Incorporating The Notion of Importance into Mixed Criticality Systems

Tom Fleming
Department of Computer Science,
University of York, UK.
Email: tdf506@york.ac.uk

Alan Burns
Department of Computer Science,
University of York, UK.
Email: alan.burns@york.ac.uk

***Abstract*—Mixed criticality systems offer the opportunity to integrate system components with different levels of assurance that previously may have been placed on different nodes. While the vast majority of mixed criticality work features a HI and a LO criticality level, LO criticality tasks should not be mistaken for tasks with little value. Such tasks might contain mission critical functionality and are still vital for the correct and efficient operation of the system. A large portion of earlier work immediately suspends all LO criticality functionality upon a criticality change. It is clear that suspending tasks at all is highly undesirable, let alone all tasks of a single criticality level at the same time. In this work we consider this issue, we propose a scheme to maintain the operation of lower criticality tasks for as long as possible, even when the system is executing in a higher mode. We introduce the notion of importance as a means of deciding which tasks are suspended first. This is done with the aim of allowing the system designer to make these decisions and have greater control over the way their system degrades. We conclude that by using essentially the same analysis and functionality that facilitates multiple criticality levels we are able to provide improved lower criticality performance.**

I. INTRODUCTION

The area of Mixed Criticality Systems builds upon an increasing desire to consolidate functionality of different criticality levels onto one platform. This is driven further by the development of more powerful hardware and pressure from industrial sectors, such as aerospace and automotive. A system that consolidates its functionality onto one platform looks to save space, power, weight and reduce hardware costs. Such systems must seek to satisfy two properties, efficient utilisation and isolation of high integrity tasks. Systems may be subject to certification against safety standards such as IEC 61508 or DO-178B, therefore the challenge is providing a means of producing a certifiable system with high overall utilisation.

The initial mixed criticality model proposed by Vestal [11] and used by others such as [3] has a strict notion of a criticality change. A dual criticality system begins executing in the LO criticality mode, if any HI criticality tasks overruns its LO criticality Worst Case Execution Time (WCET) then a criticality change occurs. LO criticality tasks are suspended (although active jobs are allowed to complete), HI criticality tasks are allowed to run to their maximum HI WCET. It is becoming increasingly apparent that the original notion of completely dropping all LO criticality tasks when a criticality change occurs is unacceptable. Although a task might be considered LO criticality, it might still contain mission critical functionality, as such these tasks should not be suspended unless absolutely necessary. It is also clear that simply dropping LO criticality tasks provides the system designer with little control over how their system degrades in the event of an overrun.

The work presented here seeks to address both the issue of immediately dropping all LO tasks and the lack of control over system degradation during overload. This is done by introducing the notion of importance, I. Importance levels are assigned to all tasks except those at the highest criticality level. Importance provides a greater degree of control and granularity over how a system degrades. This controlled degradation is facilitated by a more realistic view on the behaviour of a task during an overrun. When a task overruns its LO criticality WCET, it is unlikely that it will execute to its HI WCET. It is more likely that such a task might only overrun by a small margin. Rather than immediately dropping all LO criticality tasks our approach seeks to only drop tasks when absolutely required. Control over this degradation is given back to the system designer as the order in which tasks are dropped is determined by the assignment of importance (least important tasks are dropped first). We extend this further by considering groups of tasks as a single application. In this case applications are assigned an importance value, applications are dropped according to importance rather than individual tasks. We show the effectiveness of this technique via experimental results on randomly generated sets of tasks.

The remainder of this document is structured as follows: Section II covers related work, Section III introduces and describes Importance, Section IV evaluates the technique via experimental results and Section V provides some concluding remarks.

II. RELATED WORK

In this section we consider related work where the primary focus is investigating the possible use of existing system slack to improve the level of service provided to LO criticality tasks.

