Temporal Robustness of Real-Time Architectures Specified by Estimated WCETs

Lamine Bougueroua
LRIT
ESIGETEL
Avon, France
lamine.bougueroua@esigetel.fr

Laurent George
LACSC
ECE
Paris, France
lgeorge@ieee.org

Serge Midonnet
LIGM, UMR CNRS 8049
Université Paris-Est
Champs sur Marne, France
Serge.Midonnet@univ-paris-est.f

***Abstract*— Real-time dimensioning depends on the Worst Case Execution Time (WCET) of its tasks. Using estimated WCETs for the dimensioning is less conservative but execution overruns are more likely to happen. Fault tolerant mechanisms must be implemented to preserve the real-time system from timing failures, associated to late task termination deadlines misses, in the case of WCETs overruns. We show in this article how to compute the extra duration (allowance) on the WCETs that can be given to faulty tasks while still preserving all the deadline constraints of the tasks. This allowance is used on-line to tolerate WCET overruns. We present a mechanism called the Latest Execution Time (LET) using the allowance of the tasks for the temporal robustness of real-time systems. This mechanism only requires classical timers. Its benefits are presented in the context of a java virtual machine meeting the Real-Time Specification for Java (RTSJ) with estimated WCETs.**

Real-time System; fault tolerance; estimated WCET; allowance; slack time; temporal robustness

I. INTRODUCTION

Real-time scheduling theory can be used at the design stage for checking the timing constraints of a real-time system, whenever a model of its software architecture is available. In specific cases, standard real-time scheduling analysis techniques can significantly shorten the development cycle and reduce the time to market. The correct dimensioning of a real-time system depends on the determination of the Worst Case Execution Time (WCET) of the tasks.

Based on the WCET, a feasibility condition (e.g. [2][3][4]) can be established to ensure that the deadlines of all the tasks are always met, whatever their release times. The computation of the WCET can be performed either by analyzing the code on a given architecture or by measurement of the execution duration [5]. In both cases, the correctness of the WCET is hard to guarantee. The WCET depends on the condition of execution; the type of architecture, memory or cache. This can lead to an imprecise WCET. Also a very complex analysis may be necessary to obtain it. The obtained WCET can therefore be either pessimistic or optimistic compared to the exact execution duration obtained at run time. In the first case, we have reserved CPU resources that may not be used, leading to a pessimistic dimensioning of the system but the deadlines can always be guaranteed. In the second case we might have execution overruns, i.e. task durations exceeding their WCET but more CPU resources left to deal with such a situation. We are interested in this paper in the second case observing that an execution overrun does not necessarily lead to a deadline miss. With enough free CPU resources, a system can self stabilize and still meet the deadlines of all the tasks. So the problem consists of determining how much time may be allocated to the execution overrun.

In this paper, we consider that the WCETs are estimated by benchmark on a given architecture. The lack of precision on the WCETs is taken into account in this paper with the concept of allowance introduced in section 3.

We consider two types of faults captured by two errors: the *ExecutionOverrun* error and the *DeadlineMiss* error. In the first case, the task execution duration exceeds its WCET while in the second case the task does not meet its deadline. Yet, among all possible execution overruns, only those leading to a deadline miss, if nothing is done to correct the system, should be considered. We therefore need to determine how long we can let a faulty task proceed with its execution without compromising the deadlines of all other tasks. We call this duration the allowance of the task. Based on the allowance, we can determine when an execution overrun error should be raised.

In this paper, we consider a set $\tau = \{\tau_1, \tau_2,..., \tau_n\}$ of n sporadic or periodic tasks with timeliness constraints.

A task τ_i, is defined by:

- C_i, the estimated Worst Case Execution Time
- T_i, the inter-arrival time also called the period
- D_i, the relative deadline (a task released at time t must be executed by its absolute deadline $t + D_i$. We consider in this paper, that for any task τ_i , $D_i \leq T_i$
- P_i, the priority

In this paper, we first present a short description of related work (see Section 2). We then focus on the allowance of the WCETs in Section 3 when tasks are scheduled with Fixed Priority/highest priority first (FP/HPF) scheduling (e.g. [3][4][6]). We then show how to use the allowance on-line, with a mechanism called the Latest Execution Time (LET), in Section 4. The LET analysis was first introduced by Bougueroua and al. in [7]. Moreover [1], extends this analysis further by considering the EDF scheduling. In Section 5 we give results of some simulations, obtained from a tool we developed, so as to compare the different execution overrun management mechanisms. We do this firstly for a task deadline miss, and secondly for a task execution overrun, showing the benefits of the LET mechanism. Finally, we give some conclusions.

II. RELATED WORK

Most fault tolerant real-time systems present solutions to deal with deadline miss by stopping the execution of the tasks that miss their deadlines. In the case of overloaded systems, the idea is to stop some tasks so as to prevent the others from missing their deadlines and to come back to a normal load condition. Tasks are scheduled according to their importance: Locke's Best Effort Scheduling Algorithm in [8], D-Over in [9] and Robust EDF in [10]. The problem with this solution is that a task missing its deadline might already have had cascading effects on the other tasks of the system. The reaction might be too late.

In fact, a task which does not respect its deadlines might result in unacceptable delays on other lower priority tasks.

Several approaches have been considered in the state of the art: either in the dimensioning phase or on-line.

A. In a dimensioning phase:

In a dimensioning phase, a sensitivity analysis can be used to compute the maximum acceptable deviations on the WCETs, Period on Deadlines preserving the schedulability of the tasks (e.g. [11][12][13]). Most of the existing solutions to the sensitivity problem consider only one parameter can change. We place ourselves in this context for this article by considering the WCETs overrun. The reader interested by multidimensional analyses of sensitivity will be able to refer to work of Racu in [14] and [15], where a stochastic approach is proposed to deal with the variations of several parameters.

Bini in [13] shows how to calculate the maximum value of the multiplicative factor λ, applied to the set of tasks. In this case when scheduling FP of *n* periodic tasks, if for any task τ_i, $D_i \leq T_i$, then for any task τ_i, its WCET becomes $C_i + \lambda C_i$.

λ is the maximum value so that the set of tasks is schedulable. If for a set of tasks given, $\lambda < 0$ then initial set of tasks is not schedulable and it is necessary to reduce the WCET of λC_i so that the tasks set become schedulable.

The computation complexity of λ is pseudo-polynomial. The authors in [13] Show also how to calculate λ for a subset of tasks.

However, the use of a multiplicative factor λC_i for any task τ_i, gives more allowance to the task τ_i when its C_i is large, a choice which inevitably does not reflect the importance of the task.

In [16], the authors show how to determine for a scheduling FP, in the case $D_i \leq T_i$, the feasibility domain to *n* dimensions of WCETs (called C-space). They show that when the number of tasks is reasonable, it is possible to express the parametric equation of the WCETs feasibility domain in the form of a set of inequations to be tested. This approach is useful in a phase of dimensioning but from its polynomial pseudo complexity (the number of inequation is pseudo polynomial), is not applicable on line to determine if it is possible to continue the execution of a task exceeding its WCET.

Slack time analysis has been extensively investigated in real-time systems in which aperiodic (or sporadic) tasks are jointly scheduled with periodic tasks (e.g. [17][18][19]). In these systems, the purpose of slack time analysis is to improve the response time of aperiodic tasks or to increase their acceptance ratio. Yet, those approaches require high time overheads to determine the available slack time at a given time. Some research has been proposed to approximate the slack time (e.g. [19] and [20]). Nevertheless, the slack is computed for a single aperiodic task occurrence whereas the allowance is valid for all the requests of a periodic task. Computing the slack time for every periodic request would be time consuming.

