Revisiting the Computational Complexity of Mixed-Critical Scheduling

Rany Kahil[1], Peter Poplavko[2], Dario Socci[2], Saddek Bensalem[1]
1. Université Grenoble Alpes (UGA), VERIMAG, F-38000 Grenoble, France
2. Mentor® A Siemens Business, F-38334 Inovallée Montbonnot, France

***Abstract*— In this paper we revisit, refute, and give missing proofs for some previous results on mixed critical scheduling. We find a counter-example to the proof that this optimisation problem belongs to the class NP, which reopens the question on its computational complexity upper bound. Further, we consider a restricted problem formulation, the 'fixed priority per mode' (FPM) scheduling, to show that it is in NP. To do this, one has to demonstrate that FPM is sustainable. We prove (in the extended version, [1]) that is true in the single-processor case. We also show that FPM is not sustainable in the multiprocessor case.**

I. Introduction

When defining and solving optimisation problems care should be taken to provide a rigorous procedure to test the correctness of solutions. For mixed critical scheduling in general it has been believed proven in [2] that the testing can be done by a "canonical" algorithm in time polynomial on the number of jobs. In [2] this has served a proof that the computational complexity of this problem is *at most* of class NP (for a fixed number of criticality levels). By reduction from an NP-Hard problem to a mixed criticality problem, it was shown that the problem is also *at least* NP-Hard.

However, in this paper we present a counterexample that refutes the proof in [2] of polynomial complexity of the "canonical" algorithm. Consequently, the *upper bound* cannot be anymore considered proven, and the problem may, in fact have a complexity beyond the class NP. The question of complexity upper bound is thus re-opened.

We also have considered a restricted problem formulation where the polynomial complexity of the canonical algorithm is true by construction. In this case, the above mentioned refutation poses no problem to NP complexity claim. The considered restriction is well-known fixed-priority per mode (FPM) scheduling policy. For this problem the NP-Hard complexity lower bound from [2] still holds, and it would be useful to establish the class NP as an upper bound.

It turns out that there is another obstacle in doing it, unrelated to the refuted polynomial complexity proof. One can reapply the proof NP complexity given in [2] to a concrete scheduling policy only if that policy is *sustainable*, in a generalized mixed-critical sense. In the previous work, *e.g.,* [3], it was taken for granted that FPM is sustainable, but a closer study has revealed that it can only be the case for single processor but not multiple processors. Even for single processor case, currently we prove the result only for dual-critical instances. Therefore, so far only dual-critical single-processor FPM policy can be demonstrated to be in class NP, the other cases is an open problem.

II. Problem Formulation

Since our topic is problem complexity results, in this paper we will focus on simplest – dual-criticality – problem, as in this particular case it is the easiest to understand and revisit these results. The dual-criticality systems are systems that have only two levels of criticality, the high level, being denoted as 'HI', and the low (normal) level, denoted as 'LO'. Every job gets a pair of WCET values: the LO WCET and the HI WCET. The former one is for normal safety assurance, used to assess the sharing of processor with the LO jobs, and the other one, a higher value, is used to ensure certification.

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, $D_j \geq A_j$
- $\chi_j \in \{\text{LO}, \text{HI}\}$ 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 [2]: $C_j(\text{LO}) \leq C_j(\text{HI})$ The latter makes sense, since $C_j(\text{HI})$ is a more pessimistic estimation of the WCET than $C_j(\text{LO})$. We also assume that the LO jobs are forced to terminate after $C_j(\text{LO})$ time units of execution, so: $(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$.

An *instance* of the scheduling problem is a set of jobs $\mathbf{J}$. A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $c = (c_1, c_2, \dots, c_K)$, where K is the number of jobs. We only consider scenarios where no c_j exceeds $C_j(\text{HI})$. The *criticality of scenario* $c = (c_1, c_2, \dots, c_K)$ is LO if $c_j \leq C_j(\text{LO}), \quad \forall j \in [1, K]$, is HI otherwise. A scenario c is *basic* if:

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

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 $|\mathbf{J}'| \leq m$, 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 scheduled at any available processor at any time.

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 $\chi_{mode} = \text{LO}$ and switched to 'HI' as soon as a HI job exceeds $C_j(\text{LO})$. A scheduling policy is *correct* for the 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 be even dropped.

