Revisiting the Computational Complexity of Mixed-Critical Scheduling

Rany Kahil, Peter Poplavko, Dario Socci, Saddek Bensalem

Verimag Research Report n^o TR-2017-7

October 1, 2017
last updated: November 8, 2017

Revisiting the Computational Complexity of Mixed-Critical Scheduling

Rany Kahil, Peter Poplavko, Dario Socci, Saddek Bensalem

October 1, 2017
last updated: November 8, 2017

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 that is true in the single-processor case. We also show that FPM is not sustainable in the multiprocessor case.

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

Reviewers: Saddek Bensalem

Notes: Extended version of a paper at 5th International Workshop on Mixed Criticality Systems (WMC'17) at the Real Time Systems Symposium (RTSS 2017)

1 Introduction

When defining and solving optimisation problems care should be taken to provide a rigorous procedure to test the correctness of solutions. The testing procedure has impact on the problem complexity class.

For mixed critical scheduling in general it has been believed proven in [1] that the testing can be done by a "canonical" algorithm in time polynomial on the number of jobs. In [1] 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 [1] 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 [1] 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 [1] to a concrete scheduling policy only if that policy is *sustainable*, in a generalized mixed-critical sense. In the previous work, *e.g.,* [2], 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.

2 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 [1]: $C_j(\text{LO}) \leq C_j(\text{HI})$ 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: $c = (c_1, c_2, \ldots, c_K)$, where K is the number of jobs. We only consider scenarios where no

c_j exceeds $C_j(\text{HI})$. The *criticality of scenario* $c = (c_1, c_2, \ldots, c_K)$ is LO if $c_j \le C_j(\text{LO}),\ \forall j \in [1, K]$, is HI otherwise. A scenario c is *basic* if:

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

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 $|\mathbf{J}'| \le m$, 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 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 $C_j(\text{LO})$. It should be noted that in a given schedule it is the first job that exceeds its $C(\text{LO})$ that *switches* the mode and then the mode remains HI until the end of the schedule. 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.

2.1 Generalized and Restricted Formulations

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. One can also generalize it to certain scheduling behaviors.

Definition 1 (Artifact behavior)**.** *Suppose a HI job has $C(LO)$=$C(HI)$. By default the MC scheduling does not permit such jobs to switch the χ_{mode} from LO to HI because they cannot execute for strictly more than $C(LO)$. If we generalize the problem formulation so that when such jobs execute for $C(LO)$ then this may be interpreted as if they slightly exceeded $C(LO)$ thus entailing a mode switch. This generalized problem formulation is called artifact behavior.*

The mixed criticality theory usually assumes that the mode switch occurs *immediately* after the moment when the job that causes the switch has executed for $C(\text{LO})$ time. No finite threshold for exceeding $C(\text{LO})$ is defined in the theory. Unlike LO jobs, the HI jobs are not forced to stop if they try to exceed their worst-case execution time. Therefore, considering artifact behavior can make sense for robustness reasons, and *apriori* forbidding it is not an obvious choice. We assume artifact behavior is allowed, but the goal is to demonstrate that the jobs with $C(\text{LO}) = C(\text{HI})$ can have negative impact on sustainability of scheduling. In this paper we also show that even if artifact behavior is not allowed 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 [1], 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 [1], 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 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), 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: $PT_{\text{LO}} \sim PT_{\text{HI}}$

3 Correctness Test and Complexity

3.1 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 [3]. 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. If at least one of these two conditions is violated then B may terminate earlier. 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[1]. Note that in this situation the second condition is violated only if after the increase of execution time of A it is A that causes the schedule to switch the mode.

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".

3.2 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 [1], 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*

[1]If artifact behavior is allowed, and job A has $C(\text{LO})=C(\text{HI})$ and after the increase the execution time of A reaches $C(\text{LO})$ then the second condition can also be violated.

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 , thus giving useful indication of the limits of the complexity of the problem. 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 [1] 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 [1], 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 4 we will refute Step (2) proposed in [1], *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). In the later sections we show that fortunately FPM policies do not need the application of Step (2), as they already possess the property ensured by that step. This makes it possible to "rehabilitate" the line of reasoning of [1] for showing that the problem is in class NP for the case of FPM policies. We also show in Section 5 that Step (1) as well can pose complications for the claim of NP complexity, for the case of FPM scheduling.

