Predictability in Mixed-Criticality Systems

Rany Kahil, Peter Poplavko, Dario Socci, Saddek Bensalem

Verimag Research Report n^{o} TR-2018-8

July 02, 2018
last updated: July 2, 2018

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UGA
Bâtiment IMAG
Université Grenoble Alpes
700, avenue centrale
38401 Saint Martin d'Hères
France
tel : +33 4 57 42 22 42
fax : +33 4 57 42 22 22
http://www-verimag.imag.fr/

Predictability in Mixed-Criticality Systems

Rany Kahil, Peter Poplavko, Dario Socci, Saddek Bensalem

July 02, 2018
last updated: July 2, 2018

Abstract

Showing that a policy is predictable is a means to ensure sustainability of a solution proposed by a scheduling algorithm. A typical way to test a solution for correctness consists in conservative evaluation of its behavior under a specially selected set of high-workload runtime scenarios. Sustainability is a property that states that if a solution is deemed correct then it satisfies all the deadlines not only under the set of high-workload scenarios, but also in all less workload scenarios. The predictability property is a special case of sustainability under the assumptions that the workload is measured only by job execution times and that the correctness test consists of a simulation of the scheduling policy for finite set of jobs. In this paper we extend the predictability property to the case of mixed criticality jobs. We restrict ourselves for simplicity with just two levels of criticality. The ordinary, single criticality, scheduling policies commonly used in practice, are predictable in the usual sense, and thus it is sufficient to test them just for a single scenario with the worst-case execution times. We show that the typical generalization of the common scheduling policies to support the mixed criticality specifications renders them non-predictable, whereas they are still predictable according to our proposed extended definition. We also show that our definition implies applicability of a previously existing method for testing mixed-criticality jobs by simulation, which previously had only a narrow application.

Keywords: mixed-critical systems, real-time scheduling, computational complexity

Reviewers: Saddek Bensalem

Notes: Extended version of a short paper at 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'2018)

How to cite this report:

```
@techreport {TR-2018-8,
  title = { Predictability in Mixed-Criticality Systems },
  author = {Rany Kahil, Peter Poplavko, Dario Socci, Saddek Bensalem},
  institution = {{Verimag} Research Report},
  number = {TR-2018-8},
  year = { }
}
```

1 Introduction

In the context of this paper, under "scheduling algorithm" we understand a union of three ingredients. Firstly, it is an offline algorithm to construct a scheduling *solution* (*e.g.,* a priority table) from the system description. These offline solutions are used to configure the *online scheduling policy*, which is the second ingredient. Thirdly, an offline *correctness test* is required to give a verdict whether the system is deemed schedulable or not with the given solution and policy.

Determining the exact execution workload required by a system is generally not possible. Instead upper bounds are specified which are used to estimate the Worst Case Execution Time (WCET) of the system. Scheduling correctness tests base their analysis and correctness tests on these estimated upper bounds. The system that was proven correctly schedulable in the worst case scenario is expected to remain so as it performs better. Sustainability [1] is a characteristic of a scheduling algorithm which assures that if a system is deemed schedulable by that algorithm, it will remain schedulable if its runtime behavior is better than expected. Sustainability is a necessary property for any scheduling algorithm for timing-critical (hard real-time) applications.

In this paper we focus on correctness tests to ensure sustainability of a mixed-critical systems (MCS). We follow the well-known and generally accepted Vestal model [2] for the definition of mixed-criticality aspects of the system behavior. This model assumes that if a job exceeds the low-criticality WCET bound then the system switches the mode, whereby one does not care anymore about low-criticality jobs. We focus on the most basic variant of the MCS scheduling problem – scheduling a fixed finite set of jobs. This problem has obvious theoretical relevance, but also is applicable in practice for periodic systems where initial task release offsets are known apriori as well as for the online tests of correctness, for policies that compute their solutions partially online. We consider both single- and multi-processor system scheduling of mixed critical jobs. Moreover, like numerous previous research works, to simplify the problem we focus on dual-criticality problems (with only two criticality levels).

The correctness test for a fixed set of jobs is usually based on simulation of a single worst-case scenario, but if the system behavior is characterized by multiple corner cases one tests for a set of respective "basic scenarios" [3]. Very common online policies in the scheduling theory are *fixed job priority* (FP) policies. These policies include EDF, for example, and, for task systems, fixed priority (of tasks) such as RM (rate monotonic). These policies are *predictable*, which means that a decrease of execution time of jobs can only lead to a decrease (and never to an increase) of job termination times in a whole system. For this reason, testing the correctness of their schedulability by simulation may be restricted to just one worst-case scenario.

However, recall that MCS policies extend the above policies by introducing the mode switching. In this way, the concept of an FP policy is extended to fixed job priority per mode (FPM). For task systems, numerous single-processor FPM algorithms have been proposed, many of them being based on modifications of EDF, for example EDF-VD [4]. For fixed set of jobs, we proposed two FPM algorithms, MCEDF [5] for single processor, and MCPI [6] for multiprocessors.

An important observation is that, due of the mode switch, a policy may in general become non-predictable in the usual sense, and a simple test of simulation in one scenario does not apply anymore. Therefore previous literature [3] also generalized the correctness test by simulation such that it can be applicable also for mode-switched policies. This generalization extends the testing from a single basic scenario to what we call *canonical correctness test* on a specific set of basic scenarios. However, the authors specify only one particular scheduling policy – let us call it the reference policy – for which the canonical correctness test is applicable. They showed that a valid solution for all policies can be converted without loss of optimality into a reference policy solution.

In our previous work [5], [6] we have discovered that the canonical test is in fact directly applicable, without any conversion, to a quite general class of online policies, which satisfy a weaker definition of predictability, which we call here *weak-predictability*. We also previously conjectured that FPM policies belong to that class of predictability. However in the previous publications we only gave an imprecise definition of the weaker predictability and did not provide a proof that FPM is in that class. In the present work we close this gap. We give a definition and provide the missing proof for the single-processor case. Thereby we give the missing formalization of our previous work on FPM, whereas we think it can be important also for any other work where offline or online simulation of jobs is done to test correctness.

We also make an important correction to our previous work by showing that, in fact, contrary to what we believed before, FPM is not predictable in MCS sense for multiple processors. Consequently, in this case one cannot directly use an FPM solution tested by the canonical test in the context of online global FPM policy. Luckily, for our multiprocessor algorithm MCPI [6] we already did a conversion to a reference policy [7]. It still remains future work to carefully study whether other authors wrongly apply the canonical correctness test for FPM on multiple processors.