Based on the online state, a *scheduling policy* deterministically decides which ready jobs are scheduled at every time instant on m processors. A policy is said to be *work-conserving* if it never idles the processor if there is pending workload.

An instance $\mathbf{J}$ is *MC-schedulable* if there exists a correct scheduling policy for it.

One can restrict the general MC-schedulability problem to the problem of finding solutions for certain classes of scheduling policies or to the instances that have special properties. In this paper we show that the HI jobs with $C(\text{LO}) = C(\text{HI})$ create a certain complication for checking the correctness of MC-scheduling solutions, which can be easily avoided by incrementing their $C(\text{HI})$ by a small δC. Therefore the following optional restriction is worth considering.
Restriction (i) $\chi(J_i) = \text{HI} \implies C_i(\text{LO}) < C_i(\text{HI})$.

The dual-criticality MC-scheduling problem is NP-hard and is also claimed to be in class NP [2], but in this paper we refute their proof for the latter claim. One can restrict the MC-scheduling problem to finding optimal solutions for *fixed priority* (FP) and *fixed-priority-per-mode* (FPM) scheduling policies, which we define in a moment. The former restricted problem is in class P (polynomially solvable), the latter is NP-hard, just as the general problem [2], while we show in this paper that certain cases of this problem are in class NP.

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 per mode (FPM), a natural extension of fixed-priority for mixed critical systems. FPM is mode-switched policy with 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 priority table PT_{HI}. Suppose that after removing the LO jobs from PT_{LO} while keeping the same relative order of the HI jobs we obtain the PT_{HI} table. In this case one can just keep using the same priority table, PT_{LO}, after a switch to the HI mode with exactly the same result. Therefore in this particular case we say that we have *FPM-equivalent* tables: '$PT_{\text{LO}} \sim PT_{\text{HI}}$'. The below optional restriction of FPM scheduling problem allows to ensure certain useful properties:
Restriction (ii) Generate only solutions where: $PT_{\text{LO}} \sim PT_{\text{HI}}$

III. Correctness Test and Complexity

A. The Mixed Criticality Notion of Sustainability

To test the correctness of a scheduling policy one usually evaluates it for the scenario with maximal execution times for all jobs, which in our case corresponds to HI WCET's. However, to justify this test a scheduling policy must be *sustainable*, which means that increasing the execution time of any job A – while keeping all other execution times the same – may not make any other job B terminate earlier [4]. In other words, sustainability means that the termination times must be *monotonically non-decreasing* functions of execution times.

For mixed-critical scheduling the usual sustainability definition is too restrictive, as it does not take into account that an increase of an execution time of a HI job to a level that exceeds its LO WCET may lead to a mode switch and hence to dropping the LO jobs, which, in turn may lead to an earlier termination of another HI job, and hence non-monotonic dependency of termination times. Therefore, a weaker definition of sustainability is adopted for mixed criticality problems.

The new definition poses almost the same requirement of non-decreased termination time of any job B when we increase the execution time of a job A – while keeping all other execution times the same. However, now this property required to hold only when the increase of execution time of A leads neither to a change of the criticality mode in which B terminates nor to a switch of criticality mode by A. The second condition can only be violated if before the increase A executed for at most LO WCET and after the increase it exceeds the LO WCET. If at least one of these two conditions is violated then B may terminate earlier.

The adaptation of the notion of sustainability to mixed criticality raises the problem of how to adapt the policy correctness test to this new definition, as we cannot anymore rely on the traditional method of just testing the scheduling policy using just one maximal scenario: the "plain WCET".

B. Correctness Test and Computational Complexity

A general correctness test for a solution of a mixed-critical scheduling problem with a fixed set of jobs was systematically treated in [2], with the goal to study the *computational complexity* of the problem. The point is that the algorithmic complexity of the correctness test determines the complexity class of the problem. *If the correctness test is demonstrated to be simple enough so that it has at most polynomial complexity then the problem can be demonstrated to be in class NP.* To make the test algorithmically as simple as possible the test may "require" that a general-case solution be "*preprocessed*" into another solution such that the new solution is simpler to test. This permits to reduce the complexity of the subsequent test to the minimum, thus maximizing the chances that the test can be demonstrated polynomial and the problem NP. Note also that *the preprocessing should be applicable to any correct solution* in the set of solutions of the given problem and it should produce at the output a solution that is correct if the input solution is correct.