3.3 Canonical Algorithm for Correctness Testing

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

Definition 2. *[Basically Correct Policy] 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. *[Correctness Test by Checking all Basic Scenarios] 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.*

Proof. For a given scheduling policy, let us call basic scenario $\lceil s \rceil$ the ceiling scenario of scenario s if in $\lceil s \rceil$ each J_i executes for time $C_i\,(\chi_{\mathrm{TERM}}(s,i))$, where $\chi_{\mathrm{TERM}}(s,i)$ is the mode in which job J_i terminates in scenario s. It is obvious that in $\lceil s \rceil$ all the jobs have at least the same or higher execution time and they terminate in the same or higher-criticality mode. For dual-criticality instances this implies that the jobs with $\chi_{\mathrm{TERM}}(s,i)$=HI terminate in the HI mode also in $\lceil s \rceil$. By the definition of sustainability, these jobs cannot terminate in scenario $\lceil s \rceil$ earlier than in s. It remains to consider the jobs that terminate in LO mode in s. Obviously in the LO basic scenario these jobs cannot terminate earlier. Therefore the jobs of scenario s are 'covered' by one of the two basic scenarios: $\lceil s \rceil$ and LO, in the sense that meeting the deadlines in those scenarios implies meeting deadlines in scenario s. □

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

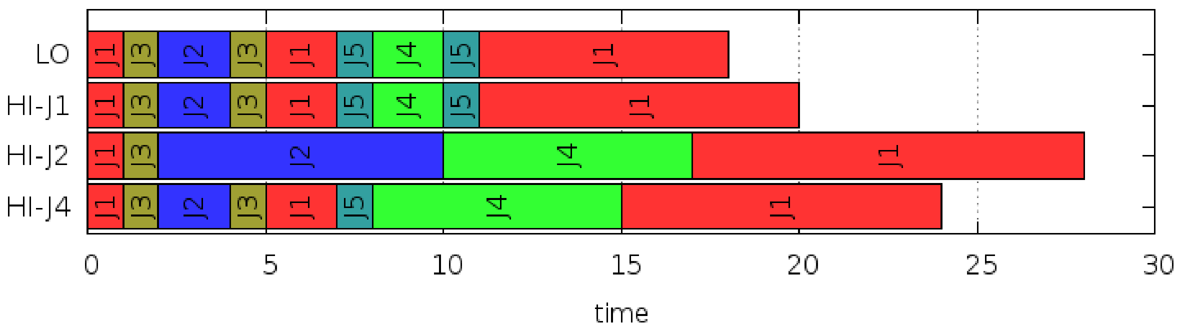

Figure 1: The job-specific scenario schedules for Example 1 obtained with priority table $PT = (2, 4, 3, 5, 1)$

The above lemma is Lemma 1 from [1]. 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 [1] they show a 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 3. *[Job-specific Basic Scenario] 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. Note also that if artifact behavior is not allowed then we should exclude from the job-specific scenarios the HI jobs for which C(LO)=C(HI), as it is not possible that such a job would 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. *[(Canonical) Correctness Test by Checking Job-specific Scenarios] 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.*

Proof. Consider any basic scenario s and simulate the policy until the first job, if any, switches. Let J_h be the job that is the first to switch. After the switch, increasing the job execution times can lead only to non-decreasing termination times, therefore we can conservatively replace s by HI-J_h. Hence, the policy is basically correct, and, by Lemma 1, also (completely) correct.

□

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 [1], 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 [1] 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, [1] 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) [4], where there are only two time-triggered tables: one per mode. We will see an example of that policy in the next section. It has been proved that for dual-critical systems one can "transform" any correct policy to STTM policy in the case of single processor [5, 6], which allows to simplify the correctness testing even further. However, there is evidence that this would not work for multiprocessors [7, 5, 6]. Also, for multiple levels of criticality there are no general results yet. Therefore, in general case one has to resort to STTBS.

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 [1] does not work in general case and hence is incorrect. We demonstrate it by a counterexample in the next section.

4 Refuting the Proof of Polynomial Schedule Size

In this section we refute a lemma given [1]. 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 $C_j(\text{LO})$, $C_j(2)$ is $C_j(\text{HI})$.

