WMC 2015

Proceedings of the 3rd International Workshop on Mixed Criticality Systems

Edited by

Liliana Cucu-Grosjean

Robert I. Davis

Organisers

Program Chairs

Liliana Cucu-Grosjean
Robert I. Davis

Steering Committee

Sanjoy Baruah
Liliana Cucu-Grosjean
Robert I. Davis
Claire Maiza

Program Committee

Abhilash Thekkilakattil
Adriana Gogonel
Arvind Easwaran
Björn B. Brandenburg
David Broman
Haohan Li
Kai Lampka
Iain Bate
Marcus Völp
Mitra Nasri
Nan Guan
Risat Mahmud Pathan
Sebastian Altmeyer
Sebastian Faucou
Sophie Quinton

Message from the Program Chairs

It is our pleasure to welcome you to the 3rd International Workshop on Mixed Criticality Systems (WMC) at the Real-Time Systems Symposium (RTSS) in San Antonio, Texas, USA on 1st December 2015.

The purpose of WMC is to share new ideas, experiences and information about research and development of mixed criticality real-time systems. The workshop aims to bring together researchers working in fields relating to real-time systems with a focus on the challenges brought about by the integration of mixed criticality applications onto single-core, multicore and manycore architectures. These challenges are cross-cutting. To advance rapidly, closer interaction is needed between the sub-communities involved in real-time scheduling, real-time operating systems / runtime environments, and timing analysis.

For this third edition of the workshop a total of 8 submissions were received. The review process involved 15 Program Committee members, with each submission receiving at least 3 reviews. All 8 papers were selected for presentation. Our thanks go to the WMC Program Committee for the time and effort they put into carefully reviewing the submissions, and for meeting the tight timescales set for reviews.

As well as regular paper presentations, there will also be a special session on "Important Open Problems in Mixed Criticality Systems" this session includes contributions from the workshop participants, and promises a lively forum for discussion.

WMC 2015 would not be possible without the hard work of a number of people involved in the organisation of RTSS 2015, including Chris Gill, Marco Caccamo, Dakai Zhu, and Linda Buss. In particular, we would like to thank the RTSS 2015 Workshops Chair, Harini Ramaprasad for her excellent organisation of the overall workshops program.

We also thank the WMC Steering Committee for their guidance and the MCC (UK EPSRC EP/K011626/1), Proxima (EU FP7 IP 611085), Departs (French BGLE), Capacites (French LEOC) and Waruna (French FUI) projects for their support.

Finally, we would like to thank *all* of the authors who submitted their work to WMC 2015, without them, this workshop would not be possible. We wish you an interesting and exciting workshop and an enjoyable stay in San Antonio. We look forward to seeing you again at WMC 2016.

Liliana Cucu-Grosjean (INRIA, Paris-Rocquencourt, France)
Robert I. Davis (University of York, UK)
WMC 2015 Program Chairs

Mixed-criticality job models: a comparison

Sanjoy Baruah Zhishan Guo
Department of Computer Science
The University of North Carolina at Chapel Hill.
{baruah,zsguo}@cs.unc.edu

***Abstract*—The Vestal model [6] in widely used in the real-time scheduling community for representing mixed-criticality real-time workloads. This model requires that multiple WCET estimates – one for each criticality level in a system – be obtained for each task. Burns suggests [3] that being required to obtain too many WCET estimates may place an undue burden on system developers, and proposes a simplification to the Vestal model that makes do with just two WCET estimates per task. Burns makes a convincing case in favor of adopting this simplified model; here, we report on our attempts at comparing the two models – Vestal's original model, and Burns' simplification – with regards to expressiveness, as well as schedulability and the tractability of determining schedulability.**

I. Introduction

In the model for real-time mixed-criticality (MC) workloads that was proposed by Vestal [6] and forms the basis of a significant fraction of the research being conducted within the mixed-criticality real-time scheduling community, each job in an MC system with L distinct criticality levels is characterized by L worst-case execution time (WCET) estimates, one corresponding to each criticality level in the system under analysis. Burns recently proposed (in, e.g., the addendum [3] to his keynote presentation at the Dagstuhl Seminar *Mixed Criticality on Multicore/Manycore Platforms*) a simplification to this model, in which each job J_i is characterized by just two WCET estimates regardless of the number of distinct criticality levels in the system. One, denoted $C_i(\text{SELF})$ or $C_i(\mathsf{SF})$, is determined at a level of assurance that is consistent with its own criticality level (denoted χ_i); a second, denoted $C_i(\text{NORMAL})$ or $C_i(\mathsf{NL})$, is determined at a level of assurance that is consistent with the *lowest* (i.e., least critical) criticality level in the entire system. (For jobs of criticality equal to the lowest criticality level in the system, these two estimates are the same.) The run-time behavior desired of the system is as follows:

- If each job J_i executes for no more than its $C_i(\mathsf{NL})$ value then all jobs' deadlines are met; intuitively, this represents the "normal" behavior of the system.
- Each job J_i is prevented, by run-time monitoring, from executing for a duration greater than $C_i(\mathsf{SF})$.
- If any job J_i of criticality level χ_i executes for more than $C_i(\mathsf{NL})$, then
  - jobs that are less critical than J_i are no longer guaranteed.
  - the remaining jobs all complete by their deadlines, provided each such job J_j executes for no more than $C_j(\mathsf{SF})$ if $\chi_j = \chi_i$, and for no more than $C_j(\mathsf{NL})$ if χ_j denotes a greater criticality level than χ_i (i.e., J_j is more critical than J_i.)

  In other words, the only jobs that are guaranteed to complete execution by their deadlines are those of criticality greater than, or equal to, the criticality of the greatest-criticality job J_i to execute beyond its $C_i(\mathsf{NL})$ value.

Burns [3] makes a strong and convincing case justifying his simplification of the Vestal model from a pragmatic implementation-oriented perspective. Burns' model turns out to bear some similarities with an earlier model proposed by de Niz *et al*. [4], which, too, was inspired by the experience of de Niz *et al*. in implementing mixed-criticality systems. The evidence is thus strong that this model is a very reasonable and potentially useful one, meriting deeper analysis. We have initiated such an analysis from a scheduling-theoretic perspective; in this paper, we report on out initial findings. We restrict attention here to the scheduling of mixed-criticality systems that are modeled as collections of independent jobs executing upon a preemptive uniprocessor. Our findings thus far may be summarized as follows.

The Burns model is strictly less expressive than the Vestal model. Determining whether a given instance can be scheduled correctly remains NP-hard in the strong sense. Lower bounds on schedulability, as quantified using the speedup factor metric, are no better for the Burns model than for the Vestal model.

That is, although the reduced expressiveness of the Burns model makes it easier to use in many practical contexts, it does not reduce the inherent intractability of schedulability analysis, nor make the scheduling problem any easier.

Organization. The remainder of this paper is organized as follows. We formally describe the Vestal and Burns models, and state some more-or-less obvious facts concerning the relationship between them, in Section II. In Section III we show that scheduling instances specified using the simpler Burns model appears to be as difficult as scheduling instances specified using the Vestal model. We conclude in Section IV with some pointers to future work.

II. Model

In this section, we start out in Section II-A briefly reviewing the Vestal model [6], and provide definitions of the major concepts – *behavior*, *criticality level* of a behavior, *correctness* criteria, *clairvoyant schedulability*, *MC schedulability*, etc. —

of mixed-criticality scheduling, and summarize some prior results concerning the preemptive uniprocessor scheduling of mixed-criticality systems that are modeled as collections of independent jobs executing upon a preemptive uniprocessor. Next, we briefly describe the Burns model [3] in Section II-B, explaining how the concepts of MC scheduling are adapted to apply to the Burns model. In Section II-C, we make some rather straightforward observations concerning the Burns model, and its relationship with the Vestal model, with regards to the preemptive uniprocessor scheduling of collections of independent jobs.

A. The Vestal model

In the Vestal model [6], a mixed-criticality (MC) *job* is characterized by a 4-tuple of parameters: $J_i = (A_i, D_i, \chi_i, C_i)$, where

- $A_i \in R^+$ is the release time.
- $D_i \in R^+$ is the deadline. We assume that $D_i \geq A_i$.
- $\chi_i \in N^+$ denotes the criticality of the job, with a larger value denoting higher criticality.
- $C_i : N^+ \rightarrow R^+$ specifies the worst case execution time (WCET) estimate of J_i for each criticality level. (It is reasonable to assume that $C_i(\ell)$ is monotonically non-decreasing with increasing ℓ.)

An MC *instance* is specified as a finite collection of such MC jobs: $I = \{J_1, J_2, \ldots, J_n\}$. Given such an instance, we are concerned here with determining how to schedule it to obtain correct behavior; in this document, we restrict our attention to scheduling on preemptive uniprocessor platforms.

Behaviors. The MC job model has the following semantics. Each job J_i is released at time-instant A_i, needs to execute for some amount of time γ_i, and has a deadline at time-instant D_i. The values of A_i and D_i are known from the specification of the job. However, the value of γ_i is not known from the specifications of J_i, but only becomes revealed by actually executing the job until it *signals* that it has completed execution. γ_i may take on very different values during different execution runs: we will refer to each collection of values $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ as a possible *behavior* of instance I.

The *criticality level* of the behavior $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ of I is the smallest integer ℓ such that $\gamma_i \leq C_i(\ell)$ for all $i, 1 \leq i \leq n$. (If there is no such ℓ, then we define that behavior to be *erroneous*.)

Scheduling strategies. A *scheduling strategy* for an instance I specifies, in a completely deterministic manner for all possible behaviors of I, which job (if any) to execute at each instant in time. A *clairvoyant* scheduling strategy knows the behavior of I — i.e., the value of γ_i for each $J_i \in I$ — prior to generating a schedule for I. By contrast, an *on line* scheduling strategy does not have a priori knowledge of the behavior of I: for each $J_i \in I$, the value of γ_i only becomes known by executing J_i until it signals that it has completed execution. Since these actual execution times – the γ_i's – only become revealed during run-time, an on-line scheduling strategy does not a priori know what the criticality level of any particular behavior is going to be; at each instant, scheduling decisions are made based only on the partial information revealed thus far.

Correctness. A scheduling strategy is *correct* if it satisfies the following criterion for each $\ell \geq 1$: when scheduling any behavior of criticality level ℓ, it ensures that every job J_i with $\chi_i \geq \ell$ receives sufficient execution during the interval $[A_i, D_i)$ to signal that it has completed execution.

MC schedulability. Let us define an instance I to be MC schedulable if there exists a correct on-line scheduling strategy for it. The *MC schedulability problem* then is to determine whether a given MC instance is MC schedulable or not.

Some prior results. In the following, let s_L denote the root of the equation

$$x^L = (1+x)^{L-1}. \tag{1}$$

For $L \leftarrow 2$, this is root of the equation $x^2 = x + 1$; it takes on the value $(\sqrt{5}+1)/2$ and is commonly called the Golden Ratio or the Divine Proportion, notated Φ.

- Determining whether a given instance is MC-schedulable is NP-hard in the strong sense [2]. This holds even if all the jobs in the instance have the same release date, and there are just two distinct criticality levels in the instance.
- It was also shown [2] that there are instances with L distinct criticality levels that are clairvoyantly schedulable upon a unit-speed processor but not scheduled correctly upon a speed-s processor by any fixed-priority (FP) algorithm[1], for each $s < s_L$.
- An FP algorithm called OCBP was defined [1] for scheduling MC instances upon a preemptive uniprocessor. It was shown [2] that any instance with L distinct criticality levels that is MC-schedulable upon a unit-speed processor is scheduled correctly by OCBP upon a speed-s_L processor. This speedup bound for OCBP was shown to be tight: there are instances with L distinct criticality levels that are MC-schedulable upon a unit-speed processor but not scheduled correctly upon a speed-s processor by OCBP for each $s < s_L$.

B. The Burns model

In the Burns model [3], a mixed-criticality (MC) *job* is characterized by a 5-tuple of parameters: $J_i = (A_i, D_i, \chi_i, C_i(\mathsf{NL}), C_i(\mathsf{SF}))$, where A_i, D_i, and χ_i have exactly the same interpretation as in the Vestal model, and

- $C_i(\mathsf{NL}) \in R^+$ specifies the WCET estimate of J_i at criticality level 1 (the lowest criticality level)
- $C_i(\mathsf{SF}) \in R^+$ specifies the WCET estimate of J_i at the criticality level χ_i. (It is reasonable to assume that $C_i(\mathsf{NL}) \leq C_i(\mathsf{SF})$.

[1] An FP algorithm determines, prior to run-time, a total ordering of the jobs in a priority list and during run-time executes at each moment in time the currently active job with the highest priority. Note that EDF is an FP algorithm according to this definition.

The notion of *instance* and *behavior* is the same for the Burns and the Vestal models. The *criticality level* of the behavior $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ is defined as follows:

- If $\gamma_j > C_j(\mathsf{SF})$ for any j, $1 \leq j \leq n$, then the behavior is erroneous.
- Else, the criticality level of the behavior is defined to be the criticality level of the greatest-criticality job J_j with execution exceeding its $C_j(\mathsf{NL})$ value:

$$\max_{j=1}^{n}\{\chi_j \mid \gamma_j > C_j(normal)\}$$

The notions of *scheduling strategy*, clairvoyance, correctness, and MC-schedulability are identical for the Vestal and Burns models.

C. Some observations

Since correctness requirements (i.e., which jobs are required to complete execution by their deadlines for the execution to be considered correct) for mixed-criticality instances are specified in a manner that depends upon the criticality level assigned to behaviors, we first investigate, in Propositions 1 and 2 below, whether the Vestal and Burns models assign behaviors the same criticality level or not.

Proposition 1: Any instance represented in the Burns model can be represented exactly in the Vestal model.

Proof: A job J_i that is specified according to the Burns model can be completely represented in the Vestal model by setting the WCET parameter values as follows:

$$C_i(\ell) \leftarrow \begin{cases} C_i(\mathsf{NL}) & \text{if } \ell < \chi_i \\ C_i(\mathsf{SF}) & \text{otherwise (i.e., if } \ell \geq \chi_i) \end{cases}$$

Consider any instance I in the Burns model, and let I' denote the instance in the Vestal model that is obtained by applying the above transformation t each job in I. Consider any behavior $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ of instance I; this can also be considered a behavior of the Vestal instance I'. It follows from the definitions in Sections II-A and II-B above that this behavior is assigned exactly the same criticality level for I and I'; hence, the correctness requirements for both I and I' are identical. ■

Proposition 2: Instances represented in the Vestal model *cannot* always be represented exactly in the Burns model.

Proof: We illustrate this by an example. Consider the following instance $I = \{J_1, J_2, J_3\}$ represented in the Vestal model:

J_i	A_i	D_i	χ_i	C_i
J_1	0	3	1	$\langle 1, 1, 1\rangle$
J_2	0	3	2	$\langle 1, 1, 1\rangle$
J_3	0	3	3	$\langle 1, 2, 3\rangle$

Its representation in the Burns model would be as follows:

J_i	A_i	D_i	χ_i	$C_i(\mathsf{NL})$	$C_i(\mathsf{SF})$
J_1	0	3	1	1	1
J_2	0	3	2	1	1
J_3	0	3	3	1	3

Under the Vestal model, a behavior of the instance with $\gamma_1 = \gamma_2 = 1$, $\gamma_3 = 2$ has criticality level equal to 2 and hence requires that jobs J_2 and J_3 both complete by their deadlines. Under the Burns model, however, this same behavior has a criticality level equal to 3, and requires only that J_3 complete by its deadline: this is a *weaker* requirement than was mandated in the original (i.e., in the Vestal model). ■

It is evident that the Vestal model requires more parameters than the Burns model in order to specify an instance. What Proposition 2 illustrates is that these additional parameters in the Vestal model do indeed allow for the specification of a more nuanced set of requirements for a given instance. Taken together, Propositions 1 and 2 above consequently yield the (not unexpected) conclusion that *the Vestal model is strictly more expressive than the Burns model*.

Next, we explore whether this reduced expressiveness buys us anything in terms of tractability of analysis with respect to determining whether a given instance is MC-schedulable or not; Proposition 3 reveals that it does not:

Proposition 3: Determining whether a given instance specified according to the Burns model is MC-schedulable is NP-hard in the strong sense. This holds even if all the jobs in the instance have the same release date, and there are just two distinct criticality levels in the instance.

Proof Sketch: It may be verified that the intractability proof for the Vestal model [2, Theorem 1] only involves instances with just two criticality levels, in which all jobs have the same release date. Since the Vestal and Burns models are identical for two criticality levels, this proof hold unchanged for the Burns model as well, and its conclusion continues to hold for the Burns model. ■

III. Priority-based scheduling

As a consequence of Proposition 3, we are unlikely to be able to design an exact schedulability test to efficiently determine whether a given instance specified in the Burns model is MC-schedulable or not. But what about *sufficient* schedulability tests? Here, Proposition 1 means that we may use prior results that were developed for instances represented using the Vestal model to schedule instances that are specified using the Burns model as well. In particular, prior algorithms such as OCBP [1], MC-EDF [5], etc. may continue to be used for scheduling MC instances specified using the Burns model; their performance metrics are guaranteed to be no worse for Burns instances than for Vestal instances. In particular, we may conclude from prior results [2] that OCBP has a speedup bound no worse than s_L (recall that s_L is defined to be the root of Equation 1) in scheduling any instance with L distinct criticality levels.

A natural question to ask at this point in time is, do these algorithms offer better performance guarantees when scheduling instances specified using the Burns model than they do when scheduling instances specified using the more expressive Vestal model? Somewhat surprisingly, the answer

turns out to be "no." A close examination of the proofs of the analogous results in [2] reveal that

1 There are instances with L distinct criticality levels that are MC-schedulable upon a unit-speed processor but not scheduled correctly upon a speed-s processor by OCBP for each $s < s_L$.
2 There are instances with L distinct criticality levels that are clairvoyantly schedulable upon a unit-speed processor but not scheduled correctly upon a speed-s processor by *any* fixed-priority (FP) scheduling policy, for each $s < s_L$.

Both these results may be proved using techniques essentially identical to the ones used in proving the corresponding results in [2] for instances specified using the Vestal model; for the sake of completeness, we formally present the second result as Theorem 1 below, and provide a complete proof.

Theorem 1: There are MC instances with L distinct criticality levels specified using the Burns model that are clairvoyantly-schedulable, but that are not Π-schedulable for any fixed priority policy Π on a processor that is less that s_L times as fast.

Proof: Consider an instance with L criticality levels and L jobs:

	A_i	D_i	χ_i	$C_i(\mathsf{NL})$	$C_i(\mathsf{SF})$
J_1	0	D_1	1	D_1	D_1
J_i $(\forall i \geq 2)$	0	D_i	i	$D_i - D_{i-1}$	D_i

where the values of the D_i's will be specified later and shown to satisfy $D_i > D_{i-1}$ for all i, $1 < i \leq L$.

For example, this instance would look as follows for $L \leftarrow 3$:

J_i	A_i	D_i	χ_i	$C_i(\mathsf{NL})$	$C_i(\mathsf{SF})$
J_1	0	D_1	1	D_1	D_1
J_2	0	D_2	2	$D_2 - D_1$	D_2
J_3	0	D_3	3	$D_3 - D_2$	D_3

The system is clairvoyantly schedulable since, for a behavior of criticality-level ℓ, a clairvoyant scheduler could have each job complete by its deadline by

- not executing jobs $J_1, \ldots, J_{\ell-1}$ at all;
- executing job J_ℓ for a duration $C_\ell(\mathsf{SF}) = D_\ell$ over the interval $[0, D_\ell)$; and
- executing each job $J_j \in \{J_{\ell+1}, \ldots, J_L\}$ for a duration $C_\ell(\mathsf{NL}) = D_j - D_{j-1}$ over the interval $[D_{j-1}, D_j)$.

In the remainder of this proof, we will derive values for the D_i parameters such that this instance cannot be scheduled correctly by any FP scheduling algorithm. That will serve to show that this instance is clairvoyantly schedulable but not FP-schedulable, and hence establish the correctness of the theorem.

In any FP algorithm, some job from amongst the L jobs $J_1, \ldots, J_L$ in the instance must be assigned the lowest priority. Suppose that that job were J_i, and consider a behavior of the instance of criticality level i in which

- each job J_j with criticality lower than that of J_i executes for an amount $C_j(\mathsf{SF}) = D_j$,
- each job J_j with criticality greater than that of J_i executes for an amount $C_j(\mathsf{NL}) = D_j - D_{j-1}$, and
- job J_i executes for an amount equal to $C_i(\mathsf{SF}) = D_i$.

Since J_i is the lowest-priority job, it will only complete after an amount of execution equal to

$$\begin{aligned} & \big(\sum_{j=1}^{i-1} D_j\big) + D_i + \big(\sum_{j=i+1}^{L} (D_j - D_{j-1})\big) \\ = \quad & \big(\sum_{j=1}^{i-1} D_j\big) + D_L \end{aligned}$$

has completed. For J_i to meet its deadline on a speed-s processor, we therefore need this amount to be $\leq s \times D_i$:

$$\begin{aligned} & s\, D_i \geq \big(\sum_{j=1}^{i-1} D_j\big) + D_L \\ \Leftrightarrow \quad & s \geq \frac{D_L + \sum_{j=1}^{i-1} D_j}{D_i} \end{aligned}$$

Since *some* job from amongst the L jobs $\{J_1, J_2, \ldots, J_L\}$ must be assigned lowest priority by a fixed-priority policy, it follows that

$$\min_{1 \leq i \leq L} \Big\{ \frac{D_L + \sum_{j=1}^{i-1} D_j}{D_i} \Big\} \tag{2}$$

is a lower bound on the speedup necessary for a fixed-priority scheduling policy to successfully guarantee to schedule the instance correctly. This minimum is maximized when all L of the terms are equal to each other (and thus define the minimum). Let x be this maximum value. Instantiating the term in Expression 2 for $i \leftarrow L - 1$, we have

$$\begin{aligned} & x = \frac{D_L + \sum_{j=1}^{L-2} D_j}{D_{L-1}} \\ \Leftrightarrow \quad & x D_{L-1} = D_L + \sum_{j=1}^{L-2} D_j \end{aligned} \tag{3}$$

Next instantiating the term in Expression 2 for $i \leftarrow L$, we have

$$\begin{aligned} x \quad = \quad & \frac{D_L + \sum_{j=1}^{L-1} D_j}{D_L} \\ = \quad & \frac{\big(D_L + \sum_{j=1}^{L-2} D_j\big) + D_{L-1}}{D_L} \quad \text{(Rearranging terms)} \\ = \quad & \frac{x D_{L-1} + D_{L-1}}{D_L} \quad \text{(By Eqn 3 above)} \\ = \quad & \frac{(1+x) D_{L-1}}{D_L} \end{aligned} \tag{4}$$

Hence we have

$$\begin{aligned} D_L &= \Big(\frac{1+x}{x}\Big) D_{L-1} \\ &= \Big(\frac{1+x}{x}\Big)^2 \times D_{L-2} \\ &= \Big(\frac{1+x}{x}\Big)^3 \times D_{L-3} \\ &\cdots \\ &= \Big(\frac{1+x}{x}\Big)^{L-1} \times D_1 \end{aligned} \tag{5}$$

Finally instantiating the term within Expression 2 for $i \leftarrow 1$, we have

$$x = \frac{D_L}{D_1} \tag{6}$$

From Equations 5 and 6 above, we are able to conclude that

$$\begin{aligned} & x = \Big(\frac{1+x}{x}\Big)^{L-1} \\ \Leftrightarrow \quad & x^L = (1+x)^{L-1} \end{aligned}$$

which is exactly Equation 1. It's solution is therefore equal to s_L, and the theorem is proved. ■

IV. Context and Conclusions

The Burns model for mixed-criticality workloads was proposed [3] as a simplification of the Vestal model [6] that has formed the basis of a large volume of research in real-time scheduling theory. From a pragmatic perspective and in terms of ease of use, there are undoubted benefits in using the Burns model in preference to the Vestal model — some of these benefits are persuasively articulated in [3]. However, this ease of use does come with some loss of expressiveness (as illustrated in Proposition 2). In our research, we are seeking to better understand whether this reduced expressiveness yields any analytical benefits in terms of reduced complexity of feasibility analysis, less schedulability loss, etc. Thus far, our results have been negative – we have not identified any such benefits.

In this paper, we have restricted attention to MC instances that are characterized as collections of independent jobs. In the future, we plan to study systems that are modeled as collections of recurrent tasks, as well as more general (e.g., multiprocessor) platforms.

Acknowledgements

We are grateful to Alan Burns and to the anonymous reviewers for detecting several typos in an earlier version of this paper.

This research was supported in part by NSF grants CNS 1115284, CNS 1218693, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors Corp.

References

[1] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

[2] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

[3] A. Burns. An augmented model for mixed criticality. In S. K. Baruah, L. Cucu-Grosjean, R. I. Davis, and C. Maiza, editors, *Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121)*, volume 5. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2015.

[4] D. de Niz, K. Lakshmanan, and R. R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the Real-Time Systems Symposium*, pages 291–300, Washington, DC, 2009. IEEE Computer Society Press.

[5] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. In *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, Paris (France), 2013. IEEE Computer Society Press.

[6] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

MC-Fluid: rate assignment strategies

Saravanan Ramanathan, Arvind Easwaran
Nanyang Technological University, Singapore
Email: {saravana016, arvinde}@ntu.edu.sg

***Abstract*—In this paper, we consider fluid scheduling of mixed-criticality implicit-deadline sporadic task systems. Fluid scheduling allows tasks to be allocated fractional processing capacity, which, although hard to implement, significantly improves the schedulability performance. For dual-criticality systems, dual-rate fluid scheduling in which each task is assigned two execution rates depending on the system criticality level has been proposed in the past. An optimal rate assignment algorithm for such systems called MC-Fluid that assigns rates with polynomial complexity has also been proposed. Another rate assignment strategy called MCF has recently been proposed with linear run-time complexity. Although MCF results in a lower schedulability ratio when compared to MC-Fluid, it is shown that both the strategies result in a speed-up optimal scheduling algorithm for dual-criticality systems. We propose two new algorithms to assign execution rates called MC-Sort and MC-Slope, both with linearithmic (i.e., *nlogn*) complexity in the number of tasks n. The proposed algorithms have a schedulability ratio that is significantly better than MCF and almost as good as MC-Fluid, but with a reduced run-time complexity when compared to MC-Fluid.**

I. Introduction

Increasing trend in the embedded industry towards platform integration has motivated Mixed-Criticality (MC) systems in the research community. These systems integrate multiple components with varying criticality onto a common hardware. Safety-critical cyber-physical systems such as automotive and avionics fall under this category. Recently, the complexity of such systems has increased owing to increased functionality. Multi-cores have therefore become a natural choice to meet the growing demand of such systems.

In this paper, we consider the problem of multi-core MC scheduling of implicit-deadline sporadic task systems. We consider a type of global scheduling called fluid scheduling in which each task is assigned a fraction of a processing core at each time instant. Assignment criteria are subjected to two constraints: 1. No task is allowed to have an assignment greater than 1 and 2. Sum of assignments of all the tasks should not exceed the total processing capacity of the system. Though fluid scheduling provides better schedulability, it is practically infeasible to implement as heavy overhead is incurred due to frequent context switching. Levin et al. [1] proposed a method to convert fluid schedules into non-fluid schedules without any loss in performance and thereby, making fluid scheduling practically feasible.

Lee et al. [2] proposed a dual-rate fluid algorithm MC-Fluid for scheduling dual-criticality (LO and HI) implicit-deadline sporadic task systems on a multi-core platform. In a dual-rate fluid scheduling, tasks are assigned two execution rates, one in each of LO and HI modes, based on their execution requirements in the two modes. These rates are criticality-dependent as tasks have different execution requirement for different criticality levels, and the criticality level of system changes at run-time. MC-Fluid determines the values of these execution rates by solving a convex optimization problem in polynomial time, and is shown to have an optimal rate assignment strategy [2]. Baruah et al. [3] derived a simplified fluid scheduling algorithm called MCF with an optimal speed-up factor of 4/3 and linear time rate assignment strategy. Although MCF compromises on the schedulability when compared to MC-Fluid, it has lower time complexity for determining the rates and is speed-up optimal.

Extending MC-Fluid to multi-rate model or to multi-criticality systems without compromising on the complexity is quite hard. Formulating and solving an optimization problem for such system can be challenging. In case of MCF, extension to such systems is rather simple, but it will significantly affect the schedulability.

Contributions: We propose two fluid algorithms: MC-Sort and MC-Slope for computing the execution rate of each task at different criticality levels. MC-Sort algorithm sorts all the high-critical tasks based on their high-critical execution requirement, and assigns a larger rate for a task with larger execution requirement. The challenge with the MC-Sort algorithm is that it does not consider the difference in execution requirement of tasks between different criticality levels. It is necessary to allocate higher rate to such tasks since they need to execute more in high-criticality. MC-Slope algorithm, on the other hand, assigns a larger rate to a task with a larger rate of change in the execution requirement between different criticality levels.

To evaluate the performance of our algorithms, experiments are conducted with randomly generated tasks sets. Experiment results in Section IV show that our algorithm performs better than MCF [3] in terms of schedulability ratio and closely follows the optimal fluid algorithm MC-Fluid [2]. It is also shown that both MC-Sort and MC-Slope algorithm have a linearithmic (i.e., *nlogn*) complexity in the number of tasks n. Thus, both the proposed algorithms significantly improve schedulability when compared to MCF, with a marginal loss in time complexity.

Dual-rate fluid scheduling of MC task systems on a multi-core is not optimal; i.e., there are tasksets that are schedulable by some algorithm but are deemed to be not schedulable by the dual-rate MC-Fluid algorithm. In Section V we present a simple example that is not schedulable by any dual-rate fluid

scheduler whereas, a multi-rate (> 2 execution rates for each task) fluid scheduler successfully schedules it. Thus, exploring multi-rate fluid scheduling algorithms to further improve schedulability is a worthy research direction to consider, even though speed-up optimality has already been achieved.

Related Work: The concept of mixed-criticality systems was introduced by Vestal [4]. Several studies have been done on single-core MC scheduling in recent years; see [5] for review. As the recent trend in the chip industry is towards multi-cores, there have been some studies on multi-core MC scheduling as well. Initial work on multi-core MC scheduling is by Anderson et al. [6] is based on hierarchical scheduling in which they proposed a mix of partitioned and global approaches. A global fixed-priority scheduling algorithm based on response time analysis was proposed by Pathan et al. [7]. Li and Baruah [8] proposed a global scheduling algorithm by combining a multi-core fixed priority algorithm fpEDF and single-core virtual deadline based MC algorithm EDF-VD. Baruah et al. [9] also presented a partitioned scheduling algorithm based on EDF-VD and showed that partitioned scheduling performs better than global scheduling with respect to schedulability ratio. Rodriguez et al. [10] compared the performance of different partitioning heuristics for the partitioned EDF-VD algorithm. Guan et al. [11] extended the work on single-core demand bound function to multi-core and presented two enhancements to improve the overall system schedulability and heavy low-critical task schedulability. Ren et al. [12] proposed a partitioned scheduling algorithm based on compositional scheduling and task grouping that offers strong isolation for high-critical tasks and improved real-time performance for low-critical tasks. In contrast to the above studies, we focus on global fluid scheduling algorithms because they have been shown to have good, theoretically bounded, performance.

