# Slack Time Computation for Temporal Robustness in Embedded Systems

Serge Midonnet, Damien Masson and Rémi Lassalle

**Abstract**—We propose to handle execution duration overruns (temporal faults) in real-time embedded systems. When a temporal fault occurs, the slack time can be dynamically determined and assigned to the faulty task in order to complete its treatment. This mechanism improves the temporal robustness of real-time systems. We demonstrate that an approximate slack stealer algorithm like the MASS algorithm is a good solution for real-time embedded systems. We validate the feasibility of this approach by an implementation on the Lego Mindstorm NXT platform.

**Index Terms**—Slack Time, Robustness, LejosRT.

## 1 Problem Description

The constraints in a hard real-time system are defined such that no deadlines of any task are missed. Moreover, the worst case execution time (WCET) of a task is estimated or computed in order to ensure that the task never runs for a duration longer than its WCET. But this determination is very difficult to achieve.

If a task overruns its WCET, the system may fail unless this WCET overrun does not cause any deadline misses. We need to detect and isolate temporal faults to protect the system against faulty tasks.

Without this protection, known as fault isolation, it is impossible to build robust real-time applications. The problem is to know how long a periodic real-time task can exceed its WCET without violation of the fault isolation property.

In section 3 we briefly present a static solution which consists of the determination of the allowance, a value we can add to the WCET of each task.

The drawback of this approach is that it provides a very pessimistic value because it is calculated in the worst case. The approach we propose in section 4 is to use a dynamic value called the slack time. This approach also has the advantage to collect gain time, ie the time freed in the system by a task completing before its WCET.

The rest of this paper is organized as follows: Section 2 presents our task model and assumptions. Section 3 introduces the concept of allowance to increase robustness in real-time systems and related works. Section 4 explains the slack time computation algorithms used in this work. Section 5 describes the target platform we used. We expose in Section 6 experimental results on comparative overhead measurement for DASS and MASS implementations on lejosRT. Finally we propose in Section 7 a proof of the concept based on three scenarios executed.


- *D. Masson and S. Midonnet are with the Paris-Est university and the "Laboratoire d'informatique Gaspard-Monge" (UMR CNRS 8049 LIGM).*
- *D. Masson is also affiliated to ESIEE Paris.*
- *Rémi Lassalle is Capitaine in the french Nationale Gendarmerie and a second year student of the Ecole Nationale Supérieure de Techniques Avancées (ENSTA).*


## 2 Assumptions and Task Model

In this paper, we consider an application built from a set of $n$ periodic real-time tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_i$ is characterized by a period $T_i$, a worst-case execution time $C_i$ and a relative deadline $D_i$. We consider a static-priority scheduling on a single processor. We assume that tasks are indexed by decreasing priority.

## 3 Static Allowance for Robustness

One of the available approaches to increase the WCET of a task with respect of the fault isolation property is to compute a value denoted Allowance by Bougueroua et al. [1]. The Allowance is the duration which can be added to the WCET of a task such that all tasks meet their deadlines.

A first approach is to compute the Allowance from the WCRT. For a given value of allowance $A_i$, this method consists of checking that the system remains schedulable when the execution duration of task $\tau_i$ is equal to $C'_i = C_i + A_i$. The maximal value for $A_i$ is found by a binary search.

Another approach to compute the allowance on the execution duration is the *sensitivity analysis*. This approach has been introduced by Bini et al. [2]. It is interesting as it is non recursive feasibility condition. The authors propose to consider the system only at time corresponding to the activation time of the highest priority tasks.

Both approaches provide the optimal value of the allowance value but their complexities are too high to be used [3] online.

## 4 Slack Stealing for Robustness

Slack stealer algorithms was introduced in [4] in order to address the problem of jointly scheduling hard periodic tasks and soft aperiodic events. The general idea is to compute at a time instant $t$ where there is an aperiodic pending request, for each hard real-time task, a value, called the slack. This value corresponds to the amount of time the task can suspend its execution without missing its deadline. Then a time interval equal to the minimum among these values can be used to handle aperiodic traffic, or in our work to let a task commit a temporal fault.

The first proposed approach to compute slack time was based on a table generated off line. This solution suffers to a big memory complexity issue due to the static table storage. So a dynamic exact approach (DSS) was proposed in [5], and a dynamic approximate one to address the time complexity issue of the exact approach was proposed in [6]. This dynamic approximation relies on the computation of a lower bound on the slack values and is called *Dynamic Approximate Slack Stealer* (DASS).

We proposed in [7] the *Minimal Approximate Slack Stealer* (MASS) algorithm. This algorithm also computes a lower bound on the slack values. The time complexity and the overhead of this algorithm are lower than the DASS ones, but this is at the expense of losing precision on the computed lower bound. However we demonstrated through extensive simulations that the loss of precision against DASS was negligible.

