begin{aligned} &S_i(t) = \max k_i(t^*) \\ &k_i(t^*) = t^* - t - \sum_{j=1}^{i}\left[C_j\left(\left\lceil \frac{t^*}{T_j}\right\rceil - \left\lfloor \frac{t}{T_j}\right\rfloor\right) - c_i(t)\right], \forall t^* \in \mathbf{V}_i(t) \end{aligned} \tag{3}$$

4.2 Confining the search to a subset of V

In this section, we reduce the number of elements of the set **V** by choosing the elements close to the solution of equation (3).

In [9, 11, 12], it was proved that if $e_{i,0}$ satisfies equation (2) with $k = 0$ at the critical instant, then there exists a solution for equation (2) in any arbitrary interval $[l_{i,t}\ ,\ l_{i,t}+ e_{i,0}]$. This can be explain saying that if response time of task i is $e_{i,0}$ under the worst case of load, then the response time of any other invocation of the task will be up most $e_{i.0}$.

Lemma 3:

If $e_{i,0}$ satisfies equation (2) under a worst case of load for task τ_i, then a time that satisfies equation (2) with the maximum k_i can be found in the interval $[\, d_{i,t_c} - e_{i,0} + C_i\, ,\, d_{i,t_c}\,]$.

Proof:

If $e_{i,0}$ satisfies equation (2) for the worst case of load, the execution of task τ_i can be delayed at least until time $d_{i,t_c} - e_{i,0}$ [13]. Task τ_i should be executed before its deadline and consequently in the interval $[\, d_{i,t_c} - e_{i,0} + C_i\, ,\, d_{i,t_c}\ ]$. Then, there exists a t in such interval that satisfies equation (2) and as a result it supports the maximum k_i. □

4.3 Complexity of the Calculus

The maximum number of elements we obtain from equation (4) in the interval $[x_{i,t_c}, d_{i,t_c}]$ is function of the number of releases of tasks $1 \le j \le i$.

$$\left|\mathbb{k}_i\right| = \sum_{j=1}^{i-1} \left\lceil \frac{d_{i,t_c}}{T_j} \right\rceil - \left\lceil \frac{x_{i,t_c}}{T_j} \right\rceil + 1 \tag{4}$$

and for the critical instant ($t = 0$)

$$\left|\mathbb{k}_n\right| = \sum_{j=1}^{n-1} \left\lceil \frac{D_n}{T_j} \right\rceil - \left\lceil \frac{0}{T_j} \right\rceil + 1 = \sum_{j=1}^{n-1} \left\lceil \frac{D_n}{T_j} \right\rceil + 1 \tag{5}$$

Equation (5) gives the maximum number of iterations of the method.

Lemma 3 reduces the interval of inspection and consequently the number of times that Equation(3) should be evaluated:

$$\left|\mathbb{k}_i\right| = \sum_{j=1}^{i-1} \left(\left\lceil \frac{d_{i,t_c}}{T_j} \right\rceil - \left\lceil \frac{d_{i,t_c} - e_{i,0} + C_i}{T_j} \right\rceil \right) + 1 \tag{6}$$

Equation (6) gives the exact number of times that equation (3) has to be calculated in order to get the slack available. In this way, we can know the computational cost of the method beforehand.

4.4 Example

In this section we show how the slack is obtained during runtime. We use the real-time system described on Table 1. The slack of each task is calculated when the task finishes its first instance. In this way, the slack of task 1, 2 and 3 are calculated at time 1, 2 and 3 respectively. Figure 3 shows the evolution of the system after a worst case of load and Table 2 shows how the slack is calculated when each task finishes.

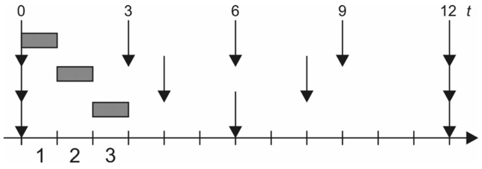

Figure 3. The slack at each level is calculated when tasks finishes.