In [2] a correctness test is proposed that generalizes the ordinary plain-WCET scenario testing to testing a polynomial number of basic scenarios. For the test to be applicable to a given scheduling policy the minimal requirement is that it must be sustainable (in the sense we defined earlier). We come back to this test in a moment.

To build the argument that the problem is in NP, Lemmas 1 and 2 in [2], in fact, define two preprocessing steps:
Step (1) – we call it "*preprocessing for sustainability*" – ensures several useful properties of the output solution at the same time. Firstly, it ensures that (a) the output policy is sustainable (even if the input policy is not), (b) that the testing requires to construct only a polynomial number of schedules, and (c) that every event in a schedule (job arrival, preemption and termination) contributes only a polynomial time to the total cost of the test.
Step (2)– we call it "*preprocessing for polynomially-sized schedules*" is supposed to ensure that the schedules have polynomial (in fact, linear) size, *i.e.,* perform only polynomial number of preemptions.

In Section IV we will refute Step (2) proposed in [2], *i.e.,* their Lemma 2, which breaks their argument for mixed critical scheduling be in class NP (but not the argument that it is NP-hard). We also show in Section V that Step (1) as well can pose complications for the claim of NP complexity, for the case of FPM scheduling.

C. Canonical Algorithm for Correctness Testing

In this subsection we describe the correctness testing algorithm and Step (1).

Definition 1: An online scheduling policy is basically correct for instance **J** if for any basic scenario of **J** the policy generates a feasible schedule.

Lemma 1: If a scheduling policy is sustainable and Restriction (i) applies to the problem instance 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.

For proof see [1] Later on we show an single-processor example that violates Restriction (i) and that is only basically but not correctly schedulable by FPM with certain priority tables.

Lemma 2: An instance **J** is MC-schedulable if it admits a basically correct scheduling policy.

The above lemma is Lemma 1 from [2]. At the first glance it seems to be contradicting to a claim we have just made to show an FPM counterexample, but it should be noted that the lemma only claims that a correct policy exists, not that this policy is necessarily the same by which basically correct solution is constructed. In the proof given in [2] they show a

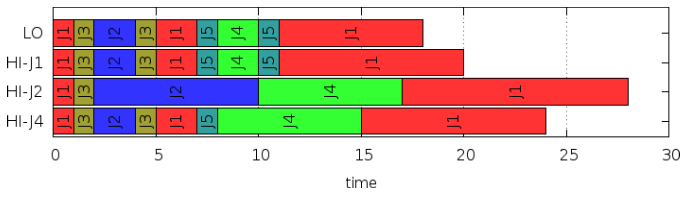

Fig. 1. The job-specific scenario schedules for Example 1 obtained with priority table $PT = (2, 4, 3, 5, 1)$

simple procedure to transform any basically correct policy into a similar policy that is, in addition, also sustainable, thus, by Lemma 1, yielding correct schedules in non-basic scenarios as well. This procedure corresponds to Step (1) of preprocessing for computational complexity, we come back to it in detail at the end of this subsection.

In fact, the above lemma implies that a complete correctness test can be reduced to testing all basic scenarios. However, this could not yield a polynomial testing algorithm, as there are exponential number of basic scenarios.

Fortunately, testing in all basic scenarios is redundant. Suppose that we have a sustainable 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.

Consider a LO basic scenario schedule $\mathcal{S}^{LO}$ and select an arbitrary HI job J_h. Let us modify this schedule by assuming that at time t_h when job J_h reaches its LO WCET (C_h(LO)) it has not yet signalled its termination, thus provoking a mode switch. Then, by Condition 2, we should ensure that J_h and all the other HI jobs that did not terminate strictly before time t_h will meet their deadlines even when continuing to execute until their maximal execution time – the HI WCET. Note that in multiprocessor scheduling multiple jobs may also terminate *exactly* at time t_h in $\mathcal{S}^{LO}$, and they are conservatively assumed to also continue their execution after time t_h in the modified schedule. The behavior described above is formalized to a basic scenario where all HI jobs that execute after time t_h have HI WCET.