### 4.1 Dynamic Slack Stealing (DSS)

The algorithm relies on the determination at time $t$ and for each priority level of the available slack time, $S_i(t)$, also denoted as the i-level laxity. It represents the maximum amount of time the task $\tau_i$ can be delayed without missing its deadline. This value is equal to the number of unused time units at priorities higher or equal to $i$ between $t$ and the next $\tau_i$ deadline. The length of this interval is noted $d_i(t)$.

The number of "stealable" time units in the system, the system laxity, is the minimum value among the i-level laxities : $S(t) = \min_{\forall i} S_i(t)$.

To compute the $S_i(t)$ values, the interval between $t$ and the next $\tau_i$ deadline that we denote $[t, t + d_i(t))$ is viewed as a succession of *i-level* busy periods[1] and *i-level* idle periods[2]. Then, $S_i(t)$ is the sum of the *i-level* idle period lengths.

Equations to compute respectively the end of a busy period starting at time $t$ and the length of an idle period starting at time $t$ can be derived from the feasibility analysis theory [8]. These two equations are then recursively applied until the reach of the next deadline to determine $S_i(t)$.

1. periods where the processor is servicing priorities higher or equal to $i$

2. processor idle periods or periods where processor serves priorities lower than $i$

Assuming that there is a time $t$ where the $S_i(t)$ was up to date for all tasks, it is possible to compute $S_i(t')$ as follow:

1) if none of the periodic hard real-time tasks ends in $[t, t')$
   a) if the processor is idle or executing soft aperiodic requests
   $$\forall j : S_j(t') = S_j(t) - (t - t') \quad (1)$$
   b) if the processor is executing hard periodic task $\tau_i$
   $$\forall j < i : S_j(t') = S_j(t) - (t - t') \quad (2)$$
2) if hard real-time task $\tau_i$ ends at time $t'' \in [t, t')$, $S_i(t'')$ has to be computed using the recursive analysis described at the beginning of this section.

This algorithm is not directly usable because of the time complexity of the recursive computation of the $S_i(t)$ to perform at each task ends. However, this part can be replaced by the computation of a lower bound.

### 4.2 Dynamic Approximate Slack Stealer (DASS)

Since $S_i(t)$ is the sum of the i-level idle period lengths in the interval $[t, t + d_i(t))$, [6] proposes to estimate this quantity by computing a bound on the maximal interference the task $\tau_i$ can suffer in this interval. A bound on this interference is given by the sum of the interferences from each task with a higher priority than $\tau_i$. Then Equation 3[3] gives the interference suffered by a task $\tau_j$ from a task $\tau_i$ in an interval $[a, b]$.

$$I_i^j(a,b) = c_i(t) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0) \quad (3)$$

Where $c_i(t)$ is the remaining cost of current instance of task $\tau_i$ at time $t$.

The function $f_i(a, b)$ returns the $\tau_i$ instance number which can begins and completes in $[a, b]$. It is given by Equation 4.

$$f_i(a,b) = \left\lfloor \frac{b - x_i(a)}{T_i} \right\rfloor_0 \quad (4)$$

The function $x_i(t)$ represents the first activation of $\tau_i$ which follows $t$. Then the interference is composed by the remaining computation time needed to complete the current pending request, by a number of entire invocations given by $f_i(a, b)$, and by a last partial request.

A lower bound on the $S_i(t)$ value is given by the length of the interval minus the sum of the interferences from each task with a higher or equal priority than $\tau_i$. It is recapitulated by Equation 5.

$$S_i(t) = \left( d_i(t) - t - \sum_{\forall j \leq i} I_j^i(t, d_i(t)) \right)_0 \quad (5)$$

3. the notation $(x)_0$ means $max(x, 0)$

We can then implement DASS by adding in the scheduler the following operations at the start and end of each periodic instance ($dt$ refers to the elapsed time since the last start or end of a periodic task):

- **Beginning of periodic task $\tau_k$:** let $l$ be the priority level of the system before the activation of $\tau_k$. We have
$$\forall i < l, S_i(t) = S_i(t') - dt \quad (6)$$
- **End of periodic task $\tau_k$:**
$$\forall i < k, S_i(t) = S_i(t') - dt \quad (7)$$
$$S_k(t) = \left( d_k(t) - t - \sum_{\forall j \leq k} I_j^k(t, d_k(t)) \right)_0 \quad (8)$$

### 4.3 Minimal Approximate Slack Stealer (MASS)

In order to reduce the complexity and the overhead of the added operations, we decompose the maximum available slack per task ($S_i(t)$) into two different pieces of data: first $\bar{w}_i(t)$, the maximum possible work at priority $i$ regardless of lower priority processes ; second $\bar{c}_i(t)$, the effective hard real-time work to process at the instant $t$ at priority $i$, ie task $\tau_i$ remaining cost. We then have $S_i(t) = \bar{w}_i(t) - \bar{c}_i(t)$.