B. On-line:

Some solutions consist of adapting the task parameters to the context of execution (e.g. approaches based on the variation of the periods (e.g. [21][22][23]) to obtain more or less precision). In the context of a network transmission, some transmission failures have been considered with the (m,k)-firm model [24] to tolerate m transmission failures among k.

This algorithm is classified as a best effort algorithm. In [25], the authors propose an extension of the model (m,k)-firm called Weakly-hard to consider non-consecutive failures.

The allowance on the WCETs proposed in Section 3 is computed with a sensitivity analysis with a sliding windows fault model. Yet, applying an identical scaling factor to a subset of tasks leads to providing more allowance to the tasks having higher WCETs. We remove this constraint by analyzing several allowance sharing strategies in Section 4. We then propose a fault prevention mechanism to prevent an execution overrun error from leading to a deadline miss error. We present the concept of Latest Execution Time a task can proceed with its execution without compromising the real-time constraints of all the tasks. The use of estimated WCET for the dimensioning of a real-time system enables us to be less conservative. However, it is necessary to take into account the faults that result in WCETs overruns and to guarantee the temporal robustness of the system in the case of execution overrun faults. The temporal robustness in our context is defined as follows: a faulty task should not have any influence on the other correct tasks. In our context, this means that an execution overrun of a task should not lead to a deadline miss of any other correct task.

III. ALLOWANCE ON WCETS – PRINCIPLES

The *processor utilization* corresponding to a task set τ is defined in [2]: $U = \sum_{n=1}^{n} \frac{C_i}{T_i}$. From this definition, $U \times 100$ represents the percentage of processor utilization. A necessary condition for the feasibility of a task set is: $U \leq 1$.

We are considering a real-time system based on the preemptive Fixed Priority, highest priority first (FP) scheduling algorithm with an arbitrary priority assignment. Preemptive scheduling means that the processing of any task can be interrupted by a higher priority task.

We define for any task τ_i scheduled with FP:

- hp(i) - the set of tasks except τ_i having a priority higher than or equal to τ_i except τ_i;

- lp(i) - the set of tasks having a lower priority than τ_i;
- hpR(i) and lpR(i) which denotes at any time the tasks released, respectively in hp(i) and lp(i).

The allowance consists of computing the allowance in the WCET tolerated by the system. It represents the maximum execution time that can be added to the WCET of a task without compromising the deadlines of all the tasks. We now give a more formal definition of the task allowance.

Definition 1:

A temporal fault (WCET overrun) of a task τ_i is said to be isolated if it does not result in any deadline miss of the other tasks.

Definition 2:

The task allowance is the maximum available CPU resources allotted to a faulty task when it exceeds its WCET. In addition, it represents the maximum duration that can be added for the execution of a task without compromising the execution of other tasks.

A. *Identification of the available CPU Resources*

The identification of CPU resources consists in computing from the available system resources the maximum extra execution duration that can be given to faulty tasks without missing the deadline of any task. The computation of available resources must be carried out during the worst case scenario thus minimizing the allowance of the tasks. Considering timeliness constraints, the worst-case scenario for the respect of the task deadlines occurs for FP scheduling in the synchronous scenario, i.e. all tasks are at their maximum rate and released synchronously at time t=0 (e.g. [4]).

We now show that the available CPU resources are also minimized in the synchronous scenario.

Let $t_i^0 \geq 0$ be the first activation request time of the task τ_i in an arbitrary processor busy period. The duration of free CPU resources left available by the system at any time $t \geq 0$ is given by:

$$t - \sum_{i=1}^{n} \max\left(0, \left\lceil \frac{t - t_i^0}{T_i} \right\rceil\right) \times C_i$$

This duration is minimum for $\forall i \in [1, n], t_i^0 = 0$, corresponding to the synchronous scenario. In this paper, we are therefore interested only in the synchronous scenario for the computation of the allowance on WCETs as the synchronous scenario is a possible scenario. According to figure 1, representing the execution of the three tasks (the task parameters are defined in table 1). We would draw your attention to the availability of many free CPU intervals resulting from processor idle times.

Let us suppose that none of the three tasks have exceeded its execution duration. In this case, the average processor utilization of the available resources U_{free} is equal to:

$$U_{free} = 1 - \sum_{n=1}^{n} \frac{C_i}{T_i}$$

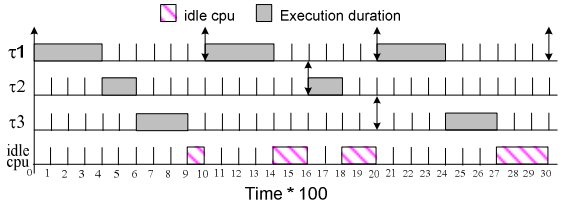

Figure 1. Available resources in the system.

TABLE 1. Average duration of free CPU resources

Task	C_i	D_i	T_i	Q_i	P_i
τ_1	400	1000	1000	325	High
τ_2	200	1600	1600	520	Medium
τ_3	300	2000	2000	650	Low

Those idle times could be used by faulty tasks. However, the amount of free resources, for each task, is not constant for each activation of the task. The average duration of free resource for a task τ_i is given by the following equation:

$$Q_i = U_{free} \times T_i$$

Table 1 gives the value of Q_i for all the three tasks τ_1, τ_2 and τ_3.

B. *Discussion - margin analyzes*

The use of the average duration of free CPU resources does not guarantee the respect of task deadlines. For example, in figure 2(synchronous scenario for the set of tasks given in table 1), during the first activation of task τ_2, the CPU resources which are really available for this task, in the event of a fault, are 300 units of time. This is less than the average quantity of available resource during future executions. (Q_2 = 520). The use of Q_2 leads to a temporal failure for task τ_3 activated at t=0.

We can use the total free resources during the lcm (least common multiple) of the periods of the tasks (also called the hyper period). The amount of available free resource during the hyper period is represented by the following equation:

$$Q_{hp} = U_{free} \times lcm(T_1, ..., T_n)$$

We obtain Q_{hp} = 2600 units of time in our example. However, this solution cannot guarantee the respect of the timeliness constraints of the tasks. In fact, task τ_1 as it has the greatest priority can use more CPU resources and this can result in deadline misses for lower priority tasks. Our goal is to use from amongst the available resources those which will not lead to deadline miss failures for all the tasks.

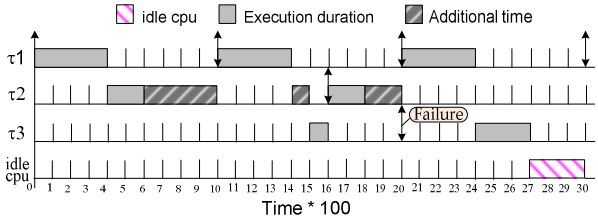

Figure 2. Available resources sharing

C. Allowance computation

The allowance on WCETs is related to the scheduling policy used by the system. In this article we are interested in FP scheduling with an arbitrary priority assignment. A classical feasibility condition for FP scheduling is obtained by calculating the worst-case response time R_i for any task τ_i. R_i is defined as being the longest duration time between release time and termination time [3]. A Necessary and Sufficient Condition (NSC) for the feasibility of Task Set Scheduled FP is then:

$$\forall i = 1..n,\ R_i \le D_i \text{ and } U \le 1$$

Many results showing the computation of the worst-case response time in a preemptive context are available [3][4]. In order to optimize the computation of the worst-case response time, [3] has proven that in the case of FP scheduling, when the deadlines are lower than or equal to the periods, the worst-case response time is obtained in the first activation of each task when all the tasks are released in the synchronous scenario.