Definition 2: For a given problem instance, LO basic-scenario schedule S^{LO} and HI job J_h, the basic scenario defined above is called 'specific' for job J_h and is denoted $HI\text{-}J_h$, whereas its schedule is denoted $\mathcal{S}^{HI\text{-}J_h}$.

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 1: Fig. 1 shows Gantt charts for the job-specific scenarios of the following single-processor problem instance:

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

We see, for example that in the LO scenario job J_2 terminates at time 4, but in the $HI\text{-}J2$ scenario job J_2 switches at time 4 and continues to execute, because, apparently, it has a HI WCET larger than the LO WCET. In fact, these schedules are obtained from FPM policy and demonstrate that this policy is correct for the given problem instance, as explained later in Example 3.

Theorem 1: Under Restriction (i), to ensure correctness of a scheduling policy that is sustainable (in the mixed criticality sense) it is enough to test it for the LO scenario and the scenarios HI-J_h of all HI jobs J_h.

For proof see [1]

The above theorem, in fact, defines – for dual-criticality case – what we call *canonical correctness test* algorithm. It can be directly derived from the correctness test procedure described in [2], which is, however, more complex and more general, as it applies 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.

Step (1) formulation. To prove NP complexity along the lines of reasoning given in [2] one has to demonstrate that any correct scheduling policy can be transformed into another one that is sustainable and basically correct and then apply Lemmas 1,2. For this, [2] specifies a procedure for this transformation, which can be reformulated as follows. As the output policy we use a mode-switched time-triggered policy which we call *Static Time-Triggered per Basic Scenario* (STTBS). This policy specifies a static time-triggered table for the LO scenario and all job-specific scenarios. The Gantt charts in Figure 1, in fact, specify such tables for the given example. The total length of all slots attributed to a given job in a given table should be at least equal to the job's execution time in the given basic scenario. The tables are obtained by simulation of the input policy in the given scenario. The execution of STTBS policy starts in the LO static table. If a job finishes earlier than the allocated time, the processors are idled in the remaining slots of that job. A job is allowed to continue execution for longer than its LO WCET, in that case the STTBS policy switches to the static table specific for that job. One can show that one can bypass Restriction (i) requirement of Lemma 1 and Theorem 1 when they are applied to STTBS solutions.

It should be noted that STTBS policy is radically different from what is usually referred to as "time-triggered scheduling" in mixed critical systems, which is *static time triggered table per mode* (STTM) [5], where there are only two time-triggered tables: one per mode. We will see an example of that policy in the next section.

Clearly, if we present as input to the canonical correctness testing algorithm the STTBS policy with $H + 1$ static time-triggered tables then the properties (a),(b) and (c) that should be guaranteed by Step (1) are, in fact guaranteed. In particular, the simulation of one preemption takes a polynomial time, because the job that preempts another job is pre-specified in the STTBS table.

In fact, what remains to be shown for proving the NP upper bound on computational complexity is that the tested solution can, in addition, be presented in the form where only a polynomial number of preemptions occur in each job-specific scenario, this is what we call Step (2). Unfortunately, we have to observe that Step (2) proposed in [2] does not work in general case and hence is incorrect. We demonstrate it by a counterexample in the next section.

IV. Refuting the Proof of Polynomial Size

In this section we refute a lemma given [2]. This lemma was used as a cornerstone to prove that the canonical correctness test algorithm, when applied in general case, can have polynomial complexity, because the schedules can be restricted to have only a polynomial number of preemptions.

The lemma is copied below for convenience.

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(LO), $C_j(2)$ is C_j(HI).

Lemma 3 (Refuted Lemma): 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 $1 \leq i \leq L$.