II. Background

A. System Model

MC scheduling problem is considered for an implicit-deadline sporadic task system scheduled on m identical cores. In this paper, we restrict ourselves to a dual-criticality system (namely LO and HI) as in [2] [3].

Tasks: We consider a sporadic taskset τ, in which each MC task τ_i is characterized by a tuple (T_i,C_i^L,C_i^H,X_i), where $T_i \in \mathbb{R}^+$ is the minimum release separation time, $C_i^L \in \mathbb{R}^+$ is the LO-criticality Worst-Case Execution Time (WCET), $C_i^H \in \mathbb{R}^+$ is the HI-criticality WCET (HI-WCET); we assume $C_i^L \leq C_i^H$ and $X_i \in \{LO, HI\}$ is the criticality level. We assume an implicit-deadline task model in which each task τ_i has a relative deadline equal to T_i.

Notation: We consider a dual-criticality sporadic taskset τ with n tasks. LO-criticality taskset τ_L and HI-criticality taskset τ_H are defined as $\tau_L \stackrel{\text{def}}{=} \{\tau_i \in \tau \mid Xi = LO\}$ and $\tau_H \stackrel{\text{def}}{=} \{\tau_i \in \tau \mid Xi = HI\}$. LO-criticality and HI-criticality utilization of a task τ_i is defined as $u_i^L \stackrel{\text{def}}{=} C_i^L/T_i$ and $u_i^H \stackrel{\text{def}}{=} C_i^H/T_i$ respectively. System-level utilizations of a taskset τ are defined as: $U_L^L \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_L} C_i^L/T_i$, $U_H^L \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_H} C_i^L/T_i$ and $U_H^H \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_H} C_i^H/T_i$.

MC Behaviour: If each task $\tau_i \in \tau$ signals completion before exceeding its LO-WCET, then the system is said to be in LO-criticality behaviour or LO-mode. If any HI-task $\tau_i \in \tau_H$ signals completion after executing beyond its LO-WCET and before exceeding its HI-WCET, then the system is said to be in HI-criticality behaviour or HI-mode. Mode change represents the change in criticality level of the system from LO to HI. System initially starts in LO-mode, and switches to HI-mode at the earliest time instant when any HI-task executes beyond its LO-WCET without signalling completion. After mode switch all LO-tasks are discarded by the system. If at any point a LO-task executes beyond its LO-WCET in LO-mode or a HI-task executes beyond its HI-WCET in HI-mode, then the system behaviour is said to be erroneous.

MC Schedulability: A taskset τ is said to be MC schedulable by a scheduling algorithm if,

- In LO-mode, each instance of each task in τ is able to complete LO-WCET execution within its deadline, and
- In HI-mode, each instance of each HI-task in τ is able to complete HI-WCET execution within its deadline.

B. Dual-Rate Fluid Scheduling

Dual-rate fluid scheduling algorithm was designed to schedule dual-criticality implicit-deadline sporadic task systems on an identical multi-core platform. Dual-rate fluid scheduling can be summarized as follows:

- In LO-mode, each task $\tau_i \in \tau$ executes at a constant rate θ_i^L (where $\theta_i^L \in (0, 1]$).
- After mode switch, all tasks in τ_L are discarded immediately and each HI-task starts executing at a constant rate θ_i^H (where $\theta_i^H \in [\theta_i^L, 1]$).

MC-Fluid uses the criticality-dependent execution rates θ_i^L and θ_i^H to schedule each task τ_i [2]. MC-Fluid formulates an optimization problem to determine the execution rates where, θ_i^H is the solution to the optimization problem,

$$
\begin{aligned}
minimize \quad & \sum_{\tau_i \in \tau_H} \frac{u_i^L(u_i^H - u_i^L)}{\theta_i^H - u_i^H + u_i^L} && \leq m \\
subject \;\; to \quad & \sum_{\tau_i \in \tau_H} \theta_i^H - m && \leq 0 \\
& \forall \tau_i \in \tau_H, \quad -\theta_i^H + u_i^H && \leq 0 \\
& \forall \tau_i \in \tau_H, \quad \theta_i^H - 1 && \leq 0
\end{aligned}
$$

The θ_i^H values are determined by solving the convex optimization problem. The speed-up factor of MC-Fluid is shown to be 4/3, which is optimal among all multi-core MC scheduling algorithms [3].

MCF is a simplified variant of MC-Fluid that has linear run-time complexity in the number of tasks n [3]. MCF, like MC-Fluid, tries to compute the execution rates θ_i^L and θ_i^H to meet the MC schedulability condition. It can be summarized as follows:

- Compute ρ where,

$$\rho \leftarrow max\left\{(\frac{U_L^L + U_H^L}{m}), (\frac{U_H^H}{m}), \underset{\tau_i \in \tau_H}{max}\{u_i^H\}\right\}$$

- **If** $\rho \leq 1$ compute θ_i^H and θ_i^L **else** declare failure;

$$\theta_i^H \leftarrow \frac{u_i^H}{\rho}, \text{for all } \tau_i \in \tau_H$$

$$\theta_i^L \leftarrow \begin{cases} \frac{u_i^L.\theta_i^H}{\theta_i^H - u_i^H + u_i^L}, & \text{if } \tau_i \in \tau_H, \\ u_i^L, & \text{otherwise} \end{cases}$$

- **If** $\sum_{\tau_i \in \tau} \theta_i^L \leq m$ declare success **else** declare failure

The key difference between MCF and MC-Fluid is that MCF uses a simple rate assignment strategy with linear run-time complexity as opposed to the convex optimization framework of MC-Fluid. Although MCF has lower schedulability performance when compared to MC-Fluid in experiments, it is also shown to have an optimal speed-up bound of 4/3.

A taskset τ is said to be MC schedulable under dual-rate fluid scheduling iff

$$\begin{aligned} \forall \tau_i \in \tau, \quad & \theta_i^L \geq u_i^L, \\ \forall \tau_i \in \tau_H, \quad & \frac{u_i^L}{\theta_i^L} + \frac{u_i^H - u_i^L}{\theta_i^H} \leq 1, \\ & \sum_{\tau_i \in \tau} \theta_i^L \leq m, \\ & \sum_{\tau_i \in \tau_H} \theta_i^H \leq m. \end{aligned}$$

III. Proposed Rate Assignment Strategies

In this section, we present two new algorithms, namely MC-Sort and MC-Slope, for computing the execution rates θ_i^L and θ_i^H for each task τ_i. MC-Sort and MC-Slope algorithm can be summarized as follows:

- Compute θ_i^H for all $\tau_i \in \tau_H$ using MC-Sort/MC-Slope HI-rate assignment,
- **If** $\sum_{\tau_i \in \tau_H} \theta_i^H \leq m$ compute θ_i^L **else** declare failure;

$$\theta_i^L \leftarrow \begin{cases} \frac{u_i^L.\theta_i^H}{\theta_i^H - u_i^H + u_i^L}, & \text{if } \tau_i \in \tau_H, \\ u_i^L, & \text{otherwise} \end{cases}$$

- **If** $\sum_{\tau_i \in \tau} \theta_i^L \leq m$ declare success **else** declare failure.

A. MC-Sort Algorithm

MC-Sort rate assignment algorithm sorts all the HI-tasks based on their HI-WCET values, and assigns the maximum possible rate in HI-mode (Θ_i^H) for each HI-task in that order. The detailed steps of this strategy for assigning the HI-rates are given in Algorithm 1.

The run-time complexity of the MC-Sort algorithm is linearithmic in the number of tasks in τ. Initial assignment of θ_i^H can be done in a single pass through all the HI-tasks in the system. Sorting θ_i^H can be done in $O(nlogn)$ time, where n is the total number of tasks in the system. Final assignment of θ_i^H and θ_i^L can be done in linear time, and therefore the overall run-time complexity of MC-Sort is $O(nlogn)$.

Algorithm 1 MC-Sort HI-rate assignment

Input: τ_H, m and for each θ_i^H assign an initial value of $\frac{u_i^H}{max\left\{(\frac{U_H^H}{m}), u_i^H\right\}}$

Output: θ_i^H for each $\tau_i \in \tau_H$

1: Sort τ_H in decreasing order of u_i^H
2: **for** j := 1 to length(τ_H) **do**
3: **if** $(m - \sum_{\tau_i \in \tau_H} \theta_i^H) > 0$ and $u_i^H \neq u_i^L$ **then**
4: **if** $(m - \sum_{\tau_i \in \tau_H} \theta_i^H) \geq (1 - \theta_i^H)$ **then**
5: Update θ_i^H to 1.0
6: **else if** $(m - \sum_{\tau_i \in \tau_H} \theta_i^H) < (1 - \theta_i^H)$ **then**
7: Update θ_i^H to $\theta_i^H + (m - \sum_{\tau_i \in \tau_H} \theta_i^H)$
8: **else**
9: Break
10: **end if**
11: **end if**
12: **end for**

If algorithm MC-Sort successfully determines the rates, then the schedule resulting from using these rates will result in a correct MC-scheduling strategy. From Line 5, it is evident that the individual tasks' execution rate never exceeds 1. The condition in Line 3 ensures that the total execution rate of all tasks do not exceed the system capacity. The initial assignment for θ_i^H, and θ_i^L computation are the same as in MCF. For correctness proof please refer to theorem 1 in [3].

B. MC-Slope Algorithm

Another rate assignment algorithm with linearithmic run-time complexity called MC-Slope is presented in this section. MC-Slope assigns execution rates based on the rate of change of task τ_is component ($O(\theta_i^H)$) in the objective function of the optimization problem in [2]. The rate of change of objective function $O(\theta_i^H)$, $R(\theta_i^H)$, is defined as follows.

$$\begin{aligned} O(\theta_i^H) &= \frac{u_i^L(u_i^H - u_i^L)}{\theta_i^H - u_i^H + u_i^L} \\ R(\theta_i^H) &= \frac{d^2 O(\theta_i^H)}{d\theta_i^{H2}} \\ &= \frac{2.u_i^L(u_i^H - u_i^L)}{(\theta_i^H - u_i^H + u_i^L)^3} \end{aligned} \tag{1}$$

Algorithm 2 below then gives the HI-rate assignment strategy of MC-Slope. Line 1 in Algorithm 2 sorts all the HI-criticality tasks in increasing order of $R(u_i^H)$. It considers one HI-task (τ_j) at a time from this sorted list (Line 2). For each HI-task (τ_i) such that $i > j$, it assigns rate θ_i^H such that $R(\theta_i^H)$ is decreased to match $R(\theta_j^H)$. This can be computed using Equation (1) by considering θ_i^H as the unknown quantity with a fixed value for θ_i^H. If the assigned θ_i^H rate is greater than 1, θ_i^H will be updated to 1. Line 5 checks if the assigned rates are feasible, and if not, then the above process is repeated

Algorithm 2 MC-Slope HI-rate assignment

Input: τ_H, m and for each θ_i^H assign an initial value of u_i^H
Output: θ_i^H for each $\tau_i \in \tau_H$

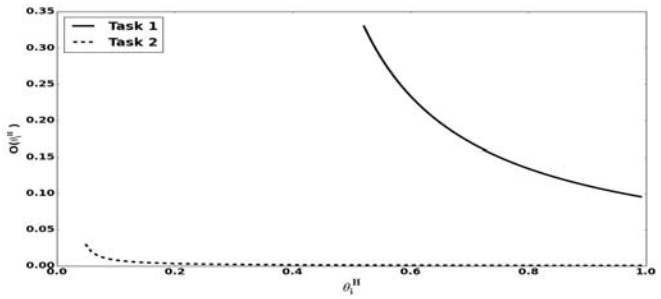
1: Sort τ_H in increasing order of $R(\theta_i^H)$ at $\theta_i^H = u_i^H$
2: **for** j := 1 to length(τ_H) **do**
3: Compute θ_i^H for each $\tau_i \in \tau_H$ s.t. $i > j$ $R(\theta_i^H) = R(\theta_j^H)$
4: Update θ_i^H to 1.0 if $\theta_i^H > 1$ for each $\tau_i \in \tau_H$
5: **if** $\sum_{\tau_i \in \tau_H} \theta_i^H \leq m$ **then**
6: Break
7: **else**
8: Continue
9: **end if**
10: **end for**
11: Slack = m - $\sum_{\tau_i \in \tau_H} \theta_i^H$
12: Sum_$O(\theta_i^H)$ = $\sum_{\tau_i \in \tau_H, \theta_i^H \neq 1} O(\theta_i^H)$
13: **for** i := length(τ_H) to 1 **do**
14: **if** $(\theta_i^H \neq 1)$ and (Slack > 0) **then**
15: Update θ_i^H to $\theta_i^H + \frac{Slack * O(\theta_i^H)}{Sum\_O(\theta_i^H)}$
16: **if** $\theta_i^H \geq 1$ **then**
17: Update θ_i^H to 1.0
18: **end if**
19: **end if**
20: **end for**

for the next HI-task τ_j in the sorted list. In the worst-case, the for loop exits without any modification to the initial assignment of θ_i^H, which is always feasible. Line 11 computes the remaining slack in the system after the above assignment. Lines 13-18 allocates this remaining slack to all the HI-tasks proportionately, based on their updated $O(\theta_i^H)$ values.

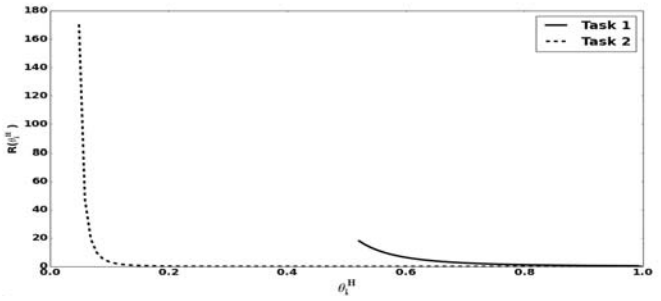
Objective function $O(\theta_i^H)$ and the rate of change of objective function $R(\theta_i^H)$ of a sample taskset is plotted against θ_i^H in Figure 1. Task 1 has a larger $O(\theta_i^H)$ value compared to task 2, whereas, task 2 has larger $R(\theta_i^H)$. MC-Slope assigns θ_i^H rate to task 2 first, until its $R(\theta_i^H)$ value becomes equal to that of task 1. If slack remains after their $R(\theta_i^H)$ becomes equal, it proportionately allocates the slack based on their updated $O(\theta_i^H)$ values. The rationale behind MC-Slope strategy is that the algorithm tries to minimize the total objective function by allocating a larger portion of the execution rate to tasks with faster decreasing $R(\theta_i^H)$.

The schedule resulting from using the execution rates computed by the MC-Slope algorithm constitutes a correct MC-scheduling strategy. Lines 4 and 17 in Algorithm 2 ensure execution rates do not exceed 1. Lines 5 and 14 guarantees that the total execution rate of tasks does not exceed the system capacity. The θ_i^L computation is the same as in other fluid algorithms. MC-Slope algorithm declares success if the total LO-rate of the tasks do not exceed the processing capacity.

MC-Slope algorithm has a run-time complexity of $O(nlogn)$ in the number of tasks in τ. Sorting $R(\theta_i^H)$ in MC-Slope algorithm can be done in $O(nlogn)$ time complexity. Selecting a task with least $R(\theta_i^H)$ that satisfies the condition



(a) Objective Function - O(θ_i^H)



(b) Rate of change of objective function - R(θ_i^H)

Fig. 1: MC-Slope Algorithm

in Line 5 can be done using binary search. Thus, the outer FOR loop of Line 2 runs at most $O(logn)$ times. Computing θ_i^H in Line 3 consumes linear time in the number of tasks n. Proportionately allocating the remaining slack to the tasks can also be done in linear time. The overall complexity of the MC-Slope algorithm is thus $O(nlogn)$.

IV. Experiments and Results

Experiments with randomly generated tasksets are conducted to compare the performance of the proposed algorithms with the existing fluid algorithms MCF [3] and MC-Fluid [2]. In this section, we first present the experiment setup and then discuss the results.

A. Experiment Setup

Task set generation: Our experiments are carried out for a dual-criticality implicit-deadline task systems scheduled on an m-core platform. We use the same approach for generating random tasksets as in earlier studies [3]. The task parameters used in our experiments are described as follows:

1) $U^B \in [0.1, 0.15, ..., 1.0]$ denotes the normalized system utilization in both LO and HI modes.
2) $P^H \in [0.1, 0.2, ..., 1.0]$ denotes the probability of a task to be HI-task.
3) Minimum and maximum individual task utilization u_{min} ($= 0.02$) and u_{max} ($= 0.90$).
4) $m \in \{2, 4, 8\}$ denotes the total number of cores.
5) T_i, the period of task τ_i is drawn uniformly at random from $[20, 300]$.
6) P_i is drawn uniformly at random from $[0, 1]$. If $P_i < P^H$, then X_i = HI else X_i = LO.
7) Task utilization u_i is drawn from the range [u_{min},u_{max}]. If X_i = HI, then $u_i^H = u_i$ else $u_i^L = u_i$. If X_i = HI,

$u_i^L = u_i^H/R$, where R is an integer drawn uniformly at random from the range $[1, 4]$.

8) Execution requirements C_i^L and C_i^H are derived as $\lceil u_i^L * T_i \rceil$ and $\lceil u_i^H * T_i \rceil$ respectively.

The steps $5-8$ are repeated to generate tasks until the system utilization condition $\max\{\frac{U_L^L+U_H^L}{m}, \frac{U_H^H}{m}\} \leq U^B$ is met. Once the condition is violated, the last generated task is discarded. If the resulting taskset has a normalized utilization between U^B - 0.05 and U^B, then the taskset is accepted, else the taskset is discarded and the procedure is repeated again. We evaluate the performance of four algorithms MC-Fluid, MC-Slope, MC-Sort and MCF for each successfully generated taskset.

B. Results

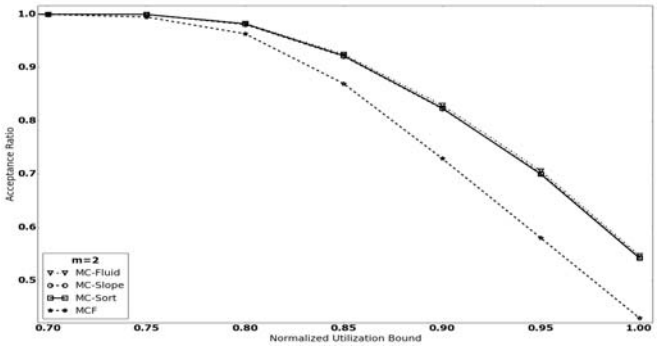
Figures 2a-2c show the acceptance ratios of the four algorithms i.e., fraction of schedulable tasksets, versus normalized average utilization U^B over varying $m \in \{2, 4, 8\}$ and fixed P^H $(= 0.5)$. Each data point in the figure corresponds to $10,000$ tasksets. The results show that both MC-Sort and MC-Slope outperform MCF, and the difference in schedulability is marginal when compared to MC-Fluid.

Figure 3a shows the effect of varying P^H values over the weighted acceptance ratio. Weighted acceptance ratio is defined as $WAR(\mathbb{S}) = \frac{\sum_{U^B \in \mathbb{S}} (AR(U^B) X U^B)}{\sum_{U^B \in \mathbb{S}} U^B}$ where, $\mathbb{S}$ is the set of U^B values and AR(U^B) is the acceptance ratio for a specific value of the normalized utilization U^B. All the algorithms perform well at the extreme probability values as tasksets contain only either LO or HI tasks. It can be seen that both MC-Sort and MC-Slope outperforms MCF algorithm for all the P^H values. The performance of MC-Slope is marginally better than that of MC-Sort for P^H values greater than 0.6. This is as expected because MC-Slope uses the rate of change in the objective function to determine HI-rates, whereas MC-Sort simply uses the execution requirement in HI-mode.

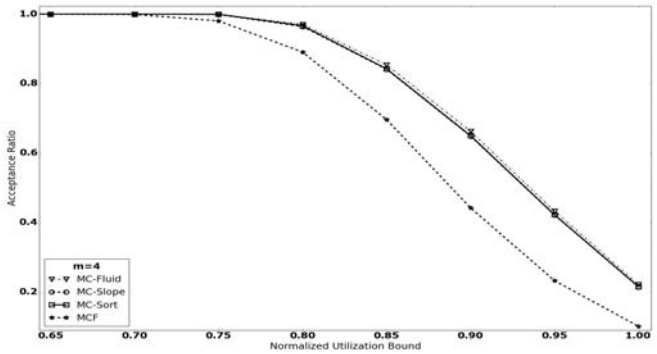
Figure 3b shows the results of weighted acceptance ratio for varying range of u_i^H/u_i^L $(\in [1, 1.5], [1.5, 2.0], [2.0, 2.5], [2.5, 3.0], [3.0, 3.5], [3.5, 4.0])$ with fixed P^H $(= 0.5)$ and u_{max} $(= 0.90)$ values. Only the upper bound of the ranges is presented in the plot. It can be seen that as the ratio u_i^H/u_i^L becomes larger there is a drop in the performance of all the algorithms. However, unlike MCF, MC-Slope and MC-Sort continue to perform exceedingly well in comparison to the optimal MC-Fluid.

V. Discussions and Future Work

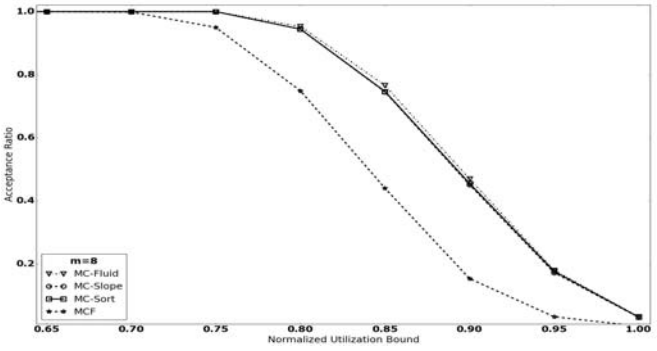
MC-Fluid is shown to be an optimal rate assignment strategy for a dual-rate dual-criticality task systems [2]. That is, if there exists an assignment of θ_i^H and θ_i^L for all the tasks that satisfies MC schedulability condition, then MC-Fluid is guaranteed to find such an assignment. As fluid schedules are not implementable on actual computing platforms due to their fractional allocations, MC-DP-Fair translates the fluid schedules to non-fluid ones without any loss in performance [2]. MCF on the other hand, presents a sub-optimal rate assignment strategy with linear complexity. Both these strategies have been shown to result in optimal speed-up bounds.



(a) m = 2



(b) m=4



(c) m=8

Fig. 2: Comparison of acceptance ratio

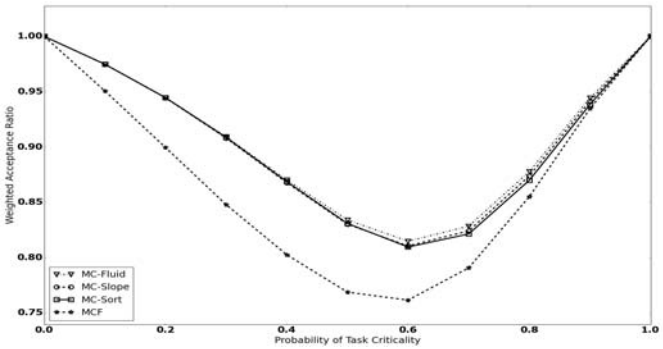
As a part of the future work, we plan to derive the speed-up bounds for the two proposed algorithms, and also identify a linearithmic complexity algorithm with an optimal rate assignment strategy.

Dual-rate fluid scheduling is not optimal among all fluid scheduling algorithms. In particular, algorithms with more than two rate assignments per task can further improve schedulability. Consider a dual-core system with the periodic taskset τ and its execution rate as shown in Table I. Taskset in the table is not MC-schedulable with the dual-rate assignment θ_i^L and θ_i^H given by the MC-Fluid algorithm. We can check that $\sum_{\tau_i \in \tau} \theta_i^L$ is greater than 2.

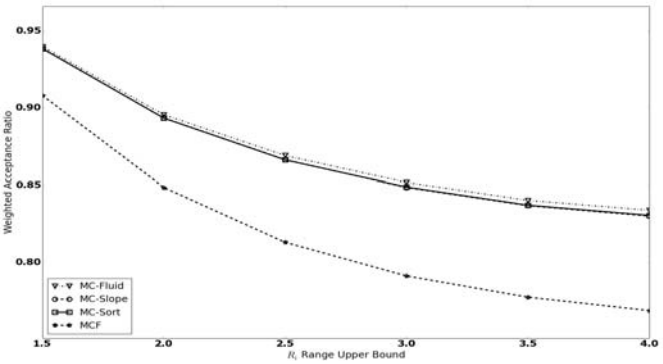
Each MC task is characterized by a sequence of job releases. A job of a task τ_i is said to be a carry-over job, if it is released

TABLE I: Example taskset and its rate assignment

Tasks						MC-Fluid		Proposed multi-rate model			
	C_i^L	C_i^H	T_i	u_i^L	u_i^H	θ_i^L	θ_i^H	δ_i^L	δ_i^{H*}	δ_i^C	δ_i^H
τ_1	1.5	4	5	0.3	0.8	0.641	0.939	0.641	0.939	-	0.8
τ_2	2.8	4.9	7	0.4	0.7	0.700	0.700	0.700	0.700	0.700	0.7
τ_3	3.5	10.5	35	0.1	0.3	0.224	0.360	0.209	0.360	0.499	0.3
τ_4	15.75	-	35	0.45	-	0.450	-	0.450	-	-	-
$\sum$						2.015	1.999	2.00	1.999	1.199	1.80



(a) Varying probability of a task to be HI-Criticality



(b) Varying u_i^H/u_i^L ranges

Fig. 3: Comparison of weighted acceptance ratio

before the mode switch and has not completed its execution until the mode switch.

We plan to improve the fluid scheduling framework by considering multiple rates for each task. The jobs of a HI-task τ_i released before mode switch execute with δ_i^L in LO-mode and the jobs released after mode switch execute with δ_i^H in HI-mode. Here, $\delta_i^H = u_i^H$ as no jobs require more than its HI-utilization for completion. The carry-over jobs of HI-tasks execute with δ_i^{H*} from mode switch until the earliest time instant at which a carry-over job of any HI-task complete its execution, and then execute with δ_i^C ($\geq \delta_i^{H*}$) until its completion. Figure 4 represents the proposed multi-rate model in which each task executes with more than two rates.

By assigning multiple rates to the HI-tasks, the taskset in Table I is shown to be MC-schedulable. Let us assume task τ_1 initiates the mode switch as shown in Figure 4. It is sufficient for jobs of τ_1 released after mode switch to execute with $u_1^H (= 0.8)$. The difference in δ_1^{H*} and $u_1^H (= 0.939 - 0.8)$ can be used for executing other HI-tasks; in this case, for task τ_3. By allowing additional execution rate for task τ_3 in HI-mode, we can bring down its LO-mode rate. Now we can check that

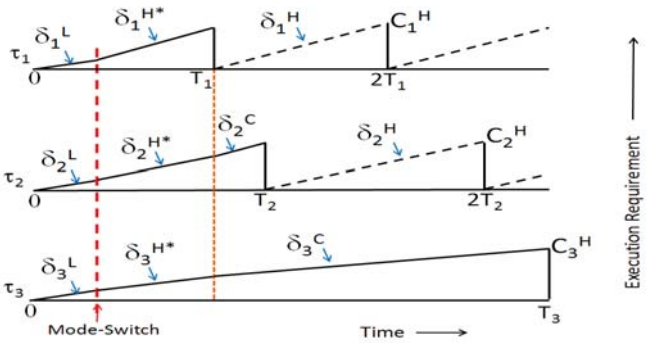


Fig. 4: Proposed multi-rate model

$\sum_{\tau_i \in \tau} \delta_i^L$ is less than or equal 2. Therefore, by assigning more than two rates to each task it is possible to schedule tasksets that are deemed to be not MC-schedulable by dual-rate fluid scheduling algorithms.

REFERENCES

[1] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *Real-Time Systems (ECRTS), 22nd Euromicro Conference on*, July 2010.

[2] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors," in *Real-Time Systems Symposium (RTSS), 35th IEEE International*, Dec 2014.

[3] S. Baruah, A. Easwaran, and Z. Guo, "MC-Fluid: simplified and optimally quantified," in *Real-Time Systems Symposium (RTSS), 36th IEEE International*, Dec 2015.

[4] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium, 28th IEEE International*, Dec 2007.

[5] A. Burns and R. I. Davis. (2013) Mixed Criticality Systems - A Review. http://www-users.cs.york.ac.uk/burns/review.pdf.

[6] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification, 2009*, Apr 2009.

[7] R. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, July 2012.

[8] H. Li and S. Baruah, "Outstanding paper award: Global mixed-criticality scheduling on multiprocessors," in *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, July 2012.

[9] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[10] P. Rodriguez, L. George, Y. Abdeddaim, and J. Goossens, "Multi-criteria evaluation of partitioned edf-vd for mixed-criticality systems upon identical processors," in *Workshop on Mixed Criticality Systems (WMC), 2013*, December.

[11] C. Gu, N. Guan, Q. Deng, and W. Yi, "Partitioned mixed-criticality scheduling on multiprocessor platforms," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2014.

[12] J. Ren and L. T. X. Phan, "Mixed-criticality scheduling on multiprocessors using task grouping," in *Real-Time Systems (ECRTS), 27th Euromicro Conference on*, July 2015.

A protocol for mixed-criticality management in switched Ethernet networks

Olivier CROS, Laurent GEORGE
Université Paris-Est, LIGM / ESIEE, France
cros@ece.fr,lgeorge@ieee.org

Xiaoting LI
ECE Paris / LACSC, France
xiaoting.li@ece.fr

***Abstract*—In real-time industrial networks, providing timing guarantees for applications of different criticalities often results in building separate physical infrastructures for each type of network at the price of cost, weight and energy consumption. Mixed-Criticality (MC) is a solution first introduced in embedded systems to execute applications of different criticality on the same platform. In order to apply MC scheduling to off-the-shelves Switched Ethernet networks, the key issue is to manage the criticality change information at the network level. The objective of this work is to propose a criticality change protocol for MC applications communicating over Switched Ethernet networks. The protocol relies on a global clock synchronization, as provided by the IEEE-1588v2 protocol, and a real-time multicast based upon it, to preserve the consistency of the criticality level information stored in all the Ethernet switches. We characterize the worst-case delay of a criticality change in the network. Simulation results show how the criticality change delay evolves as a function of the network load.**

I. Introduction

Nowadays, highly-constrained industrial systems found in defense, public transports or home automation have increasing needs in terms of reliability and performance. It is a common situation for such systems to integrate several independent network architectures in order to transmit, in each network, messages of different criticality (e.g. in bus: passenger information, mechanical control information, speed, etc.) such that each system can be certified in isolation. This solution is very expensive in terms of cost, weight and hence in terms of energy consumption, as each network must have its own infrastructures, materials and wires. For example, the mechanical functions, trajectory control and the passenger information are often treated in separated infrastructures inside a public bus, with different dedicated materials.

