Maximizing the execution rate of low-criticality tasks in mixed criticality systems

Mathieu Jan and Lilia Zaourar
CEA, LIST
Embedded Real Time Systems Laboratory
F-91191 Gif-sur-Yvette, France
Email: Firstname.Lastname@cea.fr

Maurice Pitel
Schneider Electric Industries
37, quai Paul Louis Merlin
F-38050 Grenoble, France
Email: Maurice.Pitel@schneider-electric.com

***Abstract*—Industrial fields must build at the most competitive price real-time systems made of an increasing number of functionalities. This can be achieved by hosting high-criticality tasks as well as consumer real-time low-criticality tasks on a same chip. The design of such Mixed-Criticality (MC) systems requires the use of an appropriate task model and a specific scheduling strategy. In this work, inspired by the existing elastic task model, we introduce stretching factors as a way for the low-criticality tasks to reduce their utilization, as well as a level of importance in order to define an order for applying these stretching factors. At run-time, the slack time generated by both the over-provisioned high-criticality and the low-criticality tasks is used to maximize the execution rate of the low-criticality tasks. We also show how to integrate this approach in the Time-Triggered paradigm (TT), in particular its impact on the data visibility principle between the low-criticality tasks when they have been stretched.**

I. Introduction

Traditionally, industrial systems use a dedicated (possibly multiprocessor) chip for executing a set of real-time tasks with a same level of criticality. When such tasks are safety-critical, high margins are taken on their Worst-Case Execution Time (WCET). This leads to the specification of high allocated budgets of time for such high-criticality tasks. Besides, the probability that the WCET of a set of high-criticality tasks occur simultaneously is very low. However, the schedulability demonstration of safety-critical systems must be performed in the worst-case situation, due to certification constraints. This therefore leads to a huge over-sizing of the CPU resources that are needed compared to what is really used, in average, while the system is running. This practice becomes incompatible with the current trend of tighter economical constraints of various industrial domains, such as the automotive or energy distribution fields. Therefore, there is a need to use these generally unused processing capabilities for executing the low-criticality tasks.

This type of system where both the low and high-criticality tasks are allocated on a single chip are called Mixed Criticality (MC) systems. Note that in general, two criticality levels are generally considered in existing work on this topic, as well as in the remainder of this paper. To fulfil certification requirements, mainly from the avionic domain, and enable an efficient scheduling of the high and low-criticality tasks, task models and scheduling algorithms addressing MC systems have recently been proposed ([18], [10], [3]). The goal is to increase the schedulability of the low-criticality tasks, while still guaranteeing in the worst-case scenario the schedulability of the high-criticality tasks. Classical scheduling algorithms can indeed lead to well-known priority inversion problems, as they are unaware of the criticality parameter of the tasks. Most existing work defines two modes for a MC system and each task must specify its criticality level. The MC system starts in the low-criticality mode where all tasks are executed. However, when a deadline is missed the MC system switches to the high-criticality mode, where only the high-criticality tasks are executed. The low-criticality tasks are simply dropped. This degrades the level of service provided by MC systems. Besides, processing capabilities are wasted.

Maximizing the level of service provided by a set of real-time tasks for controlling a physical system has been the subject of numerous work in the domain of real-time control ([7], [1]). When a perturbation/event suddenly affects the controlled system, the higher the sampling period is, the better the reactivity of the system is. However, the higher the processing load is. This requirement has therefore led to the proposal of a more flexible task model, called the elastic task model [5], where the periodicities of tasks can take a range of values. This task model, combined with an appropriate scheduling algorithm, avoids the need of dropping tasks when a deadline is missed. The periodicity of some tasks must simply be increased appropriately. Recently, the elastic task model has been studied in the context of MC systems for the low-criticality tasks [16]. The goal is to deal with the service abrupt problem of dropping the low-criticality tasks when the MC system switches to the high-criticality mode.