τ_i	t_c	$e_{i,0}$	$[\, d_{i,t_c} - e_{i,0}+C_i \,, d_{i,t_c} \,]$	t^*	$k_i(t^*) = t^* - t_c - \sum_{j=1}^{i}\left[C_j \left(\left\lceil \frac{t^*}{T_j} \right\rceil - \left\lfloor \frac{t_c}{T_j} \right\rfloor \right) - c_i(t_c) \right]$ (3)
1	1	1	[6-1+1,6]= [6,6]	6	$6-1-\left[1.\left(\left\lceil \frac{6}{3} \right\rceil - \left\lfloor \frac{1}{3} \right\rfloor\right) - 1\right] = 4$
2	2	2	[8-2+1,8]= [7,8]	8	$8-2-\left[1.\left(\left\lceil \frac{8}{3} \right\rceil - \left\lfloor \frac{2}{3} \right\rfloor\right) - 1 + 1.\left(\left\lceil \frac{8}{4} \right\rceil - \left\lfloor \frac{2}{4} \right\rfloor\right) - 1\right] = 3$
3	3	3	[12-3+1,12]= [10,12]	12	$12-3-\left[1.\left(\left\lceil \frac{12}{3} \right\rceil - \left\lfloor \frac{3}{3} \right\rfloor\right) + 1.\left(\left\lceil \frac{12}{4} \right\rceil - \left\lfloor \frac{3}{4} \right\rfloor\right) - 1 + 1.\left(\left\lceil \frac{12}{6} \right\rceil - \left\lfloor \frac{3}{6} \right\rfloor\right) - 1\right] = 3$

Table 2

The first column on Table 2 indicates the number of the task whose slack is calculated. The second column shows the time when the calculus is performed, because slack is calculated when task finishes, slacks for tasks 1, 2 and 3 are evaluated at times 1, 2 and 3 respectively. The third column shows the worst response time of each task (equation (1)). This value can be calculated at the design time and stored on a table to be used during runtime. The forth column contains the inspection interval obtained applying *Lemma 3*. The fifth column shows the times, that belonging to the inspection interval, satisfy *Lemma 2*. Equation (3) should be evaluated at times of column 5 in order to get the maximum slack at each level. The sixth column presents the result of equation (3). Because this is a simple example, it had to be calculated in just one point for each task.

5 Implementing the Fast Slack method

In this section we describe how a set of counters is implemented during runtime in order to reduce the computational cost of the Fast Slack method. The mechanism is similar to the one proposed in [1].

A set of counters keeps track of the slack which may be stolen at each priority level. The slack available to execute non-critical tasks will be the minimum value of the counters. Analysing how the slack varies during runtime, we can state the following rules to manage the counters:

- When a lower priority task is executed, higher priority counter should be decremented in an amount equal to the time executed.
- When a non-critical task is executed or there exists an idle time, all the counters should be decremented in an amount of time equal to the time consumed.
- When a task finishes its execution, the Fast Slack method is calculated to initialise its counter.

- When a task finishes its execution before its Worst Case Execution Time, all lower priority counters should be incremented in a time equal to the Worst Case Execution Time minus the time that the task used to complete.

We can note that if system is schedulable then no counter can hold a negative value.

On Table 3 we can see the counters of the system of the example in the previous section. Bold numbers indicate that slack was calculated because the task finished its execution. The last row shows the slack available to execute non-critical task.

t	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
S_1	**2**	**4**	3	2	**4**	3	2	**4**	3	2	**4**	3	2
S_2	**1**	1	**3**	2	2	**4**	3	3	2	**3**	3	2	1
S_3	**1**	1	1	**3**	3	3	2	2	**3**	3	3	2	1
S	1	1	1	2	2	3	2	2	2	2	3	2	1

Table 3.