Santy et al. [8] consider situations where LO criticality tasks do not need to be dropped during a criticality change. They showed that some slack often exists before any LO criticality task must be suspended. This period of slack is known as the ‘allowance’. Allowance is shared across all HI criticality tasks, if it is used up LO criticality tasks must be suspended. Their work is based on the observations made about a previously developed technique OCBP [2], they claim that: it is never necessary to drop jobs that have a lower criticality but a higher priority than the current level. Although this assumption seems counter-intuitive, it must be noted

that OCBP is an older technique which considers a finite set of jobs where 'no job is allowed to execute for more than its WCET at is own specified criticality'[4]. In short the schedulability test considers all jobs executing in each level, jobs may execute up to their own $C_j(L_j)$ or their $C_j(L_i)$ where C_j is the WCET and L_j is the criticality level of task j. They use sensitivity analysis to calculate the allowance (slack) available for HI criticality tasks. On top of this, the possibility of returning the system to the LO criticality mode is considered. The system looks for a level-ℓ (where ℓ is the criticality level) idle time, a time where no jobs of at least criticality level ℓ are waiting to execute. They use this point as the time where the criticality of the system may be reduced.

Su and Zhu [9] consider an Earliest Deadline First based technique ER-EDF (Early Release - EDF). This technique seeks to allow LO criticality tasks to release earlier than their maximum period if there is slack available from the HI criticality tasks. This work is based upon an Elastic Mixed Criticality task model which allows for variable periods from a desired T_i to a maximum T_i^{max}. The minimum service requirement for LO tasks can be determined by its largest period, LO tasks may execute more frequently if there is no impact on HI criticality tasks. They evaluate their work by assuming reductions of 2 or 5 times to LO criticality periods. The performance of ER-EDF is compared against EDV-VD and is shown to have improved performance. Su et al. [10] extended this work to a multi-core platform.

Both of these techniques represent an attempt to provide some level of service for LO criticality tasks when a system enters the HI mode. Our work goes one step further, as well as considering the potential of using slack to schedule all LO tasks if the overrun is small we also consider the way and order tasks are dropped during a more severe overload.

III. Importance

In this section we will introduce and expand on the notion of importance. We use an adaptation of the standard system model initially defined in [11]. A system constitutes a finite set of applications K. Each of these applications is assigned a criticality level, L (designated by the system designer) and consists of a finite set of sporadic tasks[1]. Each task, τ_i, is defined as $\tau_i = \{\overrightarrow{C_i}, T_i, D_i, L_i\}$ where $\overrightarrow{C_i}$ is a vector of WCETs (one for each criticality), T_i is the period, D_i is the deadline and L_i is the criticality level. Each task gives rise to an unbounded series of jobs. Additionally we consider I_i as the importance of a LO criticality τ_i. Importance might also be assigned to LO criticality applications, which in turn applies this importance level to a group of tasks. In this work we constrain ourselves to consider only criticality dependent WCETs and dual criticality systems with $C_i(LO) \leq C_i(HI)$.

In this work we group sets of tasks at the same criticality level into applications. Applications are designed to better represent a more realistic system model where a group of tasks contribute to a single application. In this case if one task of an application must be suspended, then all other tasks associated with the application must also be suspended. Applications are assigned an importance value, rather than each task individually. It is worth noting that although tasks within an application are of the same criticality, priorities may be interleaved between applications and criticality levels. In the description and examples presented below we consider the case of systems composing of tasks rather than applications. This is to allow for simpler examples that show more clearly the effect of importance.

A. Overview

This work introduces the notion of *importance*, each task within the LO criticality level[2] is assigned an importance value, this provides an order in which tasks might be suspended during an overload. The purpose of this is to provide the designer of the system with more control over the graceful degradation of their system during an overrun. Whereas criticality levels typically involve the assignment of a SIL (Safety Integrity Level) or equivalent, importance is assigned according to how the designer wishes the system to degrade. The difference between criticality and importance is discussed[3] in [6] which considers the properties of a criticality change compared to other mode changes. As mentioned above, the typical response to a criticality change is to drop all tasks of a lower criticality. By assigning importance we suspend tasks in order, lowest importance first. This is done only when the HI criticality overrun is severe enough to warrant the suspension of a task. If the overrun is not severe and there is sufficient slack in the system, it is possible that the system might move into the HI criticality mode, while maintaining all of its LO criticality tasks. As soon as an overrun reaches the point in which a LO criticality task must be dropped, the task with the lowest importance is suspended. Importance provides an extra level of granularity within a level of criticality. The key difference between these two assignments is that criticality is typically assigned due to certification requirements, whereas levels of importance are decided by the system designer. We are using 'importance' as an *ordinal* scale [7] which allows questions such as 'is application A more important than application B?' to be answered. We do not extend the notion to an *interval* or stronger scale that would allow an answer to the following question to be used at run-time: 'Is application A and B more important than application C?'