Theorem 1: [3]

The worst-case response time R_i of a task τ_i of a non-concrete periodic, or sporadic, task set (with $D_i \le T_i$, $\forall i \in [1, n]$) is found in a scenario in which all tasks are at their maximum rate and released synchronously at a critical instant t=0. R_i is computed by the following recursive equation (where hp(i) denotes the set of tasks with higher priority than τ_i):

$$R_i^{m+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^m}{T_j} \right\rceil C_j$$

The recursion ends when $R_i^{m+1} = R_i^m = R_i$ and can be solved by successive iterations starting from $R_i^0 = C_i$.

We can easily show that R_i^m is not decreasing. Consequently, the series converges or exceeds D_i. In the latter case, task τ_i is not schedulable.

Remark:

A task is said to be non-concrete if its request time is not known in advance of its execution. In this paper, we only consider only non-concrete request times.

The computation of the allowance on WCETs can be carried out:

- In the dimensioning phase or on-line. Subsequently the tasks are activated by admission control, as long as the deadlines of the tasks can always be met. This is called the static approach.
- At each execution overrun detection. This is called the Dynamic approach.
- Once the static allowance is consumed then the dynamic computation is activated. This is called the Hybrid.

The computation of the allowance has a pseudo-polynomial time complexity (see property 1). Because of this we have adopted an approach based on a static computation of the allowance.

We consider in this paper a temporal fault model *[m/n]* corresponding to a fault model where there can be at most m≤n faulty tasks exceeding their WCET on a sliding window $W = min(T_1,\ldots, T_n)$. The allowance on the WCET of a task is related to the number of faulty tasks.

Firstly let us look at the case where m=1. The general case will be given in a later section. The allowance of a periodic or sporadic task τ_i, when only one task is faulty, called $A_{i,1}$, is obtained from the Necessary and Sufficient Condition (NSC) established in theorem 1.

Lemma 1:

The maximum allowance that can be given to the periodic or sporadic task τ_i, under the fault model [1/n], is equal to the maximum duration that can be added to the WCET of the tasks τ_i without missing the deadline of any task.

$$(1)\quad R_i^{n+1} = C_i + A_{i,1} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \le D_i$$

$$\forall \tau_j \in lp(i):$$

$$(2)\quad R_j^{n+1} = C_j + \sum_{\tau_k \in hp(j), k \ne i} \left\lceil \frac{R_j^n}{T_k} \right\rceil C_k + \left\lceil \frac{R_j^n}{T_i} \right\rceil (C_i + A_{i,1}) \le D_j$$

$$(3)\quad U + \frac{A_{i,1}}{T_i} \le 1$$

Proof:

Let τ_i be the faulty task. By assumption, the maximum execution duration of τ_i is then $C_i + A_i$. Applying theorem 1 with the new execution duration of τ_i we find the formula (1) of lemma 1. Formula (2) is similar to formula (1) but relates to the tasks with lower priority than τ_i. It consists of checking that their response time is always lower than their relative deadline when τ_i uses its allowance. Formula (3) of the lemma is the Necessary Condition for the feasibility of the task set taking into account the allowance of the faulty task.

Example:

The allowance computation is done for all the tasks given in the previous example (see table 1). The following table gives the allowance values for each task:

TABLE 2. Margin $A_{i,1}$ - temporal fault model [1/n]

Task	τ_1	τ_2	τ_3
$A_{i,1}$	250	300	500

For example, when a fault occurs, task τ_2 will be able to use an allowance equals to 300 time units. Figure 3 illustrates the synchronous scenario.

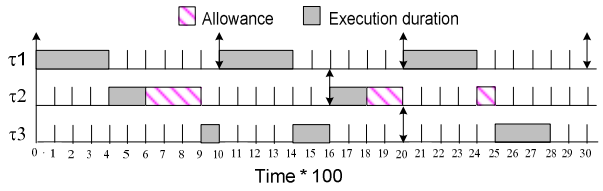

Figure 3. Task τ_2 Margin – example

These 300 time units guarantee that the other non faulty tasks will not be penalized by the faulty task and will respect their temporal constraints, as long as the faulty task does not consume more than its proper allowance.

IV. THE ALLOWANCE SHARING POLICY

Giving all the available resources to a faulty task is not an optimal solution in itself, as this can use up the totality of the resources without even correcting the fault. Worse than that, when consuming its allowance, the faulty task will reduce the allowance of more important tasks to zero. In that case, the recommended solution is the share of the available resources between the tasks. The resource sharing can be done in the following ways:

- Fair sharing allowance: this consists of allotting identical additional execution time to each faulty task, without taking into consideration the task parameters like: priority, importance and response time.
- Balanced allowance: this is based on tasks preference, i.e. when a task causes a fault, the available resources will be attributed according to importance of this task. In this case, a new parameter will be used: the weight, which will be attributed to each task according to its importance. The total amount of assigned weight must be equal to 1.

A. Fair sharing allowance

Let us look at the faulty tasks model [m/n]. In this section, we suggest an equal share of the static allowance between the m faulty tasks. For a task τ_i, we identify the set of *(m-1)* faulty tasks minimizing the allowance allotted to τ_i.

Lemma 2:

The maximum allowance that can be given to the faulty task τ_i, when using [m/n] fault model, is equal to the maximum duration that can be added to the execution of the faulty tasks. There are two conditions which must be fulfilled; firstly, (m-1) other faulty tasks must be taken into account and secondly, all deadlines must be respected.

$$(4)\quad R_i^{n+1} = C_i + A_{i,m} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + \sum_{\tau_j \in dis(i,m)} \left\lceil \frac{R_i^n}{T_j} \right\rceil A_{i,m} \le D_i$$

$\forall \tau_j \in lp(i)$:

$$(5)\quad R_j^{n+1} = C_j + \sum_{\tau_k \in hp(j), k \neq i} \left\lceil \frac{R_j^n}{T_k} \right\rceil C_k + \sum_{\tau_k \in dis(i,m) \cup \tau_i} \left\lceil \frac{R_j^n}{T_k} \right\rceil A_{i,m} \le D_j$$

$$(6)\quad \mathrm{U} + \sum_{\tau_j \in dis(i,m) \cup \tau_i} \frac{A_{i,m}}{T_j} \le 1$$

Where *dis(i,m)* = set of *(m-1)* which comprises the most unfavorable task execution time for the task τ_i in the event of faults. The most unfavorable task execution time is that which occurs when tasks are liable to consume most of the available resources, i.e. tasks which maximize the quantity of work $\left\lceil \frac{R_i}{T_j} \right\rceil$).

Proof:

Let τ_i be the faulty task. Let us suppose that there exists, at the most, *m* faulty tasks. In the case of equal shares, for a faulty task τ_i, each faulty task τ_j has an allowance $A_{i,m}$ added to its WCET. The tasks set with priority higher than the task τ_i will be divided into two sub-groups according to fault model of the task (faulty or not faulty). Applying theorem 1 with the new execution duration of τ_i we find the formula (4) of lemma 2.