Last but not the least, we exploit the fact that the canonical correctness test of FPM takes polynomial time, in order to prove that FPM scheduling for dual-critical instances on single processor belongs to complexity class NP. Note that in [8], we refuted a previously established proof that MCS scheduling in general is in NP. By showing that FPM is in NP we show that there exists an important policy whose exact solutions can be potentially constructed by known formal solving techniques for NP problems, such as SAT (satisfiability) solvers.

2 Problem Formulation

Studying the predictability of a scheduling policy requires knowledge of job execution times, in order to be able to determine if a policy behaves as predicted during runtime. For this reason we model the workload of a mixed criticality system as being a set of finite periodic mixed criticality tasks having different WCET estimates, providing different level of assurance [2]. Tasks are assumed to be independent and have no starting offsets. Our focus will be on dual-criticality systems. These systems have only two levels of criticality, the high level denoted as "HI", and the low (normal) level, denoted as "LO". An MC periodic task is defined by a period, a relative deadline, and two WCET estimates one for each criticality level, the LO WCET and the HI WCET. The former one is for normal safety assurance, used to assess the sharing of processor with the participation of LO jobs, and the other one, a higher value, is used to ensure certification of HI jobs only. An MC task can generate an infinite number of MC jobs where each job takes the criticality level of its task.
A *job* J_j is characterized by a 5-tuple $J_j = (j, A_j, D_j, \chi_j, C_j)$, where:

- $j \in \mathbb{N}_+$ is a unique index
- $A_j \in \mathbb{N}$ is the arrival time, $A_j \geq 0$
- $D_j \in \mathbb{N}$ is the deadline, $D_j > A_j$
- $\chi_j \in \{\text{LO}, \text{HI}\}$ is the job's criticality level
- $C_j \in \mathbb{N}_+^2$ is a vector $(C_j(\text{LO}), C_j(\text{HI}))$ where $C_j(\chi)$ is the WCET at criticality level χ

The index j is technically necessary to distinguish between jobs with the same parameters. The timing parameters A_j, D_j, C_j are integers that correspond to time resolution units (*e.g.,* clock cycles). We assume that: $C_j(\text{LO}) < C_j(\text{HI})$. This makes sense, since $C_j(\text{HI})$ is a more pessimistic estimation of the WCET than $C_j(\text{LO})$. We also assume that LO jobs are forced to terminate after $C_j(\text{LO})$ time units of execution, so: $(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$.
Focusing on periodic tasks enables us to study the schedulability of the system by studying the schedulability of the set of jobs generated by these tasks over a hyper-period. The hyper-period is simply the least common multiple of the periods of the tasks in a system. Tasks within a hyper-period will always generate the same set of jobs. Thus if this set is found schedulable over one hyper-period, then the task set will be schedulable. Although hyper-periods can be large in non-trivial systems, some work has been done for adjusting the periods of tasks in a system to get a smaller hyper-period [9]. Thus we define an *MC instance* of the scheduling problem to be a set of jobs $\mathbf{J}$.
In our problem definition we do not permit $C_j(\text{LO})$ to be equal to $C_j(\text{HI})$ for HI jobs, though this is commonly allowed in the literature [3]. Allowing this possibility would cause anomalies and complications with regards to correctness testing, as shown in an example in Section 4. Furthermore this should not be an issue, as in the usual case, $C_j(\text{HI})$ is larger than $C_j(\text{LO})$ since it is expected to provide higher levels of assurance.
A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $c = (c_1, c_2, \ldots, c_K)$, where K is the

number of jobs. We only consider non erroneous scenarios where no c_j exceeds $C_j(\text{HI})$. The *criticality of scenario* $c = (c_1, c_2, \ldots, c_K)$ is LO if $c_j \le C_j(\text{LO}),\ \ \forall j \in [1, K]$, is HI otherwise.

Basic scenarios are scenarios c such that:

$$\forall j = 1, \ldots, K \quad c_j = C_j(\text{LO}) \vee c_j = C_j(\text{HI})$$

One special basic scenario is the *"LO basic scenario"* where all jobs execute for exactly their $C(\text{LO})$.

A *schedule* $\mathcal{S}$ of a given scenario c is a mapping: $\mathcal{S} : T \mapsto \widehat{\mathbf{J}}_m$, where T is the physical time and $\widehat{\mathbf{J}}_m$ is the family of subsets of $\mathbf{J}$ that contains all subsets $\mathbf{J}'$ of $\mathbf{J}$ such that $|\mathbf{J}'| \le m$, where m is the number of processors. Every job J_j should start at time A_j or later and run for no more than c_j time units. We assume that the schedule is *preemptive* and that job migration is possible, *i.e.,* that any job run can be interrupted and resumed later on the same or different processor. Note that in this definition we do not include the mapping of jobs to processors, but a valid mapping, if needed, can be easily obtained from a simulation which assumes that a job can be started or resumed at any available processor.
A job J is said to be *ready* at time t if at that time or earlier it has already arrived and has not yet terminated. The online state of a run-time scheduler at every time instance consists of the set of terminated jobs, the set of *ready jobs*, the remaining workload of ready jobs, *i.e.,* for how much they should still execute in future, and the current *criticality mode*, χ_{mode}, initialized as χ_{mode} = LO and "*switched*" to "HI" as soon as a HI job exceeds $C_j(\text{LO})$. It should be noted that in a given schedule it is the first job that exceeds its $C_j(\text{LO})$ that *switches* the mode and then the mode remains HI until the end of the schedule. We assume that all LO jobs are dropped upon switching to HI mode.
Based on the online state, a *scheduling policy* deterministically decides which ready jobs are scheduled at every time instant on m processors. A scheduling policy *correctly schedules* a given problem instance if the following conditions are respected in any possible scenario:

Condition 1. If all jobs run at most for their LO WCET, then both critical (HI) and non-critical (LO) jobs must terminate before their deadline.

Condition 2. If at least one job runs for more than its LO WCET, then all critical (HI) jobs must terminate before their deadline, whereas non-critical (LO) jobs may even be dropped.