One solution to this problem is the mixed-criticality (MC) scheduling approach first proposed in the context of uniprocessor and multiprocessor systems [1]. It executes several applications of different criticalities on the same platform by adapting certification effort to the level of assurance needed at a given criticality level. Since a networked system is an interconnect system for applications of different criticalities, the objective of this work is to study how to manage criticality level information in networked systems.

With MC scheduling, each task is characterized by the maximum criticality level it is allowed to execute. A task can be non-critical, critical for the mission (mission-critical), critical for the safety of the vehicle (mechanical-critical), or for the safety of its occupants (safety-critical). Each criticality level must provide guarantees on end-to-end transmission delays, especially for high critical tasks. The more critical a task is, the more reliable the guarantee should be.

In the real-time network context, we focus on how to integrate criticality management in networked systems. The first point is to bound the number of criticality levels we use. Baruah [2] showed that the complexity of MC scheduling problems is NP-hard in the strong sense. In order to limit the complexity of our architectures, we focus on a two-level criticality network, as presented in [3]. These two criticality levels are called low-critical (LO) and high-critical (HI) levels: only a set of predefined messages can be transmitted in HI criticality level, whereas all messages can be transmitted in LO criticality level.

The main goal of the criticality management in networked systems consists in providing Quality of Service (QoS) guarantees (in terms of worst case end-to-end transmission delays), specially for high critical messages. In this context, we focus on a method to grant the consistency of the criticality level information in a Switched Ethernet network, to ensure bounds on end-to-end transmission delays of messages as a function of the criticality of information sent.

In uniprocessor and multiprocessor systems, the problem of characterizing the impact of low-criticality tasks when a criticality change from LO to HI has been considered by bounding the demand of carry-in jobs.

Assuring deterministic communications in networked systems implies to be able to bound the end-to-end delay of all messages. In [4], we proposed a tool to evaluate the worst case end-to-end delay of any message sent on a Switched Ethernet network relying on a global clock-synchronization protocol. In this paper, we also consider a clock synchronized network, synchronized with the IEEE-1588v2 synchronization protocol and its implementation in Precision Time Protocol (PTP). On top of the clock synchronization, we build a reliable multicast to consistently switch the criticality level on all the nodes of the network.

In this paper, first we present in II the network model studied in this work. In III, we illustrate a link utilization problem based on an example, and then we present the importance of managing MC in network context. We propose a MC management protocol in IV. Finally, we show by simulation the performances of this protocol in V, and conclude this paper

as well as future perspectives of this paper in VI.

II. Network Model

A. Mixed-Criticality

In this paper, we consider a tree-based topology as those found in application domains like avionics systems and recent and futur public transport systems [5]. The network is composed of a set of interconnected nodes, all organized according to a tree-based structure with one final collecting node denoted the sink node. An example of such topology is the one showed in figure 1, with S_4 as the sink node.

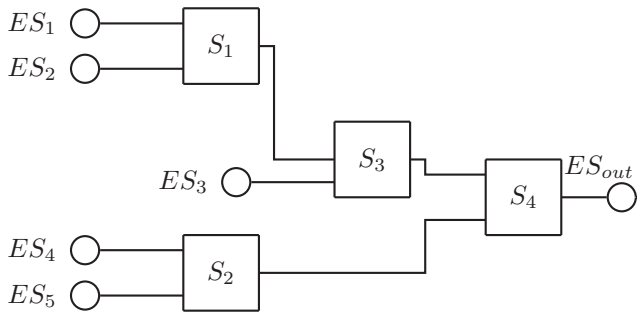


Fig. 1. Centralized Network architecture

Our goal is to propose a criticality level management protocol in a tree-based Switched Ethernet network topology. The sink node is in charge of stroring the network criticality information. All the nodes of the network have a local copy of the network criticality level information. The protocol we propose should maintain the consistency of the criticality level information in all the nodes of the network in the case of a criticality switch.

We then have two cases:

- If the network criticality level is LO, the first node sending a request to the sink-node, to change the criticality level to HI will result in a multicast sent by the sink node the all the nodes of the network with the new network criticality level. All the nodes receiving this message should update their local criticality level information so as to keep it consistent after a criticality mode switch.
- If the network criticality is HI, a switch to LO criticality level can happen only if the sink node has received from all the other nodes a request to switch to LO mode.

A simple multicast is not sufficient to guarantee the consistency of the network criticality level information in all the nodes. We introduce a real-time reliable multicast protocol in section IV-C as one solution to this consistency problem. We caracterize the maximum time needed to switch the network criticality level from LO to HI, from the request of the first node willing to change the criticality level to HI, to the time all nodes are allowed to change their criticality level (after receiving the reliable multicast from the sink node).

B. Notations and main hypothesis

In MC systems, representing different levels of criticality inside a system is mostly based on a choice between two different hypotheses : either a message has a dedicated worst-case transmission time (WCTT) for each criticality level, which means that the flow of data sent in the entry points are longer in the case of HI modes. For example, as a plane is landing, it might need more precise evaluation of the altitude. It means that the altitude sensors will send more complete, and so longer, data values.

The second hypothese is not based on longer WCTT, but on a more frequent messages. Each message has now two different periods, one for LO level and one for HI level. It corresponds to increasing the number of measures during a critical phase : for example, during landing, the measure of speed or altitude could have to be more frequent. We consider this case in our analysis.

A network is a set of interconnected switches communicating through full-duplex links connecting end-systems. On each link, we can send one or several flows v_i, and each flow produces several messages. Each flow v_i is represented as a 3-tuple $v_i = \{\mathcal{P}_i, C_i, \vec{T_i}\}$ where:

- $\mathcal{P}_i$ is the path of nodes followed by any message of v_i, starting from a source node to a destination node. We consider this path as statically defined by the designer.
- C_i defines the WCTT of any message of v_i sent in LO or HI network criticality level.
- Its period is defined as $\vec{T_i} = \{T_i^{LO}, T_i^{HI}\}$. It is a vector of different periods of the flow, corresponding to LO-critical and HI-critical period (we assume two network criticality levels in this paper).
- In the case where a flow can only be sent in LO criticality level, that means that the flow will not be sent by a switch when the network criticality level is HI.

Furthermore, we suppose that:

- Each message is independent from each other
- All message transmissions are non-preemptive
- All the switchs use a Fixed Priority (FP) scheduling with FIFO scheduling in a specific FP queue, denoted as FP/FIFO scheduling.

III. Problem statement

A. An example

We consider the case of a simple network composed of one switch S (denoted S_M is it support MC), scheduling flows with FIFO scheduling and having three entry ports $ES1$, $ES2$ and $ES3$ respectively receiving flows v_1, v_2 and v_3, with the following parameters:

Flow	T_i^{LO} (μs)	T_i^{HI} (μs)	C_i (μs)	u_i^{LO}	u_i^{HI}
v_1	500	250	100	0.2	0.4
v_2	500	250	100	0.2	0.4
v_3	300	-	100	0.33	-

Imagine a scenario where all the three flows are transmitted in the LO criticality level. The LO-utilization (u^{LO}) of the network at the most loaded node S_4 is then $u^{LO} = u_1^{LO} + u_2^{LO} + u_3^{LO} = 0.73$. Then flow v_1 and flow v_2 increase their workloads by reducing the periods of messages due to certain emergencies. Then flow v_1 and flow v_2 are transmitted

in HI criticality level. Supposing that there is no criticality management, now the utilization at the node S_4 should be $u_1^{HI} + u_2^{HI} + u_3^{LO} = 1.13$. It means that S_4 is overloaded in a mixed mode with both criticality levels.

We focus on the impact of such an overloaded link on transmission delays when criticality levem is not managed by the nodes. We suppose that, at $t = 100\ \mu$s, the system becomes high-critical : v_1 and v_2 start emitting messages according to T_i^{HI} and no more to T_i^{LO}. Basically, this results in a strong increase in the transmission delay for the frames of v_1 (see S in figure 2).

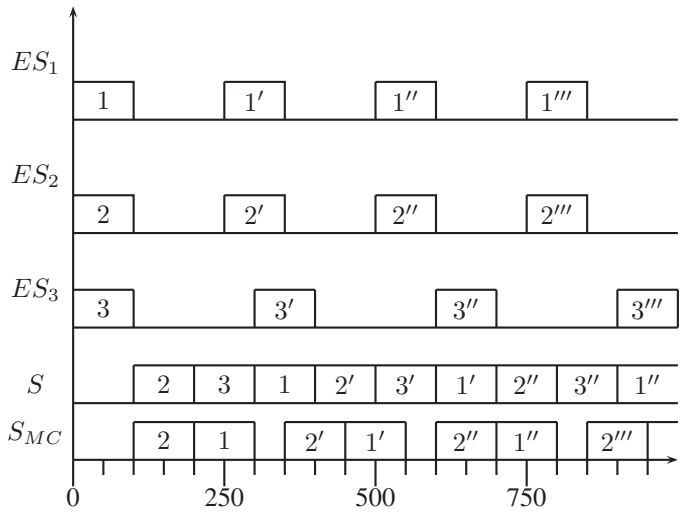


Fig. 2. Transmission delay with criticality management

We can observe that the transmission delay of messages from v_1 increases drastically with time. In fact, we can easily compute that the waiting delay of each message from v_1 in the entry point of S_4 increases of 50 ms at each new emission. Thus, in classical Ethernet context, switches do not have input buffers of infinite size, so a too high waiting delay can result in dropping out a message, and then loss of data.

If the network supports MC management (see S_{MC} in figure 2), we can observe that the transmission delay of HI-critical messages is constant and that criticality management allowed us to fix the overload problem. We show next a protocol to implement MC in network context.

IV. A CRITICALITY-CHANGE PROTOCOL

A. A two-phase protocol

Transmitting and managing criticality level inside network topology implies two different conditions. First, we need to assure that all nodes in the network have the same criticality level. Secondly, in case of criticality level switch, we have to be sure that all nodes change their local criticality level information preserving the consistency the network criticality level information.

Like we showed in II, criticality level is managed by a central entity (the sink node). It means that, even if each node has its own criticality level, it must be synchronized at all times with the one of the sink node. To assure this condition, we propose our MC managing protocol. It consists in assuring the consistent change of criticality level in all the nodes of a network topology. For this, we use a reliable multicast to ensure a total order (updates are scheduled in the same order in all nodes) for the update of criticality level information in all nodes. This preserves the serialisability of the criticality updates hence the consistency of the criticality level information [6].

This MC managing protocol works in two phases for a LO to HI criticality change. The node n in charge of initiating this criticality level change sends a criticality change request to the sink node. We name it the switch-criticality call (SCC) message, the delay needed to send this message from node n to the sink is denoted I^n_{delay}. Next, we need to send a criticality switch message with the new criticality level to all the nodes (except the sink node) of the network. This is the reliable multicast phase initiated by the sink node that send a timestamp of his local clock in the multicast message (recall that we assume a global clock synchronization). The delay needed to send and receive this message from the sink node to node n is denoted M^n_{delay}.

The total criticality switch delay S_{delay} in the network $\mathcal{N}$ can be computed by :

$$S_{delay} = \max_{n \in \mathcal{N}} (I^n_{delay} + M^n_{delay}) \tag{1}$$

Upon reception of a criticality switch message, each node will have to localy determine when to to the criticality switch. We now how yo characterize the criticality switch delay in the case of a LO to HI criticality switch.

B. Switch-criticality call

Suppose that the network criticality is LO. Calling for a criticality level change to HI consists in sending a specific message from a node n in the network requiring this criticality level change, and transmitting it to the central node responsible of criticality management. This SCC message is transmitted with the highest priority (except PTP messages), dedicated for configuration messages. Nevertheless, it can be delayed by other messages in the network: either by PTP synchronization messages, or by other messages due to the non-preemptive effect (a message even with the lowest priority cannot be stopped once its transmission has started).

The SCC message is considered as a new message in the network. It means that it is defined by its own path $\mathcal{P}_c$ from n to the central node S_h and its WCTT C_c. We consider that only one SCC message is sent by a switch (the first one received). If another nodes initiate another SCC message with the same criticality level, the switch receiving it will discard the message.

In other words, computing the delay needed to transmit the SCC consists in evaluating the delay needed for the message to be tranmitted from one node n to the sink node. To do this, we use the trajectory approach, presented in [7]. The SCC

transmission delay (noted as I^n_{delay}) can be computed by the general trajectory approach expression. Given that the call is done by a node n in the network, we apply the trajectory approach computation method to the switch-criticality call message c. This gives us the following result :

$$I^n_{delay} = \sum_{\substack{j \in hp_c \\ \mathcal{P}_c \cap \mathcal{P}_j \neq \emptyset}} \left(1 + \left\lfloor \frac{W^{last_{c,j}}_{c,t} - M^{first_{c,j}}_i + A_{c,j}}{T^{LO}_j} \right\rfloor\right)^+ \cdot C_j \quad (2)$$

$$+ \sum_{\substack{j \in sp_c \\ \mathcal{P}_c \cap \mathcal{P}_j \neq \emptyset}} \left(1 + \left\lfloor \frac{t + S^{first_{c,j}}_{max_c} - M^{first_{c,j}}_i + A_{c,j}}{T^{LO}_j} \right\rfloor\right)^+ \cdot C_j \quad (3)$$

$$+ \sum_{h \in \mathcal{P}_c \setminus \{n\}} \left(\max_{\substack{j \in sp_c \cup hp_c \\ h \in \mathcal{P}_j}} (C_j) \right) \quad (4)$$

$$+ (|\mathcal{P}_c| - 1) \cdot sl \quad (5)$$

$$+ \sum_{h \in \mathcal{P}_c} \delta^h_c \quad (6)$$

$$- C_c \quad (7)$$

- (2) is the delay induced by messages with higher priorities than the one of message c. These messages can delay c if they arrive at one shared output port with c during the maximized interval. For a higher-priority flow j , its maximized arrival jitter is $A_{c,j} = S^{first_{c,j}}_{max_c} - S^{first_{c,j}}_{min_c}$, where $S^h_{max_c}$ (resp. $S^h_{min_c}$) is the maximum (resp. minimum) delay of the message c from its source node $first_c$ till the output port h.
- (3) is the delay induced by messages with the same priority as message c. They are scheduled by FIFO policy. According to FIFO, messages arriving at the output port where $first_{c,j}$ (the first output port where they meet message c) during the maximized temporal interval $[M_{first_{c,j}}, t + S^{first_{c,j}}_{max_c}]$ can delay message c.
- (4) indicates the transmission delay of a message sequence including message c. This delay is maximized by considering the transmission time of the largest frame in the sequence at each output port along $\mathcal{P}_c$.
- (5) is the electronical latency induced by the transmission through wires (with sl, the electronical latency induced by the transmission between two nodes)
- (6) represents the delay induced by the non-preemptive effect (see [8]) of flows with a lower priority
- (7) finally, we substract C_c because the global transmission delay $W^{last_c}_{c,t}$ corresponds to the delay between the emission of c and its starting instant of emission in the final sink node

As SCC message has a high priority (dedicated for configuration messages), the only higher priority messages that will delay us are the PTP synchronization messages. We consider that PTP messages and switch-criticality call messages are the only one to be sent in this level of priority and higher, so there is no other message with the same priority as SCC. It means that we just need to compute the number of PTP messages generated in each node encountered along the path of the switch-criticality call.

Moreover, we suppose that PTP messages have their specific worst case transmission time, noted as C_{PTP}. But, for the sake of readability, we consider that C_c and C_{PTP} have the same value (the highest one of the two). Even if it is a pessimistic assumption, we can consider it true as PTP and SCC messages are both configuration messages, defined with a small and close number of bytes in Ethernet protocol. Considering these hypotheses, we obtain (with F_{PTP}, the PTP synchronization frequency) :

$$I^n_{delay} = F_{PTP} * \sum_{\substack{j \in hp_c \\ \mathcal{P}_c \cap \mathcal{P}_j \neq \emptyset}} \left(S^{first_{c,j}}_{max_c} - M^{first_{c,j}}_c + A_{c,j} \right) + \sum_{h \in \mathcal{P}_c} \delta^h_c + (|\mathcal{P}_n| - 1) * (sl + 2 * C_c) \quad (8)$$

C. Reliable multicast

In order to update the criticality level information from the central node to all the nodes in the network, we must be sure that the criticality switch order is received by all the nodes and is executed preserving the consistency of the criticality level information. Thus, in order to preserve the consistency of the current criticality level in all the nodes, we need to guarantee a total order in the criticality updates: two consecutive criticality switches have to be executed in the same order in all the nodes. A reliable real-time multicast is a method to send the same information to all nodes in a network providing total order for the update of the criticality level in all nodes. In [6], the authors show how to build a real-time reliable multicast provided that worst case messages end-to-end delays can be bounded from above. In this paper, we adapt their solution to the context of mixed criticality management with the trajectory approach to compute worst case end-to-end message delays.

To implement this, we need a deterministic computation of the transmission delay of the information. Since we can assure a bounded transmission delay for information in each physical link of the network, we will be able to provide guarantees on transmission delays in the whole network. That builds the determinism of the delay.

Suppose a network $\mathcal{N}$, composed of a set of nodes $\mathcal{S} = \{S_1, S_2, ...S_n, ES_1, ES_2, ...ES_m\}$. We note M^n_{delay} the delay needed by node n to receive the reliable real-time multicast information from the central node.

Now, we can compute the transmission delay of the criticality-switch order (which is a message) from the central node to all the nodes in the network. This multicast delay is noted as M_{delay}.

Even if we implemented PTP, we need to consider clock accuracy ϵ_i for each node i, as it impacts the reliable multicast protocol [6]. The multicast delay of the whole network can then be deduced by taking the maximum value of accuracy on any node. We have $\epsilon = \max_{n \in \mathcal{N}}(\epsilon_n)$. We obtain [6]:

$$M_{delay} = \max_{n \in \mathcal{N}}(M^n_{delay}) + \epsilon \quad (9)$$

M^n_{delay}, for a node n, is then computed by the addition of different elements: the switching latency sl induced by

electronical transmission between two nodes, and the WCTT of the criticality-switch message, noted as C_o^i. For clarity purposes, we consider that the WCTT of the switch-criticality message is the same in each node: $\forall n \in \mathcal{N}, C_o = C_o^n = C_c$.

For a node i in the network with a central node (sink node) S_h , the delay needed to receive the order directly depends on the distance between i and S_h. We note this distance d_h. Furthermore, we make the hypothesis that the switching latency sl is the same for each physical link (like in IV-C), and so that $sl = 0$ms : the electronical delay generated by the distance between each node is null.

We then obtain the following expression of M_{delay}:

$$\begin{aligned} M_{delay}^n = &\quad d_n * (C_c + sl) + \epsilon_n \\ M_{delay} = &\quad \max_{n \in \mathcal{N}}(d_n) * (C_c + sl) + \epsilon \end{aligned} \tag{10}$$

As we are computing the multicast delay, we are computing the worst case delay needed for the farthest node from the sink node to receive the switch criticality order. At the end of this multicast delay M_{delay}, we are sure that all the nodes in the network received the criticality change information.

The criticality switch occurs on any node at its local time $t_m + M_{delay}$, where t_m (respectively M_{delay}) is the timestamp (the switching delay) sent by the sink node in the criticality switch multicast message. Hence when all nodes have received the criticality switch request, hence preserving the consistency of the criticality level information in all nodes. All the nodes switch almost at the same time, with a time difference bounded by ϵ the clock synchronization accuracy.

D. Criticality-switch message

Given the expression of I_{delay}^n(8) and M_{delay}(**??**), we obtain the global expression of the criticality-switch delay S_{delay} in the network. Given the hypotheses that the switching latency is constant, and that PTP, SCC and the reliable multcast message have the same WCTT (noted as C_c), we obtain:

$$\begin{aligned} S_{delay} = &\quad F_{PTP} * \sum_{\substack{j \in hp_c \\ \mathcal{P}_c \cap \mathcal{P}_j \neq \emptyset}} \left(S_{max_c}^{first_{c,j}} - M_c^{first_{c,j}} + A_{c,j} \right) \\ &+ \sum_{h \in \mathcal{P}_c} \delta_c^h \\ &+ (2 * \max_{n \in \mathcal{N}}(d_n) - 1) * (C_c + sl) + C_c(\max_{n \in \mathcal{N}}(d_n) - 1) \\ &+ \epsilon \end{aligned} \tag{11}$$

When we compute the total delay needed to operate a criticality switch in the network, we then compute the delay represented by PTP synchronization messages, the non-preemptive effect induced by all messages in the network, the WCTT of criticality switch messages and finally the delay induced by electronical latency and clock jitter.

V. Simulation

In order to provide estimations of the transmission delay of criticality information inside a network, we provide simulation results using ARTEMIS [4]. To provide these results, we based our approach on the topology described on figure 1, and on randomly-generated tasksets to simulate traffic load in the topology. To generate these tasksets, we used the UUnifast generation algorithm presented in [9]. But as this method was designed for processor-context simulation, we first adapted it to network context.

A. UUnifast

The UUnifast method is a random tasksets generation method, first presented in [9] and used to generated tasksets in mono and multicore contexts. It consists in three steps :

- First, we generate random periods in a configurable time interval $[\epsilon; T]$, where ϵ is the clock accuracy(see IV) and T is the global simulation time. These periods can be of any size.
- On a second point, we generate a random value in $[0, 1]$ according to a uniform law, called the utilisation of a frame. It represents the individual load represented by the frame.
- Then, based on the generated period and utilisation, we compute the WCET of the task.

When it comes to adapt UUnifast method to a network context, it results in two different problems to solve. First, when we generate a random flowset, we have to specify a targetted load for the whole system (to compute the utilisation). But in the network, the load is different in each node. To solve this, we decided to focus on the load on the sink node of the topology : as it is the central node, we assume this is the one with the highest load, or at least with the most important one to focus on.

Secondly, we had to modify the UUnifast method in order to generate LO and HI critical messages. So we introduced a critical rate in the method which defines, randomly and according to this rate, the average number of critical messages inside the whole generated flowset. As we based our approach on critical periods change, each frame was defined either with just one LO-critical period, or with one LO and one HI critical period.

B. Impact of the load

We did generate different flowsets, each one representing a different scheduling scenario. We computed the utilization of each flow with a uniform distribution based on the network load, and finally deduce the WCTT of the flow. As we are working with ethernet(IEEE 802.3), the size of each frame in constrained in size between 64 and 1518 bytes. With a 100 Mb/s bandwith, all the WCTT in our network are bound between 4.9μs and 115,8μs.

We made a scenario with a set of 50 different flows (corresponding to a classical context of use). In our evaluation, we fixed a global simulation time of $t = 500\mu$s, which is enough to observe and bound the different delays we want to focus on. We made the load represented by the flows increasing from 0.4 to 1. The computed load was the one in the central node (S_4) which receives all the flows. With

these generated flows, we wanted to evaluate the impact of the network load on the criticality switch transmission delay.

Assigning the highest priority for mixed criticality management messages (dedicated for configuration messages) allows the criticality-switch messages to not be delayed more than once by messages with a lower priority (non-preemptive effect). We verified this hypothese by evaluating the MC delay switch as a function of the network load. Thus, we need to add to this delay the one due to PTP synchronization messages, considered with a higher priority (see IV).

We can observe in figure 3 that, basically, the delay of the criticality-switch as a function of the network load is linear.

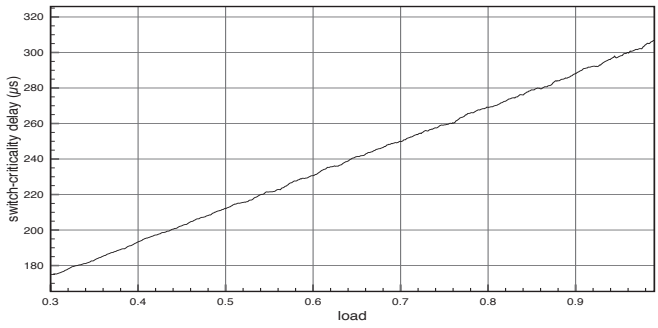


Fig. 3. Switch-criticality delay/load

We now analyse the impact of the non-preemptive delay for flows sent with a smaller priority than those of MC messages.

C. Non-preemptive effect

We picked the same parameters as for the previous scenario, but we also decided to put a limit on the highest WCTT in the network than for MC messages. We simulated 4 different cases with different limits in the highest WCTT.

The results obtained (see figure 4) shows that the criticality-switch delay is strongly influenced by the highest WCTT in the network. To evaluate this influence, we limited the highest WCTT to small sizes: 20 μs (262B), 30 μs (393B) and 40 μs (524B). We can observe that, at the highest loads, the transmission time is nearly constant.

The point is: in the UUnifast task generation method we presented in V-A , the task model bases the WCTT computation on the load. It means that the highest WCTT increases with the load. That explains why, in the first scenario without any particular limit, we obtained an increasing transmission time with the increasing load. On the contrary, when limiting the highest WCTT in the network, we obtain an constant switching delay (impacted from 1% to 6% in our examples at the highest network load, due to error margins we tolerated in the load computation).

As explained in IV, attributing to the switch-criticality and reliable multicast messages the highest priority allows us to make the criticality messages independant from the network load (the impact is very limited). No matter the traffic in the network, we can then compute the criticality switch delay in the network. As this switch delay is bound, the transmission of HI-critical messages can be assured in our topology in a bound and known time.

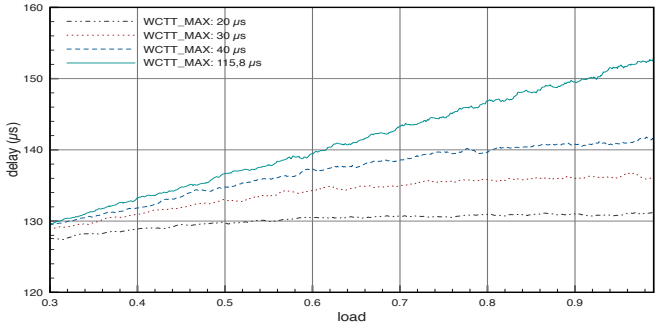


Fig. 4. Switch-criticality delay/load with limited highest WCTT

VI. Conclusion

A. Conclusion

In this paper, we present a criticality-change protocol in a clock synchronized Switched Ethernet network, in the case of two criticality levels. This criticality change protocol is based on a reliable real-time multicast used update the criticality information in all the nodes of the network while preserving its consistency. The real-time multicast builds a total order for the updates of the critality information. It relies on the based IEEE1588v2 global clock synchronization. We characterize the worst case delay for a criticality change with the trajectory approach. Through simulation, we generate random scenarios to test the time needed for a criticality change. We show that the criticality switch delay is harldy impacted by the network load. As a further work, we will show how to characterize the end to end response time of HI messages in the case of a LO to HI criticality switch.

References

[1] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed-criticality systems," in *RTSS 2011*.

[2] S. Baruah, "Mixed criticality schedulability analysis is highly intractable," 2015.

[3] A. Burns and R. Davis, *Mixed criticality systems: A review*. Department of Computer Science, University of York, 2013, vol. Tech. Rep.

[4] O. Cros, F. Fauberteau, L. George, and X. Li, "Simulating real-time and embedded networks scheduling scenarios with artemis," in *WATERS*, 2014.

[5] H.-T. Lim, L. Völker, and D. Herrscher, "Challenges in a future ip/ethernet-based in-car network for real-time applications," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024727

[6] L. George and P. Minet, "A fifo worst case analysis for a hard real-time distributed problem with consistency constraints," in *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[7] S. Martin, P. Minet, and L. George, "End-to-end response time with fixed priority scheduling: trajectory approach versus holistic approach," in *International Journal of Communication Systems*, vol. 18, no. 1. John Wiley & Sons, Ltd., 2005, pp. 37–56.

[8] X. Li, O. Cros, and L. George, "The trajectory approach for afdx fifo networks revisited and corrected," in *RTCSA'14*, 2014.

[9] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Journal of Real-Time Systems*, pp. 129–154, 2005.

Mixed Criticality Systems: Beyond Transient Faults

Abhilash Thekkilakattil[1], Alan Burns[2], Radu Dobrin[1], and Sasikumar Punnekkat[3]

[1]Mälardalen Real-Time Research Center, Mälardalen University, Sweden
[2]Real-Time Systems Research Group, Department of Computer Science, University of York, UK
[3]Department of Computer Science and Information Systems, Birla Institute of Technology and Science, India

***Abstract*—Adopting mixed-criticality architectures enable safe sharing of computational resources between tasks of different criticalities consequently leading to reduced Size, Weight and Power (SWaP) requirements. A majority of the research in mixed-criticality systems focuses on scheduling tasks whose Worst Case Execution Times (WCETs) are certified to varying levels of assurances. If any given task overruns its WCET, the system switches to a higher criticality and all the lower criticality tasks are discarded to make time for the execution of higher criticality tasks. Task execution time overruns are transient faults that are typically tolerated by simply executing an alternate task before the original deadline, or, by discarding the failed task to prevent it from interfering with higher criticality tasks. However, permanent faults such as processor failures can render the system to be useless, many times leading to unsafe states. In this paper we present a taxonomy of fault tolerance techniques to tolerate permanent faults, as well as map it to real-time mixed-criticality requirements based on the extend of fault coverage that in turn influences the associated assurance.**

I. Introduction

The complexity of software in real-time applications is increasing unlike ever before. This has created novel challenges in the design and verification of such systems, particularly with respect to timeliness guarantees. One way of reducing the complexity involved in providing temporal guarantees is to adopt a mixed criticality architecture, where only some tasks that are deemed to be critical to the operation of the system are provided strong guarantees, while others are typically provided weaker guarantees. The two main motivations behind adopting a mixed criticality architecture are i) efficient use of computational resources by integrating functionalities of varying criticalities (that are mostly certified to varying assurance levels) on the same platform [13] and ii) enable easy certification of systems by different certifying authorities [4]. Note that in this paper, ”resources” refer to ”computational resources” *e.g.,* a processor.