The formula (5) consists of checking that the response time, of tasks with low priority than τ_i, is always lower than their relative deadline when τ_i consumes its allowance. The formula (6) of the lemma is the Necessary Condition for the feasibility of the task set taking into account the allowance of the faulty tasks.

Example:

We consider the parameters of the previous example (see table 1):

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- The tasks τ_1, τ_2 and τ_3 have decreasing priorities.
- Any task can be faulty.

The following table gives the allowance values for each task:

TABLE 3. Margin value - temporal fault model [m/n]

Task	$A_{i,m}$		
	m=1	m=2	m=3
τ_1	250	125	100
τ_2	300	125	100
τ_3	500	166	100

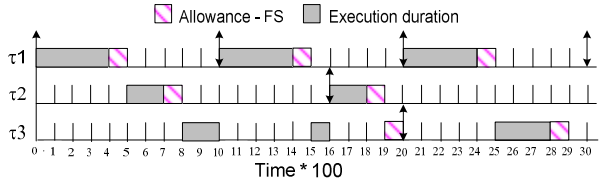

Figure 4. Margin example - fair sharing allowance

Figure 4 illustrates the scenario in which all the tasks are faulty, in this case no task misses its deadline if the faulty tasks do not consume more than their allowance.

Property 1:

The computation of the allowance, using FP and [m/n] fault model, has a pseudo-polynomial time complexity.

Proof:

The formula (6) of lemma 2 limits the maximum allowance for a task τ_i:

$$A_{i,m} \le \frac{1-U}{\sum_{\tau_j \in dis(i,m) \cup \tau_i} \frac{1}{T_j}}$$

Furthermore, we know that the deadline D_i is considered as an upper bound on the iteration number for the computation of the task response time ($R_i \le D_i$). This computation is carried out for a task τ_i and for all the tasks with lower priority than τ_i.

Thus we have $D_i + \sum_{\tau_j \in hp(i)} D_j$ iterations to execute. It follows that:

$$\left(D_i + \sum_{\tau_j \in hp(i)} D_j\right) \times \frac{1-U}{\sum_{\tau_j \in dis(i,m) \cup \tau_i} \frac{1}{T_j}} \le n \times (1-U) \times \max_{i=1..n}(D_i) \times \max_{i=1..n}(T_i)$$

The complexity is polynomial for each task, it takes $O(n \max_{i=1..n}(D_i) \max_{i=1..n}(T_i))$ time, thus the whole computation for n tasks takes $O(n^2 \max_{i=1..n}(D_i) \max_{i=1..n}(T_i))$, thus the complexity becomes pseudo-polynomial.

B. Balanced allowance

The share of the available free resources between faulty tasks should not be always being fair. A balanced approach takes into account the importance of a task. In that case, an additional parameter is used to balance the allowance among faulty tasks: a weight (φ_i) associated to task τ_i. The higher weight, the greater the importance.

Lemma 3:

The maximum allowance that can be given to the faulty task τ_i, when using [m/n] fault model, is equal to the maximum duration that can be added to the duration of the faulty tasks according to their weight.

$$(7)\quad R_i^{n+1} = C_i + A_{i,m} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + \sum_{\tau_j \in dis(i,m)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \left\lfloor A_{i,m} \times \frac{\varphi_j}{\varphi_i} \right\rfloor \le D_i$$

$$\forall \tau_j \in lp(i):$$

$$(8)\quad R_j^{n+1} = C_j + \sum_{\tau_k \in hp(j)} \left\lceil \frac{R_j^n}{T_k} \right\rceil C_k + \sum_{\tau_k \in dis(i,m) \cup \tau_i} \left\lceil \frac{R_j^n}{T_k} \right\rceil \left\lfloor A_{i,m} \times \frac{\varphi_k}{\varphi_i} \right\rfloor \le D_j$$

$$(9)\quad \forall i \in [1,n]: U + \frac{A_{i,m}}{T_i} + \sum_{\tau_j \in dis(i,m)} \left\lceil A_{i,m} \times \frac{\varphi_j}{\varphi_i} \right\rceil \frac{1}{T_j} \le 1$$

Where dis(i,m) = set of the *(m-1)* minimizing the allowance of task τ_i in the case of execution overrun faults. This set comprises tasks which maximize the quantity $\left\lceil \frac{R_i}{Tj} \right\rceil$.

Proof:

Let τ_i be the faulty task. Let us suppose that there exist at most *m* faulty tasks. In the case of balanced allowance sharing, for a faulty task τ_i, each *(m-1)* faulty tasks τ_j with higher priority than τ_i has an allowance equal to $\left\lfloor A_{i,m} \times \frac{\varphi_j}{\varphi_i} \right\rfloor$ added to its WCET. If we apply the modifications to the theorem 1 replacing the values of C_i by the C_i plus allowance, we will find equations (7) and (8) of lemma 3. Equation (9) of the lemma is the Necessary Condition for the feasibility of the task set, taking into account the allowance of the faulty tasks.

Example:

We consider the parameters of the previous example (see table 1):

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously time t=0.
- The tasks τ_1, τ_2 and τ_3 have decreasing priorities.
- Any task can be faulty *(m=n)*.
- The WCET is the degree of importance of the task: $\varphi_i = \left\lfloor \frac{C_i}{\sum_{j=1}^{n} C_j} \right\rfloor$.

The following table gives the allowance values for each task:

TABLE 4. Margin value - Balanced allowance

Task	ϕ_i	$A_{i,3}$
τ_1	44	133
τ_2	22	66
τ_3	33	100

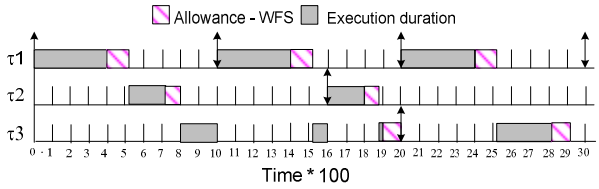

Figure 5. Margin example - balanced allowance

Figure 5 illustrates the scenario in which all the tasks are faulty. We observe that each task meets its deadline as long as the tasks do not consume more than their allowance.

V. LATEST EXECUTION TIME - LET

In order for static allowance to work properly execution overruns must be detected when they occur. This requires the presence of a detector in the system. This does not necessarily mean that available resource will be immediately consumed by a faulty task as a task with a higher priority may be running.

From this detector (WCET overrun), an approach using a budget manager (as in the approaches for management of the aperiodic tasks) is possible. [26] studies various strategies for budget management (differed server, polling server) for the allowance use with WCETs.

These solutions impose on each task a strict use of their allowance and do not allow tasks to recover the unused allowance of the other tasks. Moreover, when a task has an execution time lower than its WCET, it is impossible to recover the used duration to allocate it to other faulty tasks. Concerning the real-time java environment, the specification (RTSJ: Real-Time Specification for Java) proposes the use of handlers for the detection of the execution overrun (*CostOverRunHandler*) and for the detection of the deadline miss (*DeadlineMissHandler*) [27].

However, the execution overrun handler is rarely implemented in real-time systems (for example on *JbedRTOS* of the *Esmertec* Company [28]). In the current minimum implementation of RTSJ, the *CostOverrunHandler* is ignored, but a handler which detects an absolute deadline miss is provided. Thus we propose a new approach with an implicit recovery of resources not used by a task τ_i for all the tasks with lower priority than τ_i. Moreover, this approach solves the problem of the lack of overrun handlers.