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 $1 \le i \le L$.*

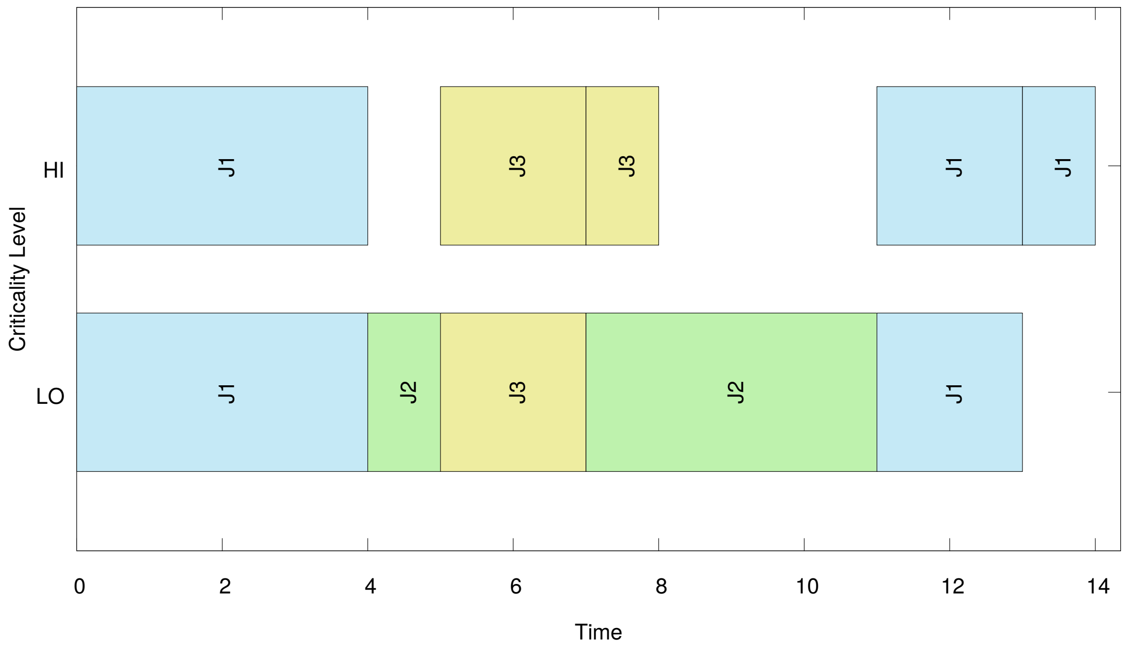

Figure 2: A Valid Scheduling Policy for the Instance in Example 2

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 $C(\text{HI})$. 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 [4] 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.

Lemma 3 was used in the proof of the following theorems:

- **Theorem** The problem of deciding MC-schedulability for L criticality levels is in NP when L is a constant.
- **Theorem** The problem of deciding MC-schedulability is in $PSPACE$.

Now whether these theorems are true becomes an open problem.

5 Rehabilitation for Fixed Priority per Mode

In this section, for dual-criticality case, we "rehabilitate" the NP complexity argument established in [1] 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 [8]. The "NP-Hard" classification as a "lower bound" on complexity, established in [1], 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 3 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 3, 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. If we applied Step (1) transformation to an FPM policy solution then we would obtain a solution for an STTBS policy. Therefore we would find ourselves proving the complexity class NP for the problem of finding optimal static tables for STTBS policy under the condition that these tables should by simulation of FPM policy. This still would have practical interest, because there exist quite efficient FPM heuristics for single and multiprocessor case [9, 6] and hence they can provide efficient time triggered tables. Still, like this we would prove NP complexity under an "unfair" problem formulation. Instead, we prefer to show that FPM problem itself belongs to class NP "fairly", *i.e.,* in the usual sense.

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 [10] 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 dual-criticality instances, under Restriction (i) the FPM policy is sustainable (in the mixed criticality sense) for single-processor problem instances.*

However, when Restriction (ii) is applied then Restriction (i) is not necessary. In this case, for dual-critical instances the FPM policy is sustainable (in the mixed-critical sense) both on single- and multiprocessor case.

We present the proof in the final part of this section, after some discussion and showing some examples. It can be also shown that if artifact behavior is not allowed then Restriction (i) can be removed from the Corollary (though, not from Theorem 1) and that, under Restriction (ii), not only FPM is sustainable, but also the correctness test algorithm is applicable to it even bypassing the Restriction (i) posed in Theorem 1. The corollary implies that under the specified conditions the FPM scheduling is in the class NP.