Mixed criticality systems has received good reception in the real-time systems community since Vestal described the problem in [13] (though some works such as [9] predates Vestal’s work). The central assumption behind the mixed-criticality model proposed by Vestal [13] is that the individual task Worst Case Execution Times (WCET) monotonically increase with criticality of the components, because, the more critical a task is with respect to the correct functioning of the system, more conservative is its WCET estimate thereby increasing confidence on the estimates. We refer to all the mixed criticality scheduling models (*e.g.,* [13][4]) that adheres to this assumption as *Vestal’s model*.

Most of the research in *mixed-criticality real-time scheduling* (which is Vestal’s model) considers, what is referred to in the dependability community as the, *transient faults*. A *transient fault* can be a task execution time overrun (as assumed by Vestal’s model) or single event upsets (as assumed by more recent works on mixed-criticality systems [10] [12]). Transient faults on real-time tasks are commonly tolerated by a simple re-execution of the same task or by an execution of an alternate task (or even by discarding the failed task so as not to jeopardize timeliness guarantees of other tasks, similar to Vestal’s model). Although the ultimate goal of mixed-criticality systems is to provide higher *assurances* (*e.g.,* reliability) to higher criticality components, the focus of the majority of research is limited to transient faults, in particular to transient faults caused by different levels of Worst Case Execution Time (WCET) assurances. Moreover, the majority of the works assume that lower-criticality tasks can be safely discarded as the system switches to a higher criticality level (which is when tasks overrun their WCETs).

On the other hand, permanent faults can render a system to be completely useless *e.g.,* a processor failure. Moreover, inherent hardware faults (such as corrupted memory) can cause failures that are hard to detect by re-executions or executions of alternate tasks. In this case, system designers need to adopt spatial redundancy coupled with a voting mechanism to determine whether or not the generated outputs are erroneous. The majority of the research in mixed-criticality systems [7] do not consider the possibility of permanent faults, the issues/problems arising out of this, as well as its implications in the context of a mixed-criticality architecture.

Adopting spatial redundancy brings forth many challenges in conjunction with mixed-criticality architectures, particularly with respect to satisfying the ”S” and ”W” of the SWaP (Size, Weight and Power) constraints. One of the goals of adopting a mixed criticality architecture is to enable *safe* sharing of hardware resources between highly critical and lesser critical software components in order to reduce SWaP. However, spatial redundancy techniques require spare hardware that increases SWaP. Consequently, one of the goals of this paper is to investigate the possibilities of implementing a mixed-criticality architecture when using spatial redundancy.

Another essential requirement for guaranteeing ”timeliness” of real-time tasks that are replicated on a specified number of processors is that the hardware on which these replicas execute need to be tightly synchronized.

Traditionally, this was achieved by implementing tight synchronization schemes that typically require hardware modifications/support. Implementing tight synchronization schemes for all the spatially redundant components increases development costs, as well as make the components dependent on a global time base. The use of loosely synchronized systems (*i.e.,* systems where time synchronization is carried out by software) enables easier use of *e.g.,* multicore systems to implement spatial redundancy. Adopting a mixed-criticality architecture, especially on multicore systems, brings forth possibilities to implement the different functionality on software and perform voting using a "time aware voter" (*e.g.,* see Aysan *et al.* [3]).

This paper presents a taxonomy of spatial redundancy techniques to tolerate permanent faults and identify how mixed-criticality architectures can be implemented when using spatial redundancy.

To summarize, the main contribution of this paper is a taxonomy of spatial redundancy techniques in the context of implementing mixed-criticality architectures. Particularly, we consider two main challenges:

- Maximizing efficiency of replica allocation to reduce cost. This can be achieved by provisioning resources based on the criticality of the associated tasks.
- Providing different levels of assurances for spatially redundant software components (tasks). The different levels of assurances are based on the extent of coverage of different faults.

The rest of the paper contains the system model in Section II, the main contributions of this paper in Section III followed by conclusions in Section IV.

II. System Model

In the following, we describe the system model in detail in order to make the context clearer.

A. Example System

In this paper, we use a running example of an autonomous vehicle adapted from Burns *et al.* [6]. The example services (tasks) are given in table I and consists of 2 high criticality functions, 4 medium criticality functions, 1 low criticality function and 1 non-critical function. These criticalities can be mapped to tasks associated with components that are assigned to specific Safety Integrity Levels (SILs) described in IEC 61508. Collision avoidance and braking control are high criticality tasks, while engine and lateral control, path finding and route planning can be considered to be of medium criticality. Display control functions are of low criticality while music streaming can be considered to be a non-critical task.

Function	Criticality
Collision Avoidance	High
Braking Control	High
Engine Control	Medium
Lateral Control	Medium
Path Finding	Medium
Route Planning	Medium
Display Controls	Low
Music Streaming	Non-Critical

TABLE I: Autonomous Vehicle Example

B. Fault Model

In this paper we consider both transient and permanent faults. As noted before, WCET overruns (such as the one assumed in the widely used Vestal's model) are one type of transient faults and is tolerated by either aborting the task or by simply re-executing the task or executing an alternate task. Avizienis *et al.* [1] defines a permanent fault to be fault whose presence is assumed to be continuous in time. This can be for example, a processor failure in a distributed system that leads to an absence of output or a physical memory failure that leads to incorrect outputs. Permanent faults can be tolerated by employing spatial redundancy, *i.e.,* replicate the required functionality on multiple hardware.

Example II.1. *Consider the autonomous vehicle example given in table I that performs collision avoidance by detecting obstacles in its path and choosing appropriate actions such as either stop, slow-down or navigate around it. Object detection is a key functionality in this system. Suppose the processor on which the "collision avoidance" software runs fails, it must be possible for the system to recognize the failure and bring the vehicle to a safe state or recover from it. A relatively easy way out is to replicate the collision avoidance functionality on multiple hardware,* e.g., *using Triple Modular Redundancy (TMR), and perform a voting. Adopting TMR enables tolerance to* 1 *fault,* i.e., *even if one of the replicas fail, in principle, the vehicle can detect it and still continue functioning.*

There are different types of permanent faults that need to be tolerated. The higher the coverage of the fault tolerance mechanism associated with a task, the higher the assurance one can give to the particular task. This provides for interesting trade-offs between task criticalities, development costs and SWaP.

Design faults: Occurs due to the deficiencies in the design and development of the system. A design fault may be due to the use of *e.g.,* a particular type of hardware or adoption of a specific implementation of a particular algorithm when building the system.

There are two types of design faults:

1) **Hardware Design faults:** Faults that either originate in the hardware or affects the hardware due to faulty design are referred to as hardware design faults. Examples of hardware design faults include manufacturing defects in the computer.
2) **Software Design faults:** Faults that affect the software of a computer system as a result of incorrect design are referred to as software design faults. Examples of software faults include faults due to incorrect interpretation of the specification, or a faulty implementation of an algorithm.

Random faults: A random fault is a fault whose time of occurrence cannot be predetermined, nor the causes can be

identified offline. It may, for instance, be the result of wear and tear due to the continuous use of the system. On the other hand, the rate of occurrence of random faults for a given system can be estimated, for example, it is possible to analyze impact of wear and tear on the system.

Byzantine faults: Byzantine faults occur when some replicas behave arbitrarily differently. Moreover, different observers will record different behaviors of the replicas. In general, byzantine faults are the worst kind of faults and requires significant replication to be tolerated. Typically, to tolerate m byzantine faults, there is a need of $3m + 1$ replicas.

C. Fault Tolerance Mechanism

There are two primary types of fault tolerance mechanisms that can be implemented:

1) Fail Stop: In this form of fault tolerance, whenever a component fails, it stops functioning completely in order to prevent interfering from other (potentially dependent) components. Majority of the research in mixed-criticality real-time systems implement this form of fault tolerance in which, in case of an execution time overrun, the system switches to a higher criticality state and discards all the lower criticality components, essentially stopping all lower criticality tasks from executing.
2) Fail-Operational: In this form of fault tolerance, even if a component fails, the system continues to give an acceptable level of service by typically employing back-ups in the form of temporal or spatial redundancy.

D. Task Model

We consider a set of n real-time tasks/functions denoted by $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ where each τ_i has a minimum inter-arrival time T_i and a deadline D_i. The transient faults on any task τ_i are tolerated either by simply re-executing τ_i or executing a back-up task to τ_i– we refer to both as *alternates*. In this context, the main execution of τ_i is referred to as the *primary* and has an execution time denoted by C_i. Permanent faults on any task τ_i are tolerated using spatial redundancy: each τ_i has a specified number of *replicas* that are executed in parallel, after which a voting is performed.

Every $\tau_i \in \Gamma$ needs to tolerate $\delta_i^t \geq 0$ transient faults and/or $\delta_i^p \geq 0$ permanent faults. This means that each τ_i requires δ_i^t *alternates* in addition to the primary execution and needs to be *replicated* on $(2\delta_i^p + 1)$ processors [11]. Alternately, the execution time of the back-ups of any task τ_i can be seen as the "extra-time" that τ_i may need in case of a WCET overrun at any given criticality level.

Example II.2. *Consider a task τ_1 that needs to tolerate $\delta_i^t = 1$ transient fault and $\delta_i^p = 0$ permanent faults. This means that τ_1 needs to be re-executed once or an alternate task must be executed if there is a transient fault on τ_1.*

This model generalizes the widely used model for mixed-criticality systems, since *e.g.,* by enforcing $\delta_i^t \leq 1$ (and $\delta_i^t = 0$ *i.e.,* no replication) we get a dual criticality system: when $\delta_i^t = 1$, τ_i becomes a high criticality task that has two execution times (analogously one alternate) and when $\delta_i^t = 0$, τ_i becomes a low criticality task that can be safely discarded upon overrun. In the context of the widely used model, hereafter referred to as Vestal's model, the recovery task can be seen as the "extra" duration for which high criticality tasks can execute in case of WCET overruns before the system switches to a higher criticality level.

III. Mixed-Criticality Design Challenges in Dependable Real-time Systems

In this section, we present a how different levels of assurances can be provided to tasks of different criticalities when building predictable mixed-criticality systems that employs spatial redundancy. We first explain the time synchronization problem when using spatial redundancy, that may lead to disasters in hard real-time systems. We then identify how mixed-criticality architectures can be implemented when using spatial redundancy to minimize SWaP requirements as well as the effort required to implement them.

Spatial redundancy, time synchronization and mixed-criticalities: Depending on the criticality of the task, the replicas of the task may be implemented as simple circuits developed by independent teams to ensure diversity. Alternately, the different replicas may be developed as software, by independent teams, and may be scheduled together with other tasks on processors from different vendors; albeit with reduced assurance when compared to the previous case. One of the key design challenge involved in providing spatial redundancy, in this case, is to ensure synchronization of different replicas to provide timely output to the voter. The need for tight synchronization is illustrated by the following simple example.

Example III.1. *Consider the sensors associated with lateral control of the example autonomous vehicle described in Table I. In order to provide high assurance to the Lateral Control function, spatial redundancy must be employed. Suppose the associated sensor is triplicated, with a voter, then there is a risk of two of the replicas giving the same output while the third gives a late output that is different because of the change in value over time. In this case, there is a risk that the voter discards the correct (albeit late) value by tagging it as "incorrect" since it does not agree with the outputs from the two other sensors.*

These errors can be tolerated by adopting a tight time synchronization between the different replicas. However, providing tight synchronization is costly and requires significant effort. Moreover, having a tight synchronization between different replicas makes the system heavily dependent on the global time base. Alternately, for lesser critical tasks, loose synchronization algorithms implemented using software may be used that requires lesser development effort and cost. When using loose synchronization scheme, care must be taken to ensure that the voter does not suffer from timing errors. Loosely synchronous systems facilitate cost reduction by enabling the use of commercially available real-time operating systems on the individual processors without requiring modifications to enable tight

synchronization. Each processor used to replicate the different tasks can execute the replicas using the local scheduling algorithm that can be, for example, EDF or FPS (as long as the individual replicas produce timely outputs). A time aware voter such as the one proposed by Aysan *et al.* [3] can detect and tolerate timing errors, consequently providing dependability guarantees.

A. Mixed-Criticality Architecture for Spatially Redundant Functions

Providing reliability and safety guarantees using spatial redundancy implies increased hardware that in turn results in increased SWaP requirements. Adopting a mixed-criticality architecture reduces SWaP requirements by provisioning the computing resources such that it reflects the task criticalities and the associated required assurance levels. Implementing mixed-criticality architectures for systems using spatial redundancy is still largely an unexplored area. In this section, we investigate methods to provide different levels of assurances to different tasks that uses spatial redundancy to improve reliability and safety while reducing SWaP. An overview of the mapping of the criticalities to tasks based on the fault coverage assurances is summarized in table II.

High Criticality Tasks: The high criticality tasks are the *most important* tasks in the system and require a very high level of assurance. Consequently, the probability of failures need to be significantly low. These tasks are associated with components classified as *e.g.,* SIL 4 of the IEC 61508 and are highly critical for the safe operation of the system. The high criticality task failures can result in disastrous consequences for the system and hence need to be provided with the highest level of assurance. For example, the collision avoidance and braking control in Table I are highly critical functionalities to ensure safe operation of an autonomous car.

Assurance Mechanism: The highest criticality tasks may be implemented on dedicated hardware to ensure isolation (as is typically done in many systems [14]), and a high integrity voter implemented as a simple electronic circuit performs voting. The use of simple electronic circuits implies that the voter can be verified to a very high degree of assurance [2], and the use of dedicated hardware guarantees that the tasks are protected from many types of faults.

- In order to provide assurances against random faults, the high criticality tasks are typically replicated *e.g.,* using Triple Modular Redundancy (TMR).
- To provide protection against design faults (both hardware and software), there is a need to ensure diversity. This can be done by N-version programming, *i.e.,* developing the different replicas using different development teams. Moreover, the different teams must use hardware and development tools from different vendors.
- To protect against byzantine faults, byzantine fault tolerance mechanisms must be adopted. Protection against byzantine faults imply further increase in hardware requirements. Typically, to tolerate δ_i^p byzantine faults, there is a need of $3\delta_i^p + 1$ replicas. Note that byzantine failures are observed more frequently than expected [8]. If the tasks are protected against byzantine faults, they are implicitly protected against all other faults (and hence does not require *e.g.,* TMR).
- The replication and diversity ensures that the system is tolerant to many transient faults since there is redundancy. Typically, by having $2\delta_i^p + 1$ redundant system implies protection against $2\delta_i^p$ transient faults. The Airbus A320, for example, uses both replication and diversity to ensure fault tolerance [5].

A key challenge here is to ensure tight synchronization between the replicas and the voter to ensure timeliness of the generated output. As a consequence of the above design, the development of high criticality tasks can be very costly.

Medium Criticality Tasks: The medium criticality tasks correspond to components of the system that are to be certified as *e.g.,* SIL 3 of IEC 61508. A failure in these tasks can cause serious consequences to the correct functioning of the system. However, consequences of medium criticality task failures are less disastrous than the high criticality failures, and the associated reliability guarantees need not be as high as the critical tasks (or the probability of a failure causing a disaster is lower compared to the high criticality tasks). Consequently, the medium criticality tasks need not be provided with the highest assurance level similar to the critical tasks. They can be provisioned less pessimistically, even using commercially available high integrity processors, than high criticality tasks to save on Size, Weight or Power. Engine control, given in Table I, is an example of a medium criticality task. Even though it may not be as critical as collision avoidance, there is a need to ensure its failure free execution.

Assurance Mechanism: The medium criticality tasks may be implemented in software on high integrity processors using *e.g.,* table driven scheduling. Even though the medium criticality tasks need not be implemented as electronic circuits to guarantee high assurance, they need to be made significantly fault tolerant.

- The medium criticality tasks can be replicated *e.g.,* using TMR to protect it against random faults. The replicas of the medium criticality tasks may be scheduled using highly predictable scheduling algorithm *e.g.,* table driven scheduling on different processors. These tasks need to execute in lock-step, and on completion pass on the output to a voter that then performs voting to mask any task failures.
- The medium criticality tasks can be made tolerant to design faults (both hardware and software) by ensuring diversity, *e.g.,* choosing processors on which the replicas execute from different vendors and by ensuring the use of N-version programming.
- The replication and diversity guarantees protection against many transient faults.

Such a setup requires tight synchronization schemes between the high integrity processors, in order to guarantee timeliness of the replica outputs and in turn guarantee timely output

Task Criticality	Transient Faults	Random Faults	Software Faults	Hardware Faults	Byzantine Faults
High	Fully covered	Fully covered	Fully covered	Fully covered	Fully covered
Medium	Fully covered	Fully covered	Fully covered	Fully covered	
Low	Fully covered	Fully covered	Fully covered	Partially covered	
Non-critical	Fully covered	Partially covered			

TABLE II: Mapping criticalities to tasks based on fault coverage.

from the voter. The advantage here is more than one medium criticality tasks can be scheduled on the same processor, as opposed to implementing them on the hardware, consequently reducing SWaP requirements. Since the tasks are implemented as software, and are scheduled on commercially available processors, the development cost associated with medium criticality tasks will be less.

Low Criticality Tasks: Low criticality tasks are associated with those components that need to be provided with *e.g.,* SIL 2 guarantees under IEC 61508 standard. Failures on low criticality tasks can cause less severe disruption of services in the system that are not disastrous (or the probability of a failure causing a disaster is low). However, these tasks are still required to ensure the normal operation of the system and needs to be provided with appropriate guarantees. Display controls in autonomous vehicles may not be as critical as collision avoidance or engine control and hence need not be provided with the same level of assurance.

Assurance Mechanism: The low criticality tasks may be implemented on commercially available multicore processors and scheduled using any standard real-time scheduling algorithm. However, some level of fault tolerance must be implemented to ensure the associated failure probabilities are low.

- The low criticality tasks may be replicated on different processors of the multicore platform. The different cores may be synchronized using relatively cheap synchronization algorithms, and a "time aware" voter (*e.g.,* [3]) can guarantee timeliness of the generated output.
- Protection against software design faults can be implemented by employing N-version programming. Since, the replicas are scheduled on the different cores of the same multicore platform, no protection exists against hardware design faults and many hardware operational faults.
- The replication ensures that the system is automatically protected against many transient faults. Moreover, if software diversity is ensured, it further increases protection against many transient faults.

The use of commercially available processors, together with the possibility of adopting loose synchronization enables the use of relevant uniprocessor or multiprocessor scheduling algorithms such as Earliest Deadline First or Fixed Priority Scheduling to schedule the replicas, consequently enabling the use of commercial real-time operating systems. The main concern here is regarding the faults that may occur due to the loose synchronization. In this case, a time aware voter such as the one propsed by Aysan *et al.* [3] can guarantee absence of timing faults in such loosely synchronized systems. Adopting such an architecture enables efficient use of the available processing power since many algorithms, *e.g.,* EDF, that are known to be optimal can be employed.

Non-Critical Tasks: Non-critical tasks are the least "important" tasks in the system as they are associated with components that can be given lowest level of assurance *e.g.,* SIL 1 guarantees of IEC 61508. Non-critical tasks can be safely discarded without affecting the normal operation of the system. The only major concern in this case is that the non-critical tasks must not "interfere" with the execution of higher criticality tasks *i.e.,* they must be protected against transient faults by implementing a fail silent mechanism. Vehicular entertainment related task, such as music streaming in Table I, are good example of non-critical tasks that are not significant for the specified mission.

Assurance Mechanism: Non-critical tasks are scheduled normally along with tasks of higher criticalities. Non-criticality tasks may re-execute whenever there is slack, and is discarded immediately upon transient faults like an execution time overrun. The presence of extra computing resources ensures that non-critical tasks have a higher possibility of re-execution. A limited form of protection against transient faults and random faults can be added by ensuring that the non-critical tasks can re-execute whenever spare computing capacity is available.

Needless to say, depending on the coverage of faults, more criticalities can be defined.

IV. Conclusions

Even though the ultimate goal of mixed-criticality systems is to enable efficient resource usage while providing different levels of assurances to different components, the majority of research in mixed-criticality systems focus only on issues related to tolerating execution time overruns, which are only one type of transient faults. Despite the fact that more recent works considered transient faults other than execution time overruns, the challenges with respect to implementing mixed-criticality architectures for tolerating permanent faults has largely remained out of focus. The hardware redundancy required to tolerate permanent faults implies increased SwaP requirements, while mixed-criticality architectures enable hardware provisioning to real-time tasks based on the associated required assurance levels. In this paper, we present a taxonomy of spatial redundancy techniques, as well as propose a mapping of the assurance levels to task criticalities based on the extend of fault coverage with respect to permanent faults. A positive side effect of using spatial redundancy is that transient faults, such as execution time overruns, are automatically covered, and hence, this paper aims to initiate a discussion on the use of spatial redundancy techniques in the context of mixed-criticality systems.

Future work include investigation of optimal resource allocation strategies for assurances against different types of faults, as well as scheduling mechanisms for the tasks and their different replicas on multiprocessor platforms to guarantee timely outputs to the voter.

REFERENCES

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, January 2004.

[2] Huseyin Aysan. Fault-tolerance strategies and probabilistic guarantees for real-time systems. In *PhD thesis, Malardalen University*, June 2012.

[3] Hüseyin Aysan, Iain Bate, Patrick Graydon, and Sasikumar Punnekkat. Improving reliability of real-time systems through value and time voting. In *The 19th IEEE Pacific Rim International Symposium on Dependable Computing*, December 2013.

[4] S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, August 2012.

[5] D. Briere and P. Traverse. Airbus a320/a330/a340 electrical flight controls - a family of fault-tolerant systems. In *The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 616–623, 1993.

[6] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *J. Syst. Archit.*, 2000.

[7] Alan Burns and Rob Davis. Mixed criticality systems - a review. In *Available:http://www-users.cs.york.ac.uk/ burns/review.pdf (accessed on 31 July 2015)*.

[8] Kevin Driscoll, Brendan Hall, Hkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 235–248. Springer Berlin Heidelberg, 2003.

[9] S. Islam, R. Lindstrom, and N. Suri. Dependability driven integration of mixed criticality sw components. In *The Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.

[10] RisatMahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 2014.

[11] Sasikumar Punnekkat. *Schedulability Analysis for Fault Tolerant Real-time Systems*. PhD thesis, University of York, UK, June 1997.

[12] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Fault tolerant scheduling of mixed criticality real-time tasks under error bursts. In *The International Conference on Information and Communication Technologies*. Elsevier Procedia Computer Science, December 2014.

[13] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *The 28th IEEE International Real-Time Systems Symposium, 2007*, December 2007.

[14] K. Vipin, S. Shreejith, S.A. Fahmy, and A. Easwaran. Mapping time-critical safety-critical cyber physical systems to hybrid fpgas. In *The IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, 2014.

Evaluating Mixed Criticality Scheduling Algorithms with Realistic Workloads

David Griffin[1] and Iain Bate[1,2] and Benjamin Lesage[1] and Frank Soboczenski[1]
[1]Department of Computer Science
University of York
York, United Kingdom
[2]School of Innovation, Design, and Engineering
Mälardalen University
Västerås, Sweden
Email: {david.griffin, iain.bate, benjamin.lesage, frank.soboczenski}@york.ac.uk

***Abstract*—Most work on mixed-criticality scheduling has considered timing-related failures to be independent of one another. In reality this is not true as in many systems the state that caused the original failure will be similar to the state in the next release (job) of the task. Therefore when arguing about the number of jobs that do not meet their deadlines, it is crucial tasks have an appropriate fault model incorporated into the tool framework (i.e. task set generators and simulators) used to evaluate scheduling policies. The second issue that affects the tool framework is the choice of Worst-Case Execution Times (WCET) for different criticality modes of tasks. In the current literature it has been argued that a WCET should be chosen which would only be exceeded incredibly rarely, e.g. 1 in 10^{16} jobs. This leads to WCET values much greater than the High WaterMark (HWM). The needs of certification and the consideration of how safety is argued leads to the conclusion that the probability of a job not meeting its deadline can be much greater. This would greatly impact the WCETs and hence the results of the evaluation. The contributions of this paper are thus a more realistic tool framework, and hence more realistic results than those previously reported, which we claim gives a better insight into how the scheduling policies would behave in practice and hence better evidence for any safety case.**

I. Introduction

Mixed-Criticality Scheduling (MCS) is basically where tasks of different criticality levels share processing resources. There are many motivations for MCS including the ability to efficiently use resources whilst giving appropriate timing guarantees and reducing the cost of certification by allowing software sharing resources to be developed and certified by different processes [1]. The basic model of MCS consists of a system with N_m modes of operation.

In each mode, M, different tasks are executed and each task (denoted i) may have a different execution time per mode $C_{i,M}$. The task itself may not change between modes, however level of confidence desired in the Worst-Case Execution Time (WCET) may differ between modes. To date most of the work has been giving definitive knowledge of the schedulability of tasks in each mode of operation. This knowledge is very useful for the safety case that is produced as part of the certification of systems [2], [3], [4], however the safety case is also interested in the loss of service when a failure occurs. More specifically, if a High-Criticality Task (HCT) exceeds the expected WCET in a particular mode then a mode change will occur which means some Lower-Criticality Tasks (LCT) are not executed for a period of time. It is important to know how often this happens and for how long the LCTs are not executed.

In [5] a simulator framework and scenario-based evaluation was used to provide information about how many LCTs would miss their deadlines in "situations of interest". The results of the evaluation gave some interesting insights including showing that a newly proposed scheduling policy, the Bailout Model (BM), caused LCTs missing their deadlines less frequently than previous state of the art algorithms. The insight gained though is only useful if the situations of interest reflect reality. The simulator framework had two main components: a Task Set Generator (TSG) that used UUnifast [6] to generate task set profiles according to specified characteristics (e.g. target utilisation) and the simulator itself. However, two issues make the presented framework unrealistic:

1) *Phasing of Failures* – Most work, including in [5], assumes that failures are independent. In reality this is not true. For example in a feedback-based control system, each job execution reuses much of the state from the previous one and the inputs to successive jobs are likely to be similar. Hence if a job exceeds the WCET in the current mode the next few jobs are also likely to exceed this value.
2) *Probability Distribution for Execution Times* – Real software doesn't conform to the "standard" distributions of execution times used in UUnifast and are more likely to be some form of extreme-value distribution [7].

In [8], Griffin used observations from real software on actual platforms to learn a model of timing-related failures. Specifically the model contained information about the likely number of successive failures and the magnitudes of these successive failures. Using this model, the DepET algorithm was proposed which was able to generate task sets with appropriate failure characteristics.

The first contribution of this paper is to use the models created as part of DepET to create DepET-RND that can generate and simulate individual jobs with dependent failures. The second contribution of this paper is to assess how generating jobs with dependent failures may affect our previously presented simulation results, in [5],where different Mixed-Criticality Scheduling (MCS) policies were evaluated with jobs that only had independent failures. Given appropriate execution time profiles, realistic results still depends on an appropriate choice of the initial probability of failure. The initial probability of failure F_i is defined as the likelihood the execution of a job exceeds its WCET for the

current mode when previous jobs have not exceeded this WCET, or in other words the likelihood a sequence of dependent failures will commence. Many works, e.g. [9], on probabilistic WCET (pWCET) have claimed that values of F_i could typically be 10^{-16}. The stated reason is that some certification standards require the likelihood of hazardous events to be once in every 10^9 operating hours and that a typical task may execute over 10^6 times an hour. The final contribution of this paper is to explain why this type of value is completely inappropriate and to propose more realistic values that can be used to guide pWCET analysis, configure DepET and evaluate MCS.

The structure of the paper is as follows. Section II presents a more realistic model of exceedance probability. Section III introduces DepET in more detail. An evaluation is presented in section IV before finally the conclusions are given in section V.

II. Choice of Exceedance Probability

As stated earlier, an important decision is to choose an exceedance probability that when combined with other sources of uncertainty[1] is suitable for the integrity level of the task such that the task's deadline is missed with an acceptable pattern. More specifically for each task and for each mode, an exceedance probability and associated WCET is needed. For example, consider a system that has the following two modes.

1) *normal mode* – All tasks are executed and the schedulability analysis is performed with a WCET referred to as C_{LO}.
2) *high-criticality mode* – This mode is triggered when any HCT exceeds its C_{LO} and where just the HCTs are executed and the schedulability analysis is performed with a WCET referred to as C_{HI}.

Given a distribution of execution times, obtaining C_{LO} and C_{HI} corresponding to a given P_{LO} and P_{HI} is straightforward, but choosing those target probabilities is not. As an example in Figure 1, assuming values P_{HI} and P_{LO} of 10^{-4} and 10^{-6}, C_{LO} and C_{HI} amount respectively to 2300 and 2800 cycles. However, knowing the effect of the choice of P_{HI} and P_{LO} on the system is not trivial; The value P_{LO} should be chosen such that the Low Criticality Tasks (LCTs) receive sufficient service, which must take into account how often these tasks are stopped from executing as well as how long the system can cope with them not being available. Schedulability analysis to date does not provide useful information for either of these. The scenario-based assessment in [5] provides a partial answer but more work is needed. Specifically statistical analysis of the results in [5] is needed to provide the information at the necessary level of confidence. It is noted the smaller the value of P_{HI}, the more C_{HI} will exceed the true WCET. The degree of this pessimism is defined as how much greater the WCET is than the actual WCET. The level of pessimism affects how much functionality can be put onto the resources whilst guaranteeing the timing requirements are met. To date most work on MCS has assumed P_{HI} is zero, however in [3] it is explained how safety-critical systems are designed to tolerate this. The rest of this section concentrates on how based on standard safety analysis the values of P_{LO} and P_{HI} could be chosen and what appropriate values for these might be.

[1]It is noted the other sources of uncertainty mean the exceedance probability used in pWCET analysis does not directly relate to the probability the WCET is exceeded. Other sources of uncertainty include having incomplete observations of execution time that feed into the pWCET analysis.

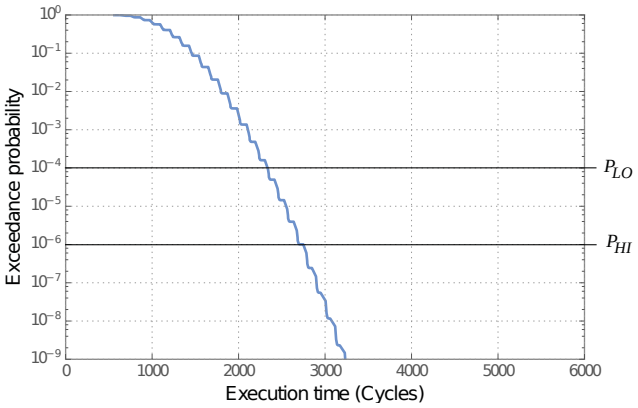


Fig. 1: Example illustrating how to choose P_{LO} and P_{HI} from an EVT execution time distribution corresponding to a task

For the purposes of this paper's discussion, we are interested in the likelihood of the conditions needed for a hazardous event (i.e. an event that might lead to death or injury) occurring and not how long these conditions are maintained. The classical approach for understanding how a hazardous event could occur is fault tree analysis [10] and the resulting fault tree can be used to give an understanding of the associated probabilities. It is also noted in [10] that these probabilities should only ever be used as a guidance in the safety case and cases are highlighted where over reliance on these figures have lead to serious incidents. A simplified example of a fault tree is given in Figure 2. Fault tree analysis considers how a hazardous event occurs (e.g. *Engine stops working* at the top of the tree through the logical combination of basic events (e.g. *Task exceeds WCET* at the bottom of the tree).