It consists of using classical timers which are initialized with their Latest Execution Time (LET), computed according to the assumption that all the allowances have been consumed as well as the WCETs. The LET principles are described in the following subsections. We introduce the static and the dynamic LET. The first can be used for soft real-time systems; the second provides hard real-time guarantees.

A. Static Latest Execution Time

The static LET (called: $LET_{i,m}$) of a task τ_i corresponds to the worst-case response time of τ_i by supposing that all the m faulty tasks consume their allowance.

Lemma 4:

The static LET of a faulty task τ_i can be seen as a relative deadline of execution (a task released at time t must be executed by its absolute LET: t + $LET_{i,m}$), beyond which the risk of deadline miss is very high. Notice that with this approach, we cannot guarantee deadline respect (see figure 7). The static LET should therefore be used in a soft real-time context. It is equal to the worst-case response time of the task when all the faulty tasks consume their allowance. It is calculated as follows:

$$(10)\quad LET_i = C_i + A_{i,m} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{LET_{i,m}}{T_j} \right\rceil C_j + \sum_{\tau_k \in disLet(i,m)} \left\lceil \frac{LET_{i,m}}{T_k} \right\rceil A_{k,m}$$

$$(11)\quad \forall \tau_k \in lp(i) \cup \tau_i : LET_k \le D_k$$

$$(12)\quad \forall i \in [1,n]: U + \sum_{\tau_j \in disLet(i,m) \cup \tau_i} \frac{A_{i,m}}{T_j} \le 1$$

Where disLet (i,m) = set of the *(m-1)* tasks minimizing the allowance of task τ_i in the event of faults (tasks suspected to consume the most free resources, i.e. tasks which maximize the quantity of work $\left\lceil \frac{LET_{i,m}}{T_k} \right\rceil A_{k,m}$).

Proof:

Let us suppose that there exist m faulty tasks. In this case, the WCET of each task is increased with a value equal to the allowance which is computed according to resource sharing mode (fair or balanced). Equation (10) corresponds to the formula of the worst-case response time computation. According to lemma 3, the new response time values must be lower than the tasks deadlines: $R_i \le LET_{i,m} \le D_i$, which respects the formula condition (11). The formula (12) of the lemma is the Necessary Condition for the feasibility of the task set.

Example:

We consider three tasks τ_1, τ_2 and τ_3 scheduled with FP, having decreasing priorities and we suppose at most *m=n=3* faulty tasks. We attribute to each task τi, i=1..3, the allowance values of $A_{i,3}$ and $LET_{i,3}$ (see table 5).

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- Any task can be faulty.

The following table gives the allowance values for each task according to an equal share of the allowances between the *m* faulty tasks:

TABLE 5. Margin value - Static LET

Task	C_i	D_i	T_i	$A_{i,3}$	$LET_{i,3}$
τ_1	1	7	7	1	2
τ_2	2	11	11	1	5
τ_3	4	17	17	1	17

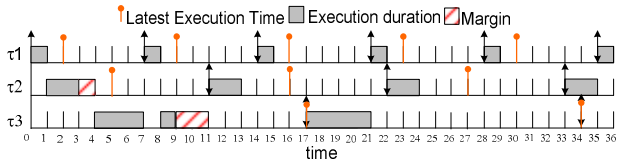

Figure 6. Execution example - Static LET

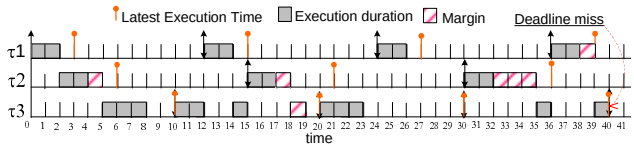

Figure 7. Execution example - Static LET limit

In figure 6, we can see that the first activation of task τ_3 may use several time units when it exceeds its WCET instead of the one granted by the allowance. With the LET mechanism, the unused allowance can be recovered by the faulty tasks.

Advantages of this solution:

- This mechanism improves system performance by the use of the allowance without requiring a cost overrun handler.
- The unused allowance of non faulty tasks can be recovered. Indeed, when a task does not use its allowance, all these saved resources (time CPU) can be used by all the other tasks.

In spite of its good performance, the LET static has a drawback; it does not guarantee the isolation of temporal faults. The computation of the static LET for task τ_i supposes that the tasks with higher priority are present in the system. This represents the worst-case scenario with the use of fixed priority driven schedulers. However, during execution, it is possible that a task with an intermediate priority can consume the allowance of task, with higher priority, not present in the system at that time. This has a knock-on effect on the execution of the task with lower priority (see figure 7). We identify this problem in the following example.

Example:

We consider three tasks τ_1, τ_2 and τ_3 scheduled with FP, having decreasing priorities and we suppose at most $m=n=3$ faulty tasks. We attribute to each task τ_i, i=1..3, the allowance values of $A_{i,3}$ and $LET_{i,3}$ (see table 6).

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- Any task can be faulty.

The following table gives the allowance values for each task according to a fair share of the allowances between the *m* faulty tasks:

TABLE 6. Margin value - Static LET - example 2

Task	C_i	D_i	T_i	$A_{i,3}$	$LET_{i,3}$
τ_1	2	12	12	1	3
τ_2	2	15	15	1	6
τ_3	3	10	10	1	10

Figure 7 shows that there are possible cases where non faulty tasks (task τ_3 in figure 7) exceed their deadlines following faults generated by tasks with higher priority (tasks τ_1 and τ_2 in figure 7).

For example at $t_1 = 30$, the task τ_2 is running and has a LET deadline at $t_2 = 36$, (calculated with the presence of task τ_1) whereas τ_1 will not be activated until t = 36. The fact that τ_2 has the greatest priority in the interval $[t_1, t_2]$ authorizes the task to be executed up to t = 35 (see figure 7). It then takes all available CPU resources, and delays the execution of the task τ_3. At t_2, τ_1 starts its execution and leaves insufficient resources for task τ_3 (only one unit in this example). The failure of the task τ_3 at $t_3 = 40$ is due to an overconsumption of CPU resources by task τ_2, which by consuming its allowance, plus a part of that of τ_1 has indirectly used a part τ_3's allowance.

The problem with the static LET is thus that it is not possible to guarantee for periodic or sporadic tasks the isolation of temporal faults. However, it enables us to anticipate a deadline miss. The majority of real-time systems propose only one detector in the event of a deadline miss.

The problem with this solution is that a task which has missed its deadline may already have had a cascading effect on the other tasks of the system. The correction may come too late. Static LET, with its preemptive correction, is a possible solution for fault prevention. Here is the dynamic LET, which solves the problems brought to our attention by the use of the static LET.

B. *Dynamic Latest Execution Time*

Definition:

Let task τi be a new task released at time t_i. The dynamic Latest Execution Time of all the tasks in $\tau_i \bigcup lp(i)$ released at time t_j and FP scheduled is computed as follows:

(13) $$LET_{i,n}(t_i) = C_i + A_{i,n} + \max(t_i, \max_{\tau_j \in hp^R(i)} (LET_{j,n}(t_j))$$

(14) $$\forall \tau_j \in lp^R(i): \quad LET_{j,n}(t_j) = LET_{j,n}(t_j) + C_i + A_{i,n}$$

Where:

- t_j: is the last request time of task t_j
- $lp^R(i)$: denotes the set of tasks released and still in the system with priority lower than τ_i.
- $hp^R(i)$: denotes the set of tasks released and still in the system with priority higher than τ_i.