Sensitivity analysis is used to determine the severity of the overrun required to drop a particular level of importance. Such analysis begins by checking if the system has any initial slack. It seeks to find a point during the HI criticality overrun at which the system is unschedulable and such a LO criticality task must be dropped. Once a point is found at which a task must be dropped, that point is recorded and the analysis begins to search for the next point by increasing the severity of the overrun. This process is repeated until a point is recorded at each time a LO criticality task must be dropped. It is possible that not all LO criticality tasks will be dropped, this depends on the properties of a task set, in particular its HI criticality utilisation. During runtime if a HI criticality task overruns its LO WCET up to the first recorded point, the

[1] All tasks within one application are of the same criticality level.

[2] For a dual criticality system, LO/HI, a system with greater than two criticality levels would see importance assigned within all but the highest level.

[3] Although importance is not named explicitly.

least important LO criticality task is suspended. This process continues if the overrun continues to increase.

We note here that in the case of high priority, high importance LO criticality tasks, we mostly consider their effect on the schedulability of the system even after they have been suspended. In order to do this we must consider their bounded interference on the task set up to the point that they are suspended. This is similar to the way AMC [3] (Adaptive Mixed Criticality) bounds the interference created by high priority LO criticality tasks during a criticality change. Consider the set of tasks in Table I:

	L	P	I
τ_k	LO	1	1
τ_j	LO	2	2
τ_i	HI	3	-

TABLE I
BASIC EXAMPLE.

Initially we consider the schedulability of the LO and HI mode as well as the criticality change [4]. The calculations for the LO mode and the change are shown in Equations (1) and (2) respectively, we exclude the HI mode as there is only one HI criticality task:

$$R_i(LO) = C_i(LO) + \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \quad (1)$$

$$R_i(HI) = C_i(HI) + \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \quad (2)$$

Such that:

$$R_i(HI) <= D_i$$

Where $R_i(HI)$ and $R_i(LO)$ are the response times of τ_i in the HI and LO criticality modes respectively.

We increase the overrun of τ_i from $C_i(LO)$ until we determine a point at which τ_j, the least important task, τ_j at $I = 2$, must be dropped.

$$\begin{aligned} R_i^{I_2}(LO) &= C_i^{I_2}(LO) + \left\lceil \frac{R_i^{I_2}(LO)}{T_k} \right\rceil C_k(LO) + \\ &\quad \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \\ R_i(HI) &= C_i(HI) + \left\lceil \frac{R_i^{I_2}(LO)}{T_k} \right\rceil C_k(LO) + \\ &\quad \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \end{aligned} \quad (3)$$

Next we attempt to increase the overrun until τ_k must be dropped. We use the response time just calculated, $R_i^{I_2}(LO)$ to bound the possible interference of τ_j.

$$\begin{aligned} R_i^{I_1}(LO) &= C_i^{I_1}(LO) + \left\lceil \frac{R_i^{I_1}(LO)}{T_k} \right\rceil C_k(LO) + \\ &\quad \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \\ R_i(HI) &= C_i(HI) + \left\lceil \frac{R_i^{I_1}(LO)}{T_k} \right\rceil C_k(LO) + \\ &\quad \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \end{aligned} \quad (4)$$

In this way we account for the possible LO criticality, high priority interference up to the point at which a task is suspended.