This work also proposes a solution to this problem. In this work, we also adapt the elastic task model for solving the following problem: *how to maximize the utilization of the processing capabilities of an architecture, in which high- and low-criticality tasks are schedulable taken separately, while the sum is not?* One goal is therefore to avoid dropping the low-criticality tasks. Off-line, we use a linear programming approach to compute the different *stretching factors* that must be applied on the periodicity of the low-criticality tasks, so that the schedulability of the high-criticality tasks are guaranteed. On-line, we bet on the availability of slack time generated by high and low-criticality tasks. No stretching factors are therefore applied in order to execute the low-criticality tasks at their fastest rates. However, when a deadline is going to be missed by a low-criticality task, its deadline is stretched up to point that prevents the deadline miss.

Another contribution of this work is to show *how our proposed approach to deal with the low-criticality tasks can be used within the TT paradigm* [13], used to build safety-critical real-time systems. Using this paradigm, the triggering of activities, that correspond to task releases and deadlines, are specified by the application designer. At these dates, data exchanged between the tasks are made visible. Stretching a task introduces a modification in these statically defined triggering points as well as in the visibility date of the produced data. Therefore, when stretching a low-criticality task, the deadline of some other low-criticality tasks must also be postponed in order to keep the system temporally consistent.

The remainder of this paper is as follows. Section II describes the related work in the scheduling algorithms for MC systems as well as their practical implementation. Then Section III formulates the problem, while Section IV presents the proposed task model and the on-line decision algorithm used within the TT paradigm to stretch the low-criticality tasks. Finally, Section V evaluates the proposed solution and Section VI concludes.

II. Related work

In [18], Vestal introduces the most used task model for specifying MC real-time systems, therefore sometimes called the MC task model. The classical periodic task model is extended with two WCET values, named $C_i(LO)$ and $C_i(HI)$, and a criticality level, which can be either low or high. As stated previously, we consider that a MC system has only two modes of criticality: high and low. $C_i(LO)$ is the maximum allowed execution time for the task in the low-criticality mode, while $C_i(HI)$ is the maximum allowed execution time for the task in the high-criticality mode ($C_i(LO) < C_i(HI)$). The rationale for specifying two WCETs is that the higher the criticality level, the more conservative the verification process and hence the greater will be the value of C_i.

In the context of digital control systems, early work has focused on an off-line analysis to compute the tasks frequencies that minimize the cost of the control and tracking error. This has lead to the definition of the so-called elastic task model [5] to increase the flexibility of the periodic task model. In addition to its execution time, each task is characterized by: a nominal period T_{i_0}, a minimum period $T_{i_{min}}$, a maximum period $T_{i_{max}}$ and an elastic coefficient $e_i \geq 0$. The elastic coefficient specifies the flexibility of the task to vary its utilization within the range of possible periods. [17] also presents a task model which allows to jointly optimize the used computing resources and the control performance of a computer-based instrumentation and control system. Each control task is able to trigger itself: the timing constraints are dynamically adjusted based on the whether the controlled system is stable or subject to perturbations. Finally, [14] proposes to integrate in the task model a parameter to specify a minimal distance between two consecutive skips of instances of a task, that is between two deadline misses.

In this area, our closest related work is [16], where the elastic task model is applied in the context of MC systems for specifying the behavior of the low-criticality tasks. However, this task model does not allow to specify an order between the low-criticality tasks, as how a set of elastic tasks is compressed depends on their utilization.

Appropriate scheduling algorithms must then be defined to support task models used within MC systems, in particular the criticality-level parameter. The goal is to ensure the schedulability in the worst-case scenario of the high-criticality tasks, while improving the schedulability of the low-criticality tasks. This is possible thanks to the introduction of $C_i(LO)$ for each high-criticality task. Several approaches have been followed: using either a fixed priority algorithm [18], a zero-slack algorithm on top a fixed priority scheduler [10], the assignment of virtual and smaller deadlines for the high-criticality tasks (EDF-VD for EDF-Virtual Deadlines) [3] or the definition or early release points for accelerating the execution rate of the low-criticality tasks [16] (ER-EDF for Early Release EDF). This last decision algorithm for releasing earlier or not the low-criticality tasks is the closest related work to ours. However, it takes the opposite approach to execute the tasks at their fastest execution rate: it computes a new (early) release point when the task finishes, while we extend the deadline of the task when it is going to miss its deadline.