A policy is said to be *work-conserving* if it never idles the processor if there is pending workload. An instance **J** is *MC-schedulable* if there exists a correct scheduling policy for it.
The predictability of different scheduling policies will be studied in the next section. A short description of each is provided in this section, starting by the conventional mixed-criticality agnostic *fixed job priority* (FP) policy. FP is a scheduling policy that can be defined by a priority table PT, which is a K-sized vector specifying all jobs in a certain order. The position of a job in PT is its *priority*, the earlier a job is to occur in PT the higher the priority it has. Among all ready jobs, the fixed-priority scheduling policy always schedules the m highest-priority jobs in PT. Fixed priority is a *work-conserving* policy. A priority table PT defines a total ordering relationship between the jobs. If job J_1 has higher priority than job J_2 in table PT, we write $J_1 \succ_{PT} J_2$ or simply $J_1 \succ J_2$, if it is clear from the context to which priority table we are referring to.
Fixed priority per mode (FPM) is a natural extension of fixed-priority for mixed critical systems. It has one priority table for each criticality mode. In the case of dual- criticality systems FPM has two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter needs to include only the HI jobs. As long as the current criticality mode χ_{mode} is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After a switch to the HI mode, this policy drops all pending LO jobs and applies table PT_{HI}.
"*FPM-equivalent tables*" relation "$PT_{\text{LO}} \sim PT_{\text{HI}}$" means that by removing the LO jobs from PT_{LO} while keeping the same relative order of the HI jobs we obtain the same job order for HI jobs as in PT_{HI}. In this case, after a mode switch to HI mode and dropping LO jobs, using PT_{LO} or PT_{HI} will result in the same schedule.

3 Predictability in MC-Scheduling

Predictability and sustainability although similar, are not identical in the mixed criticality context. To clearly distinguish the difference between the two concepts, we start by including a formal definition of both. With

regards to sustainability, a good amount of research has been devoted to the study and analysis of this property in the scheduling of real time systems. In [1] the authors formalize the sustainability characteristic in real time systems (non mixed critical) as follows.

"A schedulability test for a scheduling policy is sustainable if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual jobs are changed in any, some, or all of the following ways: decreased execution requirements; later arrival times; smaller jitter; and larger relative deadlines".

In [10], the definition of sustainability was extended to mixed criticality systems. The workload of an MC system was modeled as a finite collection of sporadic tasks with each task having a criticality and possibly generating an unbounded number of mixed criticality jobs. We include the adapted definition for reference.

*"**MC sustainability [10].** An MC scheduling policy is said to be sustainable if any MC instance that is MC-schedulable by the policy remains so if one or more of the parameters characterizing the instance is improved. Improvements to be characterizing parameters:*

1. *Decreasing WCET parameters*
2. *Increasing periods for sporadic task systems*
3. *Postponing relative deadlines*
4. *Decreasing the criticality level assignment of a task/job."*

The authors of [10] focus their work as well on dual criticality instances. Six mixed criticality scheduling policies were evaluated to check whether they are MC-sustainable for the different parameters. It was found that the polices are all sustainable with respect to WCET, periods and deadlines but some are not sustainable with respect to the criticality level parameter. Among the six evaluated policies we chose the simplest to show that it is not predictable according to the definition below.

Definition 1. A scheduling policy is said to be **predictable**, if for any scenario that is MC-schedulable by the policy, any other scenario that is better is also MC-schedulable by the policy. For predictability, "MC-schedulable" means that *simulating* the given scenario shows that the policy correctly schedules all jobs, whereas scenario S_1 is considered to be "better" than scenario S_2, if any job in S_1 executes for the same amount or less than it does in S_2.

The parameter that is most interesting to our work from the definitions of sustainability is the *"decreased execution requirements"*. Execution requirements are often given by WCET. These are upper bounds that may overestimate the worst case of the execution due to difficulties in modeling processors with caches, out of order executions, pipelines etc. Even in the case of a tight upper bound, a job can take numerous execution paths that are different from its worst-case path resulting in various execution times that are considerably less that the WCET.

In non mixed-criticality systems[1] distinction between sustainability and predictability was not needed, as sustainability trivially ensured predictability. To show this, suppose that in a system that was deemed schedulable with respect to WCETs, a job J_1 has a WCET of C_1. In a certain scenario S_1, J_1 executes for c_1 such that $c_1 < C_1$. If the scheduling policy is sustainable with respect to WCET then the system should remain schedulable in the case where $C_1' = c_1$ since $C_1' < C_1$ where C_1' is the decreased WCET of J_1. But this case is equivalent to scenario S_1 showing its sustainability with regards to execution times. We will show later that this is not in general the case in MC systems clarifying why this reasoning does not work.

3.1 The Notion of Weak Predictability

In the study of MC-sustainability characteristic the focus was on upper bounds of sporadic tasks and not exact execution times. In our study of predictability the focus is on evaluating the execution of job set to be examined with regards to execution time of jobs. We will demonstrate that looking at exact execution times can be essential to guaranty that a sustainable MC policy will be able to correctly schedule a system that executes

[1] in those tested for correctness by simulation

for less than the estimated WCET. We begin by giving an example of an MC scheduling policy that is MC sustainable but not predictable. Criticality Monotonic (CM) scheduling policy is one of the policies that were studied in [10]. A CM scheduling policy schedules at each time instant a ready job of the highest criticality. It was proven in [10] that a CM scheduling policy using deadline monotonic scheduler within a criticality level is MC sustainable. As a consequence it is sustainable with regards to the WCET parameter. In the example below we show that although this policy is MC-sustainable it is not predictable and there is a case where the policy will correctly schedule the system but will fail to schedule it when some job executes for less.

Example 1. Consider a periodic system that has two tasks with periods T = 20. At start of the system each task will release a job. The jobs to be scheduled are shown in the table below:

Job	A	D	Criticality	C(LO)	C(HI)	Priority
1	0	10	HI	6	8	1
2	0	10	LO	5	5	2

Example 1 represents a mixed criticality system with two tasks, one having LO criticality and the other is of HI criticality. Since both tasks have the same period, each task will generate one job during a hyper-period. Jobs generated are shown in the table. We assume the system is being scheduled by a CM policy with deadline monotonic scheduler within a criticality level. Testing the correctness of this policy for the worst case scenario, a simulation will be performed where J_1 is assumed to execute for 8 units of time and J_2 for 5.

The simulation of the policy will execute J_1 first, as it is the highest criticality job, until it terminates at $t = 8$. Then it will schedule J_2 which will terminate at $t = 13$ missing its deadline. Although J_2 missed its deadline, the system is deemed MC-schedulable. This is the case because J_1 executed for more than its $C(LO)$ thus only HI criticality tasks are required to meet their deadlines. It could be the case that at runtime job J_1 executes for only 6 units of time instead of 8. In this case J_2 still misses its deadline, but now the system is no longer MC-schedulable since no job executed for more than its $C(LO)$, and in this case all jobs are required to meet their deadlines. This oversimplified example shows the complications that arise in testing for correctness in MC-systems. It gives one case of an MC-sustainable policy being able to correctly schedule a scenario but failing to schedule another one with decreased execution requirements on one job.