Lemma 5:

The dynamic LET guarantees the isolation of temporal faults in the event of WCETs overrun, as long as the faulty tasks do not exceed their dynamic LET.

Proof:

The dynamic LET is updated for all tasks with priority lower than or equal to task τ_i. It takes into account the released tasks with higher priority than τ_i. The dynamic LET

of the task τ_i corresponds to its completion time, after the execution of all the released tasks in *hp(i)*. For the task τ_i, the maximum number of requests for tasks activations with higher priority than τ_i is the maximum possible for the first activation of τ_i in the synchronous scenario. The response time of the first activation of τ_i in the synchronous scenario is the maximum limit of the response time of any instances of the task τ_i in any scenario. Thus we have:

$$\forall t_i \geq 0, LET_{i,n}(t_i) - t_i \leq R_i = C_i + A_{i,n} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \left(C_j + A_{i,n}\right)$$

Where R_i is the response time, computed according to lemma 2, extended to the case $m=n$, including the allowances on WCETs, and respecting the condition: $R_i \leq D_i$.

Thus we find: $\forall t_i \geq 0, LET_{i,n}(t_i) - t_i \leq D_i$

Consequently all the tasks will respect their deadlines provided that they are not run after their dynamic LET. An execution overrun is then isolated as long as the tasks do not exceed their dynamic LET.

Example:

We consider three tasks τ_1, τ_2 and τ_3 with FP scheduling, having decreasing priorities and we suppose at most $m=n=3$ faulty tasks. Each task τ_i, i=1..3, has allowance values of $A_{i,3}$ (see table 7).

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- Any task can be faulty.

The following table gives the allowance values for each task according to fair allowance sharing between the $m=n$ faulty tasks:

TABLE 7. Fair Allowance - Dynamic LET

Task	C_i	D_i	T_i	$A_{i,3}$
τ_1	4	10	10	1
τ_2	2	16	16	1
τ_3	3	20	20	1

At t=0, the dynamic LET of each task is equal to $C_i + A_i$ (values equal to: 5, 3 and 4 respectively). During the activation of τ_2, its LET is updated when a higher priority task is released into the system (task τ_1 in this example). When t=10, τ_1 activates and recalculates the LET of task τ_3, the LET of τ_3 changes from 12 to 17. At t=16, task τ_2 modifies the LET of task τ_3 up to the value of its deadline.

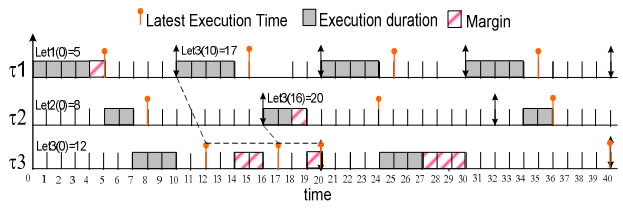

Figure 8. Execution example - Dynamic LET

The CPU resources not used by higher priority tasks can be exploited by task τ_3.

The advantages of the dynamic LET are shown in the following points:

- Error prevention: this is based on the determination of the maximum execution time that can be added to the EWCET of a faulty task without compromising the real-time constraints of all the tasks in the system.
- Failure prevention: our solution makes it possible to anticipate the failures. This leaves us a precious time to make a decision before the deadline miss and to guarantee the isolation of the fault if it occurs.
- Efficient resource management: unused resources can be recovered by faulty tasks automatically without needs of additional calculation (implicitly recovered).
- Independence of platform type: the dynamic LET is a solution, based on the use of timers' handlers, which is easily implemented on all real-times systems.
- Isolation of the faulty task: the dynamic LET guarantees the isolation of temporal faults in the event of EWCETs overrun, as long as the Dynamic LET is not exceeded by the faulty tasks.

C. Latest Execution Time implementation

We show in this section how to implement the allowance use in a real-time system. The following diagram (see figure 9-a) shows the possible execution states for a task.

- *Admission state*: ready to accept a new task τ_i. This state is only used for the first activation of the task. In this state, admission control, based on theorem 1, must be requested to check feasibility. If the new task set τ of n sporadic tasks, including τ_i is declared feasible; all the task deadlines can be met. The system then determines the allowance sets A_j, j=1..n based on lemma 2. Task τ_i can be started and then the scheduler will place the task into a *Ready state*. This means the task is ready to run. If task τ_i cannot be admitted, its execution state changes to *Halted*. The system will be notified of the admission failure. We will not deal in this paper with the treatment of this exception.
- *Running state*: based upon the behavior of the other tasks and threads in the system, a task is scheduled to begin executing, at which point it enters it's *Running state*.
- *Finished state*: while the task is running, the scheduler may preempt it and switch execution to another task. If this occurs, the Running task returns to a *Ready state*. Alternatively, the scheduler may decide to continue to execute the task. When the task has finally completed executing its run method, it enters a *finished state*.
- *Ready state*: in addition to the normal execution states, a task can enter a *Waiting state* if it needs resources which are not currently available. The scheduler may reschedule another task to begin execution. When the resources become available, the original task is return to a *Ready state* where it will be rescheduled.

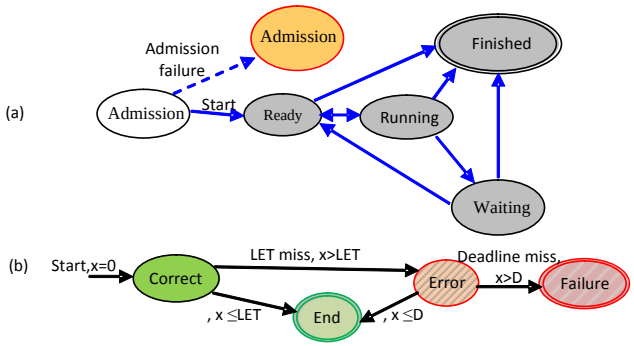

Figure 9. State diagram - Dynamic LET

The diagram b (see figure 9) shows the possible execution states of tasks:

- *Correct state*: if the task completes normally before the LET, its state changes to *end state*. Alternatively, the task becomes faulty and then the scheduler changes the task state to *error state*.
- *Error state*: different strategies can be used to deal with such a situation: stop the faulty task or execute it in the background (when no others tasks require execution). In the experiments carried out in the following section, we have chosen to stop the faulty task to prevent cascading effects on the other tasks. The state of the faulty task is set to *failure state* and an exception is used to inform the system of the failure.

VI. TESTS AND RESULTS

A. *Tests per simulation*

In order to make a comparative study on the robustness of the various solutions, it should be noted that in the following tests, we consider that the faulty tasks have real execution times which can exceed their execution time plus their allowance. We then obtain temporal failures and we compare the capacity of the various algorithms to contain these failures. Indeed, if all the tasks did not consume more than their allowance, we would not observe a temporal failure with the dynamic LET. In these simulations, we consider the following conditions:

- Given a set of 10 periodic tasks: τ_1, τ_2... τ_{10} scheduled with FP.
- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- The task τ_1 will have the greatest priority with very large execution duration.
- The tasks τ_1... τ_{10} will have very short execution duration.
- The tasks τ_1, τ_2... τ_{10} have decreasing priorities.
- The margin value for each task is calculated according to equal resource sharing between tasks.