On the implementation side of MC systems, [11] presents a first implementation of a MC hierarchical scheduling framework on a multi-core system, that addresses the criticality levels of the avionic domain. It is based on LITMUS [6], an extension to Linux that was developed to study in practice real-time multiprocessor schedulers. The focus is put on optimizing the implementation of the proposed hierarchical schedulers for MC systems, by using fine-grained locking mechanisms to reduce scheduling overheads. In our work, we are also interested in the integration of MC scheduling into real-time operating systems, but more specifically in the TT paradigm. Finally, [2] presents an implementation in ADA of mode changes in MC systems, from low-criticality to high-criticality as well as the opposite. The authors consider the problem of when returning to the original ordering, as doing it prematurely can cause a high-criticality task to miss its deadline. However, none of these works have considered the impact on data exchanges between the tasks when switching to another mode of execution.

III. Problem formulation

A. Motivation

Within an embedded system, the tasks can be either critical, less (or even non-) critical. Let us for instance, take a protection relay used within medium voltage electrical networks. The safety-function of the software part of protection relays is to first detect any faults within the supervised power network, and then ask the tripping of the circuit breakers in order to isolate the faulty portion of the network. More details on the required set of high-criticality tasks needed to achieve this functionality can be found in [12]. As any safety-related system, protection relays have to comply with a Safety Integrated Level (SIL), as defined by the *IEC 61508* standard. This standard requires that the schedulability demonstration of the high-criticality tasks must be performed. To take into account worst-case situations, high margins are taken on the WCET of these high-criticality tasks. This leads to an over sizing of the required CPU power, compared to what is required in average while the system is running.

On the other hand, there is a need to embed additional less (or non) safety functionalities, such as displaying information,

optimizing the distribution of energy to the current need, etc., in order to distinguish the product from competitors. This leads to the requirement of executing applications with different levels of criticality on the same system. Besides, assuming the WCET for the high-criticality tasks and executing the low-criticality tasks in the remaining CPU power is no longer a viable approach: too much processing power is wasted. Such an approach is no longer compatible with current economical constraints that push for minimizing the CPU power. Such products must take advantage of the slack time generated by the tasks, when they are not using their WCET, in order to execute the low-criticality tasks. Therefore, the problem we address is to *enable the design of MC systems, where taken separately the high and the low criticality tasks are schedulable but the union is not.*

B. Approach

Our goal in this work is to allow the low-criticality tasks to use the slack time and, when a deadline is going to be missed by a low-criticality task, to relax its temporal constraints. To this end, we consider the deadline of the low-criticality task as a flexible parameter that can be extended. This flexibility is handled through a so-called *stretching period factor* (or simply stretching) that we introduce in the classical implicit-deadline periodic task model. A stretching factor is a value by which the periodicity of a low-criticality task can be multiplied. Stretching factors should be specified off-line by the application designer so that a bound is defined. They can be a set of values or a range of values and a same value of stretching factor can be given to several tasks. Besides, we assign to each low-criticality task an importance level. This importance level denotes an order for choosing which low-criticality tasks should be stretched first. Our task model is simpler than the elastic task model [5], as it does not contain a minimum period and an elastic coefficient, linked to the utilization of the task.

Off-line, the stretching factors are used in the schedulability analysis of a task set, made of both the high and the low-criticality tasks. This guarantees that the stretching factors used on-line cannot lead to a situation where a deadline is missed. Besides our task model introduces a way to specify an order between the low-criticality tasks for applying the stretching factors. This gives more control to the application designer to specify a set of possible temporal behavior for the low-criticality tasks. Furthermore, the formal demonstration of the fulfilment of end-to-end constraints for these tasks is therefore still possible. This was main requirement that lead us to the definition of our task model.

IV. Using stretching factors

A. Task model and notations

Let $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ be a set of n independent, synchronous, preemptible and periodic tasks. The set Γ is partitioned into two disjoint subsets: Γ_{ct} the set of the high-criticality tasks, made of n_{high} tasks, and Γ_{nct} the set of the low-criticality tasks, made of n_{low} tasks. We therefore have $n = n_{low} + n_{high}$.