The fault tree presented is a simplified one as in fact there would be many more events involved between the basic events and the hazardous events. For example, the fault tree does not show how for many examples a single deadline miss would not be a problem, i.e. it may be we would have to stop control signals to the engine and fuel system for a period of time before it would have to stop not least due to inertia. Despite the simplifications, it is considered sufficient to illustrate the following. Its worth noting that removing the simplifications would likely mean that the points below are even more influential.

1) No single point of failure leads to the hazardous event. Wherever feasible this should be avoided and where it cannot regulatory authorities demand extra levels of rigour.
2) If the target probability for a hazardous event E is X (such as *Function Late*, then there is little benefit to a contributing event, (such as *Tasks WCRT is exceeded*) being lower than X. This can be seen due to the following facts. The behaviour of the *AND* operator is

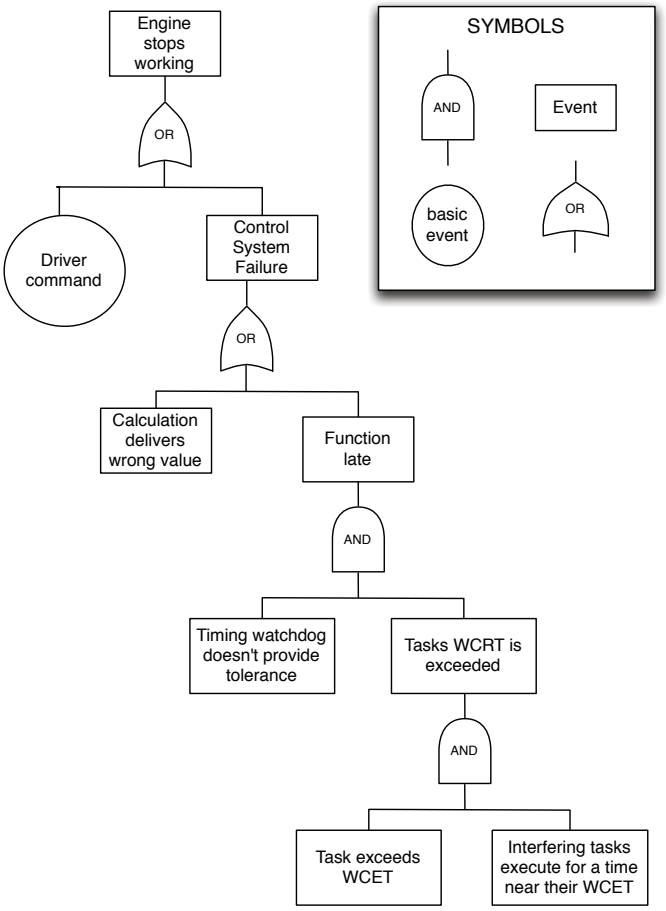


Fig. 2: Fault Tree for Car Engine Hazard - Engine Stops

such, then X provides a lower bound for the probability of events in the subtree [11]. Next, observe that when $lim_{X \to 0} X \ll 1$, *OR* gates tend to sum the probabilities of their input events [11]. As the probability of any failure due to WCET/WCRT in the fault tree is expected to be close to 0 (i.e. $\leq 10^{-4}$), then nX, where n is the number of input events to E, provides a lower bound to the probability of events in the subtree. As the number of input events n is expected to be relatively low (due to the hierarchical nature of a fault tree), and X is small, $nX \approx X$. Therefore it can be concluded that there is little or no benefit in the lower events in the tree having a smaller probability than the target probability of the hazardous event E. This leads to the corollary that as the highest target for the likelihood of a hazard in any certification standard is 10^{-9}, the likelihood of the hazardour event *Function Late* is upper bounded by 10^{-9}. Therefore, 10^{-9} provides a lower bound on the two causes of *Task exceeds WCET*, *Timing Watchdog does not provide tolerance* and *Tasks WCRT is exceeded*. Assuming that the failure rate of the timing watchdog is known, then due to the nature of *AND*, the probability of *Tasks WCRT is exceeded* can be significantly higher than 10^{-9}.

3) The probability of *Tasks WCRT is exceeded* would be low as it requires interfering tasks to execute for a time close or equal to their WCET as well as one task exceeding its WCET. This suggests *Task exceeds WCET* can be significantly greater than 10^{-9}.
4) There is little benefit in the probability of *Tasks WCRT is exceeded* being much less likely than *Calculation delivers wrong value* as the parent event *Function late* is combined with *Calculation delivers wrong value* by an *OR* gate, and hence the event *Control System Failure* has a probability at least as great as *Calculation delivers wrong value*. Therefore, *Calculation delivers wrong value* provides an effective lower bound to the values for *Function late* which significantly impact the probability of *Control System Failure*. Assuming that the software program is correct, then hardware failure is the only cause of *Calculation delivers wrong value*. In most safety-critical systems, the rate of hardware failure tends to be limited to a of 10^{-6} per hour of operation [12], and therefore for any hour of operation hardware has a probability of no greater than 10^{-6} of failure. Hence, this probability provides an effective lower bound on the *Calculation delivers wrong value*. Other factors that could increase this value include confidence on the functional testing of the task, but as the level of testing required is domain specific this is not considered in this paper. Hence there is little benefit to reducing the fault rate of *Function late* below that of *Calculation delivers wrong value*, 10^{-6}.

The second of these observations provides an upper limit to any failure probability in the fault tree. The third observation suggests that a new form of probabilistic analysis is necessary to determine the exceedance probability for the WCET given a desired exceedance probability for the WCRT. However, the final observation suggests that there is little benefit for P_{HI} being lower than 10^{-6} per hour, as any lower values would be negated by the 10^{-6} per hour rate of *Calculation delivers wrong value*, as this is an effective upper bound. To convert this failure rate to the failures per instance required when setting an exceedance probability for WCRT is difficult due to point 1; a single WCRT exceedance is not sufficient cause to failure of the system. In order to calculate the exact probability of failure it would be necessary to model the dependencies in execution times and find the probability of a sufficient number of faults (at both the execution and response time levels) occurring. It is clear that the value of P_{HI} should be quite a bit less than 10^{-16}. Using such a figure would be overly pessimistic and bring more uncertainty into the analysis as its harder to fit a curve at that type of probability level especially as it entails a significant extrapolation from the observations fed into the analysis [13]. Given an appropriate value of P_{HI}, then P_{LO} for each HCT should be chosen such that the LCT get sufficient service to support their associated hazardous events in the safety case.

In summary, this discussion shows that techniques are needed that help System Safety Engineers to convert the system-level reliability and availability targets into requirements on the computer system. The real-time systems engineers would then need a *translation method* to take these

lower-level reliability and availability requirements for tasks meeting their deadline and convert these into requirements for the pWCET community. We would suggest the real-time requirements for both deadlines and WCETs are split into the likelihood of the initial failure and the maximal duration of the failures once they have first occurred. The following section summarises a method for modelling dependent failures and then using these models when simulating task sets executing. This work could form the basis for any translation method.

III. Model of Dependent Failures

As illustrated in previous sections, limited research has been carried out as to how MCS algorithms behave when presented with realistic workloads. In particular, the dependencies between deadline overruns can create transitionary periods of high load which would not be present in randomised experiments. Recent work by Griffin et al. [8] defined the DepET algorithm, which is capable of modelling the dependencies of job execution times by utilising exceedance models; after operation, DepET returns a set of dependent execution times for the tasks it is asked to generate. In order to accomplish this, DepET defines each task as having a number of execution time $bands$ with the following properties that the user can configure in addition to the DepET algorithm internal properties $duration$, $prev$, $next$ and cET:

- mn, mx: The minimum and maximum values within this band
- d: The maximum value that an execution time may be displaced from its previous instance
- p: The probability of leaving the band
- EM: An exceedance model, used to determine the duration of the higher band

However, as previously employed, DepET is only capable of utilising an existing failure model and therefore is not usable for randomised testing. For reference, a pseudo-code implementation of DepET is given in Algorithm 1. Full details can be found in [8].

Therefore an extension to DepET is proposed, DepET-RND. DepET-RND utilises simple randomised exceedance models to control exceedance duration. The random exceedance models proposed simply selects, at random, the duration of a fault from a pre-specified randomly selected list. As DepET exposes a large number of variables, which may make targeting specific failure rates difficult, DepET-RND, produces a randomised configuration and then samples the values from that configuration. If the values are not sufficiently close to the target failure rate, the configuration is rejected and the process repeated. To hasten the search, user knowledge can provide a range of values for each parameter which are likely to produce configurations close to the desired failure rate. A pseudo-code implementation of the algorithm is given in Algorithm 2.

```
Function DepET(tasks)
    ETs ← []
    for task ∈ tasks do
        band ← task.current
        add randomnormal() * band.d to band.cET
        ETs.append(band.cET)
        clamp band.cET within band.mn and band.mx
        if band.duration = 0 then
            task.current ← band.prev
        end
        else if random() < band.p then
            task.current ← band.next
            band.next.duration ← band.EM.sample()
        end
        while band is not None do
            decrement band.duration
            band ← band.prev
        end
    end
    return ETs
end
```

Algorithm 1: The *DepET* algorithm

```
Function DepET-RND(number_of_tasks,
number_of_bands, target_failure_rate)
    tasks ← []
    for n ∈ range(number_of_tasks) do
        repeat
            random_ems ← a list of number_of_bands
            randomised exceedance models
            task ← a DepET task with random
            parameters and exceedance models
            random_ems
        until Failure rate of DepET([tasks])
        ≈ target_failure_rate;
        append task to tasks
    end
    return DepET(tasks)
end
```

Algorithm 2: The *DepET-RND* algorithm

IV. Evaluation

In order to evaluate the effects of dependent failures with regard to the effectiveness of mixed criticality scheduling algorithms, experiments were carried out using the simulation framework used by Bate et al. [5]. This simulator was extended to implement the DepET-RND algorithm. The algorithm was set up in a similar manner to Bate et al., with a simulation duration of 10^{11} time units of 0.1ms. Tasks were defined using UUniFast [6] targeting 90% maximum utilisation in low criticality mode, with tasks having harmonic periods chosen randomly from the base frequencies of 20, 40, 80, 200, 400, 800ms, as commonly found in automotive systems [14]. Deadlines were implicit. Given the number of time units simulated, the duration of the simulation was sufficient to simulate 10^5 instances of the longest task. Tasks were chosen at random to be either low or high criticality, with 50% of tasks being high criticality. As low criticality mode targeted 90% worst case utilisation, once high criticality mode is taken into account, many of the task sets exceeded 100% maximum utilisation. However, a benefit of mixed criticality algorithms is that these systems are still acceptable. With regard to generating the utilisations of each job, three configurations were tested:

1) A control simulation with an independent failure rate of 0.1%, without DepET-RND.
2) A simulation using DepET-RND with dependent failures with overall (average) failure rate of 0.1% and a

maximum number of consecutive failures 200.

3) A simulation using DepET-RND with dependent failures with initial failure rate of 0.1% and a maximum number of consecutive failures 200.

Three different state of the art mixed criticality scheduling algorithms are compared under each configuration:

1) FPPS: Fixed Priority Pre-emptive Scheduling
2) AMC+: An extended version of Adaptive Mixed Criticality scheme [15] proposed in [5] where the execution of LCTs resumes following an idle instant.
3) BM: The Bailout Mode algorithm [5] which uses the slack associated with individual high-criticality jobs to determine when it is okay to return to normal mode whilst preserving the schedulability of all necessary tasks.

We also consider enhancements to both the AMC+ and BM approaches. AMC+S and BMS respectively make use of offline computed slack to increase the budgeted C_{LO} values for individual tasks. AMC+SG and BMSG extend AMC+S and BMS by also using gain time (on-line computed slack) to increase the budgeted C_{LO} values for individual jobs. These protocols are defined in more detail in [5].

Figures 3, 4 and 5 each present, under a given configuration, the percentage of tasks not scheduled by the MCS algorithms evaluated. To capture the variance of this percentage across all the performed simulations, the plots presents for each algorithm the median, quartiles, 9^{th} and 91^{st} percentiles of the dataset.

The differences between Figures 3 and 4, assuming independent then dependent failures with the same overall failure rate, clearly shows that all the algorithms considered perform substantially better on dependent failures than independent failures. Assuming independent failures, a fair portion of simulations resulted in at least 1% to 4% of tasks not being executed (denoted by the red median in the boxes). The worst performing algorithm under dependent failures (AMC+ in Figure 4) still resulted in less than 0.3% of not-executed tasks in the majority of simulations (90% as captured by the top whisker of the plot). This is to be expected as the overheads of entering high criticality mode due to a deadline failure in all the considered algorithms are high, and by introducing dependencies the deadline failures are clustered, resulting in fewer criticality mode transitions. In addition, the relative performance of the scheduling algorithms remains the same as was observed in the independent case.

Figure 5 illustrates the performance of the algorithms when the initial failure rate is 0.1%. Under that configuration, the different algorithms still perform better than with independent failures at the same rate (as shown in Figure 3). As an example when 3% to 4% of the tasks fail to execute for most simulations (denoted by the blue quartile box) using BM and independent failures, this figure falls around 0.5% to 1% under dependent failure. This is despite the fact that due to the initial failures being 0.1%, the total number of failures observed in Figure 5 is greater than in Figure 3. However, this can be explained as follows: due to the clustering effect of DepET, the chances of observing two

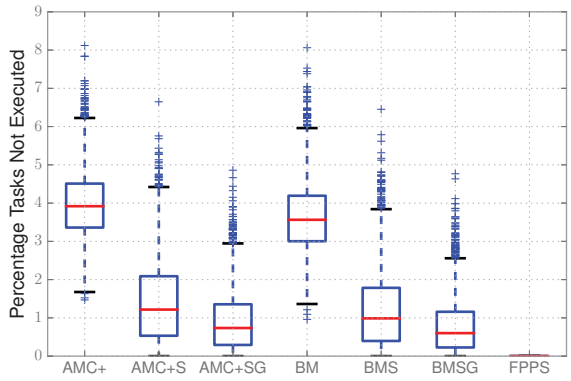


Fig. 3: Percentage of tasks not scheduled, independent failure rate 0.1%

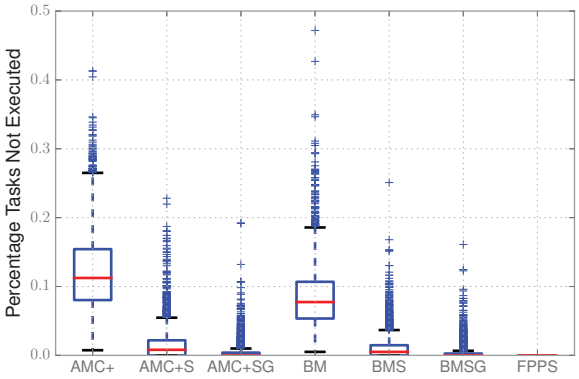


Fig. 4: Percentage of tasks not scheduled, dependent overall failure rate 0.1%

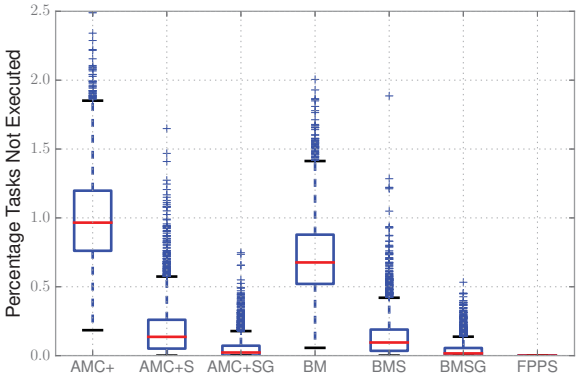


Fig. 5: Percentage of tasks not scheduled, dependent initial failure rate 0.1%

or more faults simultaneously is increased. In turn, this decreases the number of criticality mode transitions of the algorithm, and therefore decreases the overheads allowing more tasks to be scheduled.

Figures 6 and 7 examine the characteristics of two of the algorithms considered, respectively AMC+SG and BMSG, specifically examining how the maximum duration of a fault

observed effects the number of tasks scheduled. While both algorithms exhibit a spike in the percentage of tasks not executed at approximately 200 (the maximum duration of a single fault), the spike resulting from the AMC+SG approach (in Figure 6) is more defined than that seen in the BMSG approach (in Figure 7). Preliminary analysis of this effect suggests that it is caused by the harmonic periods meaning the frequency of idle periods have a regular pattern. For systems of a high utilisation these can be quite small and when C_{LO} is exceeded some of these idle periods can disappear. For the AMC-based algorithms, this could lead to longer times before a return to normal mode whereas with the BM-based policies the normal mode can be returned to any time.

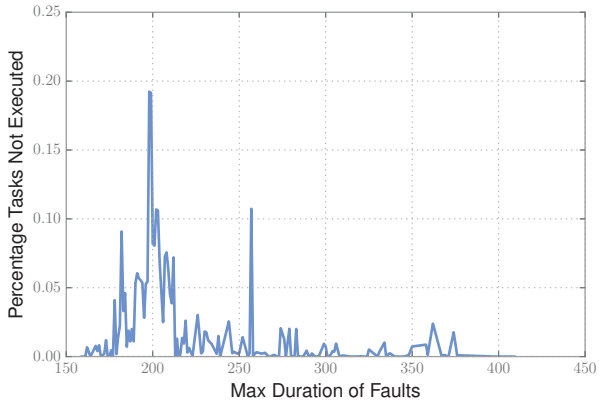


Fig. 6: Percentage of tasks not scheduled vs Max Duration of Faults, using AMC+SG

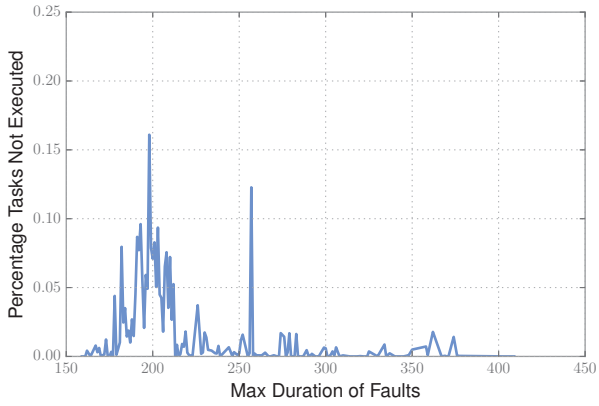


Fig. 7: Percentage of tasks not scheduled vs Max Duration of Faults, using BMSG

V. Conclusions

In this paper, three contributions outlined in the introduction have been made. Firstly an explanation has been provided as to why currently proposed exceedance probabilities for pWCET may be excessively small and different values have been proposed. Secondly, a simulation framework for MCS has been adapted to use the results of a more realistic fault model that has been previously learned using observations from "real" systems. Finally, a scenario-based evaluation of different scheduling policies have been performed. The evaluation has shown that having a dependent fault model does not affect the trends previously seen between different scheduling policies, i.e. the improvement one policy gives over another is approximately the same, however it does affect the sizes of the loss of service to LCTS.

Further, this paper has presented an argument that WCET exceedance probabilities seen in literature on probabilistic real-time systems are unrealistically low, given other components in the system and their interactions in the causes of failures. As minimising the amount of extrapolation required in pWCET from the observed data reduces the inaccuracies, and hence the uncertainty, resulting in tighter and more useful results.

Acknowledgments

This work was funded by the EPSRC grant, MCC (EP/K011626/1), the EU FP7 IP PROXIMA (611085), and the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project. EPSRC Research Data Management: No new primary data was created during this study.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th Real-Time Systems Symposium*, 2007, pp. 239–243.

[2] I. Bate, P. Conmy, T. Kelly, and J. McDermid, "Use of modern processors in safety-critical applications," *The Computer Journal*, vol. 44, no. 6, pp. 531–543, 2001.

[3] P. Graydon and I. Bate, "Realistic safety cases for the timing of systems," *The Computer Journal*, vol. 57, no. 5, pp. 759–774, 2014.

[4] ——, "Safety assurance driven problem formulation for mixed-criticality scheduling," in *Proceedings of the 1st International Workshop on Mixed Criticality Systems*, 2013, pp. 19–24.

[5] I. Bate, A. Burns, and R. Davis, "A bailout protocol for mixed criticality systems," in *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 259–268.

[6] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems Journal*, vol. 30, pp. 129–154, 2005.

[7] A. Burns and S. Edgar, "Predicting computation time for advanced processor architectures," in *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, 2000, pp. 89–96.

[8] D. Griffin, B. Lesage, I. Bate, F. Soboczenski, and R. Davis, "Modelling fault dependencies when execution time budgets are exceeded," in *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, 2015.

[9] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *Proceedings of the Euromicro Conference on Real-Time Systems*, 2012, pp. 91–101.

[10] N. Leveson, *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[11] N. Roberts, W. Vesely, D. Haasl, and F. Goldberg, "Fault tree handbook nureg-0492," *US Nuclear Regulatory Commission*, 1981.

[12] AMSC, "Military handbook, mil-hdbk-338b," *US Department of Defense*, vol. 1, 1998.

[13] D. Maxim, F. Soboczenski, I. Bate, and E. Tovar, "Study of the reliability of statistical timing analysis for real-time systems," in *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, 2015.

[14] I. Bate and A. Burns, "An integrated approach to scheduling in safety-critical embedded control systems," *Real-Time Systems Journal*, vol. 25, no. 1, pp. 5–37, Jul 2003.

[15] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings of the 32nd Real-Time Systems Symposium*, 2011, pp. 34–43.

Probabilities and Mixed-Criticalities: the Probabilistic C-Space

Luca Santinelli[1] and Laurent George[2]
[1]ONERA - Toulouse, luca.santinelli@onera.fr
[2] University of Paris Est / LIGM - ESIEE Paris, Laurent.George@univ-mlv.fr

***Abstract*—Probability distributions bring flexibility as well as accuracy in modeling and analyzing real-time systems. On the other end, the adding of probabilities increases the complexity of the scheduling problem, especially in case of mixed-criticalities where tasks of different criticalities have to be taken into account on the same computing platform. In this work we explore the flexibility of probabilistic distributions applied to mixed-critical task sets for defining the probabilistic space of Worst Case Execution Time and evaluating the effects of changes on the task execution conditions. Finally, we start formalizing and making use of probabilistic sensitivity analysis for evaluating mixed-critical scheduling performance.**

I. Introduction

During the last decade, real-time systems designers are facing the arrival of new COTS architectures and new functionalities which may result into important variability of the execution time of programs. Worst-case reasoning may have reached its limits since it considers worst-case values without taking into account that the probability of occurrence of such values may be vanishingly small [1].

In [2], the authors characterize the space of feasible Worst-Case Execution Times (WCETs) for the Earliest Deadline First (EDF) scheduling paradigm, denoted C-space. The C-space is a convex polytop of n dimensions (the number of tasks), such that for any vector of execution times inside the polytop, the task set is always feasible with EDF scheduling (deterministic approach). The deterministic approach supposes that all the jobs of the tasks can experience their WCETs at run time which is very unlikely.

Approaches that take into account tasks probabilistic worst case execution time (distributions of values instead of single values) may lead to important reduction of computing capability over-provisioning.

First papers on probabilistic approaches describe execution times of tasks by random variables, according to discrete [3], [4] or continuous [5] models. Since Edgar and Burns [6], several papers have worked on obtaining safe and reliable probabilistic Worst-Case Execution Time (pWCET) estimates [7], [8], [9].

The probabilistic worst-case reasoning leads to schedulability analysis with probabilities. Diaz et al. [10] developed the first analysis for systems with execution times described by random variables. Recent works have extended schedulability to probabilistic arrival times, using an average arrival model as in [11], or to probabilistic minimal inter-arrival times model as in [12]. Other approaches lately have revised the notion of bounding curves with probabilities, i.e. arrival curves, demand bound function and workload bound function, [13], for probabilistic guarantees of the timing constraints.

The Mixed-Criticality (MC) problem comes from the need for using the platform resources efficiently. This is facilitated by noting that the task parameters depend on their criticality level, in particular the WCET estimate will be derived by a process dictated by the criticality level. The higher the criticality level, the more conservative the verification process and hence the greater will be the WCET, [14]. In [15], the confidence on the WCET estimations has been leveraged to develop MC scheduling algorithms. In this work we intend to continue researching in that direction formalizing the relationship between pWCETs and mixed-criticalities.

Contribution: This paper formalizes the scheduling problem with tasks of different criticalities through probabilistic models. The pWCETs are applied to construct the probabilistic version of the C-space. Such fine grained probabilistic representations (pWCETs and probabilistic C-Space) are applied to leverage probabilistic information for the MC scheduling problem. This work intends to provide an initial evaluation to the flexibility brought by the probabilistic models and the probabilistic scheduling to the mixed-criticality problem. In it, the sensitivity analysis is enhanced with probabilities for the first time and it is applied to the probabilistic C-space for some initial thoughts and possible strategies for resource allocation with mixed-critical tasks.

II. Modeling with Probabilities

Jobs of tasks can exhibit multiple durations at run time due to interferences from the system elements and the environment. It is then reasonable to describe task execution time with random processes[1].

The probabilistic worst-case execution time random variable $\mathcal{C}_i$ of a task τ_i generalizes the deterministic WCET. It is defined as the worst-case distribution that upper-bounds any possible task execution time the task can exhibit, [16]. Hence, worst-case execution time distributions represents a way to account for the system variabilities as the worst-case model to all of them. In its abstract interpretation, $\mathcal{C}_i$ would includes multiple WCET values, each with the probability of being the worst-case[2]. For example, given a trace of task execution time which would be an empirical distribution due to the task execution time variability, the worst-case execution time distribution could be the distribution made out of the maximum of blocks of execution times, each block representing a specific task execution condition.

[1]A random process is a sequence of random variables describing a process whose outcomes do not follow a deterministic pattern, but follow probability distributions.

[2]Calligraphic letters are used to represent distributions while non calligraphic letters are for scalars.

The $\mathsf{pdf}_{\mathcal{C}_i}$ is the probability distribution function (pdf) representation of the random variable $\mathcal{C}_i$. Without loss of generality, we could consider discrete pWCET distributions, that is:

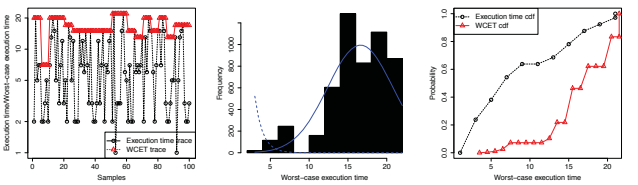
$$\mathsf{pdf}_{\mathcal{C}_i} = \begin{pmatrix} c_{i,1} & \dots & c_{i,k_i} \\ P(\mathcal{C}_i = c_{i,1}) & \dots & P(\mathcal{C}_i = c_{i,k_i}) \end{pmatrix}, \quad (1)$$

with $\mathsf{pdf}_{\mathcal{C}_i}(c_{i,r}) = P(\mathcal{C}_i = c_{i,r})$, $\sum_{r=1}^{k_i} \mathsf{pdf}_{\mathcal{C}_i}(c_{i,r}) = 1$, and k_i is the number of worst-case execution time values in the pWCET distribution of τ_i. It is $P(\mathcal{C}_i \leq C_i) = 1$, and the other values $c_{i,k}$, $1 \leq k \leq k_i$ are such that $C_{i,k} \leq C_i$.

$\mathsf{cdf}_{\mathcal{C}_i}$ denotes the cumulative distribution function (cdf) representation of $\mathcal{C}_i$ such that $\mathsf{cdf}_{\mathcal{C}_i}(c) = P(\mathcal{C}_i \leq c) = \sum_{x=1}^{c} \mathsf{pdf}_{\mathcal{C}_i}(x)$, with discrete distributions. The inverse cumulative distribution function (icdf) $\mathsf{icdf}_{\mathcal{C}_i}(c)$ outlines the exceedence thresholds, $\mathsf{icdf}_{\mathcal{C}_i}(c) = P(\mathcal{C}_i \geq c)$ as the probability of having worst-case execution time larger than c. With discrete random variable, it is $\mathsf{icdf}_{\mathcal{C}_i}(c) = 1 - \sum_{x=1}^{c} \mathsf{pdf}_{\mathcal{C}_i}(x)$.

We also assume that the pWCET are finite distributions, with finite support, such that the safe[3] worst-case (the worst-case such that its cumulative probability is 1) is finite and is the deterministic WCET C_i; $C_i \in \mathcal{C}$ and $\mathsf{cdf}_{\mathcal{C}_i}(C_i) = 1$. The finite support assumption allows to have the deterministic WCET belonging to the pWCET distribution, thus it is possible to do hard real-time analysis. Recent works have investigated how to derive continuous pWCETs estimates from execution time measurements in different system conditions, [17]. Discrete and finite pWCETs can always be derived as approximations at relatively low probabilities of such continuous pWCET estimates, [15].

A task τ_i is also characterized by a period T_i and a relative deadline $D_i \leq T_i$; thus the task model $\tau_i = (\mathcal{C}_i, T_i, D_i)$. In this paper we consider a set of n periodic tasks $\Gamma = \{\tau_1, \dots, \tau_n\}$, with the hyperperiod H being the least common multiple of all task periods, $H = \mathsf{lcm}(T_1, \dots, T_n)$. Γ is scheduled with EDF.



(a) Execution time and worst-case execution time trace (b) Worst-case execution time representation (c) Execution time and worst-case execution time cdfs

Fig. 1. Representation of discrete task execution time and worst-case execution time distributions. Execution times are in time units.

Figure 1 shows an example of discrete pWCET. Figure 1(a) depicts a possible meaning for the worst-case distribution from a trace of execution time: a worst-case value could come from the maximum of a block of execution times, and due to system variabilities and different execution conditions, the maximum could change. Thus the maximum could end up in a probabilistic model. Figure 1(b) is for the histogram representation of the pWCET, which in that case resembles to a normal distribution. Figure 1(c) outlines the differences between execution time distribution and pWCET with the cdf representation.

[3]Safety comes from the overestimation of any possible measured behavior.