Example 1 shows that unlike the single criticality case, in an MC system not all MC-sustainable scheduling policies are predictable. Another observation is that correctly testing the schedulability of a policy by simulating the worst case scenario does not imply anymore the correctness in other scenarios. We expect that the CM policy will not be the only one that is not predictable. This is primarily due to the fact that decreasing the execution of a high criticality job, might stop the system from switching to HI-criticality mode, thus all jobs are required to meet their deadlines whereas before the decrease only HI jobs had to meet their deadlines. For this reason we provide a weaker definition of predictability that will be sufficient to perform a sustainable schedulability test proposed in the next section.

Definition 2 (Weak Predictability)**.** An MC scheduling policy is weakly-predictable if for any scenario that is MC schedulable, decreasing the execution time of a job A – while keeping all other execution times the same – should not delay the termination time of any other job B under the following two conditions:

- If job A caused a mode switch, then the decrease in the execution of job A does not cancel the mode switch that was caused by A
- Job B terminates in the same criticality mode, before and after the decrease of execution of A

In a weakly-predictable policy, if at least one of the two conditions above is not met a decrease in the execution of one job is allowed to delay the termination of another job. Thus a weakly-predictable policy does not always have to be predictable. But a predictable scheduling policy is always weakly-predictable and all the results that follow from the weakly predictable property can be applicable. The main intuition behind this weak definition of predictability is that it removes the difficult case when a decrease in an execution time cancels the mode switch, adding the requirement to verify that LO jobs meet their deadlines. Thus MC-policies are more likely to be weakly-predictable and able to use the correctness test proposed later on for such policies.

We formulate below another interpretation of the definition of a weakly-predictable policy, in the case of increasing (instead of decreasing) the execution of a job. This interpretation is equivalent to the definition

given above, we only include it as it proves helpful and is used later on in the proofs.
A scheduling policy is weakly-predictable if for any scenario that is MC schedulable, increasing the execution time of any job A – while keeping all other execution times the same – should not make any other job B terminate earlier only when the following two conditions hold:

- If job A did not cause a mode switch then the increase of its execution also does not lead it to cause the mode switch.
- Job B terminates in the same criticality mode before and after the increase of execution of A

If at least one of these two conditions is violated then B may terminate earlier. The first condition can only be violated if before the increase A executed for at most LO WCET and after the increase it is the first job to exceed the LO WCET thus causing a mode switch.

3.2 Predictability in Fixed Priority per Mode

The following theorem from [11] states a very useful property, for which we formulate a corollary:

Theorem 1. Fixed-priority policy is sustainable and thus predictable with respect to execution times, for single- and multi-processor scheduling.

Corollary 1. For *single-processor* dual-criticality instances FPM is weakly predictable.

Proof. Consider a dual-criticality FPM policy with given priority tables PT_{LO} and PT_{HI}. Consider any scenario c. For a given job A, let scenario c' differ from c only by an increase in execution time of job A by Δc_A, such that this increase does not lead A to be the job that causes a mode switch in c'. Let job B be an arbitrary job. We have to prove that B can only terminate at the same time or later in c' compared to c, but never earlier, provided that B terminates in both scenarios in the same criticality mode.

If there is no mode switch in c then predictability in this case follows from the predictability of FP scheduling. So let us first consider the case where A terminates after the mode switch in c. This means that the schedules of c and c' are the same up until the switch time. At switch time the same LO jobs are dropped, if any, and both scenarios are left to execute the same jobs using same priority table PT_{HI} with only one difference, the increase in the execution of A. Thus it follows directly from the predictability of FP that in this case B will never terminate earlier in c'.

Secondly, we consider the case where both A and B terminate before the mode switch in c. If the increase in execution leads B to terminate after the mode switch then the condition that B terminates in the same criticality mode does not hold and we have nothing to prove. If after the increase in the execution of A, job B terminates before the mode switch then by predictability of FP it can not terminate earlier than in c.

Hence the only non-trivial case that we are left with is that in scenario c, job A terminates before the switch executing entirely in the LO mode, and job B terminates in both scenarios after the mode switch.

Let t_c and t'_c be the switch times of c and c'. Due to the predictability of FP scheduling, up until the switch time in c, no job other than A can exceed its $C(\text{LO})$ in c' before it does in c. And since A does not cause the mode switch, then we have $t_c \leq t'_c$.

Let t_A be the termination time of job A in c. Let t_B and t'_B be the termination time of job B in c and c' respectively. Since both terminate after the switch then we have $t_B > t_c$ and $t'_B > t'_c$. But for $t_B \leq t'_c$ we have $t_B < t'_B$ and this completes the proof for this case. Thus we still have to prove for $t_B > t'_c$.

The execution of jobs in the interval $[0, t_A]$ is the same in both scenarios c and c'. Let us consider only the HI jobs that do not terminate by time t_c in c. In the time interval $[t_A, t_c]$, both scenarios are using PT_{LO} as the priority table. A executed for the same amount or more and FP is predictable thus all other HI jobs executed for the same amount or less in c' compared to c. Then in c' compared to c, every job has to execute for at least the same amount or more between t_c and its termination.

Let us make the following assumption for scenario c. Assume that between t_c and t_B there are no idle intervals and the (HI) jobs which execute there have equal or higher priority than B (w.r.t. PT_{HI}). We refer to this set of jobs as S_B. In addition let E_B be the execution time between t_c and t_B. This is the execution time needed for all jobs in S_B to terminate, as they all have higher priority than B and hence terminate before t_B.

Now let us consider scenario c'. Recall that at time t_c at least the same set of jobs have to execute the same amount of work as in c. Let us consider what happens after time t_c. Since $t'_B > t'_c$ and all jobs in S_B have equal or higher priority than B according to PT_{HI} then all other jobs in S_B terminate before B does. We are left with two cases:

- Either only jobs from set S_B execute between t_c and t'_c and in this case $t_B \leq t'_B$ follows from the fact that FPM is a work conserving policy and these jobs have at least the same amount of work to execute in c'.

- Jobs outside of S_B execute as well (having higher priority according to PT_{LO}) between t_c and t'_c but in this case we also have $t_B \leq t'_B$ from the argument on the amount of work to be executed.

Now we remove our earlier assumption for scenario c, instead we suppose that in the time interval $[t_c, t_B]$, there are one or more sub-intervals where the processor is idle or it executes jobs with lower priority than B according to PT_{HI}. We note that this can be the case only if during these sub-intervals there are no "ready" jobs in S_B to execute since B has the lowest priority in S_B and FPM is work conserving.