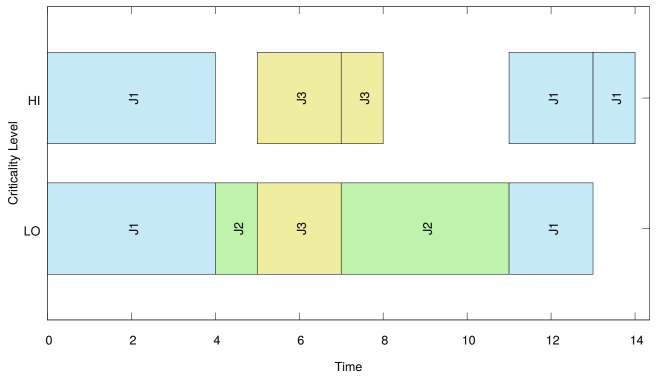

Fig. 2. A Valid Scheduling Policy for the Instance in Example 2

Example 2: Consider the following problem instance:

Job	A	D	χ	$C(1)$	$C(2)$
1	0	14	HI	6	7
2	0	11	LO	5	5
3	5	10	HI	2	3

Let us check if it is MC-schedulable according to Lemma 3. At $t = 0$ we can execute either job J_1 or job J_2, whichever job we choose it should not be preempted before $t = 5$.

1) If job J_1 is to be executed in the time interval $[0, 5)$, then in the interval $[5, 11)$, which is 6 time units, we will have to execute jobs J_2 and J_3 which combined need 7=$(5 + 2)$ units of execution in the LO scenario. Thus we cannot execute J_1 in $[0, 5)$.
2) Suppose that we execute job J_2 in $[0, 5)$. What is then left to execute is the two high criticality jobs. We take the scenario that they both execute for their C(HI). Then we need a total of 10=$(7+3)$ units in the execution window $[5, 14)$, which has space for only 9 units.

Thus, according to Lemma 3 this instance is not MC-schedulable.

Figure 2 shows a Gantt chart representing an STTM scheduling policy [5] that correctly schedules that instance, contradicting Lemma 3. This policy starts execution in static table 'LO' and keeps using this table as long as there is no switch to the HI criticality mode $\chi = HI$, in which case it switches to static table 'HI'. This example shows that an instance can be MC-schedulable but no optimal online scheduling policy exists that preempts a job j only at time points where another job is released or j has executed for exactly $C_j(i)$ units.

V. Rehabilitation for Fixed Priority per Mode

In this section, for dual-criticality case, we "rehabilitate" the NP complexity argument established in [2] for the case of FPM – fixed priority per mode – policy. This is important, because this policy is popular in the literature, see *e.g.*, EDF-VD [6]. The "NP-Hard" classification as a "lower bound" on complexity, established in [2], remains valid when we restrict ourselves to FPM, therefore it is important to establish "NP" classification as an "upper bound". The refutation of Lemma 3 is not a problem for FPM, because this policy satisfies the statement of that lemma by construction.

However when we go from general-case MC-scheduling problem to a particular case of FPM scheduling problem and want to prove that it is in class NP, now Step (1) described in Section III encounters an obstacle. The solutions presented to the correctness test algorithm *should belong to the set of solutions of the problem for which we prove that it belongs to NP*. In the general problem formulation, the set of solutions includes all possible policies, and therefore presenting STTBS policies at the input of the correctness test was legal. Unlike the general MC-scheduling case, discussed in Section III, the set of solutions of the FPM scheduling problem consists *exclusively* of applications of FPM policy with different priority tables. Therefore, to prove that FPM is in NP we have to present FPM policies at the input of the correctness test algorithm, so Step (1) cannot be applied.

Therefore, we investigate whether Step (1) can be skipped and whether the canonical test algorithm can be applied directly to FPM policy. For this, the FPM policy should guarantee all the properties (a), (b), (c) ensured by Step (1). In fact, only property (a) – "*sustainability*" (in mixed-criticality sense) is not trivial and needs investigation and proof. Therefore, *we focus on the conditions under which FPM is sustainable*. Under these conditions it is also in class NP. We will only focus on dual-criticality case, leaving generalisation to more levels of criticality to future work.

The following theorem from [7] states a very useful property, for which we formulate a corollary:

Theorem 2: Fixed-priority policy is sustainable (in the default strict sense), for single- and multi-processor scheduling.

Corollary 3: For *single-processor* dual-criticality instances FPM is sustainable (in the mixed criticality sense). If, in addition, Restriction (i) is satisfied then the canonical correctness test is applicable to test FPM correctness for such instances. However, when Restriction (ii) is satisfied then Restriction (i) is not necessary and the scope of dual-critical FPM sustainability extends from single-processor to multiprocessor instances, whereby the canonical test is applicable as well.