From $\mathcal{C}_i$ it is possible to define *WCET thresholds* $\langle c_{i,k}, p_{i,k} \rangle$, $1 \leq k \leq k_i$. The worst-case value $c_{i,k}$ is associated to the probability $p_{i,k}$ of being the WCET for task τ_i. $p_{i,k} \stackrel{def}{=} \mathsf{cdf}_{\mathcal{C}_i}(c_{i,k})$ quantifies the accuracy of the WCET $c_{i,k}$, equivalently the *confidence* on $c_{i,k}$ of being the WCET. $1-p_{i,k}$ is the probability of passing $c_{i,k}$. Depending on the granularity of the pWCET representation it would be possible to define WCET thresholds at 10^{-3}, 10^{-6}, 10^{-9}, and beyond.

A. Tasks Relationship

Most of the algebra in probability theory relies on the degree of dependence between random variables, the so called *statistical dependence*. Therefore, the task relationship can be evaluated with the degree of statistical dependence between pWCET distributions.

The joint probability, which expresses the composition of random variables, is affected by the degree of dependence among the random variables. For a couple of worst-case execution time distributions $\mathcal{C}_i$ and $\mathcal{C}_j$, respectively for τ_i and τ_j, the joint probability $P(\mathcal{C}_i = c_{i,r}, \mathcal{C}_j = c_{j,s})$ defines the probability of the worst-case execution times $\mathcal{C}_i = c_{i,r}$ and $\mathcal{C}_j = c_{j,s}$; both $\mathcal{C}_i$ and $\mathcal{C}_j$ represent events at the same time, thus concurrently executing tasks. It is $P(\mathcal{C}_i = c_{i,r}, \mathcal{C}_j = c_{j,s}) = P(\mathcal{C}_j = c_{j,s} | \mathcal{C}_i = c_{i,r}) \times P(\mathcal{C}_i = c_{i,r}) = P(\mathcal{C}_i = c_{i,r} | \mathcal{C}_j = c_{j,s}) \times P(\mathcal{C}_j = c_{j,s})$, with $P(\mathcal{C}_i = c_{i,r} | \mathcal{C}_j = c_{i,s})$ is the conditional probability defined as the probability of having $\mathcal{C}_i = c_{i,r}$ once also τ_j is executing and having $\mathcal{C}_j = c_{j,s}$. Equivalently, with the pdfs and the cdfs it is respectively:

$$\mathsf{pdf}_{\mathcal{C}_i,\mathcal{C}_j} = \mathsf{pdf}_{\mathcal{C}_i|\mathcal{C}_j} \otimes \mathsf{pdf}_{\mathcal{C}_j} = \mathsf{pdf}_{\mathcal{C}_j|\mathcal{C}_i} \otimes \mathsf{pdf}_{\mathcal{C}_i} \quad (2)$$

$$\mathsf{cdf}_{\mathcal{C}_i,\mathcal{C}_j} = \mathsf{cdf}_{\mathcal{C}_i|\mathcal{C}_j} \times \mathsf{cdf}_{\mathcal{C}_j} = \mathsf{cdf}_{\mathcal{C}_j|\mathcal{C}_i} \times \mathsf{cdf}_{\mathcal{C}_i}; \quad (3)$$

$\otimes$ being the convolution operator between random variables and $\mathsf{pdf}_{\mathcal{C}_j|\mathcal{C}_i}$ is the conditional pWCET of τ_i concurrently executing with τ_j.

Two tasks τ_i and τ_j are independent, $\tau_i \overline{\rhd} \tau_j$ (equivalently $\tau_j \overline{\rhd} \tau_i$), if the execution of one task does not have any impact on the execution of the other task. Whenever the two tasks worst-case execution times are random variables $\mathcal{C}_i$ and $\mathcal{C}_j$ respectively, the independence states that the execution of one task does not affect the distribution of the other task (statistical independence): $\mathsf{pdf}_{\mathcal{C}_i|\mathcal{C}_j} = \mathsf{pdf}_{\mathcal{C}_i}$ and $\mathsf{pdf}_{\mathcal{C}_j|\mathcal{C}_i} = \mathsf{pdf}_{\mathcal{C}_j}$. Two tasks τ_i and τ_j are dependent, $\tau_i \rhd \tau_j$ (equivalently $\tau_j \rhd \tau_i$), if the execution of one task does have impact on the execution of the other task. With $\mathcal{C}_i$ and $\mathcal{C}_j$ the pWCETs of respectively τ_i and τ_j, with dependence the execution of one task does affect the distribution of the other task (statistical dependence): $\mathsf{pdf}_{\mathcal{C}_i|\mathcal{C}_j} \neq \mathsf{pdf}_{\mathcal{C}_i}$ and $\mathsf{pdf}_{\mathcal{C}_j|\mathcal{C}_i} \neq \mathsf{pdf}_{\mathcal{C}_j}$.

1) Worst-Case Independence: Interferences to task execution from concurrently executing tasks or concurrent access to shared resources have the effect of increasing the task execution time. Assuming $\mathcal{C}_i$ to be the probabilistic worst-case distribution of τ_i, it means that the distribution has already accounted for all the possible interferences, including those from other tasks. It implies also that every time there is an interference (a concurrent task or an other system element acting at the same time as τ_i), $\mathcal{C}_i$ as already embedded its effects [18]. The distribution does not change anymore in the presence of such effects being already the pWCET: $\mathsf{pdf}_{\mathcal{C}_i|\mathcal{C}_j} = \mathsf{pdf}_{\mathcal{C}_i}$. We can say that the $\mathcal{C}_i$, with respect to the empiric execution time distributions (from the measurements of

the system actual behavior), quantifies the effect of dependence to the task executions.

III. Probabilistic Mixed-Criticality

In this paper, we consider the two-criticality-level case (high and low) of the MC problem, each task is designated as being either of high (HI) or low (LO) criticality. With the deterministic model two WCET values are specified for each task: a LO-WCET $c(\text{LO})$ determined by a less pessimistic timing analysis tools, and a larger HI-WCET $c(\text{HI})$ determined by more conservative timing analysis tools, which is sometimes larger than the LO-WCET by several orders of magnitude in COTS platforms. For τ_i, the WCET of a task is a vector $C_i = (c_i(\text{LO}), c_i(\text{HI}))$.

Existing MC analysis usually makes the most pessimistic assumption that *every* HI-criticality task may execute beyond its LO-WCET and reach its HI-WCET *simultaneously*. In real applications, the industry standards usually only require the expected probability of missing deadlines within a specified duration to be below some specified small value, as the deadline miss can be seen as a faulty condition. The expected probability of deadline miss depends on the criticality level $crit$, e.g. $crit \in \{\text{LO}, \text{HI}\}$, under which the system is running.

The pWCET distribution effectively defines different WCET threshold estimates for the same task, for different criticality levels depending on the different requirements confidence, e.g. on the maximum tolerable failure rate as the pWCET estimates embeds the effect of faults on the task executions. That translates into a MC two-criticality task model such that $\tau_i = ((\langle c_i(\text{LO}), p_i(\text{LO})\rangle, (\langle c_i(\text{HI}), p_i(\text{HI})\rangle), T_i, D_i)$, where the WCET values have a confidence of being worst-cases.

The cdf representation of the pWCET relates *probability* to *confidence* of the criticality levels. $p_i(\text{LO}) = P(\mathcal{C}_i \leq c_i(\text{LO})) \equiv \mathsf{cdf}_{\mathcal{C}_i}(c_i(\text{LO}))$ expresses the confidence of $c_i(\text{LO})$ of being the upper-bound of the task worst-case execution time in its LO-criticality. Similarly, $p_i(\text{HI}) = P(\mathcal{C}_i \leq c_i(\text{HI})) \equiv \mathsf{cdf}_{\mathcal{C}_i}(c_i(\text{HI}))$ the confidence of $c_i(\text{HI})$ of being the upper-bound of the task worst-case execution time in its HI-criticality We call $p_i(crit)$, $crit \in \{\text{LO}, \text{HI}\}$ the confidence on the criticality level $crit$.

A. Probability thresholding

It is possible to define a design parameter β as the *probability threshold* for the pWCET defining the level of confidence for a WCET limit imposed to a task. β comes from the quantile $q(p)$ as the probability threshold p, and $q(\mathcal{C}_i, \beta)$ is the WCET threshold such that $\beta \times 100\%$ of the worst-case execution time experienced by τ_i are below that threshold.

β offers another perspective to the task execution model. By fixing β it is possible to specify which is the limit WCET reachable, $c_i(\beta)$; β imposes a bound to the task WCET such that $c_i(\beta) = q(\mathcal{C}_i, \beta)$.

A trace $\mathcal{T}_{\mathcal{C}_i}$ reports the sequence of WCET values that τ_i has assumed from one execution instance to another. From $\mathcal{T}_{\mathcal{C}_i}$ it is then possible to infer the timing behavior of the task WCETs as well as identify $c_i(\beta) = q(\mathcal{T}_{\mathcal{C}_i}, \beta)$. Therefore, β can model the task (or the whole application) timing behavior and it could be applied as design parameter: by imposing $c_i(\beta)$ as the task WCET value the behavior of τ_i is limited to $c_i(\beta)$. With respect of the actual task behavior $\mathcal{T}_{\mathcal{C}_i}$ (which follow $\mathcal{C}_i$, β is the confidence that τ_i respects it WCET limit $c_i(\beta)$.

From β it is also possible to infer the criticality level $crit$ that would allow respecting $c_i(\beta)$:

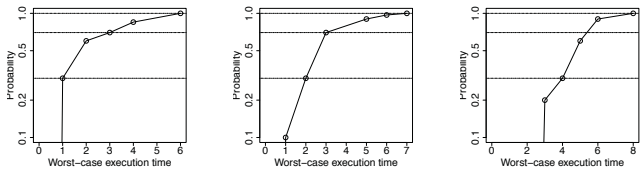
$$max\{crit\} \text{ such that } c_i(crit) \leq c_i(\beta). \tag{4}$$

It is $\beta \neq p(crit)$, as $c(\beta) \neq c(crit)$, but there is a close relationship between the two thresholds $c(\beta)$ and $c(crit)$ which come from the probabilistic modeling of the task (the pWCET).

Example 1. *Given a task set* $\Gamma = \{\tau_1, \tau_2, \tau_3\}$*, with* $\tau_1 = (\mathcal{C}_1, 7, 5)$*,* $\tau_2 = (\mathcal{C}_2, 11, 7)$*,* $\tau_1 = (\mathcal{C}_3, 13, 10)$*, and the discrete worst-case execution time distributions are*

$$\mathit{pdf}_{\mathcal{C}_1} = (\begin{pmatrix} 1 & 2 & 3 & 4 & 6 \\ 0.3 & 0.3 & 0.1 & 0.15 & 0.15 \end{pmatrix}$$
$$\mathit{pdf}_{\mathcal{C}_2} = (\begin{pmatrix} 1 & 2 & 3 & 5 & 6 & 7 \\ 0.1 & 0.2 & 0.4 & 0.2 & 0.07 & 0.03 \end{pmatrix}$$
$$\mathit{pdf}_{\mathcal{C}_3} = (\begin{pmatrix} 3 & 4 & 5 & 6 & 8 \\ 0.2 & 0.1 & 0.3 & 0.3 & 0.1 \end{pmatrix}.$$

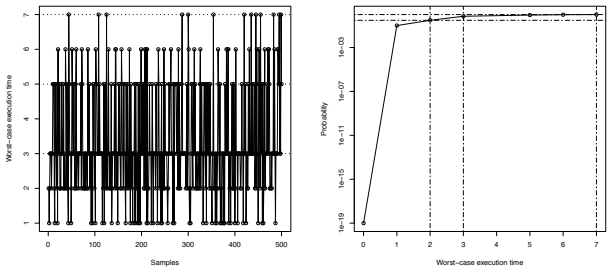
Considering criticality levels such that for τ_1 *it is* $\{c(\text{LO}) = 1, c(\text{HI}) = 4\}$*, for* τ_2 *it is* $\{c(\text{LO}) = 1, c(\text{HI}) = 7\}$*, and for* τ_3 *it is* $\{c(\text{LO}) = 4, c(\text{HI}) = 8\}$*. Figure 2 represents the tasks pWCETs*



(a) τ_1 (b) τ_2 (c) τ_3

Fig. 2. A portion of the pWCET cdf for the three tasks. Three probability thresholds are outlined, respectively at $p = 0.3$ $p = 0.7$, and $p = 1$.

with a zoom of the cdfs on the largest cumulative probabilities. In Figure 3(a) an example of thresholds β *for* $\mathcal{C}_2$ *and their effects on the task [worst-case] executions. It is represented a trace of* 500 *worst-cases extracted from* $\mathcal{C}_2$ *by randomly picking values from the distributions law, and associated to the task execution. As already mention, the trace of worst-case execution times could represent the trace of maximum among different task execution conditions. For a* $\beta_3 = 1$ *the WCET limit is* 7*, the maximum allowed value. For* $\beta_2 = 0.8$ *the task WCET would be limited to* 5*, and for* $\beta_2 = 0.5$ *the task WCET would be limited to* 3*. For example, with* $c_2(\beta_2) = 5$ *there is a confidence of* 0.8 *of remaining below that WCET values while* τ_2 *executes. Hence, imposing* $C_2 = 5$ *as the max WCET for* τ_2 *it will be respected* 80% *of the case.*



(a) βs from traces; $\beta_1 = 0.5, \beta_2 = 0.8$, and $\beta_3 = 1$

(b) cdf with criticality levels (horizontal lines - WCET thresholds) and βs (vertical lines - probability thresholds)

Fig. 3. Trace and cdf representations of the pWCET.

Figure 3(b) outlines the relationship between β and the criticality levels.

IV. The Probabilistic C-space

Real-time systems with probabilistic models require schedulability conditions which involve probabilities. Given a random process $\mathcal{C}_i$ describing the evolution of τ_i worst-case execution time, we can state the notion of probabilistic demand bound function, [13], [19].

$\mathsf{dbf}_{i,j}(t)$ is the demand bound function in the interval $[0, t]$:

$$\mathsf{dbf}_{i,j}(t) \stackrel{def}{=} \lfloor \frac{t - D_i}{T_i} + 1 \rfloor \times c_{i,j}. \tag{5}$$

The bound is the result of a specific WCET threshold $c_{i,j}$, and by definition, it represents an upper-bound to any $\mathsf{dbf}_{i,k}$ obtained from $c_{i,k} \leq c_{i,j}$. Then, there exist an associate confidence of the bound, which is the exact confidence $p_{i,j}$ of the WCET $c_{i,j}$ applied. $c_{i,j}$ and Equation (5) define a probabilistic bound $\langle \mathsf{dbf}_{i,j}(t), p_{i,j} \rangle$ to the task resource demand; $\langle \mathsf{dbf}_{i,j}(t), p_{i,j} \rangle$ is such that $p_{i,j}$ is the probability that $\mathsf{dbf}_{i,j}(t)$ is an upper-bound to the τ_i resource demand in $[0, t]$. Equivalently to $\mathsf{dbf}_{i,j}(t)$ we can write $\mathsf{dbf}_i(t, c_{i,j})$.

As $\langle \mathsf{dbf}_i(t, c_{i,j}), p_{i,j} \rangle$ represents a single demand bound function with its associated confidence, there exist a $\mathsf{dbf}_i(t, c_{i,j})$ for each $c_{i,j} \in \mathcal{C}_i$. All together the $\langle \mathsf{dbf}_i(t, c_{i,j}), p_{i,j} \rangle$ form a distribution of demand bound functions, $\mathcal{DBF}_i(t) = (\mathsf{dbf}_i(t, \cdot), p_i(\cdot))$ which is the probabilistic demand bound function (probabilistic demand curve) of τ_i in $[0, t]$. $\mathcal{DBF}_i(t)$ collects the set of all demand bound functions $\mathsf{dbf}_i(t)$ and the set of all confidences p_i. In particular, the p_is forms the the cdf of $\mathcal{DBF}_i(t)$, $\mathsf{cdf}_{\mathcal{DBF}_i(t)}$, as cumulative probabilities. To note that the set of probabilities p_i does not change with the interval $[0, t]$, therefore form one interval to another is only the bounds $\mathsf{dbf}_i(t)$ to change, but not their confidence.

The application Γ probabilistic demand curve $\mathcal{DBF} = (\mathsf{dbf}(t, \cdot), p(\cdot))$ results from the combination of tasks demand bound functions $\mathcal{DBF}_i$:

$$\mathcal{DBF}(t) = \otimes_i \mathcal{DBF}_i(t), \tag{6}$$

with $\otimes$ the convolution of the distributions. $\mathsf{dbf}(t, \cdot)$ is the set of all the demand bound functions:

$$\mathsf{dbf}(t, \cdot) \stackrel{def}{=} +_i \mathsf{dbf}_i(t, \cdot), \tag{7}$$

with $+$ the combination (sum) of all the demand bound function. $p(\cdot)$ is the set of all the confidence probabilities:

$$p(\cdot) \stackrel{def}{=} \times_i p_i(\cdot), \tag{8}$$

with $\times$ the combination (product) of all the demand bound function probabilities.

The demand bound function $\mathsf{dbf}(t, \overline{c})$ is the application demand with $\overline{c} = (c_{1,j}, c_{2,k}, \ldots)$ the array of WCET thresholds used for achieving $\mathsf{dbf}(t, \overline{c})$; $p(\overline{c})$ is the confidence of $\mathsf{dbf}(t, \overline{c})$ such that:

$$p(\overline{c}) = p_1(c_{1,j}) \times p_2(c_{2,k}) \times \ldots. \tag{9}$$

The probability multiplication for the joint probability $p(\overline{c})$ is possible due to the worst-case distribution assumption. As $\mathcal{C}_i$ are pWCETs they are independent, the distributions $\mathcal{DBF}_i$ are independent among each other; consequently the joint probability $p(\cdot)$ could result from the probability multiplication, Equation (9).

A. Probabilistic C-space Representations

The schedulability under EDF states that

$$\forall t \in D, \quad \mathsf{dbf}(t) \leq t, \tag{10}$$

with D the set of Γ deadlines within the hyperperiod, according to [20], [21].

With a probabilistic framework each condition $\mathsf{dbf}(t, \cdot) \leq t$ has a probability $p(t)$ associated, which is the confidence on the demand bound function $\mathsf{dbf}(t, \cdot)$. Being $p = p(\overline{c})$ the probability of not passing $\mathsf{dbf}(t, \overline{c})$, with the condition $\mathsf{dbf}(t) \leq t$ the probability p could be also interpreted as the probability of verifying the condition.

For all $t \in D$, there exist $\overline{c}^*$ such that $\mathsf{dbf}(t, \overline{c}^*) = max\{\mathsf{dbf}(t, \overline{c}) \mid \mathsf{dbf}(t, \overline{c}) \leq t\}$. $P(t) = p(\overline{c}^*)$ from Equation (9) is the probability for which $\mathsf{dbf}(t) \leq t$ is true. The overall schedulability probability P is given such that all the conditions are satisfied:

$$P = P(t_1) \times P(t_2) \times \ldots, \tag{11}$$

with $P(t_k)$ the schedulability probability of the k-th condition $\mathsf{dbf}(t_k) \leq t_k$, $t_k \in D$. The independence between conditions and the probability product as the joint probability, are guaranteed by the use of pWCET distributions. $1 - P$ is the probability that at least one condition is not respected, thus the probability of deadline miss.

From Condition (10) and Equation (11) it is possible to build the probabilistic version of the C-space (pC-space), [2]. The pC-space is the abstraction that applies the schedulability condition, Condition (10), to a vector of execution times $\overline{c} = \{c_1, c_2, \ldots\}$. Each point $\overline{c} = \{c_1, c_2, \ldots\}$ in the pC-Space is a combination of task WCET thresholds. Within the pC-Space, given the scheduling policy, it is possible to define the schedulability region where every point $\overline{c}$ within the region is a schedulable WCET thresholds configuration, and the points outside the region do not represent schedulable WCET thresholds configurations. [2] for the details on the definition of the deterministic C-space under EDF.

The pC-space maps also probabilities onto points. Each $\overline{c}$ within the space has a probability associated which is the probability of being the application set of worst-case execution times, Equation (9). Then, depending on where the point is with respect to the schedulability region, the probability could translate into schedulability probability. For the points at the feasibility region border, their ps, Equation (9), are exactly the schedulability probability, Equation (11). With the different probabilities P within the region and at the border it would be possible to classify portions of the regions with respect to the schedulability probability P.

B. pC-Space and Confidence

The probabilities within the pC-Space can be interpreted in various ways:

- as the confidence of not passing the WCET thresholds of $\overline{c}$. With the criticality levels there is also the probability of remaining at a certain criticality level $p(crit) = p_1(crit) \times p_2(crit) \times \ldots$. Consequently it is quantifiable the possibility of changing that level as $1 - p(crit)$;
- as the confidence on the system schedulability P, or the confidence per schedulability condition, P_k. The feasibility region is characterized by P and all the points inside

the region are schedulable but with a confidence of at least P, Equation (11). It translates into a per-condition schedulability probability of P_k;
- as the confidence β on the worst-case behavior of the tasks. β is the probability of passing the $\overline{c}(\beta)$; per-task it would be $c_j(\beta)$.

With different probability interpretations the pC-Space can be used for different purposes. At one end there is the modeling of the probabilistic applications; on the other end, it is possible to develop analysis on top of the pC-Space with probabilities.

Example 2. *Given the probabilistic task set of Example 1. The feasibility region of Γ does not depend on the input distributions (and their shape) but it describes the feasibility point according to the task period and deadline configuration, (x, T_i, D_i). What is depending on the pWCETs instead, are the probabilities of each point. In Figure 4 all the possible WCET thresholds and*

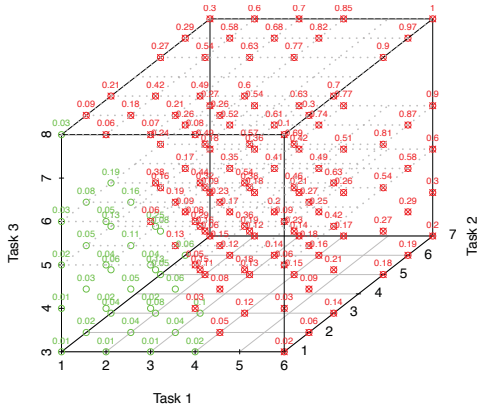
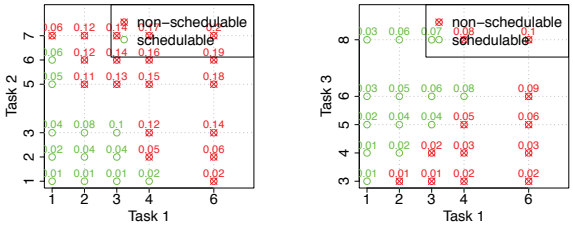


Fig. 4. Probabilistic C-space: feasibility and confidences (probabilities) within the pC-space. Circles for feasible $\overline{c}$, crosses for not feasible $\overline{c}$) and βs limitations ($\beta_1 = 0.5$,$\beta_2 = 0.8$, and $\beta_3 = 1$) for τ_3 are presented as horizontal 2D planes.



(a) 2D plane (τ_1, τ_2) with $\mathcal{C}_3 = 3$ (b) 2D plane (τ_1, τ_3) with $\mathcal{C}_2 = 1$

Fig. 5. Probabilistic C-space: 2D planes with bounding βs ($\beta_1 = 0.5$,$\beta_2 = 0.8$, and $\beta_3 = 1$).

the WCET thresholds combinations from the input pWCETs are plotted. To notice that the whole plane at $\mathcal{C}_1 = 6$ is an unfeasible plane, being outside the feasibility region. The confidences are presented together with the feasible points $\overline{c}$ (points with circles and in green). Points with crosses (in red) are WCET thresholds configurations unfeasible. Figure 5 gives a better insight with 2D representations of both the pC-Space and the feasibility regions. In both figures the βs are represented as constraints to the task execution behavior.

V. Probabilistic Sensitivity Analysis

The probabilistic version of the sensitivity analysis [2] intends to combine the information from the probabilistic models (the pWCETs, β, and the confidences β and ps) and the pC-space representation.

We have seen that $\mathcal{C}_i$s discretize the pC-Space as they maps the points to only the possible WCET thresholds of the tasks. Out of that, the probabilistic sensitivity analysis can be applied to quantify the effects of changes in terms of schedulability, probabilities/confidences, and criticalities.

- What are the resource demand that can be accommodated? Hence, which task combinations can be accounted for a schedulable systems, the criticality levels that can be considered in order to make the system schedulable, etc.
- What can be done with β? By acting on β (limiting task WCETs to $c(\beta)$) it is possible to evaluate the effect on the execution of tasks. What are the effect of β on the tasks criticality levels? With the relationship $\beta \rightarrow crit$ is is possible to infer the criticality levels which subdue to the $\overline{c}(\beta)$ bounding.

Furthermore, with the probabilistic sensitivity analysis it is possible to evaluate the effect of changes on Γ. For example a change on β, from β to β' would result into a WCET threshold change $\overline{c}$ to $\overline{c}'$, such that $\overline{c} = (c_{1,j}, c_{2,k}, \ldots)$ and $\overline{c}' = (c_{1,r}, c_{2,s}, \ldots)$. The change of probabilities, from $p(\overline{c})$ to $p(\overline{c}')$, is an immediate consequence of the change of β. It would also be evident the effects of changes on the allowed criticality levels, from $\beta \rightarrow crit$.

While P does not change by moving the points toward the feasibility region (by limiting task execution behavior with β), it is possible to increase the confidence that the feasibility condition is respected.

Example 3. *The probabilistic sensitivity analysis, with respect to the previous example, could help replying to the following questions:*

- *With a certain mixed-criticality level, is the system schedulable? For example, in Figure 7(c), if τ_3 is in HI-criticality mode, then τ_2 can only be scheduled with $\mathcal{C}_2 = 1$ (LO-criticality) for guaranteeing schedulability. Another example from Figure 7(a), where only LO-criticality modes are schedulable for τ_1 and τ_2.*
- *What can be done to make the system schedulable? What are the costs of being schedulable? β and its limitation effects on the task WCETs can give answers to those questions. From Figure 7(b), only limiting WCETs with $\beta < 0.5$ would guarantee Γ schedulability. This translates into LO-criticality execution for both τ_1 and τ_3.*

Figure 6 representing the discretized feasibility region (to the possible WCET thresholds) and the feasible $\overline{c}$s. Figure 7 for 2D representations. β limitation effects are evident to the tasks WCET thresholds and the criticality level allowed.

VI. Conclusion

In this work we have begun combining the probabilities and the mixed-criticality problem with the help of the probabilistic C-space and of the probabilistic sensitivity analysis. Some of the observations on the probabilistic task sets are just initial ideas which could help designing and applying more effective MC scheduling.

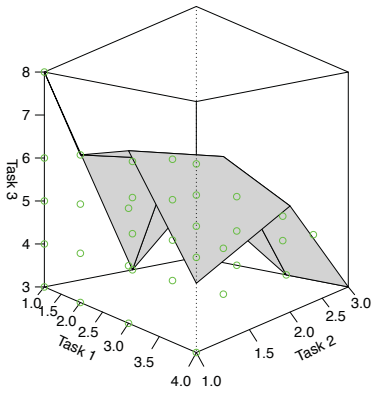
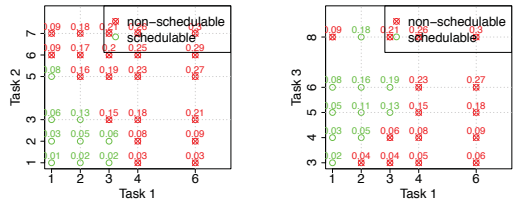
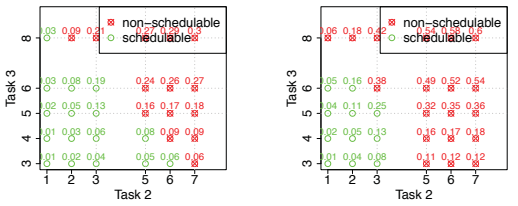


Fig. 6. Probabilistic sensitivity analysis from the feasibility region (points with circles) and the βs ($\beta_1 = 0.5$,$\beta_2 = 0.8$, and $\beta_3 = 1$).



(a) 2D plane (τ_1, τ_2) with $\mathcal{C}_3 = 3$ (b) 2D plane (τ_1, τ_3) with $\mathcal{C}_2 = 2$



(c) 2D plane (τ_2, τ_3) with $\mathcal{C}_1 = 1$ (d) 2D plane (τ_2, τ_3) with $\mathcal{C}_1 = 2$

Fig. 7. Probabilistic sensitivity analysis on the 2D planes.

In the future we intend to enhance such observations and define the probabilistic sensitivity analysis in terms of change strategies and effect evaluation. We aim at leveraging the information from the probabilistic models (pWCET distributions and confidences) and provide system design feedbacks for an optimal (at least suboptimal) system resource utilization for different criticalities, thus different requirements.

REFERENCES

[1] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. D. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "Proartis: Probabilistically analyzable real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 2s, p. 94, 2013.

[2] L. George and J. Hermant, "Characterization of the space of feasible worst-case execution times for earliest-deadline-first scheduling," *Journal of Aerospace Computing, Information and Communication (JACIC)*, vol. 6, no. 11, 2009.

[3] T. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L. Wu, and J. Liu, "Probabilistic performance guarantee for real-time tasks with varying computation times," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 1995.

[4] L. Abeni and Buttazzo, "QoS guarantee using probabilistic deadlines," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS99)*, 1999.

[5] J. Lehoczky, "Real-time queueing theory," in *10th of the IEEE Real-Time Systems Symposium (RTSS96)*, 1996, pp. 186–195.

[6] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling," in *22nd of the IEEE Real-Time Systems Symposium*, 2001.

[7] J. Hansen, S. Hissam, , and G. Moreno, "Statistical- based wcet estimation and validation," in *the 9th International Workshop on Worst-Case Execution Time*, 2009.

[8] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *ECRTS*, 2012, pp. 91–101.

[9] D. Hardy and I. Puaut, "Static probabilistic worst case execution time estimation for architectures with faulty instruction caches," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2013.

[10] J. Díaz, D. Garcia, K. Kim, C. Lee, L. Bello, L. J.M., and O. Mirabella, "Stochastic analysis of periodic real-time systems," in *23rd of the IEEE Real-Time Systems Symposium (RTSS02)*, 2002, pp. 289–300.

[11] G. Kaczynski, L. Lo Bello, and T. Nolte, "Deriving exact stochastic response times of periodic tasks in hybrid priority-driven soft real-time systems," *12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'07), Greece*, 2007.

[12] D. Maxim and L. Cucu-Grosjean, "Response time analysis for fixed-priority tasks with multiple probabilistic parameters," in *IEEE Real-Time Systems Symposium*, 2013.

[13] L. Santinelli and L. Cucu-Grosjean, "A probabilistic calculus for probabilistic real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, p. 52, 2015.

[14] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, ser. RTSS '07. IEEE Computer Society, 2007, pp. 239–243.

[15] Z. Guo, L. Santinelli, and K. Yang, "Edf schedulability analysis on mixed-criticality systems with permitted failure probability," in *21th IEEE International Conference on Embedded and Real-Time Computing System and Applications*, 2015.