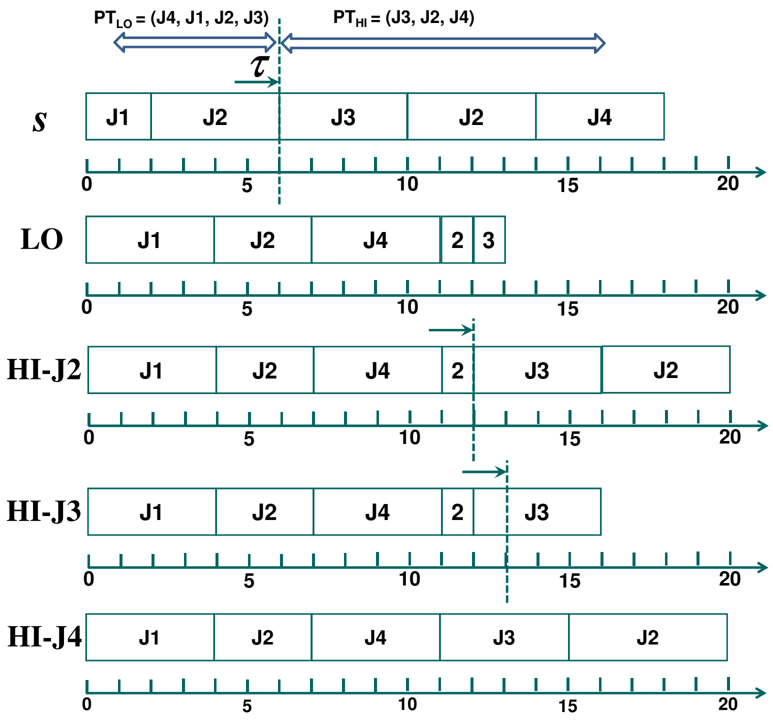

Figure 3: Gantt charts of scenario $s = (c_1 = 1, c_2 = 8, c_3 = 4, c_4 = 4)$ in Example 4. Mode switch time of scenario s is $\tau = 6$. Also the LO and job-specific basic scenarios are shown. Job J_4 misses deadline, but this is not captured in any basic scenario. This is possible because Restriction (i) requirement of Theorem 1 is not respected.

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 $PT_{LO} = PT_{HI}$, 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. Since we allow artifact behavior, job J_4 can switch the criticality mode despite the fact that $C(LO) = C(HI)$. That is why we include HI-J_4 into the set of scenarios checked by the canonical correctness test algorithm.

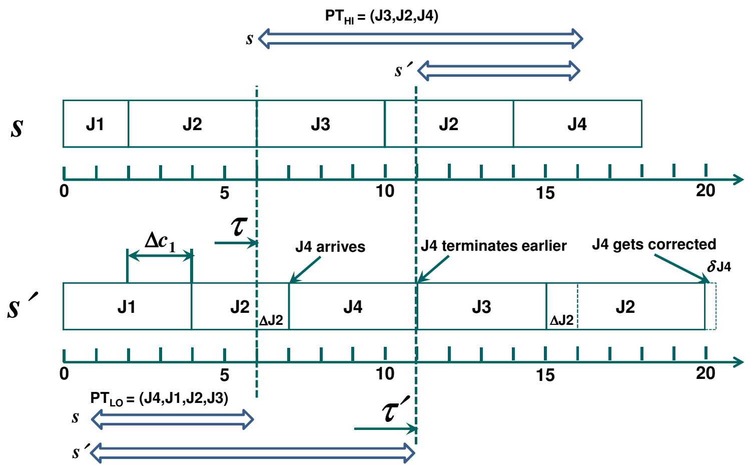

Figure 4: Demonstrating FPM non-sustainability in the case Restriction (i) is violated and artifact behaviour is allowed. We assume the problem instance of Example 4 and show the Gantt charts of two scenarios: s and s'. Mode switch times are τ and τ', resp. Scenario s is the same as in Figure 3 and scenario s' differs from s by $c'_1 = c_1 + \Delta c_1 = 2 + 2 = 4$. Job J_4 switches at time τ' and immediately terminates at that time, in HI mode (*i.e.,* artifact behavior). Since in these scenarios J_4 terminate in the same mode, sustainability requires that it cannot terminate in s' earlier, which is violated. If we eliminate the artefact behavior by adding δJ_4 to $C_4(\text{HI})$ job J_4 would get a 'corrected' termination time, respecting sustainability. If we instead restrict the problem and forbid artifact behavior then J_4 will terminate in a *different* mode in s' – in the LO mode, thus again respecting sustainability.

Example 5 (Non-sustainability of artifact behavior)**.** *Consider the same problem instance and the same priority tables as in the previous example. Figure 4 shows the FPM schedules in two scenarios: s and s'. The mode switch times in s and s' are denoted τ and τ', resp. Job J_4 violates Restriction (i), in scenario s' it switches into HI mode and immediately terminates, thus showing artifact behavior. In scenario s' job J_1 has an increased execution time* w.r.t. *scenario s, while all the other jobs have the same execution time. In both scenarios job J_4 terminates in the same mode – HI. Therefore we can apply definition of mixed criticality sustainability such that $A = J_1$ and $B = J_4$. We see that B terminates in scenario s' earlier, thus violating the sustainability.*

Nevertheless, if, to satisfy Restriction (i), we add a small increase of execution time δJ_4 then job J_4 will terminate in line with sustainability property ('gets corrected' in Fig. 4) and the deadline miss will get exposed to the canonical test, in scenario s'. Note that jobs J_2 and J_3 also terminate in scenario s' later, as required by sustainability.

For illustrating the proof of the corollary later on, let us note that the increase of execution time of J_1 in scenario s' has for the consequence that the schedule of scenario s' at time τ has two time units of more workload to execute, and this extra workload belongs to job J_2, which is illustrated in the figure by two single-unit intervals with notation ΔJ_2. Let us also note that other differences of scenario s' from s is that in s' it is job J_4 that switches, not J_2, and the switch happens later. During the window $[\tau, \tau']$ between the two switches two different priority tables act in s and s', PT_{HI} and PT_{LO}, respectively. This can lead to undesirable non-sustainability effect, but the corollary assures that this will not happen if Restriction (i) or (ii) is satisfied.