We assume that every task can cause faults; we simulate this by modifying the task execution time according to the Normal law during each period. We carried out the test 100 times in order to study the failure tendency. In this experiment, we consider the tasks set τ with a processor utilization U=0.848. The goal of these simulations is to compare the performances between the various techniques of allowance management and a mechanism being satisfied to detect the deadline (NTD: nothing to be done in the case of faults).

The values represented on the x-axes indicate the number of simulations carried out (see figures 10, 11, 12 and 13). Each simulation corresponds to an execution scenario. The duration of the test is higher than the *lcm* of the periods of the tasks (14000 units of time).

The allowance value $A_{i,10}$, $\forall i \in [1,10]$, is equal to 22 time units. The following table (see table 8) gives the static LET values for each task according to an equal share of the allowances between tasks:

TABLE 8. Simulation example - 10 tasks

Task	C_i	D_i	T_i	P_i	$LET_{i,10}$
τ_1	120	200	200	10	122
τ_2	20	300	300	9	144
τ_3	20	300	300	8	166
τ_4	20	300	300	7	188
τ_5	5	500	500	6	195
τ_6	5	500	500	5	390
τ_7	5	500	500	4	397
τ_8	5	800	800	3	547
τ_9	5	800	800	2	554
τ_{10}	5	800	800	1	561

A task has an *indirect failure* when it does not exceed its WCET but misses its deadline. The deadline miss is then due when other faulty tasks exceed their WCET. This can only happen when no allowance mechanism is used. Without the allowance mechanism, we observe the multiplication of indirect failures.

Figure 10 shows that in the case of task fault, nothing to be done i.e. let the faulty task continue its execution, does not guarantee the task against a temporal failure but exposes the other tasks in the system to successive cascading failures. Only one task fault can cause several failures on the level of the other tasks. The use of allowance preserves the system from cascading effects. We remark also that the static allowance is less powerful than the LET techniques.

When the faulty tasks consume their entire allowance or if a task reaches it LET deadline, we put the execution of these tasks in background. The results obtained in this case (see figure 11) show the benefits of the solutions using the allowance. The performances (reduction of failures) of the solutions are always in the same order: the static LET gives the best results, followed by the dynamic LET and finally the equitable allowance.

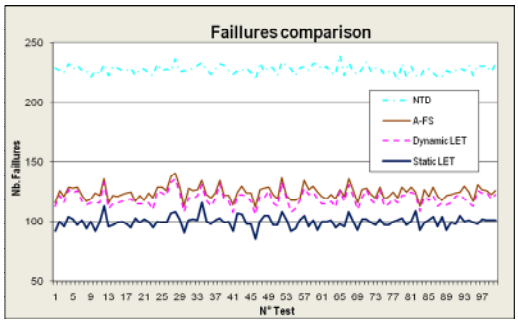

Figure 10. Failures comparison - stop after margin exceeds

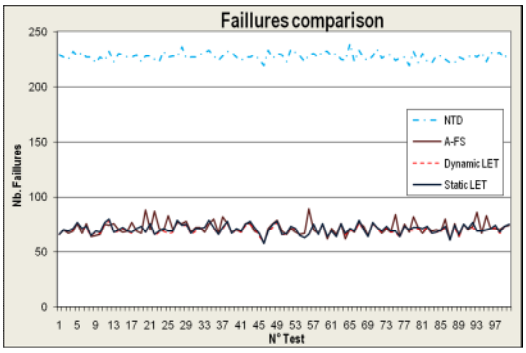

Figure 11. Failures comparison - background after margin exceeds

In the following simulation test, we increase the number of tasks in the system. We consider a set of 20 periodic tasks τ_1,.., τ_{20}. The first task will have the greatest priority. The 19 remaining tasks will have very short execution duration with decreasing priorities.

We assume that every task can cause faults, which is made possible by modifying the task execution time according to the Normal law during each period. We remake the test 100 times in order to study the failure tendency. In this experimentation, the processor utilization is equal to U=0.7766. The goal of this simulation is to study the effect of the increase on the number of tasks in the failure rate. The most important remark on this simulation (see figure 12) resides in the fact that the failure rate is considerably reduced if we use the allowance compared to the solution nothing to be done (NTD).

The increase in the number of tasks in the system amplifies the risk of the indirect failures. The results of the tests on the simulator show that the use of the allowance, whatever the adopted solution, makes it possible to reduce the number of failures in the system. We focused our tests on the LET mechanism which can be installed on a real-time platform. The results of simulations confirmed that the use of the static LET, in spite of some indirect failures, makes it possible to reduce the number of failures.

B. Practical Tests

The static LET mechanism has been tested on RedHat Linux 8 with kernel 2.4.18 and on the TimeSys RT/Linux 4 [29].

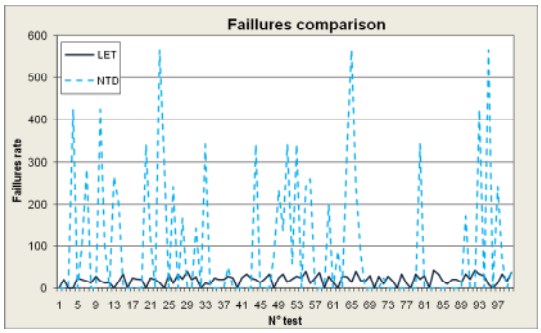

Figure 12. Failures comparison - LET and NTD

In this test, we compare the correction rates obtained by the static LET mechanism by simulation and on real platforms.

With the objective of to comparing like with like, we used the same example of tasks tested on the simulator (see table 8). The test was carried out on the platform described in the preceding point. In our test (see figure 13), we obtained on average, a 17% less good results of fault correction by using practical LET mechanism compared to the results obtained by simulation. In fact, we obtained a correction rate of faults, equal to 84% during the tests on the simulator and 67% during the tests on a real-time platform. We suspect the garbage collector of Java to be partly responsible for this degradation. The practical tests confirmed the benefits of the use of the allowance concept.

In the last experiment, we consider the same set of tasks; the difference resides in the execution duration of the task τ_1 which varies between 900 and 1500. This corresponds to a utilization ratio of the processor which varies between 0.533 and 0.883.

The tests were carried out in a synchronous scenario for a length of time equal to 10 times the *lcm* of the tasks (120000 time units). For each test, we take the average of 10 executions, which corresponds approximately to twenty minutes per test.

The test objective is to analyze, on a real-time platform *Jtime (TimeSys)*, the influence of the processor utilization rate on the performances of the static LET mechanism (see figure 14).

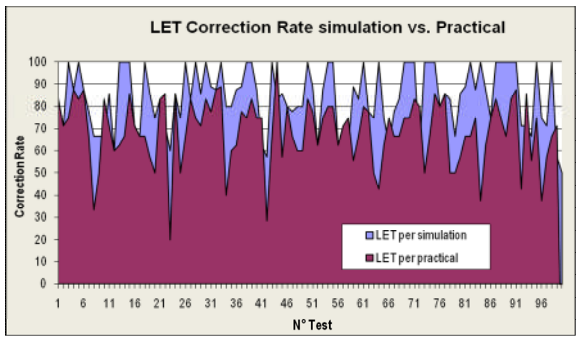

Figure 13. Correction rate - LET simulation vs. LET practice

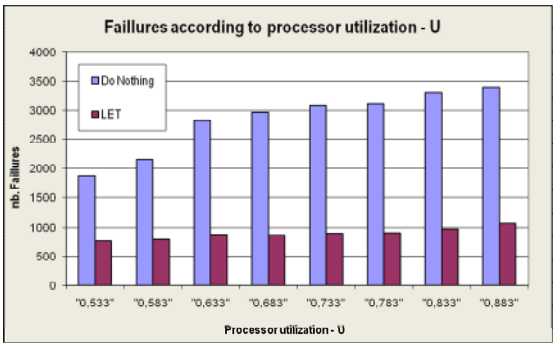

Figure 14. Failures - utilization rate of the processor

We can notice that the number of failures is more important when the static LET is used, which is not the case with solution *NTD*. The tests of the solutions on the simulator show the improvements made by the use of the allowance. The decrease of the performances in practice is explained by the system cost of the allowance mechanism.