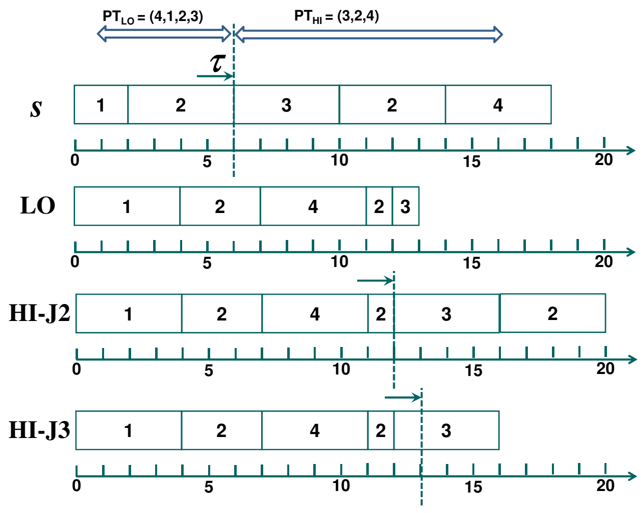

Fig. 3. Gantt charts (indices 1,2,…identify J_1, J_2, …) of scenario $s = (c_1 = 1, c_2 = 8, c_3 = 4, c_4 = 4)$ in Example 4. Mode switch time of scenario s is $\tau = 6$. Also the LO and job-specific basic scenarios are shown. Note that job HI-J_4 is not given because J_4 cannot switch. Job J_4 misses deadline, but this is not captured in any basic scenario. This is possible because Restriction (i) requirement of Corollary 3 is not respected.

For proof see [1]. The corollary implies that under the specified conditions the FPM scheduling is in the class NP.

Note that Restriction (i) is quite general, as it can be ensured by an arbitrarily small increase of C_{HI} if $C_{\text{HI}} = C_{\text{LO}}$. Unfortunately, the sustainability of FPM cannot be asserted for multiprocessor case such general conditions, in this case we can only propose a quite restrictive Restriction (ii).

Example 3: Consider single-processor independent-job problem instance **J** defined in Example 1. For a certain priority table $PT_{\text{LO}} = PT_{\text{HI}}$, the Gantt chart in Figure 1 shows the execution of FPM policy on single processor in all scenarios required by the canonical correctness test. In those scenarios all jobs meet their deadlines. Since for this instance Restriction (ii) holds, FPM is sustainable and the canonical test is indeed applicable. Therefore the FPM policy with given priority table is correct for the given problem instance.

Example 4: To illustrate that Restriction (i) may be necessary let us consider the following single-processor problem instance **J**:

Job	A	D	χ	C(LO)	C(HI)
1	0	20	LO	4	4
2	0	20	HI	4	8
3	0	20	HI	1	4
4	7	11	HI	4	4

This problem instance violates Restriction (i) and hence it is not guaranteed that the canonical correctness test is applicable to FPM solutions directly. Figure 3 shows the Gantt chart of FPM policy for a specified non-basic scenario s and specified priority tables. In scenario s job J_4 has a deadline miss, but this is not visible to the canonical test, which checks in the LO and HI-job specific scenarios, whose Gantt charts are also shown. Note that job J_4 has $C(\text{LO}) = C(\text{HI})$, so it cannot switch and therefore we do not include the respective Gantt chart for HI-J_4, but even if we included that scenario it would not reveal the deadline miss either.