[16] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of probabilistic cache related pre-emption delays," *Proceedings of the 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[17] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. J. Cazorla, "Measurement-Based Probabilistic Timing Analysis for Multi-path Programs," in *23nd Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2012.

[18] L. Cucu-Grosjean, "Independence - a misunderstood property of and for (probabilistic) real-time systems," in *the 60th Anniversary of A. Burns, York*, 2013.

[19] L. Santinelli, P. Meumeu Yomsy, D. Maxim, and L. Cucu-Grosjean, "A component-based framework for modeling and analysing probabilistic real-time systems," in *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2011.

[20] E. Bini and G. C. Buttazzo, "The space of EDF feasible deadlines," in *19th Euromicro Conference on Real-Time Systems, ECRTS'07, 4-6 July 2007, Pisa, Italy, Proceedings*, 2007, pp. 19–28.

[21] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301–324, 1990.

Semi-partitioned Cyclic Executives for Mixed Criticality Systems

A. Burns
University of York, UK.
Email: alan.burns@york.ac.uk

S. Baruah
University of North Carolina, US.
Email: baruah@cs.unc.edu

***Abstract*—In a cyclic executive, a series of frames are executed in sequence; once the series is complete the sequence is repeated. Within each frame, units of computation are executed, again in sequence. In implementing cyclic executives upon multi-core platforms, there is advantage in coordinating the execution of the cores so that frames are released at the same time across all cores. For mixed criticality systems, the requirement for separation would additionally require that, at any time, code of the same criticality should be executing on all cores. In this paper we derive algorithms for constructing such multiprocessor cyclic executives for mixed-criticality collections of independent jobs.**

I. Introduction

Recent trends in embedded computing towards the widespread use of multi-core platforms, and the increasing tendency for applications to contain components of different criticality, have thrown up major challenges to the developers of safety-critical real-time systems. In this paper we consider these two challenges in the context of highly safety-critical application domains where cyclic executives remain the scheduling mechanism of choice.

Cyclic executives. A cyclic executive is a simple deterministic scheme that consists, for a single processor, of the continuous executing of a series of *frames* (or *minor cycles* as they are often called). Each frame consists of a sequence of *jobs* that execute in the specified sequence and are required to complete by the end of the frame. The set of frames is called the *major cycle*.

Multicore CPUs. On a multi-core, or multiprocessor, platform each core should have the same frame size and the same major cycle time. The time source from which the run-time support software will execute the jobs contained within each frame, is synchronised so that all cores switch between minor cycles concurrently. Within each frame there are a series of jobs to be executed. If jobs are constrained to execute always within the same minor cycle and always on the same core then the run-time schedule is defined to be *partitioned*. Alternatively, if jobs can migrate from one active frame to another active frame on a different core then the schedule is defined to be *global*. A *semi-partitioned* schedule has a small number of constrained migrations.

Mixed criticality. In mixed-criticality scheduling (MCS) theory, a single job may be characterized by several different WCET parameters denoting different estimates of the true WCET value, these different estimates being made at different levels of assurance. (The workload model used in this paper is formally defined in Section II.) The scheduling objective is then to validate the correct execution of each job at a level of assurance that is consistent with the criticality level assigned to that job: jobs assigned greater criticality must be shown to execute correctly when more conservative WCET estimates are assumed, while less critical jobs need to have their correctness demonstrated only when less conservative WCET estimates are assumed.

Related work. A cyclic executive is a particularly restricted form of static schedule. The issue of mapping mixed criticality code to static schedules has been addressed by Tamas-Selicean and Pop [10], [11]. An alternative approach to implementing the move between criticality levels in a static schedule is by switching between previously computed schedules; one per criticality level - this approach is explored in [2], [9]. However, these schemes are only applicable to single processor systems. The notion of separation used in this paper comes from [7].

In prior work [1], [5], we introduced the concept of implementing cyclic executives for mixed-criticality workloads upon multi-core CPUs. The workshop paper [1] formalized the problem, and proposed some initial approaches towards solving it for systems represented as collections of independent jobs. The scheduling test proposed was based upon a network flow argument and used a polynomial-time reduction. *In this paper we present a much more straightforward yet still optimal scheme.* See Def 1 for a definition of optimality.

II. System Model

The cyclic executive (CE) is defined by two durations, the length of the minor cycle (or frame) T_F and the duration of the major cycle T_M. These values are related by $(T_M = k.T_F)$ where k is a positive integer (usually a power of 2), denoting the number of frames in the repeating major cycle of the CE.

The issue of how to choose T_F and T_M to best support a set of tasks with given periods is beyond the scope of this paper. Rather we follow industrial practice [3] and assume these parameters are fixed by the system definition and that application tasks' periods are constrained to be multiples of T_F (up to the value of T_M).

The mapping of tasks to frames implies that there is a set of jobs allocated to each frame. All jobs within a frame must complete by the end of the frame. However, what it means to complete will depend on the behaviour of the system in terms of its criticality levels – as will be explained shortly.

We assume that the hardware platform consists of m identical (unit speed) processors (or cores). Each job can execute on any core and has identical temporal behaviour on all cores.

In general we assume there are V criticality levels, L_1 to L_V, with L_1 being the highest criticality. Each job j_i is assigned a criticality level, denoted χ_i, and two WCET parameters. One represents its estimated execution time at its own criticality level ($C_i(\chi_i)$) and the other an estimate at the base (i.e., lowest) criticality level ($C_i(L_V)$). It follows that if a job is of the lowest criticality level (i.e., $\chi_i = L_V$) then it only has one WCET parameter. For all other jobs, $C(\chi_i) \geq C(L_V)$. The rationale for having more than one WCET parameter is covered in a number of papers on mixed criticality systems, including the initial work of Vestal [12].

This use of only two C_i values for V criticality levels is a more constrained model than the one proposed by Vestal [12], under which each criticality level may give rise to a distinct WCET estimate. However with say five criticality levels it is unlikely that five distinct estimates of the worst-case execution time of the task would be available, while it can be argued ([6], [4]) that the restriction to just two estimates is sufficient to capture the key properties of a mixed criticality system.

At run-time the system is defined to be executing in one of V *modes*. In mode L_V (the lowest-criticality mode) all deadlines of all jobs must be met. It represents 'normal' behaviour. If every job j_i executes for no more than $C_i(L_V)$ then all deadlines must be guaranteed. If some job j_i executes for more than $C_i(L_V)$ then the mode of the system will degrade towards L_1, with jobs of criticality lower than χ_i no longer guaranteed. This mode change behaviour is explained in more detail later in the paper.

Run-time support

Mixed-criticality scheduling (MCS) theory has primarily concerned itself with the sharing of CPU computing capacity in order to satisfy the computational demand, as characterized by the worst-case execution times (WCET), of pieces of code. However, there are typically many additional resources that are also accessed in a shared manner upon a computing platform, and it is imperative that these resources also be considered. An interesting approach towards such a consideration was advocated by Giannopoulou et al. [7] in the context of multicore platforms: during any given instant in time, all the cores are only allowed to execute code of the same criticality level. This approach has the advantage of ensuring that accesses to all shared resources (memory buses, cache, etc.) during any time-instant are only from code of the same criticality level. We refer to such a scheme of switching between workloads of different criticality levels as *synchronised switching*. We focus our attention in this paper on synchronized switching. That is, we seek to construct cyclic executives in which each minor cycle may be considered partitioned into V criticality levels. Initially the highest criticality jobs are executed, when they have finished the next highest criticality jobs are executed, and so on. This continues until finally the lowest criticality jobs are executed. In a simple system with just two criticality levels, HI and LO, there is a switchover time S defined within each minor frame. Before S each core is executing HI-criticality work, after S each core is executing LO-criticality work. To give resilient fault tolerant behaviour, if the HI-criticality work has not completed by time-instant S on any core then the LO-criticality work is postponded (on every core), thereby giving extra time for the HI-criticality work to execute (up to the end of the minor cycle). In this paper we will explore how to find acceptable (safe and efficient) values for the switching times.

Implementing the criticality switches. Giannopoulou et al. [7] advocated, if supported by the hardware platform, the use of synchronisation barriers. In the case of dual-criticality workloads (the generalization to > 2 criticality levels is straight-forward), each core calls the barrier upon completing its assigned HI-criticality work. When the final core completes and calls the barrier, all the calls are released from the barrier and each core continues with executing LO-criticality work.

The benefit of this barrier-based scheme is that it can take advantage of time gained by jobs executing for less than their estimated WCETs. So at the end of the HI-criticality executions if the signal occurs before the pre-computed barrier S, then all cores can move to LO-criticality executions early. Additionally, there may be situations arising at run-time when a late switch to one criticality level is compensated by time gained from under-execution within jobs of the next criticality level. For example, the switch occurs at some time $> S$, but the LO-criticality jobs end up executing for less than their $C_i(\text{LO})$ WCET values and hence all complete by the end of the frame.

III. Dual Criticality Jobs

In this section, we consider the scheduling of a collection of jobs within a single frame of an m-processor platform, when there are only two criticality levels ($V \equiv 2$). All the jobs are assumed to become available at the start of the frame (without loss of generality, denoted as being at time 0), and they all have a deadline at the end of the frame (denoted D). In keeping with prior work on the scheduling of such dual-criticality systems, we use the notation HI and LO to denote the greater and lesser criticality levels (i.e., $L_1 \equiv \text{HI}$ and $L_V \equiv L_2 \equiv \text{LO}$). The criticality of job j_i is denoted by $\chi_i \in \{\text{LO}, \text{HI}\}$; each LO-criticality job j_i is characterized by a single WCET parameter $C_i(\text{LO})$, while each HI-criticality job is characterized by two WCET parameters $C_i(\text{LO})$ and $C_i(\text{HI})$.

Given a collection of such dual-criticality jobs to be scheduled within a frame of duration D upon an m-processor platform, our objective is to determine the switching point S such that only HI-criticality jobs are executed over the interval $[0, S)$. If all HI-criticality jobs complete by time-instant S, then LO-criticality jobs are executed over $[S, D)$; else, the LO-criticality jobs are abandoned and execution of HI-criticality jobs continues over $[S, D)$ as well. It follows that there are three conditions that need to be satisfied:

1) If each HI-criticality job j_i executes for no more than $C_i(\text{LO})$, then all the HI-criticality jobs must fit into the interval $[0, S)$.

2) All the LO-criticality jobs must fit into the interval $[S, D)$
3) If each HI-criticality job j_i executes for no more than $C_i(\text{HI})$, then all the HI-criticality jobs must fit into the interval $[0, D)$.

In Section III-A below, we derive a simple and efficient algorithm for determining S (and the corresponding schedules) such that these conditions are satisfied; in Section III-B, we describe an optimization to this simple method. These algorithms assume minimal run-time support; if additional run-time support is available, then a further optimization is possible – this is described in Section III-C.

A. A simple scheme for constructing CEs

We first define two (potential) candidates for the switching point S:

$S^{\min}$ The earliest instant at which all HI-criticality jobs have completed their LO-criticality work.

$S^{\max}$ The latest instant at which a switch must occur for the LO-criticality work to complete by time D.

It is evident that any candidate S must satisfy the two inequalities $S^{\min} \leq S \leq S^{\max}$.

Let us additionally define two interval durations, which constrain the possible values of $S^{\min}$ and $S^{\max}$.

Δ^{LO} The duration (makespan) of the interval needed for all the LO-criticality jobs to (begin and) complete execution.

Δ^{HI} The duration of the interval needed for all the HI-criticality jobs to execute the extra work they must do in HI-criticality mode — i.e., the amount $(C_i(\text{HI}) - C_i(\text{LO}))$, for each j_i with $\chi_i = \text{HI}$.

To determine these durations, we employ the optimal scheme of McNaughton [8, page 6]. Given a collection of n jobs with execution requirements $c_1, c_2, \ldots, c_n$, McNaughton showed that the minimum makespan of a preemptive schedule for these jobs on m unit-speed processors is given by

$$\max\left(\frac{\sum_{i=1}^{n} c_i}{m}, \max_{i=1}^{n}\{c_i\}\right) \tag{1}$$

The actual schedule is obtained by taking the jobs (in any order) and allocating them to m intervals of the size of the makespan, each representing one of the m processors. As one interval is filled, perhaps with part of a job, the next interval starts with the rest of this job. At most $(m-1)$ jobs are split across intervals in this manner. During run-time a job that was split across two intervals will run at the beginning of the time-interval upon one processor, and towards the end of the time-interval on the other processor.

A direct application of McNaughton's result yields the conclusion that the minimum makespan for a global preemptive schedule for the jobs in LO-criticality mode is given by

$$\Delta^{\text{LO}} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{LO}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{LO}}\left\{C_i(\text{LO})\right\}\right) \tag{2}$$

We therefore set

$$S^{\max} \stackrel{\text{def}}{=} D - \Delta^{\text{LO}} \tag{3}$$

Similarly, a direct application of the makespan result allows the minimum interval for the HI-criticality work (in LO-criticality mode) to be computed:

$$S^{\min} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{HI}}\left\{C_i(\text{LO})\right\}\right) \tag{4}$$

Clearly for the whole system to be schedulable, it is necessary that $S^{\min} \leq S^{\max}$ which is equivalent to requiring that

$$\begin{aligned} & S^{\min} \leq D - \Delta^{\text{LO}} \\ \Leftrightarrow \quad & S^{\min} + \Delta^{\text{LO}} \leq D \end{aligned} \tag{5}$$

We now consider the final constraint — the scheduling of HI-criticality jobs executing in HI-criticality mode. It has been shown [1, Example 1] that this is not necessarily ensured by simply computing the makespan (using McNaughton's method, as above) with the $C_i(\text{HI})$ values, and validating that the resulting makespan is $\leq D$. We instead determine the minimal makespan for all the HI-criticality jobs, subject to each such job having received an amount of execution equal to its LO-criticality WCET by time-instant $S^{\min}$. To determine this makespan, we apply McNaughton's scheme to the work that is left to do after time-instant $S^{\min}$ (i.e. $C_i(\text{HI}) - C_i(\text{LO})$ for each job j_i with $\chi_i = \text{HI}$). Letting $C_i(\text{EX})$ denote the "excess" computational requirement of job j_i in HI-criticality mode over LO-criticality mode:

$$C_i(\text{EX}) \stackrel{\text{def}}{=} \big(C_i(\text{HI}) - C_i(\text{LO})\big),$$

we have

$$\Delta^{\text{HI}} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{EX})}{m}, \max_{\chi_i=\text{HI}}\left\{C_i(\text{EX})\right\}\right) \tag{6}$$

It is evident that $S^{\min} + \Delta^{\text{HI}} \leq D$ is sufficient for schedulability; earlier (Expression 5) we had shown that $S^{\min} + \Delta^{\text{LO}}$ should also be $\leq D$. Putting these pieces together, we may summarize this method as follows. We compute $S^{\min}, \Delta^{\text{LO}}$, and Δ^{HI} according to Expressions (4), (2), and (6) respectively, and require that

$$S^{\min} + \max\big(\Delta^{\text{LO}}, \Delta^{\text{HI}}\big) \leq D \tag{7}$$

as a sufficient schedulability condition. If this condition is satisfied, $S \leftarrow S^{\min}$ (i.e., we declare $S^{\min}$ to be the switch-point we had set out to compute).

B. An improvement

Let us now suppose that Condition 7 is violated, and $S^{\min} + \max\big(\Delta^{\text{LO}}, \Delta^{\text{HI}}\big) > D$. Since $\big(S^{\min} + \Delta^{\text{LO}} \leq D\big)$ is a necessary condition for schedulability (see Inequality 5), it must be the case that

$$S^{\min} + \Delta^{\text{HI}} > D.$$

Now if $\big(\sum_{\chi_i=\text{HI}} C_i(\text{HI}) \geq mD\big)$, there is nothing to be done. Otherwise, there must be some unused processor capacity in the McNaughton schedule constructed according to Expression 4 for the interval $[0, S)$, and/or in the McNaughton

schedule constructed according to Expression 6 for the interval after time-instant S. Let us consider the situation where the schedule has some unused processor capacity over the interval $[0, S)$ (recall that $S \leftarrow S^{\min}$ in the method of Section III-A). An inspection of Expression (4) reveals that this happens if

$$\frac{\sum_{\chi_i=\text{HI}} C_i(\text{LO})}{m} < \max_{\chi_i=\text{HI}} \{C_i(\text{LO})\}$$

Our idea, intuitively speaking, is that any such unused capacity prior to time-instant S may as well be allocated to some HI-criticality task, for use in the event of the system undergoing a mode-change into HI-criticality mode. (If the system does not undergo such a mode-change, this allocated capacity may end up remaining unused.) Doing so leaves less execution remaining to be completed after the switch instant S in HI-criticality mode, and may thus result in a smaller makespan in HI-criticality modes (i.e., a smaller value for Δ^{HI}).

Such a scheme is particularly effective if the duration of the HI-criticality schedule after S — the one of duration Δ^{HI} — is also dominated by longer jobs, i.e., if in Expression 6

$$\frac{\sum_{\chi_i=\text{HI}} C_i(\text{EX})}{m} < \max_{\chi_i=\text{HI}} \{C_i(\text{EX})\}$$

If this be the case, then the unused capacity prior to time-instant S can be filled so as to minimise the maximum $C_i(\text{EX})$ by bringing forward work to before S — this is accomplished by increasing $C_i(\text{LO})$ for such a job and decreasing its $C_i(\text{EX})$ by the same amount. However, jobs that have $(C_i(\text{LO}) = S)$ cannot have work brought forward in this manner since this would result in S increasing as well.

It is evident that this scheme is effective since:

- Any work brought forward will not change S,
- The first term in Expression (6) is not increased by bringing work forward, and
- The second term in Expression (6) is reduced by always choosing the largest value and decreasing it.

We note that if more than one job has the same $C_i(\text{EX})$ value then an arbitrary choice is made (and has no impact on optimality).

And what if there is no unused processor capacity in the schedule over $[0, S)$? In that case, the switch-point S may be increased to any value $\leq S^{\max}$ (where $S^{\max}$ is as defined by Expression (3)). An obvious choice for S is $S \leftarrow S^{\max}$; an algorithm for achieving the smallest value of S (i.e., the earliest possible switch-time) is as follows. Setting the switch point S to be $S^{\min} + 1$ will generate m free slots. So $C_i(\text{LO})$ values of HI-criticality jobs can be increased by this amount (and the corresponding $C(\text{EX})$ values decreased). If this will reduce the size of Δ^{HI} by more than one then an overall decrease in $S + \Delta^{\text{HI}}$ will have been achieved . This cycle is repeated (i.e. adding 1 to S) until either no further gain is made or S takes the value of $S^{\max}$. At each step of the cycle no $C(\text{LO})$ value should increase beyond the current value of S.

	χ_i	$C_i(\text{LO})$	$C_i(\text{HI})$	$C_i(\text{HI}) - C_i(\text{LO})$
j_1	LO	3	-	-
j_2	LO	2	-	-
j_3	LO	2	-	-
j_4	HI	2	7	5
j_5	HI	3	7	4
j_6	HI	3	3	0
j_7	HI	4	4	0

TABLE I
AN EXAMPLE DUAL-CRITICALITY JOB INSTANCE

Example 1: To illustrate the above scheme consider the scheduling of the mixed-criticality instance of Table I upon 3 unit-speed processors with a frame length of 8 ($D = 8$).

We can immediately use the equations above to compute: $\Delta^{\text{LO}} = 3$ (and hence $S^{\max} = 5$) and $S^{\min} = 4$. So the first step to schedulability is satisfied (i.e. $S^{\min} \leq S^{\max}$). We note that if we ignore mixed criticality issues then the minimum makepsan for the HI-criticality jobs (ignoring LO-criticality work) is 7. So a completely separated scheme would require a frame size of 10 $(7 + 3)$.

If we initially focus on $S^{\min}$ then we note that there are no free slots, so equation(6) gives a makespan in HI-criticality mode (Δ^{HI}) of 5. So the use of this value for S (i.e. 4) gives a required frame size of 9 (4+5); since the frame-size is 8, the instance would be deemed unschedulable with $S \leftarrow 4$.

However, if we set $S \leftarrow (S^{\min}+1)$ which equals $S^{\max} = 5$ then the total work available on three processors by time 5 is 15. The work required using $C(\text{LO})$ values for HI-criticality work is 12. Hence 3 units of work can be added to these $C(\text{LO})$ values. If we make $C_4(\text{LO}) = 4$ and $C_5(\text{LO}) = 4$ then maximum $C_i(\text{EX})$ becomes equal to 3. Hence $\Delta^{\text{HI}} = 3$ and $S^{\max} + \Delta^{\text{HI}} = 8$. Therefore the job set fits into the frame size of 8, with a switch time of 5. ■

C. More flexible implementations

The cyclic executives constructed as discussed above are implementable as lookup tables. Three lookup tables are constructed as dictated by the McNaughton procedure: one for the interval $[0, S)$, another for HI-criticality jobs over the interval $[S, D)$, and a third for LO-criticality jobs over the interval $[S, D)$. The first lookup table is always executed, while one of the other two is selected depending upon whether all HI-criticality jobs have completed or not by time-instant S.

Lookup tables are a very restrictive form of run-time dispatching. If a certain amount of additional *flexibility* is permitted, then more efficient use of platform resources may be possible, We illustrate with an example.

Suppose that $C_1(\text{LO})$ in the example instance of Table I were equal to 4 (rather than 3 as listed in Table I). It may be verified that Δ^{LO} for this instance is then equal to 4; the switch-point must therefore be $\leq (8 - 4)$ or 4. But we saw in Example 1 that this is not possible, since setting $S \leftarrow 4$ results in a makespan of $(4 + 5 =)$ 9 in HI-criticality mode.

Let us therefore choose $S \leftarrow 5$ as mandated by the arguments in Example 1, and consider the CE schedule specified in Figure 1 over the interval $[0, 5)$. Notice that this schedule is

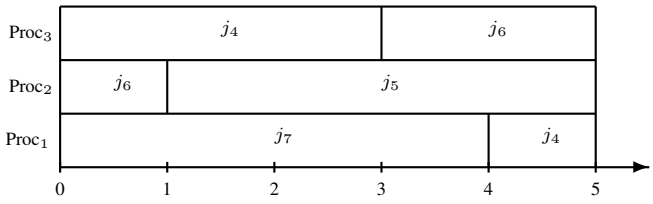


Fig. 1. Dynamic switching.

compliant with the requirements of Example 1: j_4 and j_5 both execute for 4 units over $[0, 5)$ while j_6 and j_7 each execute for their $C_i(\text{LO})$ values of 3 and 4 respectively. Hence in HI-criticality mode all HI-criticality jobs would complete by the end of the frame, at time-instant 8.

Now observe that in any LO-criticality behaviour,

- j_4 would complete $C_4(\text{LO}) = 2$ units of execution by time-instant 2; hence, the execution of j_6 on processor 3 could be moved forward[1] to the interval $[2, 4)$.
- As a consequence, j_6 would complete its $C_6(\text{LO}) = 3$ units of execution by time-instant 4.
- j_5 would complete its $C_5(\text{LO}) = 3$ units of execution over $[1, 4)$, also completing by time-instant 4, and
- j_7 would execute to completion over $[0, 4)$.

Thus, all the HI-criticality jobs processors will have completed their LO-criticality execution by time-instant 4, and the platform becomes available for the LO-criticality jobs to execute at time-instant 4 and complete by time-instant 8.

This example illustrates that the added run-time flexibility of adjusting the pre-computed schedule may permit enhanced schedulability — instances not schedulable without this flexibility can be scheduled correctly. We are currently working on better understanding what kinds of run-time flexibility are reasonable to permit within the context of cyclic executives; we leave as future work the design of algorithms that would construct schedules such as the one shown in Figure 1.

IV. Jobs with More than Two Criticality Levels

We now assume there are $V > 2$ criticality levels, L_1 (the highest) to L_V (the lowest). Recall from Section II that each job j_i, of criticality χ_i, has just has two WCET estimates, one for the base criticality level L_V, $C_i(\text{normal}) = C_i(L_V)$, abbreviated to $C_i(\text{NL})$, and one for its own criticality level, $C_i(\text{self}) = C_i(\chi_i)$, abbreviated to $C_i(\text{SF})$. We define $C_i(\text{EX}) \stackrel{\text{def}}{=} C_i(\text{SF}) - C_i(\text{NL})$. We overload the symbols L_i to also denote the set of jobs of that criticality. We seek to compute $(V-1)$ switch points S^1 to S^{V-1} that are constrained as follows (for notational convenience we let S^0 and S^V denote the start and end of the frame respectively (i.e $S^0 \equiv 0$ and $S^V \equiv D$)). So for each criticality level L_i, and frame f we require that

- If each job $j_i \in L_i$ executes for no more than $C_i(\text{NL})$, then all the jobs in the set L_i must fit into the interval $(S^{i-1}, S^i]$
- If each job $j_i \in L_i$ executes for no more than $C_i(\text{SF})$, then all the jobs in the set L_i must fit into the interval $(S^{i-1}, S^V]$.

To compute the switching times we extend the process defined in Section III above to > 2 criticality levels. It is possible to start at the lowest or highest criticality level; experimentation shows that it is better to start at the highest. So first we compute minimum makespan for criticality level L_1:

$$S_1^{\min} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{j_i \in L_1} C_i(\text{NL})}{m}, \max_{j_i \in L_1}\left\{C(\text{NL})\right\}\right) \tag{8}$$

Next we compute Δ^1 and check that $S_1^{\min} + \Delta^1$ is no greater than S^V $(= D)$:

$$\Delta^1 \stackrel{\text{def}}{=} \max\left(\frac{\sum_{j_i \in L_1} C_i(\text{EX})}{m}, \max_{j_i \in L_1}\left\{C_i(\text{EX})\right\}\right) \tag{9}$$

If $S_1^{\min} + \Delta^1 > S^V$ then work must be brought forward so that $S_1^{\min}$ is increased but Δ^1 is decreased by a greater amount. This is achieved by adding to $C(\text{NL})$ so as to minimise the maximum $C(\text{EX})$ (for jobs of criticality L_1). If such alterations cannot deliver $S_1^{\min} + \Delta^1 \leq S^V$ then the job set is unschedulable. Alternatively, S^1 is fixed to be the minimum value computed.

This process is repeated for each criticality level, L_i using:

$$S_i^{\min} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{j_i \in L_i} C_i(\text{NL})}{m}, \max_{j_i \in L_i}\left\{C_i(\text{NL})\right\}\right) \tag{10}$$

and

$$\Delta^i \stackrel{\text{def}}{=} \max\left(\frac{\sum_{j_i \in L_i} C_i(\text{EX})}{m}, \max_{j_i \in L_i}\left\{C_i(\text{EX})\right\}\right) \tag{11}$$

with the conditions

$$S^{i-1} + S_i^{\min} + \Delta^i \leq S^V \tag{12}$$

and for all jobs of criticality L_i

$$C_i(\text{NL}) \leq S_i^{\min}. \tag{13}$$

At all stages, modification to $C_i(\text{NL})$ (and hence $C_i(\text{EX})$) are made to ensure these two conditions are met. Note that some movement of computation time may be possible without increasing a $S_i^{\min}$ value. Each step fixed S^i.

A. An Example

We illustrate the above scheme upon an example with two cores, four criticality levels and three jobs per criticality level. Table II lists the parameters for the jobs. The frame length is 20 units. We note that independent makespans for the four criticality levels would require a frame length of 48 (20+15.5+8.5+4).

First $S_1^{\min}$ and Δ^1 are computed; they are seen to equal 3 and 18 respectively. Together this is too large (as the frame size is 20). So S^1 is set to 4 (i.e. $C_1(\text{NL}) = 4$ with $C_1(\text{EX}) = 16$).

[1]Note that this moving forward of j_6's execution is not permitted in a pure lookup table dispatcher; this is the additional implementation flexibility that is sought in this section.

	χ_i	$C_i(\text{NL})$	$C(\text{SF})$	$C_i(\text{EX})$
j_1	L_1	2	20	18
j_2	L_1	1	8	7
j_3	L_1	3	9	6
j_4	L_2	6	13	7
j_5	L_2	1	3	2
j_6	L_2	4	15	11
j_7	L_3	5	6	1
j_8	L_3	3	8	5
j_9	L_3	1	3	2
j_{10}	L_4	3	3	0
j_{11}	L_4	4	4	0
j_{12}	L_4	1	1	0

TABLE II
AN EXAMPLE MIXED-CRITICALITY JOB SET.

	χ_i	$C_i(\text{NL})$	$C(\text{SF})$	$C_i(\text{EX})$
j_1	L_1	4	20	16
j_2	L_1	1	8	7
j_3	L_1	3	9	6
j_4	L_2	6	13	7
j_5	L_2	1	3	2
j_6	L_2	7	15	8
j_7	L_3	5	6	1
j_8	L_3	4	8	4
j_9	L_3	1	3	2
j_{10}	L_4	3	3	0
j_{11}	L_4	4	4	0
j_{12}	L_4	1	1	0

TABLE III
THE EXAMPLE JOB SET OF TABLE II TRANSFORMED.

This now delivers $S_1^{\min} = 4$ and $\Delta^1 = 16$ which is sufficient for criticality level L_1.

Next level L_2 is checked. Note the frame size for this criticality level is, in effect, 16 (i.e. 20 - S^1). So, $S_2^{\min} = 6$ and $\Delta^2 = 11$; again this is too long so $S_2^{\min}$ is set to 7, with the result that $C_6(\text{NL})$ is made equal to 7 and $C_6(\text{EX})$ is 8. As a result Δ^2 is now equal to 8.5, and the sum of the two intervals is 15.5 which is sufficient. This fixes S^2 to be 11 (4+7).

Continuing with L_3. Frame size is now 9. Value of $S_3^{\min}$ is 5, and Δ^3 is 5 also. As $10 > 9$ there is again a need to reduce Δ^3. Here this can be done without increasing $S_3^{\min}$ (as this interval was not 'full'). Let $C_8(\text{NL}) = 4$ and hence $C_8(\text{EX}) = 4$. Now $\Delta^3 = 4$ and $S_3^{\min} + \Delta^3 = 9$ (5+4). Again this is sufficient and S^3 is set equal to 16.

The final step is to check that the lowest criticality jobs will fit into the interval left for them. The interval is of length 4, and the makespan (Δ^4) for this set is 4. So they are accommodated and the job set can be declared schedulable.

To further illustrate the process of modifying the job set to obtain a schedulable one, Table III gives the parameters of the job set obtained after modification. It is easy to observe that the new job set is obtained from the initial one by just adding to the $C(\text{NL})$ estimates. And also it is clear that the new job set is schedulable with switch points 4, 11 and 16.

Optimality. The scheme outlined at the beginning of this section via equations/conditions (10) to (13), and illustrated with the example above, is optimal in the following sense.