VII. CONCLUSION

In this article we have considered the problem of fault prevention in a real-time system. The faults correspond to WCETs overruns resulting from the use of estimated WCETs. This enables us to consider an open architecture, independent of the operating system. The use of a virtual machine such as Java allows this independence. In this context, the worst case execution times are not easily determinable by a static analysis due to portability issues. The design of real-time system then requires techniques to tolerate an uncertainty on the WCET. The solution that we propose is based on the determination of the acceptable deviations on the WCETs which is called the allowance. The allowance enables us to reduce the failure rate of the real-time tasks and to make the system more robust in the event of temporal fault. This mechanism permits the recovery of free CPU resources in the event of temporal fault. We propose equations to compute the allowance in the case of a fixed priority scheduling. We proposed an implementation based on the computation of the Latest Execution Time (LET) preserving the timeliness constraints of the tasks. Simulations carried out and the studies on a real platform enabled us to conclude that this mechanism is interesting to limit the impact of temporal faults of the WCETs overruns by increasing the rate of their correction. Furthermore, the LET provides an implicit recovery of the unused allowance.

ACKNOWLEDGMENT

The authors gratefully acknowledge Siân Cronin at ESIGETEL for providing linguistic help.

REFERENCES

[1] L. Bougueroua, L. George, S. Midonnet, Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled FP and EDF. The second International Conference on Systems – ICONS 2007.Sainte-Luce, Martinique, 22-28 April 2007.

[2] Liu, C.L., Layland, J.W., Scheduling Algorithms for multiprogramming in a Hard Real Time Environment; Journal of Association for Computing Machinery, Vol. 20, N° 1, (Jan 1973).

[3] Joseph, M., Pandya, P., Finding reponse times in a real-time system ; BCS Computer Journal, 29(5), (1986) 390-395.

[4] Lehoczky, J.P., Fixed priority scheduling of periodic task sets with arbitrary deadlines; Proc. 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Floride, USA, (Dec 1990) 201-209.

[5] Puaut, I., Méthodes de calcul de WCET (Worst-Case Execution Time) Etat de l'art; 4ème édition Ecole d'été temps-réel(ECR), Nancy, (Sep 2005) 165-175.

[6] Tindell, K., Burns, A., Wellings, A. J., An extendible approach for analyzing fixed priority hard real time tasks ; Real-Time Systems, Vol. 6, N°2, (Mar 1994)133-151.

[7] L. Bougueroua, L. George and S. Midonnet, An execution overrun management mechanism for the temporal robustness of java real-time systems. The 4th workshop on Java technologies for Real-Time and Embedded Systems (JTRES) 11-13 October 2006, Paris.

[8] Locke, C.J., Best effort decision making for real-time scheduling; PhD thesis, Computer science department, Carnegie-Mellon university, (1986).

[9] Koren, G., Shasha, D., D-over: An Optimal On-line Scheduling Algorithm for over loaded real-time system; technique report 138, INRIA, (Feb 1992).

[10] Buttazo, G., Stankovic, J.A., RED: A Robust Earliest Deadline Scheduling; 3rd international Workshop on responsive Computing, (Sept 1993).

[11] Buttazzo, G., Lipari, G., Abeni, L., Elastic Task Model for Adaptive Rate Control; Proc. IEEE Real-Time Systems Symposium, Madrid, Spain, (Dec 1998) 286-295.

[12] Buttazzo, G., Lipari, G., Caccamo, M., Abeni, L., Elastic Scheduling for Flexible Workload Management; IEEE Transactions on Computers, Vol. 51, No. 3, (Mar 2002) 289-302.

[13] Bini, E., Di Natale, M., Buttazzo, G., Sensitivity Analysis for Fixed-Priority Real-Time Systems; Proc. 18th Euromicro Conference on Real-Time Systems, ECRTS'06, (2006).

[14] Racu, R., Jersak, M., Ernst, R., Applying sensitivity analysis in real-time distributed systems; Proc. 11th Real-Time and Embedded Technology and Applications - RTAS'05, (2005).

[15] Racu, R., Hamann, A., Ernst, R., A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems; Proc. 18th Euromicro conference on realtime systems - ECRTS'06, (2006).

[16] Bini, E., Buttazzo, G., Schedulability Analysis of Periodic Fixed Priority Systems; IEEE Transactions On Computers, Vol. 53, No. 11, (Nov 2004).

[17] Lehoczky, J. P., Ramos-Thuel, S., An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks Fixed-Priority Preemptive systems; Proc. Real-Time System Symposium, (Dec 1992) 110-123.

[18] Davis, R.I. Scheduling Slack Time in Fixed Priority Pre-emptive Systems; Proc. 14th Real-Time Systems Symposium, (1993) 222-231.

[19] Spuri, M., Buttazzo, G., Scheduling aperiodic tasks in dynamic priority systems; Journal of real time systems, vol. 10, (1996) 179-210.

[20] Caccamo, M., Lipari, G., Buttazzo, G., Sharing resources among periodic and aperiodic taskc with dynamic deadlines; Proc. 20th IEEE Real Time System Symposium, (1999).

[21] Kuo, T.W., Mok, A.K., Load Adjustment in Adaptive Real-time Systems; Proc. 12th IEEE Real-Time Systems Symposium, (Dec 1991).

[22] Nakajima, T., Tezuka, H., A Continuous Media Application Supporting Dynamic QoS Control on Real-Time Mach; Proc. ACM Multimedia, (1994).

[23] Seto, D., Lehoczky, J.P., Sha, L., Shin, K.G, On Task Schedulability in Real-Time Control Systems; Proc. IEEE Real-Time Systems Symposium, (Dec 1997).

[24] Hamdaoui, M., Ramanathan, P., A dynamic priority assignment technique for streams with (m,k)-firm deadlines; IEEE Transactions on Computers, vol. 44(12), (1995) 1443-1451.

[25] Bernat, G., Burns, A.A.L., Weakly Hard Real-Time Systems; IEEE Transactions on Computers, (2001).

[26] Bougueroua L., Conception de systèmes temps réel déterministe en environnement incertain; PhD thesis, vol 1, n° 2007PA120004, SI (Mar 2007).

[27] Bollela, G., Gosling, Brosgol, Dibble, Furr, Hardin and Trunbull, The Real-Time Specification for Java; Addison Wesley, 1st edition, (2000).

[28] Esmertec, jbedRTOS: Java Bulding Embedded Operating System; http://www.esigetel.fr/images/stories/Recherche/SITR/spec-jbed-rtos.pdf.

[29] TimeSys, TimeSys' real-time Java Virtual Machine (JVM); http://www.timesys.com.