An important observation about FPM policy is that, just like FP policy, it is work-conserving. For analyzing the properties of such a policy it is useful to introduce the notion of *interference*. The interference that job B experiences from A is the total time during which A has been running while B has been ready but not running because all the processors are busy. In a work-conserving policy, the more the total interference a job experiences from the other jobs the later it terminates. On a single processor, if all jobs that interfere with a job B terminate earlier than B then the total interference they exercise on that job reaches the maximum. This observation leads to the following lemma, stated without proof.

Lemma 4 (Interference and Later Termination on a Single Processor)**.** *Consider two work-conserving single-processor schedules for the same scenario: $\mathcal{S}$ and $\mathcal{S}'$. In $\mathcal{S}'$ a job B terminates at the same time or later than in $\mathcal{S}$ if has the following properties: (1) in $\mathcal{S}$ all jobs that do not terminate before it have zero interference with it, while (2) in schedule $\mathcal{S}'$ the set of jobs that terminate before B is at least the same but may be larger than in $\mathcal{S}$. Formally:*

$$\mathbf{J} \;=\; \mathbf{J}^{BF}(B,\mathcal{S}) \cup \mathbf{J}^{ZR}(B,\mathcal{S}) \;=\; \mathbf{J}^{BF}(B,\mathcal{S}') \cup \mathbf{J}^{ZR}(B,\mathcal{S}') \cup \mathbf{J}^{OT}(B,\mathcal{S}')$$

and:

$$\mathbf{J}^{BF}(B,\mathcal{S}) \subseteq \mathbf{J}^{BF}(B,\mathcal{S}')$$

*where $\mathbf{J}$ is the set of all jobs, $\mathbf{J}^{BF}$ is a subset of jobs that terminate before given job B in given schedule, $\mathbf{J}^{ZR}$ is a subset of jobs that exercise zero interference on B (*i.e.*, do not interfere with it) and $\mathbf{J}^{OT}$ is a subset of 'other' jobs, of which we do not specify which interference they exercise or when they terminate.*

We use the above lemma in the proof of the corollary.

Proof. (of Corollary 3). Consider an FPM policy with given priority tables PT_{LO} and PT_{HI}. Consider any scenario s. Let scenario s' differ from s only by an increase in execution time of job A, Δc_A, such that A executes entirely in the LO mode in both scenarios, which means that Δc_A is not large enough to lead to a switch of A. Let job B be an arbitrary job that terminates in both scenarios in the HI mode. We have to prove that B can only terminate at the same time or later in s' than in s, but never earlier. This is the only non-trivial case to prove; in all the other cases the job A either terminates either after B, thus not having any effect, or in the same mode as B, thus being under control of the fixed priority policy, which is sustainable.

Let τ, τ' be the switch times of s and s', we have $\tau < \tau'$. In the rest of the proof we only consider the job execution *after* time τ. Thus, for convenience, let us remove in both scenarios the jobs that terminate or get dropped by time τ and subtract the progress made before τ from the execution times of the remaining jobs.

In the modified model all jobs that execute in s are HI jobs, they execute under fixed-priority policy, with table PT_{HI}. All these jobs execute in scenario s' as well. Let us partition these jobs into two subsets: $\mathbf{J}^{\text{I}}$ and $\mathbf{J}^{\text{II}}$. Jobs $\mathbf{J}^{\text{I}}$ terminate in s' at or before the switch time τ'. Unlike s, in s' they are under control of PT_{LO}. Jobs $\mathbf{J}^{\text{II}}$ terminate in s' after τ', their termination is under control of PT_{HI} in both scenarios. For example, in Fig. 4 we have $\mathbf{J}^{\text{I}} = \{J_4\}$ (assuming $\delta J_4 = 0$) and $\mathbf{J}^{\text{II}} = \{J_2, J_3\}$.

Let us first admit the hypothesis that *no jobs arrive later than* τ' and prove the lemma for this case. Afterwards we will show that adding new jobs, arriving after time τ', will preserve the result for the present jobs and will satisfy the lemma for the new ones.

For convenience, let $\mathbf{J}^{\text{III}}$ be the subset of jobs in $\mathbf{J}^{\text{II}}$ that terminate after τ' not only in s' but also in s. Because, by current hypothesis, they arrive at τ' or earlier and terminate after τ', it should be the case that they terminate in the same order in s as in s', the higher PT_{HI} priority jobs terminating earlier than the lower-priority ones. In our example, $\mathbf{J}^{\text{III}} = \{J_2\}$.