B. Priority assignment

During sensitivity analysis, as the HI criticality tasks overrun is increased, it is likely that a particular LO criticality task will miss its deadline and cause the system to be seen as unschedulable. As we must drop our LO criticality tasks in order of Importance we may not be able to drop the task that might make the set immediately schedulable again. In extreme cases several other LO tasks might need to be dropped before the offending LO task can be dropped and the system can be seen as schedulable at a particular overload level. In a fixed priority system it is highly likely that the task which misses its deadline will be at the lowest priority. By slightly adapting Audsley's priority assignment technique [1] we can attempt to place tasks of lower importance at a lower priority. This is much the same as aiming to give lower criticality tasks lower priorities. This approach would work as follows:

For each priority level, beginning at the lowest. Check the schedulability of each task at this level. If more than one task is schedulable first differentiate by assigning the lower priority to the lower criticality. If many tasks that might be assigned a particular priority are of LO criticality, assign the priority to the task with the lowest importance value.

```
for Each priority level do
    for Each task do
        if criticalityLevel < currentTask then
            if importanceLevel < currentTAsk then
                currentTask=task;
            end
        end
    end
    Assign priority level to currentTask;
end
```

Algorithm 1: Audsley's Approach [1] with importance.

By assigning lower importance tasks lower priorities this reduces the chance of having to drop multiple tasks in order to make the system schedulable again.

C. Examples

A simple example can be used to illustrate the basic functionality of Importance. Consider the task set in Table II:

[4] According to AMCrtb [3].

	C(LO)	C(HI)	T=D	L	P	I
τ_1	2	6	8	HI	1	-
τ_2	1	-	6	LO	2	1
τ_3	2	-	6	LO	3	2

TABLE II
A SIMPLE EXAMPLE.

If τ_1 were to exceed its $C_1(LO)$ by 2, τ_3 would have to be dropped as it would miss its deadline and cause the system to be unschedulable. Finally at an overrun of 4, τ_2 must be suspended. It is worth noting that such tasks would need to be suspended at time 3 for τ_3 and 5 for τ_2 if τ_1 does not signal completion at each of these times.

If the HI criticality tasks are also HI priority, they do not need to worry about interference from LO criticality tasks during their execution. Importance provides us with a set of points at which LO criticality tasks will be prevented from executing. These points give the system designer greater control over HI criticality degradation and allow the system resources to remain highly utilised. Crucially, regardless of the priority levels involved, this approach provides an improved level of service for LO criticality tasks, potentially reducing their likelihood of being suspended.

A second example can be used to highlight some interesting behaviour. Table III shows an example task set with importance assigned to the LO criticality tasks. Sensitivity analysis has been carried out on this set to determine the points at which each task must be dropped during an HI criticality overrun.

	C(LO)	C(HI)	T=D	L	P	I
τ_1	5	15	25	HI	3	-
τ_2	5	-	20	LO	4	3
τ_3	2	-	8	LO	1	2
τ_4	1	-	5	LO	2	1

TABLE III
A MORE COMPLEX EXAMPLE.

The least important task, τ_2 must be dropped when an overrun of HI criticality task τ_1 reaches 5 units of execution without signalling completion. In other words, τ_2 must be dropped as soon as an overrun is detected. Tasks τ_3 and τ_4 may continue to execute, if τ_1 does not complete after 10 units of execution, τ_3 and τ_4 must be suspended.

At each stage of the sensitivity analysis we re-check the schedulability of the system. For example if we assume an overrun of 1 to τ_1 then essentially we do the following calculation.

$$R_2(LO) = 5 + \left\lceil \frac{22}{25} \right\rceil 6 + \left\lceil \frac{22}{8} \right\rceil 2 + \left\lceil \frac{22}{5} \right\rceil 1 = 22 \quad (5)$$

The result of this shows that τ_2 will overrun its deadline if τ_1 exceeds its LO criticality execution by 1. As such it is clear that τ_2 must be suspended as soon as τ_1 reaches 5 units of execution without signalling completion.