Then, $\bar{w}_i$ value only have to be updated at the end of periodic instances. Moreover, they are not recomputed from scratch, but the computation is incremental. This permits to always compute the interferences on a time interval equal to the task period, which can be performed in constant time. However, we still have to maintain the $\bar{c}_i$ values at each preemption. We denote the time between two periodic task ends by $dt_f$ and then time between two periodic task begins by $dt_b$.

1) **End of periodic task $\tau_k$.**

$$\begin{array}{ll} \forall i < k, & \bar{w}_i(t) = \bar{w}_i(t_f) - dt_f \\ \forall i > k, & \bar{w}_i(t) = \bar{w}_i(t_f) - dt_f + \bar{c}_i \\ i = k, & \left\{ \begin{array}{lcl} \bar{w}_i(t) & = & \bar{w}_i(t_f) - dt_f + T_i - I_i(t) \\ \bar{c}_i(t) & = & C_i \end{array} \right. \end{array} \quad (9)$$

2) **Beginning of a periodic task preempting $\tau_k$.**

$$\bar{c}_k(t) = \bar{c}_k(\max(t_f, t_d)) - \min(dt_f, dt_d) \quad (10)$$

## 5 LejosRT

Lego Mindstorms NXT is a programmable robotics kit released by Lego in late July 2006. The main component in the kit is a brick-shaped computer called the NXT Intelligent Brick. It can take input from up to four sensors and control up to three motors. In addition, a lot of custom sensors e.g. gyro, compass, color detector, movement detector, temperature detector, using a wide range of technologies, analog, digital can be used.

LeJOS NXJ is an open source alternative firmware for the brick that implements a Java Virtual Machine. We forked the Lejos Project to the LejosRT project to introduce a subset of the Real-Time Specification for Java (RTSJ) [9]. We use this platform available on-line on sourceforge as a validation target for this work.

## 6 Comparative Overheads DASS/MASS on LejosRT

We present in this section an overhead comparison of DASS and MASS implementations on lejosRT.

A main issue is to obtain an accurate time measure on the system. We chose to modify the virtual machine in order to perform a time measure when the first thread starts and another one when a thread ends a periodic instance.

### 6.1 Experience description

We consider a program with $n$ real-time threads with the same release parameters (period and deadline) but with $n$ different priorities. Moreover we shift start times in order to start first the thread with the lowest priority and last the one with the highest priority. That permits to maximize the number of preemptions. The period of the thread with the lowest priority is set shorter than the other one in order to have this thread preempted twice be each other ones. We measure the needed time to execute two instances for each thread. The system load is then maximal (100%).

### 6.2 Experimental results

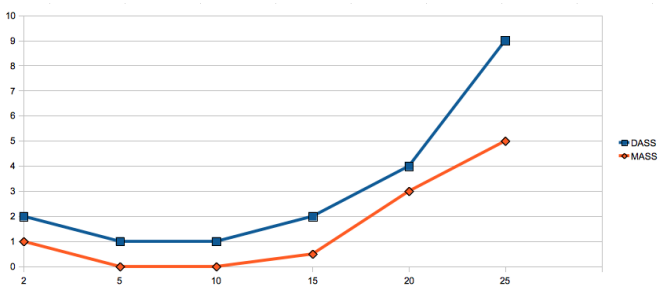


Fig. 1. System overheads with DASS and MASS depending on task number (in ms)

Figure 1 presents the results obtained for a range of thread number from 2 to 25. Each test was performed ten times and the values reported in Y axis are the average differences between the time execution of the program on LejosRT without any slack computation and with the concerned algorithm (respectively DASS and MASS). On the X axis is reported the number of thread.

We can note that the MASS overhead is always lower than the DASS one. This was the expected result, that this experiment confirmed on real implementations. Moreover, the more threads we have, the higher the measured difference is. There is an exception with 5 and 10 tasks where the MASS overhead is negligible in regards to the overheads of the scheduling algorithm on the virtual machine.

## 7 Proof of concept : executions traces

Three scenarios are presented here. On the following figures, the gray zones represent cost overruns. The times are in milliseconds.