Γ is handled using the classical implicit-deadline periodic task model. Each task $\tau_i \in \Gamma$ has the following temporal parameters $\tau_i = (P_i, C_i, D_i)$. P_i is the period of the task, C_i is its WCET, D_i its deadline and we have $P_i = D_i$. Furthermore, each non-critical task $\tau_i \in \Gamma_{nct}$ has two additional parameters V_i and $S_{i,max}$, where V_i is the importance level of the task and $S_{i,max}$ is the maximum stretching factor that can be applied to P_i. We denote by S_i an actual value for the stretching factor of task τ_i and we have: $1 \leq S_i \leq S_{i,max}$. We denote by $P_{i,max}$ the period when $S_{i,max}$ is applied and corresponds to the maximum period the task can have. The higher the value of V_i, the higher is the importance level of the low-criticality task. But the later its stretching factor will be increased first when a deadline is going to be missed.

The processor utilization of τ_i, a low-criticality task, is: $u_i = \frac{C_i}{S_i \times P_i}$. The lower bound of u_i (noted $u_{i_{min}}$) is reached when the maximum stretch factor ($S_{i,max}$) is applied to τ_i, while its upper bound (noted $u_{i_{max}}$) is reached when its nominal period is used. The processor utilization of a high-criticality task is $u_i = \frac{C_i}{P_i}$. We note respectively U_{low} and U_{high} the total utilization of the low and the high-criticality tasks: $U_{low} = \sum_{\tau_i \in \Gamma_{nct}} \frac{C_i}{S_i \times P_i}$ and $U_{high} = \sum_{\tau_i \in \Gamma_{ct}} \frac{C_i}{P_i}$. The total utilization of the system is noted U and is equals to $U_{low} + U_{high}$. Finally, let m be the number of processors.

B. Off-line CPU maximization

Off-line, we compute for each low-criticality task the minimum stretching factor ($S_{i,min}$) that must be used so that no deadline is missed, assuming that each task uses it WCET. Therefore, we have $S_{i,min} \leq S_{i,max}$. $S_{i,min}$ is a feedback to the application designer on the worst-case temporal behavior the low-criticality task may use on-line. As we focus on the low-criticality tasks only, we can therefore remove from U the utilization generated by the high-criticality tasks. We denote by U_r this remaining CPU capacity, which is equal to $m - U_{high}$. A first constraint therefore expresses the fact that U_r upper bounds the utilization that can generate the low-criticality tasks:

$$U_{low} \leq U_r \Leftrightarrow \sum_{i \in \Gamma_{nct}} \frac{C_i}{S_i \times P_i} \leq U_r \tag{1}$$

Then, a second constraint therefore expresses the fact that the utilization value of a low-criticality task is bounded, as seen in the previous section.

$$u_{i_{min}} \leq \frac{C_i}{S_i \times P_i} \leq u_{i_{max}} \tag{2}$$

Our objective is to maximize the utilization of the resources, while stretching the less important low-criticality tasks first, that is:

$$MaxZ = \sum_{i \in \Gamma_{nct}} V_i \times u_i = \sum_{i \in \Gamma_{nct}} V_i \times \frac{C_i}{S_i \times P_i} \tag{3}$$

By applying the following change of variable: $x_i = \frac{1}{S_i}, \forall \tau_i \in \Gamma_{nct}$, we obtain the following linear program:

$$\text{(LP-1)} \quad \begin{cases} \text{Max :} & \sum_{i \in \Gamma_{nct}} V_i \times x_i \times \frac{C_i}{P_i} \\ \text{s.t :} & \sum_{i \in \Gamma_{nct}} x_i \times \frac{C_i}{P_i} \leq U_r \\ & u_{i_{min}} \leq x_i \times \frac{C_i}{P_i} \leq u_{i_{max}}, \forall \tau_i \in \Gamma_{nct} \end{cases} \tag{4}$$

Algorithm 1 Decision algorithm for setting the stretching factors of low-criticality tasks.