In addition to all the subsets introduced so far, scenario s' executes certain jobs that are not executed in s. This is, firstly, the new workload that is pushed into the interval $[\tau, \infty)$ due to job A execution time increase Δc_A and, secondly, the LO jobs arriving after τ and not being dropped in s' until the mode switch at τ'. The latter jobs are denoted as subset $\mathbf{J}^{\text{IV}}$. In our example, $\mathbf{J}^{\text{IV}}$, somewhat liberally, consists of job ΔJ_2, *i.e.,* the part of job J_2 that "does not exist" in s[2].

Let Restriction (i) of the corollary hold. In that case the job that switches into HI mode in s', at time τ', does not *terminate* at that time. Therefore, by construction, job B can only belong to subset $\mathbf{J}^{\text{II}}$ and cannot be in the other subset, as by construction it should terminate in the HI mode, whereas with Restriction (i) this implies that it should terminate after τ'. In our example, Restriction (i) is satisfied if $\delta J_4 > 0$, which lets job J_4 move from subset $\mathbf{J}^{\text{I}}$ to $\mathbf{J}^{\text{II}}$. In this case we have: $\mathbf{J}^{\text{I}} = \{\}$, $\mathbf{J}^{\text{II}} = \{J_2, J_3, J_4\}$ and $\mathbf{J}^{\text{III}} = \{J_2, J_4\}$.

Now let us observe that if $B \notin \mathbf{J}^{\text{III}}$ then it is trivial to show that in s' it terminates later. Indeed, in this case in s' it terminates after τ', as being in $\mathbf{J}^{\text{II}}$, and in s it terminates at the latest at τ', as not being in $\mathbf{J}^{\text{III}}$. Therefore, it remains to study the case where $B \in \mathbf{J}^{\text{III}}$. In this case let us show that B satisfies Lemma 4 for the schedules of scenarios s and s'. For this let us classify all jobs between subsets $\mathbf{J}^{BF}$, $\mathbf{J}^{ZR}$, and $\mathbf{J}^{OT}$.

[2]in fact, removed from s when we modified the instance

1. Consider job J that is another job in $\mathbf{J}^{\text{III}}$. Since the jobs in $\mathbf{J}^{\text{III}}$ terminate in s and s' in the same order, determined by PT_{HI}, we have: if $J \succ B$ then $J \in \mathbf{J}^{BF}$ and if $J \prec B$ then $J \in \mathbf{J}^{ZR}$, and this does not depend on which scenario we consider.

2. Consider the case where $J \in \mathbf{J}^{\text{II}} \setminus \mathbf{J}^{\text{III}}$ and $J \succ B$ in PT_{HI}. In scenario s, since job $J \notin \mathbf{J}^{\text{III}}$ it must terminate at or before τ', hence $J \in \mathbf{J}^{BF}$. In scenario s', since $J \in \mathbf{J}^{\text{II}}$, its termination order *w.r.t.* B is determined by PT_{HI}, and because it has a higher priority it terminates earlier. Therefore, we have $J \in \mathbf{J}^{BF}$ in both scenarios.

3. Consider the case where $J \in \mathbf{J}^{\text{II}} \setminus \mathbf{J}^{\text{III}}$ and $J \prec B$ in PT_{HI}. Since PT_{HI} applies for whole scenario s, the above priority relation implies $J \in \mathbf{J}^{ZR}(B, s)$, whereas in s' we can always classify J as: $J \in \mathbf{J}^{OT}(B, s')$.

4. For J in the subset $\mathbf{J}^{\text{I}}$, let us show that in s it can be classified either as $J \in \mathbf{J}^{BF}(B, s)$ or as $J \in \mathbf{J}^{ZR}(B, s)$. Suppose that in s it terminates at or before τ'. Then, obviously, we can classify it as $\mathbf{J}^{BF}$, because B terminates after τ'. If, on the contrary, J terminates after τ' then both job J and job B are ready at time τ' and therefore the same argument as in Case 1 applies to show that J can go either to $\mathbf{J}^{BF}$ or $\mathbf{J}^{ZR}$. As for scenario s', since $J \in \mathbf{J}^{\text{I}}$, we see that it terminates at or before τ', and hence J can be obviously classified as $J \in \mathbf{J}^{BF}(B, s')$.

5. Finally, as for the jobs in $\mathbf{J}^{\text{IV}}$, they do not exist in s, so they can be classified as $\mathbf{J}^{ZR}$ in s and as $\mathbf{J}^{OT}$ in s'.

From the above reasoning we see that in s all jobs $J \neq B$ are classified as either $\mathbf{J}^{BF}$ and $\mathbf{J}^{ZR}$, whereas when going from s to s' no jobs have to move from subset $\mathbf{J}^{BF}$ to another subset, which shows that Lemma 4 indeed applies for $B \in \mathbf{J}^{\text{III}}$. Now it remains to show that if we add jobs from the set $\mathbf{J}^{\text{V}}$ of jobs that arrive after τ' then this result is preserved and that we can also prove the later termination for the members of this subset.