The lowest priority task is now τ_1. At an overrun of 10 (9 without signalling completion) the task set is unschedulable.

$$\begin{aligned} R_1(LO) &= 10 + \left\lceil \frac{21}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 21 \\ R_1(HI) &= 15 + \left\lceil \frac{21}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 26 \end{aligned} \quad (6)$$

Thus giving the result of 26, and making an overrun of 10 with both tasks being unschedulable. In this case τ_3 is suspended leaving just τ_1 and τ_4 executing. If we just include these two tasks it would seem that τ_4 does not need to be suspended, the calculation would be as follows:

$$\begin{aligned} R_1(LO) &= 15 + \left\lceil \frac{19}{5} \right\rceil 1 = 19 \\ R_1(HI) &= 15 + \left\lceil \frac{19}{5} \right\rceil 1 = 19 \end{aligned} \quad (7)$$

However when calculating the schedulability of this situation we must include the prior interference from τ_3, we use the previously calculated LO response time at overrun 9[5] to cap the possible interference caused by τ_3.

$$R_1(LO) = 9 + \left\lceil \frac{19}{8} \right\rceil 2 + \left\lceil \frac{19}{5} \right\rceil 1 = \mathbf{19} \quad (8)$$

$$\begin{aligned} R_1(LO) &= 10 + \left\lceil \frac{\mathbf{19}}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 21 \\ R_1(HI) &= 15 + \left\lceil \frac{19}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 26 \end{aligned} \quad (9)$$

Here it is clear that if τ_1 reaches an overrun of 9 without signalling completion both τ_3 and τ_4 must be suspended in order to allow τ_1 to meet its deadline. This can be shown further by considering the execution trace shown in Figure 1. This trace shows the situation where the interference from τ_3 is not considered. However, it is clear that τ_4 may not remain scheduled as the example then shows τ_1 executing until time 26 and exceeding its deadline. If both τ_3 and τ_4 were dropped after an overrun of 9[6], τ_1 would meet its deadline. As such, it is clear that when considering higher priority LO criticality tasks that are suspended, we must account for their impact throughout the execution. This is similar to including the bounded impact of LO criticality tasks on HI criticality tasks during a criticality change.

This example raises a very interesting point. As in the first instance, when trying to find a time in which τ_3 must be dropped we maximised the possible overrun of τ_1, it is clear that at this point τ_4 must also be suspended. This is because we must include the interference suffered from τ_3 until it is suspended. We know that all 3 tasks are not schedulable beyond point 9, as such both must be suspended at the same instant. This is due to the fact we must include the same amount of interference from τ_3 as the previous calculation. It seems likely that as we include interference from previously suspended higher priority LO criticality tasks, most higher priority LO criticality tasks will be dropped at the same instant. It is worth noting that even if such high priority LO tasks are only able to remain schedulable up to a HI criticality overrun of 10%, the likelihood of the system overrunning by 10% may be relatively low.

D. Further points

One of the fundamental assumptions of this work is that HI criticality overruns are not likely to be as severe as their HI WCET suggests. To support this assumption we can look

[5] The last schedulable point before suspension.
[6] Time 19.

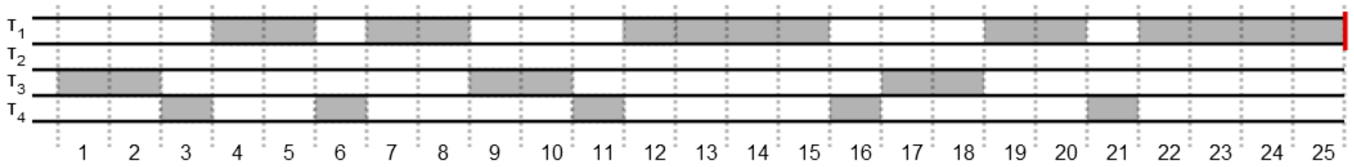

Fig. 1. An execution trace of Table III.

at work carried out on probabilistic real-time systems. Rather than the traditional view of a LO and HI criticality value, this work considers the space between the two values as a large number of points. Each point has its own degree of confidence. This set of points is known as the pWCET distribution. If such a pWCET distribution for a system showed that the likelihood of a task overrunning by more than 60% was extremely small (for example: 10^{-9} failure rate per hour), then simply dropping all LO criticality tasks when a criticality change occurs is poor use of the system resources. Importance is able to improve on this by allowing LO criticality tasks to continue execution, providing they do not effect the execution of HI criticality tasks. Although some LO criticality tasks may have to be dropped, it is likely that a good percentage of these tasks will be able to continue to execute throughout the HI criticality mode. If such a pWCET is known for a system, it would be possible to make predictions on the likelihood of a particular task being dropped. In this way the use of importance and probabilistic reasoning could help provide more detailed guarantees of system performance.