Require: $\tau_i \in \Gamma_{nct_k}$ and the current time t
1: $S_i \leftarrow ComputeStretching(\tau_i, t, D_i, S_i)$;
2: **if** $S_i \geq S_{i,min}$ **then** Stop τ_i and log the error; **end if**
3: $D_i \leftarrow S_i * P_i$;
4: UpdateReady(τ_i);
5: Call the scheduler;

The total number of decision variables of the linear program LP-1 is equal to n_{low}, the number of low-criticality tasks. The total number of constraints is equal to $2n_{low} + 1$. Indeed, there are two constraints for each decision variable in order to express the upper and lower bounds, plus the constraint of remaining CPU capacity. LP-1 can thus be solved in polynomial time.

C. On-line decision algorithm

We now focus on the on-line decision algorithm that sets the stretching factors. We assume that the high-criticality tasks have a higher priority over the low-criticality tasks. Clearly, this increases the number of times the low-criticality tasks are preempted. As in [19], a wrapper-task mechanism for the slack time can be used to avoid such situations.

Our decision algorithm is called when the system detects that one of the low-criticality task is going to miss its deadline. This is the beginning of an overloaded situation for the low-criticality tasks, during which other low-criticality tasks may reach a point where they are also going to miss or have already missed their deadlines. This last case can occur when the system reschedules the low-criticality tasks after some high-criticality tasks have been executed, while in an overloaded situation. Therefore, our decision algorithm is also called before scheduling the low-criticality tasks.

When our decision algorithm is called, we assume that the most important low-criticality task is being executed. To achieve this, the low-criticality tasks could be scheduled using a hierarchical approach: first using the importance level and then using EDF within a given importance level. Another solution would be to use EDF-VD [3] to favour important low-criticality tasks that have far away deadlines over less important low-criticality tasks that have closer deadlines. More generally, our decision algorithm can be combined with any scheduling algorithm if the aforementioned hypotheses are fulfilled. Finally, when a low-criticality task finishes, if it has stretched, then its stretching factor is reset to 1. Note that other strategies could be defined, such as a fixed-timeout strategy before resetting the stretching factor in order to avoid additional calls to our decision algorithm and the associated system calls overhead, if the overloaded situation goes on. The definition and the evaluation of such strategies is currently left as future work.

Algorithm 1 presents the major steps of our decision algorithm when the low-criticality task τ_i is going to miss its deadline. The function *ComputeStretching*, used to compute the stretching factor of τ_i (line 1), can be implemented using different strategies. An optimistic strategy would be to choose a first reasonable value that leads to set a deadline in the future ($S_i * P_i > t$). By reasonable, we mean that the task has a good chance, according to the distribution of its execution time, to finish its execution before its new deadline (set at line 5). Other strategies are possible that might reduce the number of times the stretched deadline is reached.

Algorithm 2 Additionnal steps in the decision algorithm when integrated in the TT paradigm, compared to algorithm 1.

Require: Γ_{nct_k} with $\tau_i \in \Gamma_{nct_k}$
1: **for all** $\tau_j \in \Gamma_k \neq \tau_i$ **do**
2: **if** τ_j is ready **then** RemoveFromReady(τ_j);
3: **else** RemoveFromSleeping(τ_j); **end if**
4: **if** $S_i \geq S_{j,min}$ **then** Stop Γ_{nct_k}, log the error; **end if**
5: $D_j \leftarrow P_j + (D_j - P_i)$;
6: **if** τ_j is finished **then** SetFlag(Stretched); **end if**
7: InsertReady(τ_j);
8: **end for**

D. Using stretching factors within the TT paradigm

Applying stretching factors to the low-criticality tasks within the TT paradigm raises an issue. The hypothesis of independent tasks that can be made at a system level, does not hold any more at the application level. In the TT paradigm, to each produced data is indeed associated a timestamp: the deadline of the producer task. Then, a task may only use data whose timestamps are equals or inferior to its release date, leading to a deterministic execution with demonstrable end-to-end temporal constraints. Therefore, the visibility date of data produced by a low-criticality task changes when the task is stretched. However, the low-criticality tasks have defined triggering points (release date and deadlines) assuming the non-stretched temporal behavior. This leads to an inconsistency between the expected temporal behavior of the tasks, if the stretching factor of a single task is modified. In addition, this inconsistency has an impact on the various worst-case end-to-end temporal behaviors that can fulfil the low-criticality tasks.