Definition 1: An allocation scheme (of jobs to frames) is *optimal* if it leads to the smallest possible switching points and a schedulable system.

This notion of optimal is intuitive as for each criticality level the earliest switching point maximises the time available for the lower criticality levels. The scheme produces the optimal value for each switching point, S_i, as:

- If $S_i = S_i^{\min}$ satisfies condition (12) then this is the minimum makespan by definition [8, page 6].
- If condition (12) is not satisfied the scheme increases $C(\text{NL})$ values by the minimum amount commensurate with decreasing the maximum $C(\text{SF})$ so that the condition is met. This leads to a minimum increase in S_i.
- If no S_i can be found (i.e. it continues to increase until $\Delta^i = 0$ without satisfying condition (12) then the system is unschedulable.

V. CONCLUSIONS

Single processor safety-critical systems are often constrained so that they can be implemented as a series of frames in a repeating cyclic executive. In this paper we have extended this approach to incorporate multi-core platforms and mixed criticality applications. We allow a minimum number of jobs to be split across the frames, and propose a practical means of constructing the necessary cyclic schedule. Future work will extend our approach to multi-cycle systems.

Acknowledgements. This research is partially supported by NSF grants CNS 1115284, CNS 1218693, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors Corp. It is also supported by ESPRC grant MCC (EP/K011626/1). No new primary data were created during this study.

REFERENCES

[1] S. Baruah and A. Burns. Achieving temporal isolation in multiprocessor mixed-criticality systems. In *Proc. of the 2nd Workshop on Mixed Criticality Systems (WMC)*, 2014.

[2] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proc. of the Real-Time Systems Symposium (RTSS)*, 2011.

[3] I. Bate and A. Burns. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Systems*, 25(1):5–37, 2003.

[4] A. Burns. An augmented model for mixed criticality. In Sanjoy K. Baruah, Liliana Cucu-Grosjean, Robert I. Davis, and Claire Maiza, editors, *Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121)*, volume 5(3). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2015.

[5] A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *Proc. ECRTS*, 2015.

[6] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proc. of the Real-Time Systems Symposium*, pages 291–300, 2009.

[7] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *International Conference on Embedded Software (EMSOFT)*, pages 17:1–17:15, 2013.

[8] R McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[9] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed critical scheduler. In *Proc. of the Workshop on Mixed Criticality Systems (WMC)*, pages 67–72, 2013.

[10] D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Proc. of the Real-Time Systems Symposium (RTSS)*, 2011.

[11] D. Tamas-Selicean and P. Pop. Task mapping and partition allocation for mixed-criticality real-time systems. In *Dependable Computing (PRDC), 17th Pacific Rim International Symposium on*, pages 282–283, 2011.

[12] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of the Real-Time Systems Symposium*, pages 239–243, 2007.

Investigating Mixed Criticality Cyclic Executive Schedule Generation

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 require a difficult compromise to be drawn between efficient system utilisation and sufficient separation of critical components. In addition to these challenges, hardware platforms are becoming increasingly multi-core in nature bringing up additional scheduling issues. Previous publications have met these challenges by suggesting a Cyclic Executive based approach for Mixed Criticality scheduling. They make use of a barrier protocol to separate the execution within each minor cycle, executing higher critical work, then less critical work. The barrier protocol allowed such a separation of criticalities to remain consistent across all cores in a given platform. This strict separation has the advantage that higher criticality work cannot suffer interference from lower, including communication and recourse access. One of the key challenges of using a Cyclic Executive is the construction of a valid schedule. In this work we consider the question, "Is it worth using an optimal solver such as Integer Linear Programming (ILP) for Cyclic Executive schedule generation?". We start by extending the Cyclic Executive model to include multiple minor cycles. An ILP model is described and evaluated against the heuristic worst fit. The results show that ILP significantly outperforms worst fit. Finally we show that ILP is not only effective, but also efficient in terms of runtime and scalability for the examples and parameters considered in this work, making it a practical choice for Cyclic Executive schedule generation of real systems.**

I. Introduction

With the introduction of powerful multi-core architectures comes the desire to consolidate functionality, that was previously spread across many nodes, onto a single common hardware platform. Inevitably such a consolidation gives rise to the situation where more critical work must be placed upon the same resources as less critical work. This consolidation has brought about the notion of a Mixed Criticality (MC) system. Scheduling such a system is challenging as highly critical work must often be certified and follow safety standards such as Design Assurance Levels (DAL) in the aerospace industry and ASIL (Automotive Safety Integrity Levels) in the automotive industry. Care must be taken to ensure that less critical work can not interfere with the execution of higher critical work. The design of such systems becomes a trade-off between efficiently utilising the system resources while providing an adequate level of separation to satisfy any safety requirements.

One of the most widely used scheduling policies in industry is the Cyclic Executive (CE). Such policies execute code cyclically in a pre-defined order, as such they are highly deterministic which makes them a favourable choice for highly critical applications with stringent certification requirements. Cyclic Executive systems are made up of a major cycle which is composed on a number of minor cycles, the major cycle repeats in a cyclic manner. Naturally this determinism comes with some drawbacks, Baker and Shaw [1] performed some initial evaluations on the CE model, they noted some restrictions:

- Cyclic Executives can only easily support periodic work.
- Tasks must have periods that are multiples of the minor cycle.
- Tasks must have deadlines equal to or less than the minor cycle.
- Tasks cannot have a period greater than the major cycle.

In addition to these drawbacks, the creation of CE schedules is well known to be NP-hard. Despite these issues the high level of determinism makes Cyclic Executives popular schedulers.

Baruah and Burns [3] investigate the notion of a mixed criticality cyclic executive. In order to provide the separation required between different levels of criticality they use the scheme proposed by Ginnopoulou et al. [6]. This approach uses a barrier protocol to completely separate the execution of different criticality levels. The barrier mechanism works by having each CPU call it when its execution for a particular criticality level has completed. Once all CPUs have called the barrier, they are released and allowed to execute the work for the next criticality level. The barrier protocol requires minimal hardware or OS support. Baruah and Burns make use of this protocol within a CE context. Within each minor cycle, work is executed in order of criticality, highest criticality first, each level is separated by a barrier.

Burns et al. [5] build upon the work in [3] by considering the creation of CE schedules using heuristics to allocate tasks to cores. This work considers the simple case where a system is made up of a single minor cycle (i.e. minor cycle = major cycle), they assess the performance of First Fit (FF), Worst Fit (WF) and First Fit with Branch and Bound (FFBB). They show that the barrier protocol does impact the ability to create CE schedules but conclude this it is a necessary compromise to allow for more robust systems that are easier to certify.

In this work we seek to extend the investigation into cyclic executive schedule construction. We will extend our view of the system model to include multiple minor cycles within a major cycle. Alongside heuristic based techniques there are optimal solutions which typically come with increased execution overheads. As we know that the MC scheduling problem is NP-hard in the strong sense [2] we extend our work to consider an optimal solver. One such technique is Integer Linear Programming (ILP), in this work we make use of ILP and show that not only does it allow a large number of task sets to be scheduled, even in the extended system model but for the purposes of CE schedule constriction it is efficient.

Throughout this work we will make use of the MC system model proposed by Vestal in 2007 [8]. Vestals model proposes that each task in a system has a WCET value for its own criticality level and all those below. Our model is made up of a number of dual criticality periodic tasks with the properties $\tau = \{C(LO), C(HI), T, D, L\}$ where $C(LO)$ is the LO criticality WCET, $C(HI)$ is the HI criticality WCET (and $C(HI) > C(LO)$), T is the period, D is the deadline and L is the criticality level (HI or LO). We use T^F to denote the minor cycle and T^M to denote the major cycle.

In addition to the model above, we assume a constrained system where the major cycle is a multiple of the minor cycle. As such task periods must also be multiples of the minor cycle and no greater than the major cycle. In general in this work we consider 4 minor cycles per major cycle, where $T^F = 25$ and $T^M = 100$. We do not consider the issue of assigning an arbitrary set of periods to a cyclic executive.

The remainder of this work is structured as follows, Section II will detail the construction of an ILP model used to check for feasible CE schedules and show its effectiveness against heuristic based techniques, Section III will consider whether ILP can be practically used by assessing its computational overheads and Section IV will present some conclusions to this work.

II. Using ILP to create CE schedules

In this section we will describe how the constraints of a cyclic executive can be expressed as an ILP model and how this model is used to check for a feasible schedule. We extend the work of [3] by allowing for multiple minor cycles within the major cycle, this leads to an allocation problem of tasks to frames and the cores within each frame. We will briefly recap the runtime of a mixed criticality CE using the barrier protocol and its schedulability test, following this the ILP implementation will be described by means of an example.

The runtime of a single minor cycle in a (dual criticality) mixed criticality cyclic executive is as follows:

CE Runtime:

- The minor cycle begins by executing HI criticality tasks on all cores.
- Once tasks on a core have finished executing HI work they signal the barrier protocol.
- If all cores signal the barrier before their $C(LO)$ execution times, LO criticality work may commence.
- If any core does not signal completion by their $C(LO)$ then the system moves into the HI criticality mode and all HI tasks are allowed to execute up to their $C(HI)$ execution times.

The latest time at which each core could call the barrier protocol to report HI work complete is denoted by $S(i,j)$, where i is the core and j is the minor cycle. The point at which the system changes from executing the HI work in the LO mode to executing the LO work is denoted by $S^{max}(j)$, where j is the minor cycle. As such the schedulability of a mixed criticality CE can be determined as follows (where $HI(i,j)$ is the set of high criticality tasks scheduled on core i, minor cycle j and $LO(i,j)$ is the set of LO criticality tasks scheduled on core i, minor cycle j):

1) HI criticality tasks must fit within the minor cycle:
$$\forall i \text{ and } j, \sum_{k \in HI(i,j)} C_k(HI) \leq T^F.$$
2) The value of $S(i,j)$ can be calculated for each core:
$$S(i,j) = \sum_{k \in HI(i,j)} C_k(LO)$$
3) The value of $S^{max}(j)$ used across all CPUs is:
$$S^{max}(j) = \max(S(i,j))$$
4) LO criticality jobs must fit within the time between $S^{max}(i,j)$ and the end of the minor cycle:
$$\forall i \text{ and } j, \sum_{k \in LO(i,j)} C_k(LO) \leq T^F - S^{max}(j)$$

We will describe the construction of our ILP model via the use of an example. Consider the task set shown in Table I:

τ	$C_i(LO)$	$C_i(HI)$	T_i	D_i	L_i
$T1$	3	4	25	25	HI
$T2$	4	5	50	50	HI
$T3$	5	6	50	50	HI
$T4$	13	15	25	25	HI
$T5$	10	-	25	25	LO
$T6$	2	-	50	50	LO
$T7$	3	-	25	25	LO
$T8$	5	-	100	100	LO

TABLE I
A Mixed Criticality task set with 4 minor cycles
($T^F = 25, T^M = 100$).

The construction of the ILP model to check the schedulability of the task set in Table I will now be described based on a 2 core platform. We will describe this model based on the syntax of the Gurobi optimiser [7] which is the tool used throughout this work.

In order to achieve our goal of testing for any valid CE schedule we require a simplistic method of modelling our system. In order to model the possible locations of a task

we create a variable for each location, the variables are in the format $T[tasknumber]\_[core][cycle]$. Each variable is declared as a binary value, therefore if it is set to 1 the task is scheduled in that location. For example $T1$ has a period of 25, as such is included in all 4 of the minor cycles, therefore it must be scheduled on one of the two cores within each minor cycle, (T_11 or T_21 where $T^F = 1$). In the next part of the model we define the bounds for these variables so that each task might only be scheduled the correct number of times in the correct places (no duplicate tasks etc.).

With this in mind we construct our maximize statement, the first section of the ILP model. As we are not interested in optimising any particular parameter, we need not include anything in this section. We merely seek to discover if a scheduleable assignment exists, we require at least one feasible schedule.

The second stage of the model is the *Subject To* section, in this section the key constraints of the model are defined. To model this we set-up a constraint for each minor cycle, for cycle one $T1$ would have the constraint $T1\_11 + T1\_21 = 1$, this ensures that $T1$ is only scheduled on one of the two cores in the first minor cycle. These constraints are repeated for the remaining minor cycles.

If a task has a period of 50, it must be scheduled once in the first two minor cycles and once in the second two. We use the same notation to model this, for example cycles one and two for $T2$ can be constrained with the following: $T2\_11 + T2\_21 + T2\_12 + T2\_22 = 1$.

Finally if a task has a period of 100 then it must be scheduled only once within the major cycle, we model this for $T8$ as follows: T8_11 + T8_21 + T8_12 + T8_22 + T8_13 + T8_23 + T8_14 + T8_24 = 1.

The complete set of constraints for the task set shown in Table I are shown below:

```
Subject To
T1_11 + T1_21 = 1
T1_12 + T1_22 = 1
T1_13 + T1_23 = 1
T1_14 + T1_24 = 1
T2_11 + T2_21 + T2_12 + T2_22 = 1
T2_13 + T2_23 + T2_14 + T2_24 = 1
T3_11 + T3_21 + T3_12 + T3_22 = 1
T3_13 + T3_23 + T3_14 + T3_24 = 1
T4_11 + T4_21 = 1
T4_12 + T4_22 = 1
T4_13 + T4_23 = 1
T4_14 + T4_24 = 1
T5_11 + T5_21 = 1
T5_12 + T5_22 = 1
T5_13 + T5_23 = 1
T5_14 + T5_24 = 1
T6_11 + T6_21 + T6_12 + T6_22 = 1
T6_13 + T6_23 + T6_14 + T6_24 = 1
T7_11 + T7_21 = 1
T7_12 + T7_22 = 1
T7_13 + T7_23 = 1
T7_14 + T7_24 = 1
T8_11 + T8_21 + T8_12 + T8_22 +
T8_13 + T8_23 + T8_14 + T8_24 = 1
```

Statements are now required to ensure that the taskset is schedulable in the configuration chosen. This is done in three stages.

Stage One: The first stage aims to ensure that the HI criticality work is schedulable when it executes up to its maximum $C(HI)$ WCET value. This is done by multiplying each tasks WCET with the variables representing the possible locations of the tasks. If the variable is set to 1 and the task is scheduled, then the answer will be equal to the WCET, if 0 then the answer is 0. The notation for the HI criticality tasks in the HI mode is shown below:

```
4 T1_11 + 5 T2_11 + 6 T3_11 + 15 T4_11 <= 25
4 T1_21 + 5 T2_21 + 6 T3_21 + 15 T4_21 <= 25
4 T1_12 + 5 T2_12 + 6 T3_12 + 15 T4_12 <= 25
4 T1_22 + 5 T2_22 + 6 T3_22 + 15 T4_22 <= 25
4 T1_13 + 5 T2_13 + 6 T3_13 + 15 T4_13 <= 25
4 T1_23 + 5 T2_23 + 6 T3_23 + 15 T4_23 <= 25
4 T1_14 + 5 T2_14 + 6 T3_14 + 15 T4_14 <= 25
4 T1_24 + 5 T2_24 + 6 T3_24 + 15 T4_24 <= 25
```

Stage Two: The second stage checks the schedulability of the HI criticality tasks executing to their LO WCET values, this is done in the same way as stage one. In addition to this an X value is added to the calculation, one X value per minor cycle (X_1, X_2, X_3, X_4). The X value represents the time between the point at which all cores complete their execution of the HI criticality tasks (S^{max}), and the end of the minor cycle (T^F).

```
3 T1_11 + 4 T2_11 + 5 T3_11 + 13 T4_11 +X_1<= 25
3 T1_21 + 4 T2_21 + 5 T3_21 + 13 T4_21 +X_1<= 25
3 T1_12 + 4 T2_12 + 5 T3_12 + 13 T4_12 +X_2<= 25
3 T1_22 + 4 T2_22 + 5 T3_22 + 13 T4_22 +X_2<= 25
3 T1_13 + 4 T2_13 + 5 T3_13 + 13 T4_13 +X_3<= 25
3 T1_23 + 4 T2_23 + 5 T3_23 + 13 T4_23 +X_3<= 25
3 T1_14 + 4 T2_14 + 5 T3_14 + 13 T4_14 +X_4<= 25
3 T1_24 + 4 T2_24 + 5 T3_24 + 13 T4_24 +X_4<= 25
```

Stage Three: The final stage seeks to ensure than the LO criticality tasks are schedulable within the time X we calculated above. This is achieved by a similar process to stages one and two, but this time also subtracting X. The solution must be less than or equal to 0 for the LO criticality execution to be schedulable within X.

```
10 T5_11 + 2 T6_11 + 3 T7_11 + 5 T8_11 −X_1<= 0
10 T5_21 + 2 T6_21 + 3 T7_21 + 5 T8_21 −X_1<= 0
10 T5_12 + 2 T6_12 + 3 T7_12 + 5 T8_12 −X_2<= 0
10 T5_22 + 2 T6_22 + 3 T7_22 + 5 T8_22 −X_2<= 0
10 T5_13 + 2 T6_13 + 3 T7_13 + 5 T8_13 −X_3<= 0
10 T5_23 + 2 T6_23 + 3 T7_23 + 5 T8_23 −X_3<= 0
10 T5_14 + 2 T6_14 + 3 T7_14 + 5 T8_14 −X_4<= 0
10 T5_24 + 2 T6_24 + 3 T7_24 + 5 T8_24 −X_4<= 0
```

The model then declares any bounds required, as all but 4 of the variables used are declared as binaries only 4 bounds are defined. The X values are bounded to be less than or equal to 25, in reality these variables should never reach this point.

```
Bounds
X_1 <= 25
X_2 <= 25
X_3 <= 25
X_4 <= 25
```

Finally we declare all variables used.

```
Binaries
T1_11  T1_21  T1_12  T1_22  T1_13  T1_23  T1_14  T1_24
T2_11  T2_21  T2_12  T2_22  T2_13  T2_23  T2_14  T2_24
T3_11  T3_21  T3_12  T3_22  T3_13  T3_23  T3_14  T3_24
T4_11  T4_21  T4_12  T4_22  T4_13  T4_23  T4_14  T4_24
T5_11  T5_21  T5_12  T5_22  T5_13  T5_23  T5_14  T5_24
T6_11  T6_21  T6_12  T6_22  T6_13  T6_23  T6_14  T6_24
T7_11  T7_21  T7_12  T7_22  T7_13  T7_23  T7_14  T7_24
T8_11  T8_21  T8_12  T8_22  T8_13  T8_23  T8_14  T8_24

Integers
X_1  X_2  X_3  X_4

End
```

In order to access the performance of the ILP model we compared it against the heuristic Worst Fit (WF) which performed well in the experimentation undertaken in [5]. In the work of [5] WF performed its allocation in two stages:

- **Stage One** Allocate the HI criticality tasks and locate point S^{max}.
- **Stage Two** Allocate the LO criticality tasks in the time remaining, $T^F - S^{max}$.

As the prior work dealt with the simpler single cycle model, the implementation of worst fit used in this work had to take into account the multi-cycle system. This is done as follows:

- **Stage One** Allocate the HI criticality tasks to minor cycles.
- **Stage Two** Allocate the LO criticality tasks to minor cycles.
- **Repeat For Each Minor Cycle**
  Stage One Allocate HI criticality tasks assigned to the minor cycle to cores and locate point S^{max}.
  Stage Two Allocate the LO criticality tasks assigned to the minor cycle in the time remaining, $T^F - S^{max}$.

Worst fit and ILP were compared by means of experimental data using randomly generated task sets. The parameters for this experimentation were as follows:

- Our experiments were based on a 4 core platform.
- Each task set consisted of 20 tasks.
- 10,000 task sets were generated at each 5% utilisation interval.
- Tasks were generated as follows: utilisations (U) were uniformly generated via UUniFast [4], periods were selected from the set $\{25, 50, 100\}$, $C(LO)$ values were created by, $C(LO) = U \times T$, $C(HI)$ values were created by multiplying $C(LO)$ values via a random value between 1.1 and 1.9.
- The criticality levels within a task set were evenly distributed.
- CE execution is split across 4 minor cycles where $T^F = 25$ and $T^M = 100$.
- Tasks may have periods of 25, 50 and 100. These are allocated randomly during taskset generation.

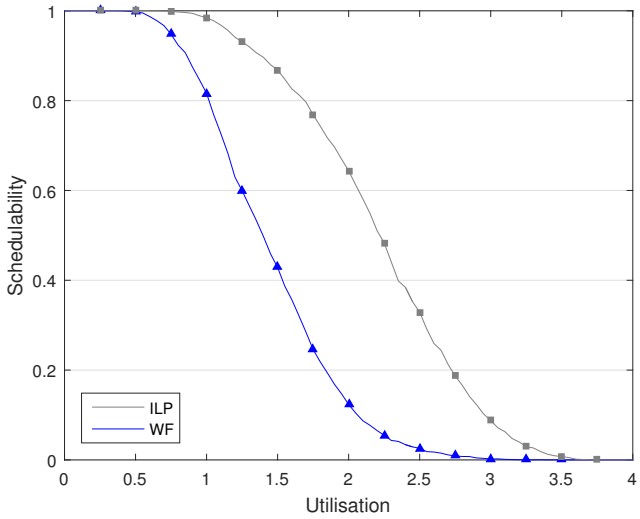


Fig. 1. The effectiveness of WF and ILP to generate CE schedules.

Figure 1 is typical of the results found over a range of parameters and shows that ILP significantly out performs the WF heuristic. Where ILP still boasts schedulability at nearly 0.8 WF is only able to manage around 0.1. ILP provides a clear improvement over the heuristic based techniques. With ILP being an optimal solver this improvement is somewhat expected, the question remains, are the overheads of using ILP in comparison to WF worth the increase in the number of CE schedules generated.

On an alternate note, if ILP is used in the constrained CE model from [5] where $T^F = T^M$ then the results are interesting.

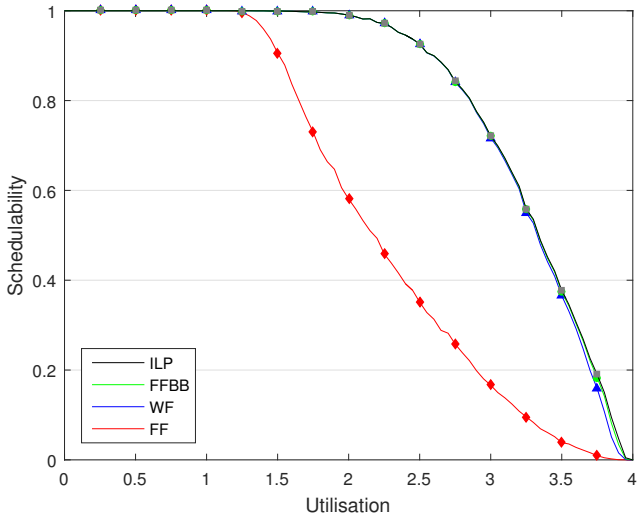


Fig. 2. A comparison between ILP and prior heuristics presented in [5].

Figure 2 shows that for the restricted case of a single minor cycle, the heuristics FFBB[1] and WF perform extremely close to the ILP solution. This makes it clear that these heuristics are very well optimised for this problem, but the additional complexity of multiple minor cycles is a significant issue. This is likely down to it becoming a two stage allocation process for the heuristic techniques (allocate to minor cycles then to cores within those cycles).

III. The efficiency of using ILP for CE schedule creation

As established in Section II, ILP can provide significant improvements for CE schedule creation, especially when a more complex CE model is considered. However ILP based solutions are well known for having high computational overheads. This section will investigate the overheads involved with our technique and show how ILP can be effectively used for schedule generation.

In order to investigate the computation time required for ILP in comparison to WF we made use of the inbuilt timing tools in Matlab. This provides a comparable baseline which allows the real world performance of each approach to be assessed. Our test platform consisted of a 32 core (AMD Opteron 6134) compute server. We timed the execution of the complex test with the aim of investigating the cost of the additional schedulability provided by ILP.

The figures in Table II show the average time taken to execute (in seconds), per task set for both ILP and WF. The parameters of this experiment were: 20 tasks per set with evenly distributed (dual) criticality levels.

	WF	ILP
Average Time (sec)	0.010	0.0125

TABLE II
THE AVERAGE EXECUTION TIME OF WF AND ILP.

While these results do show that on average the ILP solver takes longer than WF to solve the CE schedule generation problem, both times are negligible when you consider real world use. During experimentation thousands of task sets are tested in order to produce results, while the experiments take some time to run, a single set of tasks can be checked quickly. In a real use case it is likely that only a single version of the system need be checked at any given time and both WF and ILP take, on average, a very small amount of time to solve a single problem. ILP however boasts greatly increased schedulability, we observed an average increase of 0.19 and up to 0.53 at some utilisations. The question remains, if this is true for a system with 20 tasks, how do both approaches fair when the number of tasks are increased? In other words, is it scalable?

[1]Inital allocation is performed by First Fit, the largest and smallest S^{max} values are identified, these are used to perform a Branch and Bound search to attempt to minimise S^{max}. See [5] for details.

This was explored by re-running the experiments and varying the number of tasks from 20 to 100, the results of this are shown in Figure 3:

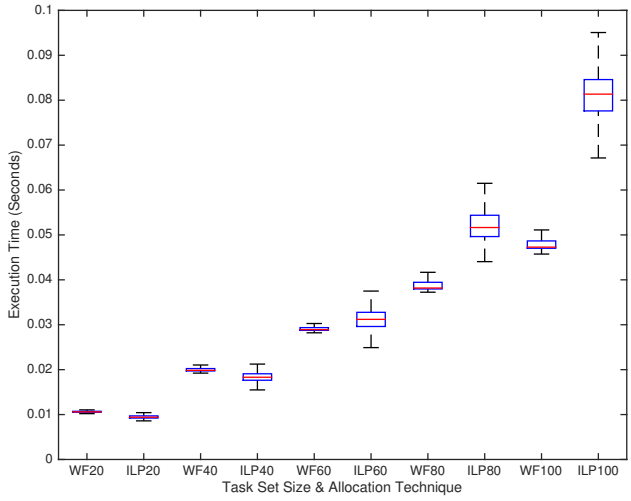


Fig. 3. The increase in computation time as the number of tasks per set is increased.

The results in Figure 3 show data for more than 99.99% of the task sets tested, the majority of the outliers not shown here completed within 4 seconds[2]. Although a rise in the average computation time can be seen as the number of tasks per set is increased, the time required still remains very low. The low execution requirements of the ILP implementation comes down to the desire to simply discover if a suitable schedule exists, no optimisation is required. As the number of tasks per set increases, the number of variables required to model the problem increases dramatically. However due to the binary nature of the variables that decide where a task is placed and the lack of a maximization requirement the execution time remains low.

It is also possible to observe this scalability with regard to the number of CPU cores in a system. Figure 4 shows again 99.99% of all task sets tested[3]. It is clear that our ILP solution is scalable both as tasks and CPU cores are added to the system.

During this investigation we do not claim to have the most efficient implementation of the WF heuristic and other aspects of the code could affect the timing results. The results are representative of the real world performance of the solutions and of their performance relative to each other. By increasing the number of tasks per set and cores we have shown that our ILP implementation is scalable, further reinforcing the argument for its use during the development of real world industrial systems.

[2]A small number did not complete within a 24 hour test period. The outliers are omitted as over 80000 task sets were tested per plot while the outliers numbered less than 100.

[3]as before all but a very small number of the outliers completed within 0.11 seconds and a small number did not completed within the 24 test period.

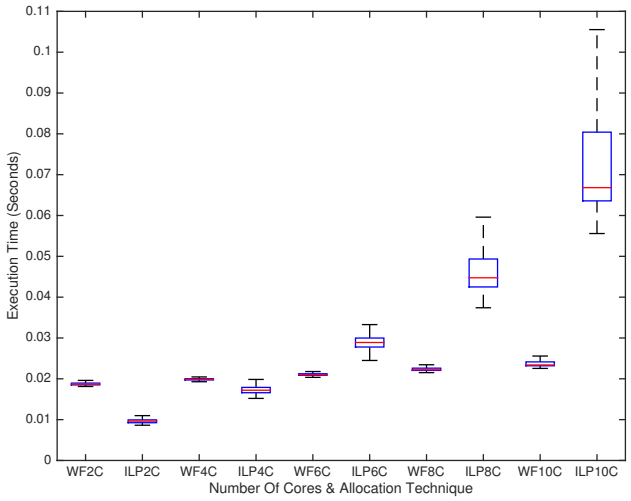


Fig. 4. The increase in computation time as the number of tasks per set is increased.

IV. Conclusions

Throughout this work we addressed the following problems. Firstly we extended the mixed criticality cyclic executive model used in [5] to consider multiple minor cycles per major cycle. This significantly increased the complexity of the allocation problem as now a schedule must be constructed by allocating tasks to the appropriate number of minor cycles, and from there to cores within each minor cycle.

Secondly we showed that when the heuristic based approach WF, is applied to the more complex system model it performs poorly due to the increased complexity of the allocation process. We introduce the notion of using ILP to model the CE system and check for a suitable valid schedule. We show that in the complex case ILP provides significant gains in schedulability over WF. Interestingly in the simple case with a single minor cycle we observed that the heuristics tested (excluding FF) perform very well and manage a level of schedulability close to that provided by ILP. In addition to this, the high MC performance of ILP clearly implies similarly good performance for the non-MC case.

Finally we showed through code timings that our ILP implementation was able to produce a result for a single task set within a very reasonable time frame. In addition to this we investigated the scalability of the solution showing that, although the execution time required did increase as a result of increasing the number of tasks in a set, the time taken was still very reasonable. We showed that for any practical application, an ILP model could be comfortably used to generate a mixed criticality cyclic executive schedule.

The specific ILP tool employed did demonstrate a small number of runs that either took an excessive amount of time to complete or indeed did not completed within a 25 hour period. Those that did complete were mainly unscheduledable. It is therefore a sensible pragmatic approach to deem the task set to be unschedulable if the tool did not obtain a result within 4 seconds.

At the start of this work we posed the question: Is it worth using an optimal solver for CE schedule generation over heuristic based techniques? We have shown that it is worth using, both from the angle of performance and computational efficiency. In addition to this we have shown that the ILP model proposed is scalable allowing it to handle practical system parameters with ease.

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). The authors also wish to thank Sanjoy Baruah for insightful discussions on the content of this work.

References

[1] T. Baker and A. Shaw. The cyclic executive model and ada. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 120–129, Dec 1988.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *Computers, IEEE Transactions on*, 61(8):1140 –1152, aug. 2012.

[3] S. Baruah and A. Burns. Achieving temporal isolation in multiprocessor mixed-criticality systems. In *WMC*, page 21, 2014.

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

[5] A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. ECRTS 2015, 2015.

[6] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15, Sept 2013.

[7] I. Gurobi Optimization. Gurobi optimizer 6.0. http://www.gurobi.com/.

[8] 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.