Let $[t_i, t_j]$ be the last subinterval where the processor was either idle or executing a job with lower priority than B. Thus in $[t_j, t_B]$, only jobs from S_B execute and we will refer to this set of jobs as S'_B. It is easy to see that none of the jobs in S'_B arrive before t_j.

Noting that in both scenarios, c and c', jobs in S'_B can not execute before t_j as they do not arrive before that time. Also,in both scenarios, all jobs in S'_B have to execute for the same amount, and terminate before B does since B terminates in HI mode in both scenarios and is scheduled by PT_{HI} when it terminates. In addition in c only jobs in S'_B are being scheduled in $[t_j, t_B]$ with no idle intervals included while in c' the same set of jobs are executed where B terminates last but also some other jobs might execute, thus B cannot finish earlier in c'. □

Corollary 2. For *single- and multi-processor* dual-criticality instances, an FPM policy that generates only FPM-equivalent tables is weakly-predictable.

Proof. Consider a dual-criticality FPM policy with given priority tables PT_{LO} and PT_{HI}. Let scenarios c, c' and jobs A, B be defined as in the proof of Corollary 1. We have to prove that B can only terminate at the same time or later in c' compared to c, but never earlier, provided that B terminates in both scenarios in the same criticality mode. The same reasoning as before can be used to show that the only non-trivial case is that in scenario c job A terminates before the mode switch and job B terminates in both scenarios after the mode switch.

Let $\mathcal{S}$ be the schedule generated for c and $\mathcal{S}'$ the schedule generated for c'. Let t and t' be the mode switch time in $\mathcal{S}$ and $\mathcal{S}'$ respectively. By the predictability of FP, and the fact that job A does not cause the mode switch, we have $t \leq t'$. Since LO jobs are dropped at switch time, only HI jobs that did not finish before the switch will execute after time t in $\mathcal{S}$.
Since FP scheduling is sustainable, no job other than A can execute in $\mathcal{S}'$ more than it executed in $\mathcal{S}$ up until t. Thus, the same set of HI jobs that have to execute after t in $\mathcal{S}$ will execute in $\mathcal{S}'$ and possibly jobs may require more execution time in $\mathcal{S}'$. In addition, some LO jobs may also execute after t in $\mathcal{S}'$ as they are dropped at t' with $t \leq t'$.

Note that as a consequence of having FPM-equivalent tables, using PT_{LO} instead of PT_{HI} after dropping the LO jobs at a mode switch, will not change the generated schedule, because the priority order of the HI jobs is the same in both tables. Thus for FPM policies having PT_{LO} and PT_{HI} FPM-equivalent, using any table after a mode switch results in the same schedule. Hence an FP scheduling policy that uses PT_{LO} to schedule the workload remaining after time t in c will generate the same schedule as $\mathcal{S}$. Also, using the same FP policy to schedule the workload remaining after time t in c' will generate the same schedule as $\mathcal{S}'$.

Since after time t the workload in $\mathcal{S}'$ is more than that in $\mathcal{S}$ and by the predictability of FP scheduling, then there is no such job B that terminates in $\mathcal{S}$ before $\mathcal{S}'$ after time t.

□

Lemma 1. An FPM policy that doesn't restrict its tables to be FPM-equivalent is not weakly predictable for multiple processors in general.

Proof. Example 2 provides an FPM scheduling policy where PT_{LO} and PT_{HI} are not FPM-equivalent. We show that it is not weakly predictable. □

Example 2. Consider the following 3-processor problem with instance **J** described in the table below:

Job	A	D	χ	C(LO)	C(HI)
1	0	6	LO	6	6
2	0	14	HI	4	5
3	6	15	HI	7	8
4	6	8	HI	1	2
5	6	9	HI	1	2
6	6	11	HI	3	4
7	6	13	HI	3	4
8	0	6	LO	6	6
9	0	7	LO	6	6

The Gantt chart in Figure 1 shows the execution in two scenarios: c and c' for an FPM scheduling policy with the priority tables specified in the figure, whereby $PT_{\text{LO}} \sim PT_{\text{HI}}$ is not satisfied and hence weak predictability is not guaranteed. Scenario c' differs from scenario c only by a larger execution time of J_1. The priority tables of the two modes in this example differ only by the relative priority of J_5 and J_6 and the window between τ and τ' is just one time unit. Nevertheless we see that job J_7 (as well as J_5) terminates in scenario c' earlier than in scenario c. This behavior contradicts the requirements of weak-predictability.

This example illustrates not just an exceptional case but well-known common properties of multiprocessor scheduling, differentiating them from single-processor case. Changing the order of job execution leads to a change of load distribution of different jobs between processors, which leads to different interference *w.r.t.* lower priority jobs. In our case, in window $[\tau, \tau']$ swapping the priority order between J_5 and J_6 has perturbed the load balance between the processors, such that a smaller priority job J_7 terminates earlier. Note that in both priority tables the set of jobs that have higher priority than J_7 is the same and all of them arrive no later than J_7. Under the same conditions on single processor these jobs would inevitably have the same total interference on J_7 in the two scenarios, but not on multiple processors.

4 Testing for Correctness

There are two general classes to test the schedulability of a system, either by simulation or response time analysis. In this section our focus is on schedulability tests by simulation. One possible schedulability test for a fixed set of jobs is the examination of basic scenarios that should represent all corner cases of execution times. This test can be applied for fixed set of jobs offline, and, potentially, for task systems online at the moment when the job arrival times are known until a point when the processors become idle.

In order to adapt with the requirements of mixed criticality systems, scheduling policies had to be modified by allowing LO jobs to miss their deadlines or even be dropped, in cases of mode switch to HI criticality. In particular, the previous literature has extended the concept of fixed job priority to fixed job priority per mode (FPM). For a fixed set of jobs, we proposed two FPM algorithms, MCEDF for single processor, and MCPI [6] for multiprocessors. For task systems, numerous single-processor FPM algorithms have been proposed, many of them being based on modifications of EDF, for example EDF-VD[4].

Unfortunately, because these policies support a mode switch, they become non-predictable in the usual sense, and a simple test of simulation in one scenario does not apply. We propose an adapted simulation-based schedulability test, that verifies the correctness of a scheduling policy and asserts a correct predictable behavior during runtime in case it is successful.

4.1 Basic Scenarios for Correctness Testing

Definition 3. [Basically Correct Policy] [3] An online scheduling policy is basically correct for instance **J** if for any basic scenario of **J** the policy generates a feasible schedule.