To solve this issue, we assume that the application designer can gather in groups the low-criticality tasks that must be kept temporally consistent between them. Therefore, Γ_{nct} is made of a set of groups, noted Γ_{nct_k}. A low-criticality task τ_i can only be inside a single group and the multiplicity of a group can range between 1 to n_{low}. Off-line, our task model must be adapted so that the importance level and the stretching-factor parameter is applied to the group level. Therefore, the only modification to the linear program LP-1 is to consider the utilization of each group Γ_{nct_k} and not the utilization of each task. The utilization of a group Γ_{nct_k} is defined as $\frac{1}{S_k} \times \sum_{\tau_i \in \Gamma_{nct_k}} \frac{C_i}{P_i}$.

Algorithm 2 then presents the additional steps that must be done before calling the scheduler (line 5 in algorithm 1) to stretch the other tasks within a group Γ_{nct_k}, in which task τ_i is going to miss its deadline. Two cases must be considered when recomputing the deadline of a task τ_j: either it has been released but is not finished (line 2) or it is already finished (line 3). In this last case, the visibility date of already produced data must be changed and the part of the task that sets the visibility date must therefore be re-executed. This is signaled by setting the flag *Stretched* (line 6) and setting back the task in the set

of tasks waiting to be executed (line 7) using the *InsertReady* function. This function is also called in the other case to sort the tasks according to the scheduling policy, as their deadlines have changed. Computing the new deadlines simply consists of translating the offsets triggering points of Γ_{nct_k} with the initial deadline to the stretched deadline of τ_i (line 5). The *Stretched* flag is also used by the tasks, from Γ_{nct_l} with $l \neq k$, to avoid updating the values of data they use while the visibility dates of the stretched tasks are recomputed.

Note that the low-criticality tasks cannot use at a given timestamps different values of the same data (i.e., a data inconsistency). Our algorithm is indeed called at the visibility date of data, produced by the initial stretched task. Therefore by definition, these data are not yet visible by the other tasks. While on a uniprocessor system implementing this is straightforward, on a multiprocessor this can be achieved using a spin-lock for the management of scheduling structures. Also note that the stretching factors of the other groups Γ_{nct_l} (with $l \neq k$) do not need to be modified. These groups are indeed by definition less important, so their stretching factors will be computed when the hierarchical scheduler will schedule them.

V. Preliminary evaluations

The goal of our simulation experiments is to validate our proposed task model in its ability to specify an order for choosing which tasks should stretch. We therefore focus on the off-line part of our proposal.

A. Simulation environment

We generate random task sets for both the low-criticality and the high-criticality tasks. The utilization of each task is computed randomly between 0.01 and 0.99 with a uniform distribution using the UUniFast-Discard algorithm from Davis and Burns [9], an extended version of the UUniFast algorithm from Bini and Buttazzo [4] targeting multiprocessor systems. We bound the range of possible periods between $10ms$ and $100ms$ and use a uniform distribution when assigning P_i. The task sets with a hyper-period larger than $10s$ are rejected to remain in a realistic bound of typical industrial systems. Each task is assigned a boolean value that determines whether it is a high-criticality or a low-criticality task. This step is repeated until the number of high-criticality tasks and their total utilization U_{high} reaches a value of 50%. Then, for each low-criticality task a value between 10 and 100 is randomly generated. This value represents the importance of this task in the system (in practice less importance levels would be used). Finally, we assume for each low-criticality task that $S_{i,max} = 2$.

For the evaluation, three task sets are generated. In order to get as close as possible to expected MC systems, we assume that each task set is made of 20% of high-criticality tasks. The first task set (TS_1) is made of 50 tasks with 5 high-criticality tasks. The second task set (TS_2) is made of 60 tasks with 12 high-criticality tasks. Finally, the third task set (TS_3) is made of 70 tasks with 14 high-criticality tasks. For each set, we generated the tasks three times so that the initial total CPU utilization is 100%, 125% and 150% on a 2 processors system.