6 Experimental results

In this section we compare the performance of the Fast Slack algorithm to the slack stealing algorithm proposed in [1]. We used the same assumptions considered in [1] in order to get a fair comparison. We used three different groups of tasks:

- Group **A**: comprised of 10 real-time tasks. The utilization factor of the systems ranges from 0.4 to 0.9 with steps of 0.1, and there are 200 real-time systems for each utilization factor. The periods of the tasks follow an *exponential* composition; 4 tasks with periods in the range 25 to 100 units, 3 tasks with periods between 100 and 1000 units and a further 3 tasks with periods between 1000 and 10000 units.
- Group **B**: comprised of 20 real-time tasks. The utilization factor of the systems ranges from 0.4 to 0.9 with steps of 0.1, and there are 200 real-time systems for each utilization factor. The periods of the tasks follow an *exponential* composition; 7 tasks with periods in the range 25 to 100 units, 7 tasks with periods between 100 and 1000 units and a further 6 tasks with periods between 1000 and 10000 units.
- Group **C**: comprised of 50 real-time tasks. The utilization factor of the systems ranges from 0.5 to 0.9 with steps of 0.1, and there are 200 real-time systems for each utilization factor. The periods of the tasks follow an *exponential* composition; 17 tasks with periods in the range 25 to 100 units, 17 tasks with periods between 100 and 1000 units and a further 16 tasks with periods between 1000 and 10000 units.

Each real-time system was generated as follows. First, the periods of the tasks were chosen at random from the desire range. Deadlines were set equal to the periods. The tasks were then sorted into deadline monotonic priority order. Next, random execution times were assigned, highest priority first. The computation times were constrained such that the partial task remained feasible according to a sufficient and necessary schedulability test. Finally tasks sets with an utilization level differing by more than 0.5% from that required were discarded. We simulated each real-time system for 15 consecutives releases of the lowest priority tasks after the worst case of load. We counted the number of iterations that each algorithm needs to get the slack available. The results presented are the averages over each group.

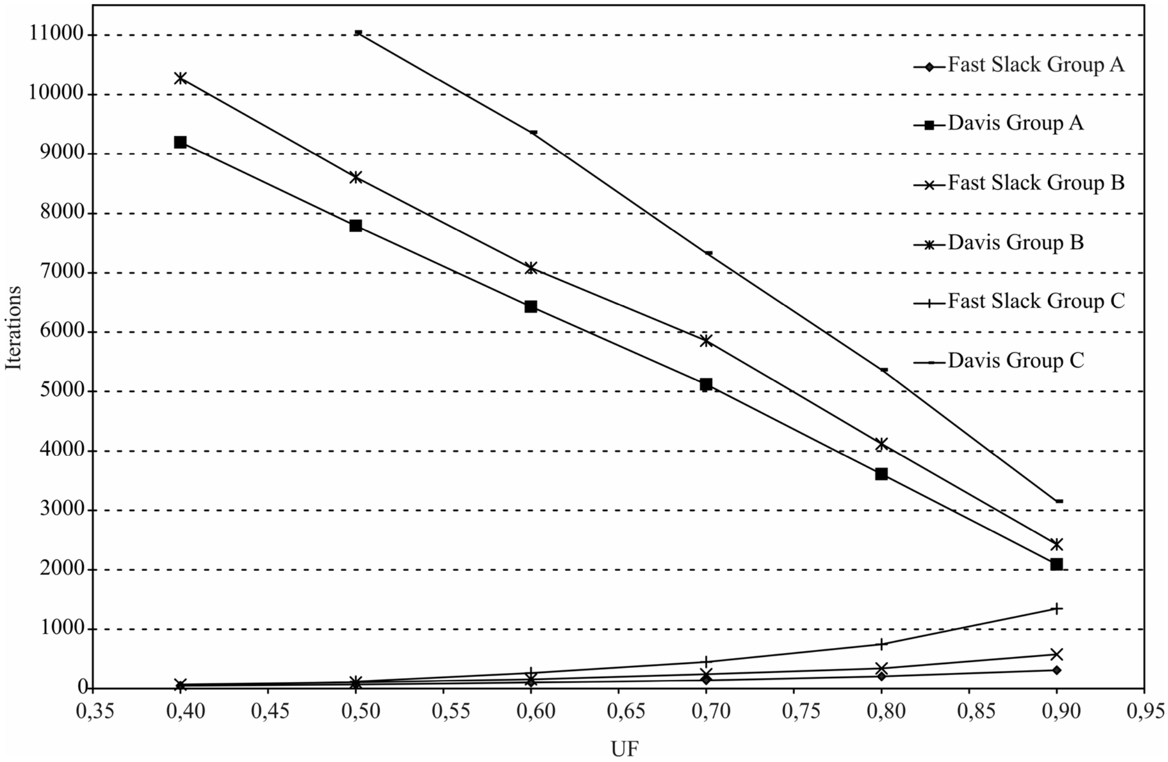

Figure 4. Comparison of Fast Slack vs Davis' slack-stealing method.