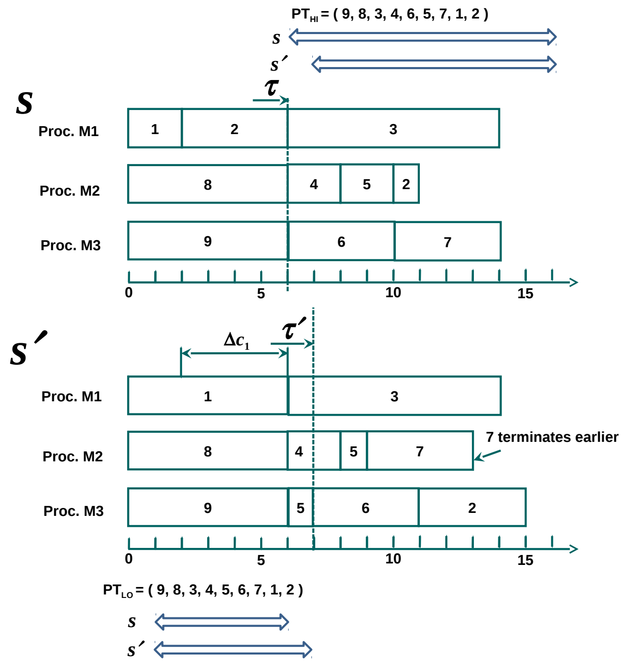

Fig. 4. FPM non-sustainability demonstration on multiprocessor case, using Example 5. Gantt charts (indices 1,2,... identify J_1, J_2, ...) of two scenarios: s and s'. Mode switch times are τ and τ', resp. Scenario s is defined by $(c_1 = 2, c_2 = 5, c_3 = 8, c_4 = 2, c_5 = 2, c_6 = 4, c_7 = 4, c_8 = c_9 = 6)$. Scenario s' differs from s by $c'_1 = c_1 + \Delta c_1 = 2 + 4$. Job J_7 violates sustainability by terminating in s' earlier than in s, while terminating in the same mode (HI) in both scenarios.

VI. Complication for Multiprocessor Case

In this section we give a counterexample for sustainability of FPM on multiple processors and discuss the consequences.

Example 5: Consider the following problem 3-processor problem instance **J** be defined by:

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 4 shows execution in two scenarios: s and s' for the priority tables specified in the figure, whereby Restriction (ii) $PT_{\text{LO}} \sim PT_{\text{HI}}$ is not satisfied and hence sustainability is not guaranteed. Scenario s' differs from scenario s only by a larger execution time of J_1. Nevertheless we see that job J_7 (as well as J_5) terminates in scenario s' earlier than in scenario s. This behavior contradicts the requirements of sustainability.

Note that in scenario s jobs J_5 and J_7 miss their deadlines, whereas in s' they do not. If correction test algorithm were used for this case, it would not check for scenario s, because it is not basic. It would check in s', which is HI-J_4, in the other job-specific scenarios and in the LO scenario. Since in these scenarios the FPM policy would not miss the deadlines, it would come to conclusion that the proposed priority tables are correct, whereas, as we see this is not true. The test algorithm would come to a wrong conclusion because the condition on sustainability of the policy is not satisfied.

Unlike Example 4, 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.

From the above it follows that FPM cannot be shown to be in NP or PSPACE for multiprocessors by following the same line of reasoning as proposed in [2].

VII. Conclusions

In this paper we have reconsidered the results concerning the computational complexity of mixed critical scheduling of a fixed set of jobs. We have refuted the proof that mixed critical schedules can be restricted to have size polynomial on the number of jobs without loosing optimality. This can mean that the problem may have a complexity beyond NP and PSPACE.

We have also studied the computational complexity of of the special case of fixed-priority per mode scheduling. We have discovered that this problem can be shown in class NP only in the single-processor case, since this policy turned out to be non-sustainable in multiprocessor case. In the extended version [1], we also give the sustainability proof for single-processor case.

References

[1] R. Kahil, P. Poplavko, D. Socci, and S. Bensalem, "Revisiting the computational complexity of mixed critical scheduling," Tech. Rep. TR-2017-7, Verimag Research Report, 2017. http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2017-7.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Trans. Comput.*, vol. 61, pp. 1140 –1152, aug. 2012.

[3] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Multiprocessor scheduling of precedence-constrained mixed-critical jobs," in *IEEE ISORC 2015*, 2015.

[4] S. K. Baruah and A. Burns, "Sustainable scheduling analysis," in *Real-Time Systems Symposium (RTSS 2006)*, pp. 159–168, 2006.

[5] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 3–12, 2011.

[6] 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, pp. 145–154, IEEE, 2012.

[7] R. Ha and J. W. S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proc. Int. Conf. Distributed Computing Systems*, pp. 162–171, Jun 1994.