TABLE I. Obtained $S_{i,min}$ according to metrics *Aver*, *Aver25+* and *Aver75* for the task sets TS_1, TS_2 and TS_3 respectively.

U	*Aver*	*Aver25+*	*Aver75*	*Aver* w/o V_i
125	1.69/1.36/1.59	1/1/1	1.94/1.48/1.79	1.65/1.3/1.48
150	1.86/1.65/1.83	1.5/1/1.37	2/1.87/2	1.97/1.67/1.74

B. Stretching factors analysis

We use the following metrics to evaluate the behavior of our task model for the different aforementioned initial total CPU utilization: *Aver*, *Aver25+* and *Aver75*. *Aver* is the average stretching factor for all the low-criticality tasks. *Aver25+* is the average stretching factor for the 25% most important low-criticality tasks, while *Aver75* is the average stretching factor for the remaining tasks, i.e. the 75% less important tasks.

Table I shows the numerical results obtained for the stretching factors $S_{i,min}$ according to the previously introduced metrics and for each initial utilizations on a 2 processors system. The last column presents the results of $S_{i,min}$ when the importance level parameter is not used, i.e. all the low-criticality tasks have the same importance. Therefore, all the low-criticality tasks will have the same $S_{i,min}$ value, making the use of the *Aver25+* and the *Aver75* metrics unnecessary. In each cell, the three values are respectively the value of $S_{i,min}$ for the task set TS_1, TS_2 and TS_3. The result for the initial utilization of 100% is omitted as all the stretching factors are equal to 1 by construction.

As expected, these results show that the stretching factors are reduced for the most important tasks (*Aver25+*) and much higher for the less important tasks (*Aver75*). For instance, when the initial utilization is 150%, the most important low-criticality tasks have in average their stretching factors ranging from 1 (stretching is not required) to 1.5. On the other hand, the less important low-criticality tasks have in average their stretching factors ranging from 1.87 to 2, the maximum possible value ($S_{i,max}$). Table I also shows that without the importance level the stretching factors are slightly inferior (column 5) than when the importance parameter is used (column 2). That is, the low-criticality tasks should be stretched more when using the importance level parameter. However, when using this parameter in the model to compute stretching factors, the most important low-criticality tasks (column 3) have their stretching factors greatly reduced and even not used in most cases. These results demonstrate that our improved model of elastic tasks allows to execute both the low and the high-criticality tasks of a MC system, while giving first priority to the high-criticality tasks and then to the low-criticality tasks according to their importance level.

Figure 1 shows the distribution of the values of $S_{i,min}$ in two different configurations. Tasks are ordered by decreasing importance level. Configuration A corresponds to the task set TS_3 with an utilization of 150% where the values of V_i are randomly generated. In the configuration B, the application designer specified that the 25% most important low-criticality tasks should have their values of $S_{i,min}$ set to 1.25, while the other tasks can have higher stretching factors ($S_{i,max} = 2$). Such control over the values of the stretching factor for each task opens the opportunity for application designers to more easily dimension the required processing power when

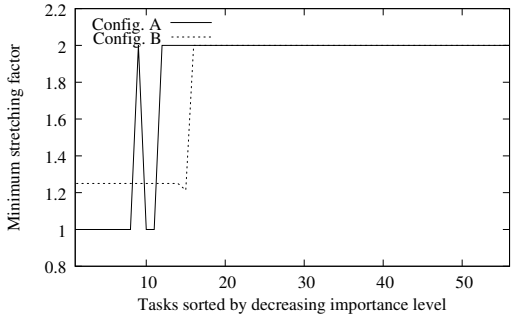

Fig. 1. Evolution of the distribution of stretching factors when setting the $S_{i,min}$ for the 25% most important low-criticality tasks.

designing a MC system.