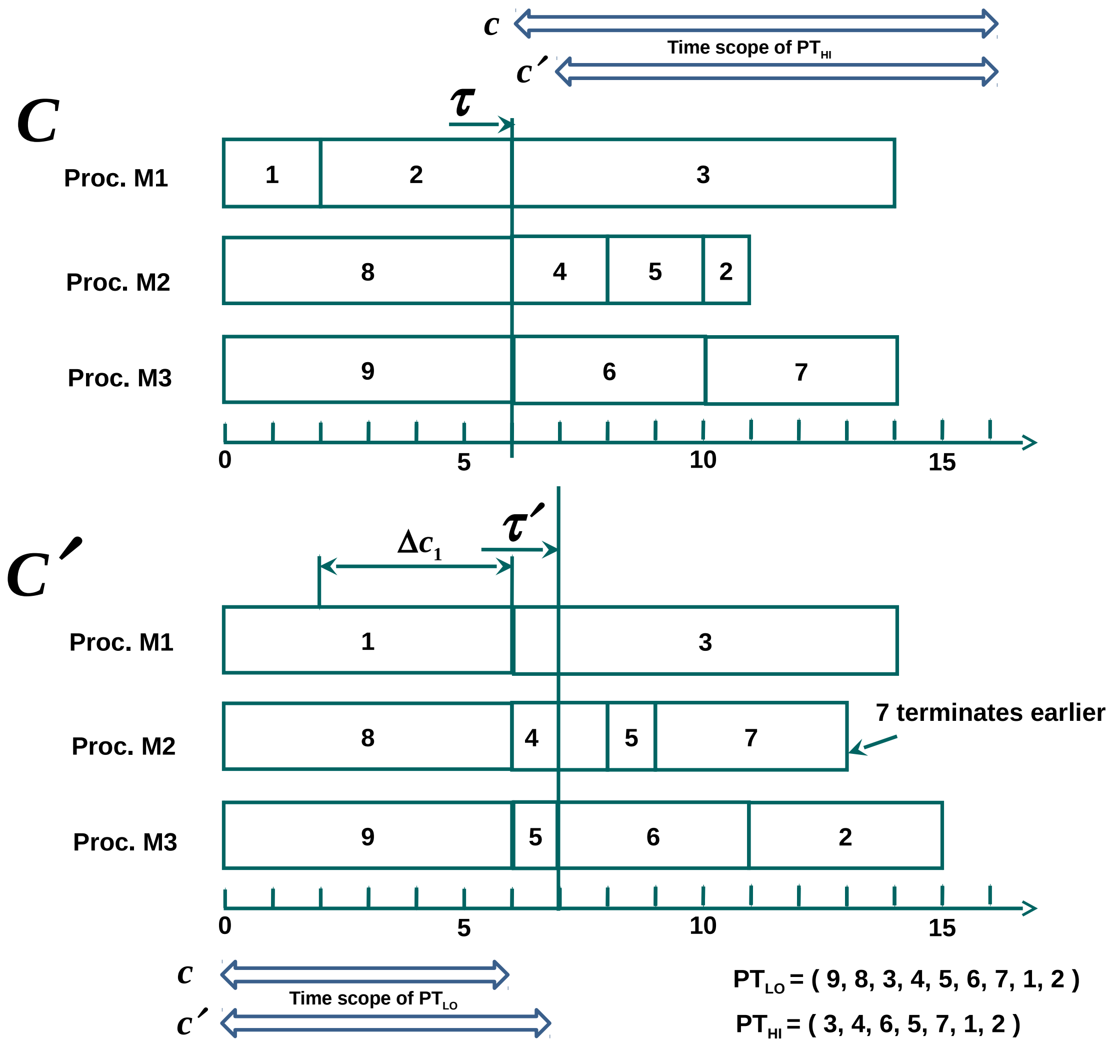

Figure 1: FPM non-predictable demonstration on multiprocessor case, using Example 2. Gantt charts of two scenarios: c and c'. Mode switch times are τ and τ', resp. Scenario c is defined by $(c_1 = 2, c_2 = 5, c_3 = 8, c_4 = 2, c_5 = 2, c_6 = 4, c_7 = 4, c_8 = c_9 = 6)$. Scenario c' differs from c by $c'_1 = c_1 + \Delta c_1 = 2 + 4$. Job J_7 violates weak-predictability conditions by terminating in c' earlier than in c, while terminating in the same mode (HI) in both scenarios.

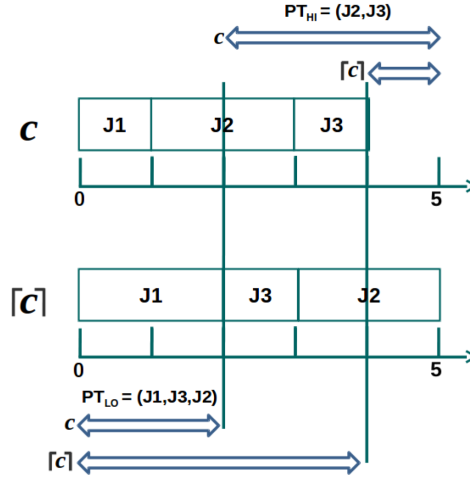

Figure 2: A scenario where for Example 3, WCETs of a HI job are equal, J_3 terminates in $\lceil c \rceil$ earlier than c

Lemma 2. [Correctness Test by Checking all Basic Scenarios] If a scheduling policy is weakly-predictable then the policy correctness follows immediately from its basic correctness. In other words, if the policy gives a feasible schedule in all basic scenarios then this is also the case for the non-basic scenarios as well.

Proof. For a given scheduling policy, let us call basic scenario $\lceil c \rceil$ the ceiling scenario of scenario c if in $\lceil c \rceil$ each J_i executes for time $C_i\ (\chi_{\text{TERM}}(c,i))$, where $\chi_{\text{TERM}}(c,i)$ is the mode in which job J_i terminates in scenario c. It is obvious that in $\lceil c \rceil$ all the jobs have at least the same or higher execution time.

The plan of the proof is as follows. Let $\mathbf{J}_{\text{TERM}}(c,\chi)$ be the set of jobs that terminate in c in mode χ. We split the set of all jobs into $\mathbf{J}_{\text{TERM}}(c,\text{LO})$ and $\mathbf{J}_{\text{TERM}}(c,\text{HI})$. It is easy to show that by weak predictability all the jobs in the first subset will terminate in the LO basic scenario no earlier than in scenario c. In the rest of the proof we show that the other subset will terminate in the ceiling scenario, $\lceil c \rceil$, no earlier than in scenario c. Thus, the correct termination can be checked in basic scenarios, namely in the "LO" scenario and in the scenario "$\lceil c \rceil$".