### 7.1 One task commits a cost overrun

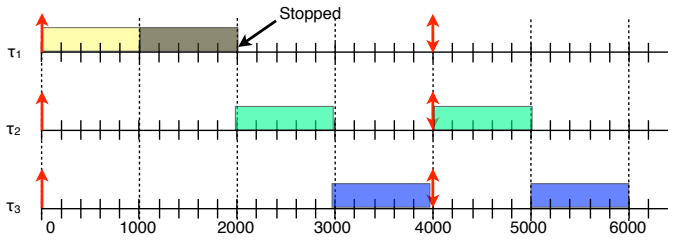


Fig. 2. Consumption of all the available laxity by one thread

The system is composed by 3 periodic tasks which are synchronously activated. The task with the higher priority performs an infinite loop, and so will obviously overrun its cost. We can see that the mechanism succeed in letting it execute as long as possible. The two other tasks respect their deadlines.

### 7.2 Two tasks commit cost overrun

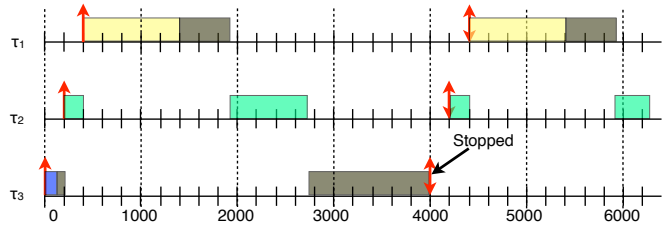


Fig. 3. Consumption of the available laxity by two threads

This time, both task $\tau_1$ and $\tau_3$ overrun their cost. Moreover, we shifted the task activations to observe a preemption during the slack time utilization. So $\tau_3$ starts first, overruns its cost, but continue its execution since there is laxity in the system. It is preempted by $\tau_2$ which is preempted in turn by $\tau_1$. $\tau_1$ overruns its cost and consumes a part of the system laxity. Finally $\tau_1$ ends its execution, $\tau_2$ can resume and complete, and $\tau_3$ consumes the rest of the laxity.

### 7.3 Gain time collection

This last experiment is quite the same as the previous, except we make $\tau_2$ finish before its WCET.

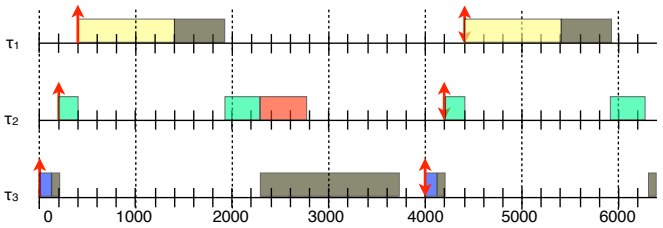


Fig. 4. Example with gain time

We can see that the gain time generated is automatically collected by MASS in the system laxity. That permits $\tau_3$ to ends before its deadline.

## 8 Conclusion

We have proposed an approximate slack stealing algorithm (MASS) and we prove in this paper its usability to enforce the robustness of real-time systems. We demonstrate that this mechanism has a lower overhead than the other existing algorithm: DASS, which is a researched quality since we want to deal with cost overruns. Indeed, the lower computation time the mechanism takes, the more time to handle the cost overrun we have. Finally, we implemented it and evaluated its behavior in the presence of temporal faults on a real embedded platform (LejosRT).

## References

[1] L. Bougueroua, L. George, and S. Midonnet, "Temporal robustness of real-time architectures specified by estimated wcets," *International Journal on Advances in Software*, vol. 2, no. 4, pp. 359–371, Dec. 2009.

[2] E. Bini, M. Di Natale, and G. C. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," in *Proceedings of the 18th Euromicro Conference on Real-time Systems (ECRTS)*. IEEE Computer Society, 2006, pp. 13–22. [Online]. Available: papers/Bini2006.pdf

[3] F. Fauberteau, S. Midonnet, and L. George, "A robust partitioned scheduling for real-time multiprocessor systems," in *Proceedings of IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*. Springer Science and Business Media, 2010, p. (to appear).

[4] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems," in *proceedings of the 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, Dec. 1992, pp. 110–123.

[5] R. I. Davis, K. Tindell, and A. Burns, "Scheduling slack time in fixed priority pre-emptive systems," in *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93)*, 1993, pp. 222–231.

[6] R. I. Davis, "On exploiting spare capacity in hard real-time systems," Ph.D. dissertation, University of York, 1995.

[7] D. Masson and S. Midonnet, "Userland Approximate Slack Stealer with Low Time Complexity," in *16th Real-Time and Network Systems (RTNS'08)*, Rennes, France, Oct. 2008, pp. 29–38, (10 pp.).

[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993. [Online]. Available: citeseer.ist.psu.edu/audsley93applying.html

[9] http://www.rtsj.org, "The real-time specification for java (RTSJ)," http://www.rtsj.org. [Online]. Available: http://www.rtsj.org