VI. Conclusion

Within real-time embedded systems, there is a need to embed less (or even non-) safety tasks in addition to hard real-time tasks. This requires executing applications with different levels of criticality on a same chip. Such systems are called Mixed-Criticality (MC) systems. Certification constraints to prove the schedulability of MC systems lead to an over sizing of the CPU power, compared to what is required in average while the system is running. Such an approach is no longer compatible with current economical constraints that push for minimizing the CPU power.

In this work, we propose a task model and an associated on-line decision algorithm to maximize the execution rate of the low-criticality tasks within a safety-critical real-time system. Our task model, inspired by the elastic task model, allows to specify an order between the low-criticality tasks for applying so called stretching factors. Off-line, the minimum value for these stretching factors so that a MC system can be scheduled are computed. On-line, we then show how these stretching factors are used to relax the temporal constraints of the low-criticality tasks in order to avoid any deadline miss, while maximizing the execution rate of the low-criticality tasks. This approach can be used to size the required processing power for designing a MC system.

In future work, we plan to evaluate the actual values stretching factors can take depending on the distribution of the actual execution time of the low-criticality tasks. Besides, we plan to evaluate the overhead introduced by the different possible on-line decision algorithms for increasing/resetting the stretching factors. We also plan to investigate a different approach for supporting the execution part of our contribution, by relying on the use of a generalized form of the time-triggered paradigm, called eXternal-Triggered (xT) [8]. Using this paradigm, recomputing the visibility dates of low-criticality tasks being stretched would no longer be necessary. Finally, it would be interesting to apply the proposed task model in order to lessen the deadline miss ratio of the low-criticality tasks when setting a trade-off with energy consumption [15].

References

[1] P. Albertos, A. Crespo, I. Ripoll, M. Valles, and P. Balbastre. Rt control scheduling to reduce control performance degrading. In *Proc. of the 39th IEEE Conf. on Decision and Control*, volume 5, pages 4889–4894, Sydney, Australia, 2000.

[2] S. Baruah and A. Burns. Implementing mixed criticality systems in ada. In *Proc of the 16th Ada-Europe Intl. Conf. on Reliable software technologies*, pages 174–188, Edinburgh, UK, 2011.

[3] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 145–154, Pisa, Italy, July 2012.

[4] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the 16th Euromicro Conf. on Real-Time Systems (ECRTS 2004)*, pages 196–203, 2004.

[5] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. Comput.*, 51(3):289–302, Mar. 2002.

[6] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 111–126, Washington, USA, 2006.

[7] A. Cervin and J. Eker. The control server: a computational model for real-time control tasks. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 113–120, 2003.

[8] D. Chabrol, D. Roux, V. David, M. Jan, M. A. Hmid, P. Oudin, and G. Zeppa. Time- and angle-triggered real-time kernel for powertrain applications. In *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE 13)*, pages 1060–1063, Grenoble, France, March 2013.

[9] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1):1–40, Jan. 2011.

[10] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 291–300, Washington, DC, USA, December 2009.

[11] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. Rtos support for multicore mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 197–208, 2012.

[12] M. Jan, V. David, J. Lalande, and M. Pitel. Usage of the safety-oriented real-time OASIS approach to build deterministic protection relays. In *Proc. of the 5^{th} Intl. Symp. on Industrial Embedded Systems (SIES 2010)*, pages 128–135, Trento, Italy, 2010.

[13] H. Kopetz. The time-triggered model of computation. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS'98)*, pages 168–177, Madrid, Spain, 1998.

[14] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 110–117, 1995.

[15] V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In *Proc. of the 1st workshop on Real-Time Mixed Criticality Systems*, Tapei, Taiwan, August 2013.

[16] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Design, Automation and Test in Europe (DATE 13)*, pages 147–152, Grenoble, France, March 2013.

[17] M. Velasco, J. M. Fuertes, and P. Marti. The self triggered task model for real-time control systems. In *24th IEEE Real-Time Systems Symposium (work in progress, RTSS 2003)*, pages 67–70, Cancun, Mexico, 2003.

[18] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of the 28th IEEE Intl. Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, Tucson, USA, 2007.

[19] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Transactions on Computers*, 58(10):1382–1397, 2009.