To prove for the second subset of jobs, suppose we could build a sequence of scenarios $c_1,\dots c_m \dots c_M$ such that $c_1 = c$, $c_M = \lceil c \rceil$ and we would obtain c_{m+1} from c_m by increasing the execution time of some job "A" in such a way that this increase does not let "A" cause a mode switch. Suppose also that the jobs from $\mathbf{J}_{\text{TERM}}(c,\text{HI})$ would terminate in all scenarios c_m in the HI mode. By weak predictability this would lead to the required conclusion.

The first subsequence of the required sequence is obtained by iteratively taking a job from $\mathbf{J}_{\text{TERM}}(c_m,\text{HI})$ that executes for less than $C_j(\text{HI})$ and increasing its execution time to $C_j(\text{HI})$ in c_{m+1}. This job terminates in HI both before and after the increase, so it does not cause a mode switch. We repeat the process until no matching jobs are left.

In the second subsequence, we take the jobs from $\mathbf{J}_{\text{TERM}}(c_m,\text{LO})$ and increase their execution times to $C_j(\text{LO})$. It is easy to show by induction that the resulting sequence satisfies the requirements. Note that if we allowed job systems with $C_j(\text{LO}) = C_j(\text{HI})$ for HI jobs then in the second subsequence some jobs that terminated in HI may start terminating in LO after a job execution time increase. In Example 3 we show that systems with $C_j(\text{LO}) = C_j(\text{HI})$ for HI jobs would violate the present lemma.

□

Example 3. Fig. 2 shows two scenarios for a job instance that allows WCETs of HI jobs to be equal:

Job	A	D	χ	$C(\text{LO})$	$C(\text{HI})$
1	0	2	LO	2	2
2	0	5	HI	1	2
3	2	3	HI	1	1

Keeping in mind that FPM policy, which we apply here, is weakly-predictable, we have two important points to observe in this example. First, observe that J_3 terminates in the HI mode in scenario c while it terminates in the LO mode in its ceiling scenario $\lceil c \rceil$. Thus, if the two WCET estimates of a HI job are allowed to be equal, weak-predictability is not always sufficient to ensure that the basic scenarios cover all other scenarios. Second observation is that this instance has only two basic scenarios, one is $\lceil c \rceil$ and the other one is exactly the same but with job J_2 terminating at t = 4 instead of switching to execute up until t=5. In both basic scenarios the instance is schedulable by the given FPM policy. Yet in c job J_3 misses its deadline. This shows that in the case where a high criticality job is allowed to have its $C(LO) = C(HI)$, even if all possible basic scenarios are simulated and successfully scheduled, this might not be enough to ensure the correctness of the scheduling policy with respect to all possible runtime behaviors. Due to this complication and the fact that we believe that disallowing HI jobs to have equal WCET does not limit the model as it can be ensured by an arbitrarily small increase of C_{HI}, we decided to include this restriction in our problem formulation.

Lemma 3. [Basically Correct Policy is Sufficient to Schedule an Instance] An instance **J** is MC-schedulable if it admits a basically correct scheduling policy.

The above lemma is Lemma 1 from [3]. At first glance, it seems to be contradicting to the observations we just made in Example 3, where the basic scenario coverage is not sufficient for FPM schedulability, but it should be noted that the lemma only claims that a correct policy exists, not that this policy is FPM. In the proof given in [3] they show a simple procedure to transform any basically correct policy into a similar policy which is, in fact, weakly predictable, and hence correct, by Lemma 2.

In fact, Lemma 2 implies that a complete correctness test can be reduced to testing all basic scenarios. However, this statement does not directly show that a complete correctness test of a weakly predictable policy can be done in polynomial time, as it requires testing all basic scenarios, whereas there is an exponential number of such scenarios. Fortunately, testing in all basic scenarios is redundant. Suppose that we have a predictable scheduling policy. It turns out that to test the policy correctness for a dual-critical instance it suffices to simulate $H + 1$ basic scenarios, where H is the total count of HI jobs in the problem instance, as shown in the next subsection.

4.2 Canonical Correctness Test

Definition 4. [Job-specific Basic Scenario] For a given problem instance, scheduling policy and a HI job J_h, the job-specific basic scenario for job J_h is denoted by $HI\text{-}J_h$ and defined as follows. Job J_h executes for its C(HI). For any other HI job, if it terminates in the LO basic scenario schedule $\mathcal{S}^{LO}$ before J_h terminates, then it executes for its C(LO) else it executes for its C(HI). The schedule for $HI\text{-}J_h$ is denoted by $\mathcal{S}^{HI\text{-}J_h}$.

In multiprocessor scheduling, for a given job J_h multiple jobs may also terminate *exactly* at the same time in $\mathcal{S}^{LO}$ as J_h, and they are conservatively assumed to also execute for their C(HI) in $HI\text{-}J_h$.

Definition 5. [Canonical Basic Set] It is the set that contains the LO basic scenario and the job-specific basic scenarios for all HI jobs of the given instance.

Note that $\mathcal{S}^{HI\text{-}J_h}$ coincides with $\mathcal{S}^{LO}$ up to the time when job J_h switches, and after the switching time it starts using HI execution times for the jobs that did not terminate before the switch.

Example 4. Figure 3 shows Gantt charts for the job-specific scenarios of the single-processor problem instance given in the table below:

Job	A	D	χ	C(LO)	C(HI)
1	0	30	HI	10	12
2	2	10	HI	2	8
3	1	8	LO	2	2
4	8	17	HI	2	7
5	7	11	LO	2	2

If we look at the termination times of HI jobs in $\mathcal{S}^{LO}$ (the schedule for the LO basic scenario) we see that J_2 finishes first followed by J_4 then J_1. Thus in scenario $HI\text{-}J_4$, J_2 will execute for its C(LO) since it

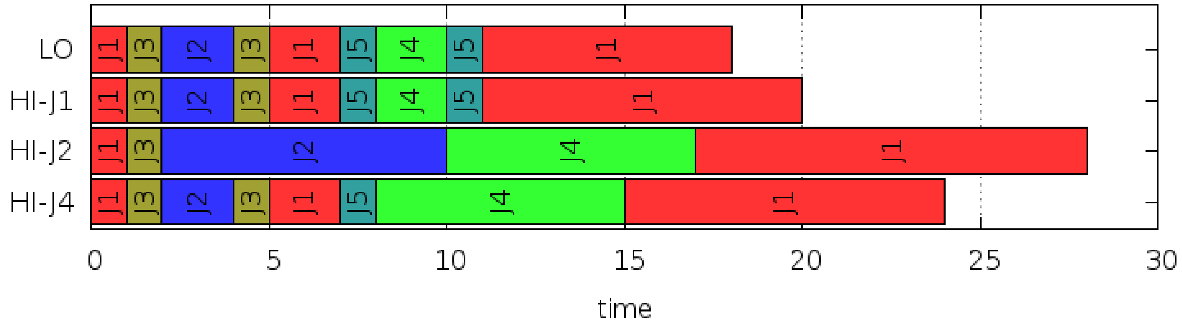

Figure 3: The job-specific scenario schedules for Example 4 obtained with priority table $PT = (J2, J4, J3, J5, J1)$

terminates before J_4 in $\mathcal{S}^{LO}$ and the rest of the jobs will execute for their $C(\text{HI})$ resulting in the schedule shown for HI-J_4 in Figure 3. The rest of the job-specific schedules are generated in the same manner.