Importance is a useful means of providing a more detailed picture of system performance under HI criticality/overload conditions while passing more control over system degradation to the designer.

IV. Evaluation

The notion of importance is relatively easy to explain, however, its effectiveness is not so easy to quantify. As schedulability is not improved via the use of this technique another means of showing its effectiveness is required. In our evaluation we firstly illustrate simply how our approach is able to stagger the process of dropping LO criticality tasks and secondly we consider a probabilistic view that considers the severity of an overrun. Both illustrations show how our approach is able to reduce the chance of LO criticality tasks being suspended.

Our experimental data was produced from 10,000 randomly generated tasks with a total LO utilisation of 85%, these were created as follows. Utilisation values were generated via the UuniFast Algorithm [5], periods between X and Y were generated in a log uniform distribution. Our task sets are dual criticality, $C(LO)$ values were created from the periods and utilisations generated $C = U^{*}T$, $C(HI)$ values were 2 times $C(LO)$. 25 LO criticality tasks and 5 HI criticality tasks were generated per task set. The LO criticality tasks we randomly grouped into 5 applications (5 tasks per application), each application was randomly assigned a level of importance. HI criticality tasks were left as individual tasks rather than applications and no assignment of importance is required for this level. Priorities were assigned via our version of Audsley's algorithm [1] as seen in Section III, part B.

It is worth noting that in experiments such as this, there are a huge number of parameters which will affect what the results look like. The total number of tasks will effect the results, as will the distribution of these tasks between HI and LO criticality. The relative utilisation of the HI and LO modes has a big impact as to when LO criticality applications must be dropped, as does the difference between a HI criticality task, τ_i's, $C_i(LO)$ and $C_i(HI)$. We have described the values we used in our experimentation, different parameters will produce different looking graphs. However, the key result remains the same regardless of the parameters used, introducing levels of importance will provide a better level of service for LO criticality tasks.

In our work we also introduced the notion of groups of tasks as applications. Tasks of one application share a level of importance and will therefore be suspended as a group. It is worth noting that although applications share a level of importance, the priorities of individual tasks may be interleaved. The purpose of this is to better capture the nature of applications as groups of tasks, although these tasks might be interconnected we only consider independent tasks in this work.

Our experiments firstly ran each generated task set through the schedulability test AMCrtb [3] to ascertain schedulability and if schedulable, provide a priority ordering. Each task set that passed this test was then run through sensitivity analysis to determine the points at which LO criticality applications must be dropped, these points were recorded and used later to present the results. In some cases a task set might only need to drop one application and in others it might need to drop all 5. During sensitivity analysis all HI criticality WCET values are increased by the same percentage, this seemed like a reasonable assumption as it is difficult to model the likelihood of each HI criticality task individually overrunning. On top of this, a single HI criticality task overrunning its LO WCET is not likely to have that much of an impact on the system, especially as our HI criticality tasks do not have particularly high utilisations individually.

Figure 2 shows the number of applications dropped on the Y axis against the severity of the overrun as a percentage increase from $C(LO)$ on the X axis. The graph clearly shows that our approach is able to maintain LO criticality functionality for a significantly increased amount of time. This is even more apparent if you consider that the probability of an overrun occurring even beyond the 5% initial slack becomes exponentially more unlikely. Figure 3 shows the potential likelihood of an overrun reaching each point and causing a task to be suspended. The probabilities used here are merely designed to illustrate the point and are not meant as realistic values.

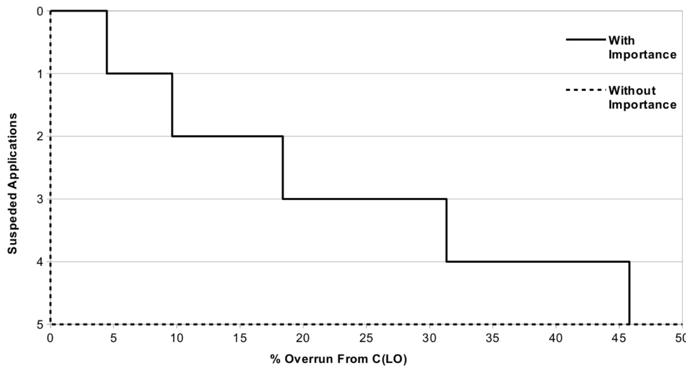

Fig. 2. Results from 10,000 random task sets.