A simplifying circumstance for the remainder of the proof is that *after* time τ' both scenarios are controlled by a priority table, PT_{HI}, therefore *any* job J in set $\mathbf{J}^{\text{V}}$ automatically satisfies a significant part of the requirements of Lemma 4 – it always can be classified as either $\mathbf{J}^{BF}$ or $\mathbf{J}^{ZR}$.

To prove that the already established result is preserved with $\mathbf{J}^{\text{V}}$ added to the picture, we have to, firstly, show that the presence of these jobs – after time τ' – does not impact the arguments in the five cases considered above. The only argument that concerns the events after time τ' is that the jobs that are ready time τ' in relative order determined by PT_{HI} table. Obviously, addition of new jobs after time τ' does not change this fact. Secondly, to the five cases given above we have to add the sixth one – namely, when jobs J are in $\mathbf{J}^{\text{V}}$. The only interesting sub-case in this case is when $J \succ B$ in PT_{HI}, because otherwise J can be obviously classified in $\mathbf{J}^{BF}$ in both scenarios. The five cases analyzed so far show that when considering the total remaining workload of a job J and all higher priority jobs at time instance τ' we see that this workload can be only equal or higher in s', otherwise J would terminate earlier in s' even without adding the jobs $\mathbf{J}^{\text{V}}$. Now, when we add to s and to s' the same subset of jobs from $\mathbf{J}^{\text{V}}$ that have higher priority than J then we conclude that the total workload of J and all its higher-priority jobs will increase by the same amount in both scenarios at any time instance after τ', and it is only this workload that determines the termination time of B. Therefore, we preserve the property that jobs B from $\mathbf{J}^{\text{III}}$ cannot terminate earlier in s'.

Now it remains to prove the same for the jobs in subset $\mathbf{J}^{\text{V}}$. For a job $B \in \mathbf{J}^{\text{V}}$ we are only interested in jobs $J \succ B$ in PT_{HI}, because the other jobs do not interfere with B in either of the scenarios. For the jobs J in sets other than set $\mathbf{J}^{\text{V}}$ holds that they arrive earlier than any B in $\mathbf{J}^{\text{V}}$. Adding to it our current assumption that $J \succ B$ we conclude that the jobs J should terminate earlier than B, thus being in $\mathbf{J}^{BF}$ in both scenarios.

Finally, for a B in $\mathbf{J}^{\text{V}}$ it remains to consider $J \in \mathbf{J}^{\text{V}}$, and, again, we are only interested by $J \succ B$. In the previous case we have just proved that if we removed after time τ' all jobs $\mathbf{J}^{\text{V}}$ and all other jobs except those that have higher priority than B then $\mathbf{J}^{\text{V}}$ would terminate no earlier in s' than in s. Therefore, when adding higher-priority jobs from $\mathbf{J}^{\text{V}}$ this property will be preserved. This can be proved by the same argument about the total workload of job B and its higher priority jobs as we gave for in the case when $J \in \mathbf{J}^{\text{V}}$ and $B \in \mathbf{J}^{\text{III}}$.

The above reasoning leads to a conclusion that B cannot terminate earlier in s' when Restriction (i) is applied. The remaining case is the Restriction (ii). However, it is almost trivial. In this case the same priority table is applied in both scenarios and the only difference between them is that in s' also the jobs from $\mathbf{J}^{\text{IV}}$ are executed, which can lead only to delaying the jobs in s'. □

6 Complication for the Multiprocessor Case

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

Example 6. *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 5 shows execution in two scenarios: s and s' for the priority tables specified in the figure, whereby Restriction (ii) $PT_{LO} \sim PT_{HI}$ is not satisfied and hence sustainability is not guaranteed. Scenario s' differs from scenario s only by a larger execution time of J_1.

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 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 [1].

7 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.

Independently from whether or not the general problem is in class NP, its restrictions may be themselves in this class or outside. In our optimisation problem the correctness testing requires the tested solutions to be either sustainable or to be transformed into a particular type of solution which is sustainable. If one insists on "fair" (*i.e.,* not transformed) solutions then the problem may become harder than the general problem, where any type of solutions are acceptable.