Figure 4 shows the average number of iterations for each utilization factor of groups A, B and C that Fast Slack and the slack-stealing mechanism proposed in [1] required to get the slack available. We can note that the number of iterations that Fast Slack requires is much less for low utilization factors and remains lower for high utilization factors.

From these results we can conclude that the number of iterations that Fast Slack requires increases as the utilization factor increases because the inspection interval increases as well and consequently equation (3) should be calculated more times.

The number of iterations of the method proposed in [1] decreases as the utilization factor increases because it calculates the slack based on the intervals when the system is idle. At low utilization factors the systems is idle most of the time and consequently the mechanism has to be evaluated at each one of these idle intervals. As the utilization factor increases, the number of idle intervals decreases and then the number of iteration decreases.

The simulations were performed considering integer execution times. When fractional executions times are considered, then the Davis's method increases the number of iterations whilst the Fast Slack method remains with the same performance.

7 Conclusions

In this paper we have shown that it is possible to reduce the complexity of a Slack Stealing method by reducing the inspection interval. Comparisons showed that the performance of the Fast Slack method is much higher that the method proposed in [1]. Besides, we proposed the equation (6) that can be used online to calculate the number of iterations needed to get the slack available in advance. With this improvement it is possible to get online the amount of slack available and consequently it can be applied to embedded systems.

8 References

[1] R. I. Davis, K. W. Tindell, and A. Burn, "*Scheduling Slack Time in Fixed-Priority Preemptive Systems*," *Proceedings of the Real Time System Symposium*, pp. 222-231, 1993.

[2] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "*Enhanced Aperiodic Responsiveness in Hard Real-Time Enviroments,*" presented at IEEE Real-Time Systems Symposium 1987.

[3] B. Sprunt, J. P. Lehoczky, and L. Sha, "*Exploiting Unused Periodic Time For Aperiodic Service Using The Extended Priority Exchange Algorithm*," 1988.

[4] L. Sha, B. Sprunt, and J. P. Lehoczky, "*Aperiodic Task Scheduling for Hard Real-Time Systems*," *The Journal of Real-Time Systems*, vol. 1, pp. 27-69, 1989.

[5] G. Fohler, T. Lennvall, and G. Buttazzo, "*Improved Handling of Soft Aperiodic Tasks in Offline Scheduled Real-Time Systems using Total Bandwidth Server,*" presented at 8th IEEE International Conference on Emerging Technologies & Factory Automation, Nice France, 2001.

[6] S. Ramos-Thuel and J. P. Lehoczky, "*On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems,*" presented at Real-Time Systems Symposium 1993.

[7] R. I. Davis, "*Approximate Slack Stealing Algorithms for Fixed Priority Pre-Emptive Systems*," Real-Time Systems Research Group, University of York, York, England 1994.

[8] T.-S. Tia, J. W. Liu, and M. Shankar, "*Aperiodic Request Scheduling in Fixed-Priority Preemptive Systems*," Department of Computer Science, University of Illinois at Urbana-Champaign UIUCDCS-R-94-1859, 1994.

[9] M. Joseph and P. Pandya, "*Finding Response Times in Real-Time System*," *The Computer Journal (England)*, vol. 29, pp. 390-395, 1986.

[10] M. Joseph, *Real-time Systems Specification, Verification and Analysis*: Prentice Hall International, 2001.

[11] J. P. Lechoczky, "*Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadline*," *Proceedings 11Th IEEE Real-Time Systems Symposium, Lake Buena Vista, Fl, USA*, pp. 201-209, 1990.

[12] N. C. Audsley, A. Burns, M. F. Richarson, and A. J. Wellings, "*Hard Real-Time Scheduling: The Deadline Monotonic Approach,*" presented at Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA, USA, 1991.

[13] R. I. Davis, "*Dual Priority Scheduling: A Means of Providing Flexibility in Hard Real-Time Systems*," Department of Computer Science, University of York, York, England 1995.