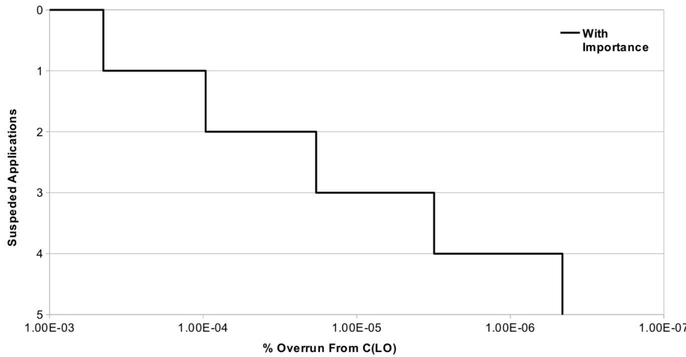

Fig. 3. Example probability of overrun in log scale.

If we consider Figure 3 with a linear scale it is easy to see that, even if the system can maintain all LO tasks during an overrun of 5%, this is still a significant improvement when taking into account the probability of the overrun actually reaching that level. This is shown in Figure 4:

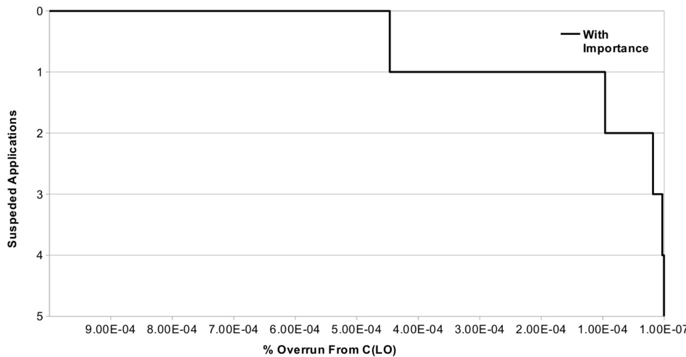

Fig. 4. Example probability of overrun in linear scale.

V. Conclusion

It is clear that as we move to consider more realistic mixed criticality implementations, simply dropping LO criticality tasks when a criticality change occurs is unacceptable. In this work we have introduced the notion of importance, we discussed the reasoning and illustrated the benefits through discussion and experimental results. Importance provides the designer of a system with a greater level of control and knowledge over the likely behaviour of their system during a criticality change. We show the effectiveness of importance by considering the reduced number of tasks dropped and the increased HI criticality system utilisation. This is done via experimental evaluation on randomly generated task sets. During the experimentation we introduced the notion of several tasks grouped as applications, applications aim to provide a more realistic system model. Further work might consider importance at greater than two criticality levels or it might consider a means of re-introducing LO criticality tasks when recovering from an overrun. To summarise, we have introduced importance as a means to provide a greater level of control and guarantees for LO criticality tasks during a criticality change.

Acknowledgements

The authors acknowledges the support and funding provided for this work by BAE Systems, and the ESPRC (UK) via MCC grant (EP/K011626/1).

References

[1] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

[2] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. In P. Hlinn and A. Kuera, editors, *Mathematical Foundations of Computer Science 2010*, volume 6281 of *Lecture Notes in Computer Science*, pages 90–101. Springer Berlin Heidelberg, 2010.

[3] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34 –43, 29 2011-dec. 2 2011.

[4] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 13 –22, april 2010.

[5] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[6] A. Burns. System mode changes - general and criticality-based. volume WMC RTSS 2014, 2014.

[7] D. Prasad, A. Burns, and M. Atkin. The measurement and usage of utility in adaptive real-time systems. *Journal of Real-Time Systems*, 25(2/3):277–296, 2003.

[8] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 155 –165, july 2012.

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

[10] H. Su, D. Zhu, and D. Mosse. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 352–357, Aug 2013.

[11] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239 –243, dec. 2007.