Theorem 2. [Canonical Correctness Test] To ensure correctness of a scheduling policy that is weakly-predictable it is enough to test it for the canonical basic set.

Proof. Consider any basic scenario c and simulate the policy until either the mode switch, if any, or the end of the schedule. Let J_h be the job that switches. After the switch, increasing the job execution times can lead only to non-decreasing termination times, therefore we can conservatively replace c by HI-J_h. Hence, the policy is basically correct, and, by Lemma 2, also (completely) correct.

□

Unfortunately, since by Lemma 1 FPM is not weakly-predictable in multiprocessor case and the canonical correctness test might not be sufficient under such general conditions, in this case by Corollary 2, we may need the FPM policy to have FPM-equivalent tables to obtain weak-predictability and use the correctness test.

4.3 Building the Case for Class NP for FPM

The *canonical correctness test* algorithm was directly derived from the correctness test procedure described in [3], used in an attempt to prove that MC-scheduling is in class NP. Note that their algorithm is more complex and more general, as it applies to a number criticality levels more than two. Though that procedure, for efficiency reasons, would organize the schedules of basic scenarios in a tree structure and use backtracking, our less efficient formulation has only polynomially higher complexity, which does not impact on the reasoning on NP complexity. We now reapply the line of reasoning of [3] to prove that FPM is in class NP.

Lemma 4 (From [3])**.** If an instance is MC-schedulable, then there exists an optimal online scheduling policy that preempts each job j only at time points t such that at time t either some other job is released, or j has executed for exactly $C_j(i)$ units of time for some $1 \leq i \leq L$.

Lemma 4 is taken from the work in [3]. Although in our previous work [8], this lemma was refuted as it does not hold for all MC polices, it is true by construction in the case of FPM policies. We use this lemma in the proof of Theorem 3. In the lemma $C_j(i)$ is the WCET estimate for job j at criticality level i and L is the number of criticality levels in the system. In our usual notations, level 1 is LO, level 2 is HI, $C_j(1)$ is $C_j(\text{LO})$, $C_j(2)$ is $C_j(\text{HI})$.

Theorem 3. Dual-criticality single processor FPM policies are in class NP.

Proof. It follows from the weak-predictability of single processor FPM polices and Theorem 2 that we can check for correctness by simulating a polynomial number of scenarios (the canonical basic set). By Lemma 4 the cost of simulating each scenario is also polynomial. Therefore the canonical correctness test has polynomial complexity when testing FPM solutions. □

Theorem 4. Under the restriction of having FPM-equivalent tables, dual-criticality single- and multi-processor FPM policies are in class NP.

Proof. Follows directly from Corollary 2 and Theorem 2 □

Note that unlike the case of Theorem 3, the case described in Theorem 4 is not known to be NP hard. We have established the upper bound as NP, but the lower bound in this case is an open problem, it may be either NP-hard or P, and it may differ for single- and multi-processor cases.

5 Conclusions

Sustainability has been studied and well formalized in the literature [1] [10] for both the single and mixed-critical scheduling policies. Although sustainability implicitly implied predictability in single criticality systems, we have given an example of an MC policy that was proven to be MC-sustainable yet is not predictable. We have shown that testing for correctness using simulation based tests can be problematic if a scheduling policy is not predictable.

Acknowledging the difficulty of proving an MC-scheduling policy predictable and the difficulty of testing by simulation of worst case execution scenarios in non-predictable policies, we proposed a weaker form of predictability that covers a larger class of scheduling policies. We proposed a canonical correctness test that required the policy to be only weakly predictable. The correctness test verifies the correctness of the policy by evaluating it for a small number of basic scenarios. If successful the test ensures that the problem instance will remain correctly schedulable by the policy for any scenario that behaves better than the anticipated worst case estimations.

We have also shown that the well known class of FPM scheduling policies in dual-criticality, single processor case is not predictable but weakly predictable. We proved that the canonical test can be applicable in such cases and we studying the computational complexity showing that this special class belongs to NP. We showed that in the case of FPM policy having FPM-equivalent tables these results can be extended to the multiprocessor case.

References

[1] Sanjoy K. Baruah and Alan Burns. Sustainable scheduling analysis. In *Real-Time Systems Symposium (RTSS 2006)*, pages 159–168, 2006. 1, 3, 5

[2] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium*, RTSS'07, pages 239–243. IEEE, 2007. 1, 2

[3] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Comput.*, 61(8):1140 –1152, aug. 2012. 1, 2, 3, 4.1, 4.3, 4, 4.3

[4] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pages 145–154. IEEE, 2012. 1, 4

[5] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Euromicro Conf. on Real-Time Systems*, ECRTS'13, pages 93–102. IEEE, 2013. 1

[6] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. In *IEEE ISORC 2015*, 2015. 1, 4

[7] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler on single-and multi-processor platforms. In *17th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2015. 1

[8] Rany Kahil, Peter Poplavko, Dario Socci, and Saddek Bensalem. Revisiting the computational complexity of mixed-critical scheduling. In *Proc. Workshop on Mixed Criticality (WMC)*, 2017. 1, 4.3

[9] Ismael Ripoll and Rafael Ballester-Ripoll. Period selection for minimal hyperperiod in periodic task systems. *IEEE Transactions on Computers*, 62(9):1813–1822, 2013. 2

[10] Zhishan Guo, Sai Sruti, Bryan C Ward, and Sanjoy K Baruah. Sustainability in mixed-criticality scheduling. In *RTSS'17*. 3, 3, 3.1, 5

[11] Rhan Ha and J. W S Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. Int. Conf. Distributed Computing Systems*, pages 162–171, Jun 1994. 3.2