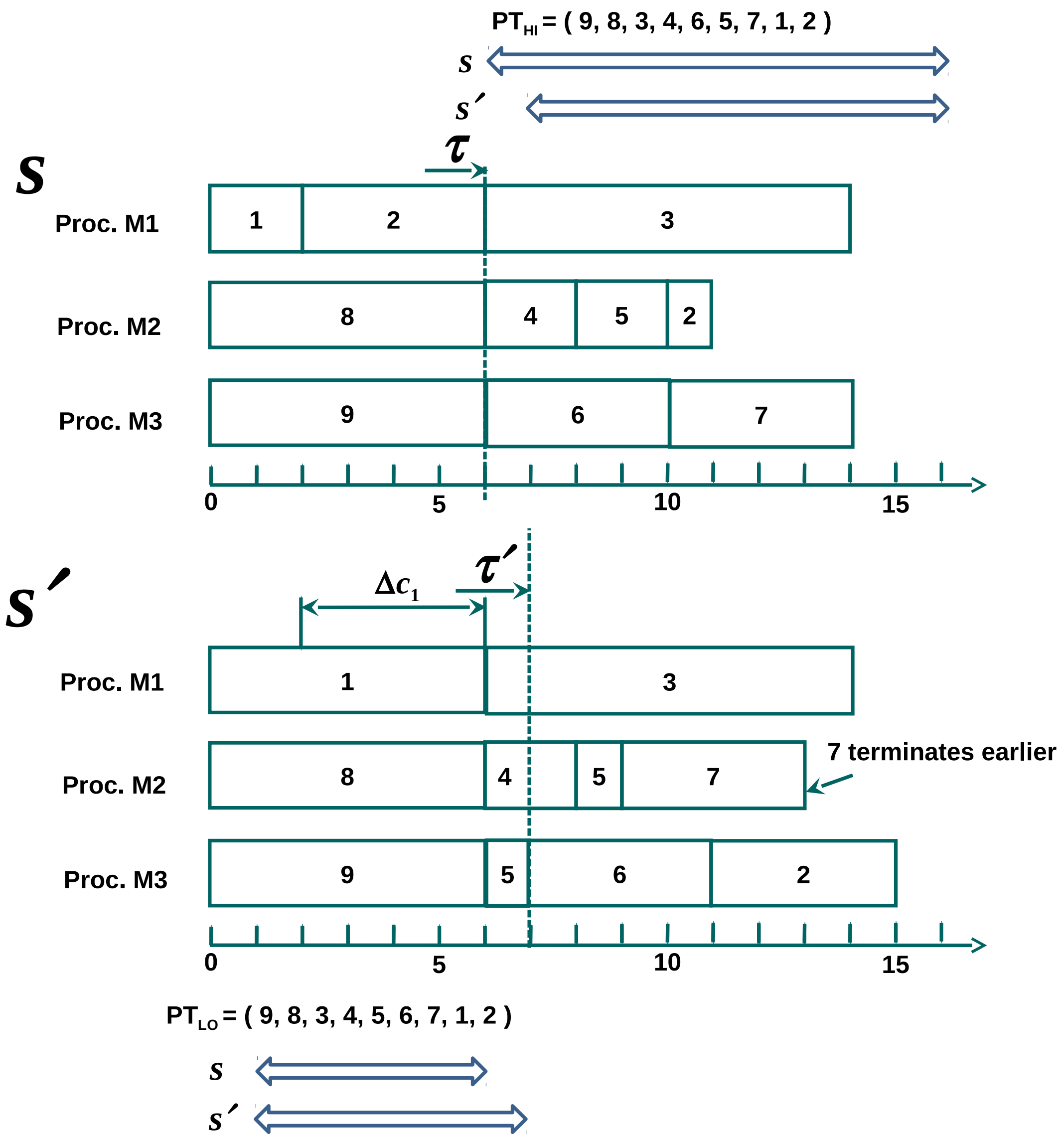

Figure 5: FPM non-sustainability demonstration on multiprocessor case, using Example 6. Gantt charts of two scenarios: s and s'. Mode switch times are τ and τ', resp. Scenario s is defined by $(c_1 = 2, c_2 = 5, c_3 = 8, c_4 = 2, c_5 = 2, c_6 = 4, c_7 = 4, c_8 = c_9 = 6)$. Scenario s' differs from s 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.

In that context, we have 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. Luckily, this does not remove practical application of the respective optimisation algorithms, such as MCPI [6]. These policies have been shown, [5], to successfully generate efficient time triggered tables. An alternative for multiprocessor scheduling is to avoid switching the priority table but just to drop the less critical jobs. We give the sustainability proof for single-processor case.

References

[1] 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, 2.1, 3.2, 3.3, 3.3, 4, 5, 6

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

[3] Sanjoy K. Baruah and Alan Burns. Sustainable scheduling analysis. In *Real-Time Systems Symposium (RTSS 2006)*, pages 159–168, 2006. 3.1

[4] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 3–12, 2011. 3.3, 4

[5] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler on single- and multi-processor platforms (revised version). Technical Report TR-2015-8, Verimag Research Report, 2015. 3.3, 7

[6] Dario Socci. *Scheduling of Certifiable Mixed-Criticality Systems*. PhD thesis, VERIMAG Research Center, Université Grenoble Alpes, 2016. 3.3, 5, 7

[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. 3.3

[8] 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. 5

[9] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. Technical Report TR-2014-11, Verimag Research Report, 2014. 5

